Metaprogramming with Traits

John Reppy and Aaron Turon

University of Chicago
{jhr,adrassi}@cs.uchicago.edu

Abstract. In many domains, classes have highly regular internal structure. For example, so-called business objects often contain boilerplate code for mapping database fields to class members. The boilerplate code must be repeated per-field for every class, because existing mechanisms for constructing classes do not provide a way to capture and reuse such member-level structure. As a result, programmers often resort to *ad hoc* code generation. This paper presents a lightweight mechanism for specifying and reusing member-level structure in Java programs. The proposal is based on a modest extension to traits that we have termed *trait-based metaprogramming*. Although the semantics of the mechanism are straightforward, its type theory is difficult to reconcile with nominal subtyping. We achieve reconciliation by introducing a hybrid structural/nominal type system that extends Java's type system. The paper includes a formal calculus defined by translation to Featherweight Generic Java.

1 Introduction

In mainstream object-oriented languages, programming amounts to class creation. While a programmer may write classes from scratch, good style dictates that existing code be used when possible. Several mechanisms exist to aid the programmer in this endeavor: inheritance combines existing classes with extensions or modifications; mixins and traits capture such extensions, allowing them to be reused; and generic classes are instantiated with type parameters to produce specialized classes. Each of these mechanisms allows programmers to capture and reuse useful structure at the level of classes, but they provide limited support for capturing structure at the level of class members.

In many domains, classes have highly regular *internal* structure. As a simple example, consider a thread-safe class in which all methods obtain a single lock before executing. Manually writing this boilerplate code results in clutter and rigidity: the locking strategy cannot easily be changed after the fact. In Java, thread-safe methods were considered important enough to warrant the **synchronized** keyword, but adding keywords is a kind of magic that only the language designer, not the language user, can perform. In this paper, we propose a mechanism that allows programmers to capture, reuse, and modify such *member-level patterns* in a coherent way.

The **synchronized** pattern consists of behavior common to otherwise unrelated members of a class. Another common member-level pattern is when a class

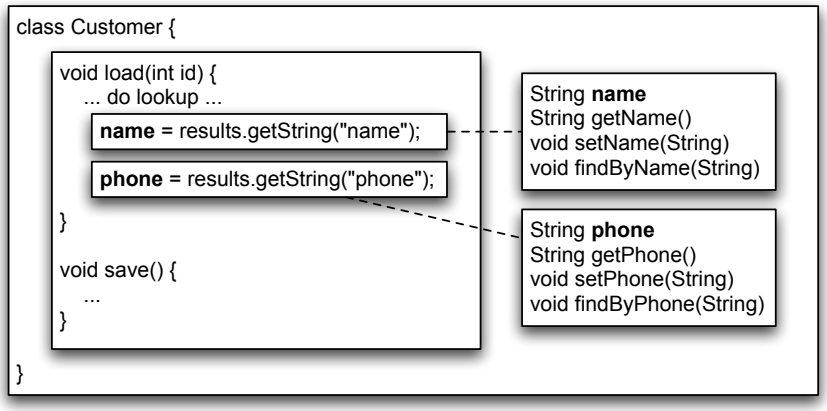


Fig. 1. A more complex member-level pattern

contains collections of similar members that are intended to match a domain model. For example, consider a `Customer` class that provides access to a customer table in a database. For each field present in the table, the `Customer` class will contain a cluster of members: for the `name` field, the `Customer` class might contain an instance variable `name` and methods `getName`, `setName`, and `findByName`. Moreover, the class will provide `load` and `save` methods that load and store the class's instance variables. This situation is shown diagrammatically in Figure 1. While additional behavior may be needed for particular fields, it is desirable to abstract the common structure and implementation; once defined, the abstraction answers the question "what does it mean for a class to provide access to a database field?" We show how this can be done with our mechanism at the end of Section 3.

Our proposal is based on a modest extension to *traits* [9] that allows programmers to write *trait functions*, which are parameterized by member names. Trait functions are applied at compile time to build classes, supporting what we term *trait-based metaprogramming*. In describing our mechanism as a form of metaprogramming, we mean that (1) it operates entirely at compile-time and (2) it allows both *generation* and *introspection* of code.[1] There are many frameworks available for metaprogramming; our proposal's strength is its singular focus on member-level patterns. We believe that the territory between classes and individual class members is a fruitful place to do metaprogramming, and by focusing our efforts there, we are able to provide a succinct mechanism with good guarantees about the generated code. A detailed discussion of related work is given in Section 5.

The language design is presented informally in Section 3. In Section 4 we model our mechanism as an extension to Featherweight Generic Java (FGJ), giv-

[1] This paper focuses on generation; we discuss introspection in a technical report [28].

ing our semantics as a translation to FGJ. While the translation is very simple, its type theory is difficult to reconcile with nominal subtyping because abstraction over member labels is allowed. We achieve reconciliation by introducing a hybrid structural/nominal type system that extends Java's type system. The type system is not strongly tied to our broader proposal, and we hope that the ideas will find broad application in metaprogramming systems for nominally-subtyped languages, a possibility we discuss in Section 5.2.

2 Background

Traits were originally introduced by Schärli *et al.* in the setting of Smalltalk [9] as a mechanism for sharing common method definitions between classes. In their proposal, a trait is simply a collection of named methods. These methods cannot directly reference instance variables; instead, they must be "*pure behavior*." The methods defined in a trait are called the *provided methods*, while any methods that are referenced, but not provided, are called *required methods*. An important property of traits is that while they help structure the implementation of classes, they do not affect the inheritance hierarchy. In particular, traits are distinguished from mixins [5] because they can be composed without the use of inheritance.[2] Traits can be formed by definition (*i.e.*, listing a collection of method definitions) or by using one of several trait operations:

Symmetric sum merges two disjoint traits to create a new trait. [3]

Override forms a new trait by layering additional methods over an existing trait. This operation is an asymmetric sum. When one of the new methods has the same name as a method in the original trait, the override operation replaces the original method.

Alias creates a new trait by adding a new name for an existing method. This operation is *not* renaming, in that it does not replace references to the old name with the new one.

Exclusion forms a new trait by removing a method from an existing trait. Combining the alias and exclusion operations yields a renaming operation, although the renaming is shallow.

The other important operation on traits is *inlining*, the mechanism whereby traits are integrated with classes. This operation merges a class `C`, a trait, and additional fields and methods to form a new subclass of `C`. Often, the additional methods, called *glue methods* [9], provide access to the newly added fields. The glue methods, plus the methods inherited from `C`, provide the required methods of the trait. An important aspect of traits is that the methods of a trait are only loosely coupled; they can be removed and replaced by other implementations.

[2] Bracha's *Jigsaw* [4], one of the first formal presentations of mixins, supports a similar notion of composition, but most other constructs under the name "mixin" do not.

[3] Smalltalk traits allow name conflicts, but replace the conflicting methods with a special method body **conflict** that triggers a run-time error if evaluated.

Traits provide a lightweight alternative to multiple inheritance, and they have been the focus of much recent interest, including formal calculi [11, 20] and other language designs for traits [24, 27, 23, 14]. While the details of these various mechanisms vary, they all share a focus on sharing common method implementations across unrelated classes. Our design shifts the focus toward sharing member-level patterns that can occur within a single class.

3 A design for trait-based metaprogramming

We present our design in the setting of Java, though there is little that is Java-specific. Like other language designs that incorporate traits, a trait in our language has a collection of members it *provides* and a collection of members it *requires*. What is new in our design is that traits may be parameterized over the names and types of these members: our traits are really *trait functions*. The basic form of a trait is as follows:

trait *trait-name* `(`*member-name parameters*`,` *type parameters*`,` *value parameters*`)`
  requires `{` *requirements* `}`
  provides `{` *member definitions* `}`

Note that traits may be parameterized over values, such as constant values that vary between instances of a member-level pattern. Member-name parameters are prefixed with "`$`" so that member-name variables never shadow actual member names; in our experience, having a clear distinction between `obj.foo` and `obj.$foo` makes trait code easier to understand.

The **requires** and **provides** sections also differ from previous designs. In addition to giving the signatures of required class members, the requirements section is also used to place constraints on type parameters, as illustrated in the `DelegateT` example near the end of this section. Another departure in our design is that the **provides** section can contain field declarations. When such declarations are inlined in a class, the class's constructors are responsible for initializing them. Traits are inlined using the **use** construct, which is syntactically just another form of member definition. Since traits are actually functions, the **use** construct applies the trait function to its arguments and inlines the resulting member definitions. As shown below, the **provides** section of a trait can also have **use** declarations, which is how traits are composed. Conflicting method or field declarations within the body of a trait or class, whether defined directly or inlined from traits, are rejected by the type system.

3.1 Some illustrative examples

In the remainder of this section, we present a series of examples that illustrate our mechanism and the kinds of patterns it captures. We begin by with the notion of a "property" — a field along with getter and setter methods. In this example, the variables `$f`, `$g`, and `$s` range over field and method names, while the variable `T` ranges over types. The access modifiers **public** and **private** determine the visibility the members will have after they are inlined into a class:

```
trait PropT ($f, $g, $s, T)
  provides {
    private T $f;
    public void $s (T x) { $f = x; }
    public T $g () { return $f; }
  }
```

We can use `PropT` to define a 2D point class by "*using*" it twice with different member names:

```
class Point2 {
  use PropT (x, getX, setX, int);
  use PropT (y, getY, setY, int);
  Point2 () { x = 0; y = 0; }
}
```

Note also that the `Point2` constructor initializes the fields introduced by the traits.

Next, we revisit the **synchronized** example from Section 1:

```
trait SyncT ($op, R, A...)
  requires {
    ThisType implements {
      Mutex lock;
      R $op (A...);
    }
  }
  provides {
    override public R $op (A...) {
      lock.acquire();
      R res = outer.$op (...);
      lock.release();
      return res;
    }
  }
```

This example illustrates several features of our design. Often, as here, we use a trait to wrap behavior around methods in a way that does not depend on the parameters or return type of the method. Since Java does not treat parameter sequences as tuples, we introduce the notation "$x...$" as a way to name parameter sequences with heterogeneous types, where the arity may vary from instance to instance. This notation can be used in the signatures of methods; within their bodies, the actual value of the parameter sequence is denoted by "...". When the trait is inlined, a tuple of types is given for the parameter sequence, as in the following example that synchronizes a string comparison method:

```
use SyncT (compare, int, (String, String));
```

The second feature to note is the **ThisType** keyword, which denotes the class that is using the trait. Here, we use **ThisType** to state the requirement that the class provides the `lock` field and an implementation of the `$op` method to be overridden by the trait. The scope of **ThisType** acts is the entire trait, so it may appear as an argument or return type of a method, for example. In particular, this means that traits can provide binary methods.

The last feature is the use of the **override** and **outer** keywords in the declaration of the provided method. The **override** keyword states that the method is replacing an existing method in the class, which could either be inherited or locally defined. The **outer** keyword is used to invoke the version of the method that is being overridden. The **outer** keyword is similar to **super**, except that it may only be used to invoke methods that have the **override** annotation. After a method is overridden by inlining a trait, it is considered locally defined, and so it can be overridden again by inlining another trait; this technique can be used to concatenate partial method implementations from multiple traits, as we show in a later example.

The following class uses the `SyncT` trait to implement an atomic test-and-set operation:

```
class C {
  private boolean x;
  private Mutex lock;
  boolean testAndSet () { boolean t = x; x = true; return t; }
  use SyncT (testAndSet, boolean, ());
  C () { lock = new Mutex(); x = false; }
}
```

Note that without the **override** annotation in the `SyncT` trait, there would be a conflict between the definition of `testAndSet` given in the body of `C` and the one provided by `SyncT`.

The **requires** clause of a trait can also be used to impose constraints on any type parameters the trait might have. These constraints can be *nominal* (using **extends**) or *structural* (using **implements**), with the latter allowing us to capture patterns like delegation, as in the following example:

```
trait DelegateT ($m, $f, T, A..., R)
  requires {
    T implements { R $m (A...); }
    ThisType implements { T $f; }
  }
  provides {
    R $m (A...) { return $f.$m(...); }
  }
```

We conclude with a more substantial example: the `Customer` class from Section 1. Classes like `Customer` are quite common in database applications, where relational databases are mapped onto the class hierarchy. Usually, such classes include large amounts of boilerplate code for performing this mapping. Numerous mechanisms have been proposed to alleviate this burden, including code generation and other forms of metaprogramming; sophisticated frameworks like *Hibernate*[4] and *Ruby on Rails*[5] are currently used to automate this mapping.

Figure 2 presents a code fragment using trait-based metaprogramming to tackle the mapping problem. Our solution uses two related traits: `BObjectT` factors out the code needed to query an SQL database, and `StringFieldT` maps a

[4] http://www.hibernate.org/
[5] http://www.rubyonrails.org/

```
trait BObjectT(String table)
  provides {
    protected void loadData(ResultSet r) {}
    protected void findBy(String whereClause) throws DataNotFound {
      Connection con = ... open connection to database ...
      Statement stmt = con.createStatement();
      String sql = "SELECT * FROM " + table + " WHERE " + whereClause;
      ResultSet r = stmt.executeQuery(sql);
      if (r.next()) {
        loadData(r);
      } else {
        throw new DataNotFound();
      }
    }
  }

trait StringFieldT($f, $g, $s, $fBy, String fieldName, int length)
  requires {
    ThisType implements {
      void loadData(ResultSet r);
      void findBy(String whereClause) throws DataNotFound;
    }
  }
  provides {
    use PropT($f, $g, $s, String);
    override String $s(String x) throws FieldTooSmall {
      if (x.length() > length) throw new FieldTooSmall();
      outer.$s(x);
    }
    override void loadData(ResultSet r) {
      $f = r.getString(fieldName);
      outer.loadData(r);
    }
    void $fBy(String x) throws DataNotFound, FieldTooSmall {
      if (x.length() > length) throw new FieldTooSmall();
      findBy(fieldName + " = '" + x + "'");
    }
  }

class Customer {
  use BObjectT("customers");
  use StringFieldT(name, getName, setName, findByName, "name", 40);
  use StringFieldT(addr, getAddr, setAddr, findByAddr, "address", 40);
  use StringFieldT(phone, getPhone, setPhone,
                   findByPhone, "phone_num", 40);
    ... etc ...
}
```

Fig. 2. Business objects: a sketch

field in an object to a string field in a database. The latter is a trait function with value parameters: `fieldName` and `length`. As a whole, the example demonstrates an idiom allowing traits to define "partial methods:" a base trait(`BObjectT`) is used to seed a class with an empty implementation of a method (`loadData`). Then a trait function (`StringFieldT`) is applied multiple times, each time extending the method's behavior before invoking the **outer** implementation.

3.2 From no parameters to too many?

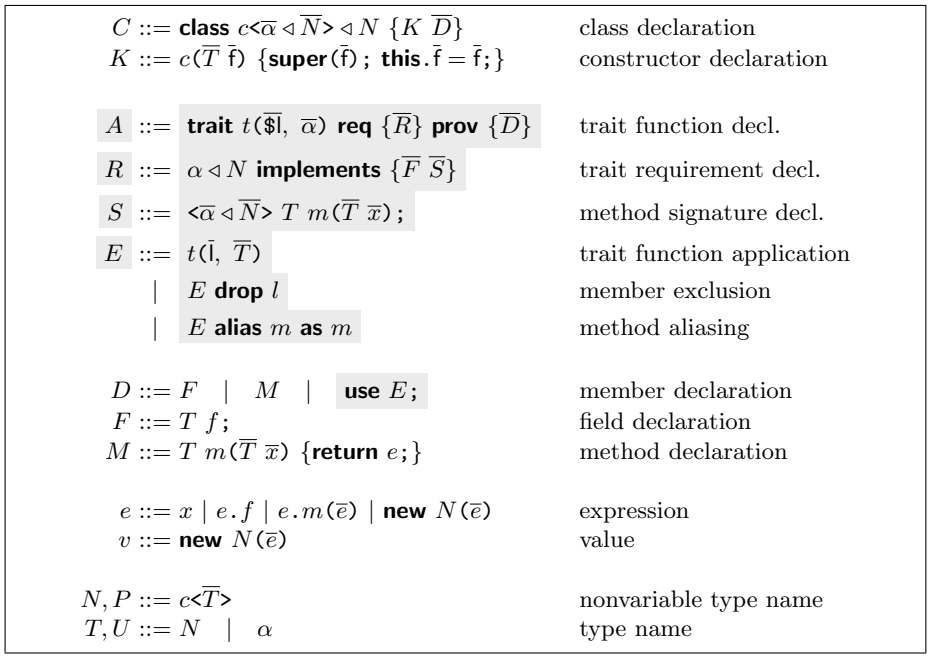
One apparent downside of the proposed mechanism is that, having introduced parameters, we need too many of them in order to encode interesting patterns. The `StringFieldT` trait, for example, takes a total of six parameters, and one can easily imagine adding more for a more sophisticated implementation. This problem is exacerbated by parameter sequences, where the user of a trait must tediously spell out a tuple of types. In many of these cases, however, the appropriate value for a parameter can be inferred or explicitly computed. For instance, if the `$f` parameter to `StringFieldT` is `name`, we can derive that `$g` should be `getName`, `$s` should be `setName`, and so on. Given a few primitives for label manipulation, these rules are easy to write down. Likewise, the type arguments to the `SyncT` trait can be inferred based on the actual method that the trait overrides, as long as no method overloading has occurred. Having the compiler infer these arguments makes our mechanism less cumbersome to use, and we take up the idea in a companion technical report [28]; as it turns out, this leads directly to a powerful form of *pattern matching* for trait functions.

4 A formal model: Meta-trait Java

Having informally described trait-based metaprogramming, we proceed to the formal model. The primary goal of this model is to study the type theory of our mechanism in the context of Java's nominal type system. Thus, we model only the core features of our proposal: we drop **super**, **outer**, and variable-arity parameters, since they do not substantially alter the type system, but do clutter its presentation. In earlier work, we presented a detailed semantics for compiling traits with hiding and renaming [25]; here, we give a simpler semantics that performs renaming only through trait functions. The relationship between the two models is discussed in Section 5.3.

Our calculus, MTJ, is essentially an extension of Featherweight Generic Java (FGJ); we drop FGJ's type casts and method type parameters since they do not interact with our type system in any interesting way.[6] Featherweight Java was designed to capture the minimal essence of Java, with particular focus on its type system and proof of soundness, and FGJ extends FJ with generics [17]. Our calculus adds traits and trait functions to FGJ, along with the additional

[6] For the remainder of this paper, when we refer to FGJ, we mean this restricted calculus.



C	::= **class** c<$\overline{\alpha} \lhd \overline{N}$> $\lhd N$ $\{K\ \overline{D}\}$	class declaration
K	::= $c(\overline{T}\ \overline{\mathsf{f}})$ $\{$**super**$(\overline{\mathsf{f}})$; **this**$.\overline{\mathsf{f}} = \overline{\mathsf{f}}$;$\}$	constructor declaration
A	::= **trait** $t(\overline{\$\mathsf{l}},\ \overline{\alpha})$ **req** $\{\overline{R}\}$ **prov** $\{\overline{D}\}$	trait function decl.
R	::= $\alpha \lhd N$ **implements** $\{\overline{F}\ \overline{S}\}$	trait requirement decl.
S	::= <$\overline{\alpha} \lhd \overline{N}$> $T\ m(\overline{T}\ \overline{x})$;	method signature decl.
E	::= $t(\overline{\mathsf{l}},\ \overline{T})$	trait function application
	| E **drop** l	member exclusion
	| E **alias** m **as** m	method aliasing
D	::= F | M | **use** E;	member declaration
F	::= $T\ f$;	field declaration
M	::= $T\ m(\overline{T}\ \overline{x})$ $\{$**return** e;$\}$	method declaration
e	::= x | $e.f$ | $e.m(\overline{e})$ | **new** $N(\overline{e})$	expression
v	::= **new** $N(\overline{e})$	value
N, P	::= c<$\overline{T}$>	nonvariable type name
T, U	::= N | α	type name

Fig. 3. MTJ: syntax

type-theoretic machinery needed to support those features. Like FGJ, we omit assignment, interfaces, overloading, and **super**-sends. MTJ is not equipped with its own dynamic semantics; instead, we define a translation from MTJ programs to FGJ programs. The type system, however, is given directly, and it conservatively extends FGJ's type system.

4.1 Syntax

The syntax of MTJ is given in Figure 3; portions highlighted in grey are extensions to FGJ's syntax. For the calculus, we abbreviate **extends** to $\lhd$, **requires** to **req**, and **provides** to **prov**. The metavariables c and d range over class names and t ranges over trait names. For field names and method names (collectively called *labels*), we separate variables from concrete names, as follows:

	Concrete	Variable	Either
Field names	f, g	$\$\mathsf{f}$	f
Method names	m	$\$\mathsf{m}$	m
Member names (labels)	l	$\$\mathsf{l}$	l, k

Note we assume the sets of field and method names are disjoint. Object is a class name, but cannot be defined in an MTJ program; **this** is a variable name, but cannot occur as a parameter.

To keep notation compact, we make heavy use of overbar sequence notation: $\bar{\mathsf{f}}$ denotes the possibly empty sequence $\mathsf{f}_1, \ldots, \mathsf{f}_n$, for example. Pairs of sequences are interleaved: $\overline{T}\ \bar{\mathsf{f}}$ stands for $T_1\ \mathsf{f}_1, \ldots, T_n\ \mathsf{f}_n$, and $\mathsf{this}.\bar{\mathsf{f}} = \bar{\mathsf{f}};$ stands for $\mathsf{this}.\mathsf{f}_1 = \mathsf{f}_1; \ldots; \mathsf{this}.\mathsf{f}_n = \mathsf{f}_n;$. Sequences are delimited as necessary to match Java syntax. Sequences of parameters are also assumed to contain no duplicate names. The empty sequence is denoted by $\bullet$, and sequence concatenation by the $\cdot$ operator. Finally, sequences with named elements are sometimes used as finite maps taking names to sequence elements. Thus, $\overline{D}(\texttt{foo})$ denotes the field or method declaration in $\overline{D}$ named `foo` (unambiguous because method and field names must be distinct).

A class table CT is a map from class names c to class declarations. Likewise, a trait table TT maps trait names t to trait declarations. A program is a triple $(\mathrm{CT}, \mathrm{TT}, e)$. In defining the semantics of MTJ, we assume fixed, global tables CT and TT. We further assume that these tables are *well-formed*: the class table must define an acyclic inheritance hierarchy, and the trait table must define an acyclic trait use graph.

4.2 Translation to FGJ

An FGJ program is an MTJ program with an empty trait table (and thus no trait use declarations). The semantics of MTJ are given by a translation function $[\![-]\!]$ that takes MTJ class declarations to FGJ class declarations. The translation *flattens* trait use declarations into sequences of FGJ member declarations, incorporating the bodies of traits into the classes in which they are used. As a consequence, the so-called *flattening property* [22] holds by construction: class members introduced through traits cannot be distinguished from class members defined directly within a class.[7]

Much of the work of translation is performed by substitution. Since trait functions are strictly first-order, the definitions of the various substitution forms (types for types, labels for labels, *etc.*) are straightforward and hence omitted.

The details of the translation are shown in Figure 4. Class declarations are translated by flattening the class body, keeping track of the name of the class so that any occurrences of **ThisType** can be replaced by it. Fields and methods are already "flat," so the only interesting member-level translation is for trait use declarations. To flatten a trait function application, we first substitute the actual parameters for the formal parameters within the trait body, and then flatten the result. To drop a member for an inlined trait, we simply remove it from the flattened collection of member delcarations. There is a subtlety in the semantics for aliasing: when recursive methods are aliased, do their recursive invocations refer to the original method or to the alias? We have chosen the latter interpretation, following Liquori and Spiwack [20]. This choice does not affect our type system, but does affect finer-grained type systems that track individual method requirements [25].

[7] A similar property, called the *copy principle*, has been defined for mixins [2].

$$[\![\,\textsf{class}\ c\langle\overline{\alpha} \lhd \overline{N}\rangle \lhd N\ \{K\ \overline{D}\}\,]\!] = \textsf{class}\ c\langle\overline{\alpha} \lhd \overline{N}\rangle \lhd N\ \{K\ [\![\overline{D}]\!]_{c\langle\overline{\alpha}\rangle}\}$$

$$\begin{aligned}
[\![F]\!]_N &= F \\
[\![M]\!]_N &= M \\
[\![\textsf{use}\ E;]\!]_N &= [\![E]\!]_N \\
\\
[\![t(\overline{\mathsf{l}},\ \overline{T})]\!]_N &= [\![[\overline{\mathsf{l}}/\overline{\$\mathsf{l}},\ \overline{T}/\overline{\alpha}, N/\textsf{ThisType}]\overline{D}]\!]_N \\
&\quad \text{where } \mathrm{TT}(t) = \textsf{trait}\ t(\overline{\$\mathsf{l}},\ \overline{\alpha})\ \textsf{req}\ \{\overline{R}\}\ \textsf{prov}\ \{\overline{D}\} \\
[\![E\ \textsf{drop}\ \mathsf{l}]\!]_N &= [\![E]\!]_N \setminus \mathsf{l} \\
[\![E\ \textsf{alias}\ \mathsf{m}\ \textsf{as}\ \mathsf{m}']\!]_N &= [\![E]\!]_N \cdot [\mathsf{m}'/\mathsf{m}]\,([\![E]\!]_N(\mathsf{m}))
\end{aligned}$$

Fig. 4. MTJ to FGJ translation

Note that translation is guaranteed to terminate, since the trait use graph is required to be acyclic.

4.3 Types in MTJ

We now turn to the static semantics for MTJ. One approach for constructing a type system for traits is to defer type checking of trait members until the trait is used in a class, then check the trait members as if they were declared within that class [20]. While this approach is pleasantly simple, requiring no changes to the existing type system for classes, it has at least one significant downside: type errors in a trait function may not be detected until that function is used, perhaps by a programmer using a library of such trait functions.

Our goal, in contrast, is to subsume FGJ's type system while *separately* type checking trait definitions, expressions, and uses. To achieve this goal, our calculus must give types to traits and trait expressions. Trait types must also be available at the expression level, because **this** and **ThisType** may appear in trait method bodies. In a structural type system, these requirements can be easily met by introducing *incomplete* object types to track trait requirements and assigning these types to traits [11, 3]; the type of a trait would then be a (structural) supertype of all classes that include that trait. Determining the status of trait types in a nominal type system is more difficult. One route is to associate a type name with each trait declaration [27], as is done for class declarations. Typing trait expressions involving aliasing or exclusion, however, is awkward with this approach.

The situation in MTJ is further complicated by the fact that trait functions are abstracted over labels and types, and may constrain their type parameters to implement interfaces that include abstract labels (Section 3). In principle these features could be supported in a purely nominal way, but we believe that the resulting type system would be too brittle and cumbersome, and would limit the programmer's ability to use existing classes as type parameters to traits.

N, P	$::= c\texttt{<}\overline{T}\texttt{>}$	nonvariable type name
T, U	$::= N \mid \alpha$	type name
τ	$::= N \diamond \sigma$	object type
	$\mid T$	type name
	$\mid \forall \overline{\alpha} <: \overline{\tau}.\tau$	bounded polymorphic type
	$\mid \overline{\tau} \to \tau$	function/method type
	$\mid \prod \overline{\$\mathsf{l}}.\tau$	label-dependent type
σ	$::= \langle l : \mu_l \ ^{l \in \mathcal{L}} \rangle_{\mathcal{R}}$	object signature
μ	$::= T \mid \overline{T} \to T$	object member signature
$\mathcal{R}$	$::= \{\bar{l}\}$	required member set

Fig. 5. MTJ: type syntax

In view of these concerns, we propose a hybrid structural/nominal type system. Purely nominal type systems must still check the structure of types to ensure soundness; the pertinent structure does not appear in the syntax of the types, but rather through auxiliary machinery (*e.g.*, *fields* and *mtype* in FGJ). Our type system exposes structural types syntactically: an object type $N \diamond \sigma$ is a pair of a type name N and an object signature σ. If an object has type $N \diamond \sigma$, then it is *nominally* a subtype of N, and *structurally* a subtype of σ. The nominal component is used for checking method arguments and return values, because in FGJ these constructions impose nominal subtyping constraints, while the structural component is used for checking field accesses and method invocations, corresponding to the structural-checking machinery in FGJ. The full syntax of MTJ types is given in Figure 5.

Of course, there is a relationship between the two components of an object type: for each nominal type N — for each class — there is a signature σ_N giving its interface. We call this signature the *canonical signature for* N. The purpose of the signature component in an object type is to impose *additional* structural constraints on the type of the object, beyond those already imposed by its canonical signature. These additional constraints can only be introduced through the **requires** and **provides** declarations in a trait function; thus, the constraints are only placed on type variables (including **ThisType**, which we treat as a type variable). The type variables in a trait are replaced by class names when trait function application is translated to FGJ. Our type system ensures that the constraints on these type variables are satisfied by the eventual class name arguments, ensuring the type-safety of the resulting FGJ code.

Notice that types τ include both object types and type names. A type name is either a nonvariable type name (which is a class name, possibly applied to type parameters) or a type variable. A nonvariable type name N stands for the object type $N \diamond \sigma_N$ that includes the canonical signature of the class. A type variable α stands for an unknown (but bounded) object type. The surface syntax

of the language prevents trait and class member declarations from introducing new object types: member declarations can only refer to named types. Thus, in the type syntax, object signatures are constrained to use type names rather than arbitrary object types. This constraint allows us to give a tidy account of recursive object types, as we shall see later.

An object signature σ is annotated with a set $\mathcal{R}$ of member names. In the object type for a trait, this set contains the name of all required members. For example, consider the following trait `BarT`, which requires a `foo` method and provides a `bar` method:

```
trait BarT
  requires { ThisType implements { Object foo(Object x); } }
  provides { Object bar(Object x) { return foo(foo(x)); } }
```

The type of `BarT` is $\texttt{Object} \diamond \langle \texttt{foo} : \texttt{Object} \to \texttt{Object},\ \texttt{bar} : \texttt{Object} \to \texttt{Object} \rangle_{\{\texttt{foo}\}}$. The nominal component of the type is `Object` because `BarT` places no nominal constraints on **ThisType**. Note that expression-level typing does not distinguish between the provided and required members of an object type, because traits are ultimately incorporated into classes that must provide all required members.

Classes are also given object types, as with the following polymorphic class [17]:

```
class Pair<X ◁ Object, Y ◁ Object> ◁ Object {
  Pair(X fst, Y snd) { super(); this.fst=fst; this.snd=snd; }
  X fst; Y snd;
  Pair<X,Y> setfst(X newfst) { return new Pair<X,Y>(newfst, snd); }
}
```

Our type system will give the following type to `Pair`:

$$\forall \texttt{X} <: \texttt{Object} \diamond \langle\rangle_{\emptyset}, \texttt{Y} <: \texttt{Object} \diamond \langle\rangle_{\emptyset}\ .\ \texttt{Pair<X,Y>} \diamond \left\langle \begin{array}{l} \texttt{fst} : \texttt{X},\ \texttt{snd} : \texttt{Y}, \\ \texttt{setfst} : \texttt{X} \to \texttt{Pair<X,Y>} \end{array} \right\rangle_{\emptyset}$$

Trait functions add an additional complication: the result *type* of a trait function may depend on its label parameters, but these labels are unknown *values*, not unknown types. We introduce a very limited form of *dependent types* [15] to address this issue. In our calculus, the dependent type $\prod \$\mathsf{l}.\tau$ represents a function that takes a label parameter and yields a value of type τ, where $\$\mathsf{l}$ may occur free in τ. For example, consider the following trait function:

```
trait GetterT ($f, $g, T)
  requires { ThisType implements { T $f; } }
  provides { T $g() { $f; } }
```

In MTJ, `GetterT` has the type

$$\prod \$\texttt{f}, \$\texttt{g}\ .\ \forall \texttt{T} <: \texttt{Object} \diamond \langle\rangle_{\emptyset}\ .\ \texttt{Object} \diamond \langle \$\texttt{f} : \texttt{T},\ \$\texttt{g} : \bullet \to \texttt{T} \rangle_{\{\$\texttt{f}\}}$$

To give the typing judgments of the system, we need a few definitions. A *context* Γ is a sequence of abstract labels $\$\mathsf{l}$ and variable typings $x : T$; we write $\$\mathsf{l} \in \Gamma$ and $\Gamma(x) = T$, respectively, to denote their occurrence in Γ. Each label or variable may only occur once in Γ. A *type context* Δ is a finite map from type variables α to types τ. Just as we fixed class and trait tables in the translation semantics, we fix a global class type table CTy and trait type table TTy for the

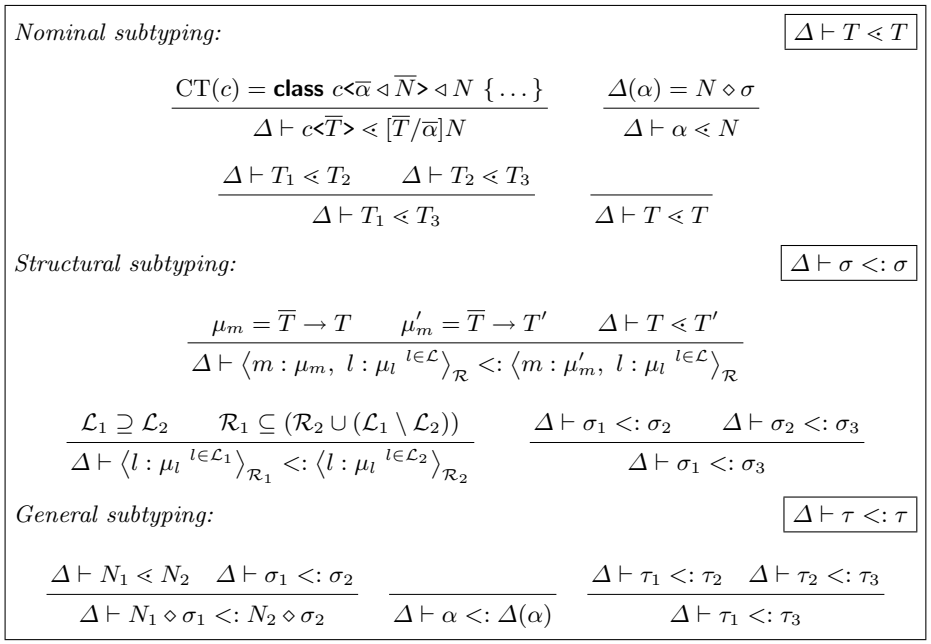


Fig. 6. MTJ: subtyping

static semantics. The former takes class names to types, the latter takes trait names to types. These tables play a role similar to a store typing: they give each class and trait a presumed type, allowing us to check mutually-recursive class definitions. Ultimately, we ensure that the actual type of each class and trait matches the type given in the table. Formally, we regard the tables as implicit contexts for our typing judgments.

4.4 Subtyping

MTJ has three forms of subtyping: *nominal subtyping*, written $\Delta \vdash N_1 \lessdot N_2$, *structural subtyping*, written $\Delta \vdash \sigma_1 <: \sigma_2$, and *general subtyping*, written $\Delta \vdash \tau_1 <: \tau_2$. These relations are defined in Figure 6.

The nominal subtyping relation is just FGJ's subtyping relation: it defines inheritance-based subtyping, which is the reflexive-transitive closure of the **extends** relation.

Structural subtyping applies to object signatures. We support both depth and width subtyping. For depth subtyping, we follow FGJ (and GJ) in providing only covariant subtyping on methods. We also consider a signature with fewer requirements to be a subtype of the same signature with more requirements; the reasons for this choice will become clear in Section 4.7.

General subtyping is defined so that the nominal and structural components of an object type may vary independently. In particular, it is sometimes neces-

Bound of type name:

$$\begin{aligned}\mathrm{type}(N) &= [\overline{T}/\overline{\alpha}]\tau_0 \text{ when } \mathrm{CTy}(c) = \forall\overline{\alpha} <: \overline{\tau}.\tau_0\\ \mathrm{bound}_\Delta(N) &= \mathrm{type}(N)\\ \mathrm{bound}_\Delta(\alpha) &= \Delta(\alpha)\end{aligned}$$

Well-formed type names: $\boxed{\Delta \vdash T \text{ OK}}$

$$\frac{\alpha \in \mathrm{dom}(\Delta)}{\Delta \vdash \alpha \text{ OK}} \qquad \frac{\mathrm{CTy}(c) = \forall\overline{\alpha} <: \overline{\tau}.\tau_0 \quad \Delta \vdash \overline{T} \text{ OK} \quad \Delta \vdash \mathrm{bound}_\Delta(\overline{T}) <: [\overline{T}/\overline{\alpha}]\overline{\tau}}{\Delta \vdash c\texttt{<}\overline{T}\texttt{>} \text{ OK}}$$

Expression typing: $\boxed{\Delta; \Gamma \vdash e \ : \ T}$

$$\frac{}{\Delta; \Gamma \vdash x \ : \ \Gamma(x)} \qquad \frac{\Delta; \Gamma \vdash e_0 \ : \ T_0 \qquad \mathrm{bound}_\Delta(T_0) = N \diamond \sigma}{\Delta; \Gamma \vdash e_0\,.f \ : \ \sigma(f)}$$

$$\frac{\Delta; \Gamma \vdash e_0 \ : \ T_0 \qquad \mathrm{bound}_\Delta(T_0) = N \diamond \sigma \qquad \sigma(m) = \overline{T} \to T' \qquad \Delta; \Gamma \vdash \overline{e} \ : \ \overline{U} \qquad \Delta \vdash \overline{U} \leqslant \overline{T}}{\Delta; \Gamma \vdash e_0\,.m(\overline{e}) \ : \ T'}$$

$$\frac{\Delta \vdash N \text{ OK} \qquad \mathrm{fields}(N) = \overline{T}\ \overline{f} \qquad \Delta; \Gamma \vdash \overline{e} \ : \ \overline{U} \qquad \Delta \vdash \overline{U} \leqslant \overline{T}}{\Delta; \Gamma \vdash \textsf{new}\ N(\overline{e}) \ : \ N}$$

Fig. 7. MTJ: expression typing

sary for the nominal component of a type to be promoted without affecting the structural component, as in the following example:

```
class HasFoo { Object foo; }
trait NeedsFooA requires { ThisType implements {foo : Object} }
trait NeedsFooB requires { ThisType ◁ HasFoo } provides { use NeedsFooA; }
```

In `NeedsFooB`, **ThisType** is bounded by $\texttt{HasFoo} \diamond \langle \texttt{foo} : \texttt{Object} \rangle_{\{\texttt{foo}\}}$. In `NeedsFooA`, however, **ThisType** is bounded by $\texttt{Object} \diamond \langle \texttt{foo} : \texttt{Object} \rangle_{\{\texttt{foo}\}}$, so a promotion of the nominal component of the bound is needed. Note that `foo` is marked required for `NeedsFooA` because it is not *provided* by the trait—in particular, it cannot be removed using **drop**—but is expected to be present in any class using the trait.

4.5 Static semantics: expressions

We present the static semantics of MTJ starting with expressions and working our way upwards.

As usual for a type system without a subsumption rule, we include a promotion function bound_Δ for computing the least nonvariable supertype of a given type, given in Figure 7. At the expression level, all types are named, so bound_Δ is only defined on type names. The type *computed* by bound_Δ, however,

is always an object type. Thus, using bound_Δ on a nonvariable type name corresponds to an iso-recursive "unfold:" the signature component of $\text{bound}_\Delta(N)$, *i.e.*, the canonical signature of N, is the one-step expansion of N. We use the function "type" to compute the canonical type; that function, in turn, uses the class table to discover the appropriate canonical signature. As in FGJ, we have a well-formedness check for type names, written $\Delta \vdash T$ OK, which ensures that the type parameters for a class respect their bounds.

The expression typing rules (Figure 7) are similar to their counterparts in FGJ, with a few notable differences. Most importantly, field access and method invocation are checked via object signatures, rather than separate machinery. These rules are the motivation for our hybrid type system, making it possible to type traits and classes in a uniform way. Our rule for method invocation is somewhat simpler than in FGJ, because we do not model generic methods. A final point to observe is that all premises involving subtyping use the *nominal* subtyping relation. Each such premise corresponds to a proposition that must hold, using FGJ's (nominal) subtyping relation, after translation of the expression.

4.6 Static semantics: member declarations and trait expressions

Type checking for classes and traits begins at the member level: the judgment $\Delta;\ \Gamma \vdash D\ :\ \tau$, given in Figure 8, assigns each member declaration an *object* type. This type should be understood as the least upper bound for the type of objects containing the declaration. For field and method declarations, the nominal component of the type will always be Object, while the structural component will give the label and type for that member. Trait use declarations are assigned the type of their trait expression, which may include nominal requirements.

Member declaration typing checks that any abstract labels are in scope ($\Gamma \vdash l$ OK). Notice that method bodies are checked via the expression typing judgment, and the type given to the body is (as usual) required to be a nominal subtype of the expected return type. We also check that any types appearing in the program text are well-formed.

Trait expression typing is fairly straightforward; here, our type system resembles Fisher and Reppy's [11]. Recall that, when trait function applications are translated, a class name is substituted for **ThisType** (Section 4.2); really, **ThisType** is an implicit type parameter to every trait. Thus, when checking a trait function application, we substitute the type of **this** (as given by Γ) for **ThisType** in the trait type. We also substitute the explicit type arguments, checking that they respect their bounds.

The **alias** operation requires that the method to be aliased is actually provided by the trait, and that no method with the aliased name is provided or required by the trait. Likewise, for **drop** we check that the member to be dropped is provided by the trait. Because the member might be mentioned in a method provided by the trait, we do not simply drop it from the trait signature, but rather mark it as required. A more precise type can be given if member requirements are tracked for each provided method [25], but this comes at a cost: it leaks fine-grained implementation details about the trait into its signature.

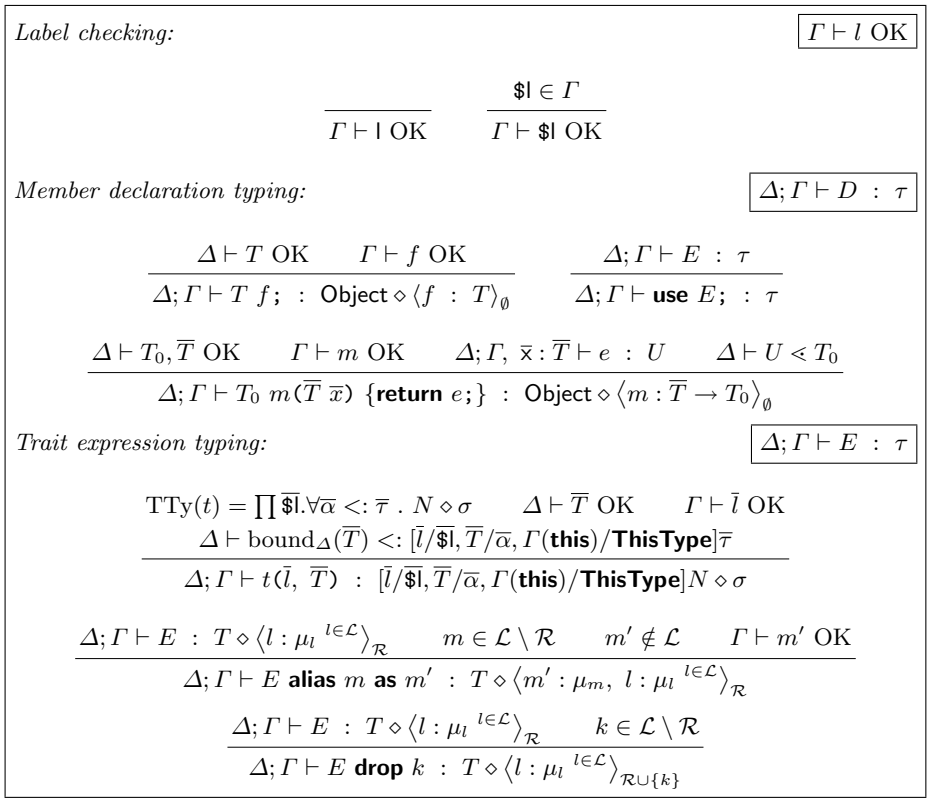


Fig. 8. MTJ: member-level typing

4.7 Static semantics: classes and traits

When typing a class or trait declaration, we attempt to find the meet (greatest lower bound) of its member declaration types. If the meet is defined, it gives us the type for the class or trait; if it is not defined, there is a type error. For example, consider the following class:

```
class C {
  int x;
  int getX() { return x; }
}
```

The types of the member declarations are

$$\texttt{Object} \diamond \langle \texttt{x} : \texttt{int} \rangle_\emptyset \quad \text{and} \quad \texttt{Object} \diamond \langle \texttt{getX} : \bullet \to \texttt{int} \rangle_\emptyset$$

respectively. The greatest lower bound of these types is

$$\texttt{Object} \diamond \langle \texttt{x} : \texttt{int},\ \texttt{getX} : \bullet \to \texttt{int} \rangle_\emptyset$$

Replacing `Object` in the nominal component of this type with `C`, we have the type of the class.

As another example, suppose we have a trivial trait that provides nothing, but requires a method `foo`:

```
trait ReqT
  requires { ThisType implements { Object foo(); } }
  provides {}
```

Notice that the type of `ReqT` is $\texttt{Object} \diamond \langle \texttt{foo} : \bullet \to \texttt{Object} \rangle_{\{\texttt{foo}\}}$. We can then define a class `A` that uses `ReqT`:

```
class A ◁ Object {
  A foo() { return this; }
  use ReqT;
}
```

Taking the meet of `ReqT`'s type with the type of `foo` defined in `A` yields the type $\texttt{Object} \diamond \langle \texttt{foo} : \bullet \to \texttt{A} \rangle_{\emptyset}$. This is why types with fewer required members are "smaller" according to the subtyping relation: when we take the meet of two types, one requiring a member and one providing it, the resulting type lists the member as provided. In the above example, the type of the required member was lowered as well. On the other hand, the class `B` is not well-typed.

```
class B ◁ Object {
  Object foo(Object x) { return x; }
  use ReqT;
}
```

The requisite meet is not defined for `B`, because its type for `foo` has no lower bound in common with `ReqT`.

The meet of two object types, written $\tau_1 \wedge_\Delta \tau_2$, is defined in Figure 9. In addition, we define *object type concatenation*, written $\tau_1 \oplus_\Delta \tau_2$, which yields the meet of its operands but also checks that they provide disjoint sets of members.

The judgment $\Delta; \Gamma \vdash R \Rightarrow \alpha <: \tau$ is used to gather trait requirements into type constraints. Recall that both nominal and structural requirements can be specified. Object type concatenation is used to compute a type encompassing the given structural requirements, while checking that there is at most one requirement for any member name. The rule takes the meet of this type with the nominal requirement, allowing structural requirements to refine, but not conflict with, its canonical signature. Thus, for instance, a trait cannot both require **ThisType** to be a subclass of `String` and also provide a `length` method that returns a `boolean`. The type constraint given by the judgment includes the labels of all required members in its requirement set.

The typing rule for trait function declarations is given in a declarative style: it uses a type context Δ mentioning types that are in turn checked under Δ. This is necessary for two reasons. First, a **requires** clause for one type parameter may mention any of the trait function's type parameters, so requirements must be checked under the the constraints they denote. Likewise, the upper bound for **ThisType** is needed for type checking member declarations, but the types given to those declarations are used to constrain **ThisType**. The result type of the trait function, τ_0, is the concatenation of the types of the provided and required

Requirements:

$$\mathrm{reqs}\left(N \diamond \left\langle l : \mu_l \ {}^{l \in \mathcal{L}} \right\rangle_{\mathcal{R}}\right) = N \diamond \left\langle l : \mu_l \ {}^{l \in \mathcal{R}} \right\rangle_{\mathcal{R}}$$

Object type meet: $\boxed{N \diamond \sigma \wedge_\Delta N \diamond \sigma = N \diamond \sigma}$

$$\frac{\begin{array}{c}\Delta \vdash N_i \lessdot N_j \text{ with } i, j \in \{1, 2\} \\ \Delta \vdash \sigma <: \sigma_1 \qquad \Delta \vdash \sigma <: \sigma_2 \qquad \Delta \vdash \sigma' <: \sigma_1, \Delta \vdash \sigma' <: \sigma_2 \implies \Delta \vdash \sigma' <: \sigma\end{array}}{N_1 \diamond \sigma_1 \wedge_\Delta N_2 \diamond \sigma_2 = N_i \diamond \sigma}$$

Object type concatenation: $\boxed{N \diamond \sigma \oplus_\Delta N \diamond \sigma = N \diamond \sigma}$

$$\frac{\sigma_1 = \left\langle l : \mu_l \ {}^{l \in \mathcal{L}_1} \right\rangle_{\mathcal{R}_1} \qquad \sigma_2 = \left\langle l : \mu_l \ {}^{l \in \mathcal{L}_2} \right\rangle_{\mathcal{R}_2} \qquad (\mathcal{L}_1 \cap \mathcal{L}_2) \subseteq (\mathcal{R}_1 \cup \mathcal{R}_2)}{N_1 \diamond \sigma_1 \oplus_\Delta N_2 \diamond \sigma_2 = N_1 \diamond \sigma_1 \wedge_\Delta N_2 \diamond \sigma_2}$$

Method signature declaration typing: $\boxed{\Delta; \Gamma \vdash S \ : \ \tau}$

$$\frac{\Delta \vdash T_0, \overline{T} \text{ OK} \qquad \Gamma \vdash m \text{ OK}}{\Delta; \Gamma \vdash T_0 \ m(\overline{T} \ \overline{x}) \, ; \ : \ \mathsf{Object} \diamond \left\langle m : \overline{T} \to T_0 \right\rangle_\emptyset}$$

Requirement constraints: $\boxed{\Delta; \Gamma \vdash R \Rightarrow \alpha <: \tau}$

$$\frac{\begin{array}{c}\Delta \vdash N \text{ OK} \qquad \Delta; \Gamma \vdash \overline{F} \ : \ \overline{\tau_\mathsf{f}} \qquad \Delta; \Gamma \vdash \overline{S} \ : \ \overline{\tau_\mathsf{s}} \\ \mathrm{bound}_\Delta(N) \wedge_\Delta \left(\bigoplus_\Delta \overline{\tau_\mathsf{f}} \cdot \overline{\tau_\mathsf{s}}\right) = N \diamond \left\langle l : \mu_l \ {}^{l \in \mathcal{L}} \right\rangle_\emptyset\end{array}}{\Delta; \Gamma \vdash \alpha \triangleleft N \ \textbf{implements} \ \{\overline{F} \ \overline{S}\} \Rightarrow \alpha <: N \diamond \left\langle l : \mu_l \ {}^{l \in \mathcal{L}} \right\rangle_\mathcal{L}}$$

Trait function declaration typing: $\boxed{A \ : \ \tau}$

$$\frac{\begin{array}{c}\Delta; \ \overline{\$\mathsf{l}} \vdash \overline{R} \Rightarrow \overline{\alpha} <: \overline{\tau} \qquad \Delta; \ \overline{\$\mathsf{l}} \vdash R_0 \Rightarrow \textbf{ThisType} <: \tau_0' \\ \Delta = \overline{\alpha} <: \overline{\tau}, \ \textbf{ThisType} <: \tau_0 \qquad \Delta; \ \overline{\$\mathsf{l}}, \ \textbf{this} : \textbf{ThisType} \vdash \overline{D} \ : \ \overline{\tau_\mathsf{decl}} \\ \tau_0 = \tau_0' \oplus_\Delta \left(\bigoplus_\Delta \overline{\tau_\mathsf{decl}}\right) \qquad \Delta \vdash \tau_0' <: \mathrm{reqs}(\tau_0)\end{array}}{\textbf{trait} \ t(\overline{\$\mathsf{l}}, \ \overline{\alpha}) \ \textbf{req} \ \{R_0 \ \overline{R}\} \ \textbf{prov} \ \{\overline{D}\} \ : \ \prod \overline{\$\mathsf{l}} \ . \ \forall \overline{\alpha} <: \overline{\tau} \ . \ \tau_0}$$

Class declaration typing: $\boxed{C \ : \ \tau}$

$$\frac{\begin{array}{c}K = c(\overline{U} \ \overline{\mathsf{g}}, \ \overline{T} \ \overline{\mathsf{f}}) \ \{\textbf{super}(\overline{\mathsf{g}}) \, ; \ \textbf{this}.\overline{\mathsf{f}} = \overline{\mathsf{f}}; \} \\ \mathrm{fields}(N) = \overline{U} \ \overline{\mathsf{g}} \qquad \mathrm{fields}(c\langle\overline{\alpha}\rangle) = \overline{U} \ \overline{\mathsf{g}}; \ \overline{T} \ \overline{\mathsf{f}} \\ \Delta = \overline{\alpha} <: \mathrm{type}(\overline{N}) \qquad \Delta \vdash \overline{N}, N \text{ OK} \qquad \Delta; \textbf{this} : c\langle\overline{\alpha}\rangle \vdash \overline{D} \ : \ \overline{\tau} \\ P \diamond \sigma = \bigoplus_\Delta \overline{\tau} \qquad \sigma = \left\langle l : \mu_l \ {}^{l \in \mathcal{L}} \right\rangle_\mathcal{R} \qquad N \diamond \sigma_N = \mathrm{type}(N) \\ \Delta \vdash \sigma <: (\sigma_N \restriction (\mathcal{L} \setminus \mathcal{R})) \qquad \mathcal{R} \subseteq \mathrm{dom}(\sigma_N) \qquad \Delta \vdash c\langle\overline{\alpha}\rangle \lessdot P\end{array}}{\textbf{class} \ c\langle\overline{\alpha} \triangleleft \overline{N}\rangle \triangleleft N \ \{K \ \overline{D}\} \ : \ \forall \overline{\alpha} <: \mathrm{type}(\overline{N}) \ . \ c\langle\overline{\alpha}\rangle \diamond \sigma \wedge_\Delta N \diamond \sigma_N}$$

Fig. 9. MTJ: class and trait typing

members of the trait. Using concatenation rather than meet ensures that the trait does not contain multiple definitions of a member.

Note that a trait function t may use other traits without fulfilling their requirements. In this case, we insist that t explicitly state the unfulfilled requirements in its **ThisType** constraints, which is checked by the hypothesis $\Delta \vdash \tau_0' <: \text{reqs}(\tau_0)$, where τ_0' is the bound t places on **ThisType** and τ_0 is the result type of the trait function.

Class declaration typing is similar to trait function typing: the types of the class's member declarations are used to partially determine the class's type via concatenation. There are several important differences, however. For one, the class type includes the canonical signature of its immediate superclass (σ_N in the rule). If a class overrides any members of its superclass, the overriding definitions must be subtypes of the originals. Hence, we check that the signature of the superclass, σ_N, restricted to the members defined in the class body, $\mathcal{L} \setminus \mathcal{R}$, is a supertype of the signature for the class body, σ. Another difference is that all trait requirements must be fulfilled by the class. This is checked in two ways. First, the set of required members from the class body, $\mathcal{R}$, must be a subset of the members provided by the superclass, $\text{dom}(\sigma_N)$. Second, class itself is required to be a nominal subtype of any nominal requirements introduced by the traits it uses ($\Delta \vdash c\langle\overline{\alpha}\rangle \lessdot P$).

4.8 Soundness

The semantics of MTJ is given by a translation to FGJ, but the resulting FGJ class table is *also* a valid MTJ class table. Thus, our soundness result is broken into two steps, one taking place entirely within MTJ and one relating the two calculi. We briefly survey the result here, with a detailed version of the proof available in a companion technical report [28].

Definition 1. *A class C is **flat** if it contains no trait use declarations.*

Note that limiting the syntax of MTJ to flat class declarations yields the syntax of FGJ, modulo the features that we dropped (casts and generic methods).

To prove soundness, we need to ensure that the presumed class and trait types from the class type and trait type tables agree with the actual classes and traits in CT and TT.

Definition 2. *A class type table CTy **agrees** with a class table CT, written $CT \vdash CTy$, if $dom(CT) = dom(CTy)$ and for all $c \in dom(CT)$, we have $CT(c) : CTy(c)$. We write $TT \vdash TTy$ for the same property relating the trait tables.*

We can now show a typical soundness result purely in terms of MTJ; here, translation acts as the "dynamic semantics" for MTJ and we prove that any well-typed program will successfully translate to a program with the same type.

Theorem 1 (Soundness of translation). *If $CT \vdash CTy$ and $TT \vdash TTy$ then, for all $c \in dom(CT)$, we have that $C = [\![CT(c)]\!]$ is defined, that C is flat, and that $C : CTy(c)$. Furthermore, if $\vdash e : T$ under CT, then $\vdash e : T$ under the translated class table.*

This theorem is straightforward to prove. First, we prove a series of standard lemmas for weakening of the context and type and label substitution. These are sufficient to prove the theorem, since translation is essentially trait function application. A minor twist comes in the lemma showing type preservation for member declaration translation. The type of the original member is not always the same as the translated member: if the original member is a trait use declaration, and the trait places requirements on **ThisType**, those requirements will not appear in the flattened trait body. Thus, the translation preserves only the *provided* elements of a member declaration type. Theorem 1 still holds, however, because the class typing rule ensures that there are no residual requirements, so the type of the class as a whole is preserved under translation.

We then show the following result, relating MTJ to FGJ.

Theorem 2 (Well-typed, flat MTJ programs are well-typed FGJ programs). *If* $(CT, \bullet, e)$ *is an MTJ program with only flat class declarations and* $CT \vdash CTy$, $TT \vdash TTy$, *and* $\vdash_{MTJ} e : T$, *then* (CT, e) *is a well-typed FGJ program and* $\vdash_{FGJ} e : T$.

This theorem is even easier to prove: we prove that our canonical type signatures give the same results as FGJ's machinery (*e.g.*, the *mtype* function), and then prove by a series of inductive arguments that our typing judgments imply the corresponding judgments in FGJ. Taking the two theorems together, we have that a type-safe MTJ program translates to a type-safe FGJ program.

5 Discussion

5.1 Related work: metaprogramming

Broadly speaking, metaprogramming consists of writing (meta) programs that manipulate (object) programs.Compilers are the best-known metaprograms, but the technique is also useful for generating high-level code. In particular, *generative programming* has been proposed as a paradigm for building families of related systems: code and other artifacts are generated from a high-level model or specification, automating much of the software development process [6]. Metaprogramming, of course, is a crucial element of this process. Since metaprogramming raises the level of abstraction and can arbitrarily modify the meaning of code, it is important that metaprogramming frameworks strike a good balance between expressiveness, invasiveness, readability, and safety guarantees.

Draheim *et al.* give a good summary of several metaprogramming frameworks for Java and similar languages, focusing specifically on their utility for generative programming [7]. A typical approach is to use so-called meta-objects to represent and alter code entities (classes, methods, *etc.*). The implementation of the meta-objects gives rise to a *meta-object protocol* (MOP) that can be overridden or extended with new features [18]. MOP frameworks have been used both to generate code and to modify the semantics of language mechanisms such as multiple inheritance. They are extremely flexible, but require manipulation of ASTs and provide very few guarantees about generated code.

An alternative approach is to incorporate metaprogramming constructs directly into the language. SafeGen [16], for example, extends Java with *cursors* and *generators*. Cursors pick out a collection of entities within the code of a program, while generators, guided by cursors, output code fragments. Generators are written in a quasi-quotation style, giving the system a great deal of flexibility. Perhaps the most interesting aspect of SafeGen is that generators are statically checked for safety, using a theorem prover to check short first-order sentences produced by the type checker. Programmers are insulated from the theorem-proving process: from their perspective, it is just another type system.

Aspect-oriented programming (AOP) is another form of metaprogramming, where *advice* is *weaved* into existing code [19]. Our proposal has significant similarities with AOP, but also significant differences. Trait functions enable programmers to abstract "cross-cutting concerns" in a way similar to aspects; advice often wraps methods with function, just as we do with examples like `SyncT`. The most important difference is a matter of control: aspects control their own application to classes through *pointcuts*, but traits are explicitly included in classes.

Fähndrich *et al.* have described an elegant pattern-based approach to metaprogramming, similar to AOP, but focused on generating new function, rather than modifying existing behavior [10]. Their system is template-based, but uses pattern matching to determine how to instantiate the templates. The patterns provide constraints that lead to strong static guarantees about the templates. In our technical report, we sketch a design inspired by this idea: the member requirements for a trait function are matched against the members defined in a class, and the trait is automatically applied for each match [28]. This brings trait-based metaprogramming much closer to AOP, but the control of trait application still remains in the hands of the class designer, who must explicitly request the pattern matching to take place. Moreover, our design has a much coarser-grained notion of pointcuts than AOP, since traits cannot be inserted at arbitrary points in the control-flow of a method.

Most similar to our proposal, the Genoupe framework [8] for C# supports code generation through parameterized classes. Classes are parameterized over types and values, and may contain code that inspects their parameters at compile-time, generating code as it does so. For example, classes can use a **@foreach** keyword to loop over the fields or methods of a type parameter. The code within the **@foreach** will be generated repeatedly for each match. Genoupe includes some static type checking of parameterized classes, but it cannot guarantee the well-formedness of the generated code. Moreover, generation results in complete classes, which cannot be combined in a single-inheritance language.

In general, the novelty of our approach is its particular focus on *member-level patterns* and its strength is in simplicity. Typed traits are composable, incomplete class implementations, and with our extension, they offer a uniform, expressive, and type-safe way to do metaprogramming without resorting to AST manipulation. In addition, the result of this metaprogramming is always just a trait, leaving ultimate control of the code to the class designer.

5.2 Related work: type systems

Nominal subtyping is a refinement of structural subtyping: type names are placed in a nominal subtyping relationship, but the types these names represent must be structurally related to guarantee type safety. In purely nominal type systems, types must be always be named, and subtyping always explicitly stated; "combining" structural and nominal subtyping amounts to relaxing these requirements. Moby [13] and Unity [21] relax them entirely, allowing the use of arbitrary structural subtyping. In Moby, there are *object types* and *class types*, the latter naming a specific class. Subtyping on class types is based on the explicit inheritance hierarchy, and so is essentially nominal, while object types are compared structurally. Unity is closer to our type system in that object types include a nominal component (called a *brand*) *and* a structural component. In both type systems, as with ours, nominal types have associated "canonical" structural types describing their interface. Programmers can choose whether to constrain types structurally or nominally, or, with Unity, both.

Our proposal also allows arbitrary structural subtyping, but only at the trait function level; subtyping for expressions is strictly nominal. We believe this paradigm to be widely applicable: a metalanguage with flexible, structural subtyping can be used to generate code for an object language with a more rigid, nominal type system. Moreover, since traits are just (incomplete) collections of class members, our type system can be used for other metaprogramming frameworks that do not make traits an explicit programming construct but still assemble classes from partial implementations. Though type parameters will not be tied to trait functions in such frameworks, they can still be used at the metaprogramming level with purely structural constraints, since they will not be present in generated code.

In Ancona *et al.*'s *polymorphic bytecode* proposal, compilation units are type-checked without complete knowledge of the inheritance hierarchy: type-checking results in a set of structural and nominal constraints to be satisfied by the eventual, dynamically-linked class hierarchy [1]. The combination of nominal and structural constraints resembles our object types, and the system retains Java's purely nominal subtyping after linking is performed. Polymorphic bytecode, in order to respect Java's type system, must place nominal constraints on types any time a method is invoked: it has no analog to our trait functions, which allow purely structural constraints to be imposed and discharged.

5.3 Related work: traits

The introduction of traits for Smalltalk [9] prompted a flurry of work on traits for statically-typed languages. Fisher and Reppy developed the first formal model of traits in a statically-typed setting [12], subsequently extending it to support polymorphic traits and stateful objects [11]. The model type checks traits in isolation from classes. The structural component of our type system is essentially a variant of Fisher and Reppy's type system. In our previous workshop paper [25], we reformulated the Fisher-Reppy trait calculus using Riecke-Stone

dictionaries [26], giving a semantics for member renaming and hiding operations on traits. The calculus renames members by modifying a dictionary, rather than substituting labels in program code. Thus, it provides a foundation for the separate compilation of trait functions in Moby, which already uses such dictionaries in its implementation. Separate compilation in Java remains future work.

Multiple designs extending Java with traits have been proposed. Smith and Drossopoulou describe a family of three such extensions, called Chai [27]. They support separate type checking of traits by introducing trait names into Java's type system; in essence, traits define interfaces, and the classes that use them are considered to have "implemented" those interfaces. As discussed in Section 4, this approach is probably too brittle to support trait functions. Another proposed design is *FeatherTrait Java* [20], which adds traits to Featherweight Java, but defers all type checking until traits have been included in a class. There are strong similarities between traits and *mixins* [5]; a good discussion of their relationship can be found in [14].

5.4 Conclusion

We have presented a language design for *metaprogramming with traits*. We believe our proposal hits a sweet spot for metaprogramming: while its semantics are very simple, it is capable of capturing a wide variety of patterns occurring at the member level of class definitions. In modeling our mechanism formally, we have developed a type system which incorporates a mixture of structural and nominal subtyping, and proved the soundness of the resulting calculus. An implementation is underway, written as a source-to-source translator for Java.

Acknowledgments We thank the anonymous reviewers for their help in catching mistakes and improving the overall presentation.

References

1. D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic bytecode: compositional compilation for Java-like languages. In *POPL'05*, pages 26–37, 2005.
2. D. Ancona, G. Lagorio, and E. Zucca. Jam–designing a Java extension with mixins. *TOPLAS*, 25(5):641–712, Sept. 2003.
3. V. Bono, M. Bugliesi, and L. Liquori. A Lambda Calculus of Incomplete Objects. In *MFCS*, pages 218–229, 1996.
4. G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, Mar. 1992.
5. G. Bracha and W. Cook. Mixin-based inheritance. In *ECOOP'90*, pages 303–311, New York, NY, Oct. 1990. ACM.
6. K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., 2000.
7. D. Draheim, C. Lutteroth, and G. Weber. An Analytical Comparison of Generative Programming Technologies. Technical Report B-04-02, Institute of Computer Science, Freie Universität Berlin, January 2004.
8. D. Draheim, C. Lutteroth, and G. Weber. A Type System for Reflective Program Generators. In *GPCE'05*, pages 327–341, 2005.

9. S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A Mechanism for fine-grained Reuse. *TOPLAS*, 28(2):331–388, Mar. 2006.
10. M. Fähndrich, M. Carbin, and J. R. Larus. Reflective program generation with patterns. In *GPCE'06*, pages 275–284, New York, NY, USA, 2006. ACM Press.
11. K. Fisher and J. Reppy. Statically typed traits. Technical Report TR-2003-13, Dept. of Computer Science, U. of Chicago, Chicago, IL, Dec. 2003.
12. K. Fisher and J. Reppy. A typed calculus of traits. In *FOOL11*, Jan. 2004.
13. K. Fisher and J. H. Reppy. Extending Moby with Inheritance-Based Subtyping. In *ECOOP'00*, pages 83–107, 2000.
14. M. Flatt, R. B. Findler, and M. Felleisen. Scheme with Classes, Mixins, and Traits. In *APLAS'06*, 2006.
15. M. Hofmann. Syntax and Semantics of Dependent Types. In *Semantics and Logics of Computation*, volume 14, pages 79–130. Cambridge University Press, 1997.
16. S. S. Huang, D. Zook, and Y. Smaragdakis. Statically Safe Program Generation with SafeGen. In *GPCE'05*, pages 309–326, 2005.
17. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.
18. G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The art of metaobject protocol*. MIT Press, Cambridge, MA, USA, 1991.
19. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97*, pages 220–242, 1997.
20. L. Liquori and A. Spiwack. Feathertrait: A modest extension of featherweight java. *TOPLAS*, to appear, 2007.
21. D. Malayeri and J. Aldrich. Combining Structural Subtyping and External Dispatch. In *FOOL/WOOD'07*, 2007.
22. O. Nierstrasz, S. Ducasse, and N. Schärli. Flattening Traits. *Journal of Object Technology*, 5(4):129–148, June 2006.
23. M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. An overview of the Scala programming language (second edition). Technical Report LAMP-REPORT-2006-001, EPFL, Lausanne, Switzerland, May 2006.
24. P. J. Quitslund. Java traits — improving opportunities for reuse. Technical Report CSE 04-005, OGI School of Science & Engineering, Sept. 2004.
25. J. Reppy and A. Turon. A foundation for trait-based metaprogramming. In *FOOL/WOOD'06*, 2006.
26. J. G. Riecke and C. A. Stone. Privacy via subsumption. *INC*, 172(1):2–28, Jan. 2002. A preliminary version appeared in FOOL5.
27. C. Smith and S. Drossopoulou. Chai: Traits for Java-Like Languages. In *ECOOP'05*, pages 453–478, 2005.
28. A. Turon. Metaprogramming with Traits. Honors thesis, forthcoming as a University of Chicago technical report, 2007.