# Featherweight Ownership and Immutability Generic Java - Technical Report

Yoav Zibin yoav.zibin@gmail.com

## 1 Introduction

This technical report contains proofs that were omitted from our paper entitled "Ownership and Immutability in Generic Java". Please read the paper first, and only then proceed to reading this technical report, because this technical report is not self contained. We only include a summary of the syntax (Fig. 1), subtyping rules (Fig. 2), expression typing rules (Fig. 3), and reduction rules (Fig. 4).

We begin with some definitions. A *rule* has the form $\frac{A}{B}$, where $A$ is the *assumption* and $B$ is the *conclusion*. If $A$ is empty, we also call the rule an *axiom*. An *instance* of a rule/assumption/conclusion is any substitution of variables in the rule. A *derivation sequence* is a sequence of elements (each element is an instance of a conclusion), where the assumptions needed for each element appear as previous elements in the sequence.

An expression/type is called *closed* if it does not contain any free variables (such as wildcards, `this`, `I`, `O`, or `This`).

To define *ftype*`(e,f,T)` and *mtype*`(e,f,T)`, we use the auxiliary function *substitute*:

$$\mathit{substitute}(\texttt{e},\texttt{C<MO,IP>},\texttt{T}) = \begin{cases} \texttt{error} & O(\texttt{T}) = \texttt{This} \text{ and } \texttt{z} = \texttt{error} \\ [\texttt{z}/\texttt{This},\texttt{MO}/\texttt{O},\texttt{IP}/\texttt{I}]\texttt{T} & \text{otherwise} \end{cases} \qquad \texttt{z} = \begin{cases} \texttt{l} & \texttt{e} = \texttt{l} \\ \texttt{This} & \texttt{e} = \texttt{this} \\ \texttt{error} & \text{otherwise} \end{cases}$$

Formally, *ftype*`(e,f,C<MO,IP>)` $=$ *substitute*$(\texttt{e},\texttt{C<MO,IP>},\mathit{ftype}(\texttt{f},\texttt{C}))$, and similarly for *mtype*.

Given an expression `e`, we define $K(\texttt{e})$ to be the set of all ongoing constructors in `e`, i.e., all locations in subexpressions `e;return l`. Formally,

$$K(\texttt{e}) = \begin{cases} K(\texttt{e}') \cup \{l\} & \text{if } \texttt{e} = (\texttt{e}'\texttt{;return l}) \\ K(\texttt{e}') & \text{if } \texttt{e} = (\texttt{e}'\texttt{.f}) \\ K(\texttt{e}') \cup K(\texttt{e"}) & \text{if } \texttt{e} = (\texttt{e}'\texttt{.f=e"}) \\ \cup K(\overline{\texttt{e}'}) & \text{if } \texttt{e} = (\texttt{new N}(\overline{\texttt{e}'})) \\ K(\texttt{e"}) \cup K(\overline{\texttt{e}'}) & \text{if } \texttt{e} = (\texttt{e".m}(\overline{\texttt{e}'})) \end{cases}$$

A *well-typed* heap $H$ satisfies: (i) there is a linear order $\preceq^T$ over $\mathrm{dom}(H)$ such that for every location `l`, $\theta(\texttt{l}) = \texttt{World}$ or $\theta(\texttt{l}) \prec^T \texttt{l}$, and $I(\texttt{l}) = \texttt{Mutable}$ or $\kappa(\texttt{l}) \preceq^T \texttt{l}$, and (ii) each non-null field location is a subtype of the declared field type. A heap $H$ is *well-typed for* `e` if $H[K \mapsto H[K] \cup K(\texttt{e})]$ is well-typed.

## 2 Subtyping

First we prove some lemmas regarding subtyping.

**Lemma 2.1.** *If* $\Gamma \vdash \texttt{C<MO,IP>} \leq \texttt{C'<MO',IP'>}$, *then*
*(i)* $\texttt{MO'} \neq \texttt{?} \Rightarrow \texttt{MO} = \texttt{MO'}$,
*(ii)* $(\texttt{IP'} \neq \texttt{Immut}_{\texttt{l}}$ *or* $(\texttt{IP'} = \texttt{Immut}_{\texttt{l}}$ *and* $(\texttt{l} \not\prec_\theta \texttt{MO'}$ *or* $\texttt{l} \notin \Gamma[K]))) \Rightarrow \Gamma \vdash \texttt{IP} \leq \texttt{IP'}$,
*(iii)* `C` *is a subclass of* `C'`,
*(iv)* $\Gamma \vdash \texttt{D<MO,IP>} \leq \texttt{D<MO',IP'>}$ *for any class* `D`,
*(v)* $\Gamma \vdash \texttt{D<l,IP>} \leq \texttt{D<l,IP'>}$ *for any class* `D` *and* $\texttt{MO'} \preceq_\theta \texttt{l}$.

| | |
|---|---|
| FT ::= C<FO, IP> | Field (and method return) Type. |
| T ::= C<MO, IP> | Type. |
| N ::= C<NO, NI> | Non-variable type (for objects). |
| NO ::= World \| l | Non-variable Owner parameter (for objects). |
| FO ::= NO \| This \| O | Field Owner parameter. |
| MO ::= FO \| ? | Method Owner parameter (including generic wildcard). |
| NI ::= Mutable \| $\texttt{Immut}_{\texttt{l}}$ | Non-variable Immutability parameter (for objects). |
| VI ::= NI \| Immut \| I | Variable Immutability for new. |
| IP ::= ReadOnly \| VI | Immutability Parameter. |
| IG ::= ReadOnly \| Immut \| Mutable \| Raw | Immutability method Guard. |
| M ::= <I extends IG>? FT m($\overline{\texttt{T}}$ $\overline{\texttt{x}}$) { return e; } | Method declaration. |
| L ::= class C<O,I> extends C′<O,I>{ $\overline{\texttt{FT}}$ $\overline{\texttt{f}}$;$\overline{\texttt{M}}$ } | cLass declaration. |
| v ::= null \| l | Values: either null or a location l. |
| e ::= v \| x \| e.f \| e.f = e \| e.m($\overline{\texttt{e}}$) \| new C<FO,VI>($\overline{\texttt{e}}$) \| e;return l | Expressions. |

Figure 1: FOIGJ Syntax. The terminals are null, owner parameters (O, This, World), immutability parameters (I, ReadOnly, Mutable, Raw, Immut). Given a location l, $\texttt{Immut}_{\texttt{l}}$ represents an immutable object with cooker l.

$$\frac{}{\Gamma \vdash \texttt{I} \leq \Gamma(\texttt{I})}\ (\text{S1}) \qquad \frac{}{\Gamma \vdash \texttt{T} \leq \texttt{T}}\ (\text{S2}) \qquad \frac{\Gamma \vdash \texttt{S} \leq \texttt{T} \quad \Gamma \vdash \texttt{T} \leq \texttt{U}}{\Gamma \vdash \texttt{S} \leq \texttt{U}}\ (\text{S3}) \qquad \frac{\texttt{class C<O,I> extends C}'\texttt{<O,I>}}{\Gamma \vdash \texttt{C<MO, IP>} \leq \texttt{C}'\texttt{<MO, IP>}}\ (\text{S4})$$

$$\frac{}{\Gamma \vdash \texttt{Mutable} \leq \texttt{Raw}}\ (\text{S5}) \qquad \frac{}{\Gamma \vdash \texttt{Raw} \leq \texttt{ReadOnly}}\ (\text{S6}) \qquad \frac{}{\Gamma \vdash \texttt{Immut} \leq \texttt{ReadOnly}}\ (\text{S7})$$

$$\frac{\Gamma \vdash \texttt{IP} \leq \texttt{IP}'}{\Gamma \vdash \texttt{C<MO, IP>} \leq \texttt{C<MO, IP}'\texttt{>}}\ (\text{S8}) \qquad \frac{}{\Gamma \vdash \texttt{C<MO, IP>} \leq \texttt{C<?, IP>}}\ (\text{S9}) \qquad \frac{\texttt{l} \in \Gamma[K]}{\Gamma \vdash \texttt{Immut}_{\texttt{l}} \leq \texttt{Raw}}\ (\text{S10})$$

$$\frac{\texttt{l} \notin \Gamma[K]}{\Gamma \vdash \texttt{Immut}_{\texttt{l}} \leq \texttt{Immut}}\ (\text{S11}) \qquad \frac{\texttt{l} \notin \Gamma[K]}{\Gamma \vdash \texttt{Immut} \leq \texttt{Immut}_{\texttt{l}}}\ (\text{S12}) \qquad \frac{\texttt{l} \prec_\theta \texttt{NO}}{\Gamma \vdash \texttt{C<NO, Immut>} \leq \texttt{C<NO, Immut}_{\texttt{l}}\texttt{>}}\ (\text{S13})$$

Figure 2: FOIGJ Subtyping Rules. Rule S13 shows the connection between cooker l and owner NO.

*Proof.* In this proof we omit $\Gamma \vdash$ because the context $\Gamma$ is clear. First note that due to S13, it is not the case that $\texttt{IP} \leq \texttt{IP}'$. Therefore, we need parts (ii) and (iv)–(v), which connects IP and IP′ in other ways.

(i) All subtyping rules maintain the same owner parameter, except S9, thus if $\texttt{MO}' \neq \texttt{?}$, then the owner must be preserved.

(ii) All subtyping rules maintain that $\texttt{IP} \leq \texttt{IP}'$ except S13. If $\texttt{l} \not\prec_\theta \texttt{MO}'$ then we cannot use S13. If $\texttt{l} \notin \Gamma[K]$ then (from S12) $\texttt{Immut} \leq \texttt{IP}'$, and if the proof used S13 then $\texttt{C<MO, IP>} \leq \texttt{C<MO, Immut>}$, thus $\texttt{IP} \leq \texttt{Immut} \leq \texttt{IP}'$.

(iii) All rules maintain the same class, and S4 permits subclassing.

(iv) We create a new derivation sequence where instances of rule S4 are deleted, and all occurrences of C are replaced with D.

(v) Similarly to part (iv), we delete S4 and S9, replace C with D, and replace MO and MO′ with l. The only rule where the owner matters is S13:

$$\frac{\texttt{l}' \prec_\theta \texttt{MO}'}{\Gamma \vdash \texttt{C<MO}', \texttt{Immut>} \leq \texttt{C<MO}', \texttt{Immut}_{\texttt{l}'}\texttt{>}}$$

and we have that $\texttt{MO}' \preceq_\theta \texttt{l}$, therefore $\texttt{l}' \prec_\theta \texttt{l}$. □

**Lemma 2.2.** *If* $\Gamma \vdash \texttt{C<MO, IP>} \leq \texttt{C}'\texttt{<MO, IP}'\texttt{>}$*, then for any class* D *and any owner parameter* W *such that* $\texttt{W} = \texttt{O} \mid \texttt{World}$*, we have that*

$$\Gamma \vdash \texttt{D<}[\texttt{MO/O}]\texttt{W}, [\texttt{IP/I}]\texttt{Z>} \leq \texttt{D<}[\texttt{MO/O}]\texttt{W}, [\texttt{IP}'\texttt{/I}]\texttt{Z>}.$$

*Proof.* If $\texttt{Z} \neq \texttt{I}$ then obviously $\texttt{D<}[\texttt{MO/O}]\texttt{W}, \texttt{Z>} = \texttt{D<}[\texttt{MO/O}]\texttt{W}, \texttt{Z>}$. If $\texttt{Z} = \texttt{I}$ then we need to prove that $\Gamma \vdash \texttt{D<}[\texttt{MO/O}]\texttt{W}, \texttt{IP>} \leq \texttt{D<}[\texttt{MO/O}]\texttt{W}, \texttt{IP}'\texttt{>}$. Because $\texttt{MO} \preceq_\theta \texttt{World}$ and $\texttt{W} = \texttt{O} \mid \texttt{World}$, then $\texttt{MO} \preceq_\theta [\texttt{MO/O}]\texttt{W}$. Therefore, we can apply Lem. 2.1 part (v). □

**Lemma 2.3.** *If* $\Gamma \vdash \texttt{C<MO, IP>} \leq \texttt{C}'\texttt{<MO, IP}'\texttt{>}$*, both types are closed, isTransitive*$(\bot, \Gamma, \texttt{C}'\texttt{<MO, IP}'\texttt{>})$ *and* $\texttt{IP}' \leq \texttt{Raw}$*, then* $\texttt{IP} = \texttt{IP}'$.

$$\frac{\Gamma[K \mapsto \Gamma[K] \cup \{\mathtt{l}\}] \vdash \mathtt{e} : \mathtt{T}}{\Gamma \vdash \mathtt{e;return\ l} : \Gamma(\mathtt{l})}\ \text{(T-RETURN)} \qquad \frac{\mathit{mtype}(\bot, \mathtt{build}, \mathtt{C<FO,VI>}) = \overline{\mathtt{T}} \to \mathtt{U} \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T'}} \quad \Gamma \vdash \overline{\mathtt{T'}} \le \overline{\mathtt{T}}}{\Gamma \vdash \mathtt{new\ C<FO,VI>}(\overline{\mathtt{e}}) : \mathtt{C<FO,VI>}}\ \text{(T-NEW)}$$

$$\frac{}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})}\ \text{(T-VAR)} \qquad \frac{}{\Gamma \vdash \mathtt{null} : \mathtt{T}}\ \text{(T-NULL)} \qquad \frac{\Gamma \vdash \mathtt{e} : \mathtt{C<MO,IP>} \quad \mathit{ftype}(\mathtt{e}, \mathtt{f}, \mathtt{C<MO,IP>}) = \mathtt{T}}{\Gamma \vdash \mathtt{e.f} : \mathtt{T}}\ \text{(T-FIELD-ACCESS)}$$

$$\frac{}{\Gamma \vdash \mathtt{l} : \Gamma(\mathtt{l})}\ \text{(T-LOCATION)} \qquad \frac{\begin{array}{c}\Gamma \vdash \mathtt{e.f} : \mathtt{T} \quad \Gamma \vdash \mathtt{e'} : \mathtt{T'} \quad \Gamma \vdash \mathtt{T'} \le \mathtt{T} \quad \Gamma \vdash \mathtt{e} : \mathtt{C<MO,IP>} \\ \Gamma \vdash \mathtt{IP} \le \mathtt{Raw} \quad \mathit{isTransitive}(\mathtt{e}, \Gamma, \mathtt{C<MO,IP>}) \quad \mathtt{MO} \neq \mathtt{?}\end{array}}{\Gamma \vdash \mathtt{e.f} = \mathtt{e'} : \mathtt{T'}}\ \text{(T-FIELD-ASSIGNMENT)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e_0} : \mathtt{C<MO,IP>} \quad \mathit{mtype}(\mathtt{e_0}, \mathtt{m}, \mathtt{C<MO,IP>}) = \overline{\mathtt{T}} \to \mathtt{T''} \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T'}} \quad \Gamma \vdash \overline{\mathtt{T'}} \le \overline{\mathtt{T}} \quad \mathit{mguard}(\mathtt{m}, \mathtt{C}) = \mathtt{IG} \\ \Gamma \vdash \mathtt{IP} \le \mathtt{IG} \quad \mathtt{IG} = \mathtt{Raw} \Rightarrow \mathit{isTransitive}(\mathtt{e_0}, \Gamma, \mathtt{C<MO,IP>}) \quad \mathit{mtype}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{U}} \to \mathtt{V} \quad O(\overline{\mathtt{T}}) = \mathtt{?} \Rightarrow O(\overline{\mathtt{U}}) = \mathtt{?}\end{array}}{\Gamma \vdash \mathtt{e_0.m}(\overline{\mathtt{e}}) : \mathtt{T''}}\ \text{(T-INVOKE)}$$

Figure 3: FOIGJ Expression Typing Rules.

$$\frac{\mathtt{l} \notin \mathrm{dom}(H) \qquad \mathtt{VI'} = \begin{cases} \mathtt{Immut_l} & \text{if } \mathtt{VI} = \mathtt{Immut} \text{ or } (\mathtt{VI} = \mathtt{Immut_C} \text{ and } \mathtt{c} \notin H[K]) \\ \mathtt{VI} & \text{otherwise} \end{cases}}{H, \mathtt{new\ C<NO,VI>}(\overline{\mathtt{v}}) \to H\big[\mathtt{l} \mapsto \mathtt{C<NO,VI'>}(\overline{\mathtt{null}})\big], \mathtt{l.build}(\overline{\mathtt{v}})\mathtt{;return\ l}}\ \text{(R-NEW)}$$

$$\frac{H[K \mapsto H[K] \cup \{\mathtt{l}\}], \mathtt{e} \to H', \mathtt{e'}}{H, \mathtt{e;return\ l} \to H'[K \mapsto H'[K] \setminus \{\mathtt{l}\}], \mathtt{e'};\mathtt{return\ l}}\ \text{(R-C1)} \qquad \frac{H[\mathtt{l}] = \mathtt{C<NO,NI>}(\overline{\mathtt{v}}) \quad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{f}}}{H, \mathtt{l.f}_i \to H, \mathtt{v}_i}\ \text{(R-FIELD-ACCESS)}$$

$$\frac{H[\mathtt{l}] = \mathtt{C<NO,NI>}(\overline{\mathtt{v}}) \quad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{f}} \quad \mathtt{NI} = \mathtt{Mutable} \text{ or } \kappa(\mathtt{l}) \in H[K] \quad \mathtt{v'} = \mathtt{null} \text{ or } \mathtt{l} \preceq_\theta \theta(\mathtt{v'})}{H, \mathtt{l.f}_i = \mathtt{v'} \to H\big[\mathtt{l} \mapsto \mathtt{C<NO,NI>}([\mathtt{v'}/\mathtt{v}_i]\overline{\mathtt{v}})\big], \mathtt{v'}}\ \text{(R-FIELD-ASSIGNMENT)}$$

$$\frac{}{H, \mathtt{v;return\ l} \to H, \mathtt{l}}\ \text{(R-RETURN)} \qquad \frac{H[\mathtt{l}] = \mathtt{C<NO,NI>}(\ldots) \quad \mathit{mbody}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e'}}{H, \mathtt{l.m}(\overline{\mathtt{v}}) \to H, [\overline{\mathtt{v}}/\overline{\mathtt{x}}, \mathtt{l}/\mathtt{this}, \mathtt{l}/\mathtt{This}, \mathtt{NO}/\mathtt{O}, \mathtt{NI}/\mathtt{I}]\mathtt{e'}}\ \text{(R-INVOKE)}$$

Figure 4: FOIGJ Reduction Rules (excluding all congruence rules except the one for `e;return l` described in R-C1).

*Proof.* If $\mathtt{IP'} = \mathtt{Mutable}$ then obviously $\mathtt{IP} = \mathtt{Mutable}$. Otherwise $\mathtt{IP'} = \mathtt{Immut_l}$ and $\mathtt{l} \in \Gamma[K]$ (because $\mathtt{IP'} \le \mathtt{Raw}$). From definition of $\mathit{isTransitive}(\bot, \Gamma, \mathtt{C'<MO,IP'>})$, $\mathtt{l} \not\preceq_\theta \mathtt{MO}$, thus S13 cannot be applied, and that is the only applicable rule where $\mathtt{Immut_l}$ appears as a supertype (because S12 cannot be applied since $\mathtt{l} \in \Gamma[K]$), thus $\mathtt{IP} = \mathtt{IP'}$. □

The next lemma shows that if $\mathtt{e'}$ was reduced to `e` (therefore the type of `e` is a subtype of $\mathtt{e'}$), then `e` can call any method that $\mathtt{e'}$ could. Phrased differently, if $\mathtt{e'}$ satisfies a method's guard then `e` would as well.

**Lemma 2.4.** *If* $\Gamma \vdash \mathtt{C<MO,IP>} \le \mathtt{C'<MO,IP'>}$*, and both types are closed, then*
*(i)* $\Gamma \vdash \mathtt{IP'} \le \mathtt{Mutable} \Rightarrow \Gamma \vdash \mathtt{IP} \le \mathtt{Mutable}$,
*(ii)* $\Gamma \vdash \mathtt{IP'} \le \mathtt{Immut} \Rightarrow \Gamma \vdash \mathtt{IP} \le \mathtt{Immut}$,
*(iii)* $\mathit{isTransitive}(\bot, \Gamma, \mathtt{C'<MO,IP'>})$ *and* $\Gamma \vdash \mathtt{IP'} \le \mathtt{Raw} \Rightarrow \Gamma \vdash \mathtt{IP} \le \mathtt{Raw}$,
*(iv) if* $\mathtt{IG} = \mathtt{Raw} \Rightarrow \mathit{isTransitive}(\bot, \Gamma, \mathtt{C'<MO,IP'>})$*, where* $\mathtt{IG} = \mathtt{ReadOnly} \mid \mathtt{Immut} \mid \mathtt{Mutable} \mid \mathtt{Raw}$*, then* $\Gamma \vdash \mathtt{IP'} \le \mathtt{IG} \Rightarrow \Gamma \vdash \mathtt{IP} \le \mathtt{IG}$.

*Proof.* (i) Trivial because we must have that $\mathtt{IP'} = \mathtt{IP} = \mathtt{Mutable}$ or ($\mathtt{IP'} = \mathtt{IP} = \mathtt{I}$ and $\mathtt{I} : \mathtt{Mutable} \in \Gamma$).

(ii) If $\mathtt{IP'} \neq \mathtt{Immut_l}$ then from Lem. 2.1 part (ii), we have $\Gamma \vdash \mathtt{IP} \le \mathtt{IP'}$, and from transitivity $\mathtt{IP} \le \mathtt{IP'} \le \mathtt{Immut} \Rightarrow \mathtt{IP} \le \mathtt{Immut}$. Otherwise, $\mathtt{IP'} = \mathtt{Immut_l}$, and because $\mathtt{IP'} \le \mathtt{Immut}$, from S11 we must have that $\mathtt{l} \notin \Gamma[K]$, and from Lem. 2.1 part (ii), we proved $\Gamma \vdash \mathtt{IP} \le \mathtt{IP'}$,

(iii) From Lem. 2.3 we know that $\mathtt{IP} = \mathtt{IP'}$ thus $\mathtt{IP} \le \mathtt{Raw}$.

(iv) Stems from parts (vi)–(viii) and the fact that for any $\mathtt{IP} \le \mathtt{ReadOnly}$. □

If the cooker is not inside the owner, then subtypes are not over-approximation (i.e., they preserve the same cooker).

**Lemma 2.5.** *If* $\Gamma \vdash \texttt{C<MO,IP>} \leq \texttt{C'<NO,Immut}_{\texttt{l}}\texttt{>}$, $\texttt{l} \not\leq_\theta \texttt{NO}$, *and* $\texttt{l} \in \Gamma[K]$, *then* $\texttt{IP} = \texttt{Immut}_{\texttt{l}}$.

*Proof.* The only subtyping rule where the supertype has a cooker $\texttt{Immut}_{\texttt{l}}$ are rules S12 and S13, and they can't be applied because we assumed that $\texttt{l} \not\leq_\theta \texttt{NO}$ and $\texttt{l} \notin \Gamma[K]$. Thus only the reflexivity rule can be applied. Note that rule $\Gamma \vdash \texttt{I} \leq \Gamma(\texttt{I})$ cannot be applied because we assume that $\Gamma(\texttt{I})$ is a method guard IG, and $\texttt{IG} \neq \texttt{Immut}_{\texttt{l}}$ because according to our syntax

$$\texttt{IG} = \texttt{ReadOnly} \mid \texttt{Immut} \mid \texttt{Mutable} \mid \texttt{Raw}.$$

□

Next we prove a substitution lemma: substituting I, O, or This, does not change the subtype relation.

**Lemma 2.6.** *If* $\Gamma \vdash \texttt{T} \leq \texttt{T}'$ *then for every* IP,MO,MO′ *such that* $\texttt{IP} \leq \Gamma(\texttt{I})$, *we have that*

$$\Gamma \vdash [\texttt{IP}/\texttt{I},\texttt{MO}/\texttt{O},\texttt{MO}'/\texttt{This}](\texttt{T} \leq \texttt{T}').$$

*Proof.* Let $S$ denote the derivation sequence for $\Gamma \vdash \texttt{T} \leq \texttt{T}'$, and $S_I$ for $\texttt{IP} \leq \Gamma(\texttt{I})$. Let $S'$ be a new sequence in which we do the substitution $[\texttt{IP}/\texttt{I},\texttt{MO}/\texttt{O},\texttt{MO}'/\texttt{This}]$ on every element in $S$, and let $S$" be the sequence starting with $S_I$ followed by $S'$. The last element in $S$ is $\Gamma \vdash \texttt{T} \leq \texttt{T}'$, therefore the last element in $S'$ and $S$" is what we need to prove: $\Gamma \vdash [\texttt{IP}/\texttt{I},\texttt{MO}/\texttt{O},\texttt{MO}'/\texttt{This}](\texttt{T} \leq \texttt{T}')$. Next we show that $S$" is a legal derivation sequence by showing that each element is a legal consequence of previous elements (by induction on the size of $S$"). Elements from $S_I$ are of course legal. By induction we proved the first $n-1$ elements are legal, and we now prove that element $n$ is legal (i.e., a legal consequence of previous elements). Let the corresponding element in $S$ be $\texttt{U} \leq \texttt{U}'$, and element $n$ is $[\texttt{IP}/\texttt{I},\texttt{MO}/\texttt{O},\texttt{MO}'/\texttt{This}](\texttt{U} \leq \texttt{U}')$. (i) If the element is an instance of rules S5–S7 or S10–S12, then substitution did not change the element. (ii) If the element is an instance of rule S1 then we replaced it with $[\texttt{IP}/\texttt{I}](\texttt{I} \leq \Gamma(\texttt{I})) = \texttt{IP} \leq \Gamma(\texttt{I})$, which is the last element of $S_I$. (iii) If the element is an instance of any other rule, then a simple application of the induction hypothesis proves the element is legal. □

We now prove that substitution preserves subtyping.

**Lemma 2.7.** *If* $\Gamma \vdash \texttt{T} \leq \texttt{T}'$, T *and* T′ *are closed, then for (*$\texttt{e} = \bot$ *and* $O(\texttt{T "}) \neq \texttt{This}$*) or* $\texttt{e} = \texttt{l}$, $O(\texttt{T}) \preceq_\theta \texttt{l}$, *we have that:*
*(i)* $\Gamma \vdash$ *substitute*$(\texttt{e},\texttt{T},\texttt{T"}) \leq$ *substitute*$(\texttt{e},\texttt{T}',\texttt{T"})$,
*(ii)* $\Gamma \vdash$ *ftype*$(\texttt{e},\texttt{f},\texttt{T}) \leq$ *ftype*$(\texttt{e},\texttt{f},\texttt{T}')$,
*(iii) let* *mtype*$(\texttt{e},\texttt{f},\texttt{T}) = \overline{\texttt{T}} \to \texttt{T}_0$ *and* *ftype*$(\texttt{e},\texttt{f},\texttt{T}') = \overline{\texttt{T}'} \to \texttt{T}'_0$, *then* $\Gamma \vdash \texttt{T}_i \leq \texttt{T}'_i$ *for* $i = 0,\ldots,\#(\overline{\texttt{T}})$.

*Proof.* From Lem. 2.1 part (i) and the fact that T and T′ are closed (no wildcards), then $O(\texttt{T}) = O(\texttt{T}')$. Let $\texttt{T} = \texttt{C<MO,IP>}$ and $\texttt{T}' = \texttt{C'<MO,IP'>}$.

Recall that *ftype*$(\texttt{e},\texttt{f},\texttt{C<MO,IP>}) =$ *substitute*$(\texttt{e},\texttt{C<MO,IP>},$*ftype*$(\texttt{f},\texttt{C}))$, and similarly for *mtype*. Therefore, parts (ii) and (iii) follow from part (i), and the fact that *ftype*$(\texttt{f},\texttt{C}) =$ *ftype*$(\texttt{f},\texttt{C}')$ and *mtype*$(\texttt{m},\texttt{C}) =$ *mtype*$(\texttt{m},\texttt{C}')$ (i.e., subclassing cannot change method signature or field type).

We now prove part (i). From the definition of *substitute*, and because ($\texttt{e} = \bot$ and $O(\texttt{T "}) \neq \texttt{This}$) or $\texttt{e} = \texttt{l}$, we have that *substitute*$(\texttt{e},\texttt{T},\texttt{T"}) = [\texttt{e}/\texttt{This},\texttt{MO}/\texttt{O},\texttt{IP}/\texttt{I}]\texttt{T"}$.

Let $\texttt{T"} = \texttt{D<MO",IP">}$. Because $\Gamma \vdash \texttt{T} \leq \texttt{T}'$, from Lem. 2.1 part (v),

$$\Gamma \vdash \texttt{D<l',IP>} \leq \texttt{D<l',IP'>} \text{ if } \texttt{MO} \preceq_\theta \texttt{l}' \tag{1}$$

(note that $\texttt{l}'$ can be MO or World). We need to prove that $\Gamma \vdash$ *substitute*$(\texttt{e},\texttt{T},\texttt{T"}) \leq$ *substitute*$(\texttt{e},\texttt{T}',\texttt{T"})$, i.e., $\Gamma \vdash [\texttt{e}/\texttt{This},\texttt{MO}/\texttt{O},\texttt{IP}/\texttt{I}]\texttt{T"} \leq [\texttt{e}/\texttt{This},\texttt{MO}/\texttt{O},\texttt{IP'}/\texttt{I}]\texttt{T"}$.

If $\texttt{IP"} \neq \texttt{I}$ then from reflexivity $\Gamma \vdash [\texttt{e}/\texttt{This},\texttt{MO}/\texttt{O}]\texttt{T"} \leq [\texttt{e}/\texttt{This},\texttt{MO}/\texttt{O}]\texttt{T"}$.

Consider now the case that $\texttt{IP"} = \texttt{I}$. We need to show that

$$\Gamma \vdash \texttt{D<}[\texttt{e}/\texttt{This},\texttt{MO}/\texttt{O}]\texttt{MO",IP>} \leq \texttt{D<}[\texttt{e}/\texttt{This},\texttt{MO}/\texttt{O}]\texttt{MO",IP'>} \tag{2}$$

Because $\texttt{MO"} = \texttt{O} \mid \texttt{World} \mid \texttt{This}$ and $\texttt{e} = \bot$ or $\texttt{e} = \texttt{l}$, we have that

$$\texttt{MO} \preceq_\theta [\texttt{e}/\texttt{This},\texttt{MO}/\texttt{O}]\texttt{MO"} \tag{3}$$

From (1) and (3), we proved (2).

□

**Lemma 2.8.** *If* $\Gamma \vdash$ `C<MO,IP>` $\leq$ `C'<MO,IP'>`*, and both types are closed, and*

$$\begin{array}{l} \mathit{mtype}(\bot, \texttt{m}, \texttt{C'<MO,IP'>}) = \overline{\texttt{T'}} \to \texttt{U} \\ \mathit{mtype}(\bot, \texttt{m}, \texttt{C<MO,IP>}) = \overline{\texttt{T}} \to \texttt{U} \\ \mathit{mguard}(\texttt{m}, \texttt{C'}) = \texttt{IG} \\ \Gamma \vdash \texttt{IP'} \leq \texttt{IG} \\ \texttt{IG} = \texttt{Raw} \Rightarrow \mathit{isTransitive}(\bot, \Gamma, \texttt{C'<MO,IP'>}) \end{array}$$

*then (i)* $\Gamma \vdash \texttt{T}_i \leq \texttt{T'}_i$ *and* $\Gamma \vdash \texttt{T'}_i \leq \texttt{T}_i$*, and (ii) for any type* `T"`*, if* $\Gamma \vdash$ `T"` $\leq \texttt{T'}_i$ *then* $\Gamma \vdash$ `T"` $\leq \texttt{T}_i$*.*

*Proof.* Part (i) implies that $\texttt{T'}_i$ and $\texttt{T}_i$ are equivalent. Part (ii) follows directly from part (i) using transitivity rule S3.

Let $\mathit{mtype}(\texttt{m}, C) = \mathit{mtype}(\texttt{m}, C') = \overline{\texttt{FT}} \to \texttt{V}$, $\texttt{FT}_i = \texttt{D<FO,IP">}$. Note that $\texttt{T'}_i = [\texttt{MO}/\texttt{O}, \texttt{IP'}/\texttt{I}]\texttt{FT}_i$ and $\texttt{T}_i = [\texttt{MO}/\texttt{O}, \texttt{IP}/\texttt{I}]\texttt{FT}_i$. Therefore, if `IP"` $\neq$ `I` then $\texttt{T'}_i = \texttt{T}_i$, qed.

Thus, `IP"` $=$ `I`, $\texttt{T'}_i =$ `D<MO",IP'>`, $\texttt{T}_i =$ `D<MO",IP>`, where `MO` $\preceq_\theta$ `MO"` (because `FO` is either `O` or `World`, but it cannot be `This`). From Lem. 2.7 part (iii), $\texttt{T}_i \leq \texttt{T'}_i$.

If `IG` $=$ `ReadOnly`, then FOIGJ ensures that `I` does not appear in method parameters, i.e., we must have that `IP"` $\neq$ `I`. If `IG` $=$ `Mutable`, then `IP'` $=$ `IP` $=$ `Mutable` (because `IP'` $\leq$ `IG`), thus $\texttt{T'}_i = \texttt{T}_i$,. If `IG` $=$ `Immut`, then `IP'` $\leq$ `Immut` (because `IP'` $\leq$ `IG`), thus from Lem. 2.4 part (ii), `IP` $\leq$ `Immut`, i.e., `IP` $\leq$ `IP'` $\leq$ `IP`. If `IG` $=$ `Raw`, then $\mathit{isTransitive}(\bot, \Gamma,$ `C'<MO,IP'>`$)$, and `IP'` $\leq$ `Raw`, thus from Lem. 2.3 we have that `IP'` $=$ `IP`. □

# 3 Typing

We next prove that a closed expression has a closed type.

**Lemma 3.1.** *If* $\Gamma \vdash$ `e"` : `T"` *and* `e"` *is closed and* `e"` $\neq$ `null`*, then* `T"` *is closed.*

*Proof.* Note that `null` can have any type `T"` (even an open type) according to rule T-NULL, therefore we require that `e"` $\neq$ `null`.

We prove by induction on the structure of `e"`.

**Value `e"` $=$ `v`** Because `e"` $\neq$ `null`, then `v` $=$ `l`, and the type of a location is always closed `C<NO,NI>`.

**Value `e"` $=$ `e;return l`** Similarly, the type of a location is always closed.

**Method parameter `e"` $=$ `x`** We assumed `e"` is closed, thus it does not contain parameters.

**Object creation `e"` $=$ `new C<FO,VI>(`$\overline{\texttt{e}}$`)`** From T-NEW, `T"` $=$ `C<FO,VI>`, and because `e"` is closed then `T"` must be closed.

**Field access `e"` $=$ `e.f`** Because $\Gamma \vdash$ `e"` : `T"`, from T-FIELD-ACCESS we have that `T"` $=$ `T` and

$$\Gamma \vdash \texttt{e} : \texttt{C<MO,IP>} \qquad \mathit{ftype}(\texttt{e}, \texttt{f}, \texttt{C<MO,IP>}) = \texttt{T}$$

By induction, `C<MO,IP>` is closed. Therefore `T` does not contains `O` nor `I`. Next we show it does not contain `This`. Because *ftype* did not return `error`, then either the field did not contain `This`, or it was substituted. Because `e` $\neq$ `this` then `e` $=$ `l`, and `This` was substituted with `l`.

**Field assignment `e"` $=$ `e.f` $=$ `e'`** Because $\Gamma \vdash$ `e"` : `T"`, from T-FIELD-ASSIGNMENT we have that `T"` $=$ `T'` and $\Gamma \vdash$ `e'` : `T'`
By induction, `T'` is closed.

**Method invocation `e"` $=$ `e`$_0$`.m(`$\overline{\texttt{e}}$`)`** Because $\Gamma \vdash$ `e"` : `T"`, from T-INVOKE we have that

$$\Gamma \vdash \texttt{e}_0 : \texttt{C<MO,IP>} \qquad \mathit{mtype}(\texttt{e}_0, \texttt{m}, \texttt{C<MO,IP>}) = \overline{\texttt{T}} \to \texttt{T"}$$

By induction `C<MO,IP>` is closed, and similar reasoning to field access concludes that `T"` is closed.

□

# 4 Heap

We now prove that if the heap is well-typed, then it remains well-typed even if we remove locations from $H[K]$, i.e., being well-typed is a *stable property*.

**Lemma 4.1.** *Given a well-typed heap $H$, then for any $S \subset H[K]$, the heap $H' = H[K \mapsto S]$ is well-typed.*

*Proof.* This is not trivial, because decreasing $H[K]$ changes the subtyping relation by turning raw objects into immutable. Recall that a well-typed heap $H$ satisfies: (i) there is a linear order $\preceq^T$ over $\mathrm{dom}(H)$ such that for every location $\mathtt{l}$, $\theta(\mathtt{l}) = \mathtt{World}$ or $\theta(\mathtt{l}) \prec^T \mathtt{l}$, and $I(\mathtt{l}) = \mathtt{Mutable}$ or $\kappa(\mathtt{l}) \preceq^T \mathtt{l}$, and (ii) each non-null field location is a subtype of the declared field type.

First, note that the owners and cookers of each location in $H$ and $H'$ are the same, i.e., the same linear order $\preceq^T$ satisfies (i) for $H'$. Suppose to the contrary that $H'$ is not well-typed, i.e., there is some field location $\mathtt{l.f}$ of type $\mathtt{T}$ that points to an object $\mathtt{o}$ of type $\mathtt{T"}$, and $\Gamma_{H'} \not\vdash \mathtt{T"} \leq \mathtt{T}$. Because $H$ is well-typed, we have that $\Gamma_H \vdash \mathtt{T"} \leq \mathtt{T}$. Consider the derivation sequence for $\Gamma_H \vdash \mathtt{T"} \leq \mathtt{T}$. We will take this sequence and transform it into a proof that $\Gamma_{H'} \vdash \mathtt{T"} \leq \mathtt{T}$, which will lead to contradiction. Specifically, we replace every usage of rule S10 ($\Gamma \vdash \mathtt{Immut_l} \leq \mathtt{Raw}$) with a proof that $\Gamma \vdash \mathtt{Immut_l} \leq \mathtt{ReadOnly}$ (using either S10 or S11). We first claim that this is a valid sequence in $\Gamma_{H'}$: rules S1–S9 and S13 do not use $\Gamma[K] = H[K]$ and therefore are identical, rule S10 was removed, and rules S11–S12 is valid because $S \subset H[K]$. We now claim that these sequence proves that $\Gamma_{H'} \vdash \mathtt{T"} \leq \mathtt{T}$, i.e., that each element in the sequence has previous elements that fulfill the assumptions of the rule. Consider an element we removed $\mathtt{Immut_l} \leq \mathtt{Raw}$. Note that $\mathtt{Raw}$ does not appear in $\mathtt{T} = \mathtt{C<MO,IP>}$ because it can only appear in a method guard. The only rule where $\mathtt{Raw}$ appears as a subtype is S6, and by using transitivity (S3), we have that the only conclusion is that $\mathtt{Immut_l} \leq \mathtt{ReadOnly}$, and we added that conclusion. □

Next we prove that owner-as-dominator holds in a well-typed heap.

**Lemma 4.2.** *If heap $H$ is well-typed, then for every location $\mathtt{l} \in \mathrm{dom}(H)$, $\mathtt{l} \mapsto \mathtt{C<NO,NI>}(\overline{\mathtt{v}})$, then either $\mathtt{v}_i = \mathtt{null}$ or $\mathtt{l} \preceq_\theta \theta(\mathtt{v}_i)$.*

*Proof.* Recall that a well-typed heap $H$ satisfies: (i) there is a linear order $\preceq^T$ over $\mathrm{dom}(H)$ such that for every location $\mathtt{l}$, $\theta(\mathtt{l}) = \mathtt{World}$ or $\theta(\mathtt{l}) \prec^T \mathtt{l}$, and $I(\mathtt{l}) = \mathtt{Mutable}$ or $\kappa(\mathtt{l}) \preceq^T \mathtt{l}$, and (ii) each non-null field location is a subtype of the declared field type. From part (i), we have that the relation $\preceq_\theta$ is a tree order.

Consider some $\mathtt{v}_i \neq \mathtt{null}$, and we will prove that $\mathtt{l} \preceq_\theta \theta(\mathtt{v}_i)$. Let

$$\begin{aligned} H[\mathtt{v}_i] &= \mathtt{C'<NO',NI'>(...)} \\ \mathit{fields}(\mathtt{C}) &= \overline{\mathtt{f}} \\ \mathit{ftype}(\mathtt{f}_i, \mathtt{C}) &= \mathtt{C"<FO,IP>} \\ \mathit{ftype}(\mathtt{l}, \mathtt{f}_i, \mathtt{C<NO,NI>}) &= \mathtt{C"<NO",NI">} \end{aligned}$$

We need to prove that $\mathtt{l} \preceq_\theta \mathtt{NO'}$. Because the heap is well-typed, then $\mathtt{C'<NO',NI'>} \leq \mathtt{C"<NO",NI">}$. From Lem. 2.1 part (i), we have that $\mathtt{NO'} = \mathtt{NO"}$. According to the syntax, $\mathtt{FO} = \mathtt{World} \mid \mathtt{This} \mid \mathtt{O}$ (the owner of a field cannot be $\mathtt{l}$ of course because the class declarations cannot use locations). By definition of $\mathit{ftype}(\mathtt{l}, \mathtt{f}_i, \mathtt{C<NO,NI>})$, then $\mathtt{NO"} = \mathtt{World} \mid \mathtt{l} \mid \theta(\mathtt{l})$, respectively. Thus we proved that

$$\mathtt{NO'} = \mathtt{NO"} = \mathtt{World} \mid \mathtt{l} \mid \theta(\mathtt{l})$$

Therefore, because $\mathtt{l} \preceq_\theta \mathtt{World}$ and $\mathtt{l} \preceq_\theta \mathtt{l}$ and $\mathtt{l} \preceq_\theta \theta(\mathtt{l})$, we proved that $\mathtt{l} \preceq_\theta \mathtt{NO'}$. □

# 5 Reduction

We consider only expressions that when reduced using th erased operational semantics, do not result in *null-pointer exceptions*. Null-pointer exceptions can be handled by adding special reduction rules that return `error`, but we prefer to leave the reduction process "stuck".

First we prove that a closed expression reduces in one step to another closed expression.

**Lemma 5.1.** *If $\mathtt{e}$ is closed and $H, \mathtt{e} \to H', \mathtt{e}'$, then $\mathtt{e}'$ is closed.*

*Proof.* Rules R-NEW, R-FIELD-ACCESS, and R-FIELD-ASSIGNMENT result in a value, which is closed. Rule R-INVOKE results in an expression, but all free variables $\overline{\mathtt{x}}, \mathtt{this}, \mathtt{This}, \mathtt{O}, \mathtt{I}$ are substituted with locations, `Immut`, $\mathtt{Immut}_\mathtt{l}$, or `Mutable`. The proof of the congruence rules uses the induction hypothesis. □

**Lemma 5.2.** *If* $H, \mathtt{e} \to H", \mathtt{e}"$ *then (i)* $H[K] = H"[K]$*, (ii)* $\Gamma_H \subseteq \Gamma_{H"}$*, (iii)* $\Gamma_H \vdash \mathtt{e}' : \mathtt{T}' \Rightarrow \Gamma_{H'} \vdash \mathtt{e}' : \mathtt{T}'$*, and (iv)* $\Gamma_H \vdash \mathtt{T} \leq \mathtt{T}' \Rightarrow \Gamma_{H'} \vdash \mathtt{T} \leq \mathtt{T}'$.

*Proof.* Proved by induction on the reduction sequence.

(i) The only reduction rule that mentions $H[K]$ is R-C1, and from the induction hypothesis we have that $H'[K] = H[K] \cup \{\mathtt{l}\}$, therefore the resulting heap is $H" = H'[K \mapsto H'[K] \setminus \{\mathtt{l}\}] = H'[K \mapsto H[K]]$, i.e., $H"[K] = H[K]$.

(ii) None of the reduction rules removes locations or changes the type of a location, therefore $H"$ only includes additional locations, and from (i) $H"[K] = H[K]$.

Parts (iii) and (iv) are trivial from (ii). □

**Theorem 5.3.** ***(Progress and Preservation)*** *For every closed expression* $\mathtt{e} \neq \mathtt{v}$*, and a heap H that is well-typed for* e*, if* $\Gamma_H \vdash \mathtt{e} : \mathtt{T}$*, then there exists* $H', \mathtt{e}', \mathtt{T}'$ *such that (i)* $H, \mathtt{e} \to H', \mathtt{e}'$*, (ii)* $\Gamma_{H'} \vdash \mathtt{e}' : \mathtt{T}'$*, and* $\Gamma_{H'} \vdash \mathtt{T}' \leq \mathtt{T}$*, (iii)* $H'$ *is well-typed for* $\mathtt{e}'$*, (iv)* T*, T', and* $\mathtt{e}'$ *are closed.*

*Proof.* Part (iv) is proved from Lem. 5.1 (we know that è is closed) and from Lem. 3.1 (we know that T" and $\grave{\mathtt{T}}$ are closed).

We assume that there are no null-pointer exceptions, i.e., that for field access, assignment and method invocation, the receiver is never `null`.

It is easy to examine the reduction rules and verify there is always at most one applicable reduction rule. We will split the proof into three stages: (i) **progress:** there is exactly one applicable reduction rule (Lem. 5.4), (ii) **preservation:** $\Gamma_{H'} \vdash \grave{\mathtt{e}} : \grave{\mathtt{T}}$ and $\Gamma_{H'} \vdash \grave{\mathtt{T}} \leq \mathtt{T}"$ (Lem. 5.6), and (iii) $H'$ is well-typed for $\mathtt{e}'$ (Lem. 5.7). □

**Lemma 5.4.** ***(Progress)*** *For every closed expression* $\mathtt{e}" \neq \mathtt{v}$*, and a heap H that is well-typed for* e"*, if* $\Gamma_H \vdash \mathtt{e}" : \mathtt{T}"$*, then there exists* $H', \grave{\mathtt{e}}$ *such that* $H, \mathtt{e}" \to H', \grave{\mathtt{e}}$.

*Proof.* We prove by examining the structure of e". Because it is closed and not a value, then according to our syntax, it must have one of the following forms:

$$\mathtt{e.f} \mid \mathtt{e.f} = \mathtt{e} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{new\ C<FO,VI>}(\overline{\mathtt{e}}) \mid \mathtt{e;return\ l}$$

If the subexpressions are not all values, then we can always apply (exactly) one of the congruence rules. For example, if $\mathtt{e}" = (\mathtt{e;return\ l})$ and e is not a value, then by induction we can apply R-C1.

Therefore, e" has one of the following forms:

$$\mathtt{v.f} \mid \mathtt{v.f} = \mathtt{v} \mid \mathtt{v.m}(\overline{\mathtt{v}}) \mid \mathtt{new\ C<FO,VI>}(\overline{\mathtt{v}}) \mid \mathtt{v;return\ l}$$

Because we assumed we do not have *null pointer exceptions* (in field access, assignment or method invocation), then e" has one of the following forms:

$$\mathtt{l.f} \mid \mathtt{l.f} = \mathtt{v} \mid \mathtt{l.m}(\overline{\mathtt{v}}) \mid \mathtt{new\ C<FO,VI>}(\overline{\mathtt{v}}) \mid \mathtt{v;return\ l}$$

We will next examine the matching five reduction rules (R-FIELD-ACCESS, R-FIELD-ASSIGNMENT, R-INVOKE, R-NEW, R-RETURN) and show that their assumptions hold.

**Rule R-FIELD-ACCESS**

$$\frac{H[\mathtt{l}] = \mathtt{C<NO,NI>}(\overline{\mathtt{v}}) \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{f}}}{H, \mathtt{l.f}_i \to H, \mathtt{v}_i}$$

We assumed that $\Gamma_H \vdash \mathtt{l.f}_i : \mathtt{T}"$, therefore from T-FIELD-ACCESS:

$$\Gamma_H \vdash \mathtt{l} : \mathtt{C<MO,IP>} \qquad \mathit{ftype}(\mathtt{l}, \mathtt{f}_i, \mathtt{C<MO,IP>}) = \mathtt{T}"$$

From the definitions of *ftype* and *fields*, we know that $\mathtt{f}_i \in \mathit{fields}(\mathtt{C})$.

**Rule R-FIELD-ASSIGNMENT**

$$\frac{H[\mathtt{l}] = \mathtt{C<NO,NI>}(\overline{\mathtt{v}}) \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{f}} \qquad \mathtt{NI} = \mathtt{Mutable} \text{ or } \kappa(\mathtt{l}) \in H[K] \qquad \mathtt{v}' = \mathtt{null} \text{ or } \mathtt{l} \preceq_\theta \theta(\mathtt{v}')}{H, \mathtt{l.f}_i = \mathtt{v}' \to H[\mathtt{l} \mapsto \mathtt{C<NO,NI>}([\mathtt{v}'/\mathtt{v}_i]\overline{\mathtt{v}})], \mathtt{v}'}$$

Because $H$ is well-typed, then from Lem. 4.2, we have that $\mathtt{v}' = \mathtt{null}$ or $\mathtt{l} \preceq_\theta \theta(\mathtt{v}')$.

We assumed that $\Gamma_H \vdash \mathtt{l.f}_i = \mathtt{v}' : \mathtt{T}'$, therefore from T-FIELD-ASSIGNMENT:

$$\Gamma_H \vdash \mathtt{l.f}_i : \mathtt{T} \qquad \Gamma_H \vdash \mathtt{l} : \mathtt{C<NO,NI>} \qquad \Gamma_H \vdash \mathtt{NI} \leq \mathtt{Raw}$$

Similarly to field access, because $\Gamma_H \vdash \mathtt{l.f}_i : \mathtt{T}$, then $\mathtt{f}_i \in \mathit{fields}(\mathtt{C})$. From our syntax NI is either Mutable or $\mathtt{Immut}_{\mathtt{l}'}$. We want to show that $\mathtt{NI} = \mathtt{Mutable}$ or $\kappa(\mathtt{l}) \in H[K]$. Therefore we need to show that if $\mathtt{NI} = \mathtt{Immut}_{\mathtt{l}'}$ then $\mathtt{l}' \in H[K]$. Because $\Gamma_H \vdash \mathtt{NI} \leq \mathtt{Raw}$, it must be from S10 and therefore $\mathtt{l}' \in H[K]$.

**Rule R-INVOKE**

$$\frac{H[\mathtt{l}] = \mathtt{C<NO,NI>}(\ldots) \qquad \mathit{mbody}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}'}{H, \mathtt{l.m}(\overline{\mathtt{v}}) \to H, [\overline{\mathtt{v}}/\overline{\mathtt{x}}, \mathtt{l}/\mathtt{this}, \mathtt{l}/\mathtt{This}, \mathtt{NO}/\mathtt{O}, \mathtt{NI}/\mathtt{I}]\mathtt{e}'}$$

We assumed that $\Gamma_H \vdash \mathtt{l.m}(\overline{\mathtt{v}}) : \mathtt{T}$", therefore from T-INVOKE we know that

$$\mathit{mtype}(\mathtt{l}, \mathtt{m}, \mathtt{C<NO,NI>}) = \overline{\mathtt{T}} \to \mathtt{T}"$$

Therefore $\mathit{mbody}(\mathtt{m}, \mathtt{C})$ is defined.

**Rule R-NEW**

$$\frac{\mathtt{l} \notin \mathrm{dom}(H) \qquad \mathtt{VI}' = \begin{cases} \mathtt{Immut}_{\mathtt{l}} & \text{if } \mathtt{VI} = \mathtt{Immut} \text{ or } (\mathtt{VI} = \mathtt{Immut}_{\mathtt{C}} \text{ and } \mathtt{c} \notin H[K]) \\ \mathtt{VI} & \text{otherwise} \end{cases} \qquad H' = H[\mathtt{l} \mapsto \mathtt{C<NO,VI'>}(\overline{\mathtt{null}})]}{H, \mathtt{new\ C<NO,VI>}(\overline{\mathtt{v}}) \to H', \mathtt{l.build}(\overline{\mathtt{v}})\mathtt{;return\ l}}$$

We assumed that $\Gamma_H \vdash \mathtt{new\ C<NO,VI>}(\overline{\mathtt{v}}) : \mathtt{T}$", therefore from T-NEW we know that

$$\mathit{mtype}(\bot, \mathtt{build}, \mathtt{C<NO,NI>}) = \overline{\mathtt{T}} \to \mathtt{U}$$

Thus there is a constructor with $\#(\overline{\mathtt{v}})$ of arguments.

**Rule R-RETURN** Trivial because there are no assumptions for $\mathtt{v;return\ l}$

□

We prove **preservation** (Lem. 5.6) for method invocation by using Lem. 5.5 that uses induction on the size of the method body.

**Lemma 5.5.** ***(Invocation Substitution)*** *For every well-typed heap $H$, location $\mathtt{l}$, values $\overline{\mathtt{v}}$, types $\mathtt{U}$, and guard $\mathtt{IG}$, where*

$$\begin{aligned} &H[\mathtt{l}] = \mathtt{C<NO,NI>} \\ &\Gamma_H \vdash \mathtt{NI} \leq \mathtt{IG} \\ &\Gamma_H \vdash \overline{\mathtt{v}} : \overline{\mathtt{U}'} \\ &\Gamma_H \vdash \overline{\mathtt{U}'} \leq \overline{[\mathtt{l}/\mathtt{This}, \mathtt{NI}/\mathtt{I}, \mathtt{NO}/\mathtt{O}]\mathtt{U}} \end{aligned} \tag{4}$$

*Then, for any $\mathtt{e}$" such that $\Gamma \vdash \mathtt{e}" : \mathtt{S}$, $\Gamma = \{\mathtt{I} : \mathtt{IG}, \overline{\mathtt{x}} : \overline{\mathtt{U}}, \mathtt{this} : \mathtt{C<O,I>}\}$, then*

$$\begin{aligned} &\Gamma_H \vdash [\mathtt{l}/\mathtt{This}, \mathtt{NI}/\mathtt{I}, \mathtt{NO}/\mathtt{O}, \overline{\mathtt{v}}/\overline{\mathtt{x}}, \mathtt{l}/\mathtt{this}]\mathtt{e}" : \mathtt{S}' \\ &\Gamma_H \vdash \mathtt{S}' \leq [\mathtt{l}/\mathtt{This}, \mathtt{NI}/\mathtt{I}, \mathtt{NO}/\mathtt{O}]\mathtt{S} \end{aligned}$$

*Proof.* We will prove by induction on the structure of $\mathtt{e}$".

$\mathtt{e}" = (\mathtt{e;return\ l})$ Impossible because $\Gamma$ does not contain locations.

$\mathtt{e}" = (\mathtt{v})$ $\Gamma$ does not contain locations. Therefore, $\mathtt{v} = \mathtt{null}$, and we can choose $\mathtt{S}' = [\mathtt{l}/\mathtt{This}, \mathtt{NI}/\mathtt{I}, \mathtt{NO}/\mathtt{O}]\mathtt{S}$.

$\mathbf{e}'' = (\mathbf{this})$ Then $\mathtt{S} = \mathtt{C<O,I>}$, and

$$
\begin{array}{l}
[\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O},\overline{\mathtt{v}}/\overline{\mathtt{x}},\mathtt{l}/\mathtt{this}]\mathtt{e}'' = \mathtt{l} \\
[\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{S} = \mathtt{C<NO,NI>} \\
\mathtt{S}' = \mathtt{C<NO,NI>} \\
\Gamma_H \vdash \mathtt{l} : \mathtt{S}' \\
\Gamma_H \vdash \mathtt{S}' \leq \mathtt{C<NO,NI>}
\end{array}
$$

$\mathbf{e}'' = (\mathbf{x}_i)$ Then $\mathtt{S} = \mathtt{U}_i$. We assumed that

$$
\begin{array}{l}
\Gamma_H \vdash \mathtt{v}_i : \mathtt{U}_i' \\
\Gamma_H \vdash \mathtt{U}_i' \leq [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{U}_i
\end{array}
$$

Therefore,

$$
\begin{array}{l}
[\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O},\overline{\mathtt{v}}/\overline{\mathtt{x}},\mathtt{l}/\mathtt{this}]\mathtt{e}'' = \mathtt{v}_i \\
\mathtt{S}' = \mathtt{U}_i' \\
\mathtt{S} = \mathtt{U}_i \\
\Gamma_H \vdash \mathtt{v}_i : \mathtt{S}' \\
\Gamma_H \vdash \mathtt{S}' \leq [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{S}
\end{array}
$$

$\mathbf{e}'' = (\mathbf{new\ D<FO,VI>(\overline{e})})$ We assumed that $\Gamma \vdash \mathtt{new\ D<FO,VI>}(\overline{\mathtt{e}}) : \mathtt{S}$. From T-NEW, $\mathtt{S} = \mathtt{D<FO,VI>}$ and

$$
\mathit{mtype}(\bot, \mathtt{build}, \mathtt{D<FO,VI>}) = \overline{\mathtt{T}} \to \mathtt{U} \qquad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}'} \qquad \Gamma \vdash \overline{\mathtt{T}'} \leq \overline{\mathtt{T}} \tag{5}
$$

By induction on $\mathtt{e}_i$, we have that

$$
\begin{array}{l}
\Gamma_H \vdash [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O},\overline{\mathtt{v}}/\overline{\mathtt{x}},\mathtt{l}/\mathtt{this}]\mathtt{e}_i : \mathtt{V}_i \\
\Gamma_H \vdash \mathtt{V}_i \leq [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{T}_i'
\end{array} \tag{6}
$$

From Lem. 2.6 and (5), we have that

$$
\Gamma_H \vdash [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{T}_i' \leq [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{T}_i \tag{7}
$$

From transitivity, (6), and (7), we have

$$
\Gamma_H \vdash \mathtt{V}_i \leq [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{T}_i \tag{8}
$$

From definition of *mtype* we have

$$
\mathit{mtype}(\bot, \mathtt{build}, [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{D<FO,VI>}) = [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}](\overline{\mathtt{T}} \to \mathtt{U}) \tag{9}
$$

From T-NEW, (8), and (9),

$$
\Gamma_H \vdash [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O},\overline{\mathtt{v}}/\overline{\mathtt{x}},\mathtt{l}/\mathtt{this}]\mathtt{new\ D<FO,VI>}(\overline{\mathtt{e}}) : [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{D<FO,VI>}
$$

$\mathbf{e}'' = (\mathbf{e.f})$ From T-FIELD-ACCESS,

$$
\begin{array}{l}
\Gamma \vdash \mathtt{e} : \mathtt{D<MO,IP>} \\
\Gamma \vdash \mathtt{e.f} : \mathtt{S} \\
\mathtt{S} = \mathit{ftype}(\mathtt{e},\mathtt{f},\mathtt{D<MO,IP>})
\end{array}
$$

Recall that $\mathtt{S} = \mathit{ftype}(\mathtt{e},\mathtt{f},\mathtt{D<MO,IP>}) = \mathit{substitute}(\mathtt{e},\mathtt{D<MO,IP>},\mathit{ftype}(\mathtt{f},\mathtt{D}))$. Note that $\mathtt{e}$ is not a location, and thus if $\mathtt{f}$ is $\mathtt{this}$-owned, then $\mathtt{e} = \mathtt{this}$. Consider first the case that $\mathtt{f}$ is $\mathtt{this}$-owned, thus $\mathtt{e} = \mathtt{this}$. Let $\mathit{ftype}(\mathtt{f},\mathtt{D}) = \mathtt{FT}$, and $O(\mathtt{FT}) = \mathtt{This}$. We assumed that $\Gamma \vdash \mathtt{this}.f : \mathtt{S}$. Then, $\mathtt{S} = \mathtt{FT}$. We need to show that

$$
\begin{array}{l}
\Gamma_H \vdash [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O},\overline{\mathtt{v}}/\overline{\mathtt{x}},\mathtt{l}/\mathtt{this}]\mathtt{this.f} : \mathtt{S}' \\
\Gamma_H \vdash \mathtt{S}' \leq [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{FT}
\end{array}
$$

From T-FIELD-ACCESS

$$\Gamma_H \vdash \texttt{l.f} : [\texttt{l}/\texttt{This},\texttt{NI}/\texttt{I},\texttt{NO}/\texttt{O}]\texttt{FT}$$

, i.e., $\texttt{S}' = [\texttt{l}/\texttt{This},\texttt{NI}/\texttt{I},\texttt{NO}/\texttt{O}]\texttt{FT}$.

Now suppose that `f` is not `this`-owned. Therefore, $\texttt{S} = [\texttt{IP}/\texttt{I},\texttt{MO}/\texttt{O}]\texttt{FT}$. We need to show that

$$\begin{array}{l}\Gamma_H \vdash [\texttt{l}/\texttt{This},\texttt{NI}/\texttt{I},\texttt{NO}/\texttt{O},\overline{\texttt{v}}/\overline{\texttt{x}},\texttt{l}/\texttt{this}]\texttt{e.f} : \texttt{S}' \\ \Gamma_H \vdash \texttt{S}' \leq [\texttt{l}/\texttt{This},\texttt{NI}/\texttt{I},\texttt{NO}/\texttt{O}]\texttt{S} \quad \texttt{S} = [\texttt{IP}/\texttt{I},\texttt{MO}/\texttt{O}]\texttt{FT}\end{array} \tag{10}$$

From the induction on `e`, we have that

$$\begin{array}{l}\Gamma_H \vdash [\texttt{l}/\texttt{This},\texttt{NI}/\texttt{I},\texttt{NO}/\texttt{O},\overline{\texttt{v}}/\overline{\texttt{x}},\texttt{l}/\texttt{this}]\texttt{e} : \texttt{S"} \\ \Gamma_H \vdash \texttt{S"} \leq [\texttt{l}/\texttt{This},\texttt{NI}/\texttt{I},\texttt{NO}/\texttt{O}]\texttt{D<MO,IP>}\end{array} \tag{11}$$

From (11), $O(\texttt{S"}) = [\texttt{l}/\texttt{This},\texttt{NO}/\texttt{O}]\texttt{MO}$. From (10) and (11), and Lem. 2.2, we have that

$$\begin{array}{l}\texttt{S}' = \mathit{ftype}(\bot,\texttt{f},\texttt{S"}) = [I(\texttt{S"})/\texttt{I},O(\texttt{S"})/\texttt{O}]\texttt{FT} \leq [([\texttt{NI}/\texttt{I}]\texttt{IP})/\texttt{I},([\texttt{l}/\texttt{This},\texttt{NO}/\texttt{O}]\texttt{MO})/\texttt{O}]\texttt{FT} = \\ = [\texttt{l}/\texttt{This},\texttt{NI}/\texttt{I},\texttt{NO}/\texttt{O}]([\texttt{IP}/\texttt{I},\texttt{MO}/\texttt{O}]\texttt{FT})\end{array}$$

**e"** $= (\texttt{e.f=e}')$ The challenge in field assignment is that (by induction) the type of the substitution of `e` changed covariantly (i.e., it is a subtype of the substitution of the type), and $\texttt{e}'$ also changed covariantly. However, we will prove that because $I(\texttt{e})$ is `Raw`, and `e` is either `this` or `this`-owned, then `e` is invariant.

We assumed that $\Gamma \vdash \texttt{e.f=e}' : \texttt{S}$. From T-FIELD-ASSIGNMENT, we know that

$$\begin{array}{l}\Gamma \vdash \texttt{e.f} : \texttt{T} \quad \Gamma \vdash \texttt{e}' : \texttt{S} \quad \Gamma \vdash \texttt{S} \leq \texttt{T} \quad \Gamma \vdash \texttt{e} : \texttt{D<MO,IP>} \\ \Gamma \vdash \texttt{IP} \leq \texttt{Raw} \quad \mathit{isTransitive}(\texttt{e},\Gamma,\texttt{D<MO,IP>}) \quad \texttt{MO} \neq \texttt{?}\end{array} \tag{12}$$

We wish to prove all the assumptions in (12) after substituting $[\texttt{l}/\texttt{This},\texttt{NI}/\texttt{I},\texttt{NO}/\texttt{O},\overline{\texttt{v}}/\overline{\texttt{x}},\texttt{l}/\texttt{this}]$.

By induction on $\texttt{e}'$ we have that

$$\begin{array}{l}\Gamma_H \vdash [\texttt{l}/\texttt{This},\texttt{NI}/\texttt{I},\texttt{NO}/\texttt{O},\overline{\texttt{v}}/\overline{\texttt{x}},\texttt{l}/\texttt{this}]\texttt{e}' : \texttt{S}' \\ \Gamma_H \vdash \texttt{S}' \leq [\texttt{l}/\texttt{This},\texttt{NI}/\texttt{I},\texttt{NO}/\texttt{O}]\texttt{S}\end{array} \tag{13}$$

By induction on `e.f` we have that

$$\begin{array}{l}\Gamma_H \vdash [\texttt{l}/\texttt{This},\texttt{NI}/\texttt{I},\texttt{NO}/\texttt{O},\overline{\texttt{v}}/\overline{\texttt{x}},\texttt{l}/\texttt{this}]\texttt{e.f} : \texttt{T}' \\ \Gamma_H \vdash \texttt{T}' \leq [\texttt{l}/\texttt{This},\texttt{NI}/\texttt{I},\texttt{NO}/\texttt{O}]\texttt{T}\end{array} \tag{14}$$

From the proof of field access above, we see that the class of $\texttt{T}'$ and `T` is the same. By induction on `e` we have that

$$\begin{array}{l}\Gamma_H \vdash [\texttt{l}/\texttt{This},\texttt{NI}/\texttt{I},\texttt{NO}/\texttt{O},\overline{\texttt{v}}/\overline{\texttt{x}},\texttt{l}/\texttt{this}]\texttt{e} : \texttt{D'<MO',IP'>} \\ \Gamma_H \vdash \texttt{D'<MO',IP'>} \leq [\texttt{l}/\texttt{This},\texttt{NI}/\texttt{I},\texttt{NO}/\texttt{O}]\texttt{D<MO,IP>}\end{array} \tag{15}$$

Because $\texttt{MO} \neq \texttt{?}$, from Lem. 2.1 part (i), we have that $\texttt{MO'} = \texttt{MO}$.

We now prove that the following holds:

$$\begin{array}{l}\Gamma_H \vdash [\texttt{NI}/\texttt{I}]\texttt{IP} \leq \texttt{Raw} \\ \mathit{isTransitive}([\texttt{l}/\texttt{this}]\texttt{e},\Gamma,[\texttt{l}/\texttt{This},\texttt{NI}/\texttt{I},\texttt{NO}/\texttt{O}]\texttt{D<MO,IP>})\end{array} \tag{16}$$

From (12), $\Gamma \vdash \texttt{IP} \leq \texttt{Raw}$. From our syntax, and because $\Gamma$ does not contain locations:

$$\texttt{IP} = \texttt{ReadOnly} \mid \texttt{Immut} \mid \texttt{Mutable} \mid \texttt{I}$$

If $\texttt{IP} = \texttt{Mutable}$ then we proved (16). Therefore, it must be that $\texttt{IP} = \texttt{I}$ (thus $[\texttt{NI}/\texttt{I}]\texttt{IP} = \texttt{NI}$) and $\Gamma(\texttt{I}) = \texttt{IG} \leq \texttt{Raw}$. From (4) ($\Gamma_H \vdash \texttt{NI} \leq \texttt{IG}$), we proved the first part of (16) that $\Gamma_H \vdash [\texttt{NI}/\texttt{I}]\texttt{IP} \leq \texttt{Raw}$. If $\texttt{IG} = \texttt{Mutable}$

then, NI = Mutable, which proved (16). Therefore IG = Raw, and from the definition of *isTransitive* we have that e=this or MO = This. If e = this then *isTransitive*(l,...), which proved (16). Thus MO = This, and

$$\begin{array}{c} \texttt{D<MO,IP>} = \texttt{D<This,I>} \\ [\texttt{l/This,NI/I,NO/O}]\texttt{D<MO,IP>} = \texttt{D<l,NI>} \end{array} \tag{17}$$

From (4), $H[\texttt{l}] = \texttt{C<NO,NI>}$. If NI = Mutable then we proved (16). Otherwise $\texttt{NI} = \texttt{Immut}_{\texttt{l}'}$. Because $H$ is well-typed, $\texttt{l}' \preceq^T \texttt{l}$, thus $\texttt{l}' \not\prec_\theta \texttt{l}$, proving (16) (because if $\texttt{a} \prec_\theta \texttt{b}$ then $\texttt{b} \prec^T \texttt{a}$).

**(i)** If e = this. Let *ftype*(f,D) = FT. We assumed in (12) that $\Gamma \vdash \texttt{this}.f : \texttt{T}$. Then, T = FT. From T-Field-Access

$$\Gamma_H \vdash \texttt{l.f} : [\texttt{l/This,NI/I,NO/O}]\texttt{FT}$$

From definition of *isTransitive*, we have that *isTransitive*(l,...) holds. From (12) ($\Gamma \vdash \texttt{S} \leq \texttt{T}$) and Lem. 2.6, we have

$$\Gamma_H \vdash [\texttt{l/This,NI/I,NO/O}]\texttt{S} \leq [\texttt{l/This,NI/I,NO/O}]\texttt{T} \tag{18}$$

From (13), (18), and transitivity, we have

$$\Gamma_H \vdash \texttt{S}' \leq [\texttt{l/This,NI/I,NO/O}]\texttt{FT} \tag{19}$$

To summarize, from (19), (16) ($\Gamma_H \vdash \texttt{NI} \leq \texttt{Raw}$), we have that

$$\begin{array}{l} \Gamma_H \vdash \texttt{l.f} : [\texttt{l/This,NI/I,NO/O}]\texttt{FT} \quad \Gamma_H \vdash \texttt{e}' : \texttt{S}' \quad \Gamma_H \vdash \texttt{S}' \leq [\texttt{l/This,NI/I,NO/O}]\texttt{FT} \\ \Gamma_H \vdash \texttt{l} : \texttt{C<NO,NI>} \quad \Gamma_H \vdash \texttt{NI} \leq \texttt{Raw} \quad \textit{isTransitive}(\texttt{l},\ldots) \end{array} \tag{20}$$

Therefore, from (20), and T-Field-Assignment, we proved that

$$\begin{array}{l} \Gamma_H \vdash \texttt{l.f=e}' : \texttt{S}' \\ \Gamma_H \vdash \texttt{S}' \leq [\texttt{l/This,NI/I,NO/O}]\texttt{FT} \end{array}$$

**(ii)** If e ≠ this, then from (16), we have

$$\textit{isTransitive}(\bot, \Gamma, [\texttt{l/This,NI/I,NO/O}]\texttt{D<MO,IP>}) \tag{21}$$

From (15) and (21) and Lem. 2.3, we have that $\texttt{IP}' = [\texttt{NI/I}]\texttt{IP}$. To summarize, from (13), (14), (18), (21),

$$\begin{array}{l} \Gamma_H \vdash [\texttt{l/This,NI/I,NO/O},\overline{\texttt{v}}/\overline{\texttt{x}},\texttt{l/this}]\texttt{e.f} : [\texttt{l/This,NI/I,NO/O}]\texttt{T} \\ \Gamma_H \vdash [\texttt{l/This,NI/I,NO/O},\overline{\texttt{v}}/\overline{\texttt{x}},\texttt{l/this}]\texttt{e}' : \texttt{S}' \\ \Gamma_H \vdash \texttt{S}' \leq [\texttt{l/This,NI/I,NO/O}]\texttt{S} \\ \Gamma_H \vdash [\texttt{l/This,NI/I,NO/O}]\texttt{S} \leq [\texttt{l/This,NI/I,NO/O}]\texttt{T} \\ \Gamma_H \vdash [\texttt{l/This,NI/I,NO/O},\overline{\texttt{v}}/\overline{\texttt{x}},\texttt{l/this}]\texttt{e} : \texttt{D}'\texttt{<MO,IP}'\texttt{>} \\ \Gamma_H \vdash \texttt{IP}' \leq \texttt{Raw} \\ \quad \textit{isTransitive}(\bot, \Gamma, \texttt{D}'\texttt{<MO,IP}'\texttt{>}) \quad \texttt{MO} \neq \texttt{?} \end{array} \tag{22}$$

Thus, from (22), and T-Field-Assignment, we proved that

$$\begin{array}{l} \Gamma_H \vdash \texttt{e.f=e}' : \texttt{S}' \\ \Gamma_H \vdash \texttt{S}' \leq [\texttt{l/This,NI/I,NO/O}]\texttt{S} \end{array}$$

**e"** = (**e**$_0$**.m(**$\overline{\textbf{e}}$**)**) The proof is similar in spirit to field assignment: the challenge is that both $\texttt{e}_0$ and $\texttt{e}_i$ change covariantly. Let IG′ be the guard of m. If IG′ = ReadOnly then the parameters of m cannot include I. If IG′ = Mutable | Immut, then I remains with the same bound. The challenge is when IG′ = Raw, then we use either the fact the $\texttt{e}_0$ is either this or this-owned, to prove that $\texttt{e}_0$ is invariant (like in field assignment).

With respect to wildcards, if the receiver $\texttt{e}_0$ has a wildcard, then after the covariant change it might no longer be the case. Therefore we require that the owner of method parameters in this case must be World. (it cannot be O nor This).

From T-INVOKE,

$$\frac{\Gamma \vdash \mathtt{e}_0 : \mathtt{D<MO,IP>} \quad \mathit{mtype}(\mathtt{e}_0,\mathtt{m},\mathtt{D<MO,IP>}) = \overline{\mathtt{T}} \to \mathtt{W} \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{T}'} \quad \Gamma \vdash \overline{\mathtt{T}'} \leq \overline{\mathtt{T}} \quad \mathit{mguard}(\mathtt{m},\mathtt{D}) = \mathtt{IG}' \\ \Gamma \vdash \mathtt{IP} \leq \mathtt{IG}' \quad \mathtt{IG}' = \mathtt{Raw} \Rightarrow \mathit{isTransitive}(\mathtt{e}_0,\Gamma,\mathtt{D<MO,IP>}) \quad \mathit{mtype}(\mathtt{m},\mathtt{D}) = \overline{\mathtt{U}} \to \mathtt{V} \quad O(\overline{\mathtt{T}}) = \mathtt{?} \Rightarrow O(\overline{\mathtt{U}}) = \mathtt{?}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{W}} \tag{23}$$

By induction on $\mathtt{e}_0$, we have that

$$\begin{array}{l} \Gamma_H \vdash [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O},\overline{\mathtt{v}}/\overline{\mathtt{x}},\mathtt{l}/\mathtt{this}]\mathtt{e}_0 : \mathtt{D'<MO',IP'>} \\ \Gamma_H \vdash \mathtt{D'<MO',IP'>} \leq [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{D<MO,IP>} \end{array} \tag{24}$$

Note that, in contrast with field assignment, here we might have $\mathtt{MO} = \mathtt{?}$, and then $\mathtt{MO'} \neq \mathtt{MO}$. However, from Lem. 2.1 part (i),

$$\mathtt{MO} \neq \mathtt{?} \Rightarrow \mathtt{MO'} = [\mathtt{l}/\mathtt{This},\mathtt{NO}/\mathtt{O}]\mathtt{MO} \tag{25}$$

By induction on $\mathtt{e}_i$, we have that

$$\begin{array}{l} \Gamma_H \vdash [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O},\overline{\mathtt{v}}/\overline{\mathtt{x}},\mathtt{l}/\mathtt{this}]\mathtt{e}_i : \mathtt{S}'_i \\ \Gamma_H \vdash \mathtt{S}'_i \leq [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{T'}_i \end{array} \tag{26}$$

From (26) and (23) and Lem. 2.6, we have

$$\Gamma_H \vdash [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{T'}_i \leq [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{T}_i \tag{27}$$

From transitivity, (26), and (27),

$$\Gamma_H \vdash \mathtt{S}'_i \leq [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{T}_i \tag{28}$$

Because method overriding maintains the same signature, we have that

$$\mathit{mtype}(\mathtt{m},\mathtt{D'}) = \mathit{mtype}(\mathtt{m},\mathtt{D}) = \overline{\mathtt{U}} \to \mathtt{V} \tag{29}$$

From definition of *mtype*, (29), and because $\mathtt{e}_0$ does not contain locations, we have that

$$\begin{array}{r} \mathit{mtype}(\mathtt{e}_0,\mathtt{m},\mathtt{D<MO,IP>}) = \overline{\mathtt{T}} \to \mathtt{W} = [\mathtt{MO}/\mathtt{O},\mathtt{IP}/\mathtt{I}](\overline{\mathtt{U}} \to \mathtt{V}) \\ \mathit{mtype}([\mathtt{l}/\mathtt{this}]\mathtt{e}_0,\mathtt{m},\mathtt{D'<MO',IP'>}) = [\mathtt{l}/\mathtt{This},\mathtt{MO'}/\mathtt{O},\mathtt{IP'}/\mathtt{I}](\overline{\mathtt{U}} \to \mathtt{V}) \end{array} \tag{30}$$

We will always prove that the parameters are invariant, i.e.,

$$\mathit{mtype}([\mathtt{l}/\mathtt{this}]\mathtt{e}_0,\mathtt{m},\mathtt{D'<MO',IP'>}) = \overline{[\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{T}_i} \to \mathtt{W'} \tag{31}$$

We will also prove that

$$\Gamma_H \vdash \mathtt{IP'} \leq \mathtt{IG'} \quad \mathtt{IG'} = \mathtt{Raw} \Rightarrow \mathit{isTransitive}([\mathtt{l}/\mathtt{this}]\mathtt{e}_0,\Gamma,\mathtt{D'<MO',IP'>}) \tag{32}$$

Because there are no wildcards after substitution, from (31) and (32), we will have that

$$\begin{array}{l} \Gamma_H \vdash [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O},\overline{\mathtt{v}}/\overline{\mathtt{x}},\mathtt{l}/\mathtt{this}]\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{W}' \\ \Gamma_H \vdash \mathtt{W}' \leq [\mathtt{l}/\mathtt{This},\mathtt{NI}/\mathtt{I},\mathtt{NO}/\mathtt{O}]\mathtt{W} \end{array}$$

Next we prove (31) just for the owner parameter, i.e., we want to show that (from (30) and (31))

$$O([\mathtt{l}/\mathtt{This},\mathtt{MO'}/\mathtt{O}]\mathtt{U}_i) = O([\mathtt{l}/\mathtt{This},\mathtt{NO}/\mathtt{O}]\mathtt{T}_i) \tag{33}$$

From (30),

$$O(\mathtt{T}_i) = O([\mathtt{MO}/\mathtt{O}]\mathtt{U}_i) \tag{34}$$

If $O(\mathtt{U}_i) = \mathtt{This}$, then both sides of (33) are $\mathtt{l}$. If $O(\mathtt{U}_i) \neq \mathtt{O}$, then both sides of (33) are $O(\mathtt{U}_i)$. The last case is that $O(\mathtt{U}_i) = \mathtt{O}$. From (23), we have

$$O(\mathtt{T}_i) = \mathtt{?} \Rightarrow O(\mathtt{U}_i) = \mathtt{?} \tag{35}$$

On the one hand, if $\texttt{MO} = \texttt{?}$, then $O(\texttt{T}_i) = \texttt{?}$, thus $O(\texttt{U}_i) = \texttt{?}$, and both sides of (33) are ?. On the other hand, if $\texttt{MO} \neq \texttt{?}$, then from (25)

$$\texttt{MO'} = [\texttt{l}/\texttt{This}, \texttt{NO}/\texttt{O}]\texttt{MO}$$

which proves (33).

From (33), in order to prove (31), we just need to show it for the immutability parameter, i.e., (from (30))

$$I([\texttt{IP'}/\texttt{I}]\texttt{U}_i) = I([\texttt{NI}/\texttt{I}]([\texttt{IP}/\texttt{I}]\texttt{U}_i)) \tag{36}$$

Recall the following: From (24), we know that $\Gamma_H \vdash \texttt{D'<MO', IP'>} \leq [\texttt{l}/\texttt{This}, \texttt{NI}/\texttt{I}, \texttt{NO}/\texttt{O}]\texttt{D<MO, IP>}$. From (23), $\Gamma \vdash \texttt{IP} \leq \texttt{IG'}$. From (4), $\Gamma_H \vdash \texttt{NI} \leq \texttt{IG}$ and $\texttt{I} : \texttt{IG} \in \Gamma$.

We will split the proof by the four possible values of $\texttt{IG'} = \texttt{ReadOnly} \mid \texttt{Mutable} \mid \texttt{Immut} \mid \texttt{Raw}$. For each case we need to prove (32) and (36).

**(i) IG' = ReadOnly** Because any immutability is a subtype of ReadOnly, we proved (32). Furthermore, when $\texttt{IG'} = \texttt{ReadOnly}$ the signature of parameters cannot contain I, i.e., $I(\texttt{U}_i) \neq \texttt{I}$, which proved (36).

**(ii) IG' = Mutable** From (23), $\Gamma \vdash \texttt{IP} \leq \texttt{Mutable}$, thus either $\texttt{IP} = \texttt{Mutable}$ or ($\texttt{IG} = \texttt{Mutable}$ and $\texttt{IP} = \texttt{I}$). If $\texttt{IP} = \texttt{I}$, then from (4), $\Gamma_H \vdash \texttt{NI} \leq \texttt{Mutable}$, thus $\texttt{NI} = \texttt{Mutable}$. Thus, $\Gamma_H \vdash [\texttt{NI}/\texttt{I}]\texttt{IP} \leq \texttt{Mutable}$. Therefore, from (24) and Lem. 2.4 part (i), we have that $\Gamma_H \vdash \texttt{IP'} \leq \texttt{Mutable}$, which proved (32).

We showed that if $\texttt{IP} = \texttt{I}$, then $\texttt{NI} = \texttt{Mutable}$ and $\texttt{IP'} = \texttt{Mutable}$, proving (36). We also showed that if $\texttt{IP} = \texttt{Mutable}$ then $\texttt{IP'} = \texttt{Mutable}$, proving (36).

**(iii) IG' = Immut** Exactly like part (ii), but we use Lem. 2.4 part (ii) instead of part (i), and ($\texttt{Immut}_{\texttt{l'}}$ where $\texttt{l'} \notin H[K]$) instead of Mutable.

**(iv) IG' = Raw** Exactly like in field assignment, we prove that:

$$\begin{array}{l} \Gamma_H \vdash [\texttt{NI}/\texttt{I}]\texttt{IP} \leq \texttt{Raw} \\ \mathit{isTransitive}([\texttt{l}/\texttt{this}]\texttt{e}, \Gamma, [\texttt{l}/\texttt{This}, \texttt{NI}/\texttt{I}, \texttt{NO}/\texttt{O}]\texttt{D<MO, IP>}) \end{array} \tag{37}$$

If $\texttt{e} = \texttt{this}$, then $\texttt{D} = \texttt{C}$, $\texttt{IP} = \texttt{I}$ (because $\Gamma \vdash \texttt{this} : \texttt{D<O, I>}$) and $\texttt{IP'} = \texttt{NI}$ (because $\Gamma_H \vdash \texttt{l} : \texttt{D<NO, NI>}$), therefore,

$$I([\texttt{NI}/\texttt{I}]\texttt{U}_i) = I([\texttt{NI}/\texttt{I}]([\texttt{I}/\texttt{I}]\texttt{U}_i))$$

which proved (36). Furthermore, because $\Gamma \vdash \texttt{IP} \leq \texttt{Raw}$ (and $\texttt{IP} = \texttt{I}$), we know that $\texttt{IG} \leq \texttt{Raw}$. Thus from (4), $\Gamma_H \vdash \texttt{NI} \leq \texttt{Raw}$. Finally because this was replaced with l, and $\mathit{isTransitive}(\texttt{l}, \ldots)$ always holds, then we proved (32).

If $\texttt{e} \neq \texttt{this}$, then from (37), we have that $\mathit{isTransitive}(\bot, \Gamma, [\texttt{l}/\texttt{This}, \texttt{NI}/\texttt{I}, \texttt{NO}/\texttt{O}]\texttt{D<MO, IP>})$, thus from definition of *isTransitive* we have that $\texttt{MO} \neq \texttt{?}$. From Lem. 2.4 part (iii) and (24), we know that $\texttt{IP'} = [\texttt{NI}/\texttt{I}]\texttt{IP}$, which proved (36). Combined with (37), we have that $\Gamma_H \vdash \texttt{IP'} \leq \texttt{Raw}$. From (25), we have $\texttt{MO'} = [\texttt{l}/\texttt{This}, \texttt{NO}/\texttt{O}]\texttt{MO}$. Therefore, $\texttt{D'<MO', IP'>} = [\texttt{l}/\texttt{This}, \texttt{NI}/\texttt{I}, \texttt{NO}/\texttt{O}]\texttt{D<MO, IP>}$, which proved (32).

□

**Lemma 5.6.** ***(Subtype preservation)*** *For every closed expression* e" $\neq$ v*, and a heap H that is well-typed for* e"*, if* $\Gamma_H \vdash \texttt{e"} : \texttt{T"}$ *and* $H, \texttt{e"} \rightarrow H', \grave{\texttt{e}}$*, then* $\Gamma_{H'} \vdash \grave{\texttt{e}} : \grave{\texttt{T}}$ *and* $\Gamma_{H'} \vdash \grave{\texttt{T}} \leq \texttt{T"}$.

*Proof.* We prove by examining all possible reduction rules.

**Congruence for field access** Consider the congruence rule for field access

$$\frac{H, \texttt{e} \rightarrow H', \texttt{e}'}{H, \texttt{e.f} \rightarrow H', \texttt{e}'\texttt{.f}}$$

We assumed that $\Gamma_H \vdash \texttt{e.f} : \texttt{T"}$ and by induction $\Gamma_H \vdash \texttt{e} : \texttt{T}$, $\Gamma_{H'} \vdash \texttt{e}' : \texttt{T}'$ and $\Gamma_{H'} \vdash \texttt{T}' \leq \texttt{T}$. Let $\texttt{T} = \texttt{C<MO, IP>}$. Because $\texttt{e} \neq \texttt{this}$ (cause e is closed) and $\texttt{e} \neq \texttt{l}$ (cause a location cannot be reduced further), then field f is not this-owned, and $\texttt{T"} = \mathit{ftype}(\bot, \texttt{f}, \texttt{T})$.

Because $\Gamma_{H'} \vdash \texttt{T}' \leq \texttt{T}$, from Lem. 2.1 part (iii), then $\texttt{C}'$ is a subtype of C. Therefore $\mathit{fields}(\texttt{C}')$ must contain the same field f which is not this-owned. Thus, $\Gamma_{H'} \vdash \grave{\texttt{e}} : \grave{\texttt{T}}$, where $\grave{\texttt{T}} = \mathit{ftype}(\bot, \texttt{f}, \texttt{T}')$. The last thing we need to prove is that $\Gamma_{H'} \vdash \grave{\texttt{T}} \leq \texttt{T"}$, which follows from Lem. 2.7.

**Congruence for method receiver** Consider the congruence rule for method receiver

$$\frac{H, \mathtt{e}_0 \to H', \mathtt{e}_0'}{H, \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \to H', \mathtt{e}_0'\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)}}$$

Similarly to field access, because $\mathtt{e}_0$ is not a location, then none of the parameters or return type of method `m` is `this`-owned. Proving that $\Gamma_{H'} \vdash \grave{\mathtt{T}} \leq \mathtt{T}$" (i.e., the return type is preserved) is done similarly to field access, by noting that method overriding maintains the same return type. (The return type could also change covariantly and the proof would still hold.) However, proving that $\Gamma_{H'} \vdash \grave{\mathtt{e}} : \grave{\mathtt{T}}$ is more challenging because: (i) we need to show that $\mathtt{e}_0'$ satisfies the guard, and (ii) the type of method parameters after substitution can change covariantly (as opposed to FGJ, which is invariant).

We will first prove that $\mathtt{e}_0'$ satisfies the guard. We assumed that $\Gamma_H \vdash \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} : \mathtt{T}$" and by induction $\Gamma_H \vdash \mathtt{e}_0 : \mathtt{T}$, $\Gamma_{H'} \vdash \mathtt{e}_0' : \mathtt{T}'$ and $\Gamma_{H'} \vdash \mathtt{T}' \leq \mathtt{T}$. From T-Invoke, we know that

$$\Gamma_H \vdash \mathtt{e}_0 : \mathtt{C<MO,IP>} \quad \mathit{mguard}(\mathtt{m},\mathtt{C}) = \mathtt{IG} \quad \Gamma_H \vdash \mathtt{IP} \leq \mathtt{IG} \quad \mathtt{IG} = \mathtt{Raw} \Rightarrow \mathit{isTransitive}(\mathtt{e}_0, \Gamma, \mathtt{C<MO,IP>})$$

Because $\mathtt{e}_0$ is neither `this` nor `l` (because it was reduced), then $\mathtt{IG} = \mathtt{Raw} \Rightarrow \mathit{isTransitive}(\bot, \Gamma, \mathtt{C<MO,IP>})$. Let $\mathtt{T}' = \mathtt{C'<MO,IP'>}$ and $\mathit{mguard}(\mathtt{m},\mathtt{C}') = \mathtt{IG}'$. Because $\Gamma_{H'} \vdash \mathtt{T}' \leq \mathtt{T}$, from Lem. 2.1 part (iii), then $\mathtt{C}'$ is a subtype of `C`. From the restriction on method overriding with guards, $\mathtt{IG} \leq \mathtt{IG}'$. From Lem. 2.4 part (iv), we have that $\mathtt{IP}' \leq \mathtt{IG}$. From transitivity, $\mathtt{IP}' \leq \mathtt{IG}'$.

Next we show that $\mathtt{IG} = \mathtt{Raw} \Rightarrow \mathit{isTransitive}(\mathtt{e}_0', \Gamma, \mathtt{C'<MO,IP'>})$. If $\mathtt{IG} = \mathtt{Raw}$ then we showed that $\mathit{isTransitive}(\bot, \Gamma, \mathtt{C<MO,IP>})$. From Lem. 2.3 we have that $\mathtt{IP}' = \mathtt{IP}$, thus $\mathtt{IG} = \mathtt{Raw} \Rightarrow \mathit{isTransitive}(\mathtt{e}_0', \Gamma, \mathtt{C'<MO,IP'>})$.

Let

$$\begin{aligned} \mathit{mtype}(\mathtt{e}_0, \mathtt{m}, \mathtt{C<MO,IP>}) &= \overline{\mathtt{U}} \to \mathtt{V} \\ \mathit{mtype}(\mathtt{e}_0', \mathtt{m}, \mathtt{C'<MO,IP'>}) &= \overline{\mathtt{U}'} \to \mathtt{V}' \\ \Gamma &\vdash \overline{\mathtt{e}} : \overline{\mathtt{U}''} \end{aligned}$$

Because $\Gamma \vdash \overline{\mathtt{U}''} \leq \overline{\mathtt{U}}$, from Lem. 2.8, we have that $\Gamma \vdash \overline{\mathtt{U}''} \leq \overline{\mathtt{U}'}$.

Therefore, all the assumptions in T-Invoke are fulfilled (the requirement for wildcards is fulfilled because all types are closed), and we proved that $\Gamma_{H'} \vdash \grave{\mathtt{e}} : \grave{\mathtt{T}}$.

**Congruence for method argument** Trivial.

**Congruence for new instance** Trivial.

**Congruence for the rvalue of field assignment** Trivial.

**Congruence for the receiver of field assignment** Consider the congruence rule for the receiver of field assignment

$$\frac{H, \mathtt{e} \to H', \mathtt{e}'}{H, \mathtt{e.f=e"} \to H', \mathtt{e}'\mathtt{.f=e"}}$$

We assumed that $\Gamma_H \vdash \mathtt{e.f=e"} : \mathtt{T}$" and by induction $\Gamma_H \vdash \mathtt{e} : \mathtt{T}$, $\Gamma_{H'} \vdash \mathtt{e}' : \mathtt{T}'$ and $\Gamma_{H'} \vdash \mathtt{T}' \leq \mathtt{T}$. We will show that $\Gamma_H \vdash \mathtt{e.f=e"} : \mathtt{T}$". From T-Field-Assignment:

$$\Gamma \vdash \mathtt{e.f} : \mathtt{F} \quad \Gamma \vdash \mathtt{e"} : \mathtt{T"} \quad \Gamma \vdash \mathtt{T"} \leq \mathtt{F} \quad \Gamma \vdash \mathtt{e} : \mathtt{C<MO,IP>} \quad \Gamma \vdash \mathtt{IP} \leq \mathtt{Raw} \quad \mathit{isTransitive}(\mathtt{e}, \Gamma, \mathtt{C<MO,IP>})$$

We need to show that:

$$\Gamma \vdash \mathtt{e}'\mathtt{.f} : \mathtt{F}' \quad \Gamma \vdash \mathtt{T"} \leq \mathtt{F}' \quad \Gamma \vdash \mathtt{e}' : \mathtt{C'<MO,IP'>} \quad \Gamma \vdash \mathtt{IP}' \leq \mathtt{Raw} \quad \mathit{isTransitive}(\mathtt{e}, \Gamma, \mathtt{C'<MO,IP'>})$$

Because `e` was reduced, we know it is not a location, so $\mathit{isTransitive}(\bot, \Gamma, \mathtt{C<MO,IP>})$. From Lem. 2.3, we have that $\mathtt{IP} = \mathtt{IP}'$. Therefore $\mathtt{F}' = \mathtt{F}$ (because $\mathit{ftype}(\mathtt{f},\mathtt{C}) = \mathit{ftype}(\mathtt{f},\mathtt{C}')$), and $\mathit{isTransitive}(\mathtt{e}, \Gamma, \mathtt{C'<MO,IP'>})$.

**Congruence for return R-C1** Consider the congruence rule for `e;return l`

$$\frac{H" = H[K \mapsto H[K] \cup \{\mathtt{l}\}] \quad H", \mathtt{e} \to \grave{H}, \mathtt{e}' \quad H' = \grave{H}[K \mapsto \grave{H}[K] \setminus \{\mathtt{l}\}]}{H, \mathtt{e;return\ l} \to H', \mathtt{e}'\mathtt{;return\ l}}$$

We assumed that

$$\Gamma_H \vdash \texttt{e;return l} : \texttt{T}" \quad \texttt{e}" = \texttt{e;return l} \quad \grave{\texttt{e}} = \texttt{e}'\texttt{;return l} \quad \Gamma_{H"} \vdash \texttt{e} : \texttt{T} \quad \Gamma_{\grave{H}} \vdash \texttt{e}' : \texttt{T}' \quad \Gamma_{\grave{H}} \vdash \texttt{T}' \leq \texttt{T}$$

We need to prove that $\Gamma_{H'} \vdash \grave{\texttt{e}} : \grave{\texttt{T}}$ and $\Gamma_{H'} \vdash \grave{\texttt{T}} \leq \texttt{T}"$.

From Lem. 5.2, we have $H[K] = H'[K]$ and $H"[K] = \grave{H}[K]$. Thus $H'[K] \cup \{\texttt{l}\} = \grave{H}[K]$. According to T-RETURN:

$$\frac{\Gamma_{H'}[K \mapsto \Gamma_{H'}[K] \cup \{\texttt{l}\}] \vdash \texttt{e}' : \texttt{T}'}{\Gamma_{H'} \vdash \texttt{e}'\texttt{;return l} : \Gamma_{H'}(\texttt{l})}$$

Because $H'[K] \cup \{\texttt{l}\} = \grave{H}[K]$, we have that $\Gamma_{H'}[K \mapsto \Gamma_{H'}[K] \cup \{\texttt{l}\}] = \Gamma_{\grave{H}}$. Because $\Gamma_{\grave{H}} \vdash \texttt{e}' : \texttt{T}'$, we proved that $\Gamma_{H'} \vdash \texttt{e}'\texttt{;return l} : \Gamma_{H'}(\texttt{l})$, i.e., $\Gamma_{H'} \vdash \grave{\texttt{e}} : \grave{\texttt{T}}$.

We still need to prove that $\Gamma_{H'} \vdash \grave{\texttt{T}} \leq \texttt{T}"$. Because $\grave{\texttt{T}} = \Gamma_{H'}(\texttt{l})$ and $\texttt{T}" = \Gamma_H(\texttt{l})$ then $\grave{\texttt{T}} = \texttt{T}"$, and from reflexivity (S2) we have $\Gamma_{H'} \vdash \grave{\texttt{T}} \leq \texttt{T}"$.

**Rule R-RETURN** Trivial

**Rule R-NEW** According to R-NEW

$$\frac{\texttt{l} \notin \mathrm{dom}(H) \quad \texttt{VI}' = \begin{cases} \texttt{Immut}_{\texttt{l}} & \text{if } \texttt{VI} = \texttt{Immut} \text{ or } (\texttt{VI} = \texttt{Immut}_{\texttt{C}} \text{ and } \texttt{c} \notin H[K]) \\ \texttt{VI} & \text{otherwise} \end{cases} \quad H' = H[\texttt{l} \mapsto \texttt{C<NO,VI'>}(\overline{\texttt{null}})]}{H, \texttt{new C<NO,VI>}(\overline{\texttt{v}}) \rightarrow H', \texttt{l.build}(\overline{\texttt{v}})\texttt{;return l}}$$

We assumed that

$$\Gamma_H \vdash \texttt{e}" : \texttt{T}" \quad \texttt{e}" = \texttt{new C<NO,VI>}(\overline{\texttt{v}}) \quad \grave{\texttt{e}} = \texttt{l.build}(\overline{\texttt{v}})\texttt{;return l}$$

We need to prove that $\Gamma_{H'} \vdash \grave{\texttt{e}} : \grave{\texttt{T}}$ and $\Gamma_{H'} \vdash \grave{\texttt{T}} \leq \texttt{T}"$.

From T-NEW

$$\frac{\mathit{mtype}(\bot, \texttt{build}, \texttt{C<NO,VI>}) = \overline{\texttt{U}} \rightarrow \texttt{Z} \quad \Gamma_H \vdash \overline{\texttt{v}} : \overline{\texttt{V}} \quad \Gamma_H \vdash \overline{\texttt{V}} \leq \overline{\texttt{U}}}{\Gamma_H \vdash \texttt{new C<NO,VI>}(\overline{\texttt{v}}) : \texttt{C<NO,VI>}} \tag{38}$$

Thus, $\texttt{T}" = \texttt{C<NO,VI>}$. From T-RETURN, $\grave{\texttt{T}} = \texttt{C<NO,VI'>}$. Because $\texttt{l} \notin \Gamma_{H'}[K]$, then $\Gamma_{H'} \vdash \texttt{Immut}_{\texttt{l}} \leq \texttt{Immut}$, thus $\Gamma_{H'} \vdash \grave{\texttt{T}} \leq \texttt{T}"$.

We still need to prove that $\Gamma_{H'} \vdash \grave{\texttt{e}} : \grave{\texttt{T}}$, and from T-RETURN we need to prove that

$$\Gamma_{H"} = \Gamma_{H'}[K \mapsto \Gamma_{H'}[K] \cup \{\texttt{l}\}] \quad \Gamma_{H"} \vdash \texttt{l.build}(\overline{\texttt{v}}) : \texttt{Z}$$

Because $\texttt{l}$ is a new location, all the equations in (38) are still true if we replace $\Gamma_H$ with $\Gamma_{H"}$. From T-INVOKE, and because the guard of $\texttt{build}$ is $\texttt{Raw}$:

$$\frac{\Gamma_{H"} \vdash \texttt{l} : \grave{\texttt{T}} \quad \mathit{mtype}(\texttt{l}, \texttt{build}, \grave{\texttt{T}}) = \overline{\texttt{W}} \rightarrow \texttt{Z}' \quad \Gamma_{H"} \vdash \overline{\texttt{v}} : \overline{\texttt{V}} \quad \Gamma_{H"} \vdash \overline{\texttt{V}} \leq \overline{\texttt{W}} \quad \Gamma_{H"} \vdash \texttt{VI}' \leq \texttt{Raw} \quad \mathit{isTransitive}(\texttt{l}, \Gamma, \grave{\texttt{T}})}{\Gamma_{H"} \vdash \texttt{l.build}(\overline{\texttt{v}}) : \texttt{Z}'}$$

Assumption $\mathit{isTransitive}(\texttt{l}, \Gamma, \grave{\texttt{T}})$ holds because $\texttt{l}$ is a location. Because $\texttt{l} \in \Gamma_{H"}[K]$ and $\texttt{VI}$ is either $\texttt{Mutable}$ or $\texttt{Immut}$ or $\texttt{Immut}_{\texttt{l}'}$, then this assumption holds $\Gamma_{H"} \vdash \texttt{VI}' \leq \texttt{Raw}$. The only assumption left to prove is that $\Gamma_{H"} \vdash \overline{\texttt{V}} \leq \overline{\texttt{W}}$. From (38) we have that $\Gamma_{H"} \vdash \overline{\texttt{V}} \leq \overline{\texttt{U}}$. If $\texttt{VI} = \texttt{Mutable}$ then $\overline{\texttt{U}} = \overline{\texttt{W}}$. Otherwise $\texttt{VI} = \texttt{Immut}$, $\grave{\texttt{T}} = \texttt{C<NO,Immut }_{\texttt{l}}\texttt{VI>}$, and we have that

$$\begin{aligned}
&\mathit{mtype}(\texttt{build}, \texttt{C}) = \overline{\texttt{FT}} \rightarrow \texttt{Z"} \\
&\mathit{mtype}(\texttt{l}, \texttt{build}, \texttt{C<NO,Immut }_{\texttt{l}}\texttt{VI>}) = \overline{\texttt{W}} \rightarrow \texttt{Z}' \\
&\mathit{mtype}(\bot, \texttt{build}, \texttt{C<NO,Immut >}) = \overline{\texttt{U}} \rightarrow \texttt{Z} \\
&\texttt{W}_i = [\texttt{NO}/\texttt{O}, \texttt{Immut}_{\texttt{l}}/\texttt{I}]\texttt{FT}_i \\
&\texttt{U}_i = [\texttt{NO}/\texttt{O}, \texttt{Immut}/\texttt{I}]\texttt{FT}_i \\
&\Gamma_{H"} \vdash \texttt{V}_i \leq \texttt{U}_i
\end{aligned}$$

We want to prove that $\Gamma_{H"} \vdash \texttt{V}_i \leq \texttt{W}_i$. If $I(\texttt{FT}_i) \neq \texttt{I}$ then $\texttt{W}_i = \texttt{U}_i$. Because $O(\texttt{FT}_i) \neq \texttt{This}$, then it is either $\texttt{O}$ or $\texttt{World}$, thus, $\theta(\texttt{W}_i)$ is either $\texttt{NO}$ or $\texttt{World}$. Finally note that $\texttt{l} \prec_\theta \texttt{NO}$ (because $\texttt{l} \mapsto \texttt{C<NO,}\ldots\texttt{>}$), and we always have that $\texttt{NO} \preceq_\theta \texttt{World}$, thus $\texttt{l} \prec_\theta \texttt{World}$. According to subtyping rule S13, we have that $\Gamma_{H"} \vdash \texttt{U}_i \leq \texttt{W}_i$, and from transitivity $\texttt{V}_i \leq \texttt{U}_i \leq \texttt{W}_i$.

**Rule R-FIELD-ACCESS** According to R-FIELD-ACCESS

$$\frac{H[\mathtt{l}] = \mathtt{C<NO,NI>}(\overline{\mathtt{v}}) \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{f}}}{H, \mathtt{l.f}_i \rightarrow H, \mathtt{v}_i}$$

We assumed that $\Gamma_H \vdash \mathtt{l.f}_i : \mathtt{T}$", and we need to prove that $\Gamma_H \vdash \mathtt{v}_i : \grave{\mathtt{T}}$ and $\Gamma_H \vdash \grave{\mathtt{T}} \leq \mathtt{T}$". If $\mathtt{v}_i = \mathtt{null}$ then we can choose $\grave{\mathtt{T}} = \mathtt{T}$", otherwise $\mathtt{v}_i \neq \mathtt{null}$, and because the heap is well-typed, then $\Gamma_H \vdash \grave{\mathtt{T}} \leq \mathtt{T}$".

**Rule R-FIELD-ASSIGNMENT** According to R-FIELD-ASSIGNMENT

$$\frac{\cdots}{H, \mathtt{l.f}_i = \mathtt{v}' \rightarrow H', \mathtt{v}'}$$

We assumed that $\Gamma_H \vdash \mathtt{l.f}_i = \mathtt{v}' : \mathtt{T}$", and we need to prove that $\Gamma_{H'} \vdash \mathtt{v}' : \grave{\mathtt{T}}$ and $\Gamma_{H'} \vdash \grave{\mathtt{T}} \leq \mathtt{T}$". Because we did not add any new locations, we have $\Gamma_H = \Gamma_{H'}$. From T-FIELD-ASSIGNMENT, we have that $\Gamma_H \vdash \mathtt{v}' : \mathtt{T}$", i.e., $\grave{\mathtt{T}} = \mathtt{T}$".

**Rule R-INVOKE** According to R-INVOKE

$$\frac{H[\mathtt{l}] = \mathtt{C<NO,NI>}(\ldots) \qquad \mathit{mbody}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}'}{H, \mathtt{l.m}(\overline{\mathtt{v}}) \rightarrow H, [\overline{\mathtt{v}}/\overline{\mathtt{x}}, \mathtt{l}/\mathtt{this}, \mathtt{l}/\mathtt{This}, \mathtt{NO}/\mathtt{O}, \mathtt{NI}/\mathtt{I}]\mathtt{e}'}$$

We assumed that

$$\Gamma_H \vdash \mathtt{e}" : \mathtt{T}" \qquad \mathtt{e}" = \mathtt{l.m}(\overline{\mathtt{v}}) \qquad \grave{\mathtt{e}} = [\overline{\mathtt{v}}/\overline{\mathtt{x}}, \mathtt{l}/\mathtt{this}, \mathtt{l}/\mathtt{This}, \mathtt{NO}/\mathtt{O}, \mathtt{NI}/\mathtt{I}]\mathtt{e}'$$

From T-INVOKE we have that

$$\mathit{mtype}(\mathtt{l},\mathtt{m},\mathtt{C<NO,NI>}) = \overline{\mathtt{T}} \rightarrow \mathtt{T}" \qquad \Gamma_H \vdash \overline{\mathtt{v}} : \overline{\mathtt{T}'} \qquad \Gamma_H \vdash \overline{\mathtt{T}'} \leq \overline{\mathtt{T}} \qquad \mathit{mguard}(\mathtt{m},\mathtt{C}) = \mathtt{IG} \qquad \Gamma_H \vdash \mathtt{NI} \leq \mathtt{IG}$$

We know the method `m` was typed-checked in `C`, i.e.,

$$\begin{aligned} \Gamma &= \{\mathtt{I} : \mathtt{IG}, \overline{\mathtt{x}} : \overline{\mathtt{U}}, \mathtt{this} : \mathtt{C<O,I>}\} \\ \mathit{mtype}(\mathtt{m},\mathtt{C}) &= \overline{\mathtt{U}} \rightarrow \mathtt{FT} \\ \Gamma &\vdash \mathtt{e}' : \mathtt{S} \\ \Gamma &\vdash \mathtt{S} \leq \mathtt{FT} \end{aligned}$$

From the definition of *mtype*:

$$\begin{aligned} \mathit{mtype}(\mathtt{l},\mathtt{m},\mathtt{C<NO,NI>}) &= \mathit{substitute}(\mathtt{l},\mathtt{C<NO,NI>},\mathit{mtype}(\mathtt{m},\mathtt{C})) \\ \mathtt{T}" &= [\mathtt{NO}/\mathtt{O}, \mathtt{NI}/\mathtt{I}, \mathtt{l}/\mathtt{This}]\mathtt{FT} \\ \mathtt{T}_i &= [\mathtt{NO}/\mathtt{O}, \mathtt{NI}/\mathtt{I}, \mathtt{l}/\mathtt{This}]\mathtt{U}_i \end{aligned}$$

We need to prove that $\Gamma_H \vdash \grave{\mathtt{e}} : \grave{\mathtt{T}}$ and $\Gamma_H \vdash \grave{\mathtt{T}} \leq \mathtt{T}$", which follows immediately from Lem. 5.5.

□

**Lemma 5.7.** ***(Well-typed heap preservation)*** *For every closed expression* $\mathtt{e}" \neq \mathtt{v}$*, and a heap* $H$ *that is well-typed for* $\mathtt{e}"$*, if* $\Gamma_H \vdash \mathtt{e}" : \mathtt{T}"$ *and* $H, \mathtt{e}" \rightarrow H', \grave{\mathtt{e}}$*, then* $H'$ *is well-typed for* $\grave{\mathtt{e}}$*.*

*Proof.* Recall that a well-typed heap $H$ satisfies: (i) there is a linear order $\preceq^T$ over $\mathrm{dom}(H)$ such that for every location `l`, $\theta(\mathtt{l}) = \mathtt{World}$ or $\theta(\mathtt{l}) \prec^T \mathtt{l}$, and $I(\mathtt{l}) = \mathtt{Mutable}$ or $\kappa(\mathtt{l}) \preceq^T \mathtt{l}$, and (ii) each non-null field location is a subtype of the declared field type. Recall also that from the definition of a heap $H$, every location `l` in $H$ has the form: (iii) $\mathtt{l} \mapsto \mathtt{C<}\underline{\mathtt{NO,NI}}\mathtt{>}(\overline{\mathtt{v}})$. Also recall that a heap $H$ is well-typed for `e` if $H[K \mapsto H[K] \cup K(\mathtt{e})]$ is well-typed.

Consider the congruence rules, such as

$$\frac{H, \mathtt{e} \rightarrow H', \mathtt{e}'}{H, \mathtt{e}.f \rightarrow H', \mathtt{e}'.f}$$

By the induction hypothesis $H'$ is well-typed.

The only rule that changes $H[K]$ is R-C1:

$$\frac{H' = H[K \mapsto H[K] \cup \{\mathtt{l}\}] \qquad H', \mathtt{e} \to H", \mathtt{e}'}{H, \mathtt{e;return\ l} \to H"[K \mapsto H"[K] \setminus \{\mathtt{l}\}], \mathtt{e}'\mathtt{;return\ l}}$$

We need to prove that $H"[K \mapsto H"[K] \setminus \{\mathtt{l}\}]$ is well-typed for $\mathtt{e}'\mathtt{;return\ l}$, or equivalently that $H"$ is well-typed for $\mathtt{e}'$. Because $H$ is well-typed for $\mathtt{e;return\ l}$, then $H'$ is well-typed for $\mathtt{e}$, and by induction $H"$ is well-typed for $\mathtt{e}'$. From Lem. 5.2, we know that $H"[K] = H'[K]$, thus $\mathtt{l} \in H"[K]$, and we have that $H"$ is well-typed for $\mathtt{e}'\mathtt{;return\ l}$. From Lem. 4.1, $H"[K \mapsto H"[K] \setminus \{\mathtt{l}\}]$ is well-typed for $\mathtt{e}'\mathtt{;return\ l}$.

Rules R-FIELD-ACCESS and R-INVOKE do not change the heap.

Rule R-RETURN does not change the heap nor $H[K]$, however $K(\grave{\mathtt{e}}) = K(\mathtt{e}") \setminus \{\mathtt{l}\}$, According to Lem. 4.1, the resulting heap $H'$ is well-typed for $\grave{\mathtt{e}}$.

Rule R-NEW creates a new object with `null` fields:

$$\mathtt{VI}' = \begin{cases} \mathtt{Immut}_{\mathtt{l}} & \text{if } \mathtt{VI} = \mathtt{Immut} \text{ or } (\mathtt{VI} = \mathtt{Immut}_{\mathtt{C}} \text{ and } \mathtt{c} \notin H[K]) \\ \mathtt{VI} & \text{otherwise} \end{cases} \qquad \mathtt{l} \notin \mathrm{dom}(H) \qquad H' = H[\mathtt{l} \mapsto \mathtt{C<NO,VI'>}(\overline{\mathtt{null}})]$$

The fields of the new object are all `null`, thus fulfilling demand (ii).

We extend the linear order $\preceq^T$ by adding the new location `l` at the end. Its owner `NO` is either `World` or an existing object $\mathtt{l}'$, and either $\mathtt{VI} = \mathtt{Mutable}$ or $\mathtt{VI} = \mathtt{Immut}_{\mathtt{l}'}$ (where $\mathtt{l}'$ is either an existing location or `l`), thus fulfilling demand (i).

Note that $\mathtt{e}" = \mathtt{new\ C<\underline{NO,VI}>}(\overline{\mathtt{v}})$, and because `e"` is closed, then we have that `NO` and `VI` do not contain `O`, `I`, nor `This`. And if $\mathtt{VI} = \mathtt{Immut}$ then it is substituted with $\mathtt{Immut}_{\mathtt{l}}$, thus fulfilling demand (iii). Finally, note that $\grave{\mathtt{e}} = (\mathtt{l.build}(\overline{\mathtt{v}})\mathtt{;return\ l})$, i.e., $K(\grave{\mathtt{e}}) = K(\mathtt{e}") \cup \{\mathtt{l}\}$. However, because `l` is a *new* location, it does not change existing subtype relations (it does not affect existing objects that do not refer to `l`). Therefore, $H'$ is well-typed for $\grave{\mathtt{e}}$.

Finally, in rule R-FIELD-ASSIGNMENT, $\mathtt{e}" = \mathtt{l.f}_i = \mathtt{v}'$, $\grave{\mathtt{e}} = \mathtt{v}'$, and $H' = H[\mathtt{l} \mapsto \mathtt{C<NO,NI>}([\mathtt{v}'/\mathtt{v}_i]\overline{\mathtt{v}})]$. Note that $K(\mathtt{e}") = K(\grave{\mathtt{e}}) = \{\}$, thus a if $H'$ is well typed then it is well-typed for $\grave{\mathtt{e}}$. Because the typing rule T-FIELD-ASSIGNMENT require that:

$$\Gamma_H \vdash \mathtt{l.f} : \mathtt{T} \qquad \Gamma_H \vdash \mathtt{v}' : \mathtt{T}' \qquad \Gamma_H \vdash \mathtt{T}' \leq \mathtt{T}$$

then the heap $H'$ is well-typed. □

What follows is the submitted version of our SPLASH2010/OOPSLA2010 submission.

# Ownership and Immutability in Generic Java

Yoav Zibin Alex Potanin
Paley Li
Victoria University of Wellington
yoav|alex|lipale@ecs.vuw.ac.nz

Mahmood Ali
Massachusetts Institute of Technology
mali@csail.mit.edu

Michael D. Ernst
University of Washington
mernst@cs.washington.edu

## Abstract

The Java language lacks the important notions of *ownership* (an object owns its representation to prevent unwanted aliasing) and *immutability* (the division into mutable, immutable, and readonly data and references). Programmers are prone to design errors such as representation exposure or violation of immutability contracts. This paper presents *Ownership Immutability Generic Java* (OIGJ), a backward-compatible purely-static language extension supporting ownership and immutability. We formally defined the OIGJ typing rules and proved them sound. We also implemented OIGJ and performed case studies on 33,000 lines of code.

Creation of immutable cyclic structures requires a "*cooking phase*" in which the structure is mutated but the outside world cannot observe this mutation. OIGJ uses *ownership* information to facilitate creation of *immutable* cyclic structures, by safely prolonging the cooking phase even after the constructor finishes.

OIGJ is easy for a programmer to use, and it is easy to implement (flow-insensitive, using only 15 rules). Yet, OIGJ is more expressive than previous ownership languages, in the sense that it can type-check more good code. OIGJ can express the factory and visitor patterns, and OIGJ can type-check Sun's `java.util` collections (except for the `clone` method) without refactoring and with only a small number of annotations. Previous work required major refactoring of existing code in order to fit its ownership restrictions. Forcing refactoring of well-designed code is undesirable because it costs programmer effort, degrades the design, and hinders adoption in the mainstream community.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features

***General Terms*** Experimentation, Languages, Theory



***Keywords*** Immutability, Java, Ownership

## 1. Introduction

This paper presents *Ownership Immutability Generic Java* (OIGJ), a simple and practical language extension that expresses both ownership and immutability information. OIGJ is purely static, without any run-time representation. This enables executing the resulting code on any JVM without runtime penalty. Our ideas, though demonstrated using Java, are applicable to any statically typed language with generics, such as C++, C#, Scala, and Eiffel.

OIGJ follows the *owner-as-dominator* discipline [1, 11, 29] where an object cannot leak beyond its owner: outside objects cannot access it. If an object owns its representation, then there are no aliases to its internal state. For example, a `LinkedList` should own all its `Entry` objects (but not its elements); entries should not be exposed to clients, and entries from different lists must not be mixed.

The keyword `private` does not offer such strong protection as ownership, because a careless programmer might write a public method that exposes a private object (a.k.a. representation exposure). Phrased differently, the name-based protection used in Java hides the variable but not the object, as opposed to ownership that ensures proper encapsulation. The key idea in ownership is that representation objects are nested and encapsulated inside the objects to which they belong. Because this nesting is transitive, this kind of ownership is also called *deep* ownership [12].

OIGJ is based on our previous type systems for ownership (OGJ [29]) and immutability (IGJ [36]). Although ownership and immutability may seem like two unrelated concepts, a design involving both enhances the expressiveness of each individual concept. On the one hand, immutability enhanced ownership by relaxing owner-as-dominator to owner-as-modifier [20], i.e., it constrains modification instead of aliasing. On the other hand, the benefits of adding ownership on top of immutability have not been investigated before. One such benefit is easier creation of immutable cyclic data-structures by using ownership information.

Constructing an immutable object must be done with care. An object begins in the *raw* (not fully initialized) state and transitions to the *cooked* state [6] when initialization

is complete. For an immutable object, field assignment is allowed only when the object is *raw*, i.e., the object cannot be modified after it is *cooked*. An immutable object should not be visible to the outside world in its raw state because it would seem to be mutating. The challenge in building an immutable cyclic data-structure is that many objects must be raw simultaneously to create the cyclic structure. Previous work restricted cooking an object to the constructor, i.e., an object becomes cooked when its constructor finishes.

Our key observation is that *an object becomes cooked when its owner's constructor finishes*. More precisely, in OIGJ, a programmer can choose between cooking an object until its constructor finishes, or until its owner becomes cooked. Because the object is encapsulated within its owner, the outside world will not see this cooking phase. By adding ownership information, we can prolong the cooking time to make it easier to create complex data-structures.

Consider, e.g., building an immutable `LinkedList` (Sun's implementation is similar):

```
LinkedList(Collection<E> c) {
  this();
  Entry<E> succ = this.header, pred = succ.prev;
  for (E e : c) {
    Entry<E> entry = new Entry<E>(e,succ,pred);
    // It's illegal to change an entry after it's cooked.
    pred.next = entry;  pred = entry;
  }
  succ.prev = pred;
}
```

An immutable list contains immutable entries, i.e., the fields `next` and `prev` cannot be changed after an entry is cooked. In IGJ and previous work on immutability, an object becomes cooked after its constructor finishes. Because `next` and `prev` are changed after that time, this code is illegal. In contrast, in OIGJ, this code type-checks if we specify that the list (the `this` object) owns all its entries (the entries are the list's representation). The entries will become cooked when their owner's (the list's) constructor finishes, thus permitting the underlined assignments during the list's construction. Therefore, there was no need to refactor the constructor of `LinkedList` for the benefit of OIGJ type-checking.

Informally, OIGJ provides the following ownership and immutability guarantees. Let $\theta(o)$ denote the owner of $o$ and let $\preceq_\theta$ denote the ownership tree, i.e., the transitive, reflexive closure of $o \preceq_\theta \theta(o)$.

**Ownership guarantee:** An object $o'$ can point to object $o$ iff $o' \preceq_\theta \theta(o)$, i.e., $o$ is owned by $o'$ or by one of its owners.

**Immutability guarantee:** An immutable object cannot be changed after it is cooked.

***Contributions*** The main contributions of this paper are:

**Simplify ownership concepts** OIGJ expresses ownership concepts without introducing new mechanisms, by using Java's underlying generic mechanisms. Specifically, owner-polymorphic methods and scoped regions [33] are implemented using *generic methods*, and existential owners [34] are implemented using *generic wildcards*. By contrast, previous work used new mechanisms that are orthogonal to generics.

**No refactoring of existing code** Java's collection classes (`java.util`) are properly encapsulated. We have implemented OIGJ, and verified the encapsulation by running the OIGJ type-checker without changing the source code (except the `clone` method). Verifying Sun's `LinkedList` requires only 3 ownership annotations (see Sec. 4). Previous approaches to ownership or immutability required major refactoring of this codebase. Refactoring of well-designed code costs programmer effort, results in worse design, and hinders adoption in the mainstream community.

**Flexibility** As illustrated by our case study, OIGJ is more flexible and practical than previous type systems. For example, OIGJ can type-check the factory and visitor design patterns (see Sec. 2), but other ownership languages cannot [22]. Another reason that OIGJ can type-check more good code is that it uses ownership information to facilitate creating immutable cyclic structures by prolonging their cooking phase.

**Formalization** We define a calculus named Featherweight OIGJ (FOIGJ) that formalizes the concepts of raw/cooked objects and wildcards as owner parameters. We prove FOIGJ is sound and provides our ownership and immutability guarantees.

**Outline.** Sec. 2 presents the OIGJ syntax, typing rules, and examples of use, such as the factory and visitor patterns. Sec. 3 defines the OIGJ formalization. Sec. 4 discusses the OIGJ implementation and the collections case study. Sec. 5 compares OIGJ to related work, and Sec. 6 concludes.

## 2. OIGJ Language

This section presents the OIGJ language extension that expresses both ownership and immutability information. We first describe the OIGJ syntax (Sec. 2.1). We then proceed with a `LinkedList` class example (Sec. 2.2), followed by the OIGJ typing rules (Sec. 2.3). We conclude by showing the factory (Sec. 2.4) and visitor (Sec. 2.5) design patterns in OIGJ.

### 2.1 OIGJ Syntax

OIGJ introduces two new type parameters to each type, called the *owner parameter* and the *immutability parameter*. For simplicity of presentation, in the rest of this paper we assume that the special type parameters are at the beginning of the list of type parameters. We stress that generics in Java are erased during compilation to bytecode and do not exist at runtime, therefore OIGJ does not incur any runtime overhead (nor does it support run-time casts).

In OIGJ, all classes are subtypes of the parameterized root type `Object<O,I>` that declares an owner and an immutabil-

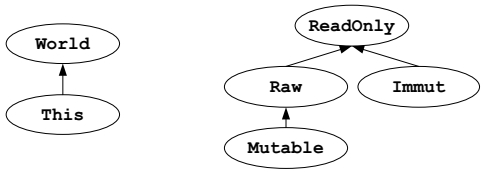


**Figure 1.** The type hierarchy of (a) ownership and (b) immutability parameters. `World` means the entire world can access the object, whereas `This` means that `this` owns the object and no one else can access it. The meaning of `Immut`/`Mutable` is obvious. A `ReadOnly` reference points to a mutable or immutable object, and therefore cannot be used to mutate the object. `Raw` represents an object under construction whose fields can be assigned.

ity parameter. All subclasses must invariantly preserve their owner and immutability parameter. The owner and immutability parameters form two separate hierarchies, which are shown in Fig. 1. These parameters cannot be extended, and they have no subtype relation with any other types. The subtyping relation is denoted by $\leq$, e.g., `Mutable` $\leq$ `ReadOnly`.

OIGJ syntax is based on *conditional Java* (cJ) [17], where a programmer can write method *guards*. A guard of the form `<X extends Y>? METHOD_DECLARATION` has a dual meaning: (i) the method is applicable only if the type argument that substituted `X` extends `Y`, and (ii) the bound of `X` inside `METHOD_DECLARATION` changes to `Y`.

***Class definition example*** Fig. 2 shows an example of OIGJ syntax. A class definition declares the owner and immutability parameters (line 1); by convention we always denote them by `O` and `I` and they always extend `World` and `ReadOnly`. If the `extends` clause is missing from a class declaration, then we assume it extends `Object<O,I>`.

***Immutability example*** Lines 2–4 show different kinds of immutability in OIGJ: immutable, mutable, and readonly. A readonly and an immutable reference may seem similar at first because neither can be used to mutate the referent. However, line 4 shows the difference between the two: a readonly reference may point to a mutable object. Phrased differently, a readonly reference may not mutate its referent, though the referent may be changed via an aliasing mutable reference.

Java's type arguments are invariant (neither covariant nor contravariant), to avoid a type loophole [18], so line 4 is illegal in Java. Line 4 is legal in OIGJ, because OIGJ safely allows covariant changes in the immutability parameter (but not in the owner parameter). Therefore, neither OIGJ nor Java subsumes the other, i.e., a legal OIGJ program may be illegal in Java (and vice versa). However, because generics are erased during compilation, the resulting byte-code can be executed on any JVM.

The immutability of `sameD` (line 5) depends on the immutability of `this`, i.e., it is (im)mutable in an (im)mutable

```
1:class Foo<O extends World,I extends ReadOnly> {
2:  // An immutable reference to an immutable date.
    Date<O,Immut> imD = new Date<O,Immut>();
3:  // A mutable reference to a mutable date.
    Date<O,Mutable> mutD = new Date<O,Mutable>();
4:  // A readonly reference to any date. Both roD and imD cannot
    mutate their referent, however the referent of roD might be
    mutated by an alias, whereas the referent of imD is immutable.
    Date<O,ReadOnly> roD = ... ? imD : mutD;
5:  // A date with the same owner and immutability as this
    Date<O,I> sameD;
6:  // A date owned by this; it cannot leak.
    Date<This,I> ownedD;
7:  // Anyone can access this date.
    Date<World,I> publicD;
8:  // Can be called on any receiver; cannot mutate this. The
    method guard "<...>?" is part of cJ's syntax [17].
    <I extends ReadOnly>? int readonlyMethod(){...}
9:  // Can be called only on mutable receivers; can mutate this.
    <I extends Mutable>? void mutatingMethod(){...}
10: // Constructor that can create (im)mutable objects.
    <I extends Raw>? Foo(Date<O,I> d){
11:   this.sameD = d;
12:   this.ownedD = new Date<This,I>();
13:   // Illegal, because sameD came from the outside.
      // this.sameD.setTime(...);
14:   // OK, because Raw is transitive for owned fields.
      this.ownedD.setTime(...);
15: }
16:}
```

**Figure 2.** An example of OIGJ syntax.

`Foo` object. Similarly, the owner of `sameD` is the same as the owner of `this`.

***Ownership example*** Lines 5–7 show three different owner parameters: `O`, `This`, and `World`. The owner parameter is invariant, i.e., the subtype relation preserves the owner parameter. For instance, the types on lines 5–7 have no subtype relation with each other because they have different owner parameters.

Reference `ownedD` cannot leak outside of `this`, whereas references `sameD` and `publicD` can potentially be accessed by anyone with access to `this`. The difference between `sameD` and `publicD` is that `publicD` can be stored anywhere on the heap (even in a static public variable) whereas `sameD` can only be stored inside its owner.

We use $O(\ldots)$ to denote the function that takes a type or a reference, and returns its owner parameter, e.g., $O(\texttt{ownedD}) = \texttt{This}$. Similarly, function $I(\ldots)$ returns the immutability parameter, e.g., $I(\texttt{ownedD}) = \texttt{I}$. We say that an object `o` is `this`-*owned* (i.e., owned by `this`) if $O(\texttt{o}) = \texttt{This}$, e.g., `ownedD` is `this`-owned, but `sameD` is not. OIGJ prevents leaking `this`-owned objects by requiring that `this`-owned fields (and methods with `this`-owned arguments or return-type) can only

be used via this. For example, this.ownedD is legal, but foo.ownedD is illegal.

***Owner vs. Owner-parameter*** Now we explain the connection between the *owner parameter* $O(o)$ (which is a generic type parameter at *compile-time*) and the *owner* $\theta(o)$ (which is an object at *run-time*). World is a special owner that represents the root of the ownership tree, and This represents an owner that is the current this object. Formally, if $O(o) =$ This then $\theta(o) =$ this, if $O(o) =$ O then $\theta(o) = \theta($this$)$, and if $O(o) =$ World then $\theta(o) =$ World (we treat World both as a type parameter and as an object that is the root of the ownership tree). Two references (in the same class) with the same owner parameter (at compile-time) will point to objects with the same owner (at run-time), i.e., $O(o_1) = O(o_2)$ iff $\theta(o_1) = \theta(o_2)$.

Finally, recall the **Ownership guarantee**: $o'$ can point to $o$ iff $o' \preceq_\theta \theta(o)$. By definition of $\preceq_\theta$, we have that for all $o$: (i) $o \preceq_\theta o$, (ii) $o \preceq_\theta \theta(o)$, and (iii) $o \preceq_\theta$ World. By part (iii), if $\theta(o) =$ World then anyone can point to $o$. On lines 5–7, we see that this can point to ownedD, sameD, publicD, whose owner parameters are This, O, World, and whose owners are this, $\theta($this$)$, World. This conforms with the ownership guarantee according to parts (i), (ii), and (iii), respectively. More complicated pointing patterns can occur (e.g., this can point to its owner's owner) by using multiple owner parameters, such as in List<This,I,Date<O,I>>.

There is a similar connection between the immutability parameter (at compile-time) and the object's immutability (at run-time). Immutability parameter Mutable or Immut implies the object is mutable or immutable (respectively), ReadOnly implies the referenced object may be either mutable or immutable and thus the object cannot be mutated. Raw implies the object is still raw and thus can still be mutated, but it might become immutable after it is cooked.

***Method guard example*** Lines 8 and 9 show a readonly and a mutating method. Note how these methods are *guarded* with <...>?. Conditional Java (cJ) [17] extends Java with such guards (a.k.a. conditional type expressions). Note that cJ changed Java's syntax by using the question mark in the guard <...>?. The exposition in this paper uses cJ for convenience. However, our implementation of OIGJ (Sec. 4) uses type-annotations without changing Java's syntax, for conciseness and compatibility with existing tools and code bases.

A guards such as <T extends U>? METHOD_DECLARATION has a dual purpose: (i) the method is included only if T extends U, and (ii) the bound of T is U inside the method. In our example, the guard on line 9 means that (i) this method can only be called on a Mutable receiver, and (ii) inside the method the bound of I changes to Mutable. For instance, (i) only a mutable Foo object can be a receiver of mutatingMethod, and (ii) field sameD is mutable in mutatingMethod. cJ also ensures that the condition of an overriding method is equivalent or weaker than the condition of the overridden method.

IGJ [36] used *declaration annotations* to denote the immutability of this. In this paper, OIGJ uses cJ to reduce the number of typing rules and handle inner classes more flexibly. (Our implementation uses *type annotations* to denote immutability of this.) OIGJ does not use the full power of cJ, i.e., it only uses guards with immutability parameters. Moreover, we modified cJ to treat guards over constructors in a special way described below in the **Object creation rule** of Fig. 4.

To summarize, on lines 8–10 we see three guards that change the bound of I to ReadOnly, Mutable, and Raw, respectively. Because the bound of I is already declared on line 1 as ReadOnly, the guard on line 8 can be removed without changing the example.

***Constructor example*** The constructor on line 10 is guarded with Raw, and therefore can create both mutable and immutable objects, because all objects start their life cycle as raw. This constructor shows the interplay between *ownership* and *immutability*, which makes OIGJ more expressive than previous work on immutability. By using ownership information, OIGJ can prolong the *cooking phase* for owned objects, i.e., the cooking phase of this-owned fields (ownedD) is longer than that of non-owned fields (sameD). This property is critical to type-check the collection classes, as Sec. 2.2 will show.

Consider the following code:

```
Date<O,Immut> d = new Date<O,Immut>();
Foo<O,Immut> foo = new Foo<O,Immut>(d);
```

Recall our **Immutability guarantee:** An immutable object cannot be changed after it is *cooked*. An object is cooked either when its owner is cooked (for foo.ownedD) or, if it is not owned by This, when its constructor finishes (for d and foo). The intuition is that ownedD cannot leak and therefore the outside world cannot observe this longer cooking phase, whereas d is visible to the world after its constructor finishes and cannot be mutated further. The constructor on lines 10–15 shows this difference between the assignments to sameD (line 11) and to ownedD (line 12): sameD can come from the outside world, whereas ownedD must be created inside this. Therefore, sameD cannot be further mutated (line 13) whereas ownedD can be mutated (line 14) until its owner is cooked.

An object in a raw method whose immutability parameter is I is still considered raw (thus we can still assign to its fields or call other raw methods) iff the object is this or this-owned. Informally, we say that Raw is *transitive* only for this or this-owned objects. For example, the receiver of the method call sameD.setTime(...) is not this nor this-owned, and therefore the call is illegal, however the receiver of ownedD.setTime(...) is this-owned, and therefore the call is legal.

### 2.2 `LinkedList` example

Fig. 3 shows an implementation of LinkedList in OIGJ that is similar in spirit to Sun's implementation. We explain this

example in three stages: (i) we first explain the data-structure, i.e., the fields of a list and its entries (lines 1–6), (ii) then we discuss the `Raw` constructors that enables creation of immutable lists (lines 7–24), and (iii) finally we dive into the complexities of inner classes and iterators (lines 26–53).

***LinkedList data-structure*** A linked list has a header field (line 6) pointing to the first entry, where each entry has an `element` and pointers to the `next` and `prev` entries (line 3). We explain first the immutability and then the ownership of each field.

Recall that we implicitly assume that `O` extends `World` and that `I` extends `ReadOnly` on lines 1, 5, and 49.

An (im)mutable list contains (im)mutable entries, i.e., the entire data-structure is either mutable or immutable as a whole. Therefore, all the fields have the same immutability `I`. The underlying generic type system propagates the immutability information without the need for special typing rules.

Next consider the ownership of the fields of `LinkedList` and `Entry`. `This` on line 6 expresses that the reference `header` points to an `Entry` owned by `this`, i.e., the entry is encapsulated and cannot be aliased outside of `this`. `O` on line 3 expresses that the owner of `next` is the same as the owner of the entry, i.e., a linked-list owns *all* its entries. Note how the generics mechanism propagates the owner parameter, e.g., the type of `this.header.next.next` is `Entry<This,I,E>`. Thus, the owner of all entries is the `this` object, i.e., the list.

Finally, note that the field `element` has no immutability nor owner parameters, because they will be specified by the client of the list, e.g.,
`LinkedList<This,Mutable,Date<World,ReadOnly>>`

***Immutable object creation*** A constructor that is making an immutable object must be able to set the fields of the object. It is not acceptable to mark such constructors as mutable (`<I extends Mutable>?`), which would permit arbitrary side effects, possibly including making mutable aliases to `this`. OIGJ uses a fourth kind of reference immutability, `Raw`, to permit constructors to perform limited side effects without permitting modification of immutable objects. Phrased differently, `Raw` represents a partially-initialized *raw* object that can still be mutated, but after it is cooked (fully initialized), then the object might become immutable. The constructors on lines 7 and 11 are guarded with `Raw`, and therefore can create both mutable and immutable lists.

Objects must not be captured in their raw state to prevent further mutation after the object is cooked. If a programmer could declare a field, such as `Date<O,Raw>`, then a raw date could be stored there, and later it could be used to mutate a cooked immutable date. Therefore, a programmer can write the `Raw` type only after the `extends` keyword, but *not* in any other way. As a consequence, in a `Raw` constructor, `this` can only escape as `ReadOnly`.

Recall that an object becomes *cooked* either when its constructor finishes or when its owner is *cooked*. The entries

```
 1: class Entry<O,I,E> {
 2:   E element;
 3:   Entry<O,I,E> next, prev;  ...
 4: }
 5: class LinkedList<O,I,E> {
 6:   Entry<This,I,E> header;
 7:   <I extends Raw>? LinkedList() {
 8:     this.header = new Entry<This,I,E>();
 9:     this.header.next = this.header; ...
10:   }
11:   <I extends Raw>? LinkedList(
12:                  Collection<?,ReadOnly,E> c) {
13:     this();  this.addAll(c);
14:   }
15:   <I extends Raw>? void addAll(
16:                  Collection<?,ReadOnly,E> c) {
17:     Entry<This,I,E> succ = this.header,
18:                     pred = succ.prev;
19:     for (E e : c) {
20:       Entry<This,I,E> en =
21:         new Entry<This,I,E>(e,succ,pred);
22:       pred.next = en;  pred = en;  }
23:     succ.prev = pred;
24:   }
25:   int size() {...}
26:   <ItrI extends ReadOnly> Iterator<O,ItrI,I,E>
27:                  iterator() {
28:     return this.new ListItr<ItrI>();
29:   }
30:   void remove(Entry<This,Mutable,E> e) {
31:     e.prev.next = e.next;
32:     e.next.prev = e.prev;
33:   }
34:   class ListItr<ItrI> implements
35:         Iterator<O,ItrI,I,E> {
36:     Entry<This,I,E> current;
37:     <ItrI extends Raw>? ListItr() {
38:       this.current = LinkedList.this.header;
39:     }
40:     <ItrI extends Mutable>? E next() {
41:       this.current = this.current.next;
42:       return this.current.element;
43:     }
44:     <I extends Mutable>? void remove() {
45:       LinkedList.this.remove(this.current);
46:     }
47:   }
48: }
49: interface Iterator<O,ItrI,CollectionI,E> {
50:   boolean hasNext();
51:   <ItrI extends Mutable>? E next();
52:   <CollectionI extends Mutable>? void remove();
53: }
```

**Figure 3.** `LinkedList<O,I,E>` in OIGJ.

of the list (line 6) are of type `Entry<This,I,E>`, and `this`-owned objects become cooked when their owner is cooked. Indeed, the entries are mutated after their constructor finished, but before the list is cooked, on lines 9 and 22. This shows the power of combining immutability and ownership: we are able to create immutable lists *only* by using the fact that the list owns its entries. If those entries were *not* owned by the list, then this mutation of entries might be visible to the outside world, thus breaking the guarantee that an immutable object never changes. By enforcing ownership, OIGJ ensures that such illegal mutations cannot occur.

Note that we always access and assign to `header` via `this` (lines 8, 9, 17, and 38). OIGJ requires that all access and assignment to a `this`-owned field (such as `header`) must be done via `this`. In contrast, fields `next` or `prev` (which are not `this`-owned) do not have such restriction, as can be seen on lines 31–32.

***Iterator implementation and inner classes*** An *iterator* has an underlying *collection*, and the immutability of these two objects might be different. For example, you can have a mutable iterator over a readonly collection (i.e., the iterator supports `next()` but not `remove()`), or a readonly iterator over a mutable collection (i.e., the iterator supports `remove()` but not `next()`). Consider the `Iterator<O,ItrI,CollectionI,E>` interface defined on lines 49–53, and used on lines 26 and 35. `ItrI` is the iterator's immutability, whereas `CollectionI` is intended to be the underlying collection's immutability (see on line 35 how the collection's immutability `I` is used in the place of `CollectionI`). Line 51 requires a mutable `ItrI` in order to call `next()`, and line 52 requires a mutable `CollectionI` to call `remove()`.

Inner class `ListItr` (lines 34–47) is the implementation of `Iterator` for list. It reuses the owner parameter `O` from `LinkedList`, but declares a new immutability parameter `ItrI`, and uses them on line 34 in `Iterator<O,ItrI,I,E>`. Observe how `ListItr` and `LinkedList` have the same owner `O`, but different immutability parameters (`ItrI` for `ListItr`, and `I` for `LinkedList`). `ListItr` use the same owner `O` because it directly accesses the (`this`-owned) representation of the collection (line 38), which would be illegal if their owner was different. For example, consider the types of `this` and `LinkedList.this` on line 38:

```
Iterator<O,ItrI,...> thisIterator = this;
LinkedList<O,I,...> thisList = LinkedList.this;
```

Because the bound of `ItrI` is `Raw` (but bound of `I` is `ReadOnly`), `this` can be mutated (but `LinkedList.this` cannot).

In general, an inner class must have a distinct immutability parameter, but it must reuse the owner parameter of its outer class. This design decision is related to the fact that the immutability typing rules do not use `this`, whereas the ownership typing rules use `this` when checking if field assignment or access are legal. In fact, the semantics of `This` is related to `this`, i.e., it means that the object is `this`-owned (regardless of whether `this` is an instance of the outer or inner class). We could have several `This` types, e.g., `LinkedList.This` vs. `ListItr.This`, but this would complicate the typing rules.

Notice the difference in syntax between a generic method on line 26 and a method guard (that ends in a question-mark) on line 15.

Finally, consider the creation of a new *inner* object on line 28 using `this.new ListItr<ItrI>()`. This expression is type-checked both as a method call (whose receiver is `this`) and as a constructor call. Observe that the bound of `ItrI` is `ReadOnly` (line 26) and the guard on the constructor is `Raw` (line 37), which is legal because a `Raw` constructor can create both mutable and immutable objects.

### 2.3 OIGJ typing rules

Fig. 4 contains all the OIGJ typing rules. In what follows we provide more detailed discussion of each of the rules. A formal type system based on a simplified version of these rules is the subject of Sec. 3. Some of the rules are identical to those found in OGJ [29] and IGJ [36] (see Sec. 5 for a comparison with OIGJ).

***Ownership nesting*** Consider the following example:

```
List<This,I,Date<World,I>> l1; // Legal nesting
List<World,I,Date<This,I>> l2; // Illegal!
```

Definition of `l2` has illegal ownership nesting because owned dates might leak, e.g., we can store `l2` in this variable:

```
public static Object<World,ReadOnly> publicAliasToL2;
```

In general, the owner parameter of type `T` must be lower or equal in the ownership tree compared to any other owner in `T`.

***Field access*** This rule enforces ownership: `this`-owned fields can be assigned only via `this`. For example, note that all accesses (and assignments) to `header` are done via `this`.

***Field assignment*** Assigning to a field should respect both immutability and ownership constraints. Part (i) of the rule enforces immutability constraints: A field can be assigned only by a `Mutable` or `Raw` reference. Part (ii) ensures `Raw` is transitive only for `this` or `this`-owned objects. Part (iii) enforces ownership constraints as in field access.

For example, consider the assignments on lines 8 and 9 of Fig. 3. Note that the bound of `I` is `Raw`, thus the assignments satisfy part (i). Part (ii) holds, i.e., `Raw` is transitive in the first assignment because the target object is `this` and in the second assignment because it is `this`-owned (the type of `this.header` is `Entry<This,I,E>`). Finally, part (iii) holds in the first assignment because `header` was assigned via `this`, and in the second assignment because field `next` (`Entry<O,I>`) is not `this`-owned.

***Method invocation*** Method invocation is handled in the same way as field access/assignment: parts (i) and (ii) are similar to field access and field assignment part (ii). For example, consider the following method: `R m(A a) { ... }`
Then, the method call `o.m(e)` is handled as if there is an assignment to a field of type `A`, and the return value is typed as if there was an access to a field of type `R`. Note that regarding

**Ownership nesting** The main owner parameter $O(\texttt{T})$ must be *inside* any other owner parameter in `T`.

**Field access** Field access `o.f` is legal iff $\underline{O(\texttt{f}) = \texttt{This} \Rightarrow \texttt{o} = \texttt{this}}$.

**Field assignment** Field assignment `o.f=...` is legal iff (i) $I(\texttt{o}) \leq \texttt{Raw}$, and (ii) $\big(\underline{I(\texttt{o}) = \texttt{Raw} \Rightarrow (\texttt{o} = \texttt{this} \text{ or } O(\texttt{o}) = \texttt{This})}\big)$, and (iii) field access `o.f` is legal.

**Method invocation** Consider method `T0 m(T1,...,Tn)`. The invocation `o.m(...)` is legal iff (i) $\underline{O(\texttt{T}_i) = \texttt{This} \Rightarrow \texttt{o} = \texttt{this}}$ for $i = 0,\ldots,n$, and (ii) $I(\texttt{m}) = \texttt{Raw}$ implies <u>field assignment part (ii)</u>.

**cJ's method guards** (i) An invocation `o.m(...)` is legal if the type of `o` satisfies the guard of `m`. (ii) When typing method `m`, the bound of type variables changes according to the guard. (iii) The guard of an overriding method is equivalent or weaker than that of the overridden method.

**Inner classes** An inner class reuses the owner parameter of the outer class. However, it has a distinct immutability parameter.

**Same-class subtype definition** Let `C<X1,...,Xn>` be a class. Type `S = C<S1,...,Sn>` is *a subtype of* `T = C<T1,...,Tn>`, written as `S ≤ T`, iff `S = T` or all immutability parameters $\texttt{T}_j$ are either `ReadOnly` or `Immut`, and for $i = 1,\ldots,n$, $\texttt{S}_i = \texttt{T}_i$ or ($\texttt{S}_i \leq \texttt{T}_i$ and $\texttt{X}_i$ is not invariant in `C`)).

**Invariant** A type parameter must be invariant if it extends `World`, or if it is used in a superclass that contains `Mutable`, a field that contains `Mutable` but is not `this`-owned, or in the position of another invariant type parameter. The immutability parameter can never be invariant.

**Erased signature** If method $\texttt{m}'$ overrides a readonly/immutable method `m`, then the erased signatures of $\texttt{m}'$ and `m`, excluding invariant type parameters, must be identical. (The *erased signature* of a method is obtained by replacing type parameters with their bounds.)

**Object creation** A constructor cannot have any `this`-owned arguments. Furthermore, `new SomeClass<X,...>(...)` is *legal* iff the constructor's guard `<I extends Y>?` satisfies: `Y = Mutable` and `X = Mutable`, or `Y = Raw`.

**Generic Wildcards** OIGJ prohibits using a generic wildcard (`?`) in the position of the immutability parameter. For the owner parameter, OIGJ prohibits using a wildcard in a field or in a method return type, but permits it for stack variables and method parameters.

**`Raw` parameter** `Raw` cannot be used in the position of a generic parameter. It can only be used after the `extends` keyword.

**Fresh owners** A *fresh owner* is a method owner parameter that is not used in the method signature. It is strictly inside all other owners in scope.

**Static context** `This` cannot be used in a static context, i.e., in static methods or fields.

**Figure 4.** All the OIGJ typing rules, in English. Also see Sec. 3 for a formalization. Underlined sentences show similarities between the first three rules.

the transitivity of `Raw`, we check both the immutability of the receiver object ($I(\texttt{o})$) and that of the method, i.e., its guard ($I(\texttt{m})$). If both are `Raw`, then we require that `o` is either `this` or `this`-owned.

***Inner classes*** Nested classes that are *static* can be treated the same as normal classes. An *inner class* is a non-static nested class, e.g., iterators in `java.util` are implemented using inner classes. An inner class reuses the owner parameter of the outer class, i.e., the inner object is seen as an extension of the outer object. However, it has a distinct immutability parameter. Therefore, both `this` and `OuterClass.this` are treated identically by the typing rules that involve ownership. (Note that the typing rules for immutability never mention "`this`".)

***Subtype relation*** Java is *invariant* in generic arguments, i.e., it prohibits *covariant* (or contravariant) changes. A `Vector<Integer>` is not a subtype of a `Vector<Object>`. If it were, then mutating the vector by inserting, e.g., a `String`, breaks type-safety.

OIGJ permits covariant changes for non-mutable references because the object cannot be mutated in a way that is not type-safe. The full subtype definition of OIGJ includes all of Java's subtyping rules, therefore OIGJ's subtype relation is a superset of Java's subtype relation. We only present a more relaxed *same-class* subtyping rule that allows covariant changes in other type parameters if mutation is disallowed, e.g., `List<O,ReadOnly,`<u>`Integer`</u>`>` is a subtype of `List<O,ReadOnly,`<u>`Number`</u>`>`. Note that covariance is allowed iff *all* immutability parameters of the supertype are `ReadOnly` or `Immut`, e.g., `Iterator<O,ReadOnly,Mutable,Integer>` is *not* a subtype of `Iterator<O,ReadOnly,`<u>`Mutable`</u>`,Number>`, but it is a subtype of `Iterator<O,ReadOnly,`<u>`ReadOnly`</u>`,Number>`.

***Invariant*** A user can annotate a type parameter `X` in class `C` with `@InVariant` to prevent covariant changes, in which case we say that `X` is invariant. Otherwise we say that `X` is covariant. The immutability parameter must be allowed to change covariantly, or else a mutable reference could not be a receiver when calling a readonly method.

A type parameter must be invariant if it is an owner parameter, if it is used in a field/superclass that contains `Mutable`, or if the erased signature differs. For example, if a class has a field of type `Foo<O,Mutable,X>`, then `X` must be invariant (the owner parameter `O` is always invariant).

***Erased signature*** When the erased signature of an overriding method differs from the overridden method, the normal `javac` compiler inserts a *bridge method* to cast the arguments to the correct type [7]. Such bridge methods work correctly only under the assumptions that subtyping is invariant. For example, consider an integer comparator `intComp` that implements `Comparable<Integer>`. If `Comparable<Integer>` were a subtype of `Comparable<Object>`, then we could pass a `String` to `intComp`'s implementation of `compareTo(Integer)`:

```
((Comparable<Object>)intComp).compareTo("a")
```

OIGJ requires that the *erased signature* of an overriding method remains the same (excluding invariant parameters) if that method is either readonly or immutable. For example, the erased signature of `compareTo` in `intComp` differs from the one in the interface `Comparable<O,I,X>`. Therefore, this rule requires that the type parameter `X` must be invariant:

```
interface Comparable<O,I, @InVariant X> {
  int compareTo(X o); }
```

***Object creation*** A constructor should not have any `this`-owned parameters, because `this`-owned objects can only be created inside `this`. (A private constructor that is called inside `this`, e.g., `this(...);`, could have `this`-owned parameters, but in our view, it is better to replace it with a method.)

Recall that the immutability of a constructor (or any method in general) is defined to be the bound of the immutability parameter in that constructor, e.g., a mutable constructor will have the guard `<I extends Mutable>?`. Note that according to the rules of cJ, calling a `Raw` constructor to create an `Immut` object is illegal because the guard is not satisfied: `Immut` is not a subtype of `Raw`. OIGJ changed cJ and treats constructor calls using this object creation rule: A `Raw` constructor can create any object (both mutable and immutable). A `Mutable` constructor can only create `Mutable` objects. A constructor cannot be `Immut` or `ReadOnly`, because it should be able to assign to the fields of `this`.

***Generic Wildcards*** Java's generics can be bypassed by using reflection or raw types, e.g., `List`. Similarly, one can bypass OIGJ when using these features. OIGJ prohibits wildcards on the owner parameter of *fields*, e.g., `Date<?,ReadOnly> field`, because one can declare a static field of that type and store a `this`-owned date, thus breaking owner-as-dominator. Wildcards on a method return type are also prohibited because they can be used to leak `this`-owned fields. However, wildcards on *stack variables* (e.g., method parameters or local variables) are allowed.

Existential owners [8, 26, 34] are used when the exact owner of an object is unknown. One motivation for existential owners is the downcast performed in the `equals` method [34]. Without existential owners, this downcast requires a runtime check on the owner parameter.

OIGJ uses Java's existing generic wildcard syntax (`?`) to express existential owners. For example, consider the `DateList` class, which is parameterized by its owner parameter (`O`) and the dates' owner parameter (`DO`):

```
class DateList<O,I,DO extends World> {
 boolean equals(Object<?,ReadOnly> o) {
  DateList<?,ReadOnly,?> l =
    (DateList<?,ReadOnly,?>) o;
  return listEquals(l);
 }
 <O2 extends World,DO2 extends World> boolean
    listEquals(DateList<O2,ReadOnly,DO2> l) {...}
}
```

Method `listEquals` shows that it is possible to name the existential owner—the unknown list's owner parameter is `O2` and the unknown dates' owner parameter is `DO2`. Phrased differently, the two wildcards in `DateList<?,ReadOnly,?>` are now named `DateList<O2,ReadOnly,DO2>`.

**`Raw` *parameter*** `Raw` can only be used after the `extends` keyword. For example, it is prohibited to write directly `Date<O,Raw>`. If it were possible, then such a date could leak from a `Raw` constructor that is building an immutable object, and then that immutable object would have an alias that could mutate it.

***Fresh owner*** A *fresh owner* is a method owner parameter that is not used in the method signature. In OIGJ, a fresh owner expresses *temporary ownership* within the method. Specifically, it allows a method to create stack-local objects with access to any object visible at the point of creation, but with a guarantee that stack-locals will not leak. Therefore, stack-local objects can be garbage-collected when the method returns. For example, consider a method that deserializes a `ByteStream` by creating a temporary `ObjectStream` that wraps it.

```
<O,TmpO> void deserialize(ByteStream<O> bs) {
  ObjectStream<TmpO,ByteStream<O>> os = ...     }
```

Note that `TmpO` is a fresh owner, whereas `O` is not. Because `TmpO` is strictly inside other owner parameters such as `O`, there cannot be any aliases from `bs` to `os`. When the method returns, `TmpO` will cease to exist, and the stack-local `os` can be garbage-collected.

Technically, *a fresh owner is strictly inside all other non-fresh owners in scope*, to make sure it cannot exist after the method returns. (Multiple fresh owners are incomparable with each other.) Because a fresh owner is inside several other owners that might be incomparable in the ownership tree, the ownership structure is a DAG rather than a tree.

To type-check temporary ownership and DAG ownership structures, OIGJ adopts Wrigstad's *scoped confinement* [33] ownership model, in which the fresh owners are owned by the current *stack-entry*. Briefly stated, each method invocation pushes a new stack-entry (the first stack-entry corresponds to the static `main` method), which is the root of a new ownership tree. Objects in this new tree may point to objects in previous trees, but not vice versa.

***Static context*** `This` represents that an object is owned by `this`, and therefore OIGJ prohibits using it in a static context, such as static fields or methods. Static fields can use the owner parameter `World`, and static methods can also use generic method parameters that extend `World`. For example, the static method:

```
static <LO extends World,E> void sort(
                    List<LO,Mutable,E> l) { ... }
```

is parameterized by the list's owner `LO`.

### 2.4 Factory method design pattern

The *factory method pattern* [15] is a creational design pattern for creating objects without specifying the exact class of the object that will be created. The solution is to define an interface with a method for creating an object. Implementers can override the method to create objects of a derived type.

The challenge of the factory method pattern with respect to ownership [22] is that the point of *creation* and *usage* are in different classes, and the created object must not be *captured* (stored in the fields of another object) between creation and usage points. For example, consider a `this`-owned object that is *used* within some class, but *created* (and possibly *captured*, thus breaking ownership) outside the class. To solve this problem, previous work suggested sophisticated ownership transfer mechanisms [21] using `capture` and `release`.

```
1: class SafeSyncList<O,I,E> implements List<O,I,E> {
2:   List<This,I,E> l;
3:   <I extends Raw>? SafeSyncList(
4:                      Factory<?,ReadOnly,E> f) {
5:     List<This,I,E> b = f.create();
6:     l = Collections.synchronizedList(b);
7:   }
8:   ... // delegate methods to l
9: }
10: class Collections<O,I> {
11:   static <O2,I2,E> List<O2,I2,E>
12:    synchronizedList(List<O2,I2,E> list)
13:      {  ...  }   // Sun's original implementation
14: }
15: interface Factory<O,I,E> {
16:   <O2,I2> List<O2,I2,E> create();
17: }
18: class LinkedListFactory<O,I,E> implements
19:                 Factory<O,I,E> {
20:   <O2,I2> List<O2,I2,E> create() {
21:     return new LinkedList<O2,I2,E>();
22:   }
23: }
```

**Figure 5.** Factory method design pattern in OIGJ. OIGJ guarantees that the backing list `b` (line 5) is never accessed directly, e.g., it cannot be captured on line 21.

OIGJ solves this problem without inventing a new mechanism. Specifically, a *generic method* can abstract over the owner (and immutability) parameter. The underlying generics mechanism finds the correct generic method arguments, and also ensures that the created object cannot be captured in the process.

We will show how to use the factory method pattern in the context of *synchronized lists*. Consider this client code:

```
b = new LinkedList<T>();
l = Collections.synchronizedList(b);
```

The documentation of `Collections.synchronizedList` states: "*In order to guarantee serial access, it is* critical *that all access to the backing list is accomplished through the returned list.*" That means that there might be concurrency problems if a programmer accidentally uses the backing list `b` instead of `l`.

Owner-as-dominator can be used to guarantee that the backing list `b` has no outside aliases. Phrased differently, you want to own a list `l`, which is backed-up by another list `b`. The challenge is that `b` should be owned by `l` (and not by you), in order to guarantee that you do not accidentally access `b` directly and comprise thread-safety.

Fig. 5 shows how ownership can ensure that the backing list cannot be accessed from the outside. This solution avoids refactoring of existing Java code by delegating calls to the synchronized list `l`. Specifically, class `SafeSyncList` (lines 1–9) owns both the list `l` (line 2) and the backing list `b` (line 5). The key observation is that *if you want to own an object, you must create it*. This is achieved by using the factory method pattern on lines 3–7.

The `Factory` interface is defined on lines 15–17. The owner and immutability of the `Factory` is irrelevant because it only has a readonly method. However, the newly created list has a generic owner and immutability, which are unknown at the creation point (line 16). The generics mechanism fills in the correct generic arguments from the usage point (line 6) to the actual creation point (line 21). Note that the factory implementation cannot capture an alias to the newly created list on line 21, because its owner parameter `O2` is a generic method parameter and therefore cannot be used in fields.

To conclude, one can use `SafeSyncList` instead of using Sun's unsafe `synchronizedList`, and be certain no one else can access the backing list. All this was achieved using *generic methods* on lines 16 and 20.

### 2.5 Visitor pattern

The *visitor design pattern* [15] is a way of separating an algorithm from a node hierarchy upon which it operates. Instead of distributing the node processing code among all the node implementations, the algorithm is written in a single *visitor* class that has a visit method for every node in the hierarchy. This is desirable when the algorithm changes frequently or when new algorithms are frequently created. The standard implementation (that does not use reflection) defines a tiny `accept` method that is overridden in all the nodes, that calls the appropriate visit method for that node.

Nageli [22] discusses ownership in design patterns, and shows that previous work was not flexible enough to express the visitor pattern. A visitor is always mutable because it may accumulate information during the traversal of the nodes hierarchy. However, some visitors only need readonly access to the nodes, and some need to modify the nodes. In the former case, the owner of the visitor and nodes may be different, and in the latter case, it must be the same owner. The challenge is to use the same `visit` and `accept` methods, and to avoid duplicating the traversal code.

OIGJ can express the visitor pattern by relying (again) on generic methods. The key idea is to use owner-polymorphic methods: the owner of an object `o`, can pass it to an *owner-polymorphic method*, which cannot capture `o`.

Fig. 6 shows the visitor pattern in OIGJ. As mentioned before, the *owner* of the visitor and nodes may be different, and some visitors may or may not *modify* the nodes. Therefore, the visitor is parameterized on line 1 by the owner (`NodeO`) and immutability (`NodeI`) of the nodes. The `visit` method on line 2 is mutable because it changes the visitor that accumulates information during the traversal. Different visitor implementations may have different immutability for the nodes, e.g., readonly on line 15 or mutable on line 24.

Finally, note how the type arguments `This,ReadOnly` of the node on line 11 match the last two arguments of the visitor on line 13, and on line 20 the type arguments `This,Mutable`

```
1: interface Visitor<O,I,NodeO,NodeI> {
2:  <I extends Mutable>? void visit(
3:         Node<NodeO,NodeI> n);
4: }
5: class Node<O,I> {
6:  void accept(Visitor<?,Mutable,O,I> v) {
7:    v.visit(this)
8:  }
9: }
10: // Visiting a readonly node hierarchy.
11: Node<This,ReadOnly> readonlyNode = ...;
12: readonlyNode.accept( new
13:  Visitor<World,Mutable,This,ReadOnly>() {
14:   <I extends Mutable>? void visit(
15:         Node<This,ReadOnly> n) {
16:    ... // Can mutate the visitor, but not the nodes.
17:   }
18:  });
19: // Visiting a mutable node hierarchy.
20: Node<This,Mutable> mutableNode = ...;
21: mutableNode.accept( new
22:  Visitor<World,Mutable,This,Mutable>() {
23:   <I extends Mutable>? void visit(
24:         Node<This,Mutable> n) {
25:    ... // Can mutate the visitor and the nodes.
26:   }
27:  });
```

**Figure 6.** Visitor pattern in OIGJ. The `Node`'s ownership and immutability are underlined. We omit the `extends` clause for generic parameters, e.g., we assume that `NodeO extends World`. A single visitor interface can be used both for `Mutable` and `ReadOnly` nodes.

match those on line 22. This shows that the same `accept` method (without duplicating the nodes' hierarchy traversal code) can be used both for readonly and mutable hierarchies.

## 3. Formalization and Type Soundness

Proving soundness is essential in the face of complexities such as wildcards and raw/cooked objects. This section gives the typing rules and operational semantics of a simplified version of OIGJ and sketches the proofs our immutability and ownership guarantees. For lack of space, the full proofs are included in our technical report [35].

Our type system, called Featherweight OIGJ (FOIGJ), is based on Featherweight Java (FJ) [18]. FOIGJ models the essence of OIGJ: the fact that every object has an ownership and immutability, and the cooking phase when creating immutable objects. FOIGJ adds imperative constructs to FJ, such as `null` values, field assignment, locations/objects and a heap. FOIGJ also adds a constructor body (to model the cooking phase), owner and immutability parameters to classes, guards as in cJ [17], wildcard owners, and the runtime notion of raw/cooked objects.

FOIGJ poses two main challenges: (i) modeling *wildcards* in the typing rules, and (ii) the representation for *raw objects*. We use the following example (similar to Sec. 2) to demonstrate these two challenges:

```
class Foo<O,I> {
 Date<O,I> sameD;
 Date<This,I> ownedD;
 Date<This,Immut> immutD;
 <I extends Raw>? void Foo(){
   this.ownedD = new Date<This,I>();
   this.immutD = new Date<This,Immut>(); ... } }
```

Wildcards pose a difficulty due to a process called *wildcard capture* in which a wildcard is replaced with a fresh type variable. For example, the two underlined wildcards below might represent two distinct owners:

```
Foo<?,I> f = ...;  Date<?,I> d = ...;
f.sameD = d; // Illegal assignment! Different owners!
```

Java's compiler rejects the assignment due to incompatible types, because the wildcards were captured by different type variables. Formalizing the full power of wildcards (with upper and lower bounds) was only recently achieved [9]. However, FOIGJ does not need the full power of wildcards, and it is enough to augment the field assignment rule with the following check: assigning to $o$ is illegal if $\theta(o) = \texttt{?}$ (similarly for method invocation). This extra check is needed only in FOIGJ, and not in OIGJ, which is built on-top of Java, which supports wildcard capture.

The second challenge is modeling raw objects in the *non-erased* operational semantics. Recall that generics are erased in Java, i.e., they are not maintained at runtime. FOIGJ's *erased* operational semantics is identical to that of normal Java: ownership and immutability information is not kept. In contrast, the *non-erased* version stores with each object its owner and immutability, and it checks at runtime the ownership and immutability guarantees (i.e., that field assignment respects owner-as-dominator and is done only on mutable or raw objects). The non-erased version is used only in the formalism because storing the owner and immutability of every object at runtime is a huge overhead.

The non-erased semantics of Featherweight Generic Java [18] (FGJ) performs variable substitution for method calls, however FGJ's way of doing substitution does not work in FOIGJ. For example, consider the following reduction as done in FGJ:

```
new Foo<World,Immut>() →
 l.ownedD = new Date<l,Immut>();
 l.immutD = new Date<l,Immut>();...
```

The variable `I` in the constructor was substituted with `Immut`, and the variables `this` and `This` were substituted with a new location `l` that was created on the heap, i.e., the heap $H$ now contains a new object in location `l` whose fields are all `null`: $H = \{\texttt{l} \mapsto \texttt{Foo<World,Immut>}(\overline{\texttt{null}})\}$. (Locations are pointers to objects; we treat locations and objects identically

because they have a one-to-one mapping, e.g., the owner of a location is defined to be the owner of its object.) Note how owner parameters ($O(o)$) at compile-time are replaced with owners ($\theta(o)$) at run-time, e.g., `This` was replaced with location `l`.

There are two reasons why substituting `I` with `Immut` does not work in FOIGJ: (i) the reduction does not type-check because we mutate (`l.ownedD = ...`) an immutable object, and (ii) we lost information about the two new `Date` objects, namely that `ownedD` can still be mutated after its constructor finishes (because it is `this`-owned) whereas `immutD` cannot.

FOIGJ solves these two issues by introducing an auxiliary type $\texttt{Immut}_\texttt{l}$. An object $o$ of immutability $I(o) = \texttt{Immut}_\texttt{l}$ becomes cooked when the constructor of `l` finishes, therefore we call `l` its *cooker*, denoted by $\kappa(o) = \texttt{l}$. Phrased differently, an object is cooked when its cooker is *constructed* (i.e., the cooker's constructor finishes). Note that the cooker `l` can be $o$ itself, its owner, or even some other incomparable object.

The connection between the cooker and the owner will be shown in the subtyping and typing rules below. Intuitively, for a reference of type $\texttt{C<o,Immut}_\texttt{l}\texttt{>}$, if the cooker `l` is inside the owner `o`, then we consider `Immut` a subtype of $\texttt{Immut}_\texttt{l}$, i.e., that reference might point to a cooked immutable object. However, if `l` is not inside `o`, then that reference must point to an object whose cooker is `l`.

In our example, the location `l` that was created with `new Foo<World,`$\underline{\texttt{Immut}}$`>()` becomes cooked when it's constructed, i.e., $\kappa(\texttt{l}) = \underline{\texttt{l}}$, and $H = \{\texttt{l} \mapsto \texttt{Foo<World,Immut}_{\underline{\texttt{l}}}\texttt{>}(\overline{\texttt{null}})\}$. Now FGJ's way of doing the substitution works for FOIGJ, because `I` is replaced with $\texttt{Immut}_\texttt{l}$, i.e.,

```
l.ownedD = new Date<l,Immut_l>();
l.immutD = new Date<l,Immut>();
```

Note how the cooker of `ownedD` is `l`, whereas the cooker of `immutD` is `immutD` itself. Therefore, `ownedD` has a longer cooking phase than `immutD`.

FOIGJ also stores in the heap $H$ the set of currently executing constructors ($H[K] \subseteq \mathrm{dom}(H)$). We maintain the invariant that a location `l` is *raw* iff $\kappa(\texttt{l}) \in H[K]$, and require that *only mutable or raw objects can be mutated*.

Faced with such major challenges, we removed from FOIGJ anything that was not needed to prove our runtime guarantees. Specifically, FOIGJ does *not* model: generics (except for the owner and immutability parameters), owner polymorphic methods, casting, inner classes, fresh owners, or multiple immutability/owner parameters. On the one hand, the interaction between generics and *immutability* (which enables covariant subtyping) was previously proven sound in Featherweight IGJ (FIGJ) [36]. On the other hand, the interaction between generics and *ownership* (as found in the ownership nesting rule) was previously proven sound in Featherweight OGJ (FOGJ) [29]. Because covariant subtyping and ownership nesting was not changed in OIGJ, we decided not to model generics in FOIGJ. We note that FOIGJ does model `Raw`, which was not modeled previously in FIGJ.

Consider the typing rules in Fig. 4. Classes in FOIGJ have a single `Raw` constructor, therefore **Object creation rule** is always satisfied and can be ignored. Furthermore, because FOIGJ does not model generics, static, or inner classes, the following rules can also be ignored: **Ownership nesting**, **Inner classes**, **Inheritance**, **Invariant**, **Erased signature**, and **Fresh owners**. Covariant subtyping and erased signatures was described in FIGJ, and ownership nesting and (limited) owner-polymorphic methods in FOGJ. We stress that FOIGJ *does* model wildcard for the owner parameter (`?`), which is used in owner-polymorphic methods such as `sort` or `equals`. In our view, extending the formalism with fresh owners or inner classes increases the complexity of the calculus without providing new insights.

The following rules are enforced by the syntax of FOIGJ (Fig. 7): **Generic Wildcards** and **`Raw` parameter**. The remaining rules are: **Field assignment**, **Field access**, **Method invocation**, **cJ's [17] method guards**, and a simplified **subtype definition**. These rules are formalized in FOIGJ in the subtyping rules of Fig. 8 ($\Gamma \vdash \texttt{T} \leq \texttt{T}'$) and the typing rules of Fig. 9 ($\Gamma \vdash \texttt{e} : \texttt{T}$).

Sec. 3.1 describes the syntax of FOIGJ, Sec. 3.2 the subtyping rules, Sec. 3.3 the typing rules, Sec. 3.4 the reduction rules, and Sec. 3.5 proves preservation, progress, and our runtime immutability and ownership guarantees.

### 3.1 Syntax of FOIGJ

FOIGJ adds imperative extensions to FJ such as assignment to fields, object locations, `null`, and a heap [28]. A constructor initializes all the fields to `null`, and then calls a `build` method that constructs the object. Having `null` values is important because `this`-owned fields must be initialized with `null` since they cannot be assigned from the outside, i.e., they must be created within `this`. For example, a list constructor cannot receive its entries as constructor arguments, instead it must create the entries within the `build` method.

Fig. 7 presents the syntax of FOIGJ. Expressions in FOIGJ include the four expressions in FJ (method parameter, field access, method invocation, and new instance creation; *without* casting), as well as the imperative extensions (field update, `e;return l`, and values). Expression `e;return l` is created when reducing a constructor call, e.g., `new N(...)` $\rightarrow$ `l.build(...);return l`, then we proceed to reduce `l.build(...)` and finally return `l`. Note that `O` and `I` are terminals, i.e., the owner and immutability parameters are always named `O` and `I`.

An evaluation of an expression (`e`) is either infinite, or is stuck on `null`-pointer exception, or terminates with a value (`v`), which is either `null` or a location `l`.

Note how the syntax limits the usage of wildcards and `Raw`: wildcards (`?`) can be used only as the owner of method arguments, and `Raw` only as a method guard (`IG`).

We represent sequences using an over-line similarly to FJ, i.e., the length of a sequence $\overline{\texttt{x}}$ is written $\#(\overline{\texttt{x}})$, comma

| | |
|---|---|
| FT ::= C<FO, IP> | Field (and method return) Type. |
| T ::= C<MO, IP> | Type. |
| N ::= C<NO, NI> | Non-variable type (for objects). |
| NO ::= World \| l | Non-variable Owner parameter (for objects). |
| FO ::= NO \| This \| O | Field Owner parameter. |
| MO ::= FO \| ? | Method Owner parameter (including generic wildcard). |
| NI ::= Mutable \| $\texttt{Immut}_{\texttt{l}}$ | Non-variable Immutability parameter (for objects). |
| VI ::= NI \| Immut \| I | Variable Immutability for new. |
| IP ::= ReadOnly \| VI | Immutability Parameter. |
| IG ::= ReadOnly \| Immut \| Mutable \| Raw | Immutability method Guard. |
| M ::= <I extends IG>? FT m($\overline{\texttt{T}}$ $\overline{\texttt{x}}$) { return e; } | Method declaration. |
| L ::= class C<O, I> extends C′<O, I>{ $\overline{\texttt{FT}}$ $\overline{\texttt{f}}$; $\overline{\texttt{M}}$ } | cLass declaration. |
| v ::= null \| l | Values: either null or a location l. |
| e ::= v \| x \| e.f \| e.f = e \| e.m($\overline{\texttt{e}}$) \| new C<FO, VI>($\overline{\texttt{e}}$) \| e;return l | Expressions. |

**Figure 7.** FOIGJ Syntax. The terminals are null, owner parameters (O, This, World), immutability parameters (I, ReadOnly, Mutable, Raw, Immut). Given a location l, $\texttt{Immut}_{\texttt{l}}$ represents an immutable object with cooker l.

$$\frac{}{\Gamma \vdash \texttt{I} \leq \Gamma(\texttt{I})}\ (\text{S1}) \qquad \frac{}{\Gamma \vdash \texttt{T} \leq \texttt{T}}\ (\text{S2}) \qquad \frac{\Gamma \vdash \texttt{S} \leq \texttt{T} \quad \Gamma \vdash \texttt{T} \leq \texttt{U}}{\Gamma \vdash \texttt{S} \leq \texttt{U}}\ (\text{S3}) \qquad \frac{\texttt{class C<O, I> extends C}'\texttt{<O, I>}}{\Gamma \vdash \texttt{C<MO, IP>} \leq \texttt{C}'\texttt{<MO, IP>}}\ (\text{S4})$$

$$\frac{}{\Gamma \vdash \texttt{Mutable} \leq \texttt{Raw}}\ (\text{S5}) \qquad \frac{}{\Gamma \vdash \texttt{Raw} \leq \texttt{ReadOnly}}\ (\text{S6}) \qquad \frac{}{\Gamma \vdash \texttt{Immut} \leq \texttt{ReadOnly}}\ (\text{S7})$$

$$\frac{\Gamma \vdash \texttt{IP} \leq \texttt{IP}'}{\Gamma \vdash \texttt{C<MO, IP>} \leq \texttt{C<MO, IP}'\texttt{>}}\ (\text{S8}) \qquad \frac{}{\Gamma \vdash \texttt{C<MO, IP>} \leq \texttt{C<?, IP>}}\ (\text{S9}) \qquad \frac{\texttt{l} \in \Gamma[K]}{\Gamma \vdash \texttt{Immut}_{\texttt{l}} \leq \texttt{Raw}}\ (\text{S10})$$

$$\frac{\texttt{l} \notin \Gamma[K]}{\Gamma \vdash \texttt{Immut}_{\texttt{l}} \leq \texttt{Immut}}\ (\text{S11}) \qquad \frac{\texttt{l} \notin \Gamma[K]}{\Gamma \vdash \texttt{Immut} \leq \texttt{Immut}_{\texttt{l}}}\ (\text{S12}) \qquad \frac{\texttt{l} \prec_\theta \texttt{NO}}{\Gamma \vdash \texttt{C<NO, Immut>} \leq \texttt{C<NO, Immut}_{\texttt{l}}\texttt{>}}\ (\text{S13})$$

**Figure 8.** FOIGJ Subtyping Rules. Rule S13 shows the connection between cooker l and owner NO.

denotes concatenation of sequences, and $\overline{\texttt{FT}}$ $\overline{\texttt{f}}$; represents the sequence $\texttt{FT}_1$ $\texttt{f}_1$; ... $\texttt{FT}_n$ $\texttt{f}_n$;

A class in FOIGJ has a single constructor that can create both mutable and immutable objects, i.e., it is a Raw constructor. The constructor is not shown in the syntax because it can be inferred from the class declaration: it always assigns null to the fields of the newly created object, and then invokes this special method (ignoring the return value):

```
<I extends Raw>? T′ build(T̄ ē) { return e; }
```

We require that each class has such a method, and that its parameters are not this-owned nor have wildcards. The reduction rules call that method after the fields are set to null.

### 3.2 Subtyping in FOIGJ

An environment $\Gamma$ is a finite mapping from variables x and locations l to types T, e.g., $\texttt{x} : \texttt{T} \in \Gamma$. The location types define the ownership tree $\preceq_\theta$ (or without reflexivity $\prec_\theta$). The set of currently executing constructors, denoted $\Gamma[K]$, is also part of $\Gamma$. (We will set $\Gamma[K] = H[K]$.) In addition, $\Gamma$ maps the immutability parameter I to its bound according to the current method's guard (IG in Fig. 7). For example, $\texttt{I} : \texttt{Raw} \in \Gamma$ when typing the expression e in method build above.

Fig. 8 shows FOIGJ subtyping rules. Rules S1–S4 are the same as FGJ rules: the first rule say that a generic variable is a subtype of its bound, second is reflexivity, third is transitivity, and fourth is that subclassing defines subtyping. Rules S5–S7 show subtyping among non-variable immutability parameters as shown in Fig. 1b. Rule S8 defines covariant subtyping for the immutability parameter. Rule S9 formalizes subtyping with a wildcard owner.

The last four rules S10–S13 concern with *cookers* such as $\texttt{Immut}_{\texttt{l}}$. Recall that an object is cooked when its cooker l is constructed, i.e., the constructor of l is no longer executing: $\texttt{l} \notin \Gamma[K]$. Rule S10 views the type as Raw, while rules S11–S12 shows the equivalence to Immut. Note that subtyping is no longer antisymmetric, i.e., there are non-equal types $\texttt{T}_1$ and $\texttt{T}_2$ for which $\texttt{T}_1 \leq \texttt{T}_2 \leq \texttt{T}_1$. For example, $\texttt{T}_1 = \texttt{C<O, Immut}_{\texttt{l}}\texttt{>}$ and $\texttt{T}_2 = \texttt{C<O, Immut>}$, if $\texttt{l} \notin \Gamma[K]$. In fact, this is not surprising because these types both represent immutable object, and after the cooker is cooked, the identity of the cooker is irrelevant.

***Cooker vs. Owner*** Rule S13 assumes that the cooker is inside the owner ($\texttt{l} \prec_\theta \texttt{NO}$), which means the object might came from the outside. This rule addresses the difference between the cooker of (i) a location l or (ii) that of an expression such as field access l.f: (i) location l will be cooked *exactly* when the constructor of $\kappa(\texttt{l})$ is finished, however, (ii) the cooker of l.f is an *over-approximation*,

i.e., the object stored in that field might have been cooked earlier. Rule S13 allows an over-approximation only when the cooker is inside the owner.

Consider this example:

```
class Foo<O,I> { Date<O,I> same;
 <I extends Raw>? Foo(Date<O,I> d) { same=d; }  }
```

Field d is assigned from the outside, but it might still be this-owned. We will show the reduction of two expressions: one where d is assigned a cooked (outside) date, and one with a raw date. The expressions are inside the constructor of some object b whose cooker is b itself.

The reduction of the first expression:

```
new Foo<This,Immut>(new Date<This,Immut>())
```

results in the heap: $H = \{\texttt{d1} \mapsto \texttt{Date<b,Immut}_{\texttt{d1}}\texttt{>()}, \texttt{f1} \mapsto \texttt{Foo<b,Immut}_{\texttt{f1}}\texttt{>(d1)}\}$. Note that the type of d1 must be a subtype of f1.same in a well-typed heap (formally defined later). The type of d1 is a subtype of Date<b,Immut> (because $\texttt{d1} \notin \Gamma[K]$ in rule S12), which is a subtype of $\texttt{Date<b,Immut}_{\texttt{f1}}\texttt{>}$ (because $\texttt{f1} \prec_\theta \texttt{b}$ in rule S13). Phrased differently, the cooker of f1.same is f1, but it may point to an object that was cooked before, and indeed it points to an object whose cooker is d1 (so it is an over-approximation).

The reduction of the second expression:

```
new Foo<This,I>(new Date<This,I>())
```

results in the heap (because $\texttt{I} = \texttt{Immut}_\texttt{b}$): $H = \{\texttt{d2} \mapsto \texttt{Date<b,Immut}_\texttt{b}\texttt{>()}, \texttt{f2} \mapsto \texttt{Foo<b,Immut}_\texttt{b}\texttt{>(d2)}\}$. Note that in this case, both d2 and f2 have the same cooker b. The type of f2.same is $\texttt{Date<b,Immut}_\texttt{b}\texttt{>}$, and because $\texttt{b} \not\prec_\theta \texttt{b}$ (see rule S13), then we know that this is not an over-approximation, i.e., that field points to an object whose cooker must be b.

To summarize, consider a type $\texttt{Foo<o,Immut}_\texttt{c}\texttt{>}$. If the cooker c is inside the owner o ($\texttt{c} \prec_\theta \texttt{o}$), or the cooker is cooked ($\texttt{c} \notin \Gamma[K]$), then the type is an over-approximation, i.e., it can point to any Immut object (that is, to any object with cooker $\texttt{c}' \notin \Gamma[K]$). Otherwise, it points to an object whose cooker is exactly c. Formally,

LEMMA 3.1. *If* $\Gamma \vdash \texttt{C<MO,IP>} \leq \texttt{C}'\texttt{<NO,Immut}_\texttt{l}\texttt{>}$, $\texttt{l} \not\prec_\theta \texttt{NO}$, *and* $\texttt{l} \in \Gamma[K]$, *then* $\texttt{IP} = \texttt{Immut}_\texttt{l}$.

We also prove in the technical report that:

LEMMA 3.2. *If* $\Gamma \vdash \texttt{C<MO,IP>} \leq \texttt{C}'\texttt{<MO}', \texttt{IP}'\texttt{>}$, *then (i)* $\texttt{MO}' \neq \texttt{?} \Rightarrow \texttt{MO} = \texttt{MO}'$, *(ii)* $(\texttt{IP}' \neq \texttt{Immut}_\texttt{l}$ *or* $\texttt{l} \not\prec_\theta \texttt{MO}') \Rightarrow \Gamma \vdash \texttt{IP} \leq \texttt{IP}'$, *and (iii)* C *is a subclass of* C′, *(iv)* $\Gamma \vdash \texttt{D<l,IP>} \leq \texttt{D<l,IP}'\texttt{>}$ *for any class* D *and location* l *where* $\texttt{MO}' \preceq_\theta \texttt{l}$.

### 3.3 Typing rules of FOIGJ

***Auxiliary functions*** We use the following auxiliary functions: *fields*(C) returns all the field names (including inherited fields) of class C, *ftype*(f,C) returns the type of field f in class C, *mtype*(m,C) returns the type of method m in class C, and *mbody*(m,C) returns its body. Their definitions are based on their counterparts in FJ, and thus omitted from this paper. In addition, function *mguard*(m,C) returns the method's guard (IG in Fig. 7).

We overload the auxiliary functions above to work also for types ($\texttt{T} = \texttt{C<MO,IP>}$) and not just classes (C) by substituting [MO/O,IP/I]. However, we also need to carefully substitute This when the receiver is this or locations. Function *ftype*(e,f,T) returns the type of the field access e.f where T is the type of e, or error if the access is illegal. For example,

$$\begin{aligned}
\textit{ftype}(\texttt{ownedD},\texttt{Foo}) &= \texttt{Date<This,I>}\\
\textit{ftype}(\texttt{this},\texttt{ownedD},\texttt{Foo<O,Immut>}) &= \texttt{Date<This,Immut>}\\
\textit{ftype}(\texttt{this.f},\texttt{ownedD},\texttt{Foo<O,Immut>}) &= \texttt{error}\\
\textit{ftype}(\texttt{l},\texttt{ownedD},\texttt{Foo<o,Immut}_\texttt{c}\texttt{>}) &= \texttt{Date<l,Immut}_\texttt{c}\texttt{>}
\end{aligned}$$

Formally, *ftype*(e,f,C<MO,IP>)=[MO/O,IP/I,z/This]*ftype*(f,C), where $\texttt{z} = \texttt{l}$ if $\texttt{e} = \texttt{l}$, $\texttt{z} = \texttt{This}$ if $\texttt{e} = \texttt{this}$, otherwise $\texttt{z} = \texttt{error}$ (and if a type contains error then it means the call to *ftype* failed). When we know the field is not this-owned, then the expression e is not used, and we write $\textit{ftype}(\bot,\texttt{f},\texttt{C})$. However, if the field is this-owned, then e must be this or a location l.

Recall that **Field access rule** in Fig. 4 required that this-owned fields can be accessed only via this. At runtime, this is substituted with a location l. Therefore, there is a *duality* between this and a location l in the definition of *ftype*. For example, the field access this.ownedD of type Date<This,I> is legal because we accessed a this-owned field via this. At runtime, this is substituted with some location l, and the access l.ownedD of type Date<l,...> is now still legal because we accessed a this-owned field via a location l. Note that if the access is not done via a location, e.g., bar.l.ownedD, then we cannot type-check the resulting expression (because we do not know what should be the substitute for This).

There is a similar duality in **Field assignment rule** part (ii), that checks that Raw is transitive for this or this-owned objects. The dual of this is a location l, and the dual of a this-owned object (C<This,I>) is an object whose cooker is not inside its owner ($\texttt{C<o,Immut}_\texttt{l}\texttt{>}$ where $\texttt{l} \not\prec_\theta \texttt{o}$). The second duality holds because, for type C<This,I>, the cooker (I) is never inside the owner (This). At runtime, the owner will be the location of this, and the cooker is either this or some other object that was created before this, i.e., the cooker is never inside the owner (but they might be equal).

Function *isTransitive* checks whether Raw is transitive. The underlined part shows the *dual* version of **Field assignment rule** part (ii): (i) this vs. l′, and (ii) $\texttt{MO} = \texttt{This}$ vs. $\texttt{l} \not\prec_\theta \texttt{MO}$.

$$\begin{aligned}
&\textit{isTransitive}(\texttt{e},\Gamma,\texttt{C<MO,IP>}) =\\
&\quad (\texttt{IP} = \texttt{I} \text{ and } \Gamma(\texttt{I}) = \texttt{Raw} \Rightarrow (\texttt{e=this or } \texttt{MO} = \texttt{This})) \text{ or}\\
&\qquad \underline{(\texttt{IP} = \texttt{Immut}_\texttt{l} \Rightarrow (\texttt{e=l}' \text{ or } \texttt{l} \not\prec_\theta \texttt{MO}))}
\end{aligned}$$

***Typing class declarations*** FOIGJ program consists of class declarations followed by the program's expression. Next, we describe in words the rules for typing the class declarations,

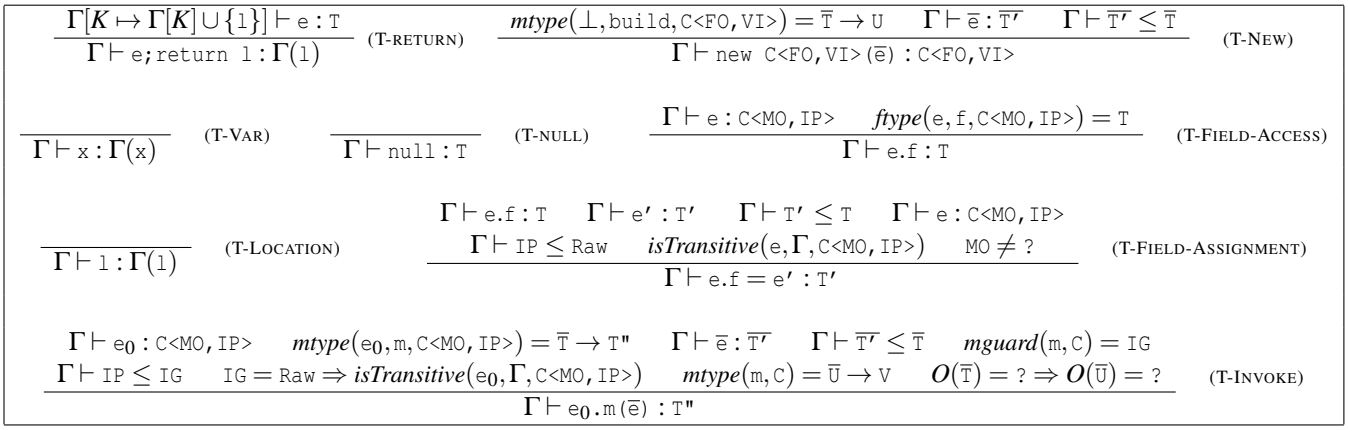


**Figure 9.** FOIGJ Expression Typing Rules.

and the rules for typing an expression are given formally in Fig. 9.

For typing the class declarations, we do all the normal checks done in FJ, e.g., that there are no cycles in the inheritance relation, that field and method names are unique in a class, that `this` is not a legal method parameter name, that an overriding method maintains the same signature, etc. FOIGJ performs additional checks related to method guards when typing method declarations, i.e., we modify rule T-METHOD in FJ as follows: (i) An overriding method can only make the guard weaker, i.e., if a method with guard `IG` overrides one with guard `IG'` then $\texttt{IG}' \leq \texttt{IG}$. (ii) In class `C`, when typing a method: `<I extends IG>? FT m(`$\overline{\texttt{T}}\ \overline{\texttt{x}}$`) { return e; }`
we use an environment $\Gamma$ in which the bound of `I` is `IG`, i.e., $\Gamma = \{\texttt{I} : \texttt{IG}, \overline{\texttt{x}} : \overline{\texttt{T}}, \texttt{this} : \texttt{C<O,I>}\}$, and we must prove that $\Gamma \vdash \texttt{e} : \texttt{S}$ and $\Gamma \vdash \texttt{S} \leq \texttt{FT}$. Finally, we require that if $\texttt{IG} = \texttt{ReadOnly}$ then $I(\texttt{T}_i) \neq \texttt{I}$. This last requirement is not really a limitation, because a programmer can replace `I` with `ReadOnly` for parameters in readonly methods, and previously legal programs would remain legal. (This requirement is needed to prove preservation for the congruence rule of method receiver, see our technical report for details.)

***Typing expressions*** Fig. 9 shows the typing rules for expressions in FOIGJ. Most of these rules are a direct translation from Fig. 4. The main challenge was handling *wildcards* correctly without resulting to wildcard-capture. Rule T-FIELD-ASSIGNMENT requires that $\texttt{MO} \neq \texttt{?}$, i.e., one cannot assign to an object with unknown owner. Typing method parameters is similar to typing field-assignment, however, method parameters can have a wildcard owner whereas fields cannot (see the difference between `T` and `FT` in Fig. 7). Therefore, rule T-INVOKE requires that $O(\overline{\texttt{T}}) = \texttt{?} \Rightarrow O(\overline{\texttt{U}}) = \texttt{?}$, i.e., if $\texttt{T}_i = \texttt{C<?,IP>}$ then $\texttt{U}_i = \texttt{C<?,IP'>}$. Phrased differently, if the owner of $\texttt{e}_0$ is unknown, then the owner of the method parameters cannot be `O`.

Rule T-NEW performs less checks compared to a method call (e.g., no need to check the guard, *isTransitive*, nor wildcards) because `build` has several restrictions: its guard is `Raw` and it does not contain wildcards nor `this`-owned parameters. Because `This` does not appear in the signature of `build`, we know it will not use $\bot$ in the call: $\mathit{mtype}(\bot, \texttt{build}, \texttt{C<FO,VI>})$.

An expression/type is called *closed* if it does not contain any free variables (such as wildcards, `this`, `I`, `O`, or `This`), but it may contain `World`, `ReadOnly`, `Mutable`, `Immut`, $\texttt{Immut}_{\texttt{l}}$ or locations. Note that the type of a closed expression is also closed.

LEMMA 3.3. *If* $\Gamma \vdash \texttt{e''} : \texttt{T''}$ *and* $\texttt{e''} \neq \texttt{null}$, *then* `T''` *is closed.*

The delicate part of the proof is showing that `T''` does not contain `This`. Note that *ftype* returns a type with `This` only if $\texttt{e} = \texttt{this}$ (which cannot happen since `e` is closed).

### 3.4 Reduction rules of FOIGJ

The initial expression to be reduced is *closed*, and we guarantee that a closed expression is always reduced to another closed expression:

LEMMA 3.4. *If* `e` *is closed and* $H, \texttt{e} \rightarrow H', \texttt{e}'$, *then* $\texttt{e}'$ *is closed.*

The heap (or store) $H = \{\texttt{l} \mapsto \texttt{C<NO,NI>}(\overline{\texttt{v}})\}$ maps each location `l` to an object, where $\theta(\texttt{l}) = \texttt{NO}$ is its owner, and $I(\texttt{l}) = \texttt{NI}$ is its immutability. If $\texttt{NI} = \texttt{Immut}_{\texttt{l}'}$ then we say that its cooker is $\kappa(\texttt{l}) = \texttt{l}'$. The heap also contains the set of currently executing constructors $H[K]$. We define a *heap-typing* $\Gamma_H : \texttt{l} \mapsto \texttt{T}$ that gives a type to each location in the obvious way; also $\Gamma_H[K] = H[K]$.

A *well-typed* heap $H$ satisfies: (i) there is a linear order $\preceq^T$ over $\mathrm{dom}(H)$ such that for every location `l`, $\theta(\texttt{l}) = \texttt{World}$ or $\theta(\texttt{l}) \prec^T \texttt{l}$, and $I(\texttt{l}) = \texttt{Mutable}$ or $\kappa(\texttt{l}) \preceq^T \texttt{l}$, and (ii) each field location is a subtype (using $\Gamma_H$) of the declared field type. The linear order $\preceq^T$ can order the objects according to their *creation time*, because $\theta(\texttt{l})$ is always created before `l`, and $\kappa(\texttt{l})$ is either `l` or created before `l`.

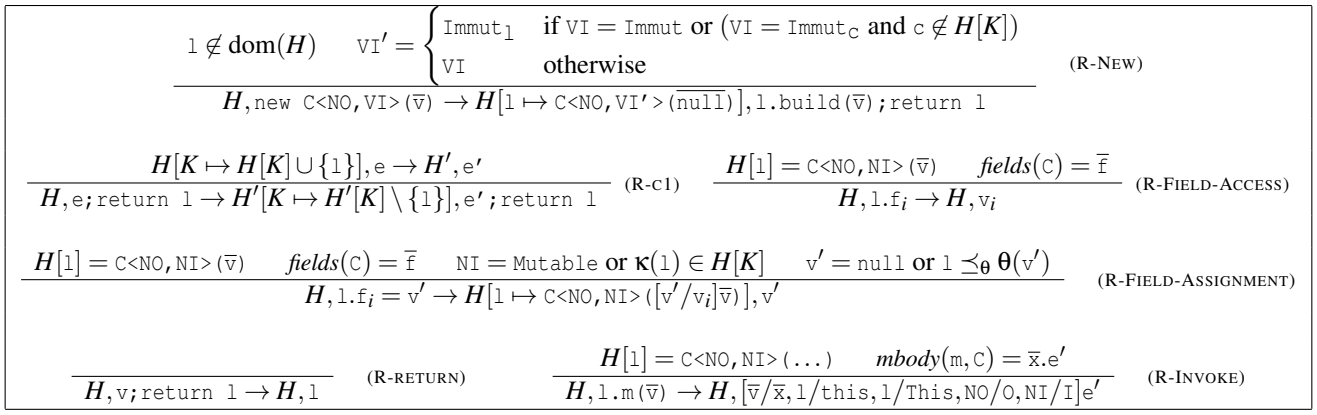


**Figure 10.** FOIGJ Reduction Rules (excluding all congruence rules except the one for `e;return l` described in R-C1).

In our technical report we prove that (a) owner-as-dominator holds in a well-typed heap (Lem. 3.5), and (b) $H$ remains well-typed even if we remove locations from $H[K]$ (Lem. 3.6), i.e., being well-typed is a *stable property*. (This is not trivial, because decreasing $H[K]$ changes the subtyping relation by turning raw objects into immutable.)

LEMMA 3.5. *If heap $H$ is well-typed, then for every location* $\texttt{l} \in \mathrm{dom}(H)$, $\texttt{l} \mapsto \texttt{C<NO,NI>}(\overline{\texttt{v}})$, *then either* $\texttt{v}_i = \texttt{null}$ *or* $\texttt{l} \preceq_\theta \theta(\texttt{v}_i)$.

LEMMA 3.6. *Given a well-typed heap $H$, then for any $S \subset H[K]$, the heap $H' = H[K \mapsto S]$ is well-typed.*

Fig. 10 presents the reduction rules in a small-step notation, excluding all congruence rules except R-C1.

Rule R-RETURN ignores the return value of `build` and returns `l`. Rule R-FIELD-ACCESS is trivial. Rule R-FIELD-ASSIGNMENT enforces our immutability guarantee (only mutable or raw objects can be mutated) and our ownership guarantee (owner-as-dominator, i.e., `l` can point to $\texttt{v}'$ iff $\texttt{l} \preceq_\theta \theta(\texttt{v}')$). Rule R-INVOKE finds the method body according to the receiver, and substitutes all the free variables in the method body.

Rule R-C1 is the congruence rule for `e;return l`. Note that this rule is the only place the set $H[K]$ is modified, i.e., when reducing `e`, we add `l` to $H[K]$, and remove it afterwards. It is easy to prove that if $H, \texttt{e} \to H', \texttt{e}'$ then (i) $H[K] = H'[K]$, and (ii) $\Gamma_H \subseteq \Gamma_{H'}$. The other congruence rules are not shown because they are trivial, e.g., in order to reduce a method call $\texttt{e}_0\texttt{.m}(\overline{\texttt{e}})$, we first reduce $\texttt{e}_0$ to a location, then reduce the first argument to a value, etc.

Rule R-NEW creates a new location `l`, sets the fields to `null`, sets the cooker of `l` (`VI'`) and finally calls `build`. In order to build the newly created object `l`, then it must be raw, i.e., its cooker `VI'` must be in $H[K]$. (Note that `l` will be in $H[K]$ according to R-C1.) Therefore, if $\texttt{VI} = \texttt{Immut}_{\texttt{C}}$ and $\texttt{c} \notin H[K]$, then we must set the cooker to `l`. This can happen if there is a method that returns `new C<O,I>(...)` and the receiver is a *cooked* immutable object.

### 3.5 Guarantees of FOIGJ

We now turn to prove various properties of FOIGJ, including preservation theorem, ownership and immutability guarantees, and an erasure property. In the remainder of this section, we assume that reduction does not get stuck on *null-pointer exceptions*, i.e., the receiver/target of field access, assignment and method calls is never `null`. Under this assumption, then `e` can always be reduced to another expression $\texttt{e}'$.

Before stating the preservation theorem, we need to establish a connection between $H[K]$ and the reduced expression `e`, which may contain `return l`. Given an expression `e`, we define $K(\texttt{e})$ to be the set of all ongoing constructors in `e`, i.e., all the locations in subexpressions $\texttt{e}'\texttt{;return l}$. Formally, $K(\texttt{e;return l}) = K(\texttt{e}) \cup \{l\}$, and for any other expression we just recurse into all subexpressions, e.g., $K(\texttt{e.f=e}') = K(\texttt{e}) \cup K(\texttt{e}')$.

A heap $H$ is *well-typed for* `e` if $H[K \mapsto H[K] \cup K(\texttt{e})]$ is well-typed. From Lem. 3.6, if $H$ is well-typed for `e`, then $H$ is well-typed.

THEOREM 3.7. ***(Progress and Preservation)*** *For every closed expression* $\texttt{e} \neq \texttt{v}$, *and a heap $H$ that is well-typed for* `e`, *if* $\Gamma_H \vdash \texttt{e} : \texttt{T}$, *then there exists* $H', \texttt{e}', \texttt{T}'$ *such that* $H, \texttt{e} \to H', \texttt{e}'$, *$H'$ is well-typed for* $\texttt{e}'$, `T`, `T'`, *and* $\texttt{e}'$ *are closed,* $\Gamma_{H'} \vdash \texttt{e}' : \texttt{T}'$, *and* $\Gamma_{H'} \vdash \texttt{T}' \leq \texttt{T}$.

Proved by showing there is always (exactly) one applicable reduction rule, which preserves subtyping. From Lem. 3.4, we know that $\texttt{e}'$ is closed, and from Lem. 3.3, we know that `T` and $\texttt{T}'$ are closed. Next we mention some highlights from the proof. In rule R-RETURN, we have that $K(\texttt{e}') = K(\texttt{e}) \setminus \{\texttt{l}\}$, but even though we shrink $H[K]$, we still have a well-typed heap from Lem. 3.6. In rule R-FIELD-ASSIGNMENT, Lem. 3.5 shows that the assumption $\texttt{l} \preceq_\theta \theta(\texttt{v}')$ holds, and the resulting heap is well-typed for $\texttt{e}'$ because $K(\texttt{e}') = \{\}$ and from T-FIELD-ASSIGNMENT. In rule R-NEW, we need to type the call $\texttt{l.build}(\overline{\texttt{v}})$, and for parameters with immutability `I`, we use the subtyping rule S13.

Our ownership and immutability guarantees follow directly from the reduction rules, because rule R-FIELD-ASSIGNMENT enforces them.

Thm. 3.8 shows that there is no need to maintain at runtime $H[K]$ nor to store the owner and immutability parameter of each object. Formally, we define an erased heap structure $E(H)$ that maps location to objects without these parameters, i.e., $\mathtt{l} \mapsto \mathtt{C}(\overline{\mathtt{v}}) \in E(H)$. We define the erasure of an expression $\mathtt{e}$, $E(\mathtt{e})$, by deleting all generic parameters, and define new reduction rules $\rightarrow_E$ in the obvious way.

THEOREM 3.8. ***(Erasure)*** *If* $H, \mathtt{e} \rightarrow H', \mathtt{e}'$ *then* $E(H), E(\mathtt{e}) \rightarrow_E E(H'), E(\mathtt{e}')$.

# 4. OIGJ Case Studies

This section describes our implementation of OIGJ: the language syntax (Sec. 4.1) and the type-checker implementation (Sec. 4.2). Sec. 4.3 presents our case study that involved annotating Sun's implementation of the `java.util` collections, and our conclusions about the design of the collection classes w.r.t. ownership and immutability.

The prototype OIGJ type-checker is implemented and distributed as part of the Checker Framework [27][1], which supports pluggable type systems using type annotations.

## 4.1 Syntax: From Generics to Annotations

Whereas this paper uses generics to express ownership and immutability (e.g., `Date<O,I>`), our OIGJ implementation uses Java 7's type annotations (e.g., `@O @I Date`). Java 7 supports receiver annotations, which play the same role as cJ's guards.

Using annotations has the advantage of compatibility with existing compilers and other tools. Another advantage is the ability to use a default value, such as `@Mutable`. Furthermore, it is possible to customize these defaults per class. Defaults are not possible in generics, because a programmer must supply arguments for all generic parameters.

Using annotations has the disadvantage that some notions are no longer explicit in the syntax, such as transitivity, wildcards, and generic methods. Use of annotations also complicates the implementation (see below). For practical use, the compatibility benefits of using annotations outweigh their disadvantages.

**OIGJ's annotations** are the Cartesian product of owner parameters and immutability parameters. Our implementation does not yet support wildcards (though in practice the `@I` and `@O` annotations subsume most need for wildcards), nor classes with multiple owner or immutability parameters, such as `Visitor<O,I,NodeO,NodeI>` in Fig. 6.

A class declaration can be annotated as `@Immut` to indicate class immutability, i.e., all instances are immutable and no mutable methods exist.

[1] `http://types.cs.washington.edu/checker-framework/`

## 4.2 OIGJ Implementation

Because a pluggable type checker augments, rather than replaces, the type system of the underlying language, the Checker Framework permits only language extensions that are stricter than ordinary Java. A pluggable type system cannot relax Java's rules, as the OIGJ subtyping rule does. For example,

```
@Immut    List<@Immut    Date>   a;
@ReadOnly List<@ReadOnly Date>   b=a; // OK
@Immut    List<@Immut    Object> c=a; // Illegal!
```

The assignment `c=a` is illegal in Java and therefore in the Checker Framework, though it is legal in OIGJ itself. Phrased differently, in our implementation, the covariance is limited to annotations.

The OIGJ type-checker incorporates, extends, and in some places overrides the IGJ checker, and adds OGJ features. It consists of about 700 source lines of code (of which 100 lines is Java boilerplate to define the annotations). Most of the code handles default and implicit types. For example, every string literal is immutable, and every local variable is initially set to be readonly (then refined by a built-in flow-sensitive analysis).

## 4.3 `java.util` Collections Case Study

As a case study, we type-checked Sun's implementations of the `java.util` collections (77 classes, 33,246 lines of code). This required us to write 85 *ownership-related* annotations and 46 *immutability-related* annotations in 102 lines of code (the lines with `new` usually contain 2 annotations).

Sun's collections are not type-safe with respect to generics because Java does not support generic arrays. However, the OIGJ implementation uses type annotations, which can be placed on arrays as well, and therefore our annotated collections type-check without any errors with respect to ownership and immutability.

Class `LinkedList` in Fig. 3 is similar in essence to Sun's implementation. We annotated the constructors with `Raw`, thus allowing creation of immutable instances. Since all instances of `Entry` are `this`-owned, using `@This @I` as the default annotation for `Entry` meant that only three *ownership-related*[2] annotations were needed in `LinkedList`:

```
@Default({This.class, I.class})
static class Entry<E> {
  E element; @O Entry<E> next; @O Entry<E> prev;
  ... }
```

Similarly, in a `HashMap`, both the array and the entries are `this`-owned: `@This @I Entry[@This @I] table;`

The case study supports these conclusions: (i) the collections classes are properly encapsulated (they own their representation), (ii) it is possible to create immutable instances (all constructors are `Raw`), and (iii) methods `Map.get` and `clone` contain design mistakes (see below). We were not previously

[2] The other annotations are immutability-related, e.g., receiver annotations.

aware of these design mistakes. We believe that if the collections were designed with ownership and immutability in mind, such mistakes could be avoided.

***Immutability of method* `get`** Let's start with a quick riddle: is there a `Map` implementation in `java.util` that might throw an exception when running the following *single-threaded* code?

```
for (Object key : map.keySet()) { map.get(key); }
```

The answer is that for a map created with

```
new LinkedHashMap(100, 1, /*accessOrder=*/ true)
```

that contains more than one element, the above code throws `ConcurrentModificationException` after printing one element.

Most programmers assume that `Map.get` is readonly, but there is no such guarantee in Java's specification. The documentation of `LinkedHashMap` states: "*A special constructor is provided to create a linked hash map whose order of iteration is the order in which its entries were last accessed, from least-recently accessed to most-recently (*access-order*). Invoking the* `put` *or* `get` *method results in an access to the corresponding entry.*"

Because calling `get` modified the list, the above code threw `ConcurrentModificationException`. Phrased differently, method `LinkedHashMap.get` is mutable! Because an overriding method can only strengthen the specification of the overridden method, `HashMap.get` and `Map.get` must be mutable as well.

***Ownership and method* `clone`** Method `clone` violates owner-as-dominator because it leaks `this`-owned references by creating a shallow copy, i.e., only immediate fields are copied. Furthermore, Sun's implementation of `LinkedList` assigns to `result.header`, which is a `this`-owned field. This violates **Field assignment rule**, which only permits assignment to `this.header`.

```
// The following code appears in LinkedList.clone().
// Calling super.clone() breaks owner-as-dominator because
// it leaked this.header to result.header.
LinkedList result = (LinkedList) super.clone();
result.header = new Entry(); // Illegal in OIGJ!
```

We sketch a solution that, instead of initializing the cloned `result` from `this`, uses the idea of *inversion of control*. The solution has two parts. (1) The programmer writes a method `constructFrom` that initializes `this` from a parameter. (This is similar to a copy-constructor in C++, and indeed this method should be given all the privileges of a constructor, such as assignment to `final` fields.) (2) The compiler automatically generates a `clone` method that first nullifies all the reference fields and then calls the user generated `constructFrom` method. This approach enforces the ownership and immutability guarantees.

# 5. Related Work

In this section we discuss related work on ownership and immutability. We first highlight the relationship between OIGJ and our previous work on ownership (OGJ) and immutability (IGJ). We also survey some of the most relevant related language designs and show how OIGJ compares to them.

## 5.1 Relationship with OGJ and IGJ

OIGJ can be thought of as the "cartesian product" of OGJ and IGJ: OIGJ uses two type parameters to express ownership and immutability. However, the delicate intricacies between ownership and immutability required changes to both OGJ and IGJ, making OIGJ more expressive than a naive combination of both.

**Ownership Generic Java (OGJ)** [29] demonstrated how ownership and generic types can be unified as a language feature. OGJ featured a single owner parameter for every class that was treated in the same way as normal generic type parameters, simplifying the language, the formalism, and the implementation.

OGJ completely prohibits wildcards as owner parameters, e.g., `Point<?>`, whereas OIGJ relaxes this rule and allows wildcards on stack variables, which enables writing the `equals` method (see **Generic Wildcards rule** in Sec. 2.3).

In OGJ, a method may have generic parameters that are owner parameters, e.g.,

```
class Foo<O extends World> {
 <O2 extends World> void bar(Object<O2> o) {...}
```

However, OGJ required that the parametric owners are *outside* the owner of the class, e.g., $\texttt{O} \preceq_\theta \texttt{O2}$. This rule is very restrictive, however it guarantees that the ownership structure is a tree. OIGJ removed this rule at the cost of complicating the ownership structure: it is a *directed acyclic graph* (DAG) instead of a tree.

Finally, OIGJ can express temporary ownership within a method by using a fresh owner parameter (see **Fresh owners** in Sec. 2.3).

**Immutability Generic Java (IGJ)** [36] showed how generic types can be used to provide support for readonly references and object immutability. OIGJ used ownership information to improve the expressiveness of IGJ. Specifically, certain restrictions in IGJ no longer apply in OIGJ for `this`-owned objects. For example, `Raw` is *not* transitive in IGJ, e.g., the assignment to `next` in Fig. 3 on lines 9 and 22 is illegal in IGJ, thus limiting creation of immutable objects. In contrast, `Raw` is transitive in OIGJ for `this`-owned fields (see **Field assignment rule**), and therefore there was no need to refactor the collections' code.

IGJ includes an `@Assignable` annotation on fields that permits field assignment even in immutable objects. The `@Assignable` annotation indicates that a given field (for example, a cache) is not part of the object's abstract state. OIGJ removed this annotation to simplify the formalism and to guarantee representation immutability as well as immutabil-

| | IOJ [16] | JOE$_3$ [26] | GUT [14] | UTT [21] | OGJ [29] | IGJ [36] | OIGJ |
|---|---|---|---|---|---|---|---|
| Owner-as-dominator | + | + | | | + | | + |
| Owner-as-modifier | | | + | + | | | |
| Readonly references | | + | + | + | | + | + |
| Immutable objects | + | + | | | | + | + |
| Uniqueness | | + | | | | | |
| Ownership transfer | | | | + | | | |
| Factory method pattern | | + | + | + | + | + | + |
| Visitor pattern | | | | | | + | + |
| Sun's `LinkedList` | | | | | | | + |
| Implementation case studies available | | | | | | + | + |

**Figure 11.** Features supported by various language designs.

ity of the abstraction: the fields of a cooked immutable object never change. Our implementation supports the `@Assignable` annotation.

IGJ only permits a single immutability parameter, which simplifies the subtyping rule. In contrast, types in OIGJ can have multiple immutability parameters, for example, `Iterator<O,ItrI,CollectionI,E>`. Because IGJ uses a single immutability parameter, the immutability of an iterator and its underlying collection must be the same. Thus, in IGJ, method `next()` must be readonly (or you couldn't iterate over a readonly list), and therefore we had to use an `@Assignable` annotation on `ListItr.current` (line 36 in Fig. 3). In contrast, in OIGJ, we guard `next()` with a mutable `ItrI` (line 51), and guard `remove()` with a mutable `CollectionI` (line 52).

### 5.2 Relationship with Other Work

OIGJ uses method guards borrowed from *cJ* [17].

In what follows, we have room to survey only closely related papers from the rich literature on immutability and ownership. Fig. 11 compares OIGJ to some of the previous work described below.

**Mutability and encapsulation** were first combined by Flexible Alias Protection (FLAP) [24]. FLAP inspired a number of proposals including ownership types [13] and confined types [32]. Capabilities for Sharing [5] describes the fundamentals underlying various encapsulation and mutability approaches by separating "mechanism" (the semantics of sharing and exclusion) from "policy" (the guarantees provided by the resulting system). Capabilities gives a lower-level semantics that can be enforced at compile- or run-time. A reference can possess any combination of these 7 access rights: read, write, identity (permitting address comparisons), exclusive read, exclusive write, exclusive identity, and ownership (giving the capability to assert rights). Immutability, for example, is represented by the lack of the write right and possession of the exclusive write right. Finally, *Fractional Permissions* [6] can give semantics to various annotations such as unique, readonly, method effects, and an ownership variant called *owner-as-effector* in which one cannot read or write owned state without declaring the appropriate effect for the owner.

**Ownership types** [2, 3, 11] impose a structure on the references between objects in a program's memory. OIGJ and other work [26, 29] enforce the owner-as-*dominator* disciplines. Generic Universe Types (**GUT**) [14, 20] enforce owner-as-*modifier* by using three type annotations: `rep`, `peer`, and `readonly`. `rep` denotes representation objects (similar to `This`), while `peer` denotes objects owned by the same owner (similar to `O`). **UTT** [21] is an extension of Universe Types that supports ownership transfer by utilising a modular static analysis, which is useful for merging data-structures or complex object initialization.

MOJO [10] can express multiple ownership, i.e., objects can have more than one owner, resulting in an ownership DAG structure. OIGJ supports a single owner. Jo$\exists$ [8] supports variant subtyping over the owner parameter by using existential types. OIGJ supports wildcards used as owners for stack variables, but those are less flexible than Jo$\exists$. For example, Jo$\exists$ can distinguish a list of students that may have different owners, from a list of student that share the same unknown owner.

**Immutability and ownership.** Similarly to OIGJ, Immutable Objects for a Java-like Language (**IOJ**) [16] associates with each type its mutability and owner. In contrast to OIGJ, IOJ does not have generics, nor readonly *references* (only readonly and immutable *objects*). Moreover, in IOJ, the constructor cannot leak a reference to `this`.

X10 [25] supports constrained types that can refer to final local variables and can be adapted for ownership purposes, e.g., `Entry{owner==list} e = list.header;`
X10 also supports cyclic immutable structures by using `proto` annotations, which are similar to our immutability `I` and the notion of cookers.

**JOE$_3$** [26] combines ownership (as dominators, not modifiers), uniqueness, and immutability. It also supports owner-polymorphic methods, but not existential owners.

**Readonly references** are found in C++ (using the `const` keyword), JAC [19], modes [30], Javari [31], etc. Previous work on readonly references lack ownership information.

Boyland [4] observes that readonly does not address observational exposure, i.e., modifications on one side of an abstraction boundary that are observable on the other side. Immutable objects address such exposure because their state cannot change.

**List iterators** pose a challenge to ownership because they require a direct pointer to the list's privately owned entries, thus breaking the owner-as-dominator property. Both OIGJ and SafeJava [3] allow an inner instance to access the outer instance's privately owned objects. Clarke [11] suggested to use iterators only with stack variables, i.e., you cannot store an iterator in a field. It is also possible to redesign the code and implement iterators without violating ownership, e.g., by using internal iterators or magic-cookies [23].

## 6. Conclusion

OIGJ is a Java language extension that supports both ownership and immutability, while enhancing the expressiveness of each individual concept. By using Java's generic types, OIGJ simplified previous type mechanisms, such as existential-owners, scoped regions, and owner-polymorphic methods. OIGJ is easy to understand and implement, using only 15 (flow-insensitive) typing rules. We have formalized a core calculus called FOIGJ and proved it sound. Our implementation is backward-compatible with Java, and it scales to realistic programs. OIGJ can type-check Sun's `java.util` collections (without the `clone` method), using a small number of annotations. Finally, various design patterns, such as the factory and visitor patterns, can be expressed in OIGJ, making it ready for practical use.

Future work includes inferring ownership and immutability annotations, conducting a bigger case study including client and library code, and extending OIGJ with concepts such as *owner-as-modifier*, *uniqueness* or *external-uniqueness* [12].

## References


[1] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT Dept. of EECS, Feb. 2004.

[2] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, pages 211–230, Oct. 2002.

[3] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL*, pages 213–223, Jan. 2003.

[4] J. Boyland. Why we should not add `readonly` to Java (yet). In *FTfJP*, July 2005.

[5] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP*, pages 2–27, June 2001.

[6] J. Boyland, W. Retert, and Y. Zhao. Comprehending annotations on object-oriented programs using fractional permissions. In *IWACO*, pages 1–11, July 2009.

[7] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*, pages 183–200, Oct. 1998.

[8] N. Cameron and S. Drossopoulou. Existential quantification for variant ownership. In *ESOP*, pages 128–142, Mar. 2009.

[9] N. Cameron, S. Drossopoulou, and E. Ernst. A model for Java with wildcards. In *ECOOP*, pages 2–26, July 2008.

[10] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple ownership. In *OOPSLA*, pages 441–460, Oct. 2007.

[11] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, Oct. 2002.

[12] D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, pages 176–200, July 2003.

[13] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, Oct. 1998.

[14] W. Dietl, S. Drossopoulou, , and P. Müller. Generic Universe Types. In *ECOOP*, pages 28–53, Aug. 2007.

[15] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[16] C. Haack, E. Poll, J. Schäfer, and A. Schubert. Immutable objects for a Java-like language. In *ESOP*, pages 347–362, Mar. 2007.

[17] S. S. Huang, D. Zook, and Y. Smaragdakis. cJ: Enhancing Java with safe type conditions. In *AOSD*, pages 185–198, Mar. 2007.

[18] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3): 396–450, May 2001. ISSN 0164-0925.

[19] G. Kniesel and D. Theisen. JAC — access right based encapsulation for Java. *Software: Practice and Experience*, 31 (6):555–576, 2001.

[20] P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In *Programming Languages and Fundamentals of Programming*, pages 131–140, 1999.

[21] P. Müller and A. Rudich. Ownership transfer in universe types. In *OOPSLA*, pages 461–478, Oct. 2007.

[22] S. Nägeli. Ownership in design patterns. Master's thesis, ETH Zürich, Zürich, Switzerland, Mar. 2006.

[23] J. Noble. Iterators and encapsulation. In *TOOLS Pacific*, pages 431–442, 2000.

[24] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP*, pages 158–185, July 1998.

[25] N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff. Constrained types for object-oriented languages. In *OOPSLA*, pages 457–474, Oct. 2008.

[26] J. Östlund, T. Wrigstad, and D. Clarke. Ownership, uniqueness and immutability. In *Tools Europe*, pages 178–197, 2008.

[27] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, July 2008.

[28] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[29] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic Ownership for Generic Java. In *OOPSLA*, pages 311–324, Oct. 2006.

[30] M. Skoglund and T. Wrigstad. A mode system for read-only references in Java. In *FTfJP*, June 2001.

[31] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, Oct. 2005.

[32] J. Vitek and B. Bokowski. Confined types. In *OOPSLA*, pages 82–96, Nov. 1999.

[33] T. Wrigstad. *Ownership-Based Alias Management*. PhD thesis, Royal Institute of Technology, Sweden, May 2006.

[34] T. Wrigstad and D. Clarke. Existential owners for ownership types. *J. Object Tech.*, 6(4):141–159, May–June 2007.

[35] Y. Zibin. Featherweight Ownership and Immutability Generic Java (FOIGJ). Technical Report 10-05, School of Engineering and Computer Science, VUW, Wellington, New Zealand, Mar. 2010. `http://ecs.victoria.ac.nz/Main/TechnicalReportSeries`.

[36] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. D. Ernst. Object and reference immutability using Java generics. In *ESEC/FSE*, pages 75–84, Sep. 2007.