Relaxing Ownership with Immutability

Hannes Mehnert   Nicholas Cameron   Alex Potanin
Victoria University of Wellington, Wellington, New Zealand
hannes|ncameron|alex@ecs.vuw.ac.nz

Abstract

Multiple ownership [4] introduced a novel approach to managing object ownership information called "owners-as-boxes". Each object is placed in a "box" or *context* and the nesting relationship between "boxes" or *contexts* respects the object accesses that happen during the program's execution. No restrictions are placed on which objects are allowed to access which other objects, rather the ownership contexts provide a hierarchical, as opposed to a more traditional, flat, view of the program's memory. As with any ownership system, the objects never point "into" another context, but can point outwards from the context they are in. We show how introducing object immutability, readonly references, and class immutability allows to safely change ownership contexts the objects belong to without breaking the ownership restrictions imposed by multiple ownership. We describe a type system for a language with ownership contexts and immutability, prove it sound, and state and prove both ownership and immutability guarantees. Thus, we show how immutability greatly benefits ownership-enabled languages.

1. Introduction

NB! The proofs in this technical report are unfinished and represent a state of play for this work at the end of the research assistanship. We hope to finish the proofs and submit this as a proper paper in the future.

In current ownership systems the owner of an object is not variant with respect to subtyping, since this would impede soundness of the type system. In this paper we introduce variant subtyping with ownership, at least for immutable objects. The type system is specified and soundness is proven.

The paper is structured as follows, in Section 2 different ownership and kinds of immutability are presented. In Section 3 we present some motivating examples which should intuitively work. In Section 4 we formalize our programming language, followed by parts of the proofs in Section 5. In Section 6 we discuss our findings and discuss related work, finally in Section 7 we conclude.

2. Background

This section presents a brief overview of the various kinds of ownership and immutability that have been subject to active research in recent years.

2.1 Variants of Ownership

With ownership certain propositions about the heap layout can be made at compile-time. There are roughly four different kinds of ownership useful for different purposes:

- **owners-as-dominators** Every reference to an object is done via its owner or any object its owner transitively owns. This is a more traditional and the most conservative kind of ownership, also known as *deep ownership* [5, 6].
- **owners-as-modifiers** Every object may have a readonly reference to any other object. Mutation of objects can only be done via the owner. The ownership relationship is a dominator tree. Universes traditionally support this kind of ownership [7, 9, 14].
- **owners-as-permissions** Every object can have a reference to and mutate any other object with compatible ownership. No dominator tree like hierarchy is present in such approach [1, 2]. This approach is sometimes known as *shallow ownership* [6].
- **owners-as-boxes** The ownership topology is preserved, but the topology is not restricted in any way. The type soundness property does not constitute an encapsulation property. However, the ownership relation can be used to construct a precise picture of the run time heap [4]. This approach is sometimes described is *descriptive* ownership as opposed to *prescriptive* ownership approaches in the items listed above.

In this paper, we concentrate on *owners-as-boxes*, or *prescriptive* approach, by bulding on the Multiple Ownership work [4] by one of the authors. Since this kind ownership has been explored the least so far, it poses the most interesting exploration target for this paper.

2.2 Kinds of immutability

There are three kinds of immutability generally distinguished in the literature:

- **Class immutability** No instance of an *immutable class* may be changed. An example is `String` in Java. This is the most familiar kind of immutability to Java programmers.
- **Readonly reference** A *readonly reference* is an immutable pointer to an object. The same object might be mutable if accessed via a different reference (an *alias*). This is useful to allow a specific interface only read only access to an object. This is the most familiar kind of immutability to C++ programmers where references can be marked with `const` to guarantee their immutability. A significant area of research on immutability concentrated on readonly references [3, 14, 17]
- **Object immutability** An *immutable object* cannot be changed, even if other instances of the same class can be. For example the keys of a `Map` in Java should be immutable, since behaviour of mutated keys is not well-defined. Object immutability can be used for optimizations and safe sharing between multiple

threads. In contrast to the `final` keyword in Java, neither the field itself nor the content of an immutable field can be modified, an example is a `char[]` in the `String` class. More recent research concentrated on object immutability as a distinct kind of immutability [10, 12, 18, 19].

In this paper, we support all three kinds of immutability above to see their effect on the ownership. In Section 6 we discuss the related work combining ownership and immutability, though this is the first paper to address owners-as-boxes when it comes to adding immutability of any kind.

3. Combination Language

The combined language which is presented in the following is based on Featherweight Java with assignment. We introduce ownership parameter annotations and immutability parameters to types. In the following paragraphs we motivate the definition of our combined language by showing sound and unsound code examples.

Unsoundness of variant ownership Type systems which integrate only ownership are non-variant about the ownership parameter. If we pretend for a moment that an ownership type system has a variant-owner subtyping rule regarding the $\preceq$ relation:

$$\frac{\Delta;\Gamma \vdash \overline{\texttt{a} \preceq \texttt{a}'}}{\Delta;\Gamma \vdash \texttt{C<}\overline{\texttt{a}}\texttt{>} <: \texttt{C<}\overline{\texttt{a}'}\texttt{>}}$$
(S-UC)

In Listing 1 an example is shown which uses S-UC.

First two classes, `Foo` and `C`, are defined in lines 1-5, where `C` has a field `f` of type `Foo` with the same owner. In line 7 an instance `b` of `Foo` is created, with the owner ◯. In line 8 another instance of `Foo` is created, using `b` as owner. Thus, the relation $\texttt{a} \preceq \texttt{b}$ holds. In line 9 `C` is created, using `a` as an owner. In line 10 an alias `cb` to `ca` is created, which has type `C<b>`. Since `a` is inside `b`, this "upcast" should be safe. In line 11 the field `f` of `cb` is assigned a new value, `new Foo<b>`. Since `cb` has owner `b` and the field `f` of class `C` has the same owner as the class, this assignment is safe. But, in line 12 it turns out that the field `f` in object `ca` fails to have type `Foo<a>`, since $\texttt{b} \not\preceq \texttt{a}$, and thus $\texttt{Foo<a>} \not<: \texttt{Foo<b>}$.

So, adding S-UC leads to unsound code. The shown example is similar to covariance of generics. Needless to say the field assignment in line 11 is the cause of unsoundness rather than the field access in line 12. If we forbid the field assignment, this code would be sound; and would enable ownership variance.

Code listing 1 $\preceq$ and unsound subtyping

```
1: Class Foo<y> { }
2:
3: Class C<x> {
4:  field Foo<x> f = new Foo<x>()
5: }
6:
7: Foo<◯> b = new Foo<◯>
8: Foo<b> a = new Foo<b>
9: C<a> ca = new C<a>()
10: C<b> cb = ca
11: cb.f = new Foo<b>()
12: Foo<a> caf = ca.f
```

Immutability Our mechanism to not allow the field assignment is to introduce immutability into the type system. Then we introduce variance of owners restricted for immutable objects:

$$\frac{\Delta;\Gamma \vdash \overline{\texttt{a} \preceq \texttt{a}'}}{\Delta;\Gamma \vdash \texttt{C}_{immutable}\texttt{<}\overline{\texttt{a}}\texttt{>} <: \texttt{C}_{immutable}\texttt{<}\overline{\texttt{a}'}\texttt{>}}$$
(S-COVARIANT)

The subscripts $immutable$ and $mutable$ are used for all different kinds of immutability:

- If a *class definition* has a subscript $immutable$, like `class` $\texttt{C}_{immutable}$ `{...}`, it is *class immutable*. All instances of a immutable class are immutable. For a given mutable class $\texttt{D}_{mutable}$, there exists an immutable class $\texttt{D}_{immutable}$, which is a superclass of the mutable one.
- If an object of an immutable class is instantiated, this object is *object immutable* $\texttt{D}_{immutable}\texttt{<a> di = new D}_{immutable}\texttt{<a>()}$.
- If the *binding type declaration* contains the subscript $immutable$, it is a *read-only reference*: $\texttt{D}_{immuable}\texttt{<a> di = d}$, where `d` may be mutable.

The code in Listing 2 shows that subtyping with immutable classes is sound. The difference to the previous example is that each class contains a mutability parameter, written as subscript (`n, m`). The instantiated `ca` (in line 9) is an immutable object. Due to the introduced subtyping rule S-COVARIANT widening the owner in line 10 is valid, similar to an upcast. The field assignment in line 11 does not type check, because `cb` is immutable.

Code listing 2 $\preceq$ and immutability and subtyping

```
1: Class Foo_m<y> { }
2:
3: Class C_n<x> {
4:  field Foo_n<x> f = new Foo_n<x>()
5: }
6:
7: Foo_immutable<◯> b = new Foo_immutable<◯>
8: Foo_immutable<b> a = new Foo_immutable<b>
9: C_immutable<a> ca = new C_immutable<a>()
10: C_immutable<b> cb = ca
11: cb.f = new Foo_immutable<b>() /* type error */
```

Another interesting question arises, what is the type of `cb.f` (in Listing 2)? By definition of `class` $\texttt{C}_m$ it should be of type $\texttt{Foo}_{immutable}\texttt{<b>}$, whereas the actual type is $\texttt{Foo}_{immutable}\texttt{<a>}$. Due to the covariant subtype relation,

$\texttt{Foo}_{immutable}\texttt{<a>} <: \texttt{Foo}_{immutable}\texttt{<b>}$

holds in the given example, and `cb.f` can be safely assumed to be of type $\texttt{Foo}_{immutable}\texttt{<b>}$.

Immediate observations We successfully relaxed ownership by introducing immutability.

An emerging question is what the subtyping relationship between a mutable and immutable object of the same class is. We will look at this issues in the following. Another pending question is how to deal with readonly references of mutable objects.

Subclassing of mutable classes If we allow immutable extension of a mutable class, as shown in Listing 3, then `setF` in lines 6-8 will be inherited in class $\texttt{C}_{immutable}$, defined in line 11. This leads to a mutation of the field `f` of the immutable class $\texttt{C}_{immutable}$ in line 17. The result is again an unsound field access in line 18, since $\texttt{a}\ preceq\ \texttt{b}\ \texttt{Foo}_{immutable}\texttt{<b>} \not<: \texttt{Foo}_{immutable}\texttt{<a>}$.

Therefore we do not allow an immutable class to be a subclass of a mutable class. It is also counterintuitive, since a mutable class has more methods (e.g. field setters) than an immutable.

Subclassing of immutable classes We allow mutable extensions of immutable classes, an example is given in Listing 4. The immutable class $\texttt{D}_{immutable}$ in lines 3-5 is extended by the mutable class $\texttt{C}_m$ in lines 7-9. The mutable instance, created in line 12, can be assigned a new object for the field `g` (as done in line 13), because this field is mutable. The immutable field `f` still cannot be mutated, shown in line 14, which results in a type error.

Code listing 3 Immutable extension of mutable class

```
class Foo_m<y> { }

class D_n<x> {
 field Foo_n<x> f = new Foo_n<x>()

 void setF_mutable (Foo_n<x> newf) {
    f = newf
 }
}

class C_immutable<x> extends D_n<x> { }

Foo_immutable<◯> b = new Foo_immutable<◯>
Foo_immutable<b> a = new Foo_immutable<b>
C_immutable<a> c = new C_immutable<a>()
C_immutable<b> cb = ca
cb.setF(new Foo_immutable<b>())
Foo_immutable<a> cf = c.f /* type error */
```

Code listing 4 Mutable extension of immutable class

```
class Foo_m<y> { }

class D_immutable<x> {
 field Foo_immutable<x> f = new Foo_immutable<x>()
}

class C_m<y> extends D_immutable {
 field Foo_m<y> g
}

Foo_immutable<◯> b = new Foo_immutable<◯>
C_mutable<a> c = new C_mutable<a>()
c.g = new Foo_mutable<a>()
c.f = new Foo_immutable<a>() /* type error */
```

We allow mutable extensions of immutable classes, furthermore a mutable class C is a subtype of the immutable class C:

$$\frac{\texttt{class } C_m\langle\overline{o \rightarrow [b_l\ b_u]}\rangle \texttt{ extends N}}{\Delta;\Gamma \vdash C_{mutable}\langle\bar{a}\rangle <: C_{immutable}\langle\bar{a}\rangle}$$

(S-MUTABLE)

Readonly references, Viewpoint adaptation A readonly reference can reference either a mutable or an immutable object. In Listing 5 a class C is defined (lines 3-5) with a single field f (line 4). In line 9 a mutable instance c is made, which is aliased by a readonly reference ci in line 10. This can be relaxed to be of type $C_{immutable}$<b> by the introduced S-COVARIANT subtyping rule (as seen with cbi in line 11). Now, since it is actually a mutable object initially, nothing prevents us from assigning the field f to a new instance of Foo in line 12. The result is a type error in line 13, when the field f of object c is accessed, since the type is $Foo_{mutable}$<b>, not $Foo_{mutable}$<a>.

In order to fix this unsoundness issue we introduce *viewpoint adaptation* [8]. The viewpoint adaptation relation (written $\triangleright$ in T-FIELD, T-ASSIGN and T-INVK) is defined for the formal and actual type. It results in an $immutable$ type if either the formal or the actual type is immutable, and a $mutable$ type if both input types are mutable. With this setup, the assignment in line 12 of Listing 5 is invalid, since the field f of reference cbi is of type $Foo_{immutable}$<b> and thus cannot be assigned to.

Code listing 5 Motivation for viewpoint adaptation

```
1: class Foo_m<y> { }
2:
3: class C_n<x> {
4:  field Foo_n<x> f = new Foo_n<x>()
5: }
6:
7: Foo_immutable<◯> b = new Foo_immutable<◯>
8: Foo_immutable<b> a = new Foo_immutable<b>
9: C_mutable<a> c = new C_mutable<a>()
10: C_immutable<a> ci = c /* readonly reference */
11: C_immutable<b> cbi = ci
12: cbi.f = new Foo_mutable<b>()
13: Foo_mutable<a> c.f /* type error */
```

4. Formalism

The developed language is based on Featherweight Java [11], extends this with assignment, ownership and immutability annotations.

The syntax is given in Figure 1, the runtime parts are distinguished by a grey background.

The possible expressions are `null`, variables, field access, field assignment, method calls, allocation, access to adresses and error.

All types are classes. A class consists of its name, the mutability parameter, ownership parameters, its superclass, a list of fields and method declarations.

A method declaration consists of owner parameters, return type, mutability parameter, parameter values and types and its body, with `return` as the last statement.

A value is either an address or null.

A runtime type consists of a class and runtime contexts, which are either world or an address. World is the top element of the ownership relation.

A type consists of a class and contexts. Each context is either a variable, a formal owner, world or an address.

A context environment is a mapping from a formal owner to lower and upper bounds, either a context or bottom.

A variable environment contains a mapping from variables and adresses to types.

A heap consists of addresses, each pointing to a runtime type followed by values of the fields.

In Figure 2 the subtyping rules are introduced. Reflexivity and transitivity S-REFLEX and S-TRANS are standard rules. The rule S-CLASS defines that a specified subclass is a subtype after substitution of the formal owner parameters. The rule S-MUTABLE was introduced in the last section and defines that a mutable class is a subtype of the immutable. The last rule, S-COVARIANT, is the core contribution. It states that if two owners are in the $\preceq$ relationship, an immutable class of the inner context is a subtype of the outer context.

In Figure 3 the inside relation is defined. It is straightforward, provides reflexivity (I-REFLEX), transitivity (I-TRANS), the top element world (I-WORLD), the bottom element (I-BOTTOM), for any variable or address the owner is inside of its types owner (I-OWNER). Finally, in a context environment the owner of an object is between the lower and upper bound.

e	::=	null | x | γ.f | γ.f = e | γ.<$\overline{\texttt{a}}$>m($\overline{\texttt{e}}$) | new T | ι | err	*expressions*
Q	::=	class C_m<$\overline{\texttt{o}\rightarrow[\texttt{b}_l\ \texttt{b}_u]}$> extends N {$\overline{\texttt{T f}}$; $\overline{\texttt{W}}$}	*class declarations*
W	::=	<$\overline{\texttt{o}\rightarrow[\texttt{b}_l\ \texttt{b}_u]}$> T m_m ($\overline{\texttt{T x}}$) {return e;}	*method declarations*
v	::=	ι | null	*values*

R	::=	C_m<$\overline{\texttt{r}}$>	*runtime types*
T,N	::=	C_m<$\overline{\texttt{a}}$>	*types*
Δ	::=	$\overline{\texttt{o}\rightarrow[\texttt{b}_l\ \texttt{b}_u]}$	*context environments*
γ	::=	x | ι | null	*vars and addresses*
Γ	::=	$\overline{\gamma\texttt{:T}}$	*var environments*
$\mathcal{H}$	::=	$\overline{\iota\rightarrow\{\texttt{R};\ \overline{\texttt{f}\rightarrow\texttt{v}}\}}$	*heaps*
m, n	::=	immutable | mutable	*mutability*
a	::=	o | x | $\bigcirc$ | ι	*contexts*
r	::=	$\bigcirc$ | ι	*runtime contexts*
b	::=	a | $\bot$	*bounds*
x, y			*variables*
o			*formal owners*
C			*classes*
ι			*addresses*

Figure 1. Syntax of FJ I O.

$$\frac{}{\Delta;\Gamma\vdash \texttt{T} <: \texttt{T}}\ \text{(S-REFLEX)} \qquad \frac{\Delta;\Gamma\vdash \texttt{T} <: \texttt{T}' \quad \Delta;\Gamma\vdash \texttt{T}' <: \texttt{T}''}{\Delta;\Gamma\vdash \texttt{T} <: \texttt{T}''}\ \text{(S-TRANS)}$$

$$\frac{\texttt{class } C_m\texttt{<}\overline{\texttt{o}\rightarrow[\texttt{b}_l\ \texttt{b}_u]}\texttt{> extends N}}{\Delta;\Gamma\vdash C_n\texttt{<}\overline{\texttt{a}}\texttt{>} <: [\overline{\texttt{a/o}}]\texttt{N}}\ \text{(S-CLASS)}$$

$$\frac{\texttt{class } C_m\texttt{<}\overline{\texttt{o}\rightarrow[\texttt{b}_l\ \texttt{b}_u]}\texttt{> extends N}}{\Delta;\Gamma\vdash C_{mutable}\texttt{<}\overline{\texttt{a}}\texttt{>} <: C_{immutable}\texttt{<}\overline{\texttt{a}}\texttt{>}}\ \text{(S-MUTABLE)} \qquad \frac{\Delta;\Gamma\vdash \overline{\texttt{a} \preceq \texttt{a}'}}{\Delta;\Gamma\vdash C_{immutable}\texttt{<}\overline{\texttt{a}}\texttt{>} <: C_{immutable}\texttt{<}\overline{\texttt{a}'}\texttt{>}}\ \text{(S-COVARIANT)}$$

Figure 2. FJ I O subtyping

$$\frac{}{\Delta;\Gamma\vdash \texttt{b}\preceq\texttt{b}}\ \text{(I-REFLEX)} \qquad \frac{\Delta;\Gamma\vdash \texttt{b}\preceq\texttt{b}'' \quad \Delta;\Gamma\vdash \texttt{b}''\preceq\texttt{b}'}{\Delta;\Gamma\vdash \texttt{b}\preceq\texttt{b}'}\ \text{(I-TRANS)} \qquad \frac{\Delta;\Gamma\vdash \texttt{b}\ \text{OK}}{\Delta;\Gamma\vdash \texttt{b}\preceq\bigcirc}\ \text{(I-WORLD)}$$

$$\frac{\Delta;\Gamma\vdash \texttt{b}\ \text{OK}}{\Delta;\Gamma\vdash\bot\preceq\texttt{b}}\ \text{(I-BOTTOM)} \qquad \frac{\Gamma(\gamma) = C_m\texttt{<}\overline{\texttt{a}}\texttt{>}}{\Delta;\Gamma\vdash\gamma\preceq\texttt{a}_0}\ \text{(I-OWNER)} \qquad \frac{\Delta(\texttt{o}) = [\texttt{b}_l\ \texttt{b}_u]}{\Delta;\Gamma\vdash\texttt{o}\preceq\texttt{b}_u \quad \Delta;\Gamma\vdash\texttt{b}_l\preceq\texttt{o}}\ \text{(I-BOUND)}$$

Figure 3. FJ I O inside relation for owners and environments.

In Figure 4 the reduction rules are shown. A field access is reduced to the field value (R-FIELD). A field assignment changes the heap $\mathcal{H}$, such that the specific field value is replaced by the new value (R-ASSIGN). In R-NEW a new object is instantiated, resulting in a modified heap $\mathcal{H}$, where the address of the new object is bound, all fields of the object are initialized to `null`. A method invocation (R-INVK) is reduced to the body of the method, substituting the method arguments with the actual parameters.

In Figure 5 the remaining reduction rules are shown, RC-ASSIGN and RC-INVK continue evaluation when the specific expression (new value or parameter) is reducible. Access or assignment of a field of the object `null` or invocation of a method of the object `null` results in the error state (R-FIELD-NULL, R-ASSIGN-NULL, R-INVK-NULL). The rules RC-ASSIGN-ERR and RC-INVK-NULL reduce to the error state if a subexpression reduces to error.

The Figure 6 shows the well-formedness rules for contexts and types. A context environment is well-formed if either the owner is in its domain (F-OWNER) or it the top element (F-WORLD) or the bottom element (F-BOTTOM).

A type environment is well-formed if it contains a mapping for every variable in its domain (F-VAR).

A class is well-formed (F-CLASS) if the actual owners are in the formal bounds and the mutability parameter is less than or equal, which is defined in Figure 10: immutable is less than mutable, and reflexivity of the mutability parameter.

In Figure 7 well-formed environments are defined. Either it is empty F-EMPTY or the upper and lower bound are well-formed and the lower bound is inside the upper bound (F-ENV).

Well-formedness of heaps and configurations is shown in Figure 8. A heap is well-formed (F-HEAP) if for all addresses of the heap $\mathcal{H}$the types are well-formed, and all non-null field values these are in the domain of $\mathcal{H}$. An expression is well-formed (F-CONFIG) if all free variables are bound in the heap $\mathcal{H}$.

$$\frac{\mathcal{H}(\iota) = \{\mathtt{R}; \overline{\mathtt{f}\to\mathtt{v}}\}}{\iota.\mathtt{f}_i; \mathcal{H} \leadsto \mathtt{v}_i; \mathcal{H}}\ \text{(R-FIELD)}
\qquad
\frac{\mathcal{H}(\iota) = \{\mathtt{R}; \overline{\mathtt{f}\to\mathtt{v}}\} \quad \mathcal{H}' = \mathcal{H}[\iota \mapsto \{\mathtt{R}; \overline{\mathtt{f}\to\mathtt{v}}[\mathtt{f}_i \mapsto \mathtt{v}]\}]}{\iota.\mathtt{f}_i\ \mathtt{=}\ \mathtt{v}; \mathcal{H} \leadsto \mathtt{v}; \mathcal{H}'}\ \text{(R-ASSIGN)}$$

$$\frac{\mathcal{H}(\iota)\ undefined \quad fields(\mathtt{C}) = \overline{\mathtt{f}} \quad \mathcal{H}' = \mathcal{H}, \iota \to \{\mathtt{C}_m\mathtt{<}\overline{\mathtt{r}}\mathtt{>}; \overline{\mathtt{f}\to\mathtt{null}}\}}{\mathtt{new}\ \mathtt{C}_m\mathtt{<}\overline{\mathtt{r}}\mathtt{>}; \mathcal{H} \leadsto \iota; \mathcal{H}'}\ \text{(R-NEW)}
\qquad
\frac{\mathcal{H}(\iota) = \{\mathtt{R}; ...\} \quad mBody(\mathtt{m}\mathtt{<}\overline{\mathtt{r}}\mathtt{>}, \mathtt{R}) = (\overline{\mathtt{x}}; \mathtt{e})}{\iota.\mathtt{<}\overline{\mathtt{r}}\mathtt{>m}(\overline{\mathtt{v}}); \mathcal{H} \leadsto [\overline{\mathtt{v/x}}]\mathtt{e}; \mathcal{H}}\ \text{(R-INVK)}$$

Figure 4. FJ I O reduction rules.

$$\frac{\mathtt{e}'; \mathcal{H} \leadsto \mathtt{e}''; \mathcal{H}' \quad \mathtt{e}'' \neq \mathtt{err}}{\iota.\mathtt{f}\ \mathtt{=}\ \mathtt{e}'; \mathcal{H} \leadsto \iota.\mathtt{f}\ \mathtt{=}\ \mathtt{e}''; \mathcal{H}'}\ \text{(RC-ASSIGN)}
\qquad
\frac{\mathtt{e}_i; \mathcal{H} \leadsto \mathtt{e}_i'; \mathcal{H}' \quad \mathtt{e}_i' \neq \mathtt{err}}{\iota.\mathtt{<}\overline{\mathtt{r}}\mathtt{>m}(\overline{\mathtt{v}},\ \mathtt{e}_i,\ \overline{\mathtt{e}}); \mathcal{H} \leadsto \iota.\mathtt{<}\overline{\mathtt{r}}\mathtt{>m}(\overline{\mathtt{v}},\ \mathtt{e}_i',\ \overline{\mathtt{e}}); \mathcal{H}'}\ \text{(RC-INVK)}$$

$$\frac{}{\mathtt{null.f}; \mathcal{H} \leadsto \mathtt{err}; \mathcal{H}}\ \text{(R-FIELD-NULL)}
\qquad
\frac{}{\mathtt{null.f}\ \mathtt{=}\ \mathtt{e}; \mathcal{H} \leadsto \mathtt{err}; \mathcal{H}}\ \text{(R-ASSIGN-NULL)}
\qquad
\frac{}{\mathtt{null.<}\overline{\mathtt{r}}\mathtt{>m}(\overline{\mathtt{e}}); \mathcal{H} \leadsto \mathtt{err}; \mathcal{H}}\ \text{(R-INVK-NULL)}$$

$$\frac{\mathtt{e}'; \mathcal{H} \leadsto \mathtt{err}; \mathcal{H}'}{\iota.\mathtt{f}\ \mathtt{=}\ \mathtt{e}'; \mathcal{H} \leadsto \mathtt{err}; \mathcal{H}'}\ \text{(RC-ASSIGN-ERR)}
\qquad
\frac{\mathtt{e}_i; \mathcal{H} \leadsto \mathtt{err}; \mathcal{H}'}{\iota.\mathtt{<}\overline{\mathtt{r}}\mathtt{>m}(\overline{\mathtt{v}},\ \mathtt{e}_i,\ \overline{\mathtt{e}}); \mathcal{H} \leadsto \mathtt{err}; \mathcal{H}'}\ \text{(RC-INVK-ERR)}$$

Figure 5. FJ I O reduction rules for congruence, `null`, and error propagation.

$$\frac{\mathtt{o} \in dom(\Delta)}{\Delta; \Gamma \vdash \mathtt{o}\ \text{OK}}\ \text{(F-OWNER)}
\qquad
\frac{}{\Delta; \Gamma \vdash \bigcirc\ \text{OK}}\ \text{(F-WORLD)}
\qquad
\frac{}{\Delta; \Gamma \vdash \bot\ \text{OK}}\ \text{(F-BOTTOM)}
\qquad
\frac{\Gamma(\gamma) = \mathtt{T}}{\Delta; \Gamma \vdash \gamma\ \text{OK}}\ \text{(F-VAR)}$$

$$\frac{\mathtt{class}\ \mathtt{C}_m\mathtt{<}\overline{\mathtt{o}\to[\mathtt{b}_l\ \ \mathtt{b}_u]}\mathtt{>}... \quad \Delta; \Gamma \vdash \overline{\mathtt{a}}\ \text{OK} \quad \mathtt{n} \leq \mathtt{m} \quad \overline{\Delta; \Gamma, \mathtt{this}{:}\mathtt{C}_m\mathtt{<}\overline{\mathtt{a}}\mathtt{>} \vdash [\overline{\mathtt{a/o}}]\mathtt{b}_l \preceq \mathtt{a}} \quad \overline{\Delta; \Gamma, \mathtt{this}{:}\mathtt{C}_m\mathtt{<}\overline{\mathtt{a}}\mathtt{>} \vdash \mathtt{a} \preceq [\overline{\mathtt{a/o}}]\mathtt{b}_u}}{\Delta; \Gamma \vdash \mathtt{C}_n\mathtt{<}\overline{\mathtt{a}}\mathtt{>}\ \text{OK}}\ \text{(F-CLASS)}$$

Figure 6. FJ I O well-formed contexts and types.

$$\frac{}{\Delta; \Gamma \vdash \emptyset\ \text{OK}}\ \text{(F-EMPTY)}
\qquad
\frac{\Delta; \Gamma \vdash \mathtt{b}_l, \mathtt{b}_u\ \text{OK} \quad \Delta; \Gamma \vdash \mathtt{b}_l \preceq \mathtt{b}_u \quad \Delta, \mathtt{o}\to[\mathtt{b}_l\ \ \mathtt{b}_u]; \Gamma \vdash \Delta'\ \text{OK}}{\Delta; \Gamma \vdash \mathtt{o}\to[\mathtt{b}_l\ \ \mathtt{b}_u], \Delta'\ \text{OK}}\ \text{(F-ENV)}$$

Figure 7. FJ I O well-formed environments.

$$\frac{\begin{array}{c}\forall \iota \to \{\mathtt{C}_m\mathtt{<}\overline{\mathtt{r}}\mathtt{>}; \overline{\mathtt{f}\to\mathtt{v}}\} \in \mathcal{H} : \\ \Delta; \mathcal{H} \vdash \mathtt{C}_m\mathtt{<}\overline{\mathtt{r}}\mathtt{>}\ \text{OK} \\ \overline{fType(\mathtt{f}, \mathtt{C}\mathtt{<}\overline{\mathtt{r}}\mathtt{>}) = \mathtt{T}'} \quad \Delta; \mathcal{H} \vdash \overline{\mathtt{v}\ :\ \mathtt{T}'} \\ \forall \mathtt{v} \in \overline{\mathtt{v}} : \mathtt{v} \neq \mathtt{null} \Rightarrow \mathtt{v} \in dom(\mathcal{H})\end{array}}{\Delta \vdash \mathcal{H}\ \text{OK}}\ \text{(F-HEAP)}
\qquad
\frac{\Delta \vdash \mathcal{H}\ \text{OK} \quad \forall \iota \in fv(\mathtt{e}) : \iota \in dom(\mathcal{H})}{\Delta; \mathcal{H} \vdash \mathtt{e}\ \text{OK}}\ \text{(F-CONFIG)}$$

Figure 8. FJ I O well-formed heaps and configurations.

In Figure 9 the auxiliary functions are shown. The function $fields$ returns the list of fields for a given class, which is empty for the top class `Object` and the union of local fields and the fields of the superclass for all other classes. The function $fType$ returns for a field and class the type of the field in the class, recursively calling itself with substitution of the owners to the superclass if the field is not defined in the class.

The function $mBody$ returns the body of a given method in a class. It also calls itself recursively if the method is not defined in the given class.

The function $mType$ returns the type and mutability parameter of a given method, again with a recursive call to itself.

The function $override$ succeeds if the given method name, mutability parameter and type signature is valid to overwrite the method of a superclass.

In Figure 10 the less than or equal relation and the mutability function I are defined. The former was already mentioned, and describes the ordering that `immutable` is less than `mutable` and any parameter m is less than or equal to itself. The function I defines what the mutability of a given class is, extracting the mutability parameter.

In Figure 11 the viewpoint adaptation is defined. If the mutability of a type `N` is `immutable`, the viewpoint adaptation results in an immutable type. If the mutability of `N` is `mutable`, the result is the mutability parameter of the actual type.

$$\frac{}{fields(\texttt{Object}) = \emptyset}$$

$$\frac{\texttt{class C}_m\texttt{<}\overline{\texttt{o}\rightarrow\texttt{[b}_l\ \texttt{b}_u\texttt{]}}\texttt{> extends N } \{\overline{\texttt{U f;}}\ \overline{\texttt{W}}\} \qquad fields(\texttt{N}) = \overline{\texttt{g}}}{fields(\texttt{C}) = \overline{\texttt{g}} + \overline{\texttt{f}}}$$

$$\frac{\texttt{class C}_m\texttt{<}\overline{\texttt{o}\rightarrow\texttt{[b}_l\ \texttt{b}_u\texttt{]}}\texttt{> extends N } \{\overline{\texttt{U f;}}\ \overline{\texttt{W}}\}}{fType(\texttt{f}_i, \texttt{C<}\overline{\texttt{a}}\texttt{>}) = [\overline{\texttt{a/o}}]\texttt{U}_i}$$

$$\frac{\texttt{class C}_m\texttt{<}\overline{\texttt{o}\rightarrow\texttt{[b}_l\ \texttt{b}_u\texttt{]}}\texttt{> extends N } \{\overline{\texttt{U f;}}\ \overline{\texttt{W}}\} \qquad \texttt{f}_i \notin \texttt{f}}{fType(\texttt{f}_i, \texttt{C<}\overline{\texttt{a}}\texttt{>}) = fType(\texttt{f}_i, [\overline{\texttt{a/o}}]\texttt{N})}$$

$$\frac{\texttt{class C}_m\texttt{<}\overline{\texttt{o}\rightarrow\texttt{[b}_l\ \texttt{b}_u\texttt{]}}\texttt{> } \{\overline{\texttt{U f;}}\ \overline{\texttt{W}}\} \qquad \texttt{<}\overline{\texttt{o}' \rightarrow \texttt{[b}'_l\ \texttt{b}'_u\texttt{]}}\texttt{> T m}_n\ (\overline{\texttt{T x}})\ \{\texttt{return e;}\} \in \overline{\texttt{W}}}{mBody(\texttt{m<}\overline{\texttt{a}'}\texttt{>}, \texttt{C<}\overline{\texttt{a}}\texttt{>}) = (\overline{\texttt{x}}; [\overline{\texttt{a/o}},\ \overline{\texttt{a}'/\texttt{o}'}]\texttt{e})}$$

$$\frac{\texttt{class C}_m\texttt{<}\overline{\texttt{o}\rightarrow\texttt{[b}_l\ \texttt{b}_u\texttt{]}}\texttt{> extends N } \{\overline{\texttt{U f;}}\ \overline{\texttt{W}}\} \qquad \texttt{m} \notin \overline{\texttt{W}}}{mBody(\texttt{m<}\overline{\texttt{a}'}\texttt{>}, \texttt{C<}\overline{\texttt{a}}\texttt{>}) = mBody(\texttt{m<}\overline{\texttt{a}'}\texttt{>}, [\overline{\texttt{a/o}}]\texttt{N})}$$

$$\frac{\texttt{class C}_m\texttt{<}\overline{\texttt{o}\rightarrow\texttt{[b}_l\ \texttt{b}_u\texttt{]}}\texttt{> } \{\overline{\texttt{U f;}}\ \overline{\texttt{W}}\} \qquad \texttt{<}\overline{\texttt{o}' \rightarrow \texttt{[b}'_l\ \texttt{b}'_u\texttt{]}}\texttt{> T m}_n\ (\overline{\texttt{T x}})\ \{\texttt{return e;}\} \in \overline{\texttt{W}}}{mType(\texttt{m<}\overline{\texttt{a}'}\texttt{>}, \texttt{C<}\overline{\texttt{a}}\texttt{>}) = ([\overline{\texttt{a/o}},\ \overline{\texttt{a}'/\texttt{o}'}]\texttt{<}\overline{\texttt{o}' \rightarrow \texttt{[b}'_l\ \texttt{b}'_u\texttt{]}}\texttt{>}(\overline{\texttt{T}} \rightarrow \texttt{T}); \texttt{n})}$$

$$\frac{\texttt{class C}_m\texttt{<}\overline{\texttt{o}\rightarrow\texttt{[b}_l\ \texttt{b}_u\texttt{]}}\texttt{> extends N } \{\overline{\texttt{U f;}}\ \overline{\texttt{W}}\} \qquad \texttt{m}_n \notin \overline{\texttt{W}}}{mType(\texttt{m<}\overline{\texttt{a}'}\texttt{>}, \texttt{C<}\overline{\texttt{a}}\texttt{>}) = mType(\texttt{m<}\overline{\texttt{a}'}\texttt{>}, [\overline{\texttt{a/o}}]\texttt{N})}$$

$$\frac{mType(\texttt{m<}\overline{\texttt{o}}\texttt{>}, \texttt{C<}\overline{\texttt{a}'}\texttt{>})\ is\ undefined}{override(\texttt{m}, \texttt{n}, \texttt{C<}\overline{\texttt{a}'}\texttt{>}, \texttt{<}\overline{\texttt{o}\rightarrow\texttt{[b}_l\ \texttt{b}_u\texttt{]}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{T}_0)}$$

$$\frac{mType(\texttt{m<}\overline{\texttt{o}}\texttt{>}, \texttt{C<}\overline{\texttt{a}'}\texttt{>}) = (\overline{\texttt{T}} \rightarrow \texttt{T}_0; \texttt{n}) \qquad \texttt{n}' \leq \texttt{n}}{override(\texttt{m}, \texttt{n}', \texttt{C<}\overline{\texttt{a}'}\texttt{>}, \texttt{<}\overline{\texttt{o}\rightarrow\texttt{[b}_l\ \texttt{b}_u\texttt{]}}\texttt{>}\overline{\texttt{T}} \rightarrow \texttt{T}_0)}$$

Figure 9. Field and method lookup functions for FJ I O.

$$I(\texttt{C}_m\texttt{<}\overline{\texttt{a}}\texttt{>}) = m$$

$$\texttt{immutable} \leq \texttt{mutable}$$

$$m \leq m$$

Figure 10. Mutability functions and relations for FJ I O.

$$\frac{I(\texttt{N}) = \texttt{immutable}}{\texttt{N} \triangleright \texttt{C}_m\texttt{<}\overline{\texttt{a}}\texttt{>} = \texttt{C}_{immutable}\texttt{<}\overline{\texttt{a}}\texttt{>}}$$

$$\frac{I(\texttt{N}) = \texttt{mutable}}{\texttt{N} \triangleright \texttt{C}_m\texttt{<}\overline{\texttt{a}}\texttt{>} = \texttt{C}_m\texttt{<}\overline{\texttt{a}}\texttt{>}}$$

Figure 11. Viewpoint adaptation for FJ I O.

The typing rules are shown in Figure 12. The type of a field access is the viewpoint adapted field type after substitution of `this` for γ in type `T` (T-FIELD).

The typing rule for field assignment T-ASSIGN states that a field assignment is possible if the viewpoint adapted field type is mutable and the expression of the new value is of the given type. Then, `this` is substituted for γ in type `T'`.

The rule T-NEW states that a newly created instance of a class is of this class if the class is well-defined.

The rule T-SUB is the subsumption rule, if an expression `e` is of type `T'`, a subtype of `T`, `e` can be used where an object of type `T` is expected.

The invocation rule T-INVK checks that mutability parameter of the method matches, checks for the ownership bounds, and returns the viewpoint adapted, substituted (again `this` for γ) type `T`.

The rule T-CLASS defines that if the owners are well-formed, and the method definition, fields and superclass are ok, the class itself is well-typed.

The rule T-METHOD shows what preconditions are needed in order to have a well-typed method, namely the type of the body expression has to be the defined return type, the helper function override must be valid, and the mutability parameter must be less or equal than the declared mutability of the class.

5. Proofs

5.1 Type Soundness

Type soundness guarantees that the types of variables accurately reflect their contents, including ownership information. Soundness is shown by proving progress and preservation (subject reduction).

$$\frac{\Delta;\Gamma \vdash \gamma : \mathtt{N} \quad fType(\mathtt{f},\mathtt{N}) = \mathtt{T}}{\Delta;\Gamma \vdash \gamma.\mathtt{f} : \mathtt{N} \triangleright [\gamma/\mathtt{this}]\mathtt{T}} \text{ (T-FIELD)}$$

$$\frac{\Delta;\Gamma \vdash \gamma : \mathtt{N} \quad fType(\mathtt{f},\mathtt{N}) = \mathtt{T} \quad \mathtt{T}' = \mathtt{N}\triangleright\mathtt{T} \quad I(\mathtt{T}') = \mathtt{mutable} \quad \Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}'}{\Delta;\Gamma \vdash \gamma.\mathtt{f} = \mathtt{e} : [\gamma/\mathtt{this}]\mathtt{T}'} \text{ (T-ASSIGN)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{C}_m\mathtt{<}\overline{\mathtt{a}}\mathtt{>}\ \text{OK}}{\Delta;\Gamma \vdash \mathtt{new}\ \mathtt{C}_m\mathtt{<}\overline{\mathtt{a}}\mathtt{>} : \mathtt{C}_m\mathtt{<}\overline{\mathtt{a}}\mathtt{>}} \text{ (T-NEW)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}' \quad \Delta;\Gamma \vdash \mathtt{T}' <: \mathtt{T} \quad \Delta;\Gamma \vdash \mathtt{T}\ \text{OK}}{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T}} \text{ (T-SUB)}$$

$$\frac{\Delta;\Gamma \vdash \gamma : \mathtt{N} \quad \Delta;\Gamma \vdash \overline{\mathtt{e} : \mathtt{N}\triangleright\mathtt{T}} \quad \Delta;\Gamma \vdash \overline{\mathtt{a}}\ \text{OK} \quad mType(\mathtt{m}\mathtt{<}\overline{\mathtt{a}}\mathtt{>},\ \mathtt{N}) = (\mathtt{<}\overline{\mathtt{o}\rightarrow[\mathtt{b}_l\ \ \mathtt{b}_u]}\mathtt{>}\overline{\mathtt{T}}\rightarrow\mathtt{T};\mathtt{n}) \quad I(\mathtt{N}) = \mathtt{n} \quad \Delta;\Gamma \vdash \overline{\mathtt{a} \preceq \mathtt{b}_u} \quad \Delta;\Gamma \vdash \overline{\mathtt{b}_l \preceq \mathtt{a}}}{\Delta;\Gamma \vdash \gamma.\mathtt{<}\overline{\mathtt{a}}\mathtt{>}\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{N}\triangleright[\gamma/\mathtt{this}]\mathtt{T}} \text{ (T-INVK)}$$

$$\frac{\mathtt{this}{:}\mathtt{C}_m\mathtt{<}\overline{\mathtt{o}}\mathtt{>} \vdash \overline{\mathtt{o}\rightarrow[\mathtt{b}_l\ \ \mathtt{b}_u]}\ \text{OK} \quad \overline{\mathtt{o}\rightarrow[\mathtt{b}_l\ \ \mathtt{b}_u]};\mathtt{this}{:}\mathtt{C}_m\mathtt{<}\overline{\mathtt{o}}\mathtt{>} \vdash \overline{\mathtt{W}},\ \overline{\mathtt{T}},\ \mathtt{N}\ \text{OK}}{\vdash \mathtt{class}\ \mathtt{C}_m\mathtt{<}\overline{\mathtt{o}\rightarrow[\mathtt{b}_l\ \ \mathtt{b}_u]}\mathtt{>}\ \mathtt{extends}\ \mathtt{N}\ \{\overline{\mathtt{T}\ \mathtt{f};}\ \overline{\mathtt{W}}\}\ \text{OK}} \text{ (T-CLASS)}$$

$$\frac{\Delta' = \Delta, \overline{\mathtt{o}\rightarrow[\mathtt{b}_l\ \ \mathtt{b}_u]} \quad \Gamma = \mathtt{this}{:}\mathtt{C}_{n'}\mathtt{<}\overline{\mathtt{o}'}\mathtt{>}, \overline{\mathtt{x}{:}\mathtt{T}} \quad \Delta;\mathtt{this}{:}\mathtt{C}_{n'}\mathtt{<}\overline{\mathtt{o}'}\mathtt{>} \vdash \overline{\mathtt{o}\rightarrow[\mathtt{b}_l\ \ \mathtt{b}_u]}\ \text{OK} \quad \Delta';\mathtt{this}{:}\mathtt{C}_{n'}\mathtt{<}\overline{\mathtt{o}'}\mathtt{>} \vdash \mathtt{T},\overline{\mathtt{T}}\ \text{OK} \quad \Delta';\Gamma \vdash \mathtt{e} : \mathtt{T} \quad override(\mathtt{m},\mathtt{n}',\mathtt{N},\mathtt{<}\overline{\mathtt{o}\rightarrow[\mathtt{b}_l\ \ \mathtt{b}_u]}\mathtt{>}\overline{\mathtt{T}} \rightarrow \mathtt{T}) \quad \mathtt{n}' \preceq \mathtt{n}}{\Delta;\mathtt{this}{:}\mathtt{C}_n\mathtt{<}\overline{\mathtt{o}'}\mathtt{>} \vdash \mathtt{<}\overline{\mathtt{o}\rightarrow[\mathtt{b}_l\ \ \mathtt{b}_u]}\mathtt{>}\ \mathtt{T}\ \mathtt{m}_{n'}(\overline{\mathtt{T}\ \mathtt{x}})\ \{\mathtt{return}\ \mathtt{e};\}\ \text{OK}} \text{ (T-METHOD)}$$

Figure 12. FJ I O expression and class typing rules.

Progress For any $\mathcal{H}$, e, T if (a) $\mathcal{H} \vdash \mathtt{e{:}T}$ and (b) $\mathcal{H}$ OK then either $\mathcal{H}'$, e′ exists such that (c) $\mathtt{e};\mathcal{H} \rightsquigarrow \mathtt{e}';\mathcal{H}'$ or (d) there exists a v, such that e = v.

The proof of the progress theorem is done by structural induction on the derivation of $\emptyset;\mathcal{H} \vdash \mathtt{e{:}T}$ with a case analysis on the last step.

Please see the appendix for the detailed proof and the additional lemma required.

Subject reduction For any Δ, $\mathcal{H}$, $\mathcal{H}'$, e, e′, T if (a) $\Delta;\mathcal{H} \vdash \mathtt{e{:}T}$ and (b) $\mathtt{e};\mathcal{H} \rightsquigarrow \mathtt{e}';\mathcal{H}'$ and (c) $\Delta;\mathcal{H} \vdash \mathtt{e}$ OKand (d) $\emptyset;\mathcal{H} \vdash \Delta$ OKand (e) $\Delta \vdash \mathcal{H}$ OKand (f) $\mathtt{e}' \neq \mathtt{err}$ then (g) $\Delta;\mathcal{H}' \vdash \mathtt{e}'{:}\mathtt{T}$ and (h) $\Delta;\mathcal{H}' \vdash \mathtt{e}'$ OKand (i) $\Delta \vdash \mathcal{H}'$ OK.

The proof of the subject reduction theorem is done by structural induction on the derivation of $\mathtt{e};\mathcal{H} \rightsquigarrow \mathtt{e}';\mathcal{H}'$ with a case analysis on the last step.

Please see the appendix for the detailed proof and the additional lemmas required.

5.2 Immutability Invariant

Let E[·] be an execution context, which describe field updates and method calls present within an expression:

```
E[·] ::= [·] | E[·].f | E[·].f = e | e.f = E[·]
       | E[·].m(e) | e.m(E[·])
```

$$\frac{\mathtt{e}';\mathcal{H} \rightsquigarrow \mathtt{e}'';\mathcal{H}' \quad \mathtt{e}'' \neq \mathtt{err}}{\mathtt{E}[\iota.\mathtt{f} = \mathtt{e}'];\mathcal{H} \rightsquigarrow \mathtt{E}[\iota.\mathtt{f} = \mathtt{e}''];\mathcal{H}'} \text{ (RC-EXECCON)}$$

INVARIANT: If $\mathtt{e};\mathcal{H}\rightsquigarrow\mathtt{e}';\mathcal{H}'$ and $\mathcal{H}(\iota) \neq \mathcal{H}'(\iota)$ with $\iota \rightarrow \{\mathtt{R}_m;\overline{\mathtt{f}\rightarrow\mathtt{v}'{:}\mathtt{R}'}\}$ then $\forall\iota$ where $\mathcal{H}(\iota) \neq \mathcal{H}'(\iota)$ there exists f, v and E[·] such that $\mathtt{e} = \mathtt{E}[\iota.\mathtt{f} := \mathtt{v}]$ and $I(R_m) = I(R') = \mathtt{mutable}$

Proof sketch: The only mutation is field assignment. The T-ASSIGN rule can only be applied if the viewpoint adapted type of the field is mutable.

Helper lemma: IMMUTABLE-OBJECT-HAS-IMMUTABLE-FIELDS: $e : T;\mathcal{H} \rightsquigarrow^* v;\mathcal{H}'$, if I(T)=immutable, all fields in v are immutable.

Proof: viewpoint adaptation definition

5.3 Ownership Invariant

$$\frac{}{owner(\mathtt{C}_m\mathtt{<}\overline{\mathtt{a}}\mathtt{>}) = \mathtt{a}_0}$$

INVARIANT: For any $\mathcal{H}\vdash\mathtt{e{:}T}$, $owner(\mathtt{T}) = \mathtt{a}$. Given $\mathtt{e};\mathcal{H}\rightsquigarrow^*\iota;\mathcal{H}'$ with $\iota \rightarrow\{\mathtt{R}_m\mathtt{<}\overline{\mathtt{r}}\mathtt{>};\ \ldots\}$, let the owner of ι be $owner(\mathtt{R}_m\mathtt{<}\overline{\mathtt{r}}\mathtt{>}) = \mathtt{r}$. Then $\mathcal{H}\vdash\mathtt{r} \preceq \mathtt{a}$.

Proof sketch: Show that for all reduction rules if e reduces to e′, the owner is preserved. The only interesting case is T-ASSIGN, and the owner of the new value has to be inside of the field type.

6. Discussion and Related Work

Ever since the Universes type system [14] that unified readonly references by introducing owners-as-modifiers there have been attempts to unify ownership and immutability [13]. The benefits of adding immutability to ownership were clear as owners-as-modifiers allow more flexible language expressions than owners-as-dominators.

The benefits of adding ownership to immutability however were only recently discovered [12, 19]. Frozen Objects demonstrated how immutable objects can be constructed even beyond the constructor allowing immutable cyclic data structures as long as a verification system is present to make sure safety [12]. Ownership and Immutability for Generic Java (OIGJ) [19] demonstrated how to provide safe immutably object construction of immutable cyclic data structures in a simple type-checkable fashion without the need for more complex program verification set up.

Other mergers included just immutable objects and deep ownership [10] but without readonly references and class-wide ownership and all three kinds of immutability [16] but with limited expressivity. In fact, over the past several years most people came to expect any ownership system to have immutability support and vice-versa. Modern languages such as X10 [15] support both ownership and immutability as the only way forward.

7. Conclusion

In this paper, our goal was simple: to prove that adding immutability to a prescriptive ownership discipline will safely allow limited ownership variance. We have described the language we set up that incorporates three kinds of immutability in addition to prescriptive ownership and provided all the theorems and proof outlines required. We hope that this workshop paper will provide a small but useful contribution to the foundations of object-oriented languages community.

References

[1] Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, volume 3086, pages 1–25, Oslo, Norway, June 2004. Springer-Verlag.

[2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, pages 311–330, Seattle, WA, USA, November 2002. ACM Press.

[3] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *OOPSLA*, pages 35–49, 2004.

[4] Nicholas Cameron, Sophia Drossopoulou, James Noble, and Matthew Smith. Multiple ownership. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 441–460, New York, NY, USA, 2007. ACM.

[5] David Clarke, John Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, pages 48–64, Vancouver, Canada, October 1998. ACM Press.

[6] David Clarke and Tobias Wrigstad. External Uniqueness is Unique Enough. In *ECOOP*, volume 2473 of *(LNCS)*, pages 176–200, Darmstadt, Germany, July 2003. Springer-Verlag.

[7] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. In *ECOOP*, 2007.

[8] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic universe types. *International Workshop on Foundations of Object Oriented Languages (FOOL) 2007*, pages 1–13, Jan 2007.

[9] Werner Dietl and Peter Müller. Universes: Lightweight ownership for jml. *Journal of Object Technology*, 4(8):5–32, Sep 2005.

[10] Christian Haack, Erik Poll, Jan Schäfer, and Aleksy Schubert. Immutable objects for a java-like language. In *ESOP*, pages 347–362, March 2007.

[11] Atsushi Igarashi, Benjamin C Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396—450, Jan 2001.

[12] K. Rustan M. Leino, Peter Müller, and Angela Wallenburg. Flexible immutability with frozen objects. In *VSTTE*, pages 192–208, October 2008.

[13] Paley Li, Alex Potanin, James Noble, and Lindsay Groves. Towars unifying ownership and immutability. In *(IWACO)*, 2008.

[14] P. Müller and A. Poetzsh-Heffter. Programming Languages and Fundamentals of Programming. Technical report, Fernuniversität Hagen, 2001. Poetzsh-Heffter, A. and Meyer, J. (editors).

[15] Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *OOPSLA*, pages 457–474. ACM Press, October 2008.

[16] Johan Östlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. Ownership, uniqueness and immutability. In *TOOLS Europe 2008*, 2008.

[17] Matthew Tschantz and Michael Ernst. Javari: adding reference immutability to java. *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Oct 2005.

[18] Yoav Zibin, Alex Potanin, Shay Artzi, Adam Kiezun, and Michael D. Ernst. Object and reference immutability using Java generics. In *Foundations of Software Engineering*, 2007.

[19] Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic java. In *OOPSLA*, Reno, Nevada, USA, October 2010. ACM Press.

Appendix: Rules and Proofs Omitted from the Paper

Progress For any $\mathcal{H}$, e, T if (a) $\mathcal{H} \vdash$ e:T and (b) $\mathcal{H}$ OK then either $\mathcal{H}'$,e$'$ exists such that (c) e;$\mathcal{H} \rightsquigarrow$ e$'$;$\mathcal{H}'$ or (d) there exists a v, such that e = v.

The detailed proof of the progress theorem:

- T-FIELD
  1. e = γ.f by def T-FIELD
  2. γ:N by premise T-FIELD
  3. γ = $\iota \in \mathcal{H}$ by CLOSED-LEMMA, 2
  4. T<:N$\triangleright$T byRHD-SUPERTYPE, S-MUTABLE
  5. done by 1, 3, R-FIELD
- T-ASSIGN
  1. $e = \gamma$.f $= e''$ by def T-ASSIGN
  2. γ:N by premise T-ASSIGN
  3. $\mathtt{T}'_{mutable}$ = N$\triangleright$T by premise T-ASSIGN
  4. e'':T$'$ by premise T-ASSIGN, def of $\triangleright$
  5. γ = $\iota \in \mathcal{H}$ by CLOSED-LEMMA, 2
  6. $e'', \mathcal{H} \rightsquigarrow e''', \mathcal{H}'$ or $\exists v : e' = v$ by 3, b, induction hyp
  7. case analysis on e'':
  8. $e'', \mathcal{H} \rightsquigarrow e''', \mathcal{H}'$ by 1, 4, RC-ASSIGN or RC-ASSIGN-ERR
  9. $\exists v : e' = v$ by 1, 4, $e'' = v$, R-ASSIGN
- T-NEW
  1. e = new $\mathtt{C}_m$<$\overline{\mathtt{a}}$> by def T-NEW
  2. $\mathcal{H} \vdash$ $\mathtt{C}_m$<$\overline{\mathtt{a}}$> OK by premise T-NEW
  3. class $\mathtt{C}_m$<$\overline{\mathtt{o}\rightarrow[\mathtt{b}_l\ \mathtt{b}_u]}$> by 2, F-CLASS
  4. $\overline{\mathtt{a}}$ OK by 2, F-CLASS
  5. $[\overline{\mathtt{a/o}}]$ $\mathtt{b}_l \preceq$ a by 2, F-CLASS
  6. a $\preceq$ $[\overline{\mathtt{a/o}}]$ $\mathtt{b}_u$ by 2, F-CLASS
  7. $\overline{\mathtt{a}}$ = $\overline{\mathtt{r}}$ by 4, syntax of a
  8. fields($\mathtt{C}_m$) = $\overline{f}$ by 3, def fields
  9. done by 1, 7, 8, R-NEW
- T-SUB
  1. e : T by def T-SUB
  2. e : T' by premise T-SUB
  3. T' <: T by premise T-SUB
  4. done by 1, 3
- T-INVK
  1. $e = \gamma$.<$\overline{\mathtt{a}}$>$\mathtt{m}_n$($\overline{\mathtt{e}}$) by def T-INVK
  2. γ:N by premise T-INVK
  3. $\overline{e\text{:N} \triangleright \text{T}}$ by premise T-INVK
  4. $\overline{\mathtt{a}}$ OK by premise T-INVK
  5. $\overline{\mathtt{a} \preceq \mathtt{b}_u}$ by premise T-INVK
  6. $\overline{\mathtt{B}_l \preceq \mathtt{a}}$ by premise T-INVK
  7. mType defined by premise T-INVK
  8. γ = $\iota \in$ dom($\mathcal{H}$) by 2, CLOSED-LEMMA
  9. $\overline{\mathtt{a}}$ = $\overline{\mathtt{r}}$ by 2, def syntax r

10. $\forall\ e_i \in \overline{\mathtt{e}}{:}e_i, \mathcal{H} \rightsquigarrow e'_i, \mathcal{H}'$ or $\exists\ \mathtt{v}{:}e_i = \mathtt{v}$ by 3, b, induction hyp
11. case analysis on $\overline{\mathtt{e}}$:
12. case $\exists\ e_i \in \overline{\mathtt{e}}{:}e_i, \mathcal{H} \rightsquigarrow e'_i, \mathcal{H}'$
13. done by 1, 8, 9, RC-INVK or RC-INVK-ERR
14. case $\forall\ e_i \in \overline{\mathtt{e}}{:}\exists\ \mathtt{v}{:}e_i = \mathtt{v}$
15. `mBody defined` by 7, def mBody, mType
16. done by 1, 8, 9, $e'' = \mathtt{v}$, R-ASSIGN

A required lemma in the proof is as follows:

- CLOSED-LEMMA Well-typed expressions are closed:
  - if $\Delta; \Gamma \vdash e{:}\mathtt{T}$ then:
  - $\forall \gamma \in fv_\gamma(e) : \gamma \in dom(\Gamma)$
  - $\forall o \in fv_o(e) : o \in dom(\Delta)$
- RHD-SUPERTYPE: if $\Delta; \Gamma \vdash \mathtt{N},\ \mathtt{T}$ then $\forall \mathtt{N},\ \mathtt{T}{:}\mathtt{T}{<:}\mathtt{N} \triangleright \mathtt{T}$

Subject reduction For any $\Delta,\ \mathcal{H},\ \mathcal{H}',\ \mathtt{e},\ \mathtt{e}',\ \mathtt{T}$ if (a) $\Delta;\mathcal{H} \vdash \mathtt{e}{:}\mathtt{T}$ and (b) $\mathtt{e};\mathcal{H} \rightsquigarrow \mathtt{e}';\mathcal{H}'$ and (c) $\Delta; \mathcal{H} \vdash \mathtt{e}$ OKand (d) $\emptyset; \mathcal{H} \vdash \Delta$ OKand (e) $\Delta \vdash \mathcal{H}$ OKand (f) $\mathtt{e}' \neq \mathtt{err}$ then (g) $\Delta; \mathcal{H}' \vdash \mathtt{e}'{:}\mathtt{T}$ and (h) $\Delta; \mathcal{H}' \vdash \mathtt{e}'$ OKand (i) $\Delta \vdash \mathcal{H}'$ OK.

The detailed proof of the subject reduction theorem:

- R-FIELD-NULL, R-ASSIGN-NULL, R-INVK-NULL, RC-ASSIGN-ERR, RC-INVK-ERR
  1. N/A by e
- R-FIELD
  1. $\mathtt{e} = \iota.f_i$ by def R-FIELD
  2. $\mathtt{e}' = \mathtt{v}_i$ by def R-FIELD
  3. $\mathcal{H}' = \mathcal{H}$by def R-FIELD
  4. $\mathcal{H}(\iota) = \{\mathtt{R}; \overline{\mathtt{f} \rightarrow \mathtt{v}}\}$ by premise R-FIELD
  5. $\Delta;\mathcal{H} \vdash \iota{:}\mathtt{N}$ by a, 1, INVERSION-LEMMA (FIELD ACCESS)
  6. $\mathtt{fType}(\mathtt{f}_i,\ \mathtt{N}) = \mathtt{T}'$ by a, 1, INVERSION-LEMMA (FIELD ACCESS)
  7. $\Delta;\mathcal{H} \vdash \mathtt{T}{<:}\mathtt{T}'$ by a, 1, INVERSION-LEMMA (FIELD ACCESS)
  8. $\Delta;\mathcal{H} \vdash \mathtt{R}{<:}\mathtt{N}$ by 5, INVERSION-LEMMA (ADDRESS), 4
  9. $\Delta \vdash \mathcal{H}$ OKby e
  10. $\Delta;\mathcal{H} \vdash \mathtt{v}_i{:}\mathtt{T}'$ by 4, 6, 8, 9, def F-HEAP
  11. $\Delta;\mathcal{H} \vdash \mathtt{T}$ OKby RUNTIME-TYPE-CHECKING-GIVES-WELL-FORMED-TYPES-LEMMA, a, d, 9
  12. $\Delta;\mathcal{H} \vdash \mathtt{v}_i{:}\mathtt{T}$ by 10, 11, 7, T-SUB
  13. $\Delta;\mathcal{H}' \vdash \mathtt{e}'{:}\mathtt{T}$ by 12, 2, 3
  14. $\forall \iota \in \mathtt{fv}(\mathtt{e}'){:}\iota \in \mathtt{dom}(\mathcal{H}')$ by 13, CLOSED-LEMMA
  15. $\Delta;\mathcal{H}' \vdash \mathtt{e}'$ OK by 3, 9, 14
  16. $\Delta \vdash \mathcal{H}'$ OKby 3, 9
  17. done by 13, 15, 16
- R-ASSIGN
  1. $\mathtt{e} = \iota.\mathtt{f}_i = \mathtt{v}$ by def R-ASSIGN
  2. $\mathtt{e}' = \mathtt{v}$ by def R-ASSIGN
  3. $\mathcal{H}(\iota) = \{\mathtt{R};\ \overline{\mathtt{f} \rightsquigarrow \mathtt{v}}\}$ by premise R-ASSIGN
  4. $\mathcal{H}' = \mathcal{H}[\iota \mapsto \{R; \overline{\mathtt{f} \rightsquigarrow \mathtt{v}[\mathtt{f}_i \mapsto \mathtt{v}]}\}]$ by premise R-ASSIGN
  5. $\Delta;\mathcal{H} \vdash \iota{:}\mathtt{N}$ by a, 1, INVERSION-LEMMA (FIELD ASSIGNMENT)
  6. $\mathrm{ftype}(\mathtt{f}_i,\ \mathtt{N}) = \mathtt{T}'$ by a, 1, INVERSION-LEMMA (FIELD ASSIGNMENT)
  7. $\Delta;\mathcal{H} \vdash \mathtt{v}{:}\mathtt{T}'$ by a, 1, INVERSION-LEMMA (FIELD ASSIGNMENT)
  8. $\Delta;\mathcal{H} \vdash \mathtt{T}'{<:}\mathtt{T}$ by a, 1, INVERSION-LEMMA (FIELD ASSIGNMENT)
  9. $\Delta \vdash \mathcal{H}$ OKby e
  10. $\Delta;\mathcal{H} \vdash \mathtt{T}$ OKby a, d, 9,RUNTIME-TYPE-CHECKING-GIVES-WELL-FORMED-TYPES-LEMMA
  11. $\Delta;\mathcal{H} \vdash \mathtt{v}{:}\mathtt{T}$ by 7, 8, 10
  12. $\Delta;\mathcal{H}' \vdash \mathtt{v}{:}\mathtt{T}$ by 11, REDUCTION-PRESERVES-HEAP-LEMMA
  13. assume that v is defined
  14. $\mathtt{v} \in \mathcal{H}'$ by 13, 12, WELL-TYPED-VALUES-HAVE-ADDRESSES-LEMMA
  15. assume $\mathtt{v} \neq \mathtt{null}$ is defined, else goto 19
  16. $\Delta; \mathcal{H} \vdash \mathtt{R}{<:}\mathtt{N}$ by 3, 5, INVERSION-LEMMA (ADDRESS)
  17. $\Delta; \mathcal{H} \vdash \mathtt{owner}(v) = \mathtt{owner}(\mathtt{T}')$ by 7, def OWNER
  18. $\Delta; \mathcal{H}' \vdash \mathtt{owner}(v) = \mathtt{owner}(\mathtt{T})$ by 7, 8, 12, 17, def OWNER
  19. $\Delta \vdash \mathcal{H}'$ OKby 9, 4, 6, 7, 13, 15 or 18, def F-HEAP
  20. $\forall \iota \in fv(e') : \iota \in dom(\mathcal{H}')$ by 12, CLOSED-LEMMA
  21. $\Delta;\mathcal{H}' \vdash \mathtt{e}'$ OKby 19, 20
  22. done by 21, 12
- R-NEW
  1. $\mathtt{e} = \mathtt{new}\ \mathtt{C}_m\mathtt{<}\overline{\mathtt{r}}\mathtt{>}$ by def R-NEW
  2. $\mathtt{e}' = \iota$ by def R-NEW
  3. $\mathcal{H}(\iota)$ undefined by premise R-NEW
  4. $\mathrm{fields}(\mathtt{C}) = \overline{\mathtt{f}}$ by premise R-NEW
  5. $\mathcal{H}' = \mathcal{H}, \iota \rightsquigarrow \mathtt{C}_m\mathtt{<}\overline{\mathtt{r}}\mathtt{>}; \overline{\mathtt{f} \rightsquigarrow \mathtt{null}}$ by premise R-NEW
  6. $\Delta; \mathcal{H} \vdash \mathtt{C<}\overline{\mathtt{r}}\mathtt{>}{<:}\mathtt{T}$ by 1, a, INVERSION-LEMMA (NEW)
  7. $\Delta; \mathcal{H} \vdash \mathtt{C<}\overline{\mathtt{r}}\mathtt{>}$ OKby 1, a, INVERSION-LEMMA (NEW)
  8. $\Delta; \mathcal{H}' \vdash \iota{:}\mathtt{C<}\overline{\mathtt{r}}\mathtt{>}$ by 5
  9. $\Delta; \mathcal{H}' \vdash \mathtt{C<}\overline{\mathtt{r}}\mathtt{>}{<:}\mathtt{T}$ by 6, 5, SUBTYPE-WEAKENING-LEMMA
  10. $\Delta \vdash \mathcal{H}$ OKby e
  11. $\Delta; \mathcal{H} \vdash \mathtt{T}$ OKby a, d, 10, RUNTIME-TYPE-CHECKING-GIVES-WELL-FORMED-TYPES-LEMMA
  12. $\Delta; \mathcal{H}' \vdash \iota{:}\mathtt{T}$ by 8, 9, 11, T-SUB
  13. let $\overline{\mathtt{fType}(\mathtt{f},\ \mathtt{C<}\overline{\mathtt{r}}\mathtt{>}) = \mathtt{U}}$
  14. $\Delta; \mathcal{H} \vdash \overline{\mathtt{U}}$ OKby 13, 7, FTYPE-WELL-FORMED-LEMMA
  15. $\Delta; \mathcal{H} \vdash \overline{\mathtt{null}{:}\mathtt{U}}$ by 14, T-NULL
  16. $\Delta; \vdash \iota \rightarrow \{\mathtt{C<}\overline{\mathtt{r}}\mathtt{>};\ \overline{\mathtt{f} \rightarrow \mathtt{null}}\}$ OKby 10, 7, 13, 15, def T-HEAP
  17. $\Delta \vdash \mathcal{H}'$ OKby 16, 5
  18. $\forall \iota \in fv(e') : \iota \in \mathtt{dom}(\mathcal{H}')$ by 2, 5
  19. $\Delta;\mathcal{H}' \vdash \mathtt{e}'$ OK by 18, 17
  20. done by 12, 19
- R-INVK
  1. $\mathtt{e} = \iota.\mathtt{<}\overline{\mathtt{r}}\mathtt{>m}(\overline{\mathtt{v}})$ by def R-INVK
  2. $\mathtt{e}' = [\overline{\mathtt{v}/\mathtt{x}}]\mathtt{e}_0$ by def R-INVK
  3. $\mathcal{H}' = \mathcal{H}$ by def R-INVK

4. $\mathcal{H}(\iota)$ = $\{\texttt{R;...}\}$ by premise R-INVK
5. mBody($\texttt{m<}\overline{\texttt{r}}\texttt{>,R}$) = $\overline{\texttt{x}}\texttt{;e}_0$ by premise R-INVK
6. $\Delta;\mathcal{H} \vdash \iota\texttt{:N}$ by 1, a, INVERSION-LEMMA (INVOKE)
7. $\texttt{mType(m,N)=(}\overline{\texttt{T}}\rightarrow\texttt{T;n)}$ by 1, a, INVERSION-LEMMA (INVOKE)
8. $\Delta;\mathcal{H} \vdash \overline{\texttt{e:T}}$ by 1, a, INVERSION-LEMMA (INVOKE)
9. $\Delta;\mathcal{H} \vdash \overline{\texttt{r}}$ OKby 1, a, INVERSION-LEMMA (INVOKE)
10. $\Delta;\mathcal{H} \vdash \overline{\texttt{U}}$ OKby 1, a, INVERSION-LEMMA (INVOKE)
11. $\Delta;\mathcal{H} \vdash \texttt{T}'\texttt{<:T}$ by 1, a, INVERSION-LEMMA (INVOKE)
12. $\Delta;\mathcal{H} \vdash \texttt{R<:N}$ by 4, 6, INVERSION-LEMMA (ADDRESS)
13. $\Delta \vdash \mathcal{H}$ OKby e
14. $\Delta;\mathcal{H}\overline{\texttt{x:T}}\vdash\texttt{e}_0\texttt{:T}'$ by 5, 7, 6, 9, 10, d, 13, BODY-HAS-RETURN-TYPE-LEMMA
15. $\Delta;\mathcal{H} \vdash [\overline{\texttt{v/x}}]\texttt{e}_0\texttt{:}[\overline{\texttt{v/x}}]\texttt{T}'$ by 14, 6, 13, d, S-REFLEX, VALUE-SUBSTITUTION-PRESERVES-TYPING-LEMMA
16. $\Delta;\mathcal{H} \vdash \texttt{T}'$ OKby 14, 11, RUNTIME-TYPE-CHECKING-GIVES-WELL-FORMED-TYPES-LEMMA
17. $\Delta;\mathcal{H} \vdash [\overline{\texttt{v/x}}]\texttt{e}_0\texttt{:T}'$ by 15, 16
18. $\Delta;\mathcal{H} \vdash \texttt{T}$ OKby a, d, 13, RUNTIME-TYPE-CHECKING-GIVES-WELL-FORMED-TYPES-LEMMA
19. $\Delta;\mathcal{H} \vdash [\overline{\texttt{v/x}}]\texttt{e}_0\texttt{:T}$ by 17, 11, 18, T-SUB
20. $\Delta;\mathcal{H}' \vdash \texttt{e}'\texttt{:T}$ by 2, 3, 19
21. $\forall\iota \in \texttt{fv}(\overline{\texttt{v}})\texttt{:}\iota \in \texttt{dom}(\mathcal{H})$ by 1, c
22. $\forall\iota \in \texttt{fv}(\overline{\texttt{e}_0})\texttt{:}\iota \in \texttt{dom}(\mathcal{H})\texttt{,}\overline{\texttt{x}}$ by 14, CLOSED-LEMMA
23. $\forall\iota \in \texttt{fv}(\overline{[\overline{\texttt{v/x}}]\texttt{e}_0})\texttt{:}\iota \in \texttt{dom}(\mathcal{H})$ by 21, 22
24. $\forall\iota \in \texttt{fv}(\texttt{e}')\texttt{:}\iota \in \texttt{dom}(\mathcal{H})$ by 23, 2
25. $\Delta;\mathcal{H}' \vdash \texttt{e}'$ OKby 3, 13, 24
26. done by 20, 25

- RC-ASSIGN
  1. $\texttt{e} = \iota\texttt{.f} = \texttt{e}''$ by def RC-ASSIGN
  2. $\texttt{e}' = \iota\texttt{.f} = \texttt{e}'''$ by def RC-ASSIGN
  3. $\texttt{e}''\texttt{;}\mathcal{H}\rightsquigarrow \texttt{e}'''\texttt{;}\mathcal{H}'$ by premise RC-ASSIGN
  4. $\texttt{e}''' \neq \texttt{err}$ by premise RC-ASSIGN
  5. $\Delta\mathcal{H} \vdash \iota\texttt{:N}$ by 1, a, INVERSION-LEMMA (FIELD ASSIGNMENT)
  6. $\texttt{fType(f, N) = U}$ by 1, a, INVERSIO-LEMMA (FIELD ASSIGNMENT)
  7. $\Delta;\mathcal{H} \vdash \texttt{e}''\texttt{:U}$ by 1, a, INVERSION-LEMMA (FIELD ASSIGNMENT)
  8. $\Delta;\mathcal{H} \vdash \texttt{U<:T}$ by 1, a, INVERSION-LEMMA (FIELD ASSIGNMENT)
  9. $\Delta;\mathcal{H} \vdash$e” OKby c, 1, def F-CONFIG
  10. $\Delta;\mathcal{H}' \vdash \texttt{e}'''\texttt{:U}$ by 7, 3, 9, d, 4, induction hyp
  11. $\Delta;\mathcal{H}' \vdash$e''' OKby 7, 3, 9, d, 4, induction hyp
  12. $\Delta;\mathcal{H}' \vdash \gamma\texttt{.f=e}'''\texttt{:U}$ by 5, 6, 10, T-ASSIGN
  13. $\Delta \vdash \mathcal{H}'$ OKby 11, def F-CONFIG
  14. $\texttt{fv(e}'''\texttt{)}\subseteq\texttt{dom}(\mathcal{H}')$ by 11, def F-CONFIG
  15. $\Delta;\mathcal{H}' \vdash \texttt{T}$ OKby a, d, 13, RUNTIME-TYPE-CHECKING-GIVES-WELL-FORMED-TYPES-LEMMA
  16. $\Delta;\mathcal{H}' \vdash \gamma\texttt{.f=e}'''\texttt{:T}$ by 12, 8, 15, T-SUB
  17. $\Delta;\mathcal{H} \vdash \texttt{e}'\texttt{:T}$ by 16, 2
  18. $\texttt{fv(e}'\texttt{)} \subseteq \texttt{dom}(\mathcal{H}')$ by c, 1, 2, 14
  19. $\Delta;\mathcal{H}' \vdash \texttt{e}'$ OKby 13, 18, F-CONFIG
  20. done by 17, 19
- RC-INVK
  1. $\texttt{e} = \iota\texttt{.<}\overline{\texttt{r}}\texttt{>m(}\overline{\texttt{v}}\texttt{, e}_i\texttt{, }\overline{\texttt{e}}\texttt{)}$ by def RC-INVK
  2. $\texttt{e}' = \iota\texttt{.<}\overline{\texttt{r}}\texttt{>m(}\overline{\texttt{v}}\texttt{, e}'_i\texttt{, }\overline{\texttt{e}}\texttt{)}$ by def RC-INVK
  3. $\texttt{e}_i\texttt{;}\mathcal{H}\rightsquigarrow \texttt{e}_i\texttt{';}\mathcal{H}'$ by premise RC-INVK
  4. $\texttt{e}'_i \neq \texttt{err}$ by premise RC-INVK
  5. $\Delta;\mathcal{H} \vdash \iota\texttt{:N}$ by 1, a, INVERSION-LEMMA (INVOKE)
  6. $\texttt{mType(m, N)=(}\overline{\texttt{U}}\rightarrow\texttt{U;n)}$ by 1, a, INVERSION-LEMMA (INVOKE)
  7. $\Delta;\mathcal{H} \vdash \texttt{(}\overline{\texttt{v}}\texttt{,e}_i\texttt{,}\overline{\texttt{e}}\texttt{:}\overline{\texttt{U}}\texttt{)}$ by 1, a, INVERSION-LEMMA (INVOKE)
  8. $\Delta;\mathcal{H} \vdash \overline{\texttt{r}}$ OKby 1, a, INVERSION-LEMMA (INVOKE)
  9. $\Delta;\mathcal{H} \vdash \overline{\texttt{T}}$ OKby 1, a, INVERSION-LEMMA (INVOKE)
  10. $\Delta;\mathcal{H} \vdash \texttt{U<:T}$ by 1, a, INVERSION-LEMMA (INVOKE)
  11. $\Delta;\mathcal{H} \vdash \texttt{e}_i$ OKby d, 1, def F-CONFIG
  12. $\Delta;\mathcal{H}' \vdash \texttt{e}'_i\texttt{:U}_i$ by 7, 3, 11, d, 4, induction hyp
  13. $\Delta;\mathcal{H}' \vdash \texttt{e}'_i$ OKby 7, 3, 11, d, 4, induction hyp
  14. $\Delta;\mathcal{H}' \vdash \iota\texttt{.m<}\overline{\texttt{r}}\texttt{>(}\overline{\texttt{v}}\texttt{, e}'_i\texttt{, }\overline{\texttt{e}}\texttt{):U}$ by 5, 6, 7, 12, 8, 9, T-INVK
  15. $\Delta;\mathcal{H}' \vdash \texttt{e}'\texttt{:U}$ by 14, 2
  16. $\Delta \vdash \mathcal{H}'$ OKby 13, def F-CONFIG
  17. $\texttt{fv(e}'_i\texttt{)} \subseteq \texttt{dom}(\mathcal{H}')$ by 13, def F-CONFIG
  18. $\Delta;\mathcal{H}' \vdash \texttt{T}$ OKby a, d, 16, RUNTIME-TYPE-CHECKING-GIVES-WELL-FORMED-TYPES-LEMMA
  19. $\Delta;\mathcal{H}' \vdash \texttt{e}'\texttt{:T}$ by 15, 10, 18, T-SUB
  20. $\texttt{fv(e}'\texttt{)} \subseteq \texttt{dom}(\mathcal{H}')$ by c, 1, 2, 17
  21. $\Delta;\mathcal{H}' \vdash \texttt{e}'$ OKby 16, 20, F-CONFIG
  22. done by 19, 21

The required lemmas in the proof are as follows:

- INVERSION-LEMMA (FIELD ACCESS):
  - If $\Delta;\Gamma \vdash \gamma\texttt{.f:T}$ then
  - $\Delta;\Gamma \vdash \gamma\texttt{:N}$ and
  - $\Delta;\Gamma \vdash \texttt{fType(f, N)<:T}$
- INVERSION-LEMMA (ADDRESS):
  - If $\Delta;\Gamma \vdash \iota\texttt{:T}$ then
  - $\Delta;\Gamma \vdash \Gamma(\iota)\texttt{<:T}$
- INVERSION-LEMMA (FIELD ASSIGNMENT)
  - If $\Delta;\Gamma \vdash \gamma\texttt{.f = e:T}$ then
  - $\Delta;\Gamma \vdash \gamma\texttt{:N}$
  - $fType(\texttt{f, N}) = \texttt{U}$
  - $\Delta;\Gamma \vdash \texttt{e:U}$
  - $\Delta;\Gamma \vdash \texttt{U<:T}$
- RUNTIME-TYPE-CHECKING-GIVES-WELL-FORMED-TYPES-LEMMA
  - If $\Delta;\mathcal{H} \vdash \texttt{e:T}$ and

- $\Delta \vdash \mathcal{H}$ OKand
- $\emptyset \vdash \Delta$ OKthen
- $\Delta; \mathcal{H} \vdash \mathtt{T}$ OK

- REDUCTION-PRESERVES-HEAP-LEMMA
  - if $\mathtt{e};\mathcal{H} \leadsto \mathtt{e}';\mathcal{H}'$ and
  - ...$\mathcal{H}$... $\vdash$ ... and
  - $\mathtt{e}' \neq \mathtt{err}$ then
  - ...$\mathcal{H}'$... $\vdash$ ...
- WELL-TYPED-VALUES-HAVE-ADDRESSES-LEMMA
  - If $\Delta; \Gamma \vdash \mathtt{v}:\mathtt{T}$ and
  - $\mathtt{v} \neq \mathtt{null}$, then
  - $\mathtt{v} \in \mathtt{dom}(\Gamma)$
- INVERSION-LEMMA (NEW)
  - If $\Delta; \Gamma \vdash \mathtt{new\ C<\overline{a}>:T}$ then
  - $\Delta; \Gamma \vdash \mathtt{C<\overline{a}><:T}$ and
  - $\Delta; \Gamma \vdash \mathtt{C<\overline{a}>}$ OK
- SUBTYPE-WEAKENING-LEMMA
  - If $\Delta, \Delta'; \Gamma, \Gamma' \vdash T <: T'$ and
  - $\mathtt{dom}(\Delta, \Delta') \cap \mathtt{dom}(\Delta'') = \emptyset$ and
  - $\mathtt{dom}(\Gamma, \Gamma') \cap \mathtt{dom}(\Gamma'') = \emptyset$ then
  - $\Delta, \Delta'', \Delta'; \Gamma, \Gamma'', \Gamma \vdash T <: T'$
- FTYPE-WELL-FORMED-LEMMA
  - If $\mathtt{fType(f,\ C<\overline{a}>)\ =\ T}$ and
  - $\Delta; \mathcal{H} \vdash \mathtt{C<\overline{a}>}$ OKand
  - $\Delta; \mathcal{H} \vdash \iota:\mathtt{C<\overline{a}>}$ and
  - $\emptyset; \vdash \Delta$ OKand
  - $\Delta \vdash \mathcal{H}$ OKthen
  - $\Delta; \mathcal{H} \vdash \mathtt{T}$ OK
- INVERSION-LEMMA (INVOKE)
  - If $\Delta; \Gamma \vdash \gamma.\mathtt{<\overline{a}>m(\overline{e}):T}$ then
  - $\Delta; \Gamma \vdash \gamma:\mathtt{N}$ and
  - $\mathtt{mType(m,N)=(\overline{T} \rightarrow T;n)}$ and
  - $\Delta; \Gamma \vdash \overline{\mathtt{e:T}}$ and
  - $\Delta; \Gamma \vdash \overline{\mathtt{a}}$ OKand
  - $\Delta; \Gamma \vdash \overline{\mathtt{U}}$ OKand
  - $\Delta; \Gamma \vdash \mathtt{T}'\mathtt{<:T}$
- BODY-HAS-RETURN-TYPE-LEMMA
  - If $\mathtt{mBody(m,\ R)\ =\ (\overline{x};e)}$ and
  - $\mathtt{mType(m,\ R)\ =\ (\overline{T} \rightarrow T;n)}$ and
  - $\Delta \vdash \mathcal{H}$ OKand
  - $\mathcal{H} \vdash \Delta$ OKthen
  - $\Delta;\mathcal{H},\overline{\mathtt{x:T}} \vdash \mathtt{e:T}$
- VALUE-SUBSTITUTION-PRESERVES-TYPING-LEMMA
  - If $\Delta, \Delta'; \Gamma, \mathtt{x:U}, \Gamma' \vdash \mathtt{e:T}$ and
  - $\Delta; \Gamma \vdash \mathtt{v:U}'$ and
  - $\Delta; \Gamma \vdash \mathtt{U}'\mathtt{<:U}$ and
  - $\Delta \vdash \Gamma$ OKand
  - $\mathtt{x} \notin \mathtt{fv}(\Delta)$ then
  - $\Delta, \mathtt{[v/x]}\Delta'; \Gamma, \mathtt{[v/x]}\Gamma' \vdash \mathtt{[v/x]e:[v/x]T}$