Homogeneous Family Sharing:
Technical report
Xin Qi  Andrew C. Myers
{qixin, 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&₉ 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&₉ calculus. The soundness of the J&₉ type system is proved. The J&₉ 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&₉.

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 [44] 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., [22, 15, 28, 30, 20]. 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&₉ language¹, which extends the J& language [30].

Class sharing as an adaptation mechanism was first introduced in the J&₄ language [37]. In J&₄, sharing is a relationship between individual classes, with some coordination at the family level. Unlike with homogeneous sharing, J&₄ 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.

¹pronounced “jet-h”
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 [36] 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&; 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& 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 [35]. Section 5 presents the language implementation, and Section 6 describes experience with it. Section 7 discusses related work, and Section 8 concludes.

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&amp;, the new PrettyButton class may be declared to share with the Button class, rather than to inherit it:

```java
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&amp; [37], but its original, heterogeneous form has limitations: first, J&amp; only allows sharing between corresponding classes—classes of the same name and related through overriding—from different families. Second, unlike family sharing in J&amp;h, class sharing in J&amp; 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 $⟨\ell, C!⟩$ 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 $⟨view T⟩e$, 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:

```java
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
```

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&amp;h, a reference $⟨\ell, C!⟩$ 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.
package GUI;
class Button {...}
class RadioButton extends Button {...}
class Window { Button close; ... }
...

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

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
ew shared class is not statically in scope.

In both J&k 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 [21, 22, 13, 16, 9].
Nested intersection supports composing families with generalized intersection types [38, 11] — 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&k 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&k 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&k preserves all three kinds of relationships in the derived family: referencing, subclassing, and sharing. Because class sharing in J&k is heterogeneous, sharing relationships are not guaranteed to be preserved; therefore, J&k 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:

```java
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:

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

Moreover, field accesses in J&k 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&k, 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 [37], which is declared for individual pairs of classes between two families. The J&k 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&k type system to be modularly type-safe without complicated mechanisms like masked types and sharing constraints [37], simplifying programming.

### 3.3 Shadow classes and shadow methods

Shadow classes are the key to homogeneous sharing. In the J&k 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.

```java
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 version 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 [37] 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 receiver 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&k 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 a While object still in the IMP_for family. Translation of If (lines 19–24) lazily generates a new If node only if its children

![Figure 4: Sketch of the IMP compiler structure. Shadow classes in dashed boxes.](image-url)
have changed. After the recursive translation is done on an AST, a single view change operation applied to the root moves the entire AST to the IMP\_while family:

```java
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 families.

### 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, $T_1.C \iff T_2.C$.

For example, a variable $b_1$ of type $\text{GUI!}.\text{Button}$ might point to an object of the $\text{GUI!}.\text{RadioButton}$ class. In either case, the view change $(\text{view} \ xlucent\text{GUI!}.\text{Button})b_1$ is type-safe, because sharing is declared between $\text{GUI!}$ and $xlucent\text{GUI}$—and therefore, all classes in $\text{GUI!}$ that are subtypes of $\text{GUI!}.\text{Button}$ have shared counterparts in $xlucent\text{GUI}$.

On the contrary, with heterogeneous sharing, in order to prove that a view change $(\text{view} \ T)e$ 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& 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 $\text{For}$ 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 $\text{IMP}_{\text{for}}$ to IMP is implemented in J&$_s$, the system would ensure that no $\text{For}$ 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 system no longer checks for unfinished translation. For example, if a $\text{For}$ object is left untranslated, it will end up as an object of the shadow class $\text{For}$ in the target family—either IMP or IMP$_{\text{while}}$—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 $\sigma$ is used for both the list $a_1, \ldots, a_n$ and the set $\{a_1, \ldots, a_n\}$, for $n \geq 0$, and $\#(\sigma)$ represents the size of $\sigma$.

A program $Pr$ is a pair $(L, e)$ of a set of class declarations $L$ and an expression $e$ (the $\text{main}$ method). Each class $C$ has a superclass declaration, which is either a normal superclass declaration $\text{extends}$ $T$, or a shared superclass declaration $\text{shares}$ $T$. 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 $\text{Object}$ in Java; $\circ$ is the single top-level class that all other classes are nested within.

Method declarations may have an extra modifier $\text{shadow}$ for declaring shadow methods. Fields may be declared $\text{final}$, 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 \( T.C! \), 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 \((\text{view } T)e\).

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 \( \vdash P_1 \subseteq P_2 \) 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 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: \( \text{super}(P) \) gives the superclass declaration of \( P \), either a normal superclass or a shared one; \( \text{shadowMethods}(P) \) collects all the shadow method declarations from \( P \), with the \( \text{shadow} \) modifier removed; the functions \( \text{fields}(P) \) and \( \text{methods}(P) \) collect all the field and method declarations from \( P \) and its superclasses; \( \text{fnames}(\mathcal{F}) \) is the set of all field names in field declarations \( \mathcal{F} \); \( \text{ftype}_{\text{decl}}(\Gamma, T, f) \) returns the declared type of field \( f \), which might be a type dependent on \text{this}, and whether the field is final or not; \( \text{ftype}(\Gamma, T, f) \) substitutes the receiver type \( T \) for \text{this}.class; \( \text{mbody}(P, m) \) looks up all the declarations of a method \( m \) that are not overridden; \( \text{mtype}(\Gamma, T, m) \) looks up the method signature. For any well-typed class \( P \), \( \text{mbody}(P, m) \) never contains more than one element (see \( L:\text{OK} \) in Figure 13).

Note that although shadow methods do not appear in \( \text{ownMethods}(P) \) 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 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.
Shared subclassing $\vdash P_1 \sqsubseteq P_2$

\[
\text{super}(P.C) = \text{shares } T
\]
\[
\frac{\vdash P_1 \sqsubseteq^* P}{\vdash P_1 \cdot C \sqsubseteq P_2.C}
\] (SHARE-DECL)

\[
\frac{\vdash P_1 \sqsubseteq P_2}{\vdash P_1 \cdot C \sqsubseteq P_2.C}
\] (SHARE-FB)

Subclassing $\vdash P_1 \sqsubseteq P_2$

\[
\text{super}(P.C) = \text{extends } T
\]
\[
\frac{\vdash P_1 \sqsubseteq^* P}{\vdash P_1 \cdot C \sqsubseteq P_2.C}
\] (SC-DECL)

\[
\frac{\vdash P_1 \sqsubseteq P_2}{\vdash P_1 \cdot C \sqsubseteq P_2.C}
\] (SC-FB)

\[
\frac{\vdash P_1 \sqsubseteq P_2}{\vdash P_1 \sqsubseteq P_2}
\] (SC-SHARE)

Figure 7: Sharing and subclassing

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.

### 4.4 Prefix types

A non-dependent prefix type $P[P']$ 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 $\text{prefix}(P,P')$. J&h generalizes prefix types to include the second case, for naming shadow classes (see Section 3.4).

\[
\text{prefix}(P,P') = \begin{cases} 
P'' & \text{if } P' = P'' \cdot C \wedge \vdash P'' \sqsubseteq^* P \\
\perp & \text{otherwise}
\end{cases}
\]

As in [31], we only consider prefix types $P[T]$ 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 $T \{ \Gamma; T_i / x \}$ substitutes $T_i$ for $x$ 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 $T \{ \Gamma; T_i / x' \}$ substitutes $T_i$ for $x$ class in $T$ and preserves the exactness of $T$: 
Class table $CT(P)$

$Pr = (L, e)$

$CT(\circ) = \text{class } \circ \text{ extends } \top \{L\}$ \hspace{1cm} (CT-OUT)

$CT(P) = \text{class } C' \text{ ES } \{L \; F \; M\}$

$L_i = \text{class } C \ldots$

$CT(PC) = L_i$ \hspace{1cm} (CT-EXP)

$CT(P) = \text{class } C' \ldots \{L \; F \; M\}$ \quad \text{class } $C \ldots \not\in L$

$\vdash P \sqsubseteq P'$

$CT(P', C) = \text{class } C \text{ ES } \{\ldots\}$

$CT(PC) = \text{class } C \text{ ES } \{\}$ \hspace{1cm} (CT-IMP)

Extended class table $CT'(P)$

$CT(P) \neq \bot$ \hspace{1cm} (CT'-NORM)

$\vdash P' \equiv^* P$

$CT(P', C) = \text{class } C \text{ ES } \{L \; F \; M\}$

$\forall P^\prime, \vdash P \sqsubseteq P' \Rightarrow CT(P', C) = \bot$

$M = \text{shadowMethods}(P', C)$

$CT'(PC) = \text{class } C \text{ ES } \{M\}$ \hspace{1cm} (CT'-SHADOW)

$CT'(P', C) = \text{class } C \text{ ES } \{\ldots\}$

$CT'(PC) = \text{class } C \text{ ES } \{\}$ \hspace{1cm} (CT'-PROP)

Figure 8: Class lookup

$T \{\Gamma; \ T_k/x\} = T'$

$\forall k. \ prefix\text{Exact}_k(T) \Rightarrow prefix\text{Exact}_k(T')$

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

Exactness is defined using $prefix\text{Exact}_k(T)$, 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 well-formedness $\Gamma \vdash T$, non-dependent type bound $\Gamma \vdash T \sqsubseteq P$, final path typing $\Gamma \vdash_{\text{final}} p: T$, final path equality $\Gamma \vdash p_1 = p_2$, type sharing $\Gamma \vdash T_1 \leftrightarrow T_2$, subtyping $\Gamma \vdash T_1 \sqsubseteq T_2$, and expression typing $\Gamma \vdash e: T$. Figure 13 defines program well-formedness.

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&k 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.
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 paths(T), which represents the set of all the access paths p in dependent types p.class that are part of type T. For example, paths(P[this.class].C) = {this}. 

Subtyping. The subtyping judgment \( \Gamma \vdash T_1 \triangleleft T_2 \) 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 Jk 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 \( T[T; \ x/x] \). See Section 4.5 for its definition.
\( \top \{ \Gamma; T_i/x \} = \top \)

\( \circ \{ \Gamma; T_i/x \} = \circ \)

\( t.c \{ \Gamma; T_i/x \} = t \{ \Gamma; T_i/x \} \cdot c \)

\( T \{ \Gamma; T_i/x \} = T' \)

\( P[T] \{ \Gamma; T_i/x \} = P[T'] \)

\( v.\text{class}(\Gamma; T_i/x) = v.\text{class} \)

\( p.\text{class}(\Gamma; T_i/x) = p'.\text{class} \)

\( p.f.\text{class}(\Gamma; T_i/x) = p'.f.\text{class} \)

\( x \neq y \)

\( y.\text{class}(\Gamma; T_i/x) = y.\text{class} \)

\( p.\text{class}(\Gamma; T_i/x) = T_p \)

\( T_p \neq p'.\text{class} \)

\( \text{type}(\Gamma, T_p, f) = T_f \)

\( p.f.\text{class}(\Gamma; T_i/x) = T_f \)

![Figure 11: Type substitution](image)

### 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 \( (\ell, f) \) of memory locations and field names to values \( v \). Heap updates are represented as \( H[(\ell, f) := v] \).

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 \( e, H, R \rightarrow e', H', R' \). Most of them are standard. \( R.\text{GET} \) 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 \( P!\bar{c} \), 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 \( (\ell, P!) \).\text{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 \( \text{view} \) defines the operational semantics for view changes. Because sharing is homogeneous, the generated run-time view \( P'!\bar{c}! \) is well defined, shares with the original view \( P''!\bar{c}! \), and is a subtype of the target type \( P'!\bar{c} \). Therefore, as long as the source type and the target type are shared, a view change expression is well-formed.

\[
\text{view}((\ell, P!), P'!\bar{c}) = (\ell, P'!\bar{c}!)
\]

where \( P = P''!\bar{c} \) and \( P'! \leftrightarrow P''! \)

### 4.8 Soundness

In the soundness proof, expressions are typed using a typing environment \( [H,R] \) constructed from the heap \( H \) and the reference set \( R \), which contains aliasing information for fields. Figure 19 shows the definition of \( [H,R] \).

A run-time configuration is well-formed, represented as \( \vdash e, H, R \), shown in Figure 20, 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 [45].

**Theorem 4.1 (Soundness)** If \( \vdash (\ell, e) \text{ ok} \), and \( \vdash e : T, \) and \( e, \emptyset, \emptyset \rightarrow \ast e', H, R, \) then either \( \exists v \), such that \( e' = v \) and \( [H,R] \vdash v : T, \) or \( \exists e'', H', R', \) such that \( e'', H, R \rightarrow e'', H', R' \).

**Lemma 4.2 (Progress)** If \( \vdash e, H, R, \) and \( [H,R] \vdash e : T \), then either \( \exists v \), such that \( e = v \), or \( \exists e', H', R', \) such that \( e, H, R \rightarrow e', H', R' \).
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash T \rightarrow T'$ (SH-REFL)</td>
<td>Type sharing $\Gamma \vdash T \rightarrow T$</td>
</tr>
<tr>
<td>$\Gamma \vdash T \rightarrow T$ (SH-SYM)</td>
<td>Type sharing $\Gamma \vdash T \rightarrow T$</td>
</tr>
<tr>
<td>$\Gamma \vdash T \vdash P$ (BD-REFL)</td>
<td>Non-dependent type bound $\Gamma \vdash T \subseteq P$</td>
</tr>
<tr>
<td>$\Gamma \vdash P, \Gamma \vdash T \vdash P$ (BD-NEST)</td>
<td>Non-dependent type bound $\Gamma \vdash T \subseteq P$</td>
</tr>
<tr>
<td>$\Gamma \vdash p, \Gamma \vdash T \vdash pT$ (BD-FIN)</td>
<td>Non-dependent type bound $\Gamma \vdash T \subseteq P$</td>
</tr>
<tr>
<td>$\Gamma \vdash T \vdash T'$ (BD-PREP)</td>
<td>Non-dependent type bound $\Gamma \vdash T \subseteq P$</td>
</tr>
<tr>
<td>$\Gamma \vdash P, \Gamma \vdash T \vdash prefix(P, T')$ (BD-PRE)</td>
<td>Non-dependent type bound $\Gamma \vdash T \subseteq P$</td>
</tr>
<tr>
<td>$\Gamma \vdash P \vdash P[T]$ (BD-PRE)</td>
<td>Non-dependent type bound $\Gamma \vdash T \subseteq P$</td>
</tr>
</tbody>
</table>

Figure 12: Static semantics
Lemma 4.3 (Subject reduction) If $\vdash e, H, R,$ and $[H, R] \vdash e : T,$ and $e, H, R \to e', H', R'$, then $\vdash e', H', R' \to e' : T$.

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.

$\text{J} \& \mathbb{h}$ 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 $\Gamma, x : T_x \vdash v : T_v$, 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 $\text{T-VIEW}$. For implicit view changes that are triggered by field accesses (see $\text{R-GET}$ in Figure 15), Lemma 4.5 ensures that they are also safe.
The compiler is a 4700-LOC (lines of code, excluding empty lines, comments, and automatically generated parser code) prototype 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).

With these lemmas, subject reduction (Lemma 4.3) is proved by induction on the derivation of \( \vdash H, R \vdash e : T \). Then the soundness theorem follows directly. See Appendix A for the proofs.

## 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-\textsc{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 \texttt{viewable\textasciitilde} operator, which are similar to type casts and the \texttt{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:

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

The static type of the object stored in the list is \texttt{Object}, which is not a shared type of \texttt{PrettyButton!}, so a view cast has to be used for the J&\(h\) run-time system to attempt the view change to \texttt{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 \texttt{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.
We compare the performance of the J&₇₉ implementation against J&₈, using some microbenchmarks to measure individual object operations. Every microbenchmark runs one operation 10⁸ 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&₈ and J&₇₉ have similar performance, but field accesses with homogeneous sharing are almost twice as fast as J&₈ 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&₇₉ 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&₇₉ [37], and J&₇₉ 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&₈ times show a 25% slowdown versus the classloader-based J& (without sharing) implementation, and 100% versus Java. On the other hand, J&₇₉ has a better performance compared to J&₈: the two overhead numbers drop to 17% and 88%. This is consistent with the microbenchmarking results.

The current implementation of J&₇₉ 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&₇₉ [37], we suggested that performance could be improved with a lower-level target language, allowing implementation techniques similar to those used for C++; the same should also apply to the J&₇₉ 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 \( \lambda \)-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 [37], 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 \( \lambda \)-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 \( \lambda \)-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:

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

```java
class covarRet extends 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&h, 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&h language, which has heterogeneous class sharing [37].
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 [44] 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 [23, 1] is an aspect-oriented language that supports adaptation with wrappers called aspect binders. Wrappers and expanders are similar. They both can extend wrapped classes with new states, operations, and superinterfaces; no duplicate wrappers are created for objects; and dynamic wrapper selection is similar to expander overriding. 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.

FickleIII [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. FickleIII does not support class families.

CaesarJ [23, 1] allows traits to be dynamically substituted to change object behaviors, similar to view changes. However, CaesarJ 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 [21, 22, 13, 16, 9] 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 [38, 11] 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 [39] 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 [42, 6, 43, 19] 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&c. 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, 41, 26, 25], 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.

JOLVE [41] 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 inexact types 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.

References


\[ \langle \ell, P! \rangle \in R \vdash \text{ftype}(0, P, f) \preceq P_f \]
\[ \langle \ell', P'! \rangle = \text{view}(H(\ell, f), P_f) \quad \text{final } T_f \ f = e \in \text{fields}(P) \]
\[ \langle \ell, P! \rangle.f = \langle \ell', P'! \rangle \in [H, R] \]

Figure 19: Runtime typing environments

\[ \text{FV}(e) = \emptyset \quad \text{refs}(e) \subseteq R \]
\[ \langle l, P! \rangle, \langle l', P'! \rangle \in R \Rightarrow \vdash P! \leftrightarrow P'! \]
\[ H(\ell, f) = v \Rightarrow \exists P : \langle \ell, P! \rangle \in R \land \vdash \text{ftype}(0, P, f) \]
\[ \vdash e, H, R \]

Figure 20: Runtime configuration well-formedness

### A Proof of soundness

#### A.1 Run-time configurations

In the soundness proof, expressions are typed using a typing environment \([H, R]\) constructed from the heap \(H\) and the reference set \(R\), which contains aliasing information for fields. Figure 19 shows the definition of \([H, R]\).

A run-time configuration is well-formed, represented as \(\vdash e, H, R\), shown in Figure 20, if \(e\) has no free variable, 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.

#### A.2 Extensions of typing environments

A typing environment \(\Gamma'\) is an extension of \(\Gamma\), if \(\Gamma \subseteq \Gamma'\). Extensions preserve almost all the typing judgments, as described in Lemma A.1.

**Lemma A.1** If \(\Gamma_2\) is an extension of \(\Gamma_1\), then the following statements are true:

- If \(\Gamma_1 \vdash T\), then \(\Gamma_2 \vdash T\).
- If \(\Gamma_1 \vdash T \preceq P\), then \(\Gamma_2 \vdash T \preceq P\).
- If \(\text{ftype}(\Gamma_1, T, f) = T_f\), then \(\text{ftype}(\Gamma_2, T, f) = T_f\).
- If \(\text{mtype}(\Gamma_1, T, m) = (\mathbf{x}\mathbf{T}) \rightarrow T_{n+1}\), then \(\text{mtype}(\Gamma_2, T, m) = (\mathbf{x}\mathbf{T}) \rightarrow T_{n+1}\).
- If \(\Gamma_1 \vdash p = p'\), then \(\Gamma_2 \vdash p = p'\).
- If \(T \llbracket \Gamma_1; T_x / x \rrbracket = T'\), then \(T \llbracket \Gamma_2; T_x / x \rrbracket = T'\).
- If \(\Gamma_1 \vdash T_1 \preceq T_2\), then \(\Gamma_2 \vdash T_1 \preceq T_2\).
- If \(\Gamma_1 \vdash v : T\), then \(\Gamma_2 \vdash v : T\).

**Proof:** The proof is by induction on the derivation of these judgments. □

#### A.3 Substitutions

This section describes several preliminary lemmas about both type and value substitutions.

The typing of field accesses and that of method calls both depend on the typing of their receivers, although the static type of the receiver might not be as precise as the dynamic type of the receiver object. Therefore, we need Lemma A.2, which states that type substitution is covariant.

**Lemma A.2** If \(\Gamma \vdash T_1 \preceq T_2\), then for any type \(T\), we have \(\Gamma \vdash T \llbracket \Gamma; T_1 / x \rrbracket \preceq T \llbracket \Gamma; T_2 / x \rrbracket\).
PROOF: The proof is by structural induction on $T$. Note that there is no arrow type in J&h. □

As a corollary of Lemma A.2, we can prove Lemma A.3, since the ftype function basically just involves a type substitution.

**Lemma A.3** If $\Gamma \vdash T_1 \leq T_2$, and $\text{ftype}(\Gamma, T_1, f) = T_f$, and $\text{ftype}(\Gamma, T_2, f) = T'_f$, then $\Gamma \vdash T_f \leq T'_f$.

J&h supports dependent types, and therefore value substitution may also apply to types. The following lemma states that value substitution preserves subtyping.

**Lemma A.4** If $\Gamma, x:T_1 \vdash T_2$, and $\Gamma \vdash v:T_2$, then $\Gamma\{v/x\} \vdash T_1\{v/x\}$.

**Proof:** The proof is by induction on the derivation of $\Gamma, x:T_1 \vdash T_2$. □

Lemma A.5 states that sharing is preserved by value substitutions.

**Lemma A.5** If $\Gamma, x:T_1 \rightarrow T_2$, and $\Gamma \vdash v:T_2$, then $\Gamma\{v/x\} \vdash T_1\{v/x\} \rightarrow T_2\{v/x\}$.

**Proof:** The proof is by induction on the derivation of $\Gamma, x:T_1 \rightarrow T_2$. □

Now we can proof the value substitution lemma:

**Lemma A.6** (Value substitution) If $\Gamma, x:T_1 \vdash e:T$, and $\Gamma \vdash v:T_1$, then $\Gamma\{v/x\} \vdash e\{v/x\}:T\{v/x\}$.

**Proof:** The proof is by induction on the derivation of $\Gamma, x:T_1 \vdash e:T$, using Lemma A.4 and Lemma A.5. □

The following lemma helps make the value substitution lemma applicable to the case of proving preservation for method calls, since T-CALL uses exactness-preserving substitution.

**Lemma A.7** If $\Gamma, x:T_1 \vdash T$, and $\Gamma \vdash v:T_1$, then $\Gamma\{v/x\} \vdash T\{v/x\} \leq T\{\text{class}/x\}$.

**Proof:** Note that $T\{v/x\}$ is actually $T\{\text{class}/x\}$, and $\Gamma \vdash v:T_1$ implies that $\Gamma \vdash v:\text{class} \leq T_1$. Then we can apply Lemma A.2 to get the proof. □

### A.4 Sharing of field types

Unlike the J&h calculus, the homogeneous sharing J&h core language ensure that field types are always shared for all the shared views of the containing object. Therefore, there is no need to duplicate fields or to use masked types. This property is proved in Lemma A.8 (Lemma 4.5 in the text).

**Lemma A.8** If $\vdash P! \leftrightarrow P'!$, and $\text{ftype}(\emptyset, P!, f) = T_f$, and $\text{ftype}(\emptyset, P'!, f) = T'_f$, then $\vdash T_f \leftrightarrow T'_f$.

**Proof:** It is easy to see that both $\text{ftype}(\emptyset, P!, f)$ and $\text{ftype}(\emptyset, P'!, f)$ are obtained from the same declared field type $T_f^{\text{decl}}$, where $\text{ftype}_{\text{decl}}(\emptyset, P!, f) = [\text{final}] T_f^{\text{decl}}$.

If $T_f^{\text{decl}}$ is not dependent, then the proof is trivial. If $T_f^{\text{decl}}$ is a dependent type without any prefix types in it, that is, this.class.$\overline{C}$, the proof is also not hard, by SH-NEST. The interesting case is when $T_f^{\text{decl}}$ is a prefix type, which can be proved using Lemma A.9. □

**Lemma A.9** 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 $\vdash P_1!:\overline{C}_1 \leftrightarrow P_2!:\overline{C}_2$. There are several cases:

- **SH-REFL.**
  Trivial.
• **SH-SYM**
  Proved by directly using the induction hypothesis.

• **SH-TRANS**
  Proved by directly using the induction hypothesis.

• **SH-EXACT**
  Then both prefix types are evaluated using S-PRE-E2, \( P_1! \cdot \overline{C}_1 = P_1! \) and \( P_2! \cdot \overline{C}_2 = P_2! \). By SH-EXACT, we have \( \vdash T_1 \to T_2 \), and \( T_1! = P_1! \) and \( T_2! = P_2! \), i.e., \( T_1 \) and \( T_2 \) are just \( P_1 \) and \( P_2 \) with exactness inserted somewhere. Then by the induction hypothesis, \( \vdash P[T_1!] \to P[T_2!] \). By S-PRE-E2, we know that \( P[P_1!] \) evaluates to \( P[T_1!] \), and \( P[P_2!] \) evaluates to \( P[T_2!] \). Therefore, by SH-EXACT, the proof follows.

• **SH-NEST**
  Then both \( P[P_1! \cdot \overline{C}_1] \) and \( P[P_2! \cdot \overline{C}_2] \) can be evaluated according to S-PRE-E1, since they are not exact. By SH-NEST, \( P_1! \cdot \overline{C}_1 = T_1! \cdot C, P_2! \cdot \overline{C}_2 = T_2! \cdot C, \) and \( \vdash T_1 \to T_2 \). By S-PRE-E1, \( P[P_1! \cdot \overline{C}_1] \) evaluates to \( T_1! \), and \( P[P_2! \cdot \overline{C}_2] \) evaluates to \( T_2! \). Then the proof follows.

• **SH-DECL**
  Then there exists \( P' \) and \( C \), such that \( P_1! \cdot \overline{C}_1 = P_1! = P'! \cdot C! \), and \( P_2! \cdot \overline{C}_2 = P_2! \), and \( CT'(P'.C) = \ldots \) shares \( T' \ldots \), and \( T'(\emptyset; P'/\text{this}) \) = \( P_2! \). By SH-OK in Figure 13, \( T'(\emptyset; P'/\text{this}) \) = \( P'.C' \). Now we can see that \( P_2 = P'.C' \). Therefore \( P[P_1!] = P'! = P[P_2!] \). Then the proof follows by SH-REFL.

\( \square \)

### A.5 Progress

We first prove the progress lemma (Lemma A.10), which is easier than proving subject reduction.

**Lemma A.10** (Progress) If \( \vdash e, H, R, \) and \( |H, R| \vdash e; T \), then either \( \exists v, \) such that \( e = v, \) or \( \exists e', H', R', \) such that \( e, H, R \rightarrow e', H', R' \).

**PROOF:** The proof is by structural induction on \( e \). There are the following cases:

• \( e = v \)
  Trivial.

• \( e = x \)
  Vacuously true, since \( e \) has no free variable.

• \( e = e_0.f \)
  There are two cases:
  - \( e_0 \neq v \)
    Then by the induction hypothesis, \( \exists e'_0, H', R' \cdot e_0, H, R \rightarrow e'_0, H', R' \). Then R-CONG applies.
  - \( e_0 = (\ell, P!) \)
    Then by R-GET, \( e' = \text{view}(H(\ell, f), \text{ftype}(\emptyset, P!, f)) \), and \( H' = H, \) and \( R' = R, \text{view}(H(\ell, f), \text{ftype}(\emptyset, P!, f)) \).

• \( e = (e_0.f = e_1) \)
  There are again two cases:
  - \( e_0 \neq v \) or \( e_1 \neq v \)
    Then the induction hypothesis and R-CONG applies.
  - \( e_0 = v_0 \) and \( e_1 = v_1 \)
    The proof is similar to that for the second case with \( e = e_0.f \), using R-SET instead of R-GET.

29
• $e = e_0, m(\overline{r})$
  There are two cases:
  
  - $e_i \neq v$ for some $i$ ($0 \leq i \leq n$)
    The proof follows from the induction hypothesis and R-CONG.
  
  - $e_i = v_i$ ($0 \leq i \leq n$)
    Suppose $v_0 = (\ell, P')$. By T-CALL, we know that $\text{mtype}(P, m)$ is well-defined. Then $\text{mbody}(P, m)$ is a singleton set according to the definition of $\text{mtype}(P, m)$. Therefore R-CALL applies.

• $e = e_1; e_2$
  There are two cases:
  
  - $e_1 \neq v$
    The proof follows from the induction hypothesis and R-CONG.
  
  - $e_1 = v$
    Then R-SEQ applies.

• $e = \text{new } T(\overline{f} = \overline{r})$
  There are three cases:
  
  - $T \neq P!C$
    Then we should continue evaluating the type $T$, as described in Section 4.7.
  
  - $T = P!C$ and $e_i \neq v$ for some $i$ ($1 \leq i \leq n$)
    The proof follows from the induction hypothesis and R-CONG.
  
  - $T = P!C$ and $e_i = v_i$ ($1 \leq i \leq n$)
    Then R-ALLOW applies.

• $e = (\text{view } T)e_0$
  There are again three cases:
  
  - $T \neq P!C$
    Then we need to continue evaluating the type $T$.
  
  - $T = P!C$ and $e_0 \neq v$
    The proof follows from the induction hypothesis and R-CONG.
  
  - $T = P!C$ and $e_0 = v$
    Then R-VIEW applies.

\[ \square \]

### A.6 Subject reduction

Before proving that typing is preserved by evaluating an expression, we first prove that evaluation of a final access path preserves the typing.

**Lemma A.11** If $\vdash e, H, R$, and $\vdash_{\text{final }} p : T$, and $p, H, R \rightarrow p', H', R'$, then $\vdash p', H', R'$, and $\vdash_{\text{final }} p' : T'$, and $\vdash H', R' : T' \leq T$.

**Proof:** The proof is by induction on the derivation of $\vdash_{\text{final }} p : T$. Since the final access path $p$ can take a step, it must be of the form $p_0.f$. There are then two cases:
Then \( R\text{-CONG} \) applies, and \( p_0, H, R \rightarrow p'_0, H', R' \). By \( F\text{-GET} \), \( [H, R] \vdash_{\text{final}} p_0:T_0 \), and \( T = \text{ftype}([H, R], T_0, f) \). Then by the induction hypothesis, \( [H', R'] \vdash_{\text{final}} p'_0:T'_0 \), and \( [H', R'] \vdash T'_0 \leq T_0 \). Then by \( F\text{-GET} \), \( [H', R'] \vdash_{\text{final}} p'_0.f:\text{ftype}([H', R'], T_0, f) \), that is, \( T' = \text{ftype}([H', R'], T'_0, f) \). Then by Lemma A.3, \( [H', R'] \vdash T' \leq T \).

\[ p_0 = \{\ell, P!\} \]

Then \( R\text{-GET} \) applies. Let \( H(\ell, f) = v \), and \( \text{ftype}(0, P!, f) = P'! \mathcal{C} \), and \( \langle \ell', P''! \rangle = \text{view}(v, P'! \mathcal{C}) \). Then \( p' = \langle \ell', P''! \rangle \). By \( F\text{-REF} \), \( [H, R] \vdash_{\text{final}} p_0:P! \). By \( F\text{-GET} \), \( T = \text{ftype}([H, R], P!, f) \). By \( S\text{-FIN} \), \( [H, R] \vdash p\text{.class} \leq T \), and by the definition of run-time typing environment, \( [H, R] \vdash p = p' \). Thus by \( S\text{-ALIAS} \), \( [H, R] \vdash p'\text{.class} \leq T \).

Since \( H' = H \) and \( R' = R, p', [H', R'] \) is an extension of \([H, R]\). By Lemma A.1, \( [H', R'] \vdash p'\text{.class} \leq T \). By \( F\text{-REF} \), \( [H', R'] \vdash_{\text{final}} p':P''! \), that is, \( T' = P''! \). By \( S\text{-FIN-EXACT} \), \( [H', R'] \vdash p'\text{.class} = P''! \). Therefore \( [H', R'] \vdash P''! \leq T \) by \( S\text{-TRANS} \), that is, \( [H', R'] \vdash T' \leq T \).

\[ \square \]

**Lemma A.12 (Subject reduction)** If \( \vdash e, H, R \) and \( [H, R] \vdash e:T \), and \( e, H, R \rightarrow e', H', R' \), then \( \vdash e', H', R' \) and \( [H', R'] \vdash e':T \).

**Proof:** It is not hard to prove that the resulting run-time configuration \( e', H', R' \) is still well-formed, by inspecting the definition of configuration well-formedness, and how the configuration can change for each of the operational semantic rules.

Now let us prove \( [H', R'] \vdash e':T \), by induction on the derivation of \( [H, R] \vdash e:T \).

First, if the derivation ends with \( T\text{-SUB} \), we can use the induction hypothesis and the extension lemma to do the proof. Therefore, let us now only consider the case without \( T\text{-SUB} \) as the last rule in the derivation. We also consider different cases for \( e \). (Note that most of the congruence cases are very easy to prove, just using the induction hypothesis and the appropriate typing rules).

\[ e = \nu \]

Vacuously true, because there is no step to take.

\[ e = x \]

Vacuously true, because \( e \) contains no free variables.

\[ e = e_0.f \]

There are several cases for \( e_0 \):

\[ e_0 = \{\ell, P!\} \]

There are again two cases for the rule used for typing \( e \), depending on whether \( f \) is a final field or not.

\[ \star \text{T-FIN} \]

Then by \( R\text{-GET} \), \( e, H, R \rightarrow v, H, R' \), where \( H(\ell, f) = v' \), and \( v = \text{view}(v', \text{ftype}(0, P!, f)) \), and \( R' = R, v \).

We can still use \( T\text{-FIN} \) to type \( v \). According to Lemma A.11, the proof follows.

\[ \star \text{T-GET} \]

Then operationally, it is still \( R\text{-GET} \), the same as in the previous case. We still have \( e, H, R \rightarrow v, H, R' \), where \( H(\ell, f) = v' \), and \( v = \text{view}(v', \text{ftype}(0, P!, f)) \), and \( R' = R, v \).

Although the typing of \( \{\ell, P!\} \) might not be as precise as \( P! \), by Lemma A.3, we only need to consider \( P! \) as the receiver type.

Let \( \text{ftype}(0, P!, f) = T_f \). We need to prove \( [H, R'] \vdash v:T_f \).

By the definition of \( \vdash e, H, R \), we have \( H(\ell, f) = v' \rightarrow \exists \langle \ell, P''! \rangle \in R \land v : \text{ftype}(0, P''!, f) \). Since \( \{\ell, P''!\}, \{\ell, P''!\} \in R \), we have \( P''! \leftarrow P''! \). Then by Lemma A.8, \( \vdash \text{ftype}(0, P!, f) \leftarrow \text{ftype}(0, P''!, f) \).

Then by the definition of the auxiliary function \( \text{view} \), \( \vdash \text{view}(v', T_f) : T_f \), and therefore \( [H, R'] \vdash v:T_f \).

\[ e_0 \neq \nu \]

Then \( R\text{-CONG} \) applies. There are two cases for deriving \( [H, R] \vdash e_0.f:T \).
There are two cases:

- $e = (e_0.f = e_1)$

  There are several cases for $e_0$.

  - $e_0 = (\ell, P!)$
    Again, there are two cases for $e_1$:

      - $e_1 = v$
        Then R-SET applies. By T-SET, $[H, R] \vdash v:T$, where $T = \text{ftype}([H, R], P!, f)$. Here, we still only
        consider $P!$ with the same reason mentioned before for the case of $e = (\ell, P!).f$. Since $[H', R]$ is an
        extension of $[H, R]$, we still have $[H', R] \vdash v:T$.

      - $e_1 \neq v$
        This is the congruence case.

  - $e_0 \neq v$
    This is the congruence case.

- $e = e_0.m(\overline{e})$

  There are two cases:

  - $e_i \neq v$ for any $i$ ($0 \leq i \leq n$)
    This is the congruence case.

  - $e_0 = (\ell, P!)$, and $\overline{e} = \overline{v}$
    Then R-CALL applies, and $H' = H$, $R' = R$, and $e' = e_0.m(\ell, P!)/\text{this}, \overline{v}/\overline{e}$, where
    $\text{mbody}(P, m) = \{T_{n+1} m(T \overline{e}) \{e_m\}\}$. We also have $[H, R] \vdash e_0.T_{n+1}'$, where $T_{i}' = T_{i}[\ell, [H, R]; \ P!]/\overline{\text{this}}\}$ ($i = 1, \ldots, n + 1$).
    According to M-OK, $\text{this} : P', \overline{x} : T \vdash e_0.T_{n+1}'$, where $\vdash P! \leq P'$. Then the proof follows from Lemma A.6
    and Lemma A.7.

- $e = e_1; e_2$

  There are two cases:

  - $e_1 \neq v$
    This is the congruence case.

  - $e_1 = v$
    Then R-SEQ applies. We also have $e' = e_2, H' = H$, and $R' = R$. The proof follows easily.

- $e = \text{new} P!.\overline{C}(\overline{f} = \overline{v})$

  There are two cases:

  - $e_i \neq v$ for any $i$ ($1 \leq i \leq n$)
    This is the congruence case.

  - $\overline{e} = \overline{v}$
    Then R-ALLOC applies, and $e' = (\ell, P.\overline{C})$. By T-NEW, $T = P.\overline{C}!$. Then by T-FIN, $[H', R'] \vdash (\ell, P.\overline{C})!; P.C!$.

- $e = (\text{view} P!.\overline{C})e_0$

  By T-VIEW, $T = P!.\overline{C}$. There are two cases for $e_0$:

  - $e_0 \neq v$
    This is the congruence case.
\[ e_0 = (\ell, P_0!) \]

Then R-VIEW applies. We have \( e' = \text{view}(\langle \ell, P_0! \rangle, P_1! \overline{C}) \). By T-VIEW, there exists \( T \) such that \( [H, R] \vdash (\ell, P_0!); T \) and \( [H, R] \vdash T \leftrightarrow P_1! \overline{C} \). Now it is easy to see that \( T \) has the form of \( P_1! \overline{C} \), otherwise it cannot be shared with \( P_1! \overline{C} \). Therefore \( [H, R] \vdash P_0! \leq P_1! \overline{C} \), and \( \vdash P_1! \leftrightarrow P_1! \). Since \( [H', R'] \) is an extension of \( [H, R] \), \( [H', R'] \vdash P_0! \leq P_1! \overline{C} \). By the definition of the view function, \( e' = (\ell, P_1! \overline{C}) \), where \( P_0 = P_1! \overline{C} \). By S-SHARE, \( [H', R'] \vdash P_1! \overline{C} \leq [H', R'] \). Therefore \( [H', R'] \vdash P_1! \overline{C} \leq T \), which implies that \( [H', R'] \vdash e':T \).

\[ \square \]

### A.7 Soundness

Finally, the soundness theorem (Theorem A.13, also Theorem 4.1) can be proved.

**Theorem A.13 (Soundness)** If \( \vdash (L, e) \) ok, and \( \vdash e:T \), and \( e, 0, 0 \rightarrow^* e', H, R \), then either \( \exists v \), such that \( e' = v \) and \( [H, R] \vdash v:T \), or \( \exists e'', H', R' \), such that \( e', H, R \rightarrow^* e'', H', R' \).

**Proof:** Follows from Lemma A.10 and Lemma A.12. \( \square \)