# Homogeneous Family Sharing

Xin Qi
Facebook Inc.
xqi@facebook.com

Andrew C. Myers
Department of Computer Science
Cornell University
andru@cs.cornell.edu

## Abstract

Recent work has introduced class sharing as a mechanism for adapting a family of related classes with new functionality. This paper introduces homogeneous family sharing, implemented in the J&$_h$ language, in which the sharing mechanism is lifted from class-level sharing to true family-level sharing. Compared to the original (heterogeneous) class sharing mechanism, homogeneous family sharing provides useful new functionality and substantially reduces the annotation burden on programmers by eliminating the need for masked types and sharing declarations. This is achieved through a new mechanism, shadow classes, which permit homogeneous sharing of all related classes in shared families. The new sharing mechanism has a straightforward semantics, which is formalized in the J&$_h$ calculus. The soundness of the J&$_h$ type system is proved. The J&$_h$ language is implemented as an extension to the J& language. To demonstrate the effectiveness of family sharing, the Polyglot compiler framework is ported to J&$_h$.

***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, shadow classes

## 1. Introduction

The goal of *adaptation* is to add new functionality to existing objects without modifying the original class definition. The premise of this work is that language support for adaptation provides expressiveness needed for the extension, evolution, and reuse of software systems, so programmers can build extensible systems from code that is simpler and clearer.

The adapter design pattern [17] supports limited forms of adaptation, but it adapts only a single object at a time. In general, program behavior is implemented by data structures composed of interacting objects. Adapting program behavior to add new functionality therefore requires coordinated changes to objects in multiple, related classes. However, most language mechanisms that support adaptation, such as open classes [10], aspect binders [23], classboxes [3], and expanders [45] operate on individual classes and consequently cannot support these coordinated changes.

In the context of inheritance rather than adaptation, coordinated changes to multiple classes are supported by *family inheritance* mechanisms developed in recent work, e.g., [15, 20, 22, 28, 30]. Family inheritance enables a group of classes contained in a common namespace—the base family—to be extended together to form a new group of new classes—the derived family—while preserving the relationships that the classes had in the base family. In this work, we build on the family inheritance mechanism of *nested inheritance* [28].

However, family inheritance does not support adaptation, because inheritance does not affect the objects of the base family. The new mechanism introduced in this paper, *homogeneous family sharing*, does support adaptation at the family level. With homogeneous family sharing, a derived family may be declared to *share* with a base family. Just as with nested inheritance, the derived family inherits from the base family. Unlike nested inheritance, the base family also inherits from the derived family, acquiring any new functionality that the derived family adds. Thus, sharing is a bidirectional inheritance mechanism. New functionality, new state, and even new classes introduced in the derived family are also present in the base family.

Homogeneous family sharing gives the ability to adapt families of interacting classes in a type-safe, modular way. It enables new ways to safely extend software systems, even at run time as dynamic updates. This paper introduces homogeneous family sharing in the J&$_h$ language[1], which extends the J& language [30].

*Class sharing* as an adaptation mechanism was first introduced in the J&$_s$ language [36]. In J&$_s$, sharing is a re-

[1] pronounced “jet-h”

lationship between individual classes, with some coordination at the family level. Unlike with homogeneous sharing, J&$_s$ families need not share all of their nested classes; we call this *heterogeneous* sharing because classes are not uniformly shared between families. This work addresses some of the shortcomings of heterogeneous sharing.

The principle of *scalable extensibility* [28] says that the code needed to make a change should be proportional to the change in functionality. One shortcoming of heterogeneous sharing is a lack of scalability that arises because, in general, sharing must be declared for each class individually. Homogeneous family sharing is more scalable because sharing can be declared at the family level.

A second shortcoming of heterogeneous sharing is that to handle unshared classes safely, additional type-level mechanisms are needed: masked types [35] and sharing constraints. By contrast, in homogeneously shared families, all classes and all inheritance and sharing relationships are shared. As a result, J&$_h$ programs have lighter-weight type annotations than in J&$_s$; in particular, J&$_h$ dispenses with masked types and sharing constraints.

The contributions of this paper are, in summary:

- A clean generalization of sharing to families of classes. Compared to the previous work on heterogeneous class sharing, homogeneous family sharing offers simpler typing mechanisms, requires simpler reasoning from programmers, and improves scalable extensibility.
- *Shadow classes*, a new mechanism needed for safe homogeneous sharing. Shadow classes also provide a new kind of extensibility unavailable in previous work: analogously to open classes [10], families become *open*, allowing nested classes to be added without modifying the code of the family.
- A core language for homogeneous sharing, which is significantly simpler than the J&$_s$ calculus, and a proof of the soundness of its type system.
- An efficient implementation of homogeneous sharing, supporting shadow classes.
- An description of how to use J&$_h$ to implement various kind of extensibility, and a report on experience using J&$_h$ to build a version of the Polyglot compiler framework [29].

The rest of the paper is organized as follows. Section 2 introduces sharing for individual classes. Section 3 describes the core mechanisms underlying homogeneous family sharing. Section 4 formalizes the semantics of family sharing, and sketches the proof of soundness. Full proofs are available elsewhere [38]. Section 5 presents the language implementation, and Section 6 describes experience with it. Section 7 discusses related work, and Section 8 concludes.

```
class Button {
  void draw(...) { ... }
}
```

```
class PrettyButton extends Button {
  void draw(...) { ... draw the shadow ... }
}
```

**Figure 1.** A GUI button class and its extension

## 2. Sharing and adaptation

### 2.1 From inheritance to sharing

In an object-oriented language, inheritance offers a way to reuse and extend existing code. For example, in the Java code of Figure 1, a graphical user interface (GUI) library contains a `Button` class with a method `draw` for drawing the button. Suppose the programmer wants buttons with a shadow underneath. The programmer could declare a `PrettyButton` class that extends the original `Button` class through inheritance. In Figure 1, the subclass `PrettyButton` overrides the `draw` method, originally introduced in the superclass `Button`, so that when the `draw` method is called on an object of `PrettyButton`, the overriding version in the subclass is executed.

However, inheritance does not make the new shadow functionality available to the objects of the existing `Button` class. An application that creates instances of the `Button` class cannot enjoy the prettier GUI element without code changes, nor can a running application with `Button` objects be easily upgraded to the new look.

*Adaptation* is the ability to augment existing objects with new functionality. The adapter design pattern [17] uses wrappers to implement adaptation, but in general, it requires the programmer to write error-prone code relying on statically unsafe type casts. The programmer also has to manually manage the relationship between the wrapper and the adapted object, which becomes especially awkward for data structures made of objects requiring adaptation.

Sharing provides a language-based solution to the problem of adaptation. In J&$_h$, the new `PrettyButton` class may be declared to *share* with the `Button` class, rather than to inherit it:

```
class PrettyButton shares Button { ... }
```

The meaning of the declaration is twofold: first, that `PrettyButton` is a subtype of `Button`, just as with ordinary inheritance; but second, that every `Button` object is also an object of `PrettyButton`. Because `Button` and `PrettyButton` share the same set of object instances, the functionality implemented in `PrettyButton` is available to every object of `Button`—even to objects created before `PrettyButton` was loaded into a running program—when the programmer explicitly changes the view from `Button` to `PrettyButton` on the object.

An object of either of the two classes inherits functionality from the other class, and each class may also override the other. Although classes share the same instances, the behavior of an object depends on which class an object is being viewed from. If viewed as a `Button`, an object created as a `PrettyButton` may still access the original version of the `draw` method. Sharing provides *bidirectional* adaptation while preserving object identity.

A sharing declaration establishes a *sharing relationship* between the two classes. All the sharing declarations together induce an equivalence relation on classes that is the reflexive, symmetric, and transitive closure of the declared sharing relationships. We write $T_1 \leftrightarrow T_2$ for two types that are (transitively) shared. To represent a sharing relationship that holds between just the two classes `A` and `B` but not between any subclasses of `A` or `B`, we use *exact types* [5]. The exact type `A!` represents just `A`, but not its subclasses. Therefore `Button!` $\leftrightarrow$ `PrettyButton!` signifies a sharing relationship on just these two classes.

Class sharing was introduced in J&$_s$ [36], but its original, heterogeneous form has limitations: first, J&$_s$ only allows sharing between corresponding classes—classes of the same name and related through overriding—from different families. Second, unlike family sharing in J&$_h$, class sharing in J&$_s$ is a relationship on the shared classes themselves; it does not cause sharing of classes nested inside the sharing classes. The homogeneous sharing mechanism proposed in this paper removes both of these limitations, and thereby adds expressive power.

### 2.2 Views and view changes

When two classes are shared, an instance of either is also an instance of the other. Each class is a distinct *view* of that object, which defines the run-time behavior of the object. At run time, an object reference may be modeled as a pair $\langle \ell, C! \rangle$ of a heap location $\ell$ and an exact type $C!$ specifying the view.

The view controls how methods are dispatched on the object being accessed through the reference. For example, an object created as a `Button` would use the overriding version of the `draw` method if accessed through a reference with the view `PrettyButton!`.

One object may simultaneously have several references, each with a different view. For an object to obtain a new reference with a new view, we use a *view change* operation, written $(\texttt{view}\ T)e$, where $T$ is the target type, and $e$ is the source expression. For example, a `Button` object may obtain a new reference `b2`, through which the new `draw` method can be called, although the receiver object is still the same:

```
Button! b1 = new Button(...);
PrettyButton! b2 = (view PrettyButton!)b1;
b1.draw(...); // the old draw method in Button
b2.draw(...); // the new draw method in PrettyButton
```

```
package GUI;
class Button {...}
class RadioButton extends Button {...}
class Window { Button close; ... }
...
```

```
package xlucentGUI extends GUI;
class Button {
  void draw(...) {...}
}
class Window { void draw(...) {...} }
...
```

**Figure 2.** A base GUI family and an extension

Although view changes look like dynamic type casts, view change operations are type-safe. They are type-checked statically using the sharing relation.

In J&$_h$, a reference $\langle \ell, C! \rangle$ is a first-class value whose view component $C!$ is determined dynamically. For example, after the view change above, if the value in `b2` is passed to code expecting type `Button`, the value will still behave as an object of `PrettyButton`:

```
class Renderer {
  void render(Button b) { b.draw(...); }
}
...
Renderer r;
r.render(b2); // the PrettyButton draw method is called
              // from render()
```

Therefore, view changes may update the behavior of an existing object, even when, as in the body of `render`, the new shared class is not statically in scope.

In both J&$_s$ and J&$_h$, a concrete (i.e., not abstract) class cannot be declared to share with an abstract class or an interface. This restriction ensures that an object has only concrete views, and therefore view-based method dispatch does not have message-not-understood errors.

## 3. Homogeneous family sharing

### 3.1 Family inheritance

Homogeneous family sharing builds on prior family-level extensibility mechanisms: 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. When a namespace inherits from another (base) namespace, all the namespaces nested in the base namespace are inherited, and the derived namespace can *override*, or *further bind* [21] inherited namespaces, changing nested class declarations, similarly to virtual classes [9, 13, 16, 21, 22]. Nested intersection supports *composing* families with generalized *intersection types* [11, 39] — a mechanism for multiple inheritance at the family level.

Family inheritance is useful, because in real software systems, the functionality that needs to be extended often spans multiple classes that are related to each other through either inheritance or mutual references. For example, as in Figure 2, the GUI library should also include classes for various GUI elements, including the `Button` class of Figure 1. Suppose we want to extend the entire GUI library to support translucent widgets. With the J&$_h$ language, the extension can be declared at the family level, as shown on the bottom of Figure 2. Every class in the base package `GUI` has a corresponding subclass with the same class name in the derived package `xlucentGUI`.

J&$_h$ supports *scalable extensibility* [28], because there is no need to declare cross-family inheritance for individual classes, and the code that needs to be written in the derived family is proportional to the changes in functionality.

The relationships between classes in `GUI` are preserved in `xlucentGUI` with *late binding of type names*. For example, the `Window` class has a field for a button named `close`, with type `Button`. In `GUI.Window`, the name `Button` refers to the `Button` class in `GUI`, whereas in `xlucentGUI.Window`, it refers to the corresponding class of `xlucentGUI`. Even classes that are not mentioned explicitly in the derived namespace are inherited as *implicit classes* to which late binding applies. For example, package `xlucentGUI` contains an implicit class `RadioButton` whose superclass is the `Button` in `xlucentGUI` rather than the one in `GUI`.

The type safety of late-bound type names is ensured with *prefix types* and *dependent classes*: in `GUI.Window`, the unqualified field type `Button` is sugar for the prefix type `GUI[this.class].Button`, which means that the package containing `Button` is a subtype of `GUI` that encloses the *run-time* class of the special variable `this`. By indexing the class of `close` by the `Window` object that points to it, we ensure that the window and its `close` button always come from the same family.

J&$_h$ preserves all three kinds of relationships in the derived family: referencing, subclassing, and sharing. Because class sharing in J&$_s$ is heterogeneous, sharing relationships are not guaranteed to be preserved; therefore, J&$_s$ requires sharing constraints for type safety.

## 3.2 Family sharing

Homogeneous family sharing generalizes sharing to family granularity analogously to the way that nested inheritance generalizes inheritance to the family granularity. For example, in order to adapt existing objects from the `GUI` family with the new drawing methods declared in `xlucentGUI`, the following *family sharing* declaration may be used, with the rest of the code shown in Figure 2 remaining the same:

```
package xlucentGUI shares GUI;
```

With the sharing declaration, `xlucentGUI` is a derived family of `GUI`, just as with the inheritance declaration in Figure 2. Therefore, the shared derived family `xlucentGUI` inherits all the nested classes from the base family `GUI`, and preserves all the relationships among them, including sharing relationships. If the `GUI` package contains a `PrettyButton` class that shares with `Button`, then they are still shared in the `xlucentGUI` package.

The difference from nested inheritance is that corresponding classes from the two families (e.g., `GUI.Button` and `xlucentGUI.Button`) also become shared, and therefore objects of classes from one family are also instances of classes from the other family. View changes can be used to move an object from one family to another. For example, a `Window` object from the `GUI` family acquires translucency when viewed from the `xlucentGUI` family:

```
GUI!.Window w1 = ...;
xlucentGUI!.Window w2 = (view xlucentGUI!.Window)w1;
```

Moreover, field accesses in J&$_h$ automatically and lazily trigger implicit view changes. Therefore, any object reachable from the `Window` object (e.g., the one stored in the field `close` in the `Window` class) will also obtain a view in `xlucentGUI`, when it is accessed.

In J&$_h$, a sharing relationship between two namespaces recursively applies to all their corresponding nested namespaces. Thus family sharing is homogeneous. This compares to the heterogeneous class sharing [36], which is declared for individual pairs of classes between two families. The J&$_s$ language does provide an `adapts` declaration as syntactic sugar for declaring sharing between all corresponding classes of two families, but this only causes sharing one level down, rather than recursively, and can only be used when the families agree completely on the set of nested classes.

Having homogeneous sharing at the family level allows the J&$_h$ type system to be modularly type-safe without complicated mechanisms like masked types and sharing constraints [36], simplifying programming.

## 3.3 Shadow classes and shadow methods

Shadow classes are the key to homogeneous sharing. In the J&$_h$ language, a derived family may introduce new nested classes not present in a shared base family. In order to keep view changes type-safe, these new nested classes in general need to be shared as well. Shadow classes are introduced in the base family to make this possible.

Figure 3 shows a simple example that illustrates the situation. The derived family enclosed in `A2` is shared with the base family in `A1`. Class `A2` inherits the nested class `B` from `A1`, and introduces a new class `C` that is a subclass of `A2.B`. Because sharing is homogeneous, the two types `A1!.B` and `A2!.B` are shared, and therefore a view change from `A2!.B` to `A1!.B` is allowed. But as shown in following code, the source expression `b2` with static type `A2!.B` may actually refer to an object of class `A2.C` at run time.

```
A2!.B b2 = new A2.C();
A1!.B b1 = (view A1!.B)b2;
b1.m(); // invokes the shadow method
```

```
class A1 {
  class B {
    void m() {...}
  }
}
class A2 shares A1 {
  class C extends B {       // adds shadow class in A1
    shadow void m() {...} // a shadow method
  }
}
```

**Figure 3.** Shadow classes and shadow methods

For the view change to work, A1 must contain a class shared with A2.C.

J&$_h$ introduces shadow classes to enable view changes like this one. When a shared derived family (e.g., A2) adds a new nested class (e.g., A2.C), the base family also acquires a class with the same name (here, A1.C). The derived family introducing the new nested class is called the *originating family* for the shadow class, and the nested class is called the *originating class*. The shadow class is shared with its originating class, and inherits all the fields and relationships from the originating family. Thus, the shadow class is a class that the base family inherits from the derived family.

The J&$_h$ language not only supports enhancing the shared base family with new shadow classes, but also allows the programmer to define how the shadow class should behave in the base family. The originating class may declare *shadow methods*, with the modifier shadow. The semantics of a shadow method is as if it were defined within an explicit declaration of the shadow class in the base family. As a result, A2.C in Figure 3 could also declare its own version of method m() without the shadow keyword. This vesion would override the shadow method as if it had been inherited from the shadow class A1.C.

On the other hand, there is no need to declare "shadow fields", because the shadow class shares all fields declared in the originating class.

## 3.4 Modular type checking of shadow classes

Because shadow classes are not explicitly declared with a class body, the type system needs to know the originating class before a shadow class is mentioned. For example, inside class A1 in Figure 3, the type name C is meaningless if A2 has not been created. Modularity of the type system requires that a base family can be type-checked without knowing derived families, unless they are used explicitly by the base family. Therefore, if the code in A1 does not explicitly use the shadow class, the compiler should be able to type-check A1 without knowing about A2.C.

J&$_h$ ensures the modularity of the type system by disallowing direct naming of a shadow class in source code. For example, given the declarations in Figure 3, one cannot mention A1.C, the fully qualified name of the shadow class, anywhere in the source code, nor can one mention just C in the context of A1. Instead, the language overloads the disambiguation usage of prefix types [30] to provide an indirect way of naming a shadow class that embeds the name of the originating class. In the J&$_h$ source code, a shadow class is referred to as $P$[$T$].$C$, where $P$ is the originating family, $T$ is a type in the same family as the shadow class, including the family itself, and $C$ is the simple name of the originating class. Therefore, the shadow class in Figure 3 may be denoted as A2[A1.B].C or just A2[A1].C.

This scheme also solves the problem of accidental name conflicts between shadow classes. Suppose that in addition to the declarations in Figure 3, there is another shared derived family A3, which introduces a new nested class that happens to be named C as well:

```
class A3 shares A1 { class C extends B {...} }
```

Then there will be two different shadow classes in A1 with the same name C—it is not sensible to combine them into one shadow class—and there is no modular way to detect this situation. However, indirect naming in J&$_h$ prevents the potential name conflict, by naming the two shadow classes differently, as A2[A1.B].C and A3[A1.B].C respectively. Therefore, the two (unrelated) shadow classes can be used independently of each other. (The J&$_h$ compiler also mangles the names of the two shadow classes differently.)

Although the syntax for naming shadow classes looks somewhat heavy, it is unlikely to be used frequently (see Section 3.5 for an example of the syntax). We expect shadow classes normally to be used as a way to introduce new subclasses of some known, ordinary class in the base family. The base family would generally use objects of the shadow class through the known superclass, without explicitly mentioning the shadow class. The prefix syntax is also not necessary in the originating family, including inside the declaration of a shadow method, where the syntax coincides with the automatic desugaring of unqualified type names.

## 3.5 Open families

Shadow classes and shadow methods provide a new kind of extensibility: *open families*, which are analogous to open classes [10]. An existing family can be extended in a modular way with new functionality, including new classes, without modifying the code of the family. This contrasts both with heterogeneous sharing [36] and with nested inheritance [30], where families are *locally closed worlds* that cannot be extended with new nested classes without modifying existing code.

This kind of extensibility is different from sharing-based adaptation, which modularly adds new functionality to objects that belong to an existing family, but makes new behavior accessible only by viewing the objects in the shared derived family. By contrast, shadow classes are available in the base family. For example, the shadow method m() declared by A2.C in Figure 3 may be invoked through a re-

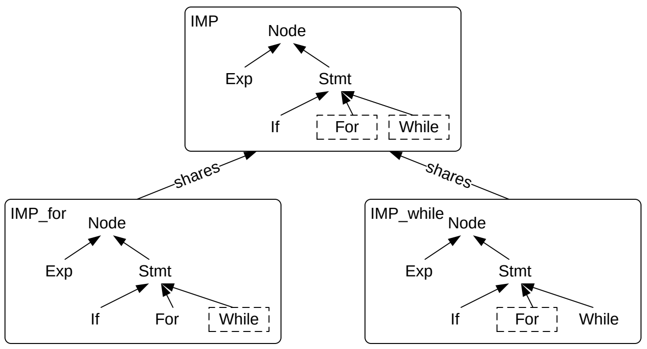

**Figure 4.** Sketch of the IMP compiler structure. Shadow classes in dashed boxes.

ceiver object of static type `A1!.B`—definitely in the base family—if the receiver object is an instance of the shadow class `A2[A1.B].C`.

### 3.6 Example: in-place translation

To show the extensibility provided by homogeneous sharing and open families, let us consider using them to build a simple language translation. A compiler built using many passes often includes translation passes that leave most of the abstract syntax tree (AST) alone. One would like to avoid creating an entirely new AST for the target language, but this is not possible if the source and target of the translation have different AST types.

In-place translation solves this problem. If the source family and the target family are shared, one can simply apply view changes to objects that do not need translation, avoiding the generation of many new objects, while still ensuring the entire data structure behaves consistently in the target family.

Figure 4 sketches the class hierarchy of a compiler for a small imperative language and two extensions. The base family `IMP` describes a simple core language without any loop construct. The two derived families `IMP_for` and `IMP_while` respectively extend `IMP` with for-loops and while-loops. The compiler translates a program from `IMP_for` to `IMP_while`, rewriting for-loops to while-loops.

Figure 5 illustrates J&$_h$ code that does in-place translation from `IMP_for` to `IMP_while`. The two derived families are both shared with the base family `IMP`, and by transitivity, they are also shared with each other. The class `For` in `IMP_for` and the class `While` in `IMP_while` both introduce shadow classes in `IMP`, which are then inherited by the other derived family. On line 31, with the syntax described in Section 3.4, the type `IMP_while[this.class].While` represents the shadow class `While` in the `IMP_for` family. This allows the translation code to safely generate an `While` object still in the `IMP_for` family. Translation of `If` (lines 19–24) lazily generates a new `If` node only if its children have changed. After the recursive translation is done on an AST,

```
 1  package IMP;
 2  abstract class Node { ... }
 3  ...
 4  class If extends Stmt {
 5    Exp cond; Stmt body;
 6    If(cond, body) {
 7      this.cond = cond; this.body = body;
 8    }
 9  }
10
11  package IMP_for shares IMP;
12  abstract class Node {
13    Node translate() {
14      return this; // by default, no translation
15    }
16  }
17  class If extends Stmt {
18    Stmt translate() { // covariant return type
19      Exp cond = this.cond.translate();
20      Stmt body = this.body.translate();
21      if (cond != this.cond || body != this.body)
22        return new If(cond, body);
23      else
24        return this;
25    }
26  }
27  class For extends Stmt {
28    Exp init, cond, inc;
29    Stmt body;
30    Stmt translate() {
31      ... new IMP_while[this.class].While(...); ...
32    }
33  }
```

**Figure 5.** In-place translation for loop statements

a single view change operation applied to the root moves the entire AST to the `IMP_while` family:

```
IMP_for!.Node source = ...;
IMP_for!.Node temp = source.translate();
IMP_while!.Node target = (view IMP_while!.Node)temp;
```

In this example, the use of shadow classes facilitates the translation, which goes directly between two "sibling" families. Heterogeneous sharing, and other adaptation mechanisms, cannot support in-place translation across the hierarchy of familes.

### 3.7 Homogeneous vs. heterogeneous sharing

Homogeneous family sharing has several advantages over heterogeneous sharing:

***Scalable sharing.*** There is no need to declare sharing for individual pairs of classes from two different families, which can be tedious and result in errors. This also makes the sharing mechanism more scalable, because if a class only needs to be shared but not overridden, its declaration may be omitted from the source code of the derived family.

***Modular type checking with reduced annotation burden.*** In particular, homogeneous sharing does not require sharing constraints, because it is easier to prove that two types are shared. If two types $T_1$ and $T_2$ are shared, all corresponding nested types are also shared, that is, $T_1.C \leftrightarrow T_2.C$.

For example, a variable `b1` of type `GUI!.Button` might point to an object of the `GUI.RadioButton` class or to one of the `GUI.PrettyButton` class. In either case, the view change (`view xlucentGUI!.Button`)`b1` is type-safe, because sharing is declared between `GUI` and `xlucentGUI`—and therefore, all classes in `GUI` that are subtypes of `GUI!.Button` have shared counterparts in `xlucentGUI`.

On the contrary, with heterogeneous sharing, in order to prove that a view change $(\texttt{view}\ T)e$ is valid, the type system has to inspect all the subclasses of both the source type (the type of $e$) and the target type $T$, and it has to recheck the view change every time the code is inherited by a different family. Therefore, heterogeneous sharing in J&$_s$ uses sharing constraints to make type-checking modular. With homogeneous sharing, these constraints are superfluous.

***Straightforward support for coordinated view changes on interconnected objects.*** As in heterogeneous sharing, field accesses in J&$_h$ trigger implicit view changes, which allows a data structure consisting of multiple interconnected objects to safely change its view from one family to another.

However, in heterogeneous sharing, a shared class may contain a field with an unshared type—often as the result of a new class in the derived family (e.g., the `For` class in Figure 18)—in which case the field actually has several duplicates, one for each class that is shared with the class that declares the field. Masked types are used to ensure unshared objects stored in these fields do not leak into incompatible families. This adds to the annotation burden, and complicates reasoning about the code. Homogeneous family sharing does not need masked types for this purpose, because shadow classes ensure that a data structure can always be homogeneously moved to a shared family.

The compiler example of Section 3.6 shows that J&$_h$ presents a different trade-off in language design than J&$_s$. The type system of heterogeneous sharing in J&$_s$ does more checking to prevent objects from leaking into other families by mistake. For example, if a translation from `IMP_for` to `IMP` is implemented in J&$_s$, the type system would ensure that no `For` object is left untranslated in the target AST. However, J&$_s$ achieves this family-closedness guarantee at the expense of heavyweight language mechanisms like sharing constraints and masked types.

J&$_h$ instead introduces shadow classes to ensure that shared families always have homogeneous structures. Although a view change is guaranteed to move a data structure completely into the new family from the type perspective, the type system no longer checks for unfinished translation. For example, if a `For` object is left untranslated, it will end up as an object of the shadow class `For` in the target family—

| | | |
|---|---|---|
| programs | $Pr$ | $::= \langle \overline{L}, e \rangle$ |
| class declarations | $L$ | $::= \texttt{class}\ C\ ES\ \{\overline{L}\ \overline{F}\ \overline{M}\}$ |
| superclass declarations | $ES$ | $::= \texttt{extends}\ T \mid \texttt{shares}\ T$ |
| field declarations | $F$ | $::= [\texttt{final}]\ T\ f = e$ |
| method declarations | $M$ | $::= [\texttt{shadow}]\ T\ m(\overline{T}\ \overline{x})\ \{e\}$ |
| types | $T$ | $::= \top \mid \circ \mid T.C \mid p.\texttt{class} \mid P\texttt{[}T\texttt{]} \mid T!$ |
| classes | $P$ | $::= \top \mid \circ \mid P.C$ |
| values | $v$ | $::= \langle \ell, P! \rangle$ |
| access paths | $p$ | $::= v \mid x \mid p.f$ |
| expressions | $e$ | $::= v \mid x \mid e.f \mid e_0.f = e_1 \mid e_0.m(\overline{e})$ |
| | | $\mid e_1;\ e_2 \mid \texttt{new}\ T(\overline{f} = \overline{e}) \mid (\texttt{view}\ T)e$ |
| typing environments | $\Gamma$ | $::= \emptyset \mid \Gamma, x{:}T \mid \Gamma, p_1 = p_2$ |

**Figure 6.** Syntax of J&$_h$

either `IMP` or `IMP_while`—but this is still type-safe in J&$_h$. The payoff is a simpler, easier-to-use language. Moreover, shadow classes also introduces a new kind of extensibility, as shown in Section 3.5. Therefore, we believe homogeneous sharing makes a better trade-off for most applications.

## 4. Formal semantics

This section formalizes homogeneous family sharing in the J&$_h$ calculus. Not all the features of the J&$_h$ language are modeled in J&$_h$ calculus, in order to focus on sharing. For example, virtual types, explicit multiple inheritance and intersection types are omitted.

### 4.1 Syntax

Figure 6 shows the syntax of the J&$_h$ calculus. The notation $\overline{a}$ is used for both the list $a_1, \ldots, a_n$ and the set $\{a_1, \ldots, a_n\}$, for $n \geq 0$, and $\#(\overline{a})$ represents the size of $\overline{a}$.

A program $Pr$ is a pair $\langle \overline{L}, e \rangle$ of a set of class declarations $\overline{L}$ and an expression $e$ (the `main` method). Each class $C$ has a superclass declaration, which is either a normal superclass declaration `extends` $T$, or a shared superclass declaration `shares` $T$. Note that there is only one superclass, because J&$_h$ only models single inheritance. There are two special classes: $\top$ is the superclass of all the other classes, similar to `Object` in Java; $\circ$ is the single top-level class that all other classes are nested within.

Method declarations may have an extra modifier `shadow` for declaring shadow methods. Fields may be declared `final`, which means they cannot be changed after initialization, as in Java.

J&$_h$ supports explicit exact types, and exactness applies to the entire type preceding the symbol "!". $T!.\overline{C}!$ is considered equivalent to $T.\overline{C}!$, and $T!$ is equivalent to $T$ if $T$ is already exact.

A value $v$, also called a reference, is a pair of a heap location $\ell$ and the associated view $P!$, which is a class in its exact form. Expressions are mostly standard, with the addition of a view change operation $(\texttt{view}\ T)e$.

Shared subclassing $\vdash P_1 \underset{\leftrightarrow}{\sqsubseteq} P_2$

$$\frac{\begin{array}{c}\mathsf{super}(P.C) = \mathtt{shares}\ T \\ \vdash P_1 \sqsubset^* P \quad T\{\!\{\emptyset;\ P_1/\mathtt{this}\}\!\} = P_2\end{array}}{\vdash P_1.C \underset{\leftrightarrow}{\sqsubseteq} P_2} \quad \text{(SHARE-DECL)}$$

$$\frac{\vdash P_1 \underset{\leftrightarrow}{\sqsubseteq} P_2}{\vdash P_1.C \underset{\leftrightarrow}{\sqsubseteq} P_2.C} \quad \text{(SHARE-FB)}$$

Subclassing $\vdash P_1 \sqsubset P_2$

$$\frac{\begin{array}{c}\mathsf{super}(P.C) = \mathtt{extends}\ T \\ \vdash P_1 \sqsubset^* P \quad T\{\!\{\emptyset;\ P_1/\mathtt{this}\}\!\} = P_2\end{array}}{\vdash P_1.C \sqsubset P_2} \quad \text{(SC-DECL)}$$

$$\frac{\vdash P_1 \sqsubset P_2}{\vdash P_1.C \sqsubset P_2.C} \quad \text{(SC-FB)}$$

$$\frac{\vdash P_1 \underset{\leftrightarrow}{\sqsubseteq} P_2}{\vdash P_1 \sqsubset P_2} \quad \text{(SC-SHARE)}$$

**Figure 7.** Sharing and subclassing

For simplicity, we omit the `null` value from the J&$_h$ calculus, and require that all field declarations come with default initializations.

The typing environment contains aliasing information about access paths. An entry $p_1 = p_2$ means $p_1$ and $p_2$ are aliases that also have the same run-time view. As in [9], this kind of information is just used by the soundness proof.

### 4.2 Sharing and subclassing

Subclassing relationships among classes are defined in Figure 7. The judgment $\vdash P_1 \underset{\leftrightarrow}{\sqsubseteq} P_2$ states that $P_1$ is a shared subclass of $P_2$. With homogeneous family sharing, when two classes are declared to be shared, all the nested classes are also automatically shared, according to SHARE-FB. On the other hand, $\vdash P_1 \sqsubset P_2$ states that $P_1$ is a subclass of $P_2$, either shared or not.

### 4.3 Lookup functions

The class table $CT$, defined in Figure 8, contains the declaration for any explicit or implicit class that is *not* a shadow class. The extended class table $CT'$ contains synthesized declarations for all the shadow classes, in addition to the declarations in $CT$. CT$'$-SHADOW introduces a shadow class $P.C$ in every shared superclass $P$ of $P'$, if $P'$ declares a nested class $C$. Note that the originating class $P'.C$ must not be inherited from any (direct or overriding) superclass of $P'$. CT$'$-PROP states that shadow classes are also inherited, which allows them to be propagated to "sibling" families, as

Class table $CT(P)$

$$\frac{Pr = \langle \overline{L}, e\rangle}{CT(\circ) = \mathtt{class}\ \circ\ \mathtt{extends}\ \top\ \{\overline{L}\}} \quad \text{(CT-OUT)}$$

$$\frac{\begin{array}{c}CT(P) = \mathtt{class}\ C'\ ES\ \{\overline{L}\ \overline{F}\ \overline{M}\} \\ L_i = \mathtt{class}\ C\ \ldots\end{array}}{CT(P.C) = L_i} \quad \text{(CT-EXP)}$$

$$\frac{\begin{array}{c}CT(P) = \mathtt{class}\ C'\ \ldots\ \{\overline{L}\ \overline{F}\ \overline{M}\} \quad \mathtt{class}\ C\ \ldots \notin \overline{L} \\ \vdash P \sqsubset P' \quad CT(P'.C) = \mathtt{class}\ C\ ES\ \{\ldots\}\end{array}}{CT(P.C) = \mathtt{class}\ C\ ES\ \{\}} \quad \text{(CT-IMP)}$$

Extended class table $CT'(P)$

$$\frac{CT(P) \neq \bot}{CT'(P) = CT(P)} \quad \text{(CT}'\text{-NORM)}$$

$$\frac{\begin{array}{c}\vdash P' \underset{\leftrightarrow}{\sqsubseteq}^* P \\ CT(P'.C) = \mathtt{class}\ C\ ES\ \{\overline{L}\ \overline{F}\ \overline{M}\} \\ \forall P''\ .\ \vdash P' \sqsubset P'' \Rightarrow CT(P''.C) = \bot \\ \overline{M'} = \mathsf{shadowMethods}(P'.C)\end{array}}{CT'(P.C) = \mathtt{class}\ C\ ES\ \{\overline{M'}\}} \quad \text{(CT}'\text{-SHADOW)}$$

$$\frac{\begin{array}{c}CT(P.C) = \bot \quad \vdash P \sqsubset^* P' \\ CT'(P'.C) = \mathtt{class}\ C\ ES\ \{\ldots\}\end{array}}{CT'(P.C) = \mathtt{class}\ C\ ES\ \{\}} \quad \text{(CT}'\text{-PROP)}$$

**Figure 8.** Class lookup

illustrated in Section 3.5. Both $CT$ and $CT'$ are assumed to be global information.

Figure 9 shows auxiliary functions for looking up various class members like fields and methods: $\mathsf{super}(P)$ gives the superclass declaration of $P$, either a normal superclass or a shared one; $\mathsf{shadowMethods}(P)$ collects all the shadow method declarations from $P$, with the `shadow` modifier removed; the functions $\mathsf{fields}(P)$ and $\mathsf{methods}(P)$ collect all the field and method declarations from $P$ and its superclasses; $\mathsf{fnames}(\overline{F})$ is the set of all field names in field declarations $\overline{F}$; $\mathsf{ftype}_{decl}(\Gamma, T, f)$ returns the declared type of field $f$, which might be a type dependent on `this`, and whether the field is final or not; $\mathsf{ftype}(\Gamma, T, f)$ substitutes the receiver type $T$ for `this.class`; $\mathsf{mbodies}(P, m)$ looks up all the declarations of a method $m$ that are not overridden; $\mathsf{mtype}(\Gamma, T, m)$ looks up the method signature. For any well-typed class $P$, $\mathsf{mbodies}(P, m)$ never contains more than one element (see L-OK in Figure 13).

Note that although shadow methods do not appear in $\mathsf{ownMethods}(P)$ of their declaring class $P$, they are conceptually treated as normal methods declared in shadow classes originated from $P$, and then inherited and invoked just as

Member lookup

$$\frac{CT'(P) = \texttt{class}\ C\ ES\ \{\overline{L}\ \overline{F}\ \overline{M}\}}{\begin{array}{c}\mathsf{super}(P) = ES \\ \mathsf{ownFields}(P) = \overline{F} \\ \mathsf{ownMethods}(P) = \overline{M} - \mathsf{shadowMethods}(\overline{M}) \\ \mathsf{shadowMethods}(P) = \{M' \mid \texttt{shadow}\ M' \in \overline{M}\}\end{array}}$$

$$\mathsf{fields}(P) = \bigcup_{\vdash P \sqsubset^* P_i} \mathsf{ownFields}(P_i)$$

$$\mathsf{methods}(P) = \bigcup_{\vdash P \sqsubset^* P_i} \mathsf{ownMethods}(P_i)$$

$$\frac{\overline{F} = [\texttt{final}]\ \overline{T}\ \overline{f} = \overline{e}}{\mathsf{fnames}(\overline{F}) = \overline{f}}$$

$$\frac{\Gamma \vdash T \trianglelefteq P \quad \mathsf{fields}(P) = \overline{F} \qquad F_i = [\texttt{final}]\ T_f\ f = e}{\mathsf{ftype}_{decl}(\Gamma, T, f) = [\texttt{final}]\ T_f}$$

$$\frac{[\texttt{final}]\ T_f^{decl} = \mathsf{ftype}_{decl}(\Gamma, T, f)}{\mathsf{ftype}(\Gamma, T, f) = T_f^{decl}\{\!\!\{\Gamma;\ T/\texttt{this}\}\!\!\}}$$

$$\frac{\begin{array}{c}M_i = \ldots\ m(\ldots)\ \ldots \\ M_i \in \mathsf{ownMethods}(P_i) \\ \forall P'.\ P \sqsubset^* P' \sqsubset^+ P_i \Rightarrow \ldots\ m(\ldots)\ \ldots \notin \mathsf{ownMethods}(P')\end{array}}{\mathsf{mbodies}(P, m) = \overline{M}}$$

$$\frac{\Gamma \vdash T \trianglelefteq P \quad \mathsf{mbodies}(P) = \{M\} \qquad M = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\}}{\mathsf{mtype}(\Gamma, T, m) = (\overline{x}{:}\overline{T}) \to T_{n+1}}$$

**Figure 9.** Class member lookup

normal methods. This is specified in CT′-SHADOW, where the synthesized shadow class $P.C$ contains all the shadow method declarations from its originating class $P'.C$, as if they are normal method declarations in $P.C$.

CT′-SHADOW also states that the originating family $P'$ of the shadow class $P.C$ must be the class that introduces $C$, i.e., the originating class $P'.C$ cannot override any non-shadow class. This ensures that every shadow class has exactly one originating class, and therefore no conflict between shadow method declarations may happen.

For simplicity, we assume there are no accidental name conflicts in J&$_h$, that is, all the field declarations use different field names, unrelated methods have different names, and classes that do not override each other also have different names. This can be achieved with name mangling. In particular, shadow classes introduced by different originating classes are always treated as unrelated, different classes, as described in Section 3.4, and their names are assumed to have been mangled differently.

$$\begin{aligned}
\mathsf{prefixExact}_k(\top) &= \mathsf{false} \\
\mathsf{prefixExact}_k(\circ) &= \mathsf{true} \\
\mathsf{prefixExact}_k(T.C) &= \begin{cases}\mathsf{false} & \text{if } k = 0 \\ \mathsf{prefixExact}_{k-1}(T) & \text{otherwise}\end{cases} \\
\mathsf{prefixExact}_k(p.\texttt{class}) &= \mathsf{true} \\
\mathsf{prefixExact}_k(P\texttt{[}T\texttt{]}) &= \mathsf{prefixExact}_{k+1}(T) \\
\mathsf{prefixExact}_k(T\texttt{!}) &= \mathsf{true}
\end{aligned}$$

**Figure 10.** Prefix exactness

## 4.4 Prefix types

A non-dependent prefix type $P\texttt{[}P'\texttt{]}$ signifies either a subclass of $P$ in which $P'$ is nested, or a shared superclass of $P$ in which $P'$ is nested, as captured in the auxiliary function $\mathsf{prefix}(P, P')$. J&$_h$ generalizes prefix types to include the second case, for naming shadow classes (see Section 3.4).

$$\mathsf{prefix}(P, P') = \begin{cases} P'' & \text{if } P' = P''.C \wedge \vdash P'' \sqsubset^* P \\ P'' & \text{if } P' = P''.C \wedge \vdash P \mathrel{\underset{\leftrightarrow}{\sqsubseteq}}^* P'' \\ \bot & \text{otherwise}\end{cases}$$

As in [31], we only consider prefix types $P\texttt{[}T\texttt{]}$ where the index $T$ is exactly one level deeper in the nesting hierarchy than the bound $P$. However, more general prefix types can be encoded.

## 4.5 Type substitution

The rules for type substitution are shown in Figure 11. Type substitution $T\{\!\!\{\Gamma;\ T_x/x\}\!\!\}$ substitutes $T_x$ for $x.\texttt{class}$ in $T$ in the context of $\Gamma$. The typing context $\Gamma$ is used to look up field types when substituting a non-dependent class into a field-path dependent class.

For type safety, type substitutions on the right-hand side of a field assignment or on the parameters of method calls must preserve the exactness of the declared type. Therefore, only values from the family that is compatible with the receiver are assigned to fields or passed to method code. *Exactness-preserving type substitution* $T\{\!\!\{\Gamma;\ T_x/x!\}\!\!\}$ substitutes $T_x$ for $x.\texttt{class}$ in $T$ and preserves the exactness of $T$:

$$\frac{\begin{array}{c}T\{\!\!\{\Gamma;\ T_x/x\}\!\!\} = T' \\ \forall k.\ \mathsf{prefixExact}_k(T) \Rightarrow \mathsf{prefixExact}_k(T')\end{array}}{T\{\!\!\{\Gamma;\ T_x/x!\}\!\!\} = T'}$$

Exactness is defined using $\mathsf{prefixExact}_k(T)$, as in Figure 10, which means that the $k$-th prefix of $T$ is an exact type.

## 4.6 Static semantics

The static semantics of J&$_h$ is summarized in Figure 12 and Figure 13. Figure 12 shows rules for type sharing $\Gamma \vdash T_1 \leftrightarrow$

$$\top\{\!\{\Gamma;\ T_x/x\}\!\} = \top$$

$$\circ\{\!\{\Gamma;\ T_x/x\}\!\} = \circ$$

$$T.C\{\!\{\Gamma;\ T_x/x\}\!\} = T\{\!\{\Gamma;\ T_x/x\}\!\}.C$$

$$\frac{T\{\!\{\Gamma;\ T_x/x\}\!\} = T'}{P\texttt{[}T\texttt{]}\{\!\{\Gamma;\ T_x/x\}\!\} = P\texttt{[}T'\texttt{]}}$$

$$\frac{T\{\!\{\Gamma;\ T_x/x\}\!\} = T'}{T!\{\!\{\Gamma;\ T_x/x\}\!\} = T'!}$$

$$x.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = T_x$$

$$\frac{x \neq y}{y.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = y.\texttt{class}}$$

$$v.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = v.\texttt{class}$$

$$\frac{p.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = p'.\texttt{class}}{p.f.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = p'.f.\texttt{class}}$$

$$\frac{p.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = T_p \quad T_p \neq p'.\texttt{class} \quad \mathsf{ftype}(\Gamma, T_p, f) = T_f}{p.f.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = T_f}$$

**Figure 11.** Type substitution

$T_2$, subtyping $\Gamma \vdash T_1 \leq T_2$, and expression typing $\Gamma \vdash e{:}T$. Figure 13 defines program well-formedness.

Rules for type well-formedness $\Gamma \vdash T$, non-dependent type bound $\Gamma \vdash T \trianglelefteq P$, final path typing $\Gamma \vdash_{\mathsf{final}} p{:}T$, and final path equality $\Gamma \vdash p_1 = p_2$ are not much different from those in previous work [37], except for the simplification of removing masked types and intersection types, and therefore are omitted here. Please see the companion technical report [38] for the complete semantics.

***Sharing relationships between types.*** The sharing judgment $\Gamma \vdash T_1 \leftrightarrow T_2$, shown in Figure 12, states that a value of type $T_1$ may become a value of $T_2$ through a view change, and vice versa. SH-NEST states that sharing is between families: when two classes are shared, all the corresponding nested types are also shared. SH-DECL collects sharing relationships from class declarations.

Shadow classes ensure that the two shared families are symmetric in the sharing relation, and therefore J&$_h$ does not need directional sharing relationships as in the J&$_s$ calculus, simplifying the semantics.

The sharing relation is reflexive, symmetric, and transitive, as shown by SH-REFL, SH-SYM, and SH-TRANS. This implies that although each class in J&$_h$ or the J& language can only declare at most one shared superclass, it is still possible to encode any sharing relation among classes.

***Subtyping.*** The subtyping judgment $\Gamma \vdash T_1 \leq T_2$ states that $T_1$ is a subtype of $T_2$ in context $\Gamma$, and type equivalence $\Gamma \vdash T_1 \approx T_2$ is sugar for a pair of subtyping judgments.

Most subtyping rules are similar to those in the J&$_s$ calculus, but without any rule about masked types or intersection types. S-SHARE states that the subtyping relationships are preserved by a shared family, and implies that shadow classes in the base family inherit subtyping relationships from the originating family.

***Expression typing.*** The rules for expression typing $\Gamma \vdash e{:}T$ (Figure 12) are mostly standard, with the addition of T-VIEW, which states a view change expression is valid when the source and the target types are shared.

Several rules (T-SET, T-NEW, and T-CALL) use exactness-preserving type substitution $T\{\!\{\Gamma;\ T_x/x!\}\!\}$. See Section 4.5 for its definition.

***Program typing.*** Program typing rules are shown in Figure 13. P-OK states the rule for a program to be well-formed; L-OK, F-OK, and M-OK are the rules for declarations of classes, fields, and methods to be well-formed, respectively. EXT-OK and SH-OK are the well-formedness rules for inheritance and sharing declarations. For simplicity, covariant return types are not modeled.

The rules in Figure 13 uses a simple auxiliary function $\mathsf{paths}(T)$, which represents the set of all the access paths $p$ in dependent types $p.\texttt{class}$ that are part of type $T$. For example, $\mathsf{paths}(P\texttt{[this.class]}.C) = \{\texttt{this}\}$.

### 4.7 Operational semantics

A small-step operational semantics for J&$_h$ is shown in Figures 14 and 15. A heap $H$ is a function mapping pairs $\langle \ell, f\rangle$ of memory locations and field names to values $v$. Heap updates are represented as $H[\langle \ell, f\rangle := v]$.

A reference set $R$, which contains all the references $v$ that have been generated during evaluation, no matter whether or not they are reachable from references in $e$, is also part of the evaluation configuration $e, H, R$. The set $R$ is only for the proof of soundness: it prevents us from losing path equalities needed in the proof.

The evaluation rules (Figure 15) take the form $e, H, R \longrightarrow e', H', R'$. Most of them are standard. R-GET shows that field accesses implicitly trigger view changes, which ensures that objects that reference each other always behave consistently.

***Type evaluation.*** Types in `new` expressions and view change expressions may be dependent, and therefore need to be evaluated according to the type evaluation contexts $TE$ (Figure 14) and the type equivalence rules (Figure 12). A fully evaluated type has the form $P!.\overline{C}$, which is a simple class that has an exact prefix and is not dependent on any access path. There is always an exact prefix, because $\circ$ is exact. Dependent class $\langle \ell, P!\rangle.\texttt{class}$ evaluates to $P!$. Prefix types are evaluated according to rules S-PRE-E1 and S-PRE-E2, which can be seen as normalization rules, reducing the types on the left-hand side of $\approx$ to those on the right-hand side.

***View changes.*** The auxiliary function $\mathsf{view}$ defines the operational semantics for view changes. Because sharing is homogeneous, the generated run-time view $P'.\overline{C'}!$ is well de-

$\boxed{\text{Type sharing } \Gamma \vdash T_1 \leftrightarrow T_2}$

$$\Gamma \vdash T \leftrightarrow T \ (\textsc{SH-refl}) \qquad \frac{\Gamma \vdash T_1 \leftrightarrow T_2}{\Gamma \vdash T_2 \leftrightarrow T_1} \ (\textsc{SH-sym}) \qquad \frac{\Gamma \vdash T_1 \leftrightarrow T_2 \quad \Gamma \vdash T_2 \leftrightarrow T_3}{\Gamma \vdash T_1 \leftrightarrow T_3} \ (\textsc{SH-trans})$$

$$\frac{\Gamma \vdash T_1 \leftrightarrow T_2}{\Gamma \vdash T_1! \leftrightarrow T_2!} \ (\textsc{SH-exact}) \qquad \frac{\begin{array}{c}\Gamma \vdash T_1 \leftrightarrow T_2 \\ \Gamma \vdash T_2.C\end{array}}{\Gamma \vdash T_1.C \leftrightarrow T_2.C} \ (\textsc{SH-nest}) \qquad \frac{\begin{array}{c}\Gamma \vdash T \trianglelefteq P \\ \mathsf{super}(P.C) = \texttt{shares}\ T' \\ T'\{\!\{\Gamma;\ T/\texttt{this}\}\!\} = T''\end{array}}{\Gamma \vdash T.C! \leftrightarrow T''!} \ (\textsc{SH-decl})$$

$\boxed{\text{Subtyping } \Gamma \vdash T_1 \leq T_2}$

$$\Gamma \vdash T \leq \top \ (\textsc{S-top}) \qquad \Gamma \vdash T \leq T \ (\textsc{S-refl}) \qquad \frac{\Gamma \vdash T_1 \leq T_2 \quad \Gamma \vdash T_2 \leq T_3}{\Gamma \vdash T_1 \leq T_3} \ (\textsc{S-trans})$$

$$\frac{\Gamma \vdash_{\mathsf{final}} p{:}T}{\Gamma \vdash p.\texttt{class} \leq T} \ (\textsc{S-fin}) \qquad \frac{\Gamma \vdash_{\mathsf{final}} p{:}T!}{\Gamma \vdash p.\texttt{class} \approx T!} \ (\textsc{S-fin-exact}) \qquad \frac{\Gamma \vdash p_1 = p_2}{\Gamma \vdash p_1.\texttt{class} \approx p_2.\texttt{class}} \ (\textsc{S-alias})$$

$$\Gamma \vdash T.C! \leq T!.C \ (\textsc{S-exact}) \qquad \frac{\Gamma \vdash P\texttt{[}T.C\texttt{]}}{\Gamma \vdash P\texttt{[}T.C\texttt{]} \approx T} \ (\textsc{S-pre-e1}) \qquad \frac{\Gamma \vdash P\texttt{[}T\texttt{]} = T'}{\Gamma \vdash P\texttt{[}T!\texttt{]} \approx T'!} \ (\textsc{S-pre-e2})$$

$$\frac{\Gamma \vdash T \quad \Gamma \vdash T \trianglelefteq P}{\Gamma \vdash T \leq P} \ (\textsc{S-bound}) \qquad \frac{\Gamma \vdash T_1 \leq T_2 \quad \Gamma \vdash T_2.C}{\Gamma \vdash T_1.C \leq T_2.C} \ (\textsc{S-nest}) \qquad \frac{\Gamma \vdash T_1 \leftrightarrow T_2 \quad \Gamma \vdash T_1.\overline{C} \leq T_1.\overline{C'}}{\Gamma \vdash T_2.\overline{C} \leq T_2.\overline{C'}} \ (\textsc{S-share})$$

$$\frac{\begin{array}{c}\Gamma \vdash T_1 \leq T_2 \\ \Gamma \vdash P\texttt{[}T_1\texttt{]} \quad \Gamma \vdash P\texttt{[}T_2\texttt{]}\end{array}}{\Gamma \vdash P\texttt{[}T_1\texttt{]} \leq P\texttt{[}T_2\texttt{]}} \ (\textsc{S-pre-i}) \qquad \frac{\begin{array}{c}\vdash P_1 \sqsubset P_2 \\ \Gamma \vdash P_1\texttt{[}T\texttt{]} \quad \Gamma \vdash P_2\texttt{[}T\texttt{]}\end{array}}{\Gamma \vdash P_1\texttt{[}T\texttt{]} \approx P_2\texttt{[}T\texttt{]}} \ (\textsc{S-pre-b}) \qquad \frac{\begin{array}{c}\Gamma \vdash T \trianglelefteq P \quad \mathsf{super}(P.C) = \ldots\ T' \\ T'\{\!\{\Gamma;\ T/\texttt{this}\}\!\} = T''\end{array}}{\Gamma \vdash T.C \leq T''} \ (\textsc{S-decl})$$

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

$$\frac{\Gamma \vdash_{\mathsf{final}} p{:}T}{\Gamma \vdash p{:}p.\texttt{class}} \ (\textsc{T-fin}) \qquad \frac{\Gamma \vdash e{:}T \quad \Gamma \vdash T \leq T'}{\Gamma \vdash e{:}T'} \ (\textsc{T-sub}) \qquad \frac{\Gamma \vdash e_1{:}T_1 \quad \Gamma \vdash e_2{:}T_2}{\Gamma \vdash e_1;e_2{:}T_2} \ (\textsc{T-seq})$$

$$\frac{\begin{array}{c}\Gamma \vdash e{:}T \\ \Gamma \vdash T \leftrightarrow T'\end{array}}{\Gamma \vdash (\texttt{view}\ T')e{:}T'} \ (\textsc{T-view}) \qquad \frac{\begin{array}{c}\Gamma \vdash e{:}T \\ \mathsf{ftype}(\Gamma,T,f) = T_f\end{array}}{\Gamma \vdash e.f{:}T_f} \ (\textsc{T-get}) \qquad \frac{\begin{array}{c}\Gamma \vdash x{:}T \quad \mathsf{ftype}_{decl}(\Gamma,T,f) = T_f^{decl} \\ T_f^{decl}\{\!\{\Gamma;\ T/\texttt{this!}\}\!\} = T_f \quad \Gamma \vdash e{:}T_f\end{array}}{\Gamma \vdash x.f = e{:}T_f} \ (\textsc{T-set})$$

$$\frac{\begin{array}{c}\Gamma \vdash T \quad \Gamma \vdash \overline{e}{:}\overline{T} \\ \mathsf{ftype}_{decl}(\Gamma,T,\overline{f}) = \texttt{[final]}\ \overline{T'} \\ \overline{T} = \overline{T'}\{\!\{\Gamma;\ T!/\texttt{this!}\}\!\}\end{array}}{\Gamma \vdash \texttt{new}\ T(\overline{f} = \overline{e}){:}T!} \ (\textsc{T-new}) \qquad \frac{\begin{array}{c}n = \#(\overline{e}) = \#(\overline{x}) \quad \Gamma \vdash e_0{:}T_0 \quad \Gamma \vdash \overline{e}{:}\overline{T'} \\ \mathsf{mtype}(\Gamma,T_0,m) = (\overline{x}{:}\overline{T}) \rightarrow T_{n+1} \\ \forall i \in 1..n+1.\ T_i\{\!\{\Gamma;\ T_0/\texttt{this!}\}\!\} = T'_i\end{array}}{\Gamma \vdash e_0.m(\overline{e}){:}T'_{n+1}} \ (\textsc{T-call})$$

**Figure 12.** Static semantics

fined, shares with the original view $P''.\overline{C'}!$, and is a subtype of the target type $P'!.\overline{C}$. Therefore, as long as the source type and the target type are shared, a view change expression is well-formed.

$$\mathsf{view}(\langle \ell, P! \rangle, P'!.\overline{C}) = \langle \ell, P'.\overline{C'}! \rangle$$
$$\text{where } P = P''.\overline{C'} \text{ and } \vdash P'! \leftrightarrow P''!$$

## 4.8 Soundness

In the soundness proof, expressions are typed using a typing environment $\lfloor H,R \rfloor$ constructed from the heap $H$ and the reference set $R$, which contains aliasing information for fields. Figure 16 shows the definition of $\lfloor H,R \rfloor$.

A run-time configuration is well-formed, represented as $\vdash e,H,R$, shown in Figure 17, if $e$ has no free variables, all the references in $e$ are included in $R$, references with the same location in $R$ have shared views, and the type of the value stored in a field is consistent with at least one view of the containing object.

We prove the following soundness theorem of the J&$_h$ core language, using the standard technique of proving progress and subject reduction [46].

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

LEMMA 4.2. *(Progress) If* $\vdash e,H,R$, *and* $\lfloor H,R \rfloor \vdash e{:}T$, *then either* $\exists v$, *such that* $e = v$, *or* $\exists e',H',R'$, *such that* $e,H,R \longrightarrow e',H',R'$.

$$\frac{\circ \vdash \overline{L}\ \mathsf{ok} \quad \emptyset \vdash e{:}T \quad \emptyset \vdash T \quad \sqsubset^{+} \text{ acyclic}}{\vdash \langle \overline{L}, e\rangle\ \mathsf{ok}} \quad \text{(P-OK)}$$

$$\frac{\begin{array}{c}\forall C'.\ CT'(P.C.C') \neq \bot \Rightarrow P.C \vdash CT'(P.C.C')\ \mathsf{ok} \\ P.C \vdash \overline{F}\ \mathsf{ok} \quad P.C \vdash \mathsf{ownMethods}(P.C)\ \mathsf{ok} \quad P \vdash ES\ \mathsf{ok} \\ \forall P_i.\ \vdash P.C \sqsubset^{+} P_i \Rightarrow \vdash P.C \text{ conforms to } P_i \\ \forall m.\ \#(\mathsf{mbodies}(P,m)) \leq 1\end{array}}{P \vdash \mathtt{class}\ C\ ES\ \{\overline{L}\ \overline{F}\ \overline{M}\}\ \mathsf{ok}} \quad \text{(L-OK)}$$

$$\frac{\begin{array}{c}T \neq \circ \quad \mathtt{this}{:}P \vdash T \quad \mathsf{paths}(T) \subseteq \{\mathtt{this}\} \\ \neg\mathsf{prefixExact}_0(T)\end{array}}{P \vdash \mathtt{extends}\ T\ \mathsf{ok}} \quad \text{(EXT-OK)}$$

$$\frac{\mathtt{this}{:}P \vdash T \quad T\{\!\{\emptyset;\ P/\mathtt{this}\}\!\} = P.C}{P \vdash \mathtt{shares}\ T\ \mathsf{ok}} \quad \text{(SH-OK)}$$

$$\frac{\begin{array}{c}CT'(P) = \mathtt{class}\ C\ ES\ \{\overline{L}\ \overline{F}\ \overline{M}\} \\ CT'(P') = \mathtt{class}\ C'\ ES'\ \{\overline{L'}\ \overline{F'}\ \overline{M'}\} \\ \forall i,j.\ \begin{pmatrix} L_i = \mathtt{class}\ D\ ES_i\ \ldots \\ \wedge L'_j = \mathtt{class}\ D\ ES'_j\ \ldots \end{pmatrix} \Rightarrow ES_i = ES'_j \\ \forall i,j.\ \begin{pmatrix} M_i = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\} \\ \wedge M'_j = T'_{n+1}\ m(\overline{T'}\ \overline{x'})\ \{e'\} \end{pmatrix} \Rightarrow P \vdash M_i \text{ overrides } M'_j\end{array}}{\vdash P \text{ conforms to } P'}$$

$$\frac{\begin{array}{c}M = T_{n+1}\ m(\overline{T}\ \overline{x})\ \ldots\ \{e\} \\ M' = T'_{n+1}\ m(\overline{T'}\ \overline{x'})\ \ldots\ \{e'\} \\ \#(\overline{x}) = \#(\overline{x'}) = \#(\overline{y}) \quad \overline{y} \cap (\overline{x} \cup \overline{x'}) = \emptyset \\ \Gamma = \mathtt{this}{:}P, \overline{y}{:}\overline{T}\{\overline{y}/\overline{x}\} \quad \vdash \Gamma\ \mathsf{ok} \\ \Gamma \vdash \overline{T}\{\overline{y}/\overline{x}\} \approx \overline{T'}\{\overline{y}/\overline{x'}\} \quad \Gamma \vdash T_{n+1}\{\overline{y}/\overline{x}\} \approx T'_{n+1}\{\overline{y}/\overline{x'}\}\end{array}}{P \vdash M \text{ overrides } M'}$$

$$\frac{\begin{array}{c}\mathtt{this}{:}P \vdash T \quad \mathsf{paths}(T) \subseteq \{\mathtt{this}\} \\ \neg\mathsf{prefixExact}_0(T) \quad \mathtt{this}{:}P \vdash e{:}T\end{array}}{P \vdash [\mathtt{final}]\ T\ f = e\ \mathsf{ok}} \quad \text{(F-OK)}$$

$$\frac{\begin{array}{c}\Gamma = \mathtt{this}{:}P, \overline{x}{:}\overline{T} \quad \vdash \Gamma\ \mathsf{ok} \quad n = \#(\overline{x}) \quad x_0 = \mathtt{this} \\ \Gamma \vdash T_{n+1} \quad \Gamma \vdash e{:}T_{n+1} \quad \mathsf{FV}(e) \subseteq \{x_0, \overline{x}\} \\ \mathsf{paths}(\overline{T}, T_{n+1}) \subseteq \{\mathtt{this}\}\end{array}}{P \vdash T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\}\ \mathsf{ok}} \quad \text{(M-OK)}$$

**Figure 13.** Program typing

$$\begin{array}{llll}
\text{heaps} & H & ::= & \emptyset \mid H, \langle \ell, f\rangle \mapsto v \\
\text{reference sets} & R & ::= & \emptyset \mid R, v \\
\text{evaluation contexts} & E & ::= & [\cdot] \mid E.f \mid E.f = v \mid E;\ e \\
 & & & \mid E.m(\overline{e}) \mid v.m(\overline{v}, E, \overline{e}) \\
 & & & \mid \mathtt{new}\ TE(\overline{f} = \overline{e}) \\
 & & & \mid \mathtt{new}\ P!.\overline{C}(\overline{f} = \overline{v}, f = E, \overline{f'} = \overline{e}) \\
 & & & \mid (\mathtt{view}\ TE)e \mid (\mathtt{view}\ P!.\overline{C})E \\
\text{type evaluation contexts} & TE & ::= & TE.C \mid E.\mathtt{class} \mid P[TE] \mid TE!
\end{array}$$

**Figure 14.** Definitions for operational semantics

$$\boxed{e, H, R \longrightarrow e', H', R'}$$

$$\frac{e, H, R \longrightarrow e', H', R'}{E[e], H, R \longrightarrow E[e'], H', R'} \quad \text{(R-CONG)}$$

$$\frac{\begin{array}{c}H(\ell, f) = v \quad \mathsf{ftype}(\emptyset, P!, f) = P'!.\overline{C} \\ R' = R, \mathsf{view}(v, P'!.\overline{C})\end{array}}{\langle \ell, P!\rangle.f, H, R \longrightarrow \mathsf{view}(v, P'!.\overline{C}), H, R'} \quad \text{(R-GET)}$$

$$\frac{H' = H[\langle \ell, f\rangle := v]}{\langle \ell, P!\rangle.f = v, H, R \longrightarrow v, H', R} \quad \text{(R-SET)}$$

$$\frac{\mathsf{mbodies}(P,m) = \{T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\}\} \quad n = \#(\overline{v}) = \#(\overline{x})}{\langle \ell, P!\rangle.m(\overline{v}), H, R \longrightarrow e\{\langle \ell, P!\rangle/\mathtt{this}, \overline{v}/\overline{x}\}, H, R} \quad \text{(R-CALL)}$$

$$\frac{\ell\ \mathtt{fresh} \quad H' = H[(\ell, \overline{f}) := \overline{v}] \quad R' = R, \langle \ell, P.\overline{C}!\rangle}{\mathtt{new}\ P!.\overline{C}(\overline{f} = \overline{v}), H, R \longrightarrow \langle \ell, P.\overline{C}!\rangle, H', R'} \quad \text{(R-ALLOC)}$$

$$v;\ e, H, R \longrightarrow e, H, R \quad \text{(R-SEQ)}$$

$$\frac{R' = R, \mathsf{view}(v, P!.\overline{C})}{(\mathtt{view}\ P!.\overline{C})v, H, R \longrightarrow \mathsf{view}(v, P!.\overline{C}), H, R'} \quad \text{(R-VIEW)}$$

**Figure 15.** Small-step operational semantics

$$\frac{\begin{array}{c}\langle \ell, P!\rangle \in R \quad \vdash \mathsf{ftype}(\emptyset, P, f) \trianglelefteq P_f \\ \langle \ell', P'!\rangle = \mathsf{view}(H(\ell, f), P_f) \quad \mathtt{final}\ T_f\ f = e \in \mathsf{fields}(P)\end{array}}{\langle \ell, P!\rangle.f = \langle \ell', P'!\rangle \in \lfloor H, R\rfloor}$$

**Figure 16.** Runtime typing environments

$$\frac{\begin{array}{c}\mathsf{FV}(e) = \emptyset \quad \mathsf{refs}(e) \subseteq R \\ \langle l, P!\rangle, \langle l, P'!\rangle \in R \Rightarrow \vdash P! \leftrightarrow P'! \\ H(\ell, f) = v \Rightarrow \exists P\ .\ \langle \ell, P!\rangle \in R \wedge \vdash v{:}\mathsf{ftype}(\emptyset, P!, f)\end{array}}{\vdash e, H, R}$$

**Figure 17.** Runtime configuration well-formedness

LEMMA 4.3. *(Subject reduction) If* $\vdash e, H, R$*, and* $\lfloor H, R\rfloor \vdash e{:}T$*, and* $e, H, R \longrightarrow e', H', R'$*, then* $\vdash e', H', R'$ *and* $\lfloor H', R'\rfloor \vdash e'{:}T$.

Lemma 4.2 is proved by structural induction on $e$. In order to prove Lemma 4.3, we need to first prove several preliminary lemmas, some of which are more related to sharing than others.

J&$_h$ contains dependent types, and a value substitution (substituting a value for a variable, e.g., for evaluating a method call) might affect the types as well. Lemma 4.4 states that value substitutions do not change sharing relationships between types.

LEMMA 4.4. *If* $\Gamma, x{:}T_x \vdash T_1 \leftrightarrow T_2$*, and* $\Gamma, x{:}T_x \vdash v{:}T_x$*, then* $\Gamma\{v/x\} \vdash T_1\{v/x\} \leftrightarrow T_2\{v/x\}$.

PROOF: By induction on the derivation of $\Gamma, x{:}T_x \vdash T_1 \leftrightarrow T_2$. □

As mentioned in Section 4.7, the auxiliary function view that implements the semantics of view changes works when the source type and the target type are shared. For explicit view change operations, this is ensured by T-VIEW. For implicit view changes that are triggered by field accesses (see R-GET in Figure 15), Lemma 4.5 ensures that they are also safe.

LEMMA 4.5. *If* $\vdash P! \leftrightarrow P'!$, *and* $\mathsf{ftype}(\emptyset, P!, f) = T_f$, *and* $\mathsf{ftype}(\emptyset, P'!, f) = T'_f$, *then* $\vdash T_f \leftrightarrow T'_f$.

PROOF: It is trivial, if the declared type of field $f$ is not dependent. Otherwise, it can be proved using Lemma 4.6. □

LEMMA 4.6. *If* $\vdash P_1!.\overline{C_1} \leftrightarrow P_2!.\overline{C_2}$, *and* $\vdash P[P_1!.\overline{C_1}]$, *and* $\vdash P[P_2!.\overline{C_2}]$, *then* $\vdash P[P_1!.\overline{C_1}] \leftrightarrow P[P_2!.\overline{C_2}]$.

PROOF: By induction on the derivation of $\vdash P_1!.\overline{C_1} \leftrightarrow P_2!.\overline{C_2}$. □

With these lemmas, subject reduction (Lemma 4.3) is proved by induction on the derivation of $\lfloor H, R \rfloor \vdash e{:}T$. Then the soundness theorem follows directly. The proofs can be found in the companion technical report [38].

## 5. Implementation

We have implemented a prototype compiler for the J&$_h$ language using the Polyglot compiler framework [29]. The compiler is a 4700-LOC (lines of code, excluding empty lines, comments, and automatically generated parser code) extension of the J& compiler [30]. The target language of the compiler is Java. There is also a 4600-LOC run-time system, most of which is a custom classloader implemented using the ASM bytecode manipulation framework [7].

The compiler and the run-time system are generally similar to those of the J&$_s$ language, though simplified by the absence of masked types and sharing constraints. Synthesis of shadow classes by the run-time system is a new feature but is similar to the synthesis of implicit classes [28].

### 5.1 Type checking

Type checking is similar to that in the J& compiler, except that the type system needs to prove sharing relationships between types. The type system collects sharing relationships from sharing declarations, recursively establishes sharing relationships to all known nested classes and packages, and forms the sharing relation via the reflexive, symmetric, and transitive closure. Type checking is modular and sound. It is also conservative: a true sharing relationship might not be recognized by the type system if it requires knowing the declarations in some derived family that has not been checked yet. In that case, the programmer must break a complex view change operation into multiple view changes that take smaller steps. This arguably has some documentation value.

### 5.2 Synthesizing shadow classes

Translation in the J&$_h$ compiler is scalable, in the sense that the amount of code generated by the compiler is proportional to the size of the source code. Therefore, no Java target code is produced for shadow classes. Instead, this is done lazily at run time.

When a nested class is loaded at run time, the classloader in the run-time system checks whether it is an originating class. If so, code for a shadow class is synthesized in each base family that is shared with the originating family. The implementation does not copy shadow-method code into shadow classes, but only generates one-line dispatch methods that call corresponding shadow methods contained in the originating class.

Recall from Section 3.4 that shadow classes are named specially in the source code, with the name of the originating class embedded in the syntax. For each explicit occurrence of a shadow class, the compiler generates code that calls the run-time system to load the originating class, triggering run-time synthesis of the shadow class. Moreover, the name of a shadow class is mangled to include the name of the originating class, to avoid name conflicts.

### 5.3 Supporting views

The compiler and run-time system supports views and view changes in a way that is almost identical to J&$_s$. Each J&$_h$ object is referenced indirectly through a *reference object*, which contains a pointer to the run-time representation of the view associated with the reference. A view change operation is translated to generating a new reference object pointing to the same J&$_h$ object, with a new view that is compatible with the target type. Reference objects are cached to improve performance.

The behavior of the J&$_h$ object is determined by the view. Method calls are dispatched on the views, and run-time type inspection is also based on the views. The only exception is field accesses. With heterogeneous sharing, a field with an unshared type has multiple copies, each for a view of the containing object, and therefore field accesses depend on views. However, with homogeneous sharing, every field is shared, and there is only one copy to access regardless of the view. Therefore field accesses in J&$_h$ are faster, while other operations have performance similar to that in J&$_s$. See Section 6.1 for some performance results.

### 5.4 Optimization of field accesses

As formalized in R-GET in the operational semantics, field accesses in J&$_h$ may trigger implicit view changes. However, not every access requires a view change, so the J&$_h$ compiler implements a static analysis to identify implicit view changes that may be elided without breaking type safety.

A field access does not need an implicit view change in several cases:

- The field type does not depend on the view of the container object.
- An explicit view change is applied immediately after the field access.
- The field is used at a place where its exact view is not important. For example, in an expression `x.f.g`, the field access `x.f` may not need an implicit view change, as long as the second field `g` is declared in a class that does not have a shared superclass.

### 5.5 Run-time type inspection

Sharing allows an object to have several different views at run time. J&$_h$ supports *view casts* and the `viewableas` operator, which are similar to type casts and the `instanceof` operator, to dynamically inspect different views of a J&$_h$ object. These operators are illustrated in the following code based on the example from Section 2.1:

```
List l = new ArrayList();
l.add(new Button());
Object o = l.get(0);
if (o viewableas PrettyButton!) {
  PrettyButton! b = (viewcast PrettyButton!)o;
  b.draw(); // the new method in PrettyButton
}
```

The static type of the object stored in the list is `Object`, which is not a shared type of `PrettyButton!`, so a view cast has to be used for the J&$_h$ run-time system to attempt the view change to `PrettyButton!`. Like type casts in Java, view casts in J&$_h$ can fail at run time with an exception if the current view of the object and the target type are not shared. By contrast, a statically type-checked view change never fails at run time, whether explicitly written or implicitly triggered by a field access.

### 5.6 Java compatibility

The J&$_h$ implementation has roughly the same Java compatibility as the J&$_s$ implementation. A J&$_h$ program may use existing Java code, and a J&$_h$ class may extend a Java class or implement Java interfaces. However, shared classes in J&$_h$ must extend the same Java class, due to the absence of multiple inheritance in Java.

Arrays of J&$_h$ objects are treated as objects having fields of the same type, and therefore, array accesses may also trigger implicit view changes.

The current J&$_h$ implementation does not support most of the Java 1.5 features, which seem to be orthogonal to sharing. Covariant return types are supported, partly because the underlying Java standard library already has covariant return types. Support for generics would help replace view casts with type-safe view changes. In the previous code example, if the list `l` were declared with type `List`$\langle$`Button!`$\rangle$, the view cast operation could then be replaced with a statically type-safe view change operation. Adding support for generics to J&$_h$ appears feasible, but we leave this as future work.

|  | view change | virtual call | static call | field read | field write | allo-cation |
|---|---|---|---|---|---|---|
| J&$_s$ | 17.6 | 10.3 | 1.09 | 5.92 | 5.31 | 26.1 |
| J&$_h$ | 18.3 | 9.26 | 1.09 | 3.13 | 2.97 | 26.3 |

**Table 1.** Microbenchmarks: average time per operation, in ns.

## 6. Experience

### 6.1 Performance results

***Microbenchmarks.*** We compare the performance of the J&$_h$ implementation against J&$_s$, using some microbenchmarks to measure individual object operations. Every microbenchmark runs one operation $10^8$ times in a loop. Table 1 shows the results. The testing hardware is a Thinkpad X200 with Intel L9400 CPU and 2GB memory, and the software environment consists of Windows Vista, Cygwin, and JVM 1.6.0_13.

The results confirm that object operations in the implementations of J&$_s$ and J&$_h$ have similar performance, but field accesses with homogeneous sharing are almost twice as fast as J&$_s$ field accesses. With homogeneous sharing, field accesses are no longer view-dependent, and therefore accessor methods can be implemented in the same class where fields are located.

***Jolden benchmarks.*** We tested the J&$_h$ implementations with the jolden benchmarks [8] to study the performance overhead for code that does not use the new extensibility features of class sharing. All ten benchmarks, with few changes, are tested with five language implementations: Java, J& [30], J& with the custom classloader, J&$_s$ [36], and J&$_h$ with homogeneous family sharing. Table 2 compares the results. The testing environment is the same as that for the microbenchmarks.

The results of the jolden benchmarks confirm that homogeneous family sharing has a lower performance overhead compared to heterogeneous sharing. The J&$_s$ times show a 25% slowdown versus the classloader-based J& (without sharing) implementation, and 100% versus Java. On the other hand, J&$_h$ has a better performance compared to J&$_s$: the two overhead numbers drop to 17% and 88%. This is consistent with the microbenchmarking results.

The current implementation of J&$_h$ still introduces noticeable overhead compared to Java, because of more complex subtyping relationships, indirections through reference objects, and the translation of field accesses into accessor method calls. Having Java as a target language does limit performance. For J&$_s$ [36], we suggested that performance could be improved with a lower-level target language, al-

| | bh | bisort | em3d | health | mst | perimeter | power | treeadd | tsp | voronoi |
|---|---|---|---|---|---|---|---|---|---|---|
| Java | 1.30 | 0.34 | 0.16 | 0.26 | 0.63 | 0.21 | 0.64 | 0.14 | 0.08 | 0.29 |
| J& [30] | 13.39 | 1.48 | 0.33 | 6.11 | 4.06 | 2.93 | 2.10 | 2.10 | 0.22 | 6.96 |
| J& with classloader | 1.78 | 0.45 | 0.24 | 0.55 | 0.93 | 0.47 | 0.83 | 0.21 | 0.12 | 0.57 |
| J&$_s$ [36] | 2.20 | 0.58 | 0.24 | 0.85 | 1.26 | 0.45 | 0.89 | 0.38 | 0.14 | 0.75 |
| J&$_h$ | 2.01 | 0.55 | 0.22 | 0.79 | 1.21 | 0.47 | 0.82 | 0.37 | 0.13 | 0.64 |

**Table 2.** Results for the jolden benchmarks. Average time over ten runs, in seconds.

lowing implementation techniques similar to those used for C++; the same should also apply to the J&$_h$ implementation.

### 6.2 Lambda compiler

We experimented with in-place translation in a λ-calculus compiler. This is not a large example, but uses the language features in a sophisticated way—combining multiple levels of sharing and family-level intersection. The example is inspired by the Polyglot framework, and it encapsulates most of the interesting issues that arise in making Polyglot extensible, while demonstrating the advantages of homogeneous family sharing.

***Implementing translations for sums and pairs.*** The compiler example translates the λ-calculus extended with sums and pairs to the simple lambda calculus. A way to accomplish this goal as an in-place translation was first demonstrated for heterogeneous sharing [36], so this example offers a way to compare the expressive power of homogeneous and heterogeneous sharing.

The compiler is implemented as four families of classes. The `base` family describes simple λ-calculus as the target language. Two derived families, `sum` and `pair`, share AST classes with the base family but extend it with sums and pairs respectively. They also implement in-place translation to the simple λ-calculus. The last derived family, `sumpair`, composes the `sum` and `pair` families, leading to a compiler that supports both sums and pairs.

***Homogeneous vs. heterogeneous sharing*** Figure 18 shows the structures of the compiler when implemented with heterogeneous sharing and with homogeneous sharing.

The J&$_s$ version has about 250 lines. Sharing is declared between individual pairs of classes, represented with dashed arrows that go across family boundaries. The class hierarchy is more complex with these interfamily relationships, and several class declarations have to be written just to declare sharing, making the code less scalable. Out of the 23 class declarations, 12 are just for sharing declarations. Moreover, the existence of new subclasses (`Pair` and `Case`) in the `pair` and `sum` families means the `Exp` type cannot be shared. Therefore, all subexpression fields with type `Exp` must be masked in sharing relationships. There are in total 17 masked types in the J&$_s$ code, and 3 sharing constraints. The extra syntax increases the annotation burden for the programmer.

The J&$_h$ version has about 200 lines. There are only three sharing declarations, shown in the right-hand side of Figure 18, and they are all between families. None of the 12 class declarations in the J&$_s$ version that were needed for declaring sharing are necessary, so extending is simpler and more scalable. For example, the declaration of `sumpair` is reduced to just one line:

```
class sumpair extends pair & sum shares base { }
```

In addition, none of the masked types and sharing constraints from the J&$_s$ version are needed.

The sharing declaration `shares base` enables `sumpair` to share with `base`, as well as with `pair` and `sum`, because of transitivity. Without this declaration, `sumpair` would inherit from `pair` and `sum`, but would not adapt them.

The comparison shows that homogeneous sharing is simpler and more scalable, and has a lower annotation burden. The annotation burden is low even in an absolute sense given what is achieved. Therefore, homogeneous sharing seems more likely to be adopted by ordinary programmers.

### 6.3 Polyglot

Following the same implementation strategy just outlined, we ported the Polyglot compiler framework version 2.4.0, originally written in Java, to J&$_h$. We also ported two small Polyglot-based language extensions to explore using homogeneous sharing to implement compiler translations. (Previous work [30] presents a port of Polyglot 1.x to the J& language, for the purpose of demonstrating extension composition, but it cannot support in-place translation).

The Java version of the base Polyglot compiler has 59 kLOC. The J&$_h$ port is substantially shorter, at 48 kLOC. The code becomes shorter through the elimination of the delegate and extension-object design patterns [29] that support extensibility. With family inheritance and homogeneous sharing, even more extensibility is obtained than from these design patterns, because the extensibility mechanism applies throughout the compiler rather than just where the design patterns were used.

As in the lambda compiler example, homogeneous sharing allows us to declare an extension that is shared with the base compiler. For example, we implemented a `covarRet` extension, which extends the base compiler with covariant return types. Because it extends the type system, this exten-

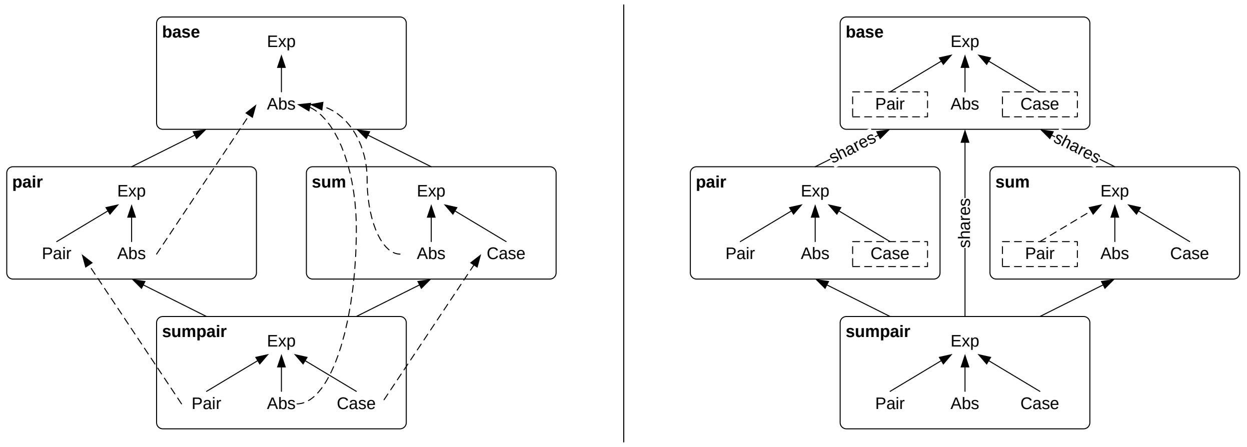

**Figure 18.** Compiler structure with heterogeneous (left) and homogeneous sharing (right). Translator and some AST nodes not shown.

sion makes a particularly interesting example. The extension is implemented by the `covarRet` package, declared to share with the base compiler:

```
package covarRet shares polyglot;
```

The `covarRet` extension parses source code, which may include methods declared with covariant return types, generates an AST (built of source family objects), type-checks it, and then translates away any covariant return types to obtain an AST in the base language (the target family), which is Java 1.4. The translated AST is then type-checked in the base language, and finally output is generated.

As in the `IMP` example of Section 3.5, the translation from `covarRet` to the base language happens in two steps. First, the AST is transformed to remove covariant return types, but the generated objects remain in the `covarRet` family. Shadow classes ensure this can always be done, even if the target language is a "sibling" family with additional AST nodes not present in the source language. Second, a view change from the source family to the target family is applied to the root of the translated AST. The view change lazily moves the whole AST, including type annotations, to the target family. Subsequent type checking and post-processing in the compiler uses the base language view, which (correctly) does not support covariant returns.

Homogeneous sharing offers some advantages for implementing this extension. First, the extension is somewhat shorter than the original Java implementation of the same extension: 166 rather than 226 lines for `covarRet`, as the result of eliminating design patterns. Second, the Java implementation must reconstruct the types of all expressions after translation, in order to type-check the translated AST. This reconstruction is avoided in the J&$_h$ implementation, because the final view change automatically shifts source-language objects representing types to the corresponding target language classes. This shows that homogeneous sharing provides more scalable extensibility than was possible through the Polyglot design patterns, which apply mainly to AST nodes.

Another ported extension was `carray`, which supports constant arrays. Compared to `covarRet`, which just extends the type system, `carray` also extends the grammar, and introduces a new AST node class `ConstArrayTypeNode`. If we were to port `carray` with J&$_s$, this new AST node class would make the code much more complicated. All the fields that might store an instance of `ConstArrayTypeNode` would have to be masked, because heterogeneous sharing does not allow two corresponding types to be shared if one of them has a new subclass. By contrast, with a shadow class implicitly added into the base family, homogeneous sharing makes porting `carray` quite simple: the J&$_h$ version has 214 LOC, and the Java version has 217 LOC, both excluding comments, empty lines, and automatically generated parser code. In the case the J&$_h$ code does not become significantly shorter, because the original extension does not implement any translation.

The performance of the J&$_h$ version of Polyglot is reasonable. Because Polyglot has many classes, and the current J&$_h$ run-time system needs to produce bytecode for generated classes, startup overhead for class loading in the current system is high compared to Java. To ignore this artificial discrepancy, we compared the compile time of the two compilers on some small programs after class loading was complete. The Java version took 149ms on average to compile these programs, whereas the J&$_h$ version took 207ms. The performance hit is modest, while writing the compiler in J&$_h$ adds even more extensibility.

## 7. Related work

### 7.1 Heterogeneous sharing

The most closely related work is our previous work on the J&$_s$ language, which has heterogeneous class sharing [36]. Section 3.7 compares the two approaches to sharing in detail.

## 7.2 Adaptation

The adapter design pattern [17] 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 sharing does.

Expanders [45] are a mechanism for adaptation. New fields, methods, and superinterfaces can be added into existing classes. Expanders are more expressive than *open classes* [10], which can only add methods. Method dispatch is statically scoped, so expanders do not change the behavior of existing clients. J&$_h$ supports upgrading existing clients without code change by allowing the shared derived family to override classes in the base family, and by providing shadow classes in the base family.

CaesarJ [1, 23] 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. They also share the similar limitation that existing client code cannot acquire the extended behavior without code change. Multiple inheritance in CaesarJ makes wrapper selection ambiguous; J&$_h$ disambiguates via views.

Fickle$_{\text{III}}$ [12] has a *re-classification* operation for objects to change their classes. Re-classifications are similar to view changes in J&$_h$, but directly change the behavior of all existing references to the object; therefore, effects are needed to track the change, adding to the annotation burden. Fickle$_{\text{III}}$ does not support class families.

*Chai*$_3$ [41] allows traits to be dynamically substituted to change object behaviors, similar to view changes. 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 previous work on adaptation, including pluggable composite adapters [24], object teams [18], and delegation layers [34], also has some notion of families of classes. However, these mechanisms either do not support method overriding and dynamic dispatch between the adapter and adaptee families [24], or have a weaker notion of families in which programmers have to manually "wire" relationships between the base family and the delegation family [18, 34]. These mechanisms all use lifting and lowering, introduced in [24], to convert between adapter and adaptee classes. Lifting and lowering are similar to view changes, but are not symmetric and do not support late binding, as they are based on the static type of the object.

## 7.3 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 mechanisms, families of classes are disjoint, and in-place extensibility is not provided.

Virtual classes [9, 13, 16, 21, 22] are inner classes that can be overridden just like methods. Path-dependent types are used to ensure type safety, proved in [16] and [9].

Nested inheritance [28] supports overriding of nested classes, which are similar to virtual classes. Nested intersection [30] adds and generalizes intersection types [11, 39] in the context of nested inheritance to provide the ability to compose extensions. A family-level multiple inheritance mechanism based on virtual classes has been explored by Ernst in the context of `gbeta` [14].

Variant path types [20] support family inheritance without dependent types, using a different style of exact types and relative path types to ensure type safety.

Mixin layers [40] generalize *mixins* [4], which are classes that can be instantiated with different superclasses. 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. However, they do not provide family polymorphism.

Virtual types [6, 19, 43, 44] 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. *Views* in Scala do not provide adaptation; they are implicitly-called conversion functions that create new instances in their target types.

Nielsen and Ernst [27] present a virtual machine that natively supports family inheritance. It would also be interesting to explore native implementation of family sharing.

## 7.4 Dynamic software updating

J&$_h$ supports safe dynamic software updating without downtime, in much the same way as in J&$_s$. Objects of the base family may be upgraded to new views in the derived family, and therefore obtain new behaviors while the software system is still running. Usually only a few root objects need to be explicitly upgraded, while other objects that are reachable from them will be automatically updated when they are accessed. This makes the updating code easier to write.

On the other hand, mechanisms for dynamic software updating [2, 25, 26, 42], by design do not preserve the old behavior of the software before the update, whereas sharing captures both the existing and the updated behaviors in the form of views. Sharing therefore provides additional functionality: safe switching between different behaviors at run time.

Barr and Eisenbach [2] propose a framework for dynamic update of Java components, 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.

JVOLVE [42] supports dynamic software updating with no overhead during steady-state execution, but it requires changes to many different components of the Java virtual machine, rather than to just the classloader as in J&$_h$.

Ginseng [26] uses static analysis to find proper timing for a given global update, and is later generalized to work with multi-threaded programs [25]. Abstract and concrete types in Ginseng bear some resemblance to inexact and exact types in J&$_h$: abstractly typed variables allow values of different concrete types, and inexactly typed variables may store objects with different exact views.

## 8. Conclusions

Homogeneous family sharing generalizes class sharing so that it operates at the level of families of interacting classes, analogously to the way nested inheritance generalizes ordinary class inheritance. Shadow classes and other new mechanisms enable a derived family to safely and homogeneously share with a base family, while still extending the base family with new classes. Shadow classes also make families open, providing new kinds of extensibility. Homogeneous sharing is formalized in a simplified language that focuses on sharing, and the type system is proved sound. Homogeneous sharing avoids complicated language mechanisms like masked types or sharing constraints that were needed by the previous heterogeneous sharing mechanism. This ease of use made it feasible to port and evaluate some substantial software and extensions to it. Our experience is that homogeneous family sharing gives expressive power comparable to that of heterogeneous sharing, but is simpler, more concise, and easier to reason about.

## Acknowledgments

We thank Anthony Jawad for helping develop the port of Polyglot to J&$_h$. We also thank Robert Grimm and the anonymous reviewers for their useful comments.

This research was supported in part by ONR Grant N00014-09-1-0652, by NSF Grants 0541217 and 0627649, and by a grant from Microsoft Corporation. The views and conclusions here are those of the authors and do not necessarily reflect those of ONR, the Navy, the NSF, or Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright annotation thereon.

## 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. ACM OOPSLA 2005*, pages 177–189, San Diego, CA, USA, October 2005.

[4] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proc. ACM OOPSLA '90*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

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

[6] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *European Conference on Object-Oriented Programming (ECOOP)*, number 1445 in Lecture Notes in Computer Science, pages 523–549. Springer-Verlag, July 1998.

[7] 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`.

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

[9] 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.

[10] 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.

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

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

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

[14] Erik Ernst. Propagating class and method combination. In *Proc. Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, number 1628 in Lecture Notes in Computer Science, pages 67–91. Springer-Verlag, June 1999.

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

[16] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proc. 33rd ACM Symp. on Principles of Programming Languages (POPL)*, pages 270–282, Charleston, South Carolina, January 2006.

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

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

[19] Atsushi Igarashi and Benjamin Pierce. Foundations for virtual types. In *Proc. Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, number 1628 in Lecture Notes in Computer Science, pages 161–185. Springer-Verlag, June 1999.

[20] 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.

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

[22] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism for object-oriented programming. In *Proc. ACM OOPSLA '89*, pages 397–406, October 1989.

[23] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *Proc. 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 90–100, Boston, Massachusetts, March 2003.

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

[25] Iulian Neamtiu and Michael Hicks. Safe and timely dynamic updates for multi-threaded programs. In *Proc. SIGPLAN 2009 Conference on Programming Language Design and Implementation*, pages 13–24, June 2009.

[26] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *Proc. SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 72–83, June 2006.

[27] Anders Bach Nielsen and Eric Ernst. Virtual class support at the virtual machine level. In *VMIL '09: Proceedings of the third workshop on Virtual Machines and Intermediate Languages*, October 2009.

[28] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proc. ACM OOPSLA 2004*, pages 99–115, October 2004.

[29] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. 12th International Compiler Construction Conference (CC'03)*, pages 138–152, April 2003. LNCS 2622.

[30] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software composition. In *Proc. ACM OOPSLA 2006*, 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. ACM OOPSLA 2005*, pages 41–57, October 2005.

[34] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *Proc. 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. 36th ACM Symp. on Principles of Programming Languages (POPL)*, pages 53–65, January 2009.

[36] Xin Qi and Andrew C. Myers. Sharing classes between families. In *Proc. SIGPLAN 2009 Conference on Programming Language Design and Implementation*, pages 281–292, 2009.

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

[38] Xin Qi and Andrew C. Myers. Homogeneous family sharing: technical report. Technical report, Computing and Information Science, Cornell University, July 2010. `http://hdl.handle.net/1813/15845`.

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

[40] 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.

[41] Charles Smith and Sophia Drossopoulou. Chai: Traits for Java-like languages. In *Proceedings of 19th European Conference on Object-Oriented Programming (ECOOP'05)*, pages 453–478, 2005.

[42] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: A VM-centric approach. In *Proc. SIGPLAN 2009 Conference on Programming Language Design and Implementation*, June 2009.

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

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

[45] 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.

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