Whiteoak: Introducing Structural Typing into Java

Joseph (Yossi) Gil Itay Maman

Department of Computer Science
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel
yogi, imaman @cs.technion.ac.il

Abstract

This paper presents WHITEOAK: a JAVA extension that introduces structural type equivalence and subtyping into the language. We argue that structural subtyping addresses common software design problems, and promotes the development of loosely coupled modules without compromising type safety.

We discuss language design issues, including subtyping in face of self-referencing structural types, compile-time operators for computing the new types from existing ones, and the semantics of constructors and non-abstract methods in structural types. We describe implementation techniques, including the compile-time and run-time challenges that we faced (in particular, preserving the identity of objects). Measurement indicate that the performance of our implementation of structural dispatching is comparable to that of the JVM's standard invocation mechanisms.

Categories and Subject Descriptors D.3.3 [*Software*]: Programming Languages

General Terms Languages, Performance

Keywords Java, Structural Subtyping, Abstraction

1. Introduction

A restaurant accounting library A purchased from an American vendor expects parameters that conform to **`interface`** `Check`, but, the British maker of a large software module B in charge of serving orders from the kitchen to customers, chose to produce objects which are instances of **`class`** `Bill`. Now, a `Bill` offers essentially the same set of services demanded by `Check`. How can components A and B be coerced to work together without modifying any one of them?

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

In programming languages which obey *nominal typing* (sometimes called *nominative typing*) rules, such retrofitting is not easy. Nominal typing dictates that two types are equivalent only if they have the same name. Accordingly, types `Check` and `Bill` are nominally unrelated; one must use techniques such as those offered by the ADAPTER design pattern [14] to make the necessary plumbing code.

How is retrofitting done in languages such as ML and HASKELL where *structural typing* is the rule? Recall that structural typing means that two types are the same if they have the same structure. Also, *structural subtyping* follows from structure.[1] Thus, in these languages compatibility can be achieved by the observation that `Bill` is a subtype of `Check` (or vice versa), or even by using the minimal supertype of the two types. The compatibility is therefore due to the overlap between the set of members of two types.

1.1 The Case for a Dual Nominal-Structural Typing

The failure of nominal typing system in situations demanding retroactive type abstraction is discussed in depth in the literature [6, 7, 11, 20, 22] making the case for using structural typing in mainstream languages. We argue further that the importance of retrofitting even increases with the advent of "*compile once run everywhere*" languages such as JAVA, RUBY and PHP: The fact that hardware speed and memory constraints are not as stifling as they used to be, complemented by the huge body of open source modules accumulated in the world wide web, has led to the emergence of a new kind of hybrid programs [13] that contain numerous modules, written by many different programmers. Crucial to such architectures is the concept of *interoperability*—the ability to integrate modules written by independent authors. Dynamic type checking of RUBY and PHP may make interoperability in these easier—it is more difficult to achieve this goal while preserving the safety, clarity, and other benefits of static typing. To make interoperability possible in a statically typed environment, independent modules must agree on the *type* of exchanged data; and structural subtyping contributes to the ability of reaching such an agreement.

[1] Note that this implication is reversed in nominal type systems: the structure of a type follows (also) from any explicit subtype declarations.

Another famous crucial issue in which nominal type systems fail is in interaction with external data, be it persistent (e.g., residing in relational or XML databases) or originating from a distributed computing environment (e.g., a query to a web service). Such data is described by its structure, and attaching internal program names to it is difficult (or even impossible as it is the case in PASCAL). This is precisely the reason that languages designed for data interchange such as ASN.1 [24] are structurally typed. Difficulties in implementing conflicting types and class hierarchies is also mentioned in the literature [7] as a failure point of nominal typing systems.

On the other hand, it is also acknowledged that structural typing has its limitations, most notably, "accidental conformance" (discussed, together with some prospective solutions by Laüfer, Baumgartner and Russo [20]), and the difficulty of defining recursive types. Other pros of nominal type systems include (to use the words of Malayeri and Aldrich [22]): the fact that these systems support the *explicit expression and enforcement of design intent, simplify declaration of recursive types, and make it possible to produce more comprehensible error messages*, etc.

These, and the fact that dispatching is less efficient in structural type systems may explain the fact that nominal typing is the dominating scheme in (statically typed) mainstream languages designed for large projects, including JAVA, C++ and EIFFEL, while at the same time, the community seeks ways of combining these two typing paradigms so that their respective benefits can be used in tandem. The Unity language [22] is a recent example of a language design which combines the two concepts. The contribution of Unity is in formally showing a language model with a sound and complete type system which combines the two paradigms. However, Unity leaves open issues such as separate compilation, dynamic loading of classes and multiple inheritance of nominal types (a-la JAVA's/C$^{\#}$'s interfaces), and a host of problems arising in actual language implementation. The recent release of SCALA [25] introduced structural typing into the language through the notion of *Refinement of Compound Types*. It appears though that the performance of this feature is still poor.

In their paper describing the Continuum project [17], Harrison Lievens and Walsh also arrive at the conclusion that nominal subtyping makes mainstream object-oriented languages inflexible. They even go further and claim that the standard dispatching mechanism, where a method is looked-up in the context of a single receiver object, creates an involuntarily coupling between the client and the structure of the service provider.

Prior work on integrating structural typing with concrete, non-research languages, includes *signatures*, a C++ language extension which combines structural typing with the existing nominative features of the languages [6, 7], *safe structural types for* JAVA [20] and the work of Büchi and Weck on *Compound Types* [11].

More generally, many aspects of the question of merging the two paradigms can be thought of as the problem of interacting such nominative languages with external, structurally typed data. Work on this dates back to the work of Schmidt on Pascal-R [27] going through the work of Andrews and Harris [3] in the context of C. (See also surveys in [4, 5]). Another related work is that of Jorgensen on Lasagne/J [19] who solved the retrofitting problem by means of language mechanisms that allow for automatic wrapping, but stays short of structural typing. Also worthy of mention here is the proposal [2] for introduction of closures to JAVA which can be viewed as indicative of the desire to introduce structural typing into the language.

1.2 WHITEOAK and JAVA

This paper describes the design and implementation of WHITEOAK, a system that introduces structural type equivalence and structural subtyping into JAVA. This addition relies on a new keyword, **`struct`**, used to define structural types whose subtyping relation is determined solely by structure.

In a sense, **`struct`** types are similar to JAVA's interfaces, and in particular support "multiple inheritance", except that subtyping among **`struct`** types is, as expected, structural rather than nominal. Thus, the type `XYZ` defined by

```
struct XYZ { int x,y,z; }
```

is a subtype of `XY` defined by

```
struct XY { int x,y; }
```

and of the unnamed structural type

```
struct { int y,x; }
```

Names of structural type are optional, but once defined, they can be used as a shorthand for the full type declaration: The function definition

```
int innerProduct(XY a, XY b) {
  return a.x * b.x + a.y * b.y;
}
```

would be no different if it used the more verbose signature

```
int innerProduct(
  struct {int x,y;} x,
  struct {int y,x;} y
)
```

Any nominal (**`class`** or **`interface`**) type that conforms to the protocol of a structural type can be upcast into that structural type: The method `innerProduct` can thus be applied to any conventional type which has public, non-final integer fields named `x` and `y`. A field access in the method, e.g., `a.x`, is mapped to the field declared by the referenced object runtime type. Thus, fields declared by a structural type follow a late binding semantics, just as methods.

[2] http://www.javac.info/closures-v04.html

We saw that WHITEOAK provides for polymorphic abstraction and retrofitting over fields. The following WHITEOAK function demonstrates retrofitting over methods:

```
struct Source { int read(); }
void exhaust(Source s) {
  while (s.read() >= 0) ;
}
```

Function `exhaust` is applicable to objects of both class `Reader` (the superclass of classes capable of digesting Unicode input) and `InputStream` (the superclass of classes for processing byte-oriented input), despite the fact that the two classes are unrelated. Further, the function call `s.read()` dispatches correctly in all cases, even though the dispatch target is not necessarily stored in the same location in the virtual functions table.

We saw that unlike interfaces, **`struct`** types allow, just like abstract classes, the declaration of fields. Another similarity to abstract classes is that **`struct`** types may define concrete method implementations: abstract methods and fields declared in a **`struct`** define the *requirements* that any conforming type must provide, where the concrete methods define *default behavior* which can be specialized by the conforming type.

Revisiting our opening dilemma, we can say that if `Check` was declared as a **`struct`** type rather than an **`interface`** type, then a `Check` variable can receive its value from `Bill` objects. But, even if this was not the case, the bridging code is simplified by using the most specific structural type to which both `Check` and `Bill` conform.

Unlike classes, **`struct`** types have no constructors, although they may pose a constraint on the protocol of the class constructors—a feature which naturally admits "virtual constructors" into the language. Also, unlike classes, fields cannot be initialized as part of their definition.

There are no direct means for defining **`struct`** literals, but anonymous classes provide a substitute. One can therefore invoke function `innerProduct` with two ad-hoc types and their values

```
int xmas = innerProduct(
  new Object() {
    int x = 3, y = 5; int a = 3;
  },
  new Object() {
    int x = 5, y = 2;
  }
);
```

Conversely, we argue that user control over anonymous classes is enhanced with structural types.

Two more features should be mentioned at this stage: (i) support for dynamic run-time subtyping tests and downcasts even against structural types; and (ii) operators for the *intersection*, commutative- and non-commutative- (i.e., overriding) *union* of structural types.

The implementation of WHITEOAK comprises two components:

1. a modified JAVA compiler (based on Sun's JAVA 5 compiler). This compiler generates standard bytecode, and conforming `.class` files, and
2. a small runtime library of functions realizing the dynamic dispatching semantics of WHITEOAK.

Unlike many research languages, WHITEOAK's design had to deal with the issue of integrating with and supporting existing language features (including genericity, reflection, annotations, dynamic loading of classes, etc.), as well as preserving the semantics of existing, previously compiled code.

A primary challenge that WHITEOAK faced was that of an efficient realization of the structural typing addition on top of the standard Java Virtual Machine [21] (JVM), which is nothing else than a *nominally* (and strongly typed) machine model. In this respect, our work was more difficult than that of e.g., Baumgartner and Russo classical implementation of signatures in C++, in which both translation to untyped assembly and the use of advertised loopholes in the type system of C++ could be used to support structural typing.

1.3 Contribution

The ideas presented here are realized in the WHITEOAK compiler which is available for download from the following address: `http://whiteoak.sourceforge.net`. In summary, the main contributions of this work are:

1. A statically typed object oriented language that enables the attachment of new, over-rideable, behavior to existing objects and the composition of these into *Virtual Objects*. This goes beyond compositional mechanisms (including inheritance, mixins [2, 8] or traits [26]) which operate solely on classes.
2. A demonstration that a rather complete structural typing system can be added to a non-toy, industrial strength language, without ignoring issues such as preservation of object identity, static and final members, visibility, anonymous types, genericity and load-time verification.
3. Implementation that supports late binding for constructors (*virtual constructors* [12]), and fields despite the inherent limitation imposed by the run-time system (the JVM).
4. Finally, we show how mechanisms such as mixins, traits or delegation can be emulated in WHITEOAK in the library level rather than the language level.

Outline. The remainder of this paper is organized as follows: Section 2 overviews the features of the language, its grammar specification and compares it with related work. Section 3 describes WHITEOAK's implementation strategy in terms of compiler modification and required run-time support. This section then proceeds to presenting performance

data and evaluation. We conclude in Section 4 where we discuss the implications of structural typing on the style of programming, reflect on this work's position within the scheme of current research effort and outline directions for future work.

2. The WHITEOAK Language

Backward compatibility, efficiency, simplicity, uniformity and expressive power where principal guidelines in the design of WHITEOAK. This section describes the main language design alternatives encountered, explains the decisions we took in light of these causes, and elaborates on the challenges that these decisions entailed. The section concludes with a detailed comparison of WHITEOAK's realization of structural types within a nominative type system with previous efforts of this sort.

2.1 Definition of Structural Types

A structural type can be used anywhere a nominal type can be used, including variable-, parameter-, return type-, and field- definition just as in constraints on type parameters to generics. Structural types cannot be used in the throw-list of exceptions, since all exceptions must inherit from the nominal library type `Throwable`. We also stayed short of allowing generic structural types.

A structural type may be defined in place, or refer to a named structural type definition. Such named definitions may be made in any location a non-anonymous JAVA class can be defined, including the outer package scope, in a class, or in a function.

Which member kinds are allowed in structural types? Figure 2.1 demonstrates WHITEOAK's answer.

```
1 struct ErrorItem {
2   // bodiless method:
3   int severity();
4   // a field:
5   String description;
6   // a requirement on a field (read−only access)
7   final int lineNumber;
8   // a method with default implementation
9   String where() {
10    return lineNumber + ": "
11       + description;
12  }
13  // constraints on constructors:
14  constructor(int l);
15  constructor(String d, int l);
16 }
```

Figure 2.1: Structural type `ErrorItem` demonstrating the variety of member kinds allowed in WHITEOAK

In the figure we see that **`struct`** types may have bodiless functions, data members which may even be **`final`**, functions with body, and constructor specification. WHITEOAK does not allow **`static`** members[3], initialized data members, or constructors with a body.

Bodiless (abstract) methods, representing a constraint on the actual type, are obviously essential.

Data members were added for uniformity and in support of interaction with external databases; The mechanisms for supporting these are no different than those required for bodiless functions.

Constructor constraints were added, again for uniformity, but also in support of virtual constructors and more expressive generics. For example, with the above definition of `ErrorItem`, one may write

```
ErrorItem bump(ErrorItem e, int diff) {
  return e.constructor(
    e.description,
    e.lineNumber + diff
  );
}
```

Again, constructor specifications are implemented as bodiless functions.

Virtual Objects allow client code to obtain an object reference that offers a different set of methods than the actual (referenced) object. This is achieved by assigning an object into a variable of structural type `S`, where `S` defines *non-abstract methods*. Each such method provides a default implementation that will be invoked if the actual object does not provide its own implementation for that method. This is demonstrated by Figure 2.2.

```
1 struct LineReader {
2   int read() throws Exception;
3   String readLine() throws Exception {
4     String s = "";
5     for (int c = read();
6       c >= 0 && c != '\n';
7         c = read())
8           s += (char) c;
9     return s;
10  }
11 }

13 void f(Reader r) throws Exception {
14   LineReader lr = r;
15   System.out.println(lr.readLine());
16 }
```

Figure 2.2: A structural type with a default function implementation.

In the figure we see the structural type `LineReader`. This type requires an **`int`** `read()` method and provides

[3] Still, we shall see that a field or method declaration in **`struct`** can be realized by a **`static`** field or method of a nominal type.

a `String readLine()` service based on this method. Assigning an instance of any class with an appropriate `read` function to a variable whose type is `LineReader` will effectively attach the implementation of this function to the object, as demonstrated in function `f` in the figure: The assignment in Line 14 in this function is legal, since the class `Reader` declares the method **`int`** `read()`.[4]

If function `f` is invoked with an object whose dynamic type `FileReader` (a subclass of `Reader` that does not offer a `readLine()` method) then the `readLine()` call in Line 15 is dynamically bound to the default implementation found in `LineReader`. In contrast, if the function is invoked with an instance of a class such as `BufferedReader`, which happens to implement this function, then Line 15 is bound to the object's own implementation.

Summarizing this program we note that the virtual object reference, `lr`, provides its own static protocol and its own run-time behavior which differ from the ones provided by the referenced object, `r`. This means that in WHITEOAK protocol and behavior can be associated with *references* (i.e.: variables) and not just with *objects*. This provides the means for achieving better localization of concerns: there is no need to define the full behavior of the object at the class definition point. If a certain concern is needed only in a certain part of the program, we can package the relevant code as a virtual object and use it only when needed.

One reservation applies: a structural type in which a certain function is unimplemented cannot be assigned from a structural type which offers a default implementation for this function. Hence, the following fails to compile

```
Reader r = ...;
LineReader lr = r;
struct { String readLine(); } x;
x = lr;
```

In the assignment `x = lr` we obtain a new reference to the actual object, `r`, which is not guaranteed to provide an implementation for the `readLine()` method (recall that implementation of `readLine()` is associated with the reference `lr` and not with the object `r`). Therefore, when assigning from a virtual object, non-abstract methods are treated as if they were abstract. Further applications of virtual objects are discussed later in Section 2.2

Static members. WHITEOAK does not allow structural types to define **`static`** members. In particular, we cannot make the demand that a function or a field is implemented as **`static`**. Still, a **`static`** member in a nominal type is allowed to realize a **`struct`** member specification, thus allowing uniform treatment of static and non-static features.

For example, an instance of class `A` defined by

```
class A {
   public static void f() {}
   public static int d;
}
```

may be assigned to a **`struct`** requiring a **`void`** function `f` and an **`int`** data member `d`:

```
struct {
   void f();
   int d;
} a = new A();
```

and `a.f()` will be bound dynamically to `A.f` while the member reference `a.d` will be dynamically delegated to the **`static`** data member `A.d`.

WHITEOAK also supports *recursive structural types* as demonstrated by Figure 2.3,

```
struct List {
  int head();
  List tail();
}

struct MutableList {
  int head();
  MutableList tail();
  void tail(MutableList t);
}

struct ReversableMutableList {
  ReversableMutableList reverse();
  int head();
  ReversableMutableList tail();
  void tail(ReversableMutableList t);
}
```

Figure 2.3: Recursive structural types. `MutableList` and `ReversableMutableList` are subtypes of `List`. `ReversableMutableList` is not a subtype of `MutableList`.

The `List` type in the figure is (self) recursive in a co-variant position: the return type of `List.tail()` is `List` itself. This allows `MutableList` to be a structural subtype of `List`. Examining `MutableList` we see that it is (self) recursive in a *contra-variant* position: the second `tail` method,

```
MutableList.tail(MutableList),
```

defines a parameter of type `MutableList`. Recursion in a contra-variant position prohibits subtyping, so the type `ReversableMutableList` is not a subtype of `MutableList`. The formal criteria for subtyping of recursive types are realized in the subtyping algorithm in Algorithm 1.

[4] The fact that `Reader.read()` is an abstract method is not a problem. The JAVA language semantics forbidding instantiation of abstract classes guarantees that the dynamic type of variable `r` will offer a concrete implementation for all its methods.

2.2 Composition

This part of the paper examines the composition techniques available in WHITEOAK. As we shall see shortly, the combination of these composition techniques along with the ability to define virtual objects (that is: structural types with non-abstract methods), allows the WHITEOAK programmer to attach units of behavior to existing objects. This allows greater flexibility than traditional, statically-typed, code reuse mechanisms—inheritance, mixins and traits—which manipulate classes but not objects.

WHITEOAK offers three type composition operators: If T_1 and T_2 are structural types, then $T_1 * T_2$ is the type obtained by the *intersection* of the set of members defined in T_1 and T_2, T_1+T_2 is the commutative *union* of these sets, and T_1 T_2 (type T_1 concatenated with type T_2) is the *overriding union* of these sets. If one of T_1 and T_2 is nominal, then it is cast to the corresponding structural type prior to the union.

A function in T_1 *conflicts* with a function in T_2 if both have the same name and the same arguments, but a different return type.[5] Similarly, two data member definitions conflict if they have the same name, but different type, or **`final`** specification. Such conflicts are reported as errors.

Otherwise, (i) pairs of data members of the same name, type and **`final`** specification, (ii) pairs of constructors with the same signature, and (iii) pairs of functions with the same signature, return type, and name, are synonymous, and included only once in the union. Type $T_1 * T_2$ is obtained by taking an element of each synonymous pair.

The following reservation applies: a synonymous pair of functions is conflicting if both have an implementation, except for non-commutative union, in which the implementation in T_2 prevails, i.e., much like in overriding inheritance.[6]

The type union operator can be used to enrich an object with additional behavior but can still allow the object to provide its own, specialized, implementation for this behavior, e.g., in writing

```
struct Input {
  int read() throws Exception;
  int read(char[] a) throws Exception;
}
struct ImprovedInput
  = Input + LineReader;
```

we define a new structural type whose specification is the union of `LineReader` and `Input`.

Mixins. Mixin composition also allows classes to be composed from smaller building blocks. A mixin composition is non-commutative, i.e.: the order of the composition is important. This order imposes a natural overriding relation on the defined methods. In WHITEOAK we use the concatenation operator to generate a structural type by applying mixin-composition on the two operands. This is shown in Figure 2.4.

```
struct MCircle {
  abstract double radius();
  double diameter() {
    return 2*radius();
  }
  String name() { return "Circle"; }
}

struct MRed {
  Color color() { return Color.RED; }
  String name() { return "Red"; }
}

struct CircleRed = MCircle MRed;
CircleRed cr = new Object() {
    double radius() { return 3.0; }
  };

System.out.println(cr.name());
// Output is: "Red"
```

Figure 2.4: A mixin composition of the `Circle`, `Red` classes. The methods of the second operand, `Red` override those of the first one.

The composition `MCircle MRed` yields a type with just one abstract method, `radius()`. Therefore, any conforming nominal type needs to specify only this method. A call to the `diameter()` method on the `cr` variable will dispatch the (only) implementation from `MCircle`. A `cr.name()` call will dispatch implementation from `MRed`, since the methods of the second operand override the methods of the first operand.

Traits. Traits were proposed [26] as a mechanism for allowing better code reuse. A class definition can use one or more traits as building blocks that supply part (or even all) of its behavior. Just like WHITEOAK's structural types, traits provide behavior but they cannot carry any state.

Figure 2.5 shows a standard example for traits using the syntax offered by Hill, Quitslund and Black [23].

The figure compares the JAVA with traits code with the WHITEOAK equivalent. The `RedCircle` class is expressed in WHITEOAK by using the "+" operator, to compose the `TCircle` and `TRed` structural types. This composition yields a type with two implemented methods, `diameter()` and `color()`, and one abstract method: `radius()`.

We then assign an instance of an anonymous class, that implements all the abstract methods, into a variable of type

[5] In reality, also thrown exceptions are checked in a similar fashion, but discussion of declared exceptions is omitted from this manuscript.

[6] In our design, in the case that only one function in the pair has an implementation, this implementation prevails, even in the union. Other alternatives are legitimate, just as more elaborate designs, e.g., computing the least common ancestor of the return type.

```
abstract class TCircle {
  abstract double radius();
  double diameter() {
    return 2*radius();
  }
}

abstract class TRed {
  Color color() { return Color.RED; }
}

class RedCircle uses TCircle, TRed {
  double radius() { return 3.0; }
}
```

```
struct TCircle {
  double radius();
  double diameter() {
    return 2*radius();
  }
}

struct TRed {
  Color color() { return Color.RED; }
}

struct RedCircle = TCircle + TRed;

RedCircle rc = new Object() {
  double radius() { return 3.0; }
};
```

Figure 2.5: A red circle class. The first program shows JAVA with traits code. The second program shows WHITEOAK code.

`RedCircle`. The resulting variable can respond to any of those three methods.

2.3 Grammar

Figure 2.6 gives a grammatical specification of the four kinds of members allowed in structural types. The productions in the figure rely on nonterminals such as *Identifier* and *FormalParameterList* defined elsewhere in JAVA's grammar [16].

Examining the figure we see that structural types may not be generic (generic recursive structural types are known to be a difficult and elusive problem; see e.g., Hosoya et al. [18] for an explanation on circumstances in which subtyping in this setting might be undecidable, or lead to counter-intuitive results, or require tagging baggage.) Also note that with the exception of **`final`** for field declarations, no modifiers are allowed. All members of a structural type are implicitly **`public`**, and as discussed below, they can be realized by both **`static`** and non-**`static`** implementations.

Figure 2.7 defines how compound structural types are created, and how these types combine with the rest of JAVA.

The first production in the figure augments JAVA's grammar by stating that a structural type (nonterminal *StructType*) can be used *anywhere* a reference type can be used. This includes e.g., arguments to generics, bounds on such arguments, etc.

We then state that a *StructType* is specified either as a named structural type, defined by a *StructDeclaration* or as an *UnnamedType* which allows a direct use of a structural type expression. Such expressions are composed by applying the three structural type operators, (i) union, specified by operator + (lowest priority), (ii) intersection (operator $\star$) and (iii) concatenation, whose semantics is reminiscent of inheritance with overriding (highest priority) to combine *StructAtom*s, i.e., atomic structural types.

Named structural types are mostly a typing aid, and the named type are more than shorthand for the entire type declaration; they make it is easy to define recursive structural types. Also note that the production

StructDeclaration: **`struct`** *StructId = StructUnion ;*

states that names can be also given to structural type expressions.

Nonterminal *StructAtom* is in turn a *ReferenceType*. The semantics of this production should be clear if this *ReferenceType* happens to be a structural type. However, if the reference type is a nominal type, the derivation effectively computes the *structural equivalent* of the nominal type, defined as the set of all **`public`** method declarations and all **`public`** fields declarations. In computing this set, all modifiers except for **`final`** data member modifiers, method bodies, initialization expressions of data members, just as annotations are eliminated.

StructBody: { *StructMember*$_{opt}$ }
StructMember: MethodDeclaration ;
| *MethodDefinition*
| *ConstructorDeclaration ;*
| *FieldDeclaration ;*
MethodDeclaration: Type Identifier
(*FormalParameterList*$_{opt}$)
Throws$_{opt}$
MethodDefinition: MethodDeclaration MethodBody
ConstructorDeclaration: **`constructor`**
(*FormalParameterList*$_{opt}$)
Throws$_{opt}$
FieldDeclaration : **`final`**$_{opt}$ *Type Identifiers*
Identifiers: Identifier | *Identifiers* **`,`** *Identifier*

Figure 2.6: Grammar specification of the body of structural types.

ReferenceType: $\cdots \mid \cdots \mid$ *StructType*
StructType: StructId | *UnnamedType*
StructId: Identifier
StructDeclaration: **struct** *StructId StructBody* |
struct *StructId = StructUnion ;*
UnnamedType: **struct** *StructUnion*
StructUnion: StructIntersection |
StructIntersection + StructUnion
StructIntersection: StructConcatenation |
StructConcatenation $\star$ *StructIntersection*
StructConcatenation: StructTerm |
StructConcatenation StructTerm
StructTerm: (*StructUnion*) | *StructAtom*
StructAtom: ReferenceType

Figure 2.7: Grammar specification of the uses of structural types

2.4 Type Checking Algorithm

This section presents the algorithm used by the WHITEOAK compiler for determining whether one type is a subtype of another. This algorithm is invoked, for example, by the type checker module when it examines assignment statements. In particular, this algorithm is used by the type checker to verify that the type of the right-hand side value in an assignment is compatible with the type of the left-hand side variable.

The algorithm is based on the algorithm of Amadio and Cardelli [1]. Its actual realization in the WHITEOAK compiler is somewhat more complicated due to optimizations. Nonetheless, the semantics (i.e.: the typing rules) is exactly the same as the semantics presented here.

The presentation uses the following notations and symbols.

- $x = y$ indicates the trivial type equality relation, that is: x and y are the same type. We naturally extend this notation for denoting equality of sets of types and of sequences of types.
- $x \preceq y$ indicates WHITEOAK's type compatibility relation: x is a subtype of y.
- As a convention we use the variable r, "required", to denote a candidate supertype (or a member thereof); we use the variable f, "found", to denote a candidate subtype (or a member thereof).

The algorithm for computing $x \preceq y$ is presented below (Algorithm 1). Following it are auxiliary procedures that are called (either directly or indirectly) from Algorithm 1.

Note that these algorithms are inherently recursive: In order to determine a type compatibility question we need to determine member compatibility questions, which in turn rely on the results of further type compatibility questions. A cache (from a pair of types to a Boolean value) is used to break this otherwise infinite recursion.

Function: $IsAssignable(f, r)$

Input: f, r types (either structural or nominal)
Output: $True$ if $f \preceq r$, $False$ otherwise
1: **if** $f = r$ **then**
2: **return** $True$
3: **if** r is a nominal type **then**
4: **return** $IsNominallyAssignable(f, r)$
5: **if** f is a non-anonymous JAVA class **then**
6: **if** f's visibility is not public **then**
7: **return** $False$
8: **if** $cache[f, r]$ is initialized **then**
9: **return** $cache[f, r]$
10: $cache[f, r] \leftarrow True$
11: $cache[f, r] \leftarrow MemberSetTest(f, r)$
12: **return** $cache[f, r]$

Algorithm 1: Testing that f is compatible to r, that is: $f \preceq r$. Function $IsNominallyAssignable(f, r)$ realizes JAVA's standard nominal subtyping test. Function $MemberSetTest(f, r)$ checks that every member of r has a compatible member in f. $cache[x, y]$ is a cache slot that holds the result of $x \preceq y$. Initially all cache slots are uninitialized.

Examining Algorithm 1 we see that in step 3 the algorithm falls back to Java's standard (nominal) typing scheme if the candidate supertype, r, is a nominal type.

Steps 5—7 ensure proper visibility of the candidate subtype. Access to public classes is allowed. Access to other classes is allowed only if they are anonymous. This poses no security risks: During compilation, the only expressions that carry an anonymous-class type are the anonymous-class instantiation expressions. Therefore, f will be an anonymous class only if the developer deliberately assigns such an instantiation expression directly into a structurally-typed variable.

Step 8 breaks the recursion: if the result for the current type-compatibility question is already cached, we return that result.

The last part of the algorithm, realizes the actual computation of the structural compatibility question. In step 10 we cache a (tentative) $True$ answer for the $f \preceq r$ question.

In step 11 the auxiliary function $MemberSetTest()$ is called. This function will return $False$ if, and only if, there is a member of r that has no compatible member in f. Hence, we assign the result returned by this function to the appropriate cache slot and then return this result (step 12).

The algorithm that realizes function $MemberSetTest()$ is depicted in Algorithm 2.

The algorithm goes over every member of the r ("required") type, finds the compatible members from f ("found"),

Function: $MemberSetTest(f, r)$

Input: f, r types
Output: $True$ if every member of r has a compatible member in f.

```
1:  for all m_r member of r do
2:      s ← φ
3:      for all m_f member of f do
4:          if MemberPairTest(m_f, m_r) then
5:              s ← s ∪ {m_f}
6:      if s = φ then
7:          return False
8:      if s has overloading conflicts then
9:          return False
10: return True
```

Algorithm 2: Testing that every member of r has exactly one compatible member in f. Function $MemberPairTest(x, y)$ checks that x can replace y.

and stores these in the set s (step 5) If no such member was found (step 6) the algorithm returns a $False$ answer. On the other hand, if such members were found the algorithms makes sure (step 8) that there is one member in s which is more specialized than all other members in s. This check (which is essentially identical to Java's standard check for ambiguity due to overloading) ensures that every member of r is unambiguously mapped to a member of f.

The details of function $MemberPairTest()$ are presented in Algorithm 3.

Function: $MemberPairTest(f, r)$

Input: f, r members
Output: $True$ if f can replace r, $False$ otherwise

```
1:  if f's visibility is not public then
2:      return False
3:  if both f and r are constructors then
4:      return ConstructorCompatibilityTest(f, r)
5:  if the names of f and r are not identical then
6:      return False
7:  if both f and r are methods then
8:      return MethodCompatibilityTest(f, r)
9:  if both f and r are fields then
10:     return FieldCompatibilityTest(f, r)
11: return False
```

Algorithm 3: Testing the compatibility of two members. The three auxiliary functions: $ConstructorCompatibilityTest(\cdot, \cdot)$ (see Algorithm 4 below), $MethodCompatibilityTest(\cdot, \cdot)$ (Algorithm 5) and $FieldCompatibilityTest(\cdot, \cdot)$ (Algorithm 6) determine the compatibility of a pair of constructors, methods and fields (respectively).

The body of Algorithm 3 is straightforward: it delegates to one of three other functions, each handling a different set of inputs (a pair of constructors, a pair of methods, or a pair of fields). The function for deciding the compatibility of constructors, is presented in Algorithm 4.

Function: $ConstructorCompatibilityTest(f, r)$

Input: f, r constructors
Output: $True$ if f can replace r, $False$ otherwise

```
1:  t_f ← f's declaring type
2:  if t_f is a nominal type then
3:      if t_f is abstract then
4:          return False
5:      if t_f is a non-static inner class then
6:          return False
7:  if Params(f) ≠ Params(r) then
8:      return False
9:  return ThrowsClauseTest(f, r)
```

Algorithm 4: Testing the compatibility of two constructors. Function $Params()$ returns the sequence of the types of the formal parameters of its operand. Function $ThrowsClauseTest()$ realizes JAVA's standard (nominal) test for conformance of the **throws** clauses of its operands.

Examining Algorithm 4 we see that steps 2—6 make sure that the candidate constructor, f, can actually be invoked to produce a concrete object. In particular, constructors of non-**static** inner classes are rejected due to their extra, implicit, parameter.

We then require no-variance of the parameters (step 7) and covariance or no-variance of the throws clauses (9).

Note that the type compatibility decision issued by step 9 is essentially a nominal subtyping test: the candidate supertype is (per the JAVA language specification) a subclass of `Throwable` which is a nominal type. Therefore it is safe to use JAVA's standard test for conformance of **throws** clauses, $ThrowsClauseTest()$.

One may expect Algorithm 4 algorithm to require covariance of the objects generated by the two constructors at hand. A careful look at Algorithm 1 reveals that this check is redundant: the $ConstructorSignatureCompatibility$ function is (indirectly) called from step 11 in Algorithm 1. In the preceding step, Algorithm 1 cached a $True$ answer for the question of subtyping of the f and r types, which are essentially the declaring types of the f and r constructors in Algorithm 4. Thus, if Algorithm 4 will check for co-variance of the objects generated by the constructors it will get the inevitable (tentative) $True$ answer from the cache.

The algorithm for determining compatibility of methods is presented in Algorithm 5.

Looking at Algorithm 5 we see that compatibility of methods is established if we have: no-variance or co-variance of the return type (step 1); no-variance of the parameters

Function: $MethodCompatibilityTest(f, r)$

Input: f, r methods
Output: $True$ if f can replace r, $False$ otherwise
1: **if** $\neg(Type(f) \preceq Type(r))$ **then**
2: **return** $False$
3: **if** $Params(f) \neq Params(r)$ **then**
4: **return** $False$
5: **return** $ThrowsClauseTest(f, r)$

Algorithm 5: Testing the compatibility of two methods. Function $Type()$ returns the return type of its operand. Function $Params()$ returns the sequence of the types of the formal parameters of its operand. Function $ThrowsClauseTest()$ realizes JAVA's standard (nominal) test for conformance of the **`throws`** clauses of its operands.

(step 3); no-variance or co-variance of the exceptions declared in the throws clauses (step 5).

Finally, the compatibility of a pair of fields is determined by Algorithm 6.

Function: $FieldCompatibilityTest(f, r)$

Input: f, r fields
Output: $True$ if f can replace r, $False$ otherwise
1: **if** r is a read-only field **then**
2: **return** $Type(f) \preceq Type(r)$
3: **if** f is a read-only field **then**
4: **return** $False$
5: **return** $Type(f) = Type(r)$

Algorithm 6: Testing the compatibility of two fields. Function $Type()$ returns the type of its operand.

Examining Algorithm 6 we see that if the required field, r, is a read-only field, it can be matched with either a read-only or a mutable field, with a possibly co-variant type (step 2). On the other hand, if the required field is mutable, the matched field must be mutable (step 3), and of the exact same type (step 5).

Having presented WHITEOAK's type compatibility algorithm we can discuss a few implications. The first point to note is that of constructor calls on a receiver typed with a type parameter. Consider a type `U` defined by

```
struct U {
    U constructor();
    void m();
}
```

and a nominal type N conforming to `U`. Then, N must have a no-arguments constructor creating an N object. Using such a value in generics [9] seems to be legal:

```
static<T extends U> void f(T t) {
  t = t.constructor(); // Returned value is T?
  t.m(); }
```

One may think that the semantics of a **`constructor`**`()` call in WHITEOAK is that of a *self type* [10]: the returned object has the same dynamic type as the receiver, thus ensuring that the assignment

```
t = t.constructor()
```

is well typed. This assumption is in some sense similar to the special handling of the `getClass()` calls in the standard JAVA compiler [16, Sec. 4.3.2].

By examining the $ConstructorCompatibilityTest()$ function one can see that if the function returns a $True$ answer we can only assert that the f constructor will return an object whose type is at least as specific as f's declaring type, which in our case is the static type of the receiver.

Due to type erasure, the static type of the receiver is `U` (the erased type of `T`). Thus, one can only assume that returned object will be as specific as `U`, which, alas, is not a specific enough type to be assigned to the variable `t`.

Another subtle issue is related to the recursive nature of the algorithms presented here. These algorithms are recursive in the sense that in order to check that one type is a subtype of another, one needs to check subtyping relationship of individual methods. Method subtyping is defined (as usual in JAVA) by the demand that the return type changes co-variantly, and the argument types are the same. The recursive call may therefore yield more subtyping problems. We deviate from the usual implementation of [1] in that if, in these generated problems, the candidate supertype is nominal, we apply JAVA's nominal type testing algorithm. Also, the subtyping test invariably fails if the candidate subtype is structural and the candidate supertype is nominal. Recursion proceeds only if the candidate supertype is structural.

This ensures that a method that declares a parameter of a nominal type can safely assume that the dynamic type of this parameter will be a nominal subtype. Otherwise, every invocation of a method, on a parameter, had to be treated as a structural invocation. Even more seriously, the method's code may use reflection in ways that will break if the dynamic type is not a nominal subtype (e.g.: a parameter of type `Serializable` is sent to a serializing data stream).

2.5 Comparison with Related Work

Tab. 1 shows a feature based comparison of WHITEOAK with some of the main work on introducing structural typing into nominative languages, or for producing synergetic language, including C++ Signatures [7], Brew [20], Compound [11], Unity [22] and Scala.

The first section in the table compares the admissible member kinds. We see that WHITEOAK's repertoire of member kinds is fairly complete. Still, unlike WHITEOAK and all the other compared languages, only C++ signatures support field definition, that is, fields together with a default initialization expressions. But since the C++ signatures' semantics is restricted, the problem of finding appropriate semantics to default initialization expressions remains open. We know of

		Signatures [7]	Brew [20]	Compound [11]	Unity [22]	Scala	Whiteoak
Kinds of Members	Method declaration	+	+		+	+	+
	Method definition	+			+		+
	Mutable Field declaration				+	+	+
	Readonly Field declaration	+			+	+	+
	Field definition	+					
	Constructor declaration						+
	Constructor definition						
Expressiveness	Parametrization	+				+	
	Bounds on type parameters					+	+
	Unnamed types	+			+	+	+
	Type operators			+		+	+
	Recursive types	+	+		+		+
Impl.	Invariance of identity	+		+	+	+	+
	Separate compilation	+	+	+		+	+
	Performance data	+					+
Formalization				+	+		

Table 1. A comparison of recent work on introducing structural typing into nominally typed languages.

no support in current work of constructor bodies in structural types.

The next section compares integration with other linguistic mechanisms and composability of structural types through recursion or type operators. As expected, most languages chose to provide support for anonymous types on top of structural types (unlike *instances* of anonymous classes, *anonymous types* are practically useless in a purely-nominal setting). Note that Scala's parametrization of structural types in restricted in the sense that type parameter can only be used in co-variant positions (e.g.: return types).

The third section is concerned with implementation. Other than WHITEOAK only C++ Signatures provide performance data. In Scala, although no such data is available, it is mentioned that it is slow, probably due to heavy reliance on reflection information.

Turning to the preservation of object identity we see that C++ signatures implement this (by enhancing the compiler existing mechanism for comparison of **`this`**-adjusted references), while Brew, a previous addition of structural types to JAVA fails to preserve object identity.

Compound also preserves object identity, by relying on a fundamental property of this JAVA language extension, by which structural types can only be defined as the union of existing nominal JAVA interfaces. Every structural reference in Compound is represented as multiple variables, one for each of what the authors call "constituent type".

Finally, as the last table row indicates, only Compound and Unity enjoy a formalized basis.

3. Implementation and Performance

This section describes the compilation and run-time techniques that we used in order to augment JAVA with WHITEOAK's additional constructs while preserving these fundamental principles:

1. WHITEOAK classes should run on any standard, JAVA 5 compliant JVM; hence the run-time system must be implemented as a JAVA library.
2. Object identities must be preserved; a structural type reference to an object must be the same as a nominal type reference to it.
3. Compilation should be separate without relying on a whole-world analysis; WHITEOAK compiler cannot consult all uses of a given reference type.
4. No executable blowup; the compiler cannot generate a nominal type system reflecting the structural hierarchy by generating a nominal type for each subset of the set members of all structural types; in fact our implementation generates one interface for each structural type, and optionally an implementing class if this reference type includes method implementations.

3.1 The Object Identity Problem

In order to better understand the difficulty in preserving the identities of objects, let us consider the following code fragment:

```
struct S { Object me() { return this; } }

Object o = new Object();
S s = o;
assert o == s && o == s.me();
```

In this fragment we have three object references, `o`, `s` and `s.me()`. Given that `s` is assigned from `o` and that `S.me()` returns **`this`**, it is only natural to expect the two equalities `o == s` and `o == s.me()` to hold. Follow-

ing JAVA's standard semantics, one can also conclude that `s == s.me()`.

Examining the standard techniques for adding behavior to existing objects, such as the DECORATOR pattern [14], we note they prescribe the use of a wrapper object whose identity is inevitably different than the identity of the wrapped object. Specifically, naïve wrapper-based techniques stop identities such as `o == s` and `o == s.me()` from holding.

Alternatively, object identity could also have been achieved by overriding the compiler default implementation of reference comparison operation. This seems to be the desired alternative in C++, where the frequent use of **`this`**-adjustment [15] forces such specialized comparison code even for nominal types. In JAVA however, such an addition is not only foreign to the language, but would have come at the cost of breaking previously compiled code. For example, the invariants of libraries relying on reflection may be invalidated even if the wrapper is hidden with a kwthis-adjustment technique.

WHITEOAK's design insists on both binary compatibility and on semantics of object identity (in particular the aforementioned assertions hold in WHITEOAK). This is achieved by a technique which may be called *Invisible Wrapper* described below.

(Note that the object identity challenge is accentuated with default function implementations: we must be able to attach an implementation to an existing object without changing the object nor the reference to it.)

3.2 Compile Time Representation

The WHITEOAK compiler represents every structural type, S, as an interface I_s. This interface is generated as soon as the parsing of S is complete and therefore it serves as the only representation of S during compilation. In other words, from the compiler's point of view, the structural type exists only as I_s; there is no representation for S per-se.

The mapping from S to I_s is straightforward: Each method (either abstract or non-abstract) declared in S is represented as an abstract method declaration in I_s. If S contains at least one method with default behavior, I_s will declare an abstract, inner, static class, C_s, which **`implements`** the interface I_s. Each definition of a non-abstract method in S is translated into a similar definition in C_s. Hence, C_s is referred to as the *Partial Implementation Class* of I_s.

Similarly, a constructor with a certain signature is represented as a method named **`constructor`** with this signature whose return type is I_s.

A field f declared in S is translated into a **`static final`** field of the same name and type in I_s. Also, a uniquely named getter method—corresponding to f—is declared in I_s. If f is a non-final field, I_s also contains a corresponding uniquely named setter method.

Finally, The I_s interface carries a special annotation that distinguishes I_s from "normal" interfaces (that is: interfaces that are not an artifact of a structural type declaration).

Under this translation scheme, I_s is a faithful representation of S: all elements of the structural type S are represented, without loss of information, as standard JAVA entities in I_s and its inner class C_s. Also note that I_s is the primary representation of S: whenever S is specified in the program as the type a variable, a return type of a method, or a bound on a type parameter, it is I_s that will replace it in the compiler's internal representation. The class C_s is merely a vehicle which "carries" the implementations of non-abstract methods from S: there is no way for the programmer to specify C_s as a type in the program.

Type Checking. Given that I_s and C_s are plain JAVA definitions, type checking can largely follow JAVA's familiar semantics. In particular, access to a method of a structural type is type-checked in the same manner as access to an interface-declared method. Access to a constructor of a structural type is syntactically identical to a method call (e.g.: `x = y.`**`constructor`**`(5)`). Consequently, constructor calls are type-checked like method calls. Finally, a read operation from a field of a structural type is handled as a read operation from a (**`static final`**) interface field.

The primary changes to the type checker are as follows:

- Assignment to a field of a structural type is allowed only if the field has a corresponding setter method.
- When a type conformance test is in order (e.g.: assignments, instantiation of generics) WHITEOAK's type checking algorithm (Section 2.4) is used in place of JAVA's nominal subtyping rules.

3.2.1 Code Generation and Invisible Wrappers

In order to support WHITEOAK constructs the bytecode generator has to deal with three primary issues: assignments into variables, run-time type tests and method invocation.

Assignments. The structural type `S` defined by

```
struct Subbable {
  String substring(int i);
}
```

is a supertype of the nominal type `String`, so if `sub` is a variable of type `Subbable` the assignment

```
sub = "whiteoak"
```

type checks correctly by the compiler. The JVM verifier [21, Sec. 4.9.1] however will reject the assignment if `sub` is a field or a method argument [7] since two the types **`class`** `String` and **`interface`** `Subbable` (representing internally the structural type) are nominally incompatible. The difficulty is resolved by a technique similar to the implementation of type erasure of generic classes:

[7] Unlike local variables, the verifier is aware of the declared type of fields and arguments.

variables of structural types take the type `Object` in their **`.class`** representation. The real (structural) type of these is preserved in an annotation so the symbol table can still reflect the correct type of definitions from separately compiled modules.[8]

Type tests. Typecasts and runtime subtyping tests [28] executed at runtime are another difficulty. The JVM does not acknowledge that `String` is a subtype of `Subbable`. This is the reason that the WHITEOAK compiler carefully replaces such tests by a call to a library function which executes, at runtime, the structural subtyping algorithm to compare the object's dynamic type with the given structural type. This function, just as other similar portions of the runtime library makes heavy use of caching. As it turns out, typecasts in the JVM are no different than subtyping tests.

More difficult is the problem of *dispatching*. Again, the call

```
sub.substring(5)
```

type checks correctly since interface `Subbable` has a public `String substring(`**`int`**`)` method. The emitted code must however dispatch the call to `substring(`**`int`**`)` from class `String` despite the fact that type `Object` has no such method (as a direct result of the erasure process described above, the JVM considers the `sub` variable to be of type `Object`). Even if the JVM type of `sub` was interface `Subbable` which includes such a method, and even if the assignment of a `String` to such a variable would have been possible, dispatching would be difficult since we have no access to the mechanism by which the JVM dispatches interface calls to the correct class implementation. (Note that in Section 4 we discuss the relation between WHITEOAK's structural dispatching and the proposed `invokedynamic` instruction).

Dispatching. Dispatching is realized in our implementation by class `Wrapper` of WHITEOAK's runtime library. This class effectively augments the JVM with an ability to virtual dispatch a method through a structural method selector. The factory method

```
public static<I> I make(Object content,
  Class<I> description);
```

in this class produces an *invisible wrapper* around its `content` argument, through which dispatching is carried out. Method `make` requires these two preconditions:

- The actual type passed to `I` is some I_s interface representing a structural type S.
- For a given I_s, the dynamic type of the `content` object is an instance of a class D that is structurally conforming to S.

The method's contract guarantees that it will return an instance of a "wrapper class", `W`, that is a *nominal* subclass of class C_s corresponding to I_s. Given that class C_s always **`implements`** I_s (ensured by the way it is built by the compiler) we have that `W` is a nominal subtype of I_s. Moreover, the fact that `W` subclasses C_s elegantly injects into the wrapper object the default implementation declared by the structural type.

The creation of `W` by `make()` follows this pattern: for each method m of I_s that has a compatible method m' in the D, class `W` defines a corresponding non-abstract method with the same signature as m whose body delegates to m' on the `content` object. This ensures that methods provided by the `content` will override default implementations of methods from S.

Method dispatching is realized by first creating an invisible wrapper around the receiver, and then using a nominal `invokeinterface` call to dispatch the call correctly. More specifically, the compiler emits bytecodes patterned after the algorithm depicted in Algorithm 7.

1: Load the receiver onto the stack.
2: Load the class literal I_s onto the stack.
3: Call `Wrapper.make()` via `invokestatic`.
4: Downcast the object at the top of the stack to I_s.
5: Load `m`'s parameters onto the stack
6: Call `m()` via `invokeinterface`.

Algorithm 7: JVM code to dispatch method `m` on a receiver of structural type `S`

It is straightforward to check that the downcast at the 4^{th} step always succeeds (since W is a nominal subtype of I_s). This ensures that the `invokeinterface` instruction at the 6^{th} step of the algorithm verifies correctly, since the receiver **`implements`** I_s. A concrete bytecode-level incarnation of this pattern is shown in Figure 3.1.

In the figure, each of the instruction (1)-(6) represents the corresponding step from Algorithm 7. Note that the invoked method takes only one parameter of type **`int`** so that step 5 becomes a single `bipush` instruction. Also note that steps `(1)`, `(5)`, `(6)` and `(7)` are the same as any virtual

```
(1) aload_1 // Load variable sub
(2) ldc_w #3 // class Subbable
(3) invokestatic #4
  // Method Wrapper.make(Object,Class)
(4) checkcast #3 // class Subbable
(5) bipush 5 // Push the constant 5
(6) invokeinterface #5
  // Method Subbable.substring(int)
(7) pop // Returned value is ignored
```

Figure 3.1: The bytecodes generated for the invocation `sub.substring(5)`, where `sub` is a variable of a structural type `Subbable` that declares the method `String substring(`**`int`**`)`.

[8] Incidentally, the JVM's type verification of interfaces makes it possible to avoid using this technique for fields (but not for method arguments).

function call, and that the invisible wrapper, generated at step (3) vanishes when the call is finished.

The WHITEOAK compiler generates the aforementioned sequence of instructions only when it generates the code for an invocation of a method on a receiver of a (static) structural type. Assignments to such variables are allowed only if the static type of assigned value is structurally conforming to the type of the variable. This guarantees that in each compiler-generated invocation of `Wrapper.make()` the preconditions required by the method are met.

We further argue that class `W` is always a concrete class: If the `content` object does not provide a suitable implementation for m and m is non abstract in S, then class C_s will have a concrete m method (again, this is ensured by its building process). class `W` will inherit this implementation from its superclass C_s.

In the dual case—D does not provide a suitable implementation for m and m is *abstract* in S—we have that in all the nominal supertypes of D (including D itself) there is no declaration of a method m, not even an **`abstract`** declaration. If there were such an abstract declaration then D were an abstract class which cannot, by definition, be the dynamic type of an object. Given that no definition for m exists in the hierarchy above D we have that D is not structurally conforming to S (see Section 2.4) in contradiction to the precondition of the `make()` method.

The fact that `W` is a concrete class guarantees the following: (i) `Wrapper.make()` will be able to create an instance of `W`; (ii) the method invocation in the 6^{th} step of Algorithm 7 will succeed.

Constructors and Fields. Dispatching to constructors is no different than methods. A call to a **`constructor`**`()` method on a structurally typed receiver is handled by a corresponding **`constructor`**`()` method in class `W` whose body carries out the standard bytecode sequence for creating a new object of type `S`.

Access to fields of a structural type is made possible by invocation of the appropriate getter or setter method introduced by the compiler into I_s. In the wrapper class `W` the implementation of such methods uses either a `GETFIELD` or a `PUTFIELD`) instruction to carry out the actual operation on the field of the `content` object.

`this` ***References.*** In a structural type that defines only abstract methods, the identity of the wrapper object is never leaked to the actual program code: the methods of the wrapper object merely delegate to the methods of the `content` object without exposing the **`this`** reference.

Things are more complicated when considering methods with default implementations. When such a method gets executed (that is: the `content` object does not provide a suitable implementation), the actual program code runs in the context of the wrapper object thereby exposing the identity of the wrapper.

To solve this difficulty, the compilation of methods with default implementations into the partial implementation class, C_s, is altered so that each **`this`** reference is replaced by a reference to the wrapped object. To achieve this, the partial implementation class defines an abstract function `getReceiver()` whose return type is I_s. In methods of a partial implementation class, the code generator inserts a `getReceiver()` call immediately after every "load **`this`**" instruction (bytecode: `aload_0`).

The net effect is two fold: First, the reference to the wrapper object is replaced by a reference to the wrapped content, thus maintaining the transparency of the identity of the wrapper object. Second, in self calls, such as m_1 invoking m_2, the static type of the receiver becomes I_s (since `getReceiver()` returns I_s) so the compiler will treat the call as a structural invocation thereby ensuring correct dispatching semantics of self calls.

We note that this process however does not need to generate a new delegating class, since both the dynamic nominal type and the structural static description are the same as in the context which generated the wrapper object.

3.3 Performance

The implementation approach described above realizes structural dispatching by these steps: (i) creation of an invisible wrapper and (ii) using the JVM's native `invokeinterface` instruction to dispatch to a delegator and (iii) delegation to the actual implementation. As it turn out, the first such step is the most time-intensive.

This step involves two operations which are notoriously slow: the examination of a reflection object, and the creation, at runtime, of a new class. In fact, in an early implementation of WHITEOAK we found that dispatching based on reflection information was at least three order of magnitude slower than native dispatching.

The current WHITEOAK implementation achieves good performance by relying on two levels of caching inside the factory method `make`: First, a fixed size cache containing 8 entries and implemented with no looping instructions is used to check whether an invisible wrapper was previously created for the specified value of the receiver and the static (structural) type of the receiver. If the pair is not found in this cache, a hash table containing all previously returned wrappers is examined.

In both caches the searching strategy is similar: we first check whether an exact match is found, that is, match in both receiver value and in the receiver's (structural) type. If this fails, the dynamic type of the receiver is fetched, and the cache is examined again to see whether a wrapper class appropriate for the receiver's dynamic type and the receiver's static type exists already and can be instantiated to create the invisible wrapper.

In addition, within any single thread the cache may even recycle wrappers by changing their contained wrapped ob-

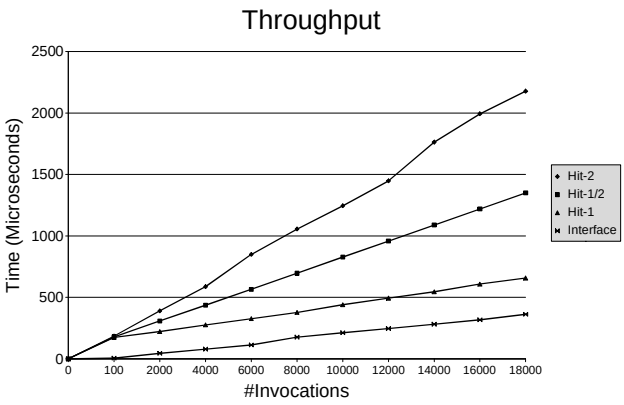

Figure 3.2: Execution time of a program vs. # number of method invocations.

jects. Such recycling is restricted to plain structural types though.

Figure 3.2 compares the timings of runs of a reference program in WHITEOAK with that of the JAVA equivalent using runtime interface dispatch. (Experiments were conducted on a single processor Pentium-4 3GB RAM, 3GHz, Windows XP machine.)

The curve marked "*Interface*" refers to the plain JAVA run. The "*Hit-1*" curve describes the WHITEOAK timing when the program mostly hits the primary cache. The curve marked "*Hit-2*" corresponds to the situation where structural dispatching mostly hits the secondary cache, while the "*Hit-1/2*" curve denotes an intermediate situation.

As expected, the curves are linear at large, indicating that the invocation time is constant. Hits on the primary cache incur a slowdown of a factor of about two in comparison to the reference `invokeinterface` implementation. Note that we cannot hope for more; the dispatching algorithm described above, even if caching requires no resources, replaces each structural dispatch with one interface dispatch and two ordinary class based dispatches. The hit-1/2 scenario is more than three times slower where the hit-2 scenario is about seven times slower.

The experiment depicted in this figure represents an unrealistic situation where the program spends an overwhelming majority of its time invoking an empty method. A more realistic program is likely to take additional computation which will mitigate the influence of dispatching on the overall execution time.

To evaluate performance under these circumstances we took two standard benchmarks, JESS and JAVALEX[9] from the famous SpecJVM98 suite and changed them such that every **`interface`** was replaced by a corresponding **`struct`**. Of course, this change meant that **`implements`** clauses were eliminated from all classes.

[9] We use the term JAVALEX instead of the formal name of this benchmark, JAVAC, which is confusing in the context of this paper.

The JESS program is a rule-based inference system. The JAVALEX benchmark measures the time needed for a JAVA compiler to compile a 60,000 lines program. In the original benchmark, the compiler whose compilation time is measured is one of the early versions of Sun's JAVA compiler. We changed the benchmark to measure the compilation time of Sun's JAVA 5 compiler against that of the bootstrapped WHITEOAK compiler (The bootstrapped version uses **`struct`**s instead of **`interface`**s). The use of interfaces in the remaining SpecJVM benchmark programs was minimal, so applying this techniques in these did not yield useful data.

In both benchmarks the performance of the WHITEOAK program was less than 5% slower than JAVA program. Specifically, In JESS WHITEOAK time was 1.95 seconds while the JAVA time was 1.89 seconds, while in JAVALEX the respective times were 3.9 and 3.75 seconds;

Finally, we compared WHITEOAK's structural dispatching implementation with that of SCALA, in the simple case of repeatedly invoking a method on a single receiver. The slowdown in SCALA was more than two orders of a magnitude, compared to less than one order of a magnitude in WHITEOAK (as shown in Figure 3.2). Note, that we chose such a simple case since some of the features that are needed for a more complicated test, e.g., array access, incur substantially different run-time costs in the two languages, thereby biasing the method invocation measurement that we seek.

Due to space considerations we could not discuss all issues related to the synergy of the two typing paradigms. In particular, we left out difficulties arising from Java's security system, and the implications of multithreading on our caching mechanism.

4. Summary and Future Work

The introduction of structural types into a nominally typed language has two major implication on the style of programming and design in that language.

First, the use of structural types increase locality: If a type is needed in some place in the program we only need to define it in the point where it is used. There is no need to change the definition of classes (which are typically scattered around the program) in order to make this new type viable.

This enables concerns to be confined to a well-defined region of the program, resulting in increased coherence of the program's modules. Locality is particularly important in maintaining the balance of power between supplier's and client's code. A change in a supplier module due to a client's needs may eventually lead to deterioration of the design of the supplier, especially when there are multiple clients that inflict changes onto one supplier. Retroactive abstraction obviates many of these changes, thereby preserving the quality of the supplier's design.

The second effect is that of increased uniformity. As argued by Meyer's *principle of uniform access* similar services should be accessible through a single syntactic device carrying a simple, uniform semantics. Looking at JAVA, we see several breaches of this principle, for example: methods and fields are accessed through the same syntax, but with different binding semantics; Procedural abstraction is realized by two kinds of methods, **`static`** and instance methods accessible by slightly different notations, and carrying different binding semantics; finally, there is a dedicated syntax for constructor calls. WHITEOAK allows the programmer to abstract away most of these differences, thus making it easier to read, write, and maintain the source code.

Moreover, the on-going struggle of augmenting the precision and expressiveness of static type systems leads to the emergence of multi-paradigm languages which exhibit a large, feature-rich core that only emphasizes the point in favor of uniform access (In contrast, the elegant, minimal structure of dynamically typed languages enable these to better maintain Meyer's principle).

The semantics of dispatching methods on plain structural types is quite close to that of the proposed `invokedynamic` instruction[10]. Indeed every language implementation with structural types could rely on this new instruction when it becomes available. Alternatively, the performance data reported in this paper provide a reference point for evaluating future implementations of `invokedynamic`.

Support for **`struct`**s with non-abstract methods, exceeds, the abilities of `invokedynamic`. The implementation details presented here are therefore important for any language that will attempt to provide similar services, without changing the underlying virtual machine. We assume that the research effort aimed at better utilization of the JVM will become more and more important as the JVM gradually turns into a safe and powerful computing platform that can host a large array of languages. (The JVM's support for scripting languages, in which `invokedynamic` plays a major role, is one of the signs in that direction). In fact, at this point one can reasonably predict that the JVM will actually outlive the JAVA language.

A promising direction for future research is the use of structural types in conjunction with first-class support for data queries. One of the major predicaments facing the integration of (say) SQL queries into JAVA is the correct typing of the query results. If two different program-embedded SQL queries return similarly structured data it is only natural to expect this data to be of the same type. Locally inferring the nominal type of the result is only a partial solution since the queries may be located at two separately compiled modules. In the absence of a whole program analysis, the inferred types will be nominally different. A structural typing scheme is therefore the natural solution for allowing the interplay of such two types.

[10] http://www.jcp.org/en/jsr/detail?id=292

References

[1] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Trans. on Prog. Lang. Syst.*, 15(4):575–631, 1993.

[2] D. Ancona, G. Lagorio, and E. Zucca. Jam — a smooth extension of Java with mixins. In E. Bertino, editor, *Proc. of the 14th Euro. Conf. on OO Prog. (ECOOP'00)*, volume 1850 of *LNCS*, pages 154–178, Sophia Antipolis and Cannes, France, June 12–16 2000. Springer.

[3] T. Andrews and C. Harris. Combining language and database advances in an object-oriented development environment. In N. K. Meyrowitz, editor, *Proc. of the 2nd Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'87)*, pages 430–440, Orlando, Florida, Oct. 4-8 1987. ACM SIGPLAN Notices 22(12).

[4] M. P. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–170, 1987.

[5] M. P. Atkinson and R. Welland. *Fully Integrated Data Env.: Persistent Prog. Lang., Object Stores, and Prog. Env.* Springer, Secaucus, NJ, USA, 2000.

[6] G. Baumgartner and V. F. Russo. Implementing signatures for C++. In *Proc. of the 6th USENIX C++ Conf.*, pages 37–56, Cambridge, MA, Apr. 1994. USENIX Association.

[7] G. Baumgartner and V. F. Russo. Implementing signatures for C++. *ACM Trans. on Prog. Lang. Syst.*, 19(1):153–187, Jan. 1997.

[8] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Dept. of Comp. Sci., University of Utah, 1992.

[9] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. of the 13th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'98)*, pages 183–200, Vancouver, British Columbia, Canada, Oct.18-22 1998. ACM SIGPLAN Notices 33(10).

[10] K. B. Bruce and J. N. Foster. LOOJ: Weaving LOOM into Java. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04)*, pages 389–413, Oslo, Norway, June 14-18 2004.

[11] M. Büchi and W. Weck. Compound types for java. In *Proc. of the 13th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'98)*, pages 362–373, Vancouver, British Columbia, Canada, Oct.18-22 1998. ACM SIGPLAN Notices 33(10).

[12] B. Eckel. *Thinking in C++*. Prentice-Hall, Upper Saddle River, NJ, USA, 1995.

[13] R. P. Gabriel, L. Northrop, D. C. Schmidt, and K. Sullivan. Ultra-large-scale systems. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications*, pages 632–634. ACM Press, 2006.

[14] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing series. Addison-Wesley, Reading, Massachusetts, 1995.

[15] J. Gil and P. F. Sweeney. Space- and time-efficient memory layout for multiple inheritance. In *Proc. of the 14th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'99)*, pages 256–275, Denver, Colorado, Nov.1–5 1999. ACM Press, ACM SIGPLAN Notices 34 (10).

[16] J. Gosling, B. Joy, G. L. J. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 3rd edition, June 2005.

[17] W. Harrison, D. Lievens, and T. Walsh. Using recombinance to improve modularity. Technical Report 104 Software Structures Group, Trinity College Dublin, Dublin, Ireland, March 2007.

[18] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In J. Palsberg and M. Abadi, editors, *Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Lang. (POPL'05)*, pages 50–62. ACM Press, 2005.

[19] B. N. Jørgensen. Integration of independently developed components through aliased multi-object type widening. *Journal of Object Technology*, 3(11):55–76, 2004.

[20] K. Läufer, G. Baumgartner, and V. F. Russo. Safe structural conformance for Java. *The Computer Journal*, 43(6):469–481, 2001.

[21] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1999.

[22] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, pages 260–284, Paphos, Cyprus, July 7-11 2008.

[23] E. R. Murphy-Hill, P. J. Quitslund, and A. P. Black. Removing duplication from java.io: a case study using traits. In *OOPSLA '05: Companion to the 20th ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications*, pages 282–291, New York, NY, USA, 2005. ACM Press.

[24] G. W. Neufeld and S. T. Vuong. An overview of ASN.1. *Comp. Networks and ISDN Sys.*, 23(5):393–415, Feb. 1992.

[25] M. Odersky. The Scala experiment: can we provide better language support for component systems? In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*, pages 166–167, Charleston, South Carolina, USA, January 11-13 2006.

[26] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behavior. In L. Cardelli, editor, *Proc. of the 17th Euro. Conf. on OO Prog. (ECOOP'03)*, volume 2743 of *LNCS*, pages 248–274, Darmstadt, Germany, July 21–25 2003. Springer.

[27] J. W. Schmidt. Some high level language constructs for data of type relation. *ACM Trans. on Database Sys.*, 2(3):247–261, Sept. 1977.

[28] Y. Zibin and J. Gil. Efficient subtyping tests with PQ-encoding. In *Proc. of the 16th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'01)*, pages 96–107, Tampa Bay, Florida, Oct. 14–18 2001. ACM Press, ACM SIGPLAN Notices 36(11).