Sharing Classes Between Families

Xin Qi Andrew C. Myers

Department of Computer Science
Cornell University
{qixin,andru}@cs.cornell.edu

Abstract

Class sharing is a new language mechanism for building extensible software systems. Recent work has separately explored two different kinds of extensibility: first, family inheritance, in which an entire family of related classes can be inherited, and second, adaptation, in which existing objects are extended in place with new behavior and state. Class sharing integrates these two kinds of extensibility mechanisms. With little programmer effort, objects of one family can be used as members of another, while preserving relationships among objects. Therefore, a family of classes can be adapted in place with new functionality spanning multiple classes. Object graphs can evolve from one family to another, adding or removing functionality even at run time.

Several new mechanisms support this flexibility while ensuring type safety. Class sharing has been implemented as an extension to Java, and its utility for evolving and extending software is demonstrated with realistic systems.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Object-oriented languages; D.3.3 [*Language Constructs and Features*]: Classes and objects, frameworks, inheritance, modules, packages

General Terms Languages

Keywords family inheritance, views, masked types

1. Introduction

It has long been observed that much of the difficulty of building reliable software systems arises from their size and complexity [5]. Language mechanisms for modular and object-oriented programming have helped make large systems easier to build and to reason about. However, it remains challenging to extend, evolve, and update software systems. Incremental updates to software systems may require coordinated changes to state and behavior that span multiple software components. We posit that new mechanisms are needed to enable software to be reused and extended in a modular, scalable, safe way.

Inheritance does provide a modular way to extend existing code with new behavior, which helps explain why large software systems are often built using object-oriented languages such as Java or C++. But ordinary inheritance has two serious limitations. First, interacting classes can only be extended individually; inheritance does not support changes that span multiple classes. Second, new functionality cannot be added to objects of an existing class without modifying the class definition—which would not be modular.

These limitations have been addressed separately by two lines of research. In the first, the ability to extend classes as a group is provided by *family inheritance* mechanisms, which provide inheritance at the granularity of a *family* of related classes rather than on individual classes. Family inheritance mechanisms include virtual classes [24, 18, 10], mixin layers [41], variant path types [22], and nested inheritance [28, 30]. The second, separate line of research has explored language support for *adaptation*. Adaptation mechanisms allow the modular addition of functionality to an existing class, without any change to the original class definition. Objects of the class are augmented with new operations or state. Mechanisms for adaptation include open classes [11], expanders [50], classboxes [3], and aspect binders [26]. Typically, adaptation operates on individual classes; it does not support coordinated changes while preserving relationships among augmented objects.

This paper integrates these two distinct approaches to extensibility for the first time. The result is a powerful new capability: interacting objects from a family of classes can be adapted with new functionality, while preserving the identities of objects and the relationships among them. A single operation can cause an entire data structure to be augmented in place with new behavior.

The key new idea is *class sharing*: different families can have classes that are shared between the families. An object that is an instance of the shared class in one family is also an instance of the corresponding class in any family that shares the class. Objects can then be viewed from either family. Class sharing supports *bidirectional adaptation*: not only can objects of a base family be adapted into a derived family, but those of the derived family can be adapted to the base family.

Class sharing has been realized in a language J&$_s$ (pronounced "jet-ess"), an extension of the Java language that adds class sharing to the language J& [30]. In J&$_s$, each object reference provides a *view* that dynamically determines the family and thus the behavior of the object when used through that reference. The view of an object also transitively affects the behavior of other objects reached via that object reference, because it determines which family to interpret those objects with respect to. An explicit *view change* produces a reference with a new view, while preserving object identity.

Class sharing enables new ways to extend software systems:

- *Family adaptation.* New functionality spanning multiple classes can be added to an entire family in-place, adapting objects with new state and behavior.
- *Dynamic object evolution.*

  The family from which an object is viewed is determined dynamically by its reference, so collections of interacting objects can be updated with new behavior at run time, without changing the interface to existing clients. This allows running servers to be evolved with new state and behavior.

- *In-place translation.* In-place translation is the transfer of data structures from one family to another without creating new objects for shared classes and without side effects. We have used this approach to build compilers: an abstract syntax tree from one language can be translated to another language while updating only the nodes that require it.

Despite all this expressive power, J&$_s$ is type-safe—programs never create run-time errors—and type checking is modular. Because the behavior of an object depends on the family from which it is being viewed, designing a sound type system is challenging. To accomplish this, J&$_s$ uses *view-dependent types* to ensure that late-bound type names belong to consistent families, and uses *masked types* [35] to control access to fields that are not shared between families. The key features of J&$_s$ have been formalized in an object calculus, and the type system has been proved sound. The details of the formalization and the proof are available in a companion technical report [36].

The rest of the paper proceeds as follows. Section 2 describes how class sharing works in the new language J&$_s$, and gives examples of its use. Section 3 shows how to safely support unshared state within shared classes. Section 4 sketches the formalization of J&$_s$ and the soundness theorem. The implementation of J&$_s$ is described in Section 5. Section 6 shows that the performance of J&$_s$ is reasonable and describes its use to build realistic systems. Section 7 discusses related work, and Section 8 concludes.

2. Sharing classes

The new class sharing mechanism in J&$_s$ is a safe, modular mechanism that relaxes the disjointness of class families. A family of classes may not only inherit another family (and hence all its nested classes), but may also *share* some of the classes from the family it inherits from. This enables new kinds of extensibility.

2.1 Family inheritance

J&$_s$ builds on the family inheritance mechanisms introduced by Nystrom et al.: nested inheritance [28] and nested intersection [30].

Nested inheritance is inheritance at the granularity of a *namespace* (a package or a class), which defines a family in which related classes are grouped. Nested inheritance supports coordinated changes that span the entire family. When a namespace inherits from another (base) namespace, not only are fields and methods inherited, but also namespaces nested in the base namespace. In addition, the derived namespace can *override*, or *further bind* [25] inherited namespaces, changing nested class declarations. Nested inheritance does not provide adaptation, however. Nested classes in the inheriting namespace, even those not overridden (*implicit classes*), are different classes than those of the same name in the base namespace.

Nested intersection supports *composing* families with generalized *intersection types* [39, 12]. Given classes S and T, their intersection $S\&T$ inherits all members of S and T. When two namespaces are intersected, their common nested namespaces are themselves intersected, i.e., $(S\&T).C = (S.C)\&(T.C)$.

Figure 1 shows an example with two families of classes. Class `AST` contains a family of classes for representing expressions, and class `TreeDisplay` contains a family of classes for graphically visualizing trees. Figure 2 shows the skeleton of code that implements the functionality of displaying an expression as a tree—without changing the existing families. `ASTDisplay` inherits both `AST` and `TreeDisplay`, and therefore inherits all nested classes from both of them. The GUI classes are implicit in `ASTDisplay`,

```
class AST {                 | class TreeDisplay {
  class Exp {...}           |   class Node {
  class Value               |     void display() {...}
     extends Exp {...}      |   }
  class Binary              |   class Composite extends Node {
     extends Exp            |     Node getChild(int i) {...}
  { Exp l, r; }             |   }
  ...                       |   class Leaf extends Node {...}
}                           | }
```

Figure 1. An expression family and a GUI family

```
1  class ASTDisplay extends AST & TreeDisplay {
2    class Exp extends Node { void display() { ... } }
3    class Value extends Exp & Leaf { ... }
4    class Binary extends Exp & Composite
5      { void display() { ... l.display(); ... } }
6    void show(Exp e) { e.display(); }
7  }
```

Figure 2. Mixing display into expressions, with nested inheritance

and expression classes are further bound to inherit GUI classes in addition to their original superclasses, and to override GUI methods with appropriate rendering code. As the `show` method in Figure 2 demonstrates, expression classes in the new family support the added `display` method.

Two mechanisms are essential for nested inheritance to work: *late binding of type names* and *exact types*. These mechanisms are also important for class sharing.

Late binding of type names. Late binding of type names ensures relationships between classes are preserved in the derived family.

When the name of a class is inherited into a new namespace, the name is interpreted in the context of the new namespace. For example, inside the family `ASTDisplay`, the type `Exp` refers to `Exp` in `ASTDisplay`. Consider the field `l`, which is declared in `AST.Binary` with type `Exp`, and then inherited by the class `ASTDisplay.Binary`. When inherited, its type is the `Exp` of `ASTDisplay`, *not* the original type. This late binding makes the call `l.display()` on line 5 legal. Similarly, the superclass of `ASTDisplay.Binary` is `ASTDisplay.Exp`, not `AST.Exp`.

Two mechanisms make the late binding of type names type-safe: *dependent classes* and *prefix types*. The dependent class p`.class` represents the run-time class of the object referred to by the *final access path* p. A final access path is either a final local variable, including `this` and final formal parameters, or a field access p`.f`, where p is a final access path and `f` is a final field. In general, the class represented by p`.class` is statically unknown, but fixed. A prefix type P`[`T`]` represents the enclosing namespace of the class or interface T that is a subtype of the namespace P, i.e., the family at the level of P that contains T [28]. In Figure 2, if one writes `AST[this.class]`, it refers to either `AST` or `ASTDisplay`, depending on the run-time class of the value stored in `this`.

Type names that are not fully qualified are sugar for members of prefix types that depend on the current class `this.class`, and in an inheriting family, they will be reinterpreted. In Figure 1, the type `Exp` inside class `AST` is sugar for `AST[this.class].Exp`.

Exact types. Both dependent classes and prefix types of dependent classes are *exact types* [6]: all instances of these types must have the same run-time class.

Simple types may be exact too. If A is a class, the exact type $A!$ represents values of the run-time class A. Even if class B inherits from A, neither B nor $B!$ is a subtype of $A!$. Exactness applies to the entire type preceding "!", so supertypes of a simple exact type can be obtained by shifting the exactness outward. For example, `ASTDisplay.Exp!` is not a subtype of `AST.Exp!`,

```
1   class ASTDisplay extends AST & TreeDisplay {
2     class Exp extends ... shares AST.Exp { ... }
3     class Value extends ... shares AST.Value { ... }
4     class Binary extends ... shares AST.Binary
5       { void display() { ... l.display(); ... } }
6     void show(AST!.Exp e) sharing AST!.Exp = Exp {
7       Exp temp = (view Exp)e;
8       temp.display();
9     }
10  }
```

Figure 3. Using class sharing to adapt `Exp` to `TreeDisplay`

but it is a subtype of `ASTDisplay!.Exp`, which is a subtype of `ASTDisplay.Exp`.

Exact types also restrict the subtyping relationships of nested types. For example, `ASTDisplay!.Binary` is not a subtype of `AST!.Binary`, even though `ASTDisplay.Binary` is a subtype of `AST.Binary`. Therefore exact types can mark the boundary of a family: classes nested in `ASTDisplay!` form one family and those nested in `AST!` form another. Non-dependent exact types provide a *locally closed world*: at compile time, one can enumerate *all* classes that are subtypes of $A!.C$, without inspecting subclasses of A. Exact types are important for the modularity of J&$_S$ (Section 2.5).

2.2 Sharing declarations

J&$_S$ introduces shared classes with *sharing declarations* in the derived family, such as the declaration "`shares A.C`" in this code:

```
class A { class C ... } // the base family
class B extends A {     // the derived family
  class C extends D shares A.C { ... }
}
```

This sharing declaration establishes a *sharing relationship* between classes `A.C` and `B.C`. Sharing declarations induce an equivalence relation on classes that is the reflexive, symmetric, and transitive closure of the declared sharing relationships. If two classes have a sharing relationship, they have the same set of object instances. However, subclasses of the two shared classes are not automatically shared, unless the subclasses also have appropriate sharing declarations. Therefore, the sharing relationship established in the above example can be represented as a relationship between two exact types, written $\mathtt{A.C!} \leftrightarrow \mathtt{B.C!}$.

J&$_S$ requires that only an overriding class in a derived family (e.g., `B.C`) may declare a sharing relationship with the overridden class in a base family (e.g., `A.C`). This restriction helps keep shared classes similar to each other.

Sharing vs. subtyping. The J&$_S$ language keeps subtyping largely separate from sharing. Adding a sharing relationship does not change the subtyping relation. In the above example, `B.C` is a subtype of `A.C`, and `B!.C` is not a subtype of `A!.C` due to the exactness, whether there is a sharing declaration or not. The sharing relationship does not make `A.C` a subtype of `B.C`, nor does it create any subtyping relationship between `A!.C` and `B!.C`.

Since sharing is not subtyping, it is in general not allowed to directly treat an object of a shared class as an instance of the other class in the sharing relationship; an explicit *view change* may be required (see Section 2.3 for more details).

Family adaptation. Sharing solves the problem of *object adaptation* [50], in which the goal is to augment existing objects with new behavior or state. Adaptation is different from inheritance, where only objects of new classes have the new behaviors. J&$_S$ is the first language to fully integrate family inheritance with in-place adaptation that preserves object identity. Moreover, because sharing is an equivalence relation on classes, J&$_S$ supports bidirectional, transitive adaptation.

Adaptation can improve the example code from Figures 1 and 2. Although `ASTDisplay` in Figure 2 provides expression classes extended with the ability to display themselves, this new functionality is not available for instances of the original classes in `AST`. This is unfortunate, because instances of the original classes might be created by existing library code or deserialized from a file. We can avoid this limitation by using class sharing as shown in Figure 3. Here, `shares` clauses cause the two families `ASTDisplay` and `AST` to share all the expression classes. Instances of `AST` expression classes are also instances of corresponding `ASTDisplay` expression classes.

Because of sharing, expression objects from the `AST` family can possess GUI display operations, even if the implementer of `AST` was not aware of `TreeDisplay` or `ASTDisplay`; client code—for example, a visualization toolkit, which expects `TreeDisplay` objects—would obtain the ability to handle existing `AST` objects. Thus every expression class in the `ASTDisplay` family becomes an *adapter* [19] for the corresponding class in `AST`; the `ASTDisplay` family provides *family adaptation* for `AST`.

Given a tree of expression objects from `AST`, family adaptation ensures that the whole tree is safely adapted to `ASTDisplay`. The adaptation preserves the original tree structure, and the relationships between objects in the tree. This contrasts with prior adaptation mechanisms that work on individual objects, which do not guarantee safety or the preservation of object relationships. With prior mechanisms such as the adapter design pattern [19], one might forget to adapt the left child of a `Binary` object, and the call `l.display()` on line 5 would fail.

In Figure 3, every expression class has a sharing declaration, which could be tedious to write. J&$_S$ provides the `adapts` clause as a shorthand for adding sharing declarations to all inherited member classes. For example, `ASTDisplay` may be declared with the following class header, without any individual sharing declarations:

```
class ASTDisplay extends ... adapts AST { ... }
```

2.3 Views and view changes

Views. If two classes are shared, a single object might be treated as an instance of either one. Each class is a distinct *view* of that object. A J&$_S$ object can have any number of views, all equally valid. This contrasts with an ordinary object-oriented language like Java (or even J&), where an object has exactly one view: its run-time class.

In J&$_S$, an object reference is not just a heap location; it is essentially a pair $\langle \ell, T \rangle$ of a heap location ℓ and a type T, where T is the view, represented as a non-dependent exact type. The view T determines the behavior of the object when accessed through that reference.

For example, the method `display` in Figure 3 cannot be directly called on an object created as an instance of the class `AST.Binary`, because the reference has the view `AST.Binary!`. However, when the object obtains a new reference—for example, by storing the object in a local variable—with the view `ASTDisplay.Binary!`, it also obtains a new behavior—the method `display` becomes available through the new reference. Moreover, methods that are available through the original reference might behave differently when they are called through the new reference. See Section 2.4 for an example.

View changes. J&$_S$ has a *view change* operation $(\mathtt{view}\ T)e$, which generates a new reference with the same heap location but a different view. On line 7 in Figure 3, the method `show` contains a view change expression $(\mathtt{view\ Exp})\mathtt{e}$, the result of which is a reference that still points to the same object as `e` but with a new view that is a subtype of `Exp`. (Recall from Section 2.1 that within `ASTDisplay`, `Exp` is sugar for `ASTDisplay[this.class].Exp`).

View changes support late binding. Although the expression (`view` T)e has a statically known type T, the actual run-time view of the result is a subtype of T that is shared with the run-time view of the value of e. For example, in line 7 of Figure 3, if `e` evaluates to a value with the view `AST.Value!`, then within the family `ASTDisplay`, the expression (`view Exp`)`e` produces a value with the view `ASTDisplay.Value!`, which is a subtype of `ASTDisplay!.Exp`, and shared with `AST.Value`.

A view change (`view` T)e looks syntactically like a type cast (T)e, and it does have the same static target type T, but it is actually quite different. First, a type cast might fail at run time, but view changes that type-check always succeed. Second, no matter whether it is an upcast or a downcast, an ordinary cast only works if the target type is a supertype of the *run-time* class of the object. However, a view change has in general a target type that is neither a supertype nor a subtype of the current view of the object, but from another family. For example, if families `A` and `B` share the class `C`, together with all its subclasses within each family, the following code is legal:

```
A!.C a = new A.C();
B!.C b = (view B!.C)a;
```

The initial view of the created object is `A.C!`. The target type `B!.C` in the view change is neither a supertype nor a subtype of `A.C!`. Of course, if the type in a view change is indeed a supertype of the current view of the instance, the view change is a no-op.

Third, a type cast checks run-time typing information, but has no other effects at run time. By contrast, a view change can affect the behavior of the object.

View-dependent types. Nested inheritance uses the dependent class p`.class` to indicate the family that the object referenced by p belongs to. J&$_S$ generalizes p`.class` to be *view-dependent*, since an object may be a member of multiple families. For example, suppose `ASTDisplay` contained the following code:

```
AST!.Binary a = new AST.Binary();
Binary b = (view Binary)a;
```

At run time, `a.class` would denote `AST.Binary!`, and `b.class` would denote `ASTDisplay.Binary!`. This example shows that in J&$_S$, the dependent classes associated with different aliases (`a` and `b` in this example) are not necessarily equal; they are interpreted as the views associated with the respective references.

Prefix types of dependent classes are also view-dependent, ensuring that late-bound type names belong to consistent families. Consider the left child of class `Binary`, stored in the field `l`. The type of the field is `Exp`, sugar for `AST[this.class].Exp`, which depends on the view of the object that contains the field. Accessing the left child with `a.l` returns an object in the family `AST`, whereas accessing it with `b.l` returns the same object, but with a view in the family `ASTDisplay`. In either case, the left child object and the containing `Binary` object have views that are in the same family. This means that for a tree of objects in one family, a single explicit view change on the root object effectively moves the whole tree to another family, implicitly triggering view changes on child objects as they are accessed.

2.4 Dynamic object evolution via view change

View-based dynamic method dispatching. In J&$_S$, method calls are dispatched on the current *view* of the receiver object, rather than on the receiver itself as in Java. When two references to the same receiver object have different views, the same method call may invoke different code. Therefore, when a J&$_S$ class overrides a method inherited from its declared shared class, both versions of the method become available to the object, and the choice is made at run time, based on the view associated with the reference.

```
package service;
class SomeService {
  void handle(Packet p) {...}
}
...
class Dispatcher {
  SomeService s;
  void dispatch(Packet p) {
    switch (p.kind) {
      case 0: s.handle(p);
      ...
    }
  }
}
```

```
package logService extends
  service;
class SomeService shares
  service.SomeService {...}
...
class Logger {...}
class Dispatcher shares
  service.Dispatcher {
  Logger logger;
  void dispatch(Packet p) {
    logger.log(...);
    ...
  }
}
```

Figure 4. Evolution of a network service package

Method dispatching in J&$_S$ differs from that of nonvirtual methods in C++ [44], and from the statically scoped adaptation in *expanders* [50]. For nonvirtual methods in C++, the static type of the receiver acts as a static view that selects the method to call. By contrast, the view change operation in J&$_S$ affects method dispatching, but still allows late binding, since the type T in a view change (`view` T)e is a supertype of the statically unknown run-time view.

Expanders dispatch methods within the same family dynamically, but unlike with class sharing, the choice between original code and expander code is static. Expander methods can be invoked only when expanders are in scope, and therefore the behavior of existing code written before the expanders cannot change. Since expansion is *not* inheritance, expander methods can only *overload* original methods rather than *overriding* them.

Dynamic object evolution. View-based method dispatching enables a new form of dynamic object evolution, in which existing objects are updated with different behavior without breaking running code. It is more powerful than object adaptation, which generally only adds new behavior to existing objects, whereas evolution can also make objects change behavior, even in a context that does not mention the updated classes. J&$_S$ supports evolution at the family level, evolving interacting objects consistently to the new family.

For example, Figure 4 shows a package `service` that implements several network services, and a dispatcher that calls different services based on the kind of the received packet. The server code has a static field storing the dispatcher, and an event loop:

```
static service.Dispatcher disp;
...
while (true) { ... disp.dispatch(p); ... }
```

Suppose the system implemented in the `service` family has started running, and then an updated package `logService` is developed that extends `service` with logging at various places (Figure 4 only shows the additional logging in the `dispatch` method). The goal is to update the system with logging ability, without having to stop it from running.

To evolve the system from `service` to `logService`, the view of the dispatcher object stored in the static field `disp` needs to be changed. This could be done in initialization code in the extended package, as follows:

```
service!.Dispatcher d =
  (service!.Dispatcher)Server.disp; // cast
Server.disp = (view Dispatcher)d;   // view change
```

After this view change, the `dispatch` method overridden in the extended package will be called.

More importantly, although just a single explicit view change is applied to the dispatcher object, all the other objects transitively reachable from the dispatcher, such as through the field `s` of type

`SomeService`, will also obtain new views when they are accessed, resulting in using the versions of their methods that have logging enabled. Thus, a single explicit view change causes a consistent evolution of many objects to the new family. This kind of evolution is simpler to implement and likely to be more efficient than going through all the objects and updating them individually.

2.5 Sharing constraints

Sharing relationships between types are not always preserved by derived families, either by design or as the result of changed class hierarchy in the derived family. For example, in a family inherited from `ASTDisplay`, one might choose not to share any class, or to share classes from `TreeDisplay`, and in either case, the new family no longer shares classes with the `AST` family.

Therefore, a view-change operation that works in the base family might not make sense in the derived family. J&$_s$ does not try to check all inherited method code for inapplicable view changes, but rather makes checking modular via *sharing constraints*. A J&$_s$ method can have sharing constraints of the form `sharing` $T_1 = T_2$, which means that any value of type T_1 can be viewed as of type T_2, and vice versa; in other words, the sharing relationship $T_1 \leftrightarrow T_2$ may be assumed in the method body. A view change can only appear in a method with an enabling sharing constraint. Outside the scope of sharing constraints, the type checker (and programmer) need not be concerned with sharing. Therefore, reasoning about class sharing is local.

For example, in Figure 3, the method `show` has a sharing constraint `AST!.Exp = Exp` (line 6), which allows the view change (`view Exp`)`e` to be applied to the variable `e` of static type `AST!.Exp`.

To know statically that the view change (`view Exp`)`e` will succeed, we must know that every subclass of `AST!.Exp` has a corresponding shared subclass under `ASTDisplay!.Exp`. J&$_s$ requires that some prefix of each type in the constraint is exact, and either non-dependent or only dependent on the path `this`; thus, we can check all the subclasses in a locally closed world (Section 2.1), without a whole-program analysis. In this example, the exact prefixes in question are `AST!` and `ASTDisplay[this.class]`.

The type checker verifies that sharing constraints in the base family still hold in the derived family; base family methods whose sharing constraints do not hold must be overridden.

Although sharing constraints support modular type checking, they do introduce an annotation burden for the programmer. Our experience suggests that the annotation burden is manageable. While it appears possible to automatically infer sharing constraints, by inspecting the type of the source expression and the target type of every view change operation in the method body, we leave this to future work.

3. Protecting unshared state

3.1 Unshared fields

In J&$_s$, shared classes do not necessarily share all their fields. Unshared fields are important for greater extensibility, but they pose challenges for the safety of the language. J&$_s$ uses *masked types* [35] and duplicate fields to ensure safety in the presence of unshared fields.

Figure 5 illustrates the two kinds of unshared fields. First, new object fields may be introduced by shared classes in the derived family. Because these fields do not exist in the base family, they cannot be shared. In Figure 5, classes `A1.B` and `A2.B` are shared, but `A2.B` introduces a new field `f` that does not exist in `A1.B`. When a view change from the base family to the derived family—for example, from `A1!.B` to `A2!.B`—is applied, the new field (e.g.,

```
class A1 {
  class B { }
  class C { D g; }
  class D { }
}
class A2 extends A1 {
  class B shares A1.B {
    T f;                    // a new field
  }
  class C shares A1.C\g { } // shared with a mask
  class E extends D { }     // a new subclass of D
}
```

Figure 5. Shared classes with unshared fields

the field `f` in Figure 5) would be uninitialized, which may be unexpected to the code in the derived family.

J&$_s$ uses masked types to prevent the possibly uninitialized new field from being read. A masked type, written $T\backslash$`f`, is the type T without read access to the field `f`. We say that the field `f` is *masked* in $T\backslash$`f`. The mask on a field can be removed with an assignment to that field. Masked types introduce the subtyping relationship $T \leq T\backslash$`f`. For the example in Figure 5, a view change from `A1!.B` to `A2!.B` must have a mask on the target type:

```
A1!.B b1 = new A1.B();
A2!.B\f b2 = (view A2!.B\f)b1;
```

Therefore, after this view change, the field `f` of `b2` cannot be read before it is initialized.

The second kind of unshared field is those with unshared types. In Figure 5, the field `g` has type `D`, and it cannot be shared, because in the `A2` family, `g` might store an object of class `E`. When a view change is applied from the derived family to the base family (for example, from `A2!.C` to `A1!.C`), an `A2.E` object stored in `g` would not have a view compatible with the base family.

Therefore, J&$_s$ requires that fields with unshared types are masked in the sharing declaration. A duplicate field with the same name for the shared class is also generated automatically in the derived family. For example, in Figure 5, it is as if the class `A2.C` has its own implicit declaration of field `g`:

```
class C shares A1.C\g {
  D g;
}
```

An instance of `A1.C` and `A2.C` contains two copies of the field `g`, each appearing as a "new" field to the other class. Which copy is accessed depends on the current view of the instance through which the field `g` is accessed. Field duplication prevents objects of an unshared class from being accidentally accessed in other families. Therefore, interacting objects always have consistent views.

When shared classes have duplicate fields, it is up to the programmer how to keep the copies in sync, or even whether to keep them in sync. The programmer may choose to construct a corresponding object in the target family, storing it in the duplicate field, as in the example in Section 3.2, or just to leave the field masked in the target family.

3.2 In-place translation with unshared classes

J&$_s$ supports in-place translation of data structures between families in which not all the classes are shared (translation is trivial if all classes are shared). As the data structure is translated, some objects in the structure may remain the same, with only a view change, and other objects, particularly those of unshared types, are explicitly translated. One use of this kind of translation is for compilers, where the data structure is an abstract syntax tree, and many parts of the AST do not need to change during a given compiler pass.

```
package base;                package pair extends base;
abstract class Exp           abstract class Exp
{ ... }                        { abstract base!.Exp translate
class Var extends Exp {          (Translator v); }
  String x;                  class Pair extends Exp
  ...                          { Exp fst, snd; ...}
}                            class Translator {
class Abs extends Exp {        base!.Abs reconstructAbs
  String x;                      (Abs old, String x,
  Exp e;                          base!.Exp exp) { ... }
  ...                          ...
}                            }
```

Figure 6. Lambda calculus and pair compiler extension

```
1  package pair extends base;
2  abstract class Exp shares base.Exp {
3    abstract base!.Exp translate(Translator v);
4  }
5  class Var extends Exp shares base.Var { ... }
6  class Abs extends Exp shares base.Abs\e {
7    base!.Exp translate(Translator v) {
8      base!.Exp exp = e.translate(v);
9      return v.reconstructAbs(this, x, exp);
10   }
11 }
12 class Pair extends Exp {
13   Exp fst, snd;
14   base!.Exp translate(Translator v) {
15     return new ...; // (λx.λy.λf. f x y) ⟦fst⟧ ⟦snd⟧
16   }
17 }
18 class Translator {
19   base!.Abs reconstructAbs(Abs old, String x,
20                            base!.Exp exp)
21   sharing Abs\e = base!.Abs\e {
22     if (old.x == x && old.e == exp) {
23       base!.Abs\e temp = (view base!.Abs\e)old;
24       temp.e = exp;
25       return temp;
26     }
27     else return new base.Abs(x, exp);
28   }
29   ...
30 }
```

Figure 7. In-place translation of the pair language

Figure 6 shows the skeleton of a simple compiler implemented with family inheritance but without class sharing, modeled on the Polyglot extensible compiler framework [29]. This example translates the λ-calculus extended with pairs, into the ordinary λ-calculus. The package `base` defines the target language through class declarations for AST nodes of the λ-calculus (e.g., `Abs` for λ abstractions); the package `pair` extends the source language with one additional AST node `Pair`. Classes inherited from `base` are further bound in `pair` with `translate` methods that recursively translate an AST from `pair` to `base`. The class `Translator` provides the methods AST classes use to (re)construct nodes in the target family using translated versions of their child nodes; the method `reconstructAbs` does this for λ abstractions. However, without sharing, the translation from `pair` to `base` has to recreate the whole AST, even for trivial cases like `Var`, because the two families `base` and `pair` are completely disjoint despite their structural similarity.

By contrast, Figure 7 shows the J&$_s$ code that does in-place translation. The `base` family remains the same as in Figure 6, and is omitted. Classes `Var` and `Abs` in `pair` are declared to share corresponding classes in `base` (lines 5–6). `Pair` is not shared, because it does not exist in `base`.

Consider the two shared classes `pair.Abs` and `base.Abs`. The type of their field `e`, which is `base[this.class].Exp`, is interpreted as `pair!.Exp` and `base!.Exp` in the two families. The two interpreted field types are not shared, because a value of type `pair!.Exp` might have run-time class `pair.Pair`, which has no corresponding shared class in the `base` family. Therefore the two `Abs` classes each have their own copies of the field `e`, and the sharing declaration on line 6 has a mask on `e`.

Similarly, the sharing constraint on line 21 also has masks, and so does the view change operation on line 23, where the `Abs` instance is reused if all its subexpressions have been translated in place. The sharing constraint, together with the corresponding view change operation, has a mask on the field `e`, in case it points to an instance of an unshared subclass of `Exp`, such as `Pair`. The mask is removed on line 24, after which the type of the variable `temp` becomes `base!.Abs`.

As shown in the above example, J&$_s$ uses masked types to prevent objects of unshared types, such as `Pair`, from being leaked into an incompatible family (here, `base`). Masked types do introduce some annotation burden. However, class sharing is expected to be used in practice between families that are similar to each other, where extensibility and reuse are needed and make sense. In that case, there should not be many masks.

3.3 Translation from the base family to the derived family

Translations in different directions are not always of the same difficulty. Section 3.2 proposes a solution for in-place translation from the derived family (`pair`) to the base family (`base`). The complexity of the solution arises mostly because the objects of the `Pair` class must be translated away. However, as noted in [13], in-place translation in the other direction, that is, from `base` to `pair`, should be almost trivial, by simply treating an AST in `base` as an AST in `pair`.

To capture the asymmetry, J&$_s$ supports *directional* sharing relationships between types that are possibly masked, represented as $T_1 \rightsquigarrow T_2$, which means that an object of static type T_1 may be applied a view change with target type T_2. In Figure 7, the J&$_s$ compiler may infer the sharing relationship `base!.Exp` $\rightsquigarrow$ `pair!.Exp`, (the other direction `pair!.Exp` $\rightsquigarrow$ `base!.Exp` does not hold, because of the class `Pair`), and allows a constant-time in-place translation from `base` to `pair`, by a view change from `base!.Exp` to `pair!.Exp`.

Note that in this case, the sharing declaration on line 6 in Figure 7 actually induces the following two directional sharing relationships:

$$\begin{array}{l}\texttt{base.Abs!} \rightsquigarrow \texttt{pair.Abs!} \\ \texttt{pair.Abs!\textbackslash e} \rightsquigarrow \texttt{base.Abs!\textbackslash e}\end{array}$$

rather than just a bidirectional sharing relationship `base.Abs!\e` $\leftrightarrow$ `pair.Abs!\e`.

4. Type safety

The key aspects of the J&$_s$ language are formalized in an object calculus. This calculus has some similarities with other formal semantics that support family inheritance, e.g., Jx [28], Tribe [10], and *vc* calculus [18], and especially the J& calculus [30, 31]. The most notable differences come from adding sharing. The soundness of the calculus is proved using subject reduction and progress [51]. Due to the space limit, this section only sketches the formalization of the language and the proof. See the companion technical report [36] for details.

4.1 Preliminaries

A J&$_s$ program Pr is defined as a set of class declarations $\overline{L}$ and an expression e, modeling the `main` method. A class declaration L contains a superclass declaration T_1, a shared class declaration T_2, nested classes $\overline{L}$, fields $\overline{F}$, and methods $\overline{M}$:

$$L ::= \texttt{class}\ C\ \texttt{extends}\ T_1\ \texttt{shares}\ T_2\ \{\overline{L}\ \overline{F}\ \overline{M}\}$$

The shared class declaration may include a set of masks $\overline{f}$ on unshared fields. If the class is not shared with any other class, it is declared to share with itself.

Every method is declared with a (possibly empty) set of sharing constraints:

$$M ::= T\ m(\overline{T}\ \overline{x})\ \texttt{sharing}\ \overline{T_1} \rightsquigarrow \overline{T_2}\ \{e\}$$

The semantics uses directional sharing constraints and relationships, because view changes are directional. Inference of directional sharing relationships (see Section 3.3) is outside the scope of this paper and is not formalized here.

Expressions in J&$_s$ are mostly standard, with the addition of the view change expression $(\texttt{view}\ T)e$:

$$\begin{array}{rl} e & ::= v \mid x \mid e.f \mid x.f = e \mid e_0.m(\overline{e}) \mid e_1;\ e_2 \\ & \mid \texttt{new}\ T \mid \texttt{final}\ T\ x = e_1;\ e_2 \\ & \mid (\texttt{view}\ T)e \end{array}$$

Values v (that is, references) in J& are pairs $\langle \ell, S\rangle$ of a heap location ℓ and a view, represented as a non-dependent type S. Every field is assumed to have an initializer, and therefore object creations `new` T do not provide initial values for fields. With the help of masked types, a distinguished `null` value is not needed, simplifying the calculus.

4.2 Operational semantics

J&$_s$ is modeled with a small-step operational semantics. A configuration consists of an expression e, a stack σ, a heap H, and a reference set R, and evaluation takes the form $e, \sigma, H, R \longrightarrow e', \sigma', H', R'$. The stack σ maps variables x to values v. The heap H is a function mapping tuples $\langle \ell, P, f\rangle$ of memory locations, fully qualified class names, and field names to values v. The class P is included in the domain to distinguish copies of duplicate fields, which have the same name. The reference set R collects all the references r that have been generated during evaluation, and is only used by the soundness proof.

Most evaluation rules are straightforward. The two notable ones are R-SET and R-VIEW for field assignments and view changes, respectively:

$$\frac{\begin{array}{c}\sigma(x) = \langle \ell, P!\backslash\overline{f'}\rangle \quad \sigma' = \mathsf{grant}(\sigma, x.f) \\ H' = H[\langle \ell, \mathsf{fclass}(P,f), f\rangle := v] \\ R' = R, \langle \ell, P!\backslash(\overline{f'} - f)\rangle\end{array}}{x.f = v, \sigma, H, R \longrightarrow v, \sigma', H', R'} \quad (\text{R-SET})$$

$$\frac{R' = R, \mathsf{view}(v, S)}{(\texttt{view}\ S)v, \sigma, H, R \longrightarrow \mathsf{view}(v,S), \sigma, H, R'} \quad (\text{R-VIEW})$$

A field assignment $x.f = v$, where $\sigma(x) = \langle \ell, S\rangle$, not only updates the heap, but also removes the mask on f, if any, in the type S. The auxiliary function grant captures the mask removal, and $\mathsf{fclass}(P,f)$ finds the shared class of P that introduces the field f, or the current copy of f, if it is a duplicate field.

A view change $(\texttt{view}\ S)\langle \ell, S'\rangle$ generates a new value $\langle \ell, S''\rangle$ with a new view S'' compatible with S. The auxiliary function view implements the view change operation, and is defined as follows:

$$\mathsf{view}(\langle \ell, P'!\backslash\overline{f'}\rangle, PS\backslash\overline{f}) = \begin{cases} \langle \ell, P'!\backslash\overline{f}\rangle & \vdash P'!\backslash\overline{f'} \le PS\backslash\overline{f} \\ \langle \ell, P!\backslash\overline{f}\rangle & \exists! P\ .\ \exists\overline{f''} \supseteq \overline{f'}\ . \\ & \quad \vdash P! \le PS\ \wedge \\ & \quad \vdash P'!\backslash\overline{f''} \rightsquigarrow P!\backslash\overline{f} \end{cases}$$

4.3 Static semantics

Type checking is performed in a typing context Γ, which is a set of variable type bindings $x : T$, path equivalence constraints $p_1 = p_2$, and directional sharing constraints $T_1 \rightsquigarrow T_2$. Path equivalence constraints are used to assert equivalence of dependent types during evaluation, and unlike previous calculi on family inheritance, paths with different views are not included, even if they are actually aliases of the same location (see Section 2.3). Sharing constraints are declared by the current method being checked, and are used to type-check view change operations.

Since assignments may remove masks, the J&$_s$ type system is flow-sensitive. Typing judgments are of the form $\Gamma \vdash e : T, \Gamma'$, where Γ' represents the updated typing environment after evaluating e.

A view change expression $(\texttt{view}\ T)e$ type-checks, if the type of the source expression e is shared with the target type T:

$$\frac{\Gamma \vdash e : T', \Gamma' \quad \Gamma \vdash T' \rightsquigarrow T}{\Gamma \vdash (\texttt{view}\ T)e : T, \Gamma'} \quad (\text{T-VIEW})$$

The directional sharing judgment $\Gamma \vdash T_1 \rightsquigarrow T_2$ states that a value of type T_1 can be transformed to T_2 through a view change. A bidirectional sharing judgment $\Gamma \vdash T_1 \leftrightarrow T_2$ is sugar for a pair of directional sharing judgments.

Figure 8 shows all the derivation rules for sharing judgments. In SH-DECL, auxiliary function $\mathsf{share}(P.C)$ returns the declared shared type for class $P.C$, and $\mathsf{fnames}(\mathsf{fields}(P.C) - \mathsf{fields}(P'))$ gives all the new fields that are not contained in the shared superclass P', which are also masked in the generated sharing relationship, in addition to fields $\overline{f}$ that are masked in the sharing declaration.

SH-CLS states that for two types to be shared, every subclass of the source type must share a unique subclass of the target type, with appropriate masks. The uniqueness is necessary for the auxiliary view function to set the type component in the generated reference. Moreover, the two types in SH-CLS have to be both nested in simple exact types, and therefore we can enumerate all their subclasses in the locally closed worlds.

The type system clearly distinguishes sharing relationships between types from subtyping relationships.

The subtyping judgment $\Gamma \vdash T_1 \le T_2$ means that a value of type T_1 can be directly used as a value of T_2. Subtyping rules are basically the same as in the J& calculus, with the addition of S-MASK and S-SUB-MASK:

$$\Gamma \vdash T \le T\backslash f \ (\text{S-MASK}) \qquad \frac{\Gamma \vdash T_1 \le T_2}{\Gamma \vdash T_1\backslash f \le T_2\backslash f} \ (\text{S-SUB-MASK})$$

Subtyping rules in J&$_s$ do not depend on any sharing judgment. In particular, $\Gamma \vdash T_1 \rightsquigarrow T_2$ does not imply $\Gamma \vdash T_1 \le T_2$.

Decidability. The static type system of J&$_s$ is decidable. First, most of the typing rules are syntax-directed, except for one that is based on subtyping. Any subtyping relationship can be derived using a given type at most once, and a finite number of types can appear in the derivation. The number of types that can appear in a derivation is finite because nesting depth is bounded, and the static typing environment contains no path equivalence constraints. Second, the flow-sensitive checking of masks reaches a fixed point in finite steps, because there are only a finite number of fields in a class to mask.

$$\Gamma \vdash T \rightsquigarrow T \ (\text{SH-REFL}) \qquad \frac{\Gamma \vdash T_1 \rightsquigarrow T_2 \quad \Gamma \vdash T_2 \rightsquigarrow T_3}{\Gamma \vdash T_1 \rightsquigarrow T_3} \ (\text{SH-TRANS}) \qquad \frac{T_1 \rightsquigarrow T_2 \in \Gamma}{\Gamma \vdash T_1 \rightsquigarrow T_2} \ (\text{SH-ENV}) \qquad \frac{\Gamma \vdash T_1 \rightsquigarrow T_2}{\Gamma \vdash T_1 \backslash f \rightsquigarrow T_2 \backslash f} \ (\text{SH-MASK})$$

$$\frac{\begin{array}{c}\mathsf{share}(P.C) = P' \backslash \overline{f} \\ \mathsf{fnames}(\mathsf{fields}(P.C) - \mathsf{fields}(P')) = \overline{f'}\end{array}}{\Gamma \vdash P.C! \backslash \overline{f} \backslash \overline{f'} \leftrightarrow P'! \backslash \overline{f}} \ (\text{SH-DECL}) \qquad \frac{\begin{array}{c} PS_1 = P'_1!.\overline{C_1} \quad PS_2 = P'_2!.\overline{C_2} \\ \forall P_1.\ \Gamma \vdash P_1! \leq PS_1 \Rightarrow (\exists! P_2\ .\ \exists \overline{f''} \supseteq \overline{f}\ .\ \Gamma \vdash P_2! \leq PS_2 \wedge \Gamma \vdash P_1! \backslash \overline{f''} \rightsquigarrow P_2! \backslash \overline{f'}) \end{array}}{\Gamma \vdash PS_1 \backslash \overline{f} \rightsquigarrow PS_2 \backslash \overline{f'}} \ (\text{SH-CLS})$$

Figure 8. Rules for sharing judgments

4.4 Soundness

The soundness theorem is standard:

THEOREM 4.1. *(Soundness) If* $\vdash \langle \overline{L}, e \rangle$ ok, *and* $\emptyset \vdash e : T$, *and* $e, \emptyset, \emptyset, \emptyset \longrightarrow^* e', \sigma, H, R$, *then either* $e' = v$ *and* $\lfloor \sigma, H, R \rfloor \vdash v : T$, *or* $\exists e'', \sigma', H', R'\ .\ e', \sigma, H, R \longrightarrow e'', \sigma', H', R'$.

In the theorem, $\lfloor \sigma, H, R \rfloor$ represents the typing environment extracted from a run-time configuration.

The proof is by showing subject reduction and progress, which are proved respectively by induction on appropriate typing derivations and structural induction on expressions.

5. Implementation

We have implemented a prototype compiler of J&$_s$ in the Polyglot framework [29]. The compiler is an extension based on the latest implementation of the Jx/J& compiler [28, 30], which is itself an extension of the Polyglot base Java compiler. The J&$_s$ compiler extension has 5,200 lines of code, excluding blank lines and comments. J&$_s$ also has a run-time system that supports implicit classes and provides machinery for view changes and run-time type inspection. The run-time system is written in Java, and has 1,500 lines of code, most of which implements a custom classloader, using the ASM bytecode manipulation framework [8].

J&$_s$ is implemented as a translation to Java. The amount of code produced by the compiler is proportional to the size of the source code. The compiler generates class declarations only for explicit classes. For implicit classes, the custom classloader lazily synthesizes classes representing them at run time.

5.1 Type checking

Because of masks, J&$_s$ has a flow-sensitive type system. Every method is checked in two phases. The first phase is flow-insensitive, ignoring masks, and generates typing information necessary for building the control-flow graph. The second phase is essentially an intraprocedural data-flow analysis, which computes a type binding, possibly with masks, for each local variable (including `this`) at every program point.

5.2 Translating classes

The translation is similar to but more efficient than the J& translation described in [30], primarily because of the custom classloader.

First, masks are erased after type checking. Then, every explicit J&$_s$ class is translated into several Java classes and interfaces, among which the most important ones are the *instance class* and the *class class*. The instance class contains all the object fields of the J&$_s$ class. At run time, each object of a J&$_s$ class is represented as an object of the instance class. The class class contains method implementations, static fields, and type information needed to implement casts, `instanceof`, and prefix types. The class class also includes getter and setter methods for all object fields, since unshared fields may be duplicated, and therefore, field accesses are also view-dependent. Various interfaces are generated for simulating multiple inheritance.

An implicit class also has an instance class and a class class, both synthesized by the custom classloader. The instance class collects all object fields from superclasses. The class class contains a dispatch method for each inherited J&$_s$ method. The dispatch method calls the appropriate method implementation in the class class of some explicit J&$_s$ class, identified by the classloader.

For each set of shared J&$_s$ classes, the classloader maintains a representative instance class that collects all the object fields, including the duplicate ones. At run time, objects of all these shared classes are created as instances of the representative instance class. When a new shared class that contains more object fields is loaded, the classloader updates the representative instance class to include the new fields. All the existing objects of the old representative instance class will be lazily converted to the most up-to-date representative instance class. The conversion works because objects of instance classes are referenced indirectly, as described in Section 5.3.

5.3 Supporting views and view changes

A J&$_s$ object can be an instance of several shared classes, and every reference to the object could have a different view. Therefore, the J&$_s$ implementation adds a level of indirection. Each object is referenced indirectly through a *reference object*, containing two fields: one points to the instance class object; and the other points to a class class object, which is the view associated with the reference.

The view determines the behavior of the translated J&$_s$ object. Method calls are dispatched on the view rather than on the object itself. Prefix types are evaluated with the view. Casts and `instanceof`s check the view. Which copy of a duplicated field is accessed also depends on the view.

A view change operation $(\texttt{view}\ T)e$ is translated to generating a new reference object with the same instance class object, and a view compatible with T. The implementation memoizes the result to the most recent view change operation on any reference object, to avoid repeatedly generating the same reference object.

A J&$_s$ object might obtain a new view implicitly. Moving an object from one family to another would implicitly move all the other objects that are transitively reachable through shared fields. Implicit view changes are carried out lazily, only at the time when objects are accessed through fields.

Reference objects add some overhead for accessing members of an object. With a lower-level target language, a probably faster implementation is possible in which different object views are represented as different pointers into the same object, like the C++ implementation of multiple inheritance. Since methods are dispatched on views, the object would contain pointers to several dispatch tables, one for each view. Method dispatch should then have performance similar to C++ virtual method calls.

5.4 Java compatibility

J&$_s$ is mostly backward-compatible with Java. Any Java code with no nested classes is also legal J&$_s$ code. J&$_s$ programs may use existing Java code, including libraries that are compiled by a Java compiler. Of course, precompiled Java code does not enjoy the benefit of class sharing. The J&$_s$ reference object forwards method calls to `hashCode`, `equals`, etc., and therefore J&$_s$ objects may be passed to Java code, for example, to be stored in a `HashSet`. There is one limitation in the current implementation: a J&$_S$ class cannot be shared if it inherits from any Java superclass other than `java.lang.Object`, or if it implements most Java interfaces. This is the artifact of using reference objects to support views; a lower-level implementation should not have this limitation.

The J&$_S$ language currently extends Java 1.4. It does not support generics, which seem to be an orthogonal feature. We leave the interaction between generics and class sharing to future work.

5.5 Concurrency

Class sharing is compatible with multi-threaded code. The J&$_S$ compiler ensures that in the generated code, `synchronized` statements and methods are synchronized on the wrapped instance class object, rather than on the reference object. The implementation of the reference object also relays method calls to `wait`, `notify` and `notifyAll` to the underlying instance class object. A view change operation usually just creates a new reference object that contains a reference to an existing instance class object, without affecting existing references, and therefore requires no synchronization. The only operation that needs to be synchronized is the lazy conversion of an object to the most up-to-date instance class.

6. Experience

6.1 Jolden benchmarks

We tested the J&$_S$ implementation with the jolden benchmarks [9] to study the performance overhead for code that does not use the new extensibility features of J&$_S$. All ten benchmarks, with few changes, are tested with four language implementations: Java, J& as described in [30], J& with a classloader similar to that described in Section 5.2, and J&$_S$. Table 1 compares the results. The testing hardware is a Lenovo Thinkpad T60 with Intel T2600 CPU and 2GB memory, and the software environment consists of Microsoft Windows XP, Cygwin, and JVM 1.6.0_07.

Table 1 shows that the use of a custom classloader greatly improves the performance, comparing the two implementations of J&. Unsurprisingly, nested inheritance and class sharing do introduce overhead. The J&$_S$ times show a 37% slowdown versus the classloader-based J& implementation, and 94% versus the highly optimized Java HotSpot VM. The overhead seems reasonable, especially considering that the current implementation works as a source-to-source translation to Java, precluding many optimizations and implementation techniques. We expect that a more sophisticated implementation could remove much of the overhead.

Running programs in J&$_S$ have the latent capability to be extended in many ways, so it is not surprising that there is some performance cost. But it seems software designers are often willing to pay a cost for extensibility, because they often add indirections and use design patterns that promote extensibility but have run-time overhead. We believe that J&$_S$ removes the need for many such explicit extensibility hooks, while making code simpler. For systems where extensibility is more important than high performance [46], the existing implementation may already be fast enough.

6.2 Tree traversal

The jolden benchmarks do not use the new features provided by J&$_S$. To study the performance of view changes, especially on large data structures, we wrote a small benchmark in which two families share classes that implement binary trees. A complete binary tree of a given height is first created in the base family, and an explicit view change is applied to the root of the tree. A depth-first traversal is carried out to trigger all the lazy implicit view changes. The testing environment is the same as in Section 6.1.

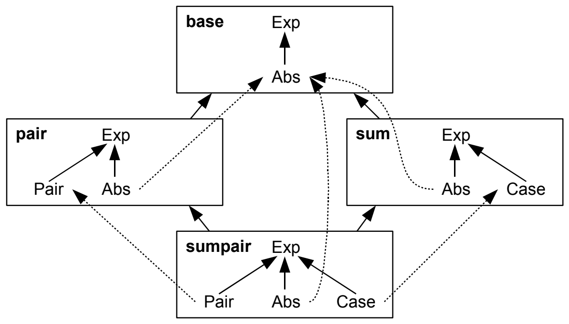

Figure 9. Lambda compiler structure. Translator and some AST nodes not shown.

Table 2 summarizes the results. In-place adaptation, even with a traversal that triggers all the implicit view changes, is faster than an explicit translation that creates new objects in the derived family, and the running time is also close to that of the initial creation of the tree. The fourth row shows that once the implicit view changes are complete and the new reference objects have been memoized, traversals execute as fast as a traversal before the view changes. This benchmark does a complete traversal. Since view changes are lazily triggered, the relative performance of adaptation would look even better if not all nodes needed to be visited after adaptation.

6.3 Lambda compiler

The J&$_S$ implementation successfully compiles and executes the completed version of the lambda compiler shown in Figure 6, which compiles λ-calculus enhanced with sums and pairs down to simple λ-calculus.

The structure of the lambda compiler is illustrated in Figure 9. Solid arrows represent class and family inheritance, and dashed arrows represent sharing declarations.

The lambda compiler has a `base` family with classes representing AST nodes for simple λ-calculus. There are two families directly derived from the base family, extended with sums and pairs respectively. The `sum` family and the `pair` family each share AST classes from the `base` family, and each implement in-place translation to simple λ-calculus, as shown in Figure 6. The last derived family `sumpair` composes the `sum` family and the `pair` family, leading to a compiler that supports both sums and pairs. The `sumpair` family shares `Abs` and other AST classes from the simple λ-calculus with the `base` family, and by transitivity, with `sum` and `pair`. The code of `sumpair` just sets up the sharing relationships, without a single line of translation code. The in-place translation code from `sum` and `pair` is composed to translate away sums and pairs.

This program is about 250 lines long, but uses the features of J&$_S$ in a sophisticated way. For example, families have both shared and unshared classes, masked types are used to ensure objects of new AST classes (pair and sum) are translated away, and the two translations are composed to translate pairs and sums at once. The lambda compiler example is inspired by the Polyglot framework, and it encapsulates most of the interesting issues that arise in making Polyglot extensible. The results suggest Polyglot would be simpler in J&$_S$, but this is left for future work.

6.4 CorONA

Our second significant example shows that the adaptation capability of J&$_S$ can be used to seamlessly upgrade a running server and its existing state with entirely new functionality. We ported CorONA, an RSS feed aggregation system [37], to J&$_S$, and suc-

	bh	bisort	em3d	health	mst	perimeter	power	treeadd	tsp	voronoi
Java	1.74	0.53	0.17	0.41	0.88	0.22	1.00	0.14	0.12	0.31
J& [30]	13.91	1.77	0.48	8.45	4.43	2.82	2.43	2.20	0.37	7.19
J& with classloader	2.02	0.71	0.22	0.71	1.06	0.39	1.14	0.21	0.16	0.59
J&$_s$	2.61	0.88	0.23	1.61	1.54	0.47	1.27	0.45	0.17	0.83

Table 1. Results for the jolden benchmarks. Average time over ten runs, in seconds.

Tree height	16	18	20
Tree creation	0.110	0.287	1.295
Traversal before view changes	0.008	0.027	0.105
View changes	0.125	0.367	1.303
Traversal after view changes	0.006	0.025	0.099
Explicit translation	0.145	0.622	1.669

Table 2. Comparing view changes with explicit translation. Average time over ten runs, in seconds.

cessfully used class sharing to update the system at run time with a new caching algorithm.

CorONA is originally an extension of Beehive [38], which is itself an extension of Pastry [40] that provides a distributed hash table. Beehive extends Pastry with active replication, whereas PC-Pastry [38] extends Pastry with passive caching. We refactored the ported CorONA, and composed it with PC-Pastry and Beehive respectively, creating two applications (named PCCorONA and BeeCorONA) with different caching strategies. The system is tested by first running PCCorONA for a while, and then evolving the running system from passive caching to active caching by compiling and loading a new package, BeeCorONA.

Sharing declarations are added so that classes representing host nodes, data objects, and network addresses are shared between CorONA and its two derived families. Classes for network messages and cache management are not shared. The evolution code goes over all the host nodes, which are the top-level objects in the system, changing their views, and creating new caching managers.

The amount of code to implement evolution is relatively small (less than 40 lines of code, compared to 8300 for the whole system, excluding comments and empty lines). Very little code is needed because we only need to change the views of host nodes; all the other referenced objects will have their views changed implicitly when they are accessed. In the host node class, fields storing the caching managers are not shared, and masked types ensure that they are initialized in the evolved system. With a slightly different configuration, we can actually run the two variants of the system at the same time, using the same set of host node objects. View-dependent types ensure that network messages of correct versions are created and accepted for each of the systems.

7. Related work

Adaptation. The Adapter design pattern [19] is a protocol for implementing adaptation. However, this and other related patterns are tedious and error-prone to implement, rely on statically unsafe type casts, and do not preserve object identity or provide bidirectional adaptation as J&$_s$ does.

Expanders [50] are a mechanism for adaptation. New fields, methods, and superinterfaces can be added into existing classes. Expanders are more expressive than *open classes* [11], which can only add new methods. Method dispatch is statically scoped, so expanders do not change the behavior of existing clients. New state is added by wrapper classes; a map from objects to wrappers ensures uniqueness of wrapper instances.

CaesarJ [26, 1] is an aspect-oriented language that supports adaptation with *wrappers* called *aspect binders*. Wrappers and expanders are similar. They both can extend wrapped classes with new states, operations, and superinterfaces; no duplicate wrappers are created for objects; and dynamic wrapper selection is similar to expander overriding. Wrappers in CaesarJ are less transparent: a wrapper constructor must be called to get a wrapper instance, whereas expander operations can be applied directly to objects. Multiple inheritance in CaesarJ makes wrapper selection ambiguous; J&$_s$ disambiguates via views.

Both expanders and CaesarJ wrappers share limitations: it is impossible to override methods in the original family, and therefore there is no dynamic dispatching across families; object identity is not preserved; the use of expanders and wrappers is limited to adaptation, since the adapter family cannot be used independently of its original family. Class sharing in J&$_s$ provides more flexibility.

The Fickle$_{\text{III}}$ [14] language has a *re-classification* operation for objects to change their classes. Re-classifications are similar to view changes in J&$_s$, but directly change the behavior of all existing references to the object; therefore, effects are needed to track the change. A re-classification might leave fields uninitialized, while masked types in J&$_s$ ensure that after a view change, fields are initialized before use. Fickle$_{\text{III}}$ does not support class families.

Chai$_3$ [42] allows traits to be dynamically substituted to change object behaviors, similar to view changes in J&$_s$. However, *Chai*$_3$ does not support families, and the fact that traits do not have fields makes it harder to support manipulation of data structures.

Some work on adaptation, including pluggable composite adapters [27], object teams [20], and delegation layers [34], also has some notion of families of classes. However, family extensibility does not apply to the relationship between the family of adapter classes and the family of adaptee classes. Therefore, these mechanisms either do not support method overriding and dynamic dispatch between the adapter and adaptee families [27, 20], or have a weaker notion of families in which programmers have to manually "wire" inheritance relationships between the base family and the delegation family [34]. These mechanisms all use lifting and lowering, introduced in [27], to convert between adapter and adaptee classes. Lifting and lowering are similar to view changes in J&$_s$, but are not symmetric and do not support late binding.

Family inheritance. Several different mechanisms have been proposed to support family inheritance, including virtual classes, nested inheritance, variant path types, mixin layers, etc. In all these family inheritance mechanisms, families of classes are disjoint, whereas with class sharing, different families can share classes and their instances.

Virtual classes [25, 24, 16, 18, 10] are inner classes that can be overridden just like methods. Path-dependent types are used to ensure type safety. The soundness of virtual classes has been formally proved by Ernst et al. [18], and by Clarke et al. [10].

Nested inheritance [28] supports overriding of nested classes, which are similar to virtual classes. Nested intersection [30] adds and generalizes intersection types [39, 12] in the context of nested inheritance to provide the ability to compose extensions.

Virtual classes support *family polymorphism* [17], where families are identified by the enclosing instance. Nested inheritance supports what Clarke et al. [10] called *class-based family polymorphism*, where each dependent class defines a family of classes nested within and also enclosing it. With prefix types, any instance of a class in the family can be used to name the family.

Variant path types [22] support family inheritance without dependent types, using exact types and relative path types (similar to `this.class`) for type safety. These exact types are very different from those in J&$_S$: in a J&$_S$ exact type $A.B!.C$, exactness applies to the whole prefix before !, that is, $A.B$; in an exact type $A@B.C$ in [22], exactness applies to the simple type name right after @, that is, B.

Mixin layers [41] generalizes *mixins* [4]. Mixins are classes that can be instantiated with different superclasses, and mixin layers are mixins that encapsulate other mixins. Mixin layers support family inheritance: when a mixin layer is instantiated, all the inner mixins are instantiated correspondingly.

Virtual types [47, 7, 48, 21] are type declarations that can be overridden. Virtual types are more limited than virtual classes: they provide family polymorphism but not family inheritance. Scala [32, 33] supports family polymorphism and composition through virtual types, path-dependent types, and mixin composition. It also supports parametric polymorphism. Scala does not have virtual classes and does not support family inheritance. Scala has *views* that are implicitly-called conversion functions. Scala views do not provide adaptation: they create new instances in the target types.

Sharing in functional languages. Wadler's views [49] are an isomorphism between a new data type and an existing one, which is similar to a sharing declaration. Of course, there are obvious differences: Wadler's views are for a functional setting, and primarily relate abstract data types and types inductively defined with pattern matching, whereas J&$_S$ creates sharing relationships only between overridden and overriding classes.

The SML module system [23] has *sharing constraints*, which require functor module parameters to agree on type components. SML sharing constraints dictate applicability of functors; J&$_S$ sharing constraints dictate applicability of method bodies.

Safe dynamic software updating. There is prior work on the problem of safely updating software without downtime.

Barr and Eisenbach [2] propose a framework to support dynamic update of Java components that satisfy the binary compatibility requirement [45], which also includes a custom classloader. The goal is to provide a tool that keeps Java libraries up to date, rather than to improve the extensibility of the language.

Duggan [15] describes a calculus that combines Wadler's views and SML sharing constraints to support hot-swapping modules. J&$_S$ differs because it does not copy values across different versions; instead, it generates different views on the same object.

Proteus [43] finds proper timing for a given global update with static analysis. Abstract and concrete types in Proteus bear some resemblance to inexact and exact types in J&$_S$: abstractly typed variables allow values of different concrete types, and inexactly typed variables may store objects with different exact views.

8. Conclusions

This paper introduces class sharing, a flexible extensibility mechanism that combines the advantages of both family inheritance and adaptation. As several examples show, class sharing adds new capabilities such as family adaptation, dynamic object evolution, and in-place translation. These capabilities are supported by a variety of mechanisms. Dynamic views typed with dependent classes statically track the families of values; masked types ensure shared and unshared classes can be mixed safely. The language is proved sound, showing that the new extensibility provided by class sharing does not come at a price in type safety.

Acknowledgments

We thank the many people who have offered helpful suggestions on this work or its presentation, including Steve Chong, Michael Clarkson, Dexter Kozen, Wojtek Moczydlowski, Nate Nystrom, K. Vikram, Xin Zheng, and the anonymous reviewers.

This research was supported in part by ONR Grant N000140910652, by NSF Grants 0430161, 0541217, and 0627649, and by Air Force Research Laboratory under agreement #FA8750-08-2-0079. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright annotation thereon. The views and conclusions here are those of the authors and do not necessarily reflect the policies or endorsements of AFRL, ONR, the Navy, NSF, or the U.S. Government.

References

[1] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In Awais Rashid and Mehmet Aksit, editors, *Lecture Notes in Computer Science : Transactions on Aspect-Oriented Software Development I*, pages 135–173. Springer-Verlag, 2006.

[2] Miles Barr and Susan Eisenbach. Safe upgrading without restarting. In *Proceedings of 19th International Conference on Software Maintenance (ICSM)*, pages 129–137, 2003.

[3] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proc. 20th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 177–189, San Diego, CA, USA, October 2005.

[4] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proc. 5th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[5] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1978.

[6] Kim B. Bruce. Safe static type checking with systems of mutually recursive classes and inheritance. Technical report, Williams College, 1997. `http://cs.williams.edu/~kim/ftp/RecJava.ps.gz`.

[7] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 523–549, Brussels, Belgium, July 1998.

[8] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems, 2002. `http://asm.objectweb.org/current/asm-eng.pdf`.

[9] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2001.

[10] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: A simple virtual class calculus. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, pages 121–134, 2007.

[11] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proc. 15th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 130–145, 2000.

[12] Adriana B. Compagnoni and Benjamin C. Pierce. Higher order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, 1996.

[13] Bruno C. d. S. Oliveira. Modular visitor components: A practical solution to the expression families problem. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, July 2009. to appear.

[14] Ferruccio Damiani, Sophia Drossopoulou, and Paola Giannini. Refined effects for unanticipated object re-classification: Fickle$_{\mathrm{III}}$. In *ICTCS*, pages 97–110, 2003.

[15] Dominic Duggan. Type-based hot swapping of running modules. *Acta Inf.*, 41(4):181–220, 2005.

[16] Erik Ernst. *gbeta—A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[17] Erik Ernst. Family polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, LNCS 2072, pages 303–326, 2001.

[18] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proc. ACM Symp. on Principles of Programming Languages (POPL) '06*, pages 270–282, January 2006.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.

[20] Stephan Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *Proc. Net Object Days*, 2002.

[21] Atsushi Igarashi and Benjamin Pierce. Foundations for virtual types. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*, pages 161–185, June 1999.

[22] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In *Proc. 22nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 113–132, New York, NY, USA, 2007. ACM.

[23] David MacQueen. Modules for Standard ML. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, pages 198–204, August 1984.

[24] O. Lehrmann Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism for object-oriented programming. In *Proc. 4th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 397–406, October 1989.

[25] O. Lehrmann Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.

[26] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 90–100, Boston, Massachusetts, March 2003.

[27] Mira Mezini, Linda Seiter, and Karl Lieberherr. Component integration with pluggable composite adapters. *Software Architectures and Component Technology*, 2000.

[28] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 99–115, October 2004.

[29] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, pages 138–152, Warsaw, Poland, April 2003.

[30] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software composition. In *Proc. 21st ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 21–36, October 2006.

[31] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. Nested intersection for scalable software composition. Technical report, Computer Science Dept., Cornell University, September 2006. `http://www.cs.cornell.edu/nystrom/papers/jet-tr.pdf`.

[32] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language, June 2004. `http://scala.epfl.ch/docu/files/ScalaOverview.pdf`.

[33] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proc. 20th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 41–57, San Diego, CA, USA, October 2005.

[34] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *Lecture Notes in Computer Science*, pages 89–110, Málaga, Spain, 2002. Springer-Verlag.

[35] Xin Qi and Andrew C. Myers. Masked types for sound object initialization. In *Proc. ACM Symp. on Principles of Programming Languages (POPL) '09*, pages 53–65, January 2009.

[36] Xin Qi and Andrew C. Myers. Sharing classes between families. Technical report, Computer and Information Science, Cornell University, March 2009. `http://hdl.handle.net/1813/12141`.

[37] Venugopalan Ramasubramanian, Ryan Peterson, and Emin Gün Sirer. Corona: A high performance publish-subscribe system for the World Wide Web. In *Proceedings of Networked System Design and Implementation (NSDI)*, May 2006.

[38] Venugopalan Ramasubramanian and Emin Gün Sirer. Beehive: $O(1)$ lookup performance for power-law query distributions in peer-to-peer overlays. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[39] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, June 1996.

[40] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[41] Yannis Smaragdakis and Don Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, April 2002.

[42] Charles Smith and Sophia Drossopoulou. Chai: Traits for Java-like languages. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*, pages 453–478, 2005.

[43] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *Proc. ACM Symp. on Principles of Programming Languages (POPL) '05*, pages 183–194, 2005.

[44] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

[45] Sun Microsystems. *Java Language Specification*, version 1.0 beta edition, October 1995. Available at `ftp://ftp.javasoft.com/docs/javaspec.ps.zip`.

[46] Tim Sweeney. The next mainstream programming language: a game developer's perspective. In *Proc. ACM Symp. on Principles of Programming Languages (POPL) '06*, page 269, January 2006.

[47] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in Lecture Notes in Computer Science, pages 444–471. Springer-Verlag, 1997.

[48] Mads Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1998.

[49] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proc. ACM Symp. on Principles of Programming Languages (POPL) '87*, pages 307–312, January 1987.

[50] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In *Proc. 21st ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Portland, OR, October 2006.

[51] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.