Implementing Fully Modular, Statically Typed, Symmetric Multimethod Dispatch

Guy L. Steele Jr.
Sun Labs, Oracle
guy.steele@oracle.com

David Chase
Sun Labs, Oracle
david.r.chase@oracle.com

Eric Allen
Sun Labs, Oracle
eric.allen@oracle.com

Victor Luchangco
Sun Labs, Oracle
victor.luchangco@oracle.com

Sukyoung Ryu
KAIST
sryu@cs.kaist.ac.kr

Abstract

The Fortress programming language integrates traditional mathematical notation into an object-oriented framework based on traits with multiple inheritance, overloading (of both methods and functions) resolved by symmetric dynamic dispatch, static types, and separately compiled modules. An innovation of particular interest is *functional methods*, which (like conventional "dotted methods") are declared within traits and may be inherited, but are invoked by ordinary function calls (or mathematical operator syntax) rather than conventional "dotted method calls," and therefore compete in overloading resolution with ordinary function declarations. A component/API system governs visibility of traits, objects, and functions, but it is desirable to regard overriding functions and methods that are visible to the callee as part of its implementation, even though they may not be visible to the caller.

A longstanding problem with multiple inheritance is what to do when methods inherited from several parents conflict. Many approaches have been explored in the literature; some unfortunately violate the intuitively desirable requirement that the function or method invoked be the uniquely most specific one that is both accessible and applicable. Fortress requires that the signatures in every overload set form a meet-bounded lattice; therefore it is impossible for any function or method call to be ambiguous. This idea goes back nearly two decades, but Fortress appears to be the first programming language to adopt and statically enforce it. Because this rule guarantees confluence, it enables a distributed implementation of dispatching that allows selective export and selective optimization.

We exhibit a source-to-source rewrite from a source language (a stripped-down version of Fortress) to a related target language that is simpler than the Java$^{\text{TM}}$ programming language and is readily supported by the Java Virtual Machine. The demonstrated rewriting is a practical basis for separate compilation and is easily extended to explicitly type-parameterized methods and functions.

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

General Terms Languages

Keywords object-oriented programming, traits, multiple dispatch, symmetric dispatch, multiple inheritance, overloading, modularity, multimethods, static types, components, separate compilation

1. Introduction and Background

The design of the Fortress programming language [2] began in 2003 as part of the DARPA program for High Productivity Computing Systems, aimed at improving programmer productivity for high-end scientific applications. The programming languages X10 [21] and Chapel [7] were also designed under this program. The three efforts had the same overall goal, but chose rather different design principles and premises, and consequently these efforts produced three languages with very different styles.

As its name might suggest, Fortress was intended to feel reasonably "familiar" to Fortran programmers, supporting, at the very least, an array-oriented, data-parallel style of programming in addition to explicit threads and a synchronization mechanism. But the design team, inspired by the notion that a programming language should be "growable" [22], was reluctant to cast design decisions about such an array-oriented style in concrete early on, and felt that an object-oriented framework would better provide the extensibility and modularity that would allow implementation of the array features as libraries.

1.1 Traits, Objects, Methods, and Functions

Fortress organizes objects into a multiple-inheritance hierarchy based on *traits*. It may be helpful for the reader familiar with the Java language [16] to think of a Fortress trait as similar to a Java interface that can contain concrete method definitions, and to think of a Fortress object as similar to a Java final class. Fortress objects inherit only from traits, not other objects, and fields are not inherited. Both traits and objects may contain (concrete) method definitions, and may inherit method definitions and declarations from multiple supertraits (that is, traits that they *extend*). Traits may also contain (abstract) method declarations, which impose upon objects that inherit them (directly or indirectly) the requirement to provide matching concrete definitions. Thus all traits and objects form a multiple inheritance hierarchy, a partial order defined by the `extends` relationship, in which all objects are at the leaves. As in the Java language and similar object-oriented languages that have static type systems, the name of a trait or object serves to name a type, which is the set of all objects at or below the named position in the trait hierarchy. In addition to objects (all of which are of type Object, as in the Java language), Fortress has *arrow types* $D \rightarrow R$ (the types of functions) and *tuples* written as $(e_1, e_2, \ldots, e_n)$ whose types form a partial order in the usual conjunctive elementwise manner: $(\mathrm{T}_1, \mathrm{T}_2, \ldots, \mathrm{T}_m) \trianglelefteq (\mathrm{U}_1, \mathrm{U}_2, .., \mathrm{U}_n)$ if and only if $m = n$ and, for all $1 \leq k \leq m$, $\mathrm{T}_k \trianglelefteq \mathrm{U}_k$. Objects, functions, and tuples are all *values*, and they all have type Any.

Like C++ [24] (and unlike the Java language [16] and C# [13]), Fortress provides not only methods associated with objects but also functions not associated with any object; moreover, Fortress function definitions and declarations may be either top-level or local to a block. (Note, by the way, that we use the term *definition* to refer to a syntactic construct that defines a variable, function, or method, and furthermore contains an executable expression to supply the value of the variable, invoked function, or invoked method. We use the term *declaration* when such an expression is not present.)

As in many previous languages, both functions and methods may be *overloaded*; that is, there may be many methods within an object, or many functions declared within the same scope, that have the same name. This raises the issue of *overload resolution*: given a method call or function call, how does the language determine which definition(s) should be invoked? If a single definition is to be chosen and invoked, then we call this determination process the *dispatch mechanism*. A typical dispatch mechanism identifies a pool of candidate definitions that are *accessible* and *applicable*, then selects from that pool the one that is *most specific*. Many dispatch mechanisms have been explored in the literature, and they differ in their definitions of "accessible" and "applicable" and "most specific"; in particular, they may differ in whether static type information, dynamic (run-time) information about the arguments, or both are brought to bear in defining each of these three terms. If static information is used, then the dispatch mechanism may have two stages, one performed at some time prior to program execution (such as compile time) and one performed during program execution. For example, the Java language uses a *single dynamic dispatch* mechanism for method calls, in which both static and dynamic information about the receiver object is used but only static information about the other arguments is used; a method signature is chosen at compile time based on all the static information, and further dispatch occurs at run time based on the actual class of the receiver object (which may turn out to be more specific than the static type of the receiver object expression). As another example, the Common Lisp Object System (CLOS) [14, 4, 23, 15] performs *asymmetric multiple dynamic dispatch* in which only dynamic information is used, but for all arguments, not just one designated "receiver" argument. (While Common Lisp used the term "generic function," nowadays that term is usually used to refer to a function that is polymorphic by virtue of having explicit static type parameters, and the term *multimethod* is used to refer to a set of methods with a dispatch discipline that takes into account dynamic information from more than one argument.) While CLOS performs dynamic dispatch on all arguments, the arguments are treated asymmetrically in that the most specific method is chosen by testing the parameter types of the candidate methods in a specific order (normally left-to-right, though this order can be modified for a particular overload set by an explicit declaration). In contrast, Fortress has the behavior of *symmetric multiple dynamic dispatch*: the method ultimately chosen depends on the ilks of *all* the arguments. (We use the term *ilk* for what is sometimes confusingly called a "run-time type"; just as the Java language has had the slogan "Variables have types, objects have classes" [16, §4.5.5] so Fortress has the slogan "Expressions have types, values have ilks.") However, Fortress also makes use of static type information to perform part of the dispatch at compile time, so it has a two-stage dispatch process. (As we will see, this two-stage process is designed to have the same behavior as a single-stage, fully dynamic dispatch, so the use of static information serves purely as a performance optimization.) Moreover, which methods are accessible depends in part on visibility constraints that may be imposed by dividing a program into components (see Section 1.3). One goal of this paper is to illuminate all these details.

Languages also differ in whether a method or function call can be *ambiguous* in the presence of overloading, and in how ambiguity is detected and handled. Let us assume that the type `String` is a subtype of `Object`. In the Java language, for example, one may declare two methods as follows:

```
int gnard(a: String, b: Object) { return 1; }
int gnard(a: Object, b: String) { return 2; }
```

Whatever the type of the receiver object `x`, we may observe that a method call `x.gnard("foo", "quux")` having two strings as arguments will be ambiguous, because both methods are applicable and neither is more specific than the other. As another example, consider three Java interfaces named `A`, `B`, and `C`—where `C` extends both `A` and `B`—and then consider these three methods occurring in some class:

```
void zam(p: C) { System.out.println(jax(p)); }
int jax(q: A) { return 1; }
int jax(q: B) { return 2; }
```

Although the Java language does not have multiple inheritance, it does have multiple supertypes (through the interface hierarchy), and this allows the construction of a method call `jax(p)` that is type-correct but ambiguous. *The Java Language Specification* [16, §15.11.2.2] stipulates that it is permissible to declare such sets of methods, but it is a static error for a program to contain a method call that will be ambiguous in their presence. In contrast, Fortress stipulates that an ambiguous set of overloaded method or function declarations in itself constitutes a static error, even if no call to such a method or function appears in the program.

There is yet another distinction to be made: some languages with multiple inheritance, such as MultiJava [11, 12] and Scala [20], have *asymmetric inheritance*, in which one superclass or supertrait may be treated differently from another based on a criterion such as order of declaration, and this asymmetric treatment may affect the definition of which method is considered "most specific" by the dispatch mechanism (for example, ties might be broken by choosing the method inherited from the supertrait declared earlier). Fortress, in contrast, uses *symmetric inheritance*; methods are inherited on an equal footing from all supertraits. This is another possible source of method ambiguity: two methods with apparently identical signatures may be inherited from two different supertraits. For example, suppose that we have three Fortress traits as follows:

$\texttt{trait}\ \mathrm{C}\ \texttt{extends}\ \{\,\mathrm{A},\mathrm{B}\,\}$
$\quad zam()\colon () = println\big(\texttt{self}.jax()\big)$
$\texttt{end}$

$\texttt{trait}\ \mathrm{A}$
$\quad jax()\colon \mathbb{Z}32 = 1 \qquad ⊛\ \mathbb{Z}32$ is the type of 32-bit integers
$\texttt{end}$

$\texttt{trait}\ \mathrm{B}$
$\quad jax()\colon \mathbb{Z}32 = 2$
$\texttt{end}$

This is similar to the previous example. Because definitions of method jax are inherited symmetrically by trait C, it's a tie: it is not clear which one should be invoked by the method call $\texttt{self}.jax()$ in method zam. (By the way, $\texttt{self}.jax()$ could have been written more succinctly as simply $jax()$, just as in the Java language; we chose the longer form for this example for the sake of clarity.) So this example is a static error in Fortress and will be rejected by the compiler (indeed, it would be rejected even if the method definition for zam were removed).

Fortress stipulates that such ties must be broken by the programmer. Following Castagna *et al.* [6], Fortress requires that the signatures of all the functions or methods in an overload set form a *meet-bounded lattice*—we call this the Meet Rule. Simply put, for every pair of accessible signatures for which it might be possible to construct an ambiguous call (that is, both are applicable and neither is more specific than the other), there must be a third accessible signature that (a) is more specific than each of the other two signatures, and (b) is also applicable to the call, and therefore will be chosen unambiguously over the other two. We can repair the previous example by introducing an appropriate definition of jax into trait C. This might simply supply a value on its own:

$\quad jax()\colon \mathbb{Z}32 = 47$

or choose to direct control to one of the inherited definitions:

$\quad jax()\colon \mathbb{Z}32 = (\texttt{self asif}\ \mathrm{B}).jax()$

or might even make use of more than one of the inherited definitions:

$jax(){:}\, \mathbb{Z}32 = (\texttt{self asif } \mathrm{A}).jax() + (\texttt{self asif } \mathrm{B}).jax()$

(As we shall see, the Fortress `asif` construction addresses the same issues that `super` does in the Java language.)

In addition to the Meet Rule, Fortress imposes the usual Subtype Rule: for any pair of accessible functions or methods in an overload set, if the first is more specific than the second, then the result type of the first must be a subtype of the result type of the second. This rule is necessary to maintain type soundness.

Fortress also has parametric polymorphism of traits, objects, methods, and functions; any of these may have static type parameters, and references to them may supply explicit static arguments, which may be types, boolean values, integers, or operator symbols. However, we do not address parametric polymorphism of traits and objects in this paper, and we discuss parametric polymorphism of functions and methods only briefly, in Section 8.

1.2 Functional Methods

As the Fortress design team has previously reported [3], Fortress has not only *dotted methods* that are declared much as in the Java language and are invoked with a *dotted method call* of the form:

$receiver.methodName(e_1, e_2, \ldots, e_n)$

but also *functional methods*. The declaration of a functional method differs from that of a dotted method in that exactly one of the parameter $name{:}\,\mathrm{Type}$ declarations in the header is replaced with the keyword `self`; in effect, that argument position, rather than a separate expression "before the dot," is treated as the receiver. This feature easily solves the "binary method problem" [5] by allowing, for example, a Matrix trait to define both matrix-vector multiplication and vector-matrix multiplication with respect to a third-party Vector trait:

```
trait Matrix excludes { Vector }
  multiply(self, v: Vector): Matrix = ...
  multiply(v: Vector, self): Matrix = ...
end
```

(note the declaration that Matrix `excludes` Vector, that is, no object may extend both of those traits, and therefore the overloading of *multiply* cannot be ambiguous). With dotted methods it might be necessary to modify the definition of Vector.

Functional methods are invoked using ordinary function call syntax, for example $multiply(\mathbf{w}, \mathbf{A})$ to multiply the vector $\mathbf{w}$ by the matrix $\mathbf{A}$. In that early report [3], Fortress allowed functions to be overloaded with other functions and functional methods to be overloaded with other functional methods. Since then Fortress has been changed to allow functions and functional methods to participate in the same overload set. One goal of this paper is to explain exactly how that works. A necessary restriction is that there must not be any top-level function whose signature is more specific than that of a functional method within the same overload set.

1.3 Components and APIs

Just as the Java language allows a program to be divided into *compilation units* that can be separately compiled, and also has *packages* that provide a modicum of namespace control, Fortress allows a program to be divided into *components*. Interfaces between components are described by *APIs*. Code in one component cannot refer directly to code in another component; rather, any component or API can *import* identifiers from one or more APIs, and any component can *export* one or more APIs. Each API must be exported by exactly one component.

Fortress also has mechanisms for bundling a set of components into a single larger component, and this process may be carried out recursively, thus providing namespace control for the names of components and APIs. The details of this are beyond the scope of this paper; here we will simply assume that the names of all components are distinct and the names of all APIs are distinct.

One goal of this paper is to explain how overload sets and overload resolution interact with the namespace control imposed by components and APIs in Fortress.

2. Contributions of This Work

We present a strategy for implementing symmetric multimethod dispatch in a statically typed language in which the program can be divided into components that can be separately type-checked and separately compiled, *such that the components provide complete namespace control over all names*; names of functions and methods, not just traits and objects, may be selectively exported. Obeying the Meet Rule makes this stategy possible. In addition, we show how to accommodate *functional methods*, a novel form of definition that inherits like a dotted method but overloads like a top-level function.

3. The Source Language

Here we summarize the source language. Please refer to Figure 1.

A program in the source language is a collection of components and APIs; their order does not matter.

Each API contains declarations of traits and functions; their order does not matter. (Actual Fortress APIs also allow declarations of objects and variables, but the bookkeeping related to those is not relevant to the theme of this paper.) An API may also import traits and functions from other APIs, thus allowing aggregation of APIs.

Each component contains definitions of traits, objects, top-level functions, and top-level variables, as well as (abstract) declarations of top-level functions. A component may also import names of traits and functions from APIs; their implementations will be supplied by other components. A component may export one or more APIs; in order to export an API, a component must provide a (concrete) definition corresponding to every declaration appearing in the API. (A Fortress component can "provide" a definition either by containing an actual definition or by importing it from another API; for present purposes we will require that the component itself contain the definition.) Importing a trait from an API enables invocation of dotted methods declared within that trait in the API.

Import declarations can rename traits and functions as they are imported, so that the referring component can use a name different from the one specified in the API.

A trait declaration may contain declarations and (when within a component) definitions of dotted methods and functional methods. A trait declaration may also contain what looks like a declaration of a field, but (as we shall see) this is really regarded as a requirement that appropriate getter and setter methods be defined by any implementing object. A trait may also include any or all of three clauses that relate it to other traits and objects: (1) A trait A may *extend* one or more other traits $B_1, B_2, \ldots, B_p$, in which case for each $1 \leq j \leq p$, A is a subtype of B_j, and A inherits all the methods of B_j that it does not override. (2) A trait A may *comprise* one or more other traits $C_1, C_2, \ldots, C_q$, in which case it is a static error for any trait or object D to extend A unless D is a subtype of (and possibly equal to) at least one C_j for some j such that $1 \leq j \leq q$; furthermore, each C_j must be declared in the same component as A and must extend A. (3) A trait A may *exclude* one or more other traits $E_1, E_2, \ldots, E_r$, in which case it is a static error if any trait or object F extends A and also extends E_j for any j such that $1 \leq j \leq r$. (The `comprises` and `excludes` clauses do not figure prominently in this paper, but we note that they are very important in practice in a multiple-inheritance framework in order to indicate that certain edges *cannot* occur in the type hierarchy. Without these clauses, it would be difficult or impossible to get certain useful method overloadings to conform to static error checks.)

Program ::= $\big($ *API* | *Component* $\big)^*$
API ::= `api` *ApiName* *Import** $\big($ *ApiTraitDecl* | *FunctionDecl* $\big)^*$ `end` *ApiName*
Component ::= `component` *ComponentName* *Import** *Export*$^+$ *ComponentItem** `end` *ComponentName*
ComponentItem ::= *TraitDefn* | *ObjectDefn* | *FunctionDecl* | *FunctionDefn* | *VariableDefn*
Import ::= `import` *ApiName* `.` { *ImportItemList* }
ImportItemList ::= *ImportItem* $\big($ `,` *ImportItem* $\big)^*$
ImportItem ::= *TraitName* $\big(\mapsto$ *TraitName* $\big)^?$ | *FunctionName* $\big(\mapsto$ *FunctionName* $\big)^?$
Export ::= `export` *ApiName*
ApiTraitDecl ::= `trait` *TraitName* *Extends*$^?$ *Comprises*$^?$ *Excludes*$^?$ *ApiTraitItem** `end`
ApiTraitItem ::= *VariableDecl* | *DottedMethodDecl* | *FunctionalMethodDecl*
TraitDefn ::= `trait` *TraitName* *Extends*$^?$ *Comprises*$^?$ *Excludes*$^?$ *TraitItem** `end`
ObjectDefn ::= `object` *ObjectName* $\big($ `(` *ParameterList*$^?$ `)` $\big)^?$ *Extends*$^?$ *ObjectItem** `end`
Extends ::= `extends` { *TraitNameList* }
Comprises ::= `comprises` { *TypeNameList* }
Excludes ::= `excludes` { *TypeNameList* }
TraitItem ::= *ApiTraitItem* | *DottedMethodDefn* | *FunctionalMethodDefn*
ObjectItem ::= *VariableDefn* | *DottedMethodDefn* | *FunctionalMethodDefn*
DottedMethodDecl ::= $\big($ `getter` | `setter` $\big)^?$ *MethodName* `(` *ParameterList*$^?$ `)` : *Type*
DottedMethodDefn ::= *DottedMethodDecl* = *Expression*
FunctionalMethodDecl ::= *FunctionName* `(` *SelfParameterList* `)` : *Type*
FunctionalMethodDefn ::= *FunctionalMethodDecl* = *Expression*
FunctionDecl ::= *FunctionName* `(` *ParameterList*$^?$ `)` : *Type*
FunctionDefn ::= *FunctionDecl* = *Expression*
VariableDecl ::= `var`$^?$ *VariableName* : *Type*
VariableDefn ::= *VariableDecl* $\big($ = | := $\big)$ *Expression* | *SimpleBinding* = *Expression*
SimpleBinding ::= *VariableName* | `(` *SimpleBindingList*$^?$ `)`
SimpleBindingList ::= *SimpleBinding* $\big($ `,` *SimpleBinding* $\big)^*$
TraitNameList ::= *TraitName* $\big($ `,` *TraitName* $\big)^*$
TypeNameList ::= *TypeName* $\big($ `,` *TypeName* $\big)^*$
TypeName ::= *TraitName* | *ObjectName*
Type ::= *TypeName* | `(` *TypeList*$^?$ `)` | *Type* $\rightarrow$ *Type*
TypeList ::= *Type* $\big($ `,` *Type* $\big)^*$
ParameterList ::= *VariableDecl* $\big($ `,` *VariableDecl* $\big)^*$
SelfParameterList ::= $\big($ *VariableDecl* `,` $\big)^*$ `self` $\big($ `,` *VariableDecl* $\big)^*$
Expression ::= *VariableName* | *Literal* | `self` | `(` *ExpressionList*$^?$ `)` | *Assignment* |
    FunctionCall | *DottedMethodCall* | *FieldAccess* | *FieldAssignment* | *IfThenElseEnd* | *DoEnd*
ExpressionList ::= *Expression* $\big($ `,` *Expression* $\big)^*$
Assignment ::= *VariableName* := *Expression*
FunctionCall ::= *FunctionName* `(` *CallExprList*$^?$ `)`
DottedMethodCall ::= *CallExpr* `.` *MethodName* `(` *CallExprList*$^?$ `)`
CallExpr ::= *Expression* $\big($ `asif` *Type* $\big)^?$
CallExprList ::= *CallExpr* $\big($ `,` *CallExpr* $\big)^*$
FieldAccess ::= *CallExpr* `.` *VariableName*
FieldAssignment ::= *CallExpr* `.` *VariableName* := *CallExpr*
IfThenElseEnd ::= `if` *Expression* `then` *StatementList* $\big($ `elif` *Expression* `then` *StatementList* $\big)^*$ $\big($ `else` *StatementList* $\big)^?$ `end`
DoEnd ::= `do` *StatementList* `end`
StatementList ::= $\big($ *FunctionDefn* | *VariableDefn* | *Expression* $\big)^*$ *Expression*

Figure 1. Grammar for source language

Object declarations may contain definitions of dotted methods, functional methods, and variables; variable definitions within an object declaration are regarded as fields of the object. An object declaration may contain an `extends` clause that mentions supertraits, just as for a trait declaration; in effect, an `object` declaration declares a trait of the same name that serves as a type name, but there is no need for such a trait to have a `comprises` clause or `excludes` clause because it cannot have subtraits. Each object type implicitly excludes every other object type, and indeed every type that is not its ancestor in the trait hierarchy.

If an object declaration has a parameter list, then there is an implicit top-level constructor function with the same name, and each invocation of the constructor function returns a fresh object whose ilk is the implicit trait of that same name; moreover, each parameter is also a field of the freshly constructed object. This constructor function can participate in an overload set with ordinary functions and functional methods of the same name. If an object declaration has no parameter list, then it represents a *singleton object* whose ilk is the implicit trait with the same name, and there is an implicit top-level variable with the same name whose value is that singleton object.

A dotted method declaration or definition is much as in the Java language, with slightly different syntax. A dotted method with the `getter` keyword is invoked by a field access, and a dotted method with the `setter` keyword is invoked by a field assignment. Such a getter or setter method must not be present in the same trait or object declaration as a field declaration or definition with the same name. Functional method declarations

Program ::= $\big($ *FunctionDefn* | *InterfaceDefn* | *ClassDefn* | *VariableDefn* $\big)^*$
InterfaceDefn ::= `interface` *InterfaceName* *Extends*$^?$ *DottedMethodDecl** `end`
DottedMethodDecl ::= *MethodName* `(` *ParameterList*$^?$ `)` : *Type*
DottedMethodDefn ::= *DottedMethodDecl* = *Expression*
Extends ::= `extends` { *InterfaceNameList* }
ClassDefn ::= `class` *ClassName* *Extends*$^?$ *Constructor* *ClassItem** `end`
Constructor ::= `constructor` *ClassName* `(` *ParameterList*$^?$ `)` = *Expression*
ClassItem ::= *VariableDefn* | *DottedMethodDefn*
FunctionDefn ::= *FunctionName* `(` *ParameterList*$^?$ `)` : *Type* = *Expression*
VariableDecl ::= `var`$^?$ *VariableName* : *Type*
VariableDefn ::= *VariableDecl* $\big($ = | := $\big)$ *Expression* | *SimpleBinding* = *Expression*
SimpleBinding ::= *VariableName* | `(` *SimpleBindingList*$^?$ `)`
SimpleBindingList ::= *SimpleBinding* $\big($ `,` *SimpleBinding* $\big)^*$
InterfaceNameList ::= *InterfaceName* $\big($ `,` *InterfaceName* $\big)^*$
TypeName ::= *InterfaceName* | *ClassName*
TypeList ::= *Type* $\big($ `,` *Type* $\big)^*$
Type ::= *TypeName* | `(` *TypeList*$^?$ `)` | *Type* $\rightarrow$ *Type*
ParameterList ::= *VariableDecl* $\big($ `,` *VariableDecl* $\big)^*$
Expression ::= *VariableName* | *Literal* | `self` | `(` *ExpressionList*$^?$ `)` | *TypeLiteral* | *TypeTest* | *Construction* | *Assignment* | *Conjunction* |
FunctionCall | *DottedMethodCall* | *FieldAccess* | *FieldAssignment* | *IfThenElseEnd* | *DoEnd*
ExpressionList ::= *Expression* $\big($ `,` *Expression* $\big)^*$
TypeLiteral ::= *Type* `. type`
TypeTest ::= *Expression* `instanceof` *Type*
Construction ::= `new` *ClassName* `(` *ExpressionList*$^?$ `)`
Assignment ::= *VariableName* := *Expression*
Conjunction ::= *Expression* `&&` *Expression*
FunctionCall ::= *FunctionName* `(` *ExpressionList*$^?$ `)`
DottedMethodCall ::= *Expression* `.` *MethodName* `(` *ExpressionList*$^?$ `)`
FieldAccess ::= `self .` *VariableName*
FieldAssignment ::= `self .` *VariableName* := *Expression*
IfThenElseEnd ::= `if` *Expression* `then` *StatementList* $\big($ `elif` *Expression* `then` *StatementList* $\big)^*$ $\big($ `else` *StatementList* $\big)^?$ `end`
DoEnd ::= `do` *StatementList* `end`
StatementList ::= $\big($ *FunctionDefn* | *VariableDefn* | *Expression* $\big)^*$ *Expression*

Figure 2. Grammar for target language

are distinguished syntactically from dotted method declarations by the presence of the keyword `self` in (exactly one place in) the parameter list.

Variable declarations specify a name and a type; if the keyword `var` is present, the variable is mutable. Variable definitions additionally specify an expression that provides an initial value. The symbol = indicates definition of an immutable variable; the symbol := (which is also used for assignment) indicates definition of a mutable variable (in which case `var` may optionally appear). As a convenience, within a statement list, a tuple of simple variable names may be bound to the respective elements of a tuple value.

A type may be the name of a trait or object, a parenthesized type, a tuple of types, or an arrow type. (The interesting issue of what is the precise type of an overloaded function name when used as a value is beyond the scope of this paper.)

Names of types are in a different namespace from the names of variables and functions; one may have a type and a function, or a type and a variable, with the same name. Definitions and declarations of local function and variables are not permitted to shadow outer definitions or declarations with the same name.

The Fortress parser rewrites operator syntax into function calls in a conventional manner, so there is no need to model operators separately in our source language here.

A function call may invoke either an ordinary function, a functional method, or an object constructor—but if the function call occurs within a trait or object declaration that has or inherits a definition or declaration of a dotted method with the same name, then it is considered to invoke that dotted method with `self` as the receiver. (This "convenience abbreviation" of a dotted method call—allowing the elision of a leading "`self.`" or "`this.`"—will be familiar to programmers who use the Java language [16, §15.11].)

Within an invocation of a dotted method or functional method, an immediate subexpression (that is, an argument expression or receiver expression) may have the form x `asif` T where x is an expression and T is a type; this is used to get the effect of the Java `super` construct, but specifies that T is the specific supertype of interest. If any subexpression of an invocation is an `asif` expression, then the subexpression before the dot (in a dotted method call) or corresponding to the `self` parameter of the statically most specific applicable declaration or definition, which must be a declaration or definition of a functional method (in a function call), must also be an `asif` expression.

Within a block (which for present purposes we may regard as identical to a *StatementList*, though the actual Fortress grammar draws a subtle distinction for the purpose of defining mutually recursive functions), functions and variables may be defined locally.

Identifiers may include any Unicode letter, digit or connecting punctuation, but not dollar signs '\$'.

4. The Target Language

Here we summarize the features of the target language that differ from those of the source language. Please refer to Figure 2.

A target-language program consists of a set of definitions of interfaces, classes, top-level functions, and top-level variables. There are no components or APIs. A target-language program is intended to be reminiscent of a Java compilation unit.

Interface declarations contain only dotted method declarations; there is no executable code in an interface. There are no functional methods. Methods cannot be overloaded, and `comprises` and `excludes` clauses are unnecessary.

Class declarations are similar to those in the Java language. Every class declaration has a `constructor` declaration; a constructor is invoked by a construction expression beginning with the keyword `new`. Class declarations may contain dotted method definitions, as well as declarations and definitions of fields.

For convenience, the target language includes the short-circuit (that is, conditional) Boolean AND operator `&&`, as well as the `instanceof` operator that can test whether the value of an expression belongs to a specified type. If a variable passes an `instanceof` check in the predicate part of an `if` statement, then the variable is regarded as having that type in the corresponding `then` branch of the `if` statement (thus no cast is required).

An expression of the form T`.type` is a *type literal* (analogous to an expression of the form T`.class` in the Java language); its value (which is of type Type) is a run-time reification of the type T. Such an object provides a dotted method $t.\mathit{isSubtype}(u)$ where u is also such an object, which returns true if the type represented by t is a subtype of (possibly the same as) the type represented by u, and otherwise returns false. (Compare the method `isAssignableFrom` of the Java class `class`, which provides the converse "is a supertype" test.)

Every value x provides the dotted method $x.\mathit{ilk}()$, which returns a reified type object that represents the ilk of the object, function, or tuple. The reified ilk of a function is of type $\mathrm{ArrowType}$, and provides additional methods such as $\mathit{argumentType}$ and $\mathit{resultType}$, which for a function type $D \to R$ return reified type objects for D and R, respectively. (Reified tuple types also have extra methods, but we will not need them for our exposition here.)

Field accesses and assignments are supported, but only within class declarations, and the expression before the dot must be `self`.

It is convenient to assume that target-language identifiers, in addition to Unicode letters, digits, and connecting punctuation, may also include a character '\$' that cannot appear in source-language identifiers, in order to allow construction of "mangled" names that cannot conflict with identifiers originating in source-language code. In what follows, it is assumed that any target-language identifier containing one or more '\$' characters and also one or more variables mentioned in the text actually represents an identifier constructed by substituting some string value (a name or the decimal representation of an integer value) for each variable. For example, if we say that $j = 3$ and w stands for the name *foo*, then $w\$j\$\mathit{dispatch}$ stands for the identifier $\mathit{foo}\$3\$\mathit{dispatch}$.

5. Details of the Translation

The big picture: trait declarations are rewritten into interface declarations, object definitions are rewritten into class definitions, fields are largely eliminated by using getter and setter methods, and functional method definitions are eliminated by translating them into top-level function definitions that serve as trampolines to corresponding dotted method definitions. Namespaces are eliminated by mangling names to include the name of the defining component.

The source program is first subjected to static analysis, including verification of syntactic well-formedness, static type checking, and verification that every overload set obeys the Meet Rule. Intermediate stages of the rewriting process do not maintain either well-formedness or type correctness according to either source-language or target-language rules, but the final result should be well-formed and type-correct according to target-language rules, and its execution behavior is intended to be identical to (and thereby *explain*) the execution behavior of the original source-language program.

We describe a series of rewriting steps as if each step were applied to all components and APIs before proceeding to the next step. However, up until the last step, "linking and final assembly" (see Section 5.11), each component may be processed entirely independently of all other components, relying only on information imported from APIs.

5.1 Create Object Constructors

If an object declaration with name U has a *ParameterList*, say:

$(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)$

then copy each variable declaration $p_j\colon \mathrm{T}_j$ from the parameter list into the object declaration itself, adding the `var` keyword to each one that does not already have a `var` keyword. Then add to the object declaration a constructor definition of the form:

```
constructor $\mathrm{U}(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n) =$ do
  self.$\mathit{field}\$p_1 := p_1$
  self.$\mathit{field}\$p_2 := p_2$
  ...
  self.$\mathit{field}\$p_n := p_n$
end
```

and delete the *ParameterList* and its surrounding parentheses from the object declaration. If the object declaration appears in a component, then add this top-level function definition to the component:

$\mathrm{U}(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{U} = \texttt{new}\ \mathrm{U}(p_1, p_2, \ldots, p_n)$

If an object declaration with name U has no *ParameterList*, then it is a singleton object, and it is implemented using the singleton pattern. Add to the object declaration this constructor:

`constructor` $\mathrm{U}() =$ `do end`   ⎈ It does nothing

and add to the component this top-level variable definition:

$\mathrm{U}\colon \mathrm{U} = \texttt{new}\ \mathrm{U}()$ ⎈ The unique instance of U

5.2 Eliminate Fields

Getter and setter methods behave exactly like ordinary dotted methods; the only difference is that they are eligible to be invoked by field access expressions and field assignment expressions, respectively. Therefore all getters, setters, field accesses, and field assignments can be eliminated by rewriting them to dotted methods and dotted method invocations.

In the resulting target language code, field declarations appear only within objects, and field accesses and assignments occur only locally within the definition of the object (and, indeed, only within the one or two dotted methods associated with that field by the rewriting). This transformation is conventional; for lack of space, we do not present the details.

5.3 Eliminate Functional Methods

Now for a much meatier transformation: this step eliminates functional methods by rewriting them into a combination of top-level functions (that provide the appropriate invocation interface) and dotted methods (that provide the appropriate inheritance structure).

For every component C, for every trait or object declared in that component, let U be the declared name of the trait or object, and consider each functional method declaration or definition appearing within U. It will have the general form:

$f(p_1\colon \mathrm{T}_1, \ldots, p_{j-1}\colon \mathrm{T}_{j-1}, \texttt{self}, p_{j+1}\colon \mathrm{T}_{j+1}, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T}$

or:

$f(p_1\colon \mathrm{T}_1, \ldots, p_{j-1}\colon \mathrm{T}_{j-1}, \texttt{self}, p_{j+1}\colon \mathrm{T}_{j+1}, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} = e$

where f is the name of the functional method; exactly one parameter position j in the declaration is occupied by the keyword `self`; $p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_n$ are names for its other parameters; $\mathrm{T}_1, \ldots, \mathrm{T}_{j-1}, \mathrm{T}_{j+1}, \ldots, \mathrm{T}_n$ are the respective types of those other parameters; and T is the result type. This functional method declaration or definition is replaced within U by a dotted method declaration or definition:

$f\$j(p_1\colon \mathrm{T}_1, \ldots, p_{j-1}\colon \mathrm{T}_{j-1}, p_{j+1}\colon \mathrm{T}_{j+1}, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T}$

or:

$f\$j(p_1\colon \mathrm{T}_1, \ldots, p_{j-1}\colon \mathrm{T}_{j-1}, p_{j+1}\colon \mathrm{T}_{j+1}, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} = e$

Note that the name of this dotted method is $f\$j$ (thus the information about the position of the `self` parameter has not been lost), and that the `self` parameter has been eliminated (becoming instead the receiver for the dotted method). In addition, a new top-level function definition is added to component C:

$f(p_1\colon \mathrm{T}_1, \ldots, p_{j-1}\colon \mathrm{T}_{j-1}, s\colon \mathrm{U}, p_{j+1}\colon \mathrm{T}_{j+1}, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} =$
$s.f\$j(p_1, p_2, \ldots, p_{j-1}, p_{j+1}, \ldots, p_n)$

where s is a freshly generated variable name appearing nowhere else in the program. In this manner, for any function call to a function named f for which the functional method was visible in the source language code, this newly created top-level function definition will be visible in the rewritten target language code, and will participate in overloading resolution with any other top-level functions named f within the same component.

Note that the top-level function definition created in this step will be further rewritten as described in Section 5.9, and the dotted method definition or declaration created in this step will be further rewritten as described in Section 5.10.

A related transformation is needed for functional method declarations appearing in an API. For every API A, for every trait declared in that API, let U be the declared name of the trait, and consider each functional method declaration or definition appearing within U; it will have the general form:

$f(p_1\colon \mathrm{T}_1, \ldots, p_{j-1}\colon \mathrm{T}_{j-1}, \mathtt{self}, p_{j+1}\colon \mathrm{T}_{j+1}, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T}$

The rewriting deletes it from U and adds a top-level function declaration to API A:

$f(p_1\colon \mathrm{T}_1, \ldots, p_{j-1}\colon \mathrm{T}_{j-1}, s\colon \mathrm{U}, p_{j+1}\colon \mathrm{T}_{j+1}, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T}$

where s is a freshly generated variable name appearing nowhere else in the program. In this manner, for any function call to a function named f for which the functional method has been imported from the API in the source language code, this newly created top-level function declaration will be imported instead in the rewritten target language code.

5.4 Rename Entities within Components

For every component C, for every top-level variable declaration or definition within C, change its name v to be $C\$\$v$, and change all references to v occurring within component C to be $C\$\$v$. (The double dollar sign avoids a naming conflict that might occur if C were the name "*field*". Mangling names consistently and unambiguously is tricky!)

For every component C, for every trait or object declaration within C, change its name U to be $C\$\mathrm{U}$, and change all references to U occurring within component C to be $C\$\mathrm{U}$.

In this way, entities declared with the same name in two different components are given distinct names.

Note that this step does *not* rename functions or methods. Necessary renaming of functions is taken care of in later steps (see Section 5.7 and Section 5.9).

5.5 Rewrite `import` and `export` Declarations

In every component and API, rewrite every `import` declaration with multiple import items into a series of `import` declarations each having a single import item. Next, rewrite each `import` declaration of the form: `import` $A.\{\, x \,\}$ into `import` $A.\{\, x \mapsto x \,\}$. These are, of course, merely "syntactic convenience" transformations.

Now, for every component C, process each resulting `import` declaration:

`import` $A.\{\, \mathit{old} \mapsto \mathit{new} \,\}$

(1) For every top-level function declaration in API A with the name old:

$\mathit{old}(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T}$

create in C a top-level function definition:

$\mathit{new}(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} = \$A\$\mathit{old}(p_1, p_2, \ldots, p_n)$

(The leading dollar sign '\$' indicates that the name $\$A\$\mathit{old}$ will need to be altered during the linking step described in Section 5.11.)

(2) For every trait declaration in API A with the name old, for throughout component C, change every reference to the trait new to $\$A\$\mathit{old}$. (After this point, names of traits and objects may be regarded as having global scope, transcending the boundaries of components and APIs.)

(3) For every `import` declaration in API A:

`import` $B.\{\, \mathit{old}' \mapsto \mathit{new}' \,\}$

such that $\mathit{new}' = \mathit{old}$, add to C a new `import` declaration:

`import` $B.\{\, \mathit{old}' \mapsto \mathit{new} \,\}$

and process it recursively.

After every `import` declaration in C has been processed (including the new ones generated in this step), delete every `import` declaration in component C.

5.6 Copy Inherited Methods and Assign Unique Integers

Note that the scope of a top-level function declaration or definition is the entire component or API that contains it, while the scope of a local function declaration or definition is a block.

For the sake of this step, it is necessary to process the components and APIs in such an order that any given API is processed before any component that imports it. It is permitted for APIs to import each other, but overall, the trait and object declarations in a program must be processed in such an order that any given trait is processed before any trait or object that extends it. We also assume that as each method is copied, it somehow remains tagged with the name of the trait in which it originally appeared.

Within each component or API, for each maximal set of function declarations and definitions that have the same name and whose scopes are co-extensive, assign a distinct integer to each declaration or definition in the set. (The scoping rules of the source language follow those of Fortress [2], and so these maximal sets form a partition of the set of all function declarations and definitions.)

Within each API, for each trait declaration U, copy into U each dotted method declaration inherited (and not overridden) by U from its immediate supertraits. Then for each maximal set of dotted method definitions in U that have the same name, assign a distinct integer to each definition in the set.

Within each component C, for each trait or object declaration U, there are three substeps:

(a) Consider each dotted method declaration or definition d inherited (and not overridden) by U from its immediate supertraits. If d is a declaration, say:

$m(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2 \colon, \ldots, p_n\colon \mathrm{T}_n)\colon T$

inherited from a trait V imported from an API A, and the integer assigned to d was i, then add to U a source-language dotted method definition with a target-language body that will not be further rewritten:

$m(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)\colon T =$
$\mathtt{self}\ .\$\$A\$V\$m\$i\$\mathit{entry}(p_1, p_2, \ldots, p_n)$

(The leading double dollar sign '\$\$' indicates that the name $\$\$A\$V\$m\$i\$\mathit{entry}$ will need to be altered during the linking step described in Section 5.11.)

(b) For each maximal set of dotted method definitions in U that have the same name, assign a distinct integer to each definition in the set.

(c) Revisit each declaration

$m(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)\colon T =$
  $\texttt{self}\ .\$\$A\$V\$m\$i\$\mathit{entry}(p_1, p_2, \ldots, p_n)$

introduced in substep (a), and let k be the integer that was assigned to it. Add to U a target-language dotted method definition that will not be further processed until the linking step described in Section 5.11:

```
$\$A\$V\$m\$i\$\mathit{dispatchHook}(\tau_0\colon \mathrm{Type},$
                                  $p_1\colon \mathrm{T}_1, \tau_1\colon \mathrm{Type},$
                                  $p_2\colon \mathrm{T}_2, \tau_2\colon \mathrm{Type},$
                                  $\ldots,$
                                  $p_n\colon \mathrm{T}_n, \tau_n\colon \mathrm{Type})\colon \mathrm{T} =$
  self $.C\$m\$k\$\mathit{dispatchHook}(\tau_0, p_1, \tau_1, p_2, \tau_2, \ldots, p_n, \tau_n)$
```

(This dotted method definition will overrride a definition that will be provided, as we shall see, by the component that implements API A.)

5.7 Rewrite Function Calls

For every component C, consider each function call expression that occurs within it. If the function call occurs within a trait or object declaration that has or inherits a definition or declaration of a dotted method with the same name, then the function call is rewritten into a dotted method call by adding `self.` to the front.

Otherwise, consider the statically most specific applicable function definition d, and let k be the integer that was assigned to that definition. Then there are two cases. If none of the argument expressions is an `asif` expression, then the function call has the form $f(a_1, a_2, \ldots, a_n)$. If d is a top-level definition, the function call is rewritten to:

$C\$f\$k\$\mathit{entry}(a_1, a_2, \ldots, a_n)$

(Note the incorporation of the name C of the containing component.) Otherwise it is rewritten to:

$f\$k\$\mathit{entry}(a_1, a_2, \ldots, a_n)$

In the other case, at least one of the arguments is an `asif` expression, for example:

$f(a_1, a_2\ \texttt{asif}\ \mathrm{T}_2, a_3\ \texttt{asif}\ \mathrm{T}_3, \ldots, a_n)$

Such a function call is rewritten to:

```
do
  $(z_1, z_2, z_3, \ldots, z_n) = (a_1, a_2, a_3, \ldots, a_n)$
  $w\big(z_1, z_1.\mathit{ilk}(), z_2, \mathrm{T}_2.\texttt{type}, z_3, \mathrm{T}_3.\texttt{type}, \ldots, z_n, z_n.\mathit{ilk}()\big)$
end
```

where the name w is actually $C\$f\$k\$\mathit{dispatch}$ (note the incorporation of the name C of the containing component) if d is a top-level definition, and otherwise is simply $f\$k\$\mathit{dispatch}$, and $z, z_1, z_2, \ldots, z_n$ are freshly generated variable names that occur nowhere else in the program.

Note that it would actually be correct always to use this second rewriting rule, even for calls that do not contain an `asif` expression. However, having no `asif` expression is the common case, and we choose to treat it specially. This structure also provides more flexibility for optimization (see Section 6.1).

5.8 Rewrite Dotted Method Calls

This section closely parallels the previous section.

For every component C, consider each dotted method call expression that occurs within it. Consider the statically most specific applicable dotted method definition d for that call, and let k be the integer that was assigned to that definition. Then there are two cases. If the receiver expression is not an `asif` expression, then none of the argument expressions can be an `asif` expression, so the dotted method call has the form $x.m(a_1, a_2, \ldots, a_n)$, and it is rewritten to:

$x.C\$m\$k\$\mathit{entry}(a_1, a_2, \ldots, a_n)$

In the other case, the receiver expression is an `asif` expression, and one or more arguments may also be `asif` expressions, for example:

$(y\ \texttt{asif}\ T).m(a_1, a_2\ \texttt{asif}\ \mathrm{T}_2, a_3\ \texttt{asif}\ \mathrm{T}_3, \ldots, a_n)$

Such a dotted method call is rewritten to:

```
do
  $(z, z_1, z_2, z_3, \ldots, z_n) = (y, a_1, a_2, a_3, \ldots, a_n)$
  $z.C\$m\$k\$\mathit{dispatch}\big(\mathrm{T}.\texttt{type}, z_1, z_1.\mathit{ilk}(), z_2, \mathrm{T}_2.\texttt{type},$
                              $z_3, \mathrm{T}_3.\texttt{type}, \ldots, z_n, z_n.\mathit{ilk}()\big)$
end
```

where $z, z_1, z_2, \ldots, z_n$ are freshly generated variable names that occur nowhere else in the program.

5.9 Rewrite Overloaded Functions

In every component and API, for every function definition d, let S_f be the maximal set of function declarations and definitions that have the same name f and whose scopes are co-extensive, let the integer that was assigned to d (see Section 5.6) be k, and suppose that d has n parameters:

$f(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2 :, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} = e$

Let S_f^k be the subset of S_f consisting of all definitions in S_f that are strictly more specific than definition d. Let U_f^k be the "upper fringe" of S_f^k, that is, the set of all definitions d' in S_f^k such that there is no definition d'' in S_f^k such that d' is strictly more specific than d''. Now let D_f^k be a "dispatch set," that is, any arbitrarily chosen set of function definitions such that $U_f^k \subseteq D_f^k \subseteq S_f^k$. Let q be the size of this dispatch set, and let $k_1, k_2, \ldots, k_q$ be the integers assigned to its elements (which may be ordered arbitrarily for the purpose of assigning indices $1, 2, \ldots, q$ to their respective integers). Then rewrite definition d into three functions in the same scope as follows:

$C\$f\$k\$\mathit{entry}(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} =$
  $C\$f\$k\$\mathit{dispatch}\big(p_1, p_1.\mathit{ilk}(), p_2, p_2.\mathit{ilk}(), \ldots, p_n, p_n.\mathit{ilk}()\big)$

```
$C\$f\$k\$\mathit{dispatch}(p_1\colon \mathrm{T}_1, \tau_1\colon \mathrm{Type},$
                  $p_2\colon \mathrm{T}_2, \tau_2\colon \mathrm{Type},$
                  $\ldots,$
                  $p_n\colon \mathrm{T}_n, \tau_n\colon \mathrm{Type})\colon \mathrm{T} =$
  if $C\$f\$k_1\$\mathit{applicable}(\tau_1, \tau_2, \ldots, \tau_n)$ then
    $C\$f\$k_1\$\mathit{dispatch}(p_1, \tau_1, p_2, \tau_2, \ldots, p_n, \tau_n)$
  elif $C\$f\$k_2\$\mathit{applicable}(\tau_1, \tau_2, \ldots, \tau_n)$ then
    $C\$f\$k_2\$\mathit{dispatch}(p_1, \tau_1, p_2, \tau_2, \ldots, p_n, \tau_n)$
  ...
  elif $C\$f\$k_q\$\mathit{applicable}(\tau_1, \tau_2, \ldots, \tau_n)$ then
    $C\$f\$k_q\$\mathit{dispatch}(p_1, \tau_1, p_2, \tau_2, \ldots, p_n, \tau_n)$
  else $C\$f\$k(p_1, p_2, \ldots, p_n)$ end
```

$C\$f\$k(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} = e$

where $\tau_1, \tau_2, \ldots, \tau_n$ are freshly generated variable names that occur nowhere else in the program. (A separate function definition $C\$f\$k$, identical to the original except its name, is retained, even though it is called from only one place, so that the expression e will not be duplicated if the function $C\$f\$k\$\mathit{dispatch}$ is inlined for optimization purposes; see Section 6.1.) However, if the dispatch set D_f^k is empty, then the second function is simply:

$$
\begin{aligned}
&C\$f\$k\$\mathit{dispatch}(p_1{:}\,\mathrm{T}_1, \tau_1{:}\,\mathrm{Type},\\
&\qquad p_2{:}\,\mathrm{T}_2, \tau_2{:}\,\mathrm{Type},\\
&\qquad \ldots,\\
&\qquad p_n{:}\,\mathrm{T}_n, \tau_n{:}\,\mathrm{Type}){:}\,\mathrm{T} =\\
&\quad C\$f\$k(p_1, p_2, \ldots, p_n)
\end{aligned}
$$

In addition, if there is at least one definition d' in S_f such that d is more specific than d', then create a fourth function in the same scope:

$$
\begin{aligned}
&C\$f\$k\$\mathit{applicable}(\tau_1{:}\,\mathrm{Type}, \tau_2{:}\,\mathrm{Type}, \ldots, \tau_n{:}\,\mathrm{Type}){:}\,\mathrm{Boolean} =\\
&\quad \tau_1.\mathit{isSubtype}(\mathrm{T}_1.\mathtt{type})\ \mathtt{\&\&}\\
&\quad \tau_2.\mathit{isSubtype}(\mathrm{T}_2.\mathtt{type})\ \mathtt{\&\&}\\
&\quad \ldots\ \mathtt{\&\&}\\
&\quad \tau_n.\mathit{isSubtype}(\mathrm{T}_n.\mathtt{type})
\end{aligned}
$$

5.10 Rewrite Overloaded Dotted Methods

This section is more complex than the previous section.

In every component, for every object declaration U, for every dotted method definition d in U, let S_m be the maximal set of dotted method definitions in U that have the same name m, let the integer that was assigned to d (see Section 5.6) be k, and suppose that d has n parameters:

$$m(p_1{:}\,\mathrm{T}_1, p_2{:}\,\mathrm{T}_2\,{:}, \ldots, p_n{:}\,\mathrm{T}_n){:}\,\mathrm{T} = e$$

Let S_m^k be the subset of S_m consisting of all definitions in S_m that are strictly more specific than definition d. Let U_m^k be the "upper fringe" of S_m^k, that is, the set of all definitions d' in S_m^k such that there is no definition d'' in S_m^k such that d' is strictly more specific than d''. Now let D_m^k be a "dispatch set," that is, any arbitrarily chosen set of dotted method definitions such that $U_m^k \subseteq D_m^k \subseteq S_m^k$. Let q be the size of this dispatch set, and let $k_1, k_2, \ldots, k_q$ be the integers assigned to its elements (which may be ordered arbitrarily for the purpose of assigning indices $1, 2, \ldots, q$ to their respective integers). Then rewrite definition d into three dotted method definitions within the same trait or object declaration as follows:

$$
\begin{aligned}
&C\$U\$m\$k\$\mathit{entry}(p_1{:}\,\mathrm{T}_1, p_2{:}\,\mathrm{T}_2, \ldots, p_n{:}\,\mathrm{T}_n){:}\,\mathrm{T} =\\
&\quad C\$U\$m\$k\$\mathit{dispatch}\big(\mathtt{self}.\mathit{ilk}(),\\
&\qquad p_1, p_1.\mathit{ilk}(), p_2, p_2.\mathit{ilk}(), \ldots, p_n, p_n.\mathit{ilk}()\big)
\end{aligned}
$$

$$
\begin{aligned}
&C\$U\$m\$k\$\mathit{dispatchHook}(\tau_0{:}\,\mathrm{Type},\\
&\qquad p_1{:}\,\mathrm{T}_1, \tau_1{:}\,\mathrm{Type},\\
&\qquad p_2{:}\,\mathrm{T}_2, \tau_2{:}\,\mathrm{Type},\\
&\qquad \ldots,\\
&\qquad p_n{:}\,\mathrm{T}_n, \tau_n{:}\,\mathrm{Type}){:}\,\mathrm{T} =\\
&\quad C\$U\$m\$k\$\mathit{dispatch}\big(\mathtt{self}.\mathit{ilk}(),\\
&\qquad p_1, p_1.\mathit{ilk}(), p_2, p_2.\mathit{ilk}(), \ldots, p_n, p_n.\mathit{ilk}()\big)
\end{aligned}
$$

$$
\begin{aligned}
&C\$U\$m\$k\$\mathit{dispatch}(\tau_0{:}\,\mathrm{Type},\\
&\qquad p_1{:}\,\mathrm{T}_1, \tau_1{:}\,\mathrm{Type},\\
&\qquad p_2{:}\,\mathrm{T}_2, \tau_2{:}\,\mathrm{Type},\\
&\qquad \ldots,\\
&\qquad p_n{:}\,\mathrm{T}_n, \tau_n{:}\,\mathrm{Type}){:}\,\mathrm{T} =\\
&\quad \mathtt{if\ self}.C\$U\$m\$k_1\$\mathit{applicable}(\tau_0, \tau_1, \tau_2, \ldots, \tau_n)\ \mathtt{then}\\
&\qquad \mathtt{self}.C\$U\$m\$k_1\$\mathit{dispatchHook}(\tau_0, p_1, \tau_1, p_2, \tau_2, \ldots, p_n, \tau_n)\\
&\quad \mathtt{elif\ self}.C\$U\$m\$k_2\$\mathit{applicable}(\tau_0, \tau_1, \tau_2, \ldots, \tau_n)\ \mathtt{then}\\
&\qquad \mathtt{self}.C\$U\$m\$k_2\$\mathit{dispatchHook}(\tau_0, p_1, \tau_1, p_2, \tau_2, \ldots, p_n, \tau_n)\\
&\quad \ldots\\
&\quad \mathtt{elif\ self}.C\$U\$m\$k_q\$\mathit{applicable}(\tau_0, \tau_1, \tau_2, \ldots, \tau_n)\ \mathtt{then}\\
&\qquad \mathtt{self}.C\$U\$m\$k_q\$\mathit{dispatchHook}(\tau_0, p_1, \tau_1, p_2, \tau_2, \ldots, p_n, \tau_n)\\
&\quad \mathtt{else\ self}.C\$U\$m\$k(p_1, p_2, \ldots, p_n)\ \mathtt{end}
\end{aligned}
$$

$$C\$U\$m\$k(p_1{:}\,\mathrm{T}_1, p_2{:}\,\mathrm{T}_2, \ldots, p_n{:}\,\mathrm{T}_n){:}\,\mathrm{T} = e$$

where $\tau_0, \tau_1, \tau_2, \ldots, \tau_n$ are freshly generated variable names that occur nowhere else in the program. (A separate method definition $C\$U\$m\$k\$$, identical to the original except for its name, is retained, even though it is called from only one place, so that the expression e will not be duplicated if the method $C\$U\$m\$k\$\mathit{dispatch}$ is inlined for optimization purposes; see Section 6.1.) However, if the dispatch set D_m^k is empty, then the second method is simply:

$$
\begin{aligned}
&C\$U\$m\$k\$\mathit{dispatch}(\tau_0{:}\,\mathrm{Type},\\
&\qquad p_1{:}\,\mathrm{T}_1, \tau_1{:}\,\mathrm{Type},\\
&\qquad p_2{:}\,\mathrm{T}_2, \tau_2{:}\,\mathrm{Type},\\
&\qquad \ldots,\\
&\qquad p_n{:}\,\mathrm{T}_n, \tau_n{:}\,\mathrm{Type}){:}\,\mathrm{T} =\\
&\quad \mathtt{self}.C\$U\$m\$k(p_1, p_2, \ldots, p_n)
\end{aligned}
$$

In addition, if there is at least one definition d' in S_m such that d is more specific than d', then create a fourth dotted method in the same trait or object declaration:

$$
\begin{aligned}
&C\$U\$m\$k\$\mathit{applicable}(\tau_0{:}\,\mathrm{Type}, \tau_1{:}\,\mathrm{Type}, \tau_2{:}\,\mathrm{Type}, \ldots,\\
&\qquad \tau_n{:}\,\mathrm{Type}){:}\,\mathrm{Boolean} =\\
&\quad \tau_0.\mathit{isSubtype}(\mathrm{W}.\mathtt{type})\ \mathtt{\&\&}\\
&\quad \tau_1.\mathit{isSubtype}(\mathrm{T}_1.\mathtt{type})\ \mathtt{\&\&}\\
&\quad \tau_2.\mathit{isSubtype}(\mathrm{T}_2.\mathtt{type})\ \mathtt{\&\&}\\
&\quad \ldots\ \mathtt{\&\&}\\
&\quad \tau_n.\mathit{isSubtype}(\mathrm{T}_n.\mathtt{type})
\end{aligned}
$$

where W is the name of the trait or object in which the definition d originally appeared.

5.11 Linking and Final Assembly

(Up to this point, all transformations on an API or component have required only information in that API or component plus information imported from APIs, and therefore correspond to a process of separate compilation of each component. This step corresponds to a linking step that binds multiple components into a single program by renaming certain entities and copying inherited methods. In practice, within a virtual machine implementation, one would simply copy method pointers rather than duplicating code.)

For every component C, for every name within C of the form "$\$A\$\ldots$", identify the *implementing component* D for A (the (necessarily unique) component that contains the statement `export` A), and alter the name "$\$A\$\ldots$" to be "$D\$\ldots$" instead; and for every name within C of the form "$\$\$A\$V\$m\$i\$\ldots$", identify the implementing component D for A, then identify the method declaration for m within trait V within API A that has integer i assigned to it, then identify the method definition for m within trait V within component D that has the same signature, and let k be the integer assigned to it, then alter the name to be "$D\$m\$k\$\ldots$" instead.

All APIs may now be discarded.

For every object declaration that appears in a component, add into the object declaration a copy of every dotted method definition that it inherits and is not overridden, even if that inherited definition is in another component; note that overriding should occur only for methods whose names end in "$\$\mathit{dispatchHook}$". Change the keyword `object` to `class`.

Now, for every trait declaration that appears in a component, delete the body of every dotted method definition in the trait, thus converting it to an abstract dotted method declaration. Note that the trait declaration now contains no executable code, only method declarations. Change the keyword `trait` to `interface`.

Once this has been done for all components in the program, delete every `export` declaration in every component.

The final program in the target language consists of the concatenation of the residual contents of all components, with the leading "`component` *ComponentName*" and trailing "`end`" of each component removed.

6. Variations and Optimizations

The rewriting process described in Section 5 is intended to be general and reasonably easy to understand. In practice some useful variations and optimizations are available for use when appropriate.

6.1 Inlining

The intermediary *dispatch* and *applicable* functions and methods introduced by the rewritings described in Section 5.9 and Section 5.10 may often be profitably inlined. To work a specific example, consider these definitions for an overloaded local function named *which*, with their assigned integers shown in comments:

```
$which(x\colon \text{Object}, y\colon \text{Object})\colon \text{String} =$ “neither”    ⊛ 1
$which(x\colon \text{Object}, y\colon \text{String})\colon \text{String} =$ “second”     ⊛ 2
$which(x\colon \text{String}, y\colon \text{Object})\colon \text{String} =$ “first”      ⊛ 3
$which(x\colon \text{String}, y\colon \text{String})\colon \text{String} =$ “both”       ⊛ 4
```

The process described in Section 5.9 will (assuming a minimal choice of dispatch set) rewrite the first definition to:

```
$which\$1\$entry(x\colon \text{Object}, y\colon \text{Object})\colon \text{String} =$
  $which\$1\$dispatch\big(x, x.ilk(), y, y.ilk()\big)$

$which\$1\$dispatch(x\colon \text{Object}, \tau_1\colon \text{Type},$
                    $y\colon \text{Object}, \tau_2\colon \text{Type})\colon \text{String} =$
  if $which\$2\$applicable(\tau_1, \tau_2)$ then
    $which\$2\$dispatch(x, \tau_1, y, \tau_2)$
  elif $which\$3\$applicable(\tau_1, \tau_2)$ then
    $which\$3\$dispatch(x, \tau_1, y, \tau_2)$
  else $which\$1(x, y)$ end

$which\$1\$applicable(\tau_1\colon \text{Type}, \tau_2\colon \text{Type})\colon \text{Boolean} =$
  $\tau_1.isSubtype(\text{Object}.$type$)$ && $\tau_2.isSubtype(\text{Object}.$type$)$

$which\$1(x\colon \text{Object}, y\colon \text{Object})\colon \text{String} =$ “neither”
```

and similarly for the other three definitions. If we assume that an optimizer can rewrite $x.ilk().isSubtype(\mathrm{T}.$`type`$)$ to simply x `instanceof` T, then after inlining and elimination of redundant computations, one may obtain:

```
$which\$1\$entry(x\colon \text{Object}, y\colon \text{Object})\colon \text{String} =$
  if $y$ instanceof String then
    if $x$ instanceof String then $which\$4(x, y)$
    else $which\$2(x, y)$ end
  elif $x$ instanceof String then $which\$3(x, y)$
  else $which\$1(x, y)$ end
```

which is a not unreasonable implementation of the dispatch process as an executable discrimination tree. This is exactly the sort of thing one would expect from, say, a competent Java compiler [17]. Further transformations are of course possible, perhaps based on dynamic profiling information.

6.2 Exploiting Single Inheritance

The rewriting step described in Section 5.11 copies *all* inherited dotted method definitions from traits down into objects. If the target environment for program execution supports single inheritance, then it may be advantageous to exploit it by choosing a subset of the edges of the type hierarchy so as to form a tree providing exactly one path from every object type to the root of the type hierarchy. Every Fortress trait can then be represented by a Java interface and an associated Java class that extends it. (Note that the Fortress trait named Object is *not* represented by the Java class `Object`, but rather by a Java interface named something like `Fortress$Object`, which in turn extends a Java interface named something like `Fortress$Any`.) The Fortress type hierarchy is then represented by `extends` relationships among these Java interfaces. The Java class associated with a Fortress object type is used to hold the (rewritten) dotted method definitions copied down from supertraits that are *not* accessible through the chosen tree; this class also extends the Java class associated with the Fortress supertrait that *is* accessible through the chosen tree, and therefore inherits dotted methods from it.

If this representation strategy is used, then the rewriting described in Section 5.11 (delete the body of every dotted method definition in the trait) should not be used.

```
component Lib
export Library
$p(x\colon \text{Object})\colon \text{String} =$ “/” $x$ “/”
$p(x\colon \mathbb{Z})\colon \text{String} =$ “#” $x$ “#”
end Lib

api Library
$p(x\colon \text{Object})\colon \text{String}$
end Library

component User
import Library.{ $p$ }
export Executable
object Foo($x\colon$ String)
  $p($self$)\colon \text{String} =$ “<” $x$ “>”
end
$p(x\colon \mathbb{Z})\colon \text{String} =$ “[” $x$ “]”
$run()\colon () =$ do
  $q(z\colon \text{Object})\colon \text{String} = p(z)$
  $println\big(p(\text{Foo(“hello”)})$ “ versus ” $q(\text{Foo(“hello”)})\big)$
  $println\big(p(17)$ “ versus ” $q(17)\big)$
  $println\big(p(6.375)$ “ versus ” $q(6.375)\big)$
end
end User
```

Figure 3. Example code in source language

7. Big Example

Two example components and an example API are shown in Figure 3. For the purposes of this example we extend the source and target languages to include string concatenation through expression juxtaposition, rather than translating such operations into method calls. When the *run* function is invoked, the program prints:

```
<hello> versus <hello>
[17] versus [17]
/6.375/ versus /6.375/
```

One point of this example is that the printed results are the same when p is called either directly or (through q) indirectly, even though the dispatch is achieved statically when p is called directly but dynamically when called through q (because the parameter z of function q has type Object).

Another point is that the example contains two definitions of p with parameter type $\mathbb{Z}$. If the underlying implementation were to lump all the definitions of p into a single CLOS-style “generic function” then the definitions would not form a meet-bounded lattice and the call $p(17)$ would be ambiguous; but the fact that two definitions are in a separate component Lib and only one is exported means that only three definitions are visible within component User, and they do form a meet-bounded lattice.

Space does not permit exhibiting snapshots of the code after each rewriting step. Figure 4 shows what the code looks like after integers have been assigned to function and method definitions (see Section 5.6). Figure 5 shows the final result, after all rewriting steps plus a certain amount of procedure inlining to reduce clutter. Of particular interest are the two functions $Lib\$p\$1\$entry$ and

```
component Lib
export Library
$p(x\colon \text{Object})\colon \text{String} = \text{"/"}\ x\ \text{"/"}$                    ❋ 1
$p(x\colon \mathbb{Z})\colon \text{String} = \text{"\#"}\ x\ \text{"\#"}$                         ❋ 2
end Lib

component User
export Executable
$p(x\colon \text{Object}) = \$\mathit{Library}\$p(x)$                      ❋ 1
class User$Foo
  var $\mathit{field}\$x\colon \text{String}$
  constructor $\text{Foo}(x\colon \text{String})$ = do self.$\mathit{field}\$x := x$ end
  $p\$1()\colon \text{String} = \text{"<"}$ self.$\mathit{field}\$x$ ">"                ❋ 1
end
$\mathit{Foo}(x\colon \text{String})\colon \text{User\$Foo}$ = new $\mathit{User}\$\mathit{Foo}(x)$          ❋ 1
$p(s\colon \text{Foo})\colon \text{String} = s.p\$1()$                            ❋ 2
$p(x\colon \mathbb{Z})\colon \text{String} = \text{"["}\ x\ \text{"]"}$                           ❋ 3
$\mathit{run}()\colon () =$ do                                                  ❋ 1
  $q(z\colon \text{Object})\colon \text{String} = p(z)$                            ❋ 1
  $\mathit{println}\big(p(\text{Foo}(\text{"hello"}))\ \text{" versus "}\ q(\text{Foo}(\text{"hello"}))\big)$
  $\mathit{println}\big(p(17)\ \text{" versus "}\ q(17)\big)$
  $\mathit{println}\big(p(6.375)\ \text{" versus "}\ q(6.375)\big)$
end
end User
```

Figure 4. Example code, partially rewritten

```
$\mathit{Lib}\$p\$1\$\mathit{entry}(x\colon \text{Object})\colon \text{String} =$
  if $x$ instanceof $\mathbb{Z}$ then $\mathit{Lib}\$p\$2(x)$
  else $\mathit{Lib}\$p\$1(x)$ end
$\mathit{Lib}\$p\$1(x\colon \text{Object})\colon \text{String} = \text{"/"}\ x\ \text{"/"}$
$\mathit{Lib}\$p\$2(x\colon \mathbb{Z})\colon \text{String} = \text{"\#"}\ x\ \text{"\#"}$

$\mathit{User}\$p\$1\$\mathit{entry}(x\colon \text{Object}) =$
  if $x$ instanceof User$Foo then $\mathit{User}\$p\$2(x)$
  elif $x$ instanceof $\mathbb{Z}$ then $\mathit{User}\$p\$3(x)$
  else $\mathit{Lib}\$p\$1\$\mathit{entry}(x)$ end
class User$Foo
  var $\mathit{field}\$x\colon \text{String}$
  constructor $\text{Foo}(x\colon \text{String})$ = do self.$\mathit{field}\$x := x$ end
  $p\$1\$1()\colon \text{String} = \text{"<"}$ self.$\mathit{field}\$x$ ">"
end
$\mathit{User}\$\mathit{Foo}\$1(x\colon \text{String})$ = new User$Foo($x$)
$\mathit{User}\$p\$2(s\colon \text{User\$Foo})\colon \text{String} = s.p\$1\$1()$
$\mathit{User}\$p\$3(x\colon \mathbb{Z})\colon \text{String} = \text{"["}\ x\ \text{"]"}$
$\mathit{User}\$\mathit{run}\$1()\colon () =$ do
  $q\$1(z\colon \text{Object})\colon \text{String} = \mathit{User}\$p\$1\$\mathit{entry}(z)$
  $\mathit{println}\big(\mathit{User}\$p\$2(\mathit{User}\$\mathit{Foo}\$1(\text{"hello"}))\ \text{" versus "}$
          $q\$1(\mathit{User}\$\mathit{Foo}\$1(\text{"hello"}))\big)$
  $\mathit{println}\big(\mathit{User}\$p\$3(17)\ \text{" versus "}\ q\$1(17)\big)$
  $\mathit{println}\big(\mathit{Lib}\$p\$1\$\mathit{entry}(17)\ \text{" versus "}\ q\$1(6.375)\big)$
end
```

Figure 5. Example code, completely rewritten

$\mathit{User}\$p\$1\$\mathit{entry}$ (into which the corresponding dispatch functions have been inlined).

Note that if the component User did not contain the definition:

$p(x\colon \mathbb{Z})\colon \text{String} = \text{"["}\ x\ \text{"]"}$

then the program would still be correct, and would instead print:

```
<hello> versus <hello>
#17# versus #17#
/6.375/ versus /6.375/
```

For argument 17 (of type $\mathbb{Z}$), the static or dynamic dispatch within component User would transfer control to the first definition of p in component Lib, which would then dynamically dispatch to the second definition in component Lib. While this second definition is not imported into component User and therefore is not directly visible, it does form a part of the *implementation* of the overloaded function p within component Lib.

8. Extension to Parametrically Polymorphic Functions and Methods

Now suppose that we wish for our source language to include functions and methods with explicit static type parameters. We might, for example, write a function *both* that takes two values of type T and a predicate on values of type T, and returns *true* iff the predicate is true of both values:

$\mathit{both}[\![T]\!](a\colon \text{T}, b\colon \text{T}, p\colon \text{T} \to \text{Boolean})\colon \text{Boolean} = p(a) \land p(b)$

Determining whether sets of such polymorphic functions form valid overloadings is an interesting problem that is beyond the scope of this paper but is addressed by a companion paper [1]. But once an overload set has passed the static type-checking process, we know that the signatures of the functions form a meet-bounded lattice (according to the rule that function f_1 is strictly more specific than function f_2 if and only if f_2 is applicable to every set of arguments to which f_1 is applicable and there is some set of arguments to which f_2 is applicable and f_1 is not). Therefore exactly the same rewriting schema described in Section 5.9 applies, and similarly for dotted methods. All that is necessary is to provide an appropriate implementation of the *applicable* functions or methods. In this example, $\mathit{both}\$1\$\mathit{applicable}$ needs to extract the respective ilks A, B, and P from the arguments a, b, and p, verify that the ilk of p is an arrow type, and extract the argument type C from that arrow type. If this definition of *both* is to be applicable, the type T must be at least $A \cup B$, and can be at most C. The necessary test is therefore $(A \cup B).\mathit{isSubtype}(C)$, which can be further simplified to $A.\mathit{isSubtype}(C) \land B.\mathit{isSubtype}(C)$. Putting it all together, we have:

```
$\mathit{both}\$1\$\mathit{applicable}(\tau_1\colon \text{Type}, \tau_2\colon \text{Type}, \tau_3\colon \text{Type})\colon \text{Boolean} =$
  if $\tau_3$ instanceof ArrowType then
    $\tau_4 = \tau_3.\mathit{argumentType}()$
    $\big(\tau_1.\mathit{isSubtype}(\tau_4)$ && $\tau_2.\mathit{isSubtype}(\tau_4)$ &&
     $\tau_3.\mathit{resultType}().\mathit{isSubtype}(\text{Boolean.type})\big)$
  else $\mathit{false}$ end
```

If this example were modified to bound the type parameter:

$\mathit{both}[\![T \texttt{ extends } \mathbb{Q}]\!](a\colon \text{T}, b\colon \text{T}, p\colon \text{T} \to \text{Boolean})\colon \text{Boolean} =$
$p(a) \land p(b)$

then one might imagine that one must first find an actual type X for T that is a subtype of $\mathbb{Q}$, and then decide whether the ilks of a, b, and p are subtypes of X, X, and X $\to$ Boolean, respectively. But in fact it suffices to test the extracted ilks from covariant positions against the bound:

```
$\mathit{both}\$1\$\mathit{applicable}(\tau_1\colon \text{Type}, \tau_2\colon \text{Type}, \tau_3\colon \text{Type})\colon \text{Boolean} =$
  if $\tau_3$ instanceof ArrowType then
    $\tau_4 = \tau_3.\mathit{argumentType}()$
    $\big(\tau_1.\mathit{isSubtype}(\mathbb{Q}.\text{type})$ && $\tau_1.\mathit{isSubtype}(\tau_4)$ &&
     $\tau_2.\mathit{isSubtype}(\mathbb{Q}.\text{type})$ && $\tau_2.\mathit{isSubtype}(\tau_4)$ &&
     $\tau_3.\mathit{resultType}().\mathit{isSubtype}(\text{Boolean.type})\big)$
  else $\mathit{false}$ end
```

In general, determining applicability involves deciding whether a set of type constraints is satisfiable. But note that the testing of applicability at run time is substantially less complicated than one might think when one considers that (a) the methods have already been statically checked for type correctness, and (b) it is not necessary to explicitly construct or identify types for the type parameters, but only to prove their existence.

9. Related Work

We have already made comparisons in passing between aspects of our work and features of CLOS [14, 4, 23, 15], Java [16], MultiJava [11, 12], Scala [20], and especially the $\lambda\&$-calculus of Castagna, Ghelli, and Longo [6], which is earliest work we know of to recommend symmetric multimethod dispatch as a central feature of an object-oriented language design.

Of all the related languages we have surveyed, however, the one closest in design and intent to what we present here is Dubious [19], which provides symmetric dynamic multimethod dispatch while allowing a program to be divided into modules that can be separately type-checked statically. This work differs from Dubious in these respects: Dubious is a classless (prototype-oriented) object system, whereas Fortress traits are classes in this sense; Dubious has only explicitly declared objects, whereas this work supports dynamically created (allocated) objects and state; and importing a Dubious module is an all-or-nothing proposition (though the cited paper does sketch a possible way to introduce a `private` keyword to shield some objects in a module), whereas Fortress APIs and import declarations allow fine-grained selective export and import of only parts of a component—in particular, it is possible to export only selected methods of a trait, rather than all methods. Two other languages from the same research group that are similarly closely related to the present work are Cecil [9, 8, 10] and EML [18].

10. Conclusions and Future Work

Namespace control in object-oriented languages is tricky. On the one hand, there is a desire to inherit method definitions implicitly and to be able to override and overload them. On the other hand, there is a desire to be able to control access to specific methods by controlling their names. On the third hand, if a trait T is made public but a method m is kept private, then third parties extending T should not be prevented from using the name m as a method name, and the two methods (or sets of methods) should not interact. The framework we present solves this problem effectively.

We agree with Millstein and Chambers that symmetric multimethod dispatch is far preferable to other disciplines: "[T]he symmetric multimethod dispatching semantics … is more natural and less error-prone, since it reports potential ambiguities rather than silently resolving them" [19, §4.1] Moreover, the Meet Rule has beneficial consequences not only for aiding programmer understanding but for enabling separate compilation and a distributed (rather than monolithic) implementation of multimethod dispatch that is amenable to optimization.

Functional methods are an effective approach to solving the binary method problem. The advantage over dotted methods is that any argument position may serve as the receiver; the advantage over ordinary multimethods (functions) is that a trait may declare an abstract functional method, thereby obligating any object that extends the trait to implement it. We have shown how to implement this feature in terms of functions and dotted methods in a manner that interacts well with namespace control.

In the future we hope to extend this framework to parametrically polymorphic traits and objects.

References

[1] Type-checking modular multiple dispatch with parametric polymorphism and multiple inheritance. Submitted to POPL 2011.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., S. Tobin-Hochstadt, J. Dias, C. Eastlund, C. Flood, Y. Lev, C. McCosh, J. D. Nielsen, and D. Smith. The Fortress language specfication, version 1.0. Technical report, Sun Microsystems Laboratories, Burlington, MA, Mar. 2008.

[3] E. Allen, J. J. Hallett, V. Luchangco, S. Ryu, and G. Steele. Modular multiple dispatch with multiple inheritance. In *SAC '07: Proc. 2007 ACM Symposium on Applied Computing*, pages 1117–1121, New York, NY, USA, 2007. ACM.

[4] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System specification. *SIGPLAN Notices*, 23(special issue):1–142, 1988.

[5] K. Bruce, L. Cardelli, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[6] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. In *LFP '92: Proc. 1992 ACM Conference on LISP and Functional Programming*, pages 182–192, New York, NY, USA, 1992. ACM.

[7] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[8] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, pages 33–56, London, UK, 1992. Springer-Verlag.

[9] C. Chambers. The Cecil language: Specification and rationale. Technical Report UW-CSE-93-03-05, University of Washington, Seattle, WA, Mar. 1993.

[10] C. Chambers et al. The Cecil language: Specification and rationale. Technical report, University of Washington, Seattle, WA, Feb. 2004.

[11] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00: Proc. 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, New York, NY, USA, 2000. ACM.

[12] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.*, 28(3):517–575, 2006.

[13] ECMA. *C# Language Specification*, Dec. 2001. Standard ECMA-334.

[14] R. P. Gabriel and L. DeMichiel. An overview of the Common Lisp Object System. In *Proceedings of the European Conference on Object-Oriented Programming ECOOP 1987*, pages 151–170, Paris, July 1987. Proceedings published by Springer-Verlag as LNCS 276.

[15] R. P. Gabriel, J. L. White, and D. G. Bobrow. CLOS: Integrating object-oriented and functional programming. *Comm. ACM*, 34(9):29–38, 1991.

[16] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[17] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot$^{\mathrm{TM}}$client compiler for Java 6. *ACM Trans. Arch. Code Optim.*, 5(1):1–32, 2008.

[18] T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Trans. Program. Lang. Syst.*, 26(5):836–889, 2004.

[19] T. Millstein and C. Chambers. Modular statically typed multimethods. *Information and Computation*, 175(1):76–118, 2002.

[20] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA '05: Proc. 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 41–57, New York, NY, USA, 2005. ACM.

[21] V. A. Saraswat, V. Sarkar, and C. von Praun. X10: Concurrent programming for modern architectures. In *PPoPP '07: Proc. 12th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 271–271, New York, NY, USA, 2007. ACM.

[22] G. L. Steele Jr. Growing a language. *Higher Order Symbol. Comput.*, 12(3):221–236, 1999.

[23] G. L. Steele Jr., S. E. Fahlman, R. P. Gabriel, D. A. Moon, D. L. Weinreb, D. G. Bobrow, L. G. DeMichiel, S. E. Keene, G. Kiczales, C. Perdue, K. M. Pitman, R. C. Waters, and J. L. White. *Common Lisp: The Language (Second Edition)*. Digital Press, Bedford, MA, 1990.

[24] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 1986.