Featherweight Generic Ownership Technical Report

Alex Potanin, James Noble, Dave Clarke, and Robert Biddle

January 11, 2007

Contents

1 Introduction

This is a technical report about Generic Ownership. It much extends Generic Confinement [11]. It is mostly a simple extract of the chapter 5 of my thesis (currently being written up, so this might have changed since the last update).

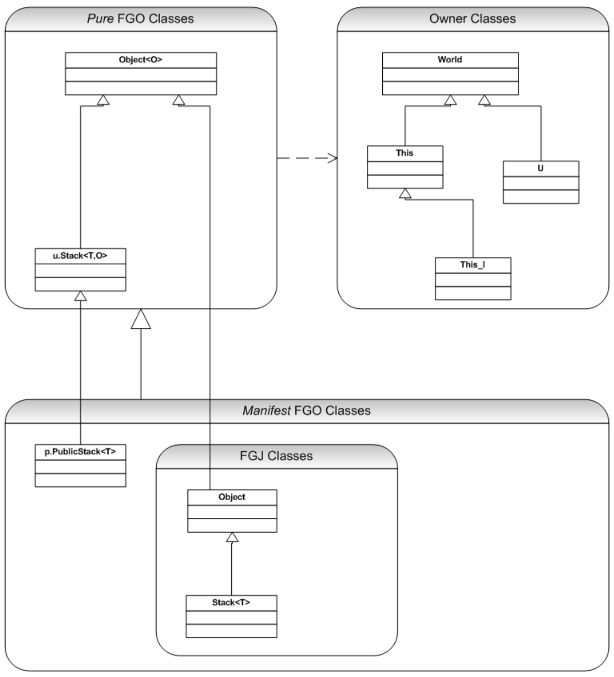

Figure 1: **FGO Classes and Owner Classes.** *Pure* FGO classes have an explicit owner type parameter. *Manifest* FGO classes have an owner fixed when subclassing a pure FGO class. *Owner* classes lie outside the FGO class hierarchy because they cannot be instantiated in FGO programs; pure FGO classes use them to bind their owner type parameters (as shown by the dashed arrow on the diagram).

2 The Rest Of The Thesis

The rest of the thesis can be obtained on request from `alex@mcs.vuw.ac.nz`.

3 Featherweight Generic Ownership

In this chapter I present Featherweight Generic Ownership (FGO) which provides object ownership support in an imperative programming language derived from Featherweight Generic Java (FGJ) [9]. FGO shows that a modern imperative language like Java (or C#) can provide both ownership and generics in a unified and type sound manner. By formulating the FGO type system and proving it sound, I demonstrate that introducing Generic Ownership into Java will not break the language.

FGO builds on the ideas presented in Chapter 2 on Featherweight Generic Confinement (FGC). FGC demonstrates that Generic Confinement can be completely subsumed by a standard type generic language like FGJ. On the other hand, FGJ (and hence FGC) cannot reason about the relationships between different instances of classes required for object ownership because they define purely functional languages. FGO specifies an imperative language with heap and references, rather than a functional language as in the case of FGC and FGJ. The presence of the object references allows FGO to provide full object ownership support, rather than confinement only support as in the case of FGC.

The FGO type system extends FGJ [9] with imperative features such as assignment, locations (modelling object references), and field updates. The Generic Ownership features of FGC, such as owner parameters and their preservation over the subtyping, are then introduced to it in a similar manner described in Chapter 2. Additionally, support for object ownership in the form of the *this* function is introduced.

The full new FGO type system has only a few more rules than FGJ. In this chapter, I describe the FGO type system in detail, provide complete type soundness proofs, and prove ownership invariant theorems. FGO is the first type system to fully support confinement, object ownership, and type genericity (type polymorphism). FGO is backed up by a prototype implementation described in Chapter 2. Furthermore, the OGJ code examples in Chapters 2 and 2 are also valid in FGO.

4 Syntax

Figure 2 shows FGO's syntax. The syntax is derived from FGJ by adding expressions for locations, `let`, field update, and `null`. The figure contains definitions for syntactical terms corresponding to types (`T`), type variables (`X`), nonvariable types (`N`), class declarations (`L`), method declarations (`M`), and expressions (`e`).

The environment Δ contains mappings from variables to their types, mappings from type variables to nonvariable types, and mappings from locations to their types. I use `<:` symbol to denote subtyping relationship. There is no explicit constructor declaration: fields are initialised to `null`. This is important because ownership implies that objects owned by `This` cannot be created before the object to which they will belong. The latter would be possible if ownership transfer was supported, but this is left for future work.

FGO uses *CT* (class table) to denote a mapping from class names `C` to class declarations `L`. An FGO program is an expression with an appropriately initialised class table *CT*. FGO does not have a sequence expression that is normally denoted by a semicolon (;). A sequence $\mathtt{e}'; \mathtt{e}$ is a syntactic sugar that can be modelled with a `let` expression of the form:

$$\mathtt{let}\ \mathtt{x} = \mathtt{e}'\ \mathtt{in}\ \mathtt{e}$$

$\mathtt{T} ::= \mathtt{X} \mid \mathtt{N}$	Type.
$\mathtt{N} ::= \mathtt{C}<\overline{\mathtt{T}}>$	Nonvariable type.
$\mathtt{L} ::= \mathtt{class}\ \mathtt{C}<\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}> \lhd \mathtt{N}\{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}$	Class declaration.
$\mathtt{M} ::= <\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}>\mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\mathtt{return}\ \mathtt{e};\}$	Method declaration.
$\mathtt{e} ::= \mathtt{e_s} \mid l \mid l\ >\ \mathtt{e} \mid \mathtt{error}$	Expressions.
$\mathtt{e_s} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}<\overline{\mathtt{T}}>(\overline{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{N}() \mid (\mathtt{N})\ \mathtt{e}$	Source expressions.
$\mid \mathtt{e.f} = \mathtt{e} \mid \mathtt{let}\ \mathtt{x} = \mathtt{e}\ \mathtt{in}\ \mathtt{e} \mid \mathtt{null}$	
$v\ ::=\ l \mid \mathtt{null}$	Values.
$l \in \mathit{locations}$	Locations.
$\Delta = \{\mathtt{x}\ :\ \mathtt{T}\} \cup \{\mathtt{X}<:\mathtt{N}\} \cup \{l\ :\ \mathtt{T}\}$	Environment that maps (1) variables to their types, (2) type variables to nonvariable types, (3) locations to their types.
$P ::= \mathtt{C} \mid l$	Permission.
$S ::= \{l \mapsto \mathtt{N}(\overline{v})\}$	Store.

Figure 2: FGO Syntax

Alternatively, nested methods calling each other in sequence can emulate sequences. To simplify the presentation, I avoid such variant of an expression. I use semicolons to separate field and method declarations, as well as to denote the end of the method's expression.

In more detail, T is a syntactical term for an FGO type that can be either a type variable (X) or a nonvariable type (N). A nonvariable type term (N) consists of a class name (C) and a list of type parameters ($\overline{\mathtt{T}}$). A class declaration (L) specifies a class name (C), bounds for every type variable used in the type parameter list ($\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}$), a super nonvariable type (N), field names with their types ($\overline{\mathtt{T}}\ \overline{\mathtt{f}}$), and a list of method declarations ($\overline{\mathtt{M}}$).

Each method declaration has a list of method type parameters supplying a bound for each type variable, followed by a return type, a method name, a list of method arguments and their types, and finally a return statement with the expression used when evaluating the method. Expression term e can be any of the expressions that can appear in the source of the program ($\mathtt{e_s}$), as well as a location (l), expression $l\ >\ \mathtt{e}$ that captures the location in the object store of the instance of the class that contains the method declaration of the method whose expression is being executed, and error arising from a bad cast or null dereference — the last three cannot appear in the source of the FGO program but appear during the evaluation of the reduction rules.

The expression $l\ >\ \mathtt{e}$ is created every time a method with receiver object l and method expression e is invoked. This expression preserves the information about the receiver location (l) so that it can be used in place of the permission P during the type checking of the subexpressions of e. Permissions are used by the visibility rules and the *this* function described later in this chapter.

The source expressions ($\mathtt{e_s}$) include five FGJ expressions: variable, field access, method invocation, object creation, and cast; as well as field update, local variable creation, and null.

$\Delta \vdash \mathtt{T}\ \mathtt{OK}$	Type $\mathtt{T}$ is OK.
$\Delta \vdash \mathtt{T} <: \mathtt{U}$	Type $\mathtt{T}$ is a subtype of type $\mathtt{U}$.
$\Delta; P \vdash \mathtt{e} : \mathtt{T}$	Expression $\mathtt{e}$ is well typed w.r.t. permission *P*.
$\Delta; P \vdash \mathit{visible}(\mathtt{e})$	Expression $\mathtt{e}$ is visible w.r.t. permission *P*.
$\Delta \vdash S$	Store (heap) is well formed.
$\Delta \vdash \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\ \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\mathtt{return}\ \mathtt{e}_0;\}\ \mathtt{FGO}\ \mathtt{IN}\ \mathtt{C},\ \mathtt{C}^{\mathtt{O}}$	Method $\mathtt{m}$ definition is OK.
$\mathtt{class}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}\ \mathtt{FGO}$	Class $\mathtt{C}$ definition is OK.

Figure 3: FGO Judgements

Finally, being an imperative language, FGO includes values (v) that can be either a location (l) or a `null`. Note that there are no primitive types supported by FGO, since to reason about aliasing only references are required as values.

Permission P (a class or a location) is used to verify if a particular type with a certain owner can be present in the current expression as described later in this chapter in the visibility rules in Section 8. Store (S) is used to represent the information about the heap — recording the objects stored in each location and values of the object's fields.

FGO owner classes are just types, but I assume that *the* owner class of a particular class is syntactically distinguishable from the other owners present among the types:

$$\mathtt{O} ::= \mathtt{X}^{\mathtt{O}} \mid \mathtt{N}^{\mathtt{O}}$$

where $\mathtt{O}$ ranges over all owners, $\mathtt{X}^{\mathtt{O}}$ ranges over owner variables, and $\mathtt{N}^{\mathtt{O}}$ ranges over nonvariable owners such as `World` and `This`, as well as the owner classes corresponding to the packages. FGO uses capitals ($\mathtt{U}$) for the owner class corresponding to a lower case package name ($\mathtt{u}$). `This` with subscript l ($\mathtt{This}_l$) is used for the owner class corresponding to the owner of an object at location l. Pure FGO types and classes are written to include an owner as their last type parameter or argument. Pure types and classes can be distinguished using the following syntax:

$$\begin{array}{lll} \mathtt{N}_{\mathit{pure}} & ::= & \mathtt{C}\mathtt{<}\overline{\mathtt{T}},\ \mathtt{O}\mathtt{>} \\ \mathtt{L}_{\mathit{pure}} & ::= & \mathtt{class}\ \mathtt{C}\mathtt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}},\ \mathtt{X}^{\mathtt{O}} \triangleleft \mathtt{N}^{\mathtt{O}}\mathtt{>} \triangleleft \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}. \end{array}$$

This syntactical distinction in no way means that the owner classes are treated differently from any other types — it is only a convenience mechanism required to distinguish *the* owner of a particular class from the other owners present in its declaration and use.

5 Type Judgements and Functions

The FGO type system uses the type judgements shown in Figure 3. These are the well formed type judgement, the well formed subtype judgement, and the well typed expression judgement that come from the FGJ type system. The *visible* judgement for expressions is the same as in

π_C	the package owner class corresponding to class $\mathtt{C}$
$this_P(\mathtt{e})$	validates the use of "*this.*" calls in expression $\mathtt{e}$
$owner_\Delta(\mathtt{T})$	the owner of type $\mathtt{T}$
$visible_\Delta(\mathtt{O},\mathtt{C})$	owner $\mathtt{O}$ is visible in class $\mathtt{C}$
$visible_\Delta(\mathtt{T},\mathtt{C})$	type $\mathtt{T}$ is visible in class $\mathtt{C}$

Figure 4: FGO Functions

FGC in Chapter 2 and is very similar to the *visible* judgement used by Zhao et al. in CFGJ [13]. The store well-formedness, method and class definition judgements are standard for FGJ-style type systems. Following the FGJ type system, the syntactical term $\mathtt{Y}$ corresponds to type variables and $\mathtt{P}$ corresponds to nonvariable types. The $\mathtt{C}^\mathtt{O}$ in method definition refers to the owner of class $\mathtt{C}$ where the method is declared.

FGO makes use of a number of functions to simplify the presentation as shown in Figure 4. π_C is assumed to be an implicit lookup function; *this*, *owner*, and *visible* are described in detail in the rest of this chapter. While the *this* function is specific to FGO, *owner* and *visible* functions were already utilised in FGC as discussed in Chapter 2.

5.1 Lookup and Auxiliary Functions

Figure 5 contains the owner lookup function that is FGO specific as well as the dictionary functions from FGJ (for fields and methods). The *owner* function gives the owner of a type. The owner of a manifest class is found by traversing the class hierarchy. Owner lookup also allows *"naked" owners* — the owner classes on their own described in Chapter 2, Section 2 — to be classified as owners of themselves.

Field lookup uses class declarations and assumes that the root of the class hierarchy has no field declarations (see F-OBJECT) — this convention is following the FGJ type system. The return of the *fields* function is a list of fields, following FGJ it is assumed that fields are not overridden using the same name so that the formulation of the FGO type system is simpler. If the field name is undefined, then field lookup function will return an empty list.

Method type and body lookups are similar to the field lookup except that they assume that the method name they are after is present at some level of the class hierarchy. If the method name is undefined, then method lookup functions will be undefined.

Figure 6 shows the FGJ function $bound_\Delta$ used to lookup the bound of a type — using the appropriate environment for the type variable and becoming an identity function for a nonvariable type. This figure also contains the subclassing rules from FGJ that enforce that subclass is a reflexive and transitive relation that is defined by the class declarations.

The last part of the figure contains the *override* rule (identical to FGJ) that validates method overriding among the subclasses. This is used in the method rule in Figure 13 later in this chapter. The reason we present the overriding rule from FGJ is for completeness of the FGO type system's

Owner Lookup (FGO-OWNER):

$$\begin{array}{lcl}
owner_{\Delta}(\mathtt{O}) & = & \mathtt{O} \\
owner_{\Delta}(\mathtt{X}) & = & owner_{\Delta}(\Delta(\mathtt{X})) \\
owner_{\Delta}(\mathtt{C<\overline{T},\ O>}) & = & \mathtt{O} \\
owner_{\Delta}(\mathtt{C<\overline{T}>}) & = & owner_{\Delta}([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}), \text{where} \\
 & & CT(\mathtt{C}) = \mathtt{class\ C<\overline{X}\lhd\overline{N}>\lhd N\{\overline{T'}\ \overline{f};\ \overline{M}\}}
\end{array}$$

Field Lookup:

$$fields(\mathtt{Object<O>}) = \bullet \qquad \text{(F-OBJECT)}$$

$$\frac{\mathtt{class\ C<\overline{X}\lhd\overline{N}>\ \lhd\ N\ \{\overline{S}\ \overline{f};\ \overline{M}\}} \qquad fields([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}{fields(\mathtt{C<\overline{T}>}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}}} \qquad \text{(F-CLASS)}$$

Method Type Lookup:

$$\frac{\begin{array}{c}\mathtt{class\ C<\overline{X}\lhd\overline{N}>\ \lhd\ \overline{N}\ \{\overline{S}\ \overline{f};\ \overline{M}\}} \\ \mathtt{<\overline{Y}\lhd\overline{P}>\ U\ m(\overline{U}\ \overline{x})\{\ return\ e;\ \}} \in \overline{\mathtt{M}}\end{array}}{mtype(\mathtt{m},\ \mathtt{C<\overline{T}>}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{<\overline{Y}\lhd\overline{P}>\overline{U}} \rightarrow \mathtt{U})} \qquad \text{(MT-CLASS)}$$

$$\frac{\mathtt{class\ C<\overline{X}\lhd N>\ \lhd\ \overline{N}\ \{\overline{S}\ \overline{f};\ \overline{M}\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{mtype(\mathtt{m},\ \mathtt{C<\overline{T}>}) = mtype(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})} \qquad \text{(MT-SUPER)}$$

Method Body Lookup:

$$\frac{\begin{array}{c}\mathtt{class\ C<\overline{X}\lhd\overline{N}>\ \lhd\ \overline{N}\ \{\overline{S}\ \overline{f};\ \overline{M}\}} \\ \mathtt{<\overline{Y}\lhd\overline{P}>\ U\ m(\overline{U}\ \overline{x})\{\ return\ e_0;\ \}} \in \overline{\mathtt{M}}\end{array}}{mbody(\mathtt{m<\overline{V}>},\ \mathtt{C<\overline{T}>}) = \overline{\mathtt{x}}.[\overline{\mathtt{T}}/\overline{\mathtt{X}},\ \overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{e_0}} \qquad \text{(MB-CLASS)}$$

$$\frac{\mathtt{class\ C<\overline{X}\lhd N>\ \lhd\ \overline{N}\ \{\overline{S}\ \overline{f};\ \overline{M}\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{mbody(\mathtt{m<\overline{V}>},\ \mathtt{C<\overline{T}>}) = mbody(\mathtt{m<\overline{V}>},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})} \qquad \text{(MB-SUPER)}$$

Figure 5: FGO Lookup Functions

presentation. In Chapter 2, the override rule and the lookup functions for fields and methods are also implicitly present in the FGJ+c type system, since the FGJ type system presented in Appendix 2 has them. Following FGJ, method overriding allows covariance of the return types, but not contravariance of the parameter types.

5.2 The This Function

The *this* function — shown in Figure 7 — is used extensively during the typing of the FGO expressions. This function helps enforce ownership, as it ensures that types involving `This` can only be used within the current object, that is, as part of message sends or field accesses upon `this`. Basically, every occurrence of `This` in the type of a method call or field access is substituted with the result of calling the *this* function; if the type involves `This`, the expression will typecheck only if the target of the call or field access is `this`.

Bound of Type:

$$\begin{array}{rcl} bound_{\Delta}(\mathtt{X}) & = & \Delta(\mathtt{X}) \\ bound_{\Delta}(\mathtt{N}) & = & \mathtt{N} \end{array}$$

Subclassing:

$$\mathtt{C} \trianglelefteq \mathtt{C} \qquad \frac{\mathtt{C} \trianglelefteq \mathtt{D} \quad \mathtt{D} \trianglelefteq \mathtt{E}}{\mathtt{C} \trianglelefteq \mathtt{E}} \qquad \frac{\mathtt{class}\ \mathtt{C}<\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}> \ \triangleleft\ \mathtt{D}<\overline{\mathtt{T}}>\{\ldots\}}{\mathtt{C} \trianglelefteq \mathtt{D}}$$

Valid Method Overriding:

$$\frac{\begin{array}{c} mtype(\mathtt{m},\ \mathtt{N}) = <\overline{\mathtt{Z}} \triangleleft \overline{\mathtt{Q}}>\overline{\mathtt{U}} \rightarrow \mathtt{U}_0 \\ \Rightarrow\ \overline{\mathtt{P}}, \overline{\mathtt{T}} = [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}](\overline{\mathtt{Q}}, \overline{\mathtt{U}}) \text{ and } \overline{\mathtt{Y}} <: \overline{\mathtt{P}} \vdash \mathtt{T}_0 <: [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}]\mathtt{U}_0 \end{array}}{override(\mathtt{m},\ \mathtt{N},\ <\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}>\overline{\mathtt{T}} \rightarrow \mathtt{T}_0)}$$

Figure 6: FGO Auxiliary Functions

$$this_{\mathtt{C}}(\mathtt{this}) \ = \ \mathtt{This} \qquad this_{l}(l) \ = \ \mathtt{This}_{l} \qquad this_{P}(\ldots) \ = \ \perp$$

Figure 7: FGO This Function

In detail, there are two distinct places where *this* function is used. They are distinguished by the permission P that is present on the left hand side of the expression type rules. The permission can be thought of as the *current context*: it denotes one of either the current class and package during the type checking of program's source, or the current instance that is also known as `this` during the run-time. The first place is during the validation of FGO class declarations (in the class and method typing rules in Figure 13, which rely on expression typing in Figure 10). Here, the permission P is set to the class `C` currently being validated. When typing a field access or method call inside `C`, the *this* function is called upon the expression $\mathtt{e}_0$ that is the target of the field access ($\mathtt{e}_0$.`f`) or method call ($\mathtt{e}_0$.`m`()). Then, all occurrences of `This` in the types of the method or field are substituted by the result of the *this* function. If the target is `this` (e.g., `this.f`), then *this* function returns `This`, the substitution will replace `This` with itself, and so the expression typechecks, even if it involves `This` types. If the target is other than `this`, the *this* function returns an undefined ($\perp$) result, so `This` is substituted by $\perp$ (leaving other types unaffected), and any expressions with `This` types will fail to typecheck. In this way, FGO ensures that `This` types can only be used upon `this`.

The second place the *this* function appears is during the reduction of FGO expressions (e.g., R-METHOD in Figure 16). In this case, expression types include locations (l). Every occurrence of a `This` owner is replaced by a location specific $\mathtt{This}_l$. The expression typing rules further ensure that every occurrence of `This` is made location specific. To achieve this, the T-CONTEXT rule (discussed later in this chapter) in Figure 10 sets the permission P to the current location l. As the expression typing rules recurse into the structure of the expression `e`, every occurrence of `This` is replaced appropriately by $this_l$ to be $\mathtt{This}_l$.

In either case, the *this* function causes any expression containing an invalid use of the `This` owner class to have an undefined type. When any FGO program — a list of class declarations followed by an expression — is type checked, a proof tree is constructed that during the validation of the expression type expands the *this* function. If the expression contains an invalid use of `This`, then the resulting expression type is undefined and thus the whole FGO program fails to type check.

FGO type soundness guarantees that any well formed FGO class will not allow access to a field or method with owner $\texttt{This}_l$ during the reduction unless the current execution context is location l.

Since the presence of the *this* function implies that operations on the objects owned by `This` are restricted to the current instance, it becomes impossible to pass the owned object to another instance. While this is a good thing since it makes it easy to prove that no other object gets the reference to the object owned by `This`, this approach can be considered too restrictive. For example, the type system cannot detect a situation where the current object is stored in a different variable:

```
class Foo {
  private Secret<This> s;
  void bar() {
    Foo me = this;
    this.s; // OK
    me.s; // Not OK
  }
}
```

This restriction is to be expected since the FGO type system does not perform any data flow analysis required to detect such uses of the `this` variable.

6 Well Formed Types and Subtyping

FGO's type well-formedness rules are shown in Figure 8. These are the same as those of FGJ, except that the root of the class hierarchy — `Object` — is parameterised (just like the root of FGJ+c class hierarchy). The grey clause in the type formation rule ensures that FGO supports deep ownership: WF-TYPE enforces the nesting of owner parameters essential to ensure *owners as dominators* (see Chapter 2, Section 2) object encapsulation. The owner nesting follows the rule that the main owner parameter is *outside or equals to* the rest of the owner parameters as discussed in Chapter 2, Section 2. The nesting of owner parameters is used to prove the ownership invariant theorem at the end of this chapter.

The nesting of owners is checked both at the class declaration time (using the class declaration rule described later in this chapter) and at the type instantiation time (using WF-TYPE rule

$$\frac{\mathtt{X} \in dom(\Delta)}{\Delta \vdash \mathtt{X}\ \mathtt{OK}}\ \text{(WF-VAR)} \qquad \frac{\Delta \vdash \mathtt{O} <: \mathtt{World}}{\Delta \vdash \mathtt{Object}<\mathtt{O}>\ \mathtt{OK}}\ \text{(WF-OBJECT)}$$

(WF-TYPE):

$$\frac{\begin{array}{ccc} \mathtt{class}\ \mathtt{C}<\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}> \ \triangleleft\ \mathtt{N}\,\{\ldots\} & \Delta \vdash \mathtt{N} <: \mathtt{Object}<\mathtt{O}> & \Delta \vdash \mathtt{O} <: \mathtt{World} \\ \Delta \vdash \overline{\mathtt{T}}\ \mathtt{OK} \quad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}} & \multicolumn{2}{c}{\forall \mathtt{T} \in \overline{\mathtt{T}} : owner_{\Delta}(\mathtt{C}<\overline{\mathtt{T}}>) <: owner_{\Delta}(\mathtt{T})} \end{array}}{\Delta \vdash \mathtt{C}<\overline{\mathtt{T}}>\ \mathtt{OK}}$$

Figure 8: FGO Type Well-Formedness Rules

presented here). The owner of the class has to be *outside* all of the other owners coming from its various type parameters to prohibit occurrences similar to:

```
class BreaksDeepOwnership<SecretOwner extends World,
  Owner extends World> extends Object<Owner> {
    Object<World> doIt() {
      return new BreaksDeepOwnership<This, World>();
    }
}
```

Here one can see how an object owned by someone can be passed around the entire object graph since its owner is `World`, while having permission to access inner workings of a particular private instance. This can lead to more than one path from the root of the object graph to the owned instance, thus breaking owners-as-dominators property.

Please observe that although Δ contains three possible syntactic categories in its domain ($\mathtt{X}, \mathtt{x}, l$), only type variables ($\mathtt{X}$) are covered by the WF-VAR rule. Also, most of this rules are also present implicitly in the FGJ+c type system in Chapter 2 since they are standard FGJ rules (except for the parts dealing with instance owners and with deep ownership).

Figure 9 shows FGO's subtyping rules. Apart from S-OWNER they are taken verbatim from FGJ. The first two rules enforce that the subtyping relationship is reflexive and transitive. The second two rules enforce that type variables are subtypes of their bounds and that the subtyping information for classes comes from their declarations.

The first of the two S-OWNER rules ensures that `World` forms the top of the owner class hierarchy which any package owner class extends directly. The second rule states that the location-specific owner class $\mathtt{This}_l$ extends the owner of the class whose instance is stored at location l. These two subtyping relationships together are required for the ownership invariant's definition of owner classes being *inside* one another presented later in Section 13. In particular, it ensures that `This` is *inside* `Owner` is *inside* `World` at all times. The owner nesting (*inside*) relationship is thus made to follow the owner subtyping relationship for the convenience of the ownership invariant's proof.

Subtyping:

$$\frac{}{\Delta \vdash \mathtt{T} \mathrel{<:} \mathtt{T}} \quad (\text{S-REFL}) \qquad \frac{\Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{T} \qquad \Delta \vdash \mathtt{T} \mathrel{<:} \mathtt{U}}{\Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{U}} \quad (\text{S-TRANS})$$

$$\frac{}{\Delta \vdash \mathtt{X} \mathrel{<:} \Delta(\mathtt{X})} \quad (\text{S-VAR}) \qquad \frac{\mathtt{class}\ \mathtt{C} < \overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}} > \ \triangleleft\ \mathtt{N}\,\{\ldots\}}{\Delta \vdash \mathtt{C} < \overline{\mathtt{T}} > \mathrel{<:} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}} \quad (\text{S-CLASS})$$

$$\frac{\mathtt{class}\ \mathtt{C} < \overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}} > \ \triangleleft\ \mathtt{N}\,\{\ldots\}}{\Delta \vdash \pi_{\mathtt{C}} \mathrel{<:} \mathtt{World}} \qquad \frac{l \in \mathit{dom}(\Delta)}{\Delta \vdash \mathtt{This}_l \mathrel{<:} \mathit{owner}_{\Delta}(\Delta(l))} \quad (\text{S-OWNER})$$

Figure 9: FGO Subtyping Rules

The rest of the owner class hierarchy (for location-specific owners and for owner variables) is built up during the type checking as discussed in class rule description in Section 9. Finally, please note that although Δ contains three possible syntactic categories in its domain ($\mathtt{X}$, $\mathtt{x}$, l), only locations (l) are covered by the second S-OWNER rule.

7 Expressions

Figure 10 shows the expression typing rules. These are the standard FGJ rules with added support for locations, assignment, `null`, and `let` expressions [10]. These rules define the types and possible well-formedness constraints for all possible expressions in the source of the FGO program or during its execution. They utilise both the environment Δ and the permission P denoting the current class or instance depending on the use of the expression typing rule. The reduction rules are discussed later in this chapter and contain the behaviour of object locations l present in some of the expressions. The expressions containing locations cannot occur in FGO program source and only appear during the reductions.

The T-FIELD rule constrains a field access. The index i is used to refer to the position of the field in the fields list returned by the *fields* function. This rule checks that the expression that is the receiver of the field access is a well typed FGO expression and that the field type is a well formed FGO type. This rule also ensures that the field exists in the corresponding class declaration and applies the *this* function check to make sure that if a field is private (has owner `This`) then it can only be accessed using a `this` call.

The T-FIELD-SET rule describes an expression that sets the value of a field. This rule checks that the receiver expression is well typed in FGO and that the expression being assigned is well typed. This rule also checks that the resulting type is a well formed FGO type and that the field requested exists. Finally, it applies the *this* function to ensure that fields owned by `This` are not used by other instances — similar to the T-FIELD rule above.

The T-METHOD rule — the most complex expression rule — checks that the method type parameters are OK in FGO or are valid owner variables (hence both a well-formedness check

(T-FIELD):

$$\frac{\begin{array}{c}\Delta; P \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}\ \mathrm{OK} \\ \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0)) <: \overline{\mathtt{T}}\,\overline{\mathtt{f}} \qquad \mathtt{T} = [\mathit{this}_P(\mathtt{e}_0)/\mathtt{This}]\mathtt{T}_i\end{array}}{\Delta; P \vdash \mathtt{e}_0.\mathtt{f}_i : \mathtt{T}}$$

(T-FIELD-SET):

$$\frac{\begin{array}{c}\Delta; P \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta; P \vdash \mathtt{e} : \mathtt{T} \qquad \Delta \vdash \mathtt{T}\ \mathrm{OK} \\ \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0)) <: \overline{\mathtt{T}}\,\overline{\mathtt{f}} \qquad \mathtt{T} = [\mathit{this}_P(\mathtt{e}_0)/\mathtt{This}]\mathtt{T}_i\end{array}}{\Delta; P \vdash \mathtt{e}_0.\mathtt{f}_i = \mathtt{e} : \mathtt{T}}$$

(T-METHOD):

$$\frac{\begin{array}{c}\forall \mathtt{V}' \in \overline{\mathtt{V}} : (\Delta \vdash \mathtt{V}'\ \mathrm{OK} \ \vee\ \mathtt{V}' <: \mathtt{World}) \ \wedge\ (\ \mathit{owner}_\Delta(\mathtt{T}_0) <: \mathit{owner}_\Delta(\mathtt{V}')\ ) \\ \mathit{mtype}(\mathtt{m},\ \mathit{bound}_\Delta(\mathtt{T}_0)) = <\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}>\overline{\mathtt{U}} \rightarrow \mathtt{U} \\ \Delta; P \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \Delta; P \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}\ \mathrm{OK} \\ \mathtt{T} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}},\ \mathit{this}_P(\mathtt{e}_0)/\mathtt{This}]\mathtt{U} \qquad \Delta \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}},\ \mathit{this}_P(\mathtt{e}_0)/\mathtt{This}]\overline{\mathtt{P}} \\ \Delta \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}},\ \mathit{this}_P(\mathtt{e}_0)/\mathtt{This}]\overline{\mathtt{U}}\end{array}}{\Delta; P \vdash \mathtt{e}_0.\mathtt{m}<\overline{\mathtt{V}}>(\overline{\mathtt{e}}) : \mathtt{T}}$$

(T-CAST):

$$\frac{\Delta \vdash \mathtt{N}\ \mathrm{OK} \qquad \Delta; P \vdash \mathtt{e}_0 : \mathtt{T}_0}{\Delta; P \vdash (\mathtt{N})\mathtt{e}_0 : \mathtt{N}}$$

(T-LET):

$$\frac{\Delta; P \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta,\ \mathtt{x} : \mathtt{T}_0; P \vdash \mathtt{e} : \mathtt{T}}{\Delta; P \vdash \mathtt{let}\ \mathtt{x} = \mathtt{e}_0\ \mathtt{in}\ \mathtt{e} : \mathtt{T}}$$

$$\frac{\Delta; l \vdash \mathtt{e} : \mathtt{T}}{\Delta; P \vdash l > \mathtt{e} : \mathtt{T}}\ \text{(T-CONTEXT)} \qquad \frac{\Delta \vdash \mathtt{N}\ \mathrm{OK}}{\Delta; P \vdash \mathtt{new}\ \mathtt{N}() : \mathtt{N}}\ \text{(T-NEW)}$$

$$\frac{\Delta \vdash \mathtt{T}\ \mathrm{OK}}{\Delta; P \vdash \mathtt{error} : \mathtt{T}}\ \text{(T-ERROR)} \qquad \frac{\Delta \vdash \mathtt{T}\ \mathrm{OK}}{\Delta; P \vdash \mathtt{null} : \mathtt{T}}\ \text{(T-NULL)}$$

$$\frac{}{\Delta; P \vdash \mathtt{x} : \Delta(\mathtt{x})}\ \text{(T-VAR)} \qquad \frac{}{\Delta; P \vdash l : \Delta(l)}\ \text{(T-LOC)}$$

Figure 10: FGO Expression Typing

and allowing $\mathtt{V}'$ to be a subtype of `World`). These checks allow "naked" owners discussed in Chapter 2. This rule then checks that owners of method type parameters are properly nested inside the owner of the class that the method is part of (supplied via permission P) — this is required by the owners-as-dominators property in the same manner as the grey check in WF-TYPE rule earlier.

The rest of the checks are similar to FGJ: the T-METHOD rule looks up the method's type, checks that the method arguments are well typed FGO expressions and that the resulting type is OK in FGO. Finally, the rule applies the *this* function to the method's return type, type parameters, and arguments to make sure that the methods using private (with owner `This`) instances are only called on `this`.

The T-CAST rule checks if the expression being cast is well typed. The reduction rules will check the actual type of the expression and if it matches the requested type, failing with run-time error if necessary. It is possible to avoid some run-time casting failures using a static check whether the upcast or downcast is valid and failing to type check if neither succeed — this is the technique employed by FGJ [9]. But since in the presence of locations and `null` I already model the `error` arising in the case of `null` dereferences, one may as well model casting `error` in the same way for simplicity.

The T-LET rule checks that the expression that gets the local variable (`e`) and the expression that is the local variable ($\mathtt{e}_0$) are both well typed in FGO. An appropriate type for the local variable `x` is stored in the typing environment Δ before type checking the expression `e`.

The T-CONTEXT rule only arises during the type preservation proof — it ensures that the expression is well typed with respect to the receiver location l. This makes it possible for the other rules to replace the owner class `This` with a correct location-specific owner $\mathtt{This}_l$. This is one of the expressions that cannot appear in the source of a FGO program.

The T-NEW rule checks that the type of the instance being created is a well formed FGO type. The T-ERROR and T-NULL rules make `error` and `null` well formed FGO types. Finally, the T-VAR and T-LOC rules lookup an appropriate type for variables and locations using the typing environment Δ.

The important observation about the expression rules is that these are not sufficient to ensure ownership — an additional set of visibility rules is required to prevent incorrect accesses for different owners. These are presented in the next section.

8 Visibility

Visibility plays an essential role in FGO and FGC. Term visibility is similar to FGC and ConfinedFGJ [13] rules except for accounting for the additional expressions present in FGO. Owner and type visibility is FGC and FGO specific and takes into account owner class `This` for the FGO version of the rules.

Figure 11 shows the owner visibility rule that checks if an owner `O` is visible inside class `C`. This is the case if the owner is `World`, belongs to the same package as `C`, or is an owner of one of the type parameters used when instantiating `C`. Supplying an actual owner parameter to a class gives that class permission to access everything owned by that parameter. This, for example, can allow a type polymorphic class to have private access to more than one package.

FGO does not restrict visibility of the `This` owner class, and relies on the *this* function described earlier to stop illegal uses of `This`. Type visibility checks the owner of a given type for visibility.

Term visibility (Figure 12) recursively checks all the types involved in the possible expres-

$$visible_\Delta(\mathtt{T},\ \mathtt{C}) \quad = \quad visible_\Delta(owner_\Delta(\mathtt{T}),\ \mathtt{C}) \qquad \text{(V-TYPE)}$$

$$visible_\Delta(\mathtt{O},\ \mathtt{C}) \quad = \quad \mathtt{O} \in owners(\mathtt{C}) \cup \{\mathtt{This}, \pi_\mathtt{C}, \mathtt{World}\} \qquad \text{(V-OWNER)}$$

$$\text{where } owners(\mathtt{C}) \quad = \quad \begin{cases} \{owner_\Delta(\mathtt{N}') \mid \mathtt{N}' \in \overline{\mathtt{N}}, \mathtt{N}\}, \\ \qquad \text{if } CT(\mathtt{C}) = \mathtt{class}\ \mathtt{C} < \overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}} > \triangleleft \mathtt{N}\{\ldots\} \\ \{\mathtt{X}^\mathtt{O}\} \cup \{owner_\Delta(\mathtt{N}') \mid \mathtt{N}' \in \overline{\mathtt{N}}\}, \\ \qquad \text{if } CT(\mathtt{C}) = \mathtt{class}\ \mathtt{C} < \overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}},\ \mathtt{X}^\mathtt{O} \triangleleft \mathtt{N}^\mathtt{O} > \triangleleft \mathtt{N}\{\ldots\} \end{cases}$$

Figure 11: FGO Type and Owner Visibility Rules

sions of FGO to make sure that they are visible in a given class C. Since these checks are performed on class declarations and are not required during reduction, locations are not present in these expressions (which for example means that T-CONTEXT expression is not covered by the visibility rules).

9 Classes and Methods

Figure 13 presents FGO class definition rules for standard FGO classes (FGO-CLASS-PURE) and manifest classes described earlier in Chapter 2 (FGO-CLASS-MANIFEST). The figure also presents the FGO method definition rule.

The first rule, FGO-CLASS-PURE, defines a standard FGO class. The first line initialises the FGO type environment (Δ) with information about subtype relationships read from the class declaration, and it also initialises the missing owner variables as described in the placeholder owners function in the end of this section. The second line ensures that the current class's owner ($\mathtt{X}^\mathtt{O}$) is nested inside the other owners. This grey clause allows us to check deep ownership when proving an ownership invariant. The second line also checks the well-formedness of the super class and the types of the fields. The third line of the rule allows "naked" owners as type parameters and ensures that the method declarations are valid for the current class and its owner. Finally, the fourth line ensures that the super class (N) has the same owner ($\mathtt{X}^\mathtt{O}$) and checks that the current classes' principal owner bound ($\mathtt{N}^\mathtt{O}$), field types, and type parameters are all visible inside the current class. FGO checks the visibility of the owner bound $\mathtt{N}^\mathtt{O}$ rather than owner $\mathtt{X}^\mathtt{O}$ to disallow declarations that might try for example to confine a class declared in package p to package u making it unusable. The latter will be prevented by the visibility check since owner U is invisible inside package p.

The second rule deals with the manifest FGO classes — classes that have a fixed owner taken from one of their superclasses. These are usually used to represent standard FGJ (or Java) classes

$$\frac{\Delta;\mathtt{C}\vdash \mathit{visible}(\mathtt{e_0}) \qquad \Delta;\mathtt{C}\vdash \mathtt{e.f_i} : \mathtt{T_0} \qquad \mathit{visible}_{\Delta}(\mathtt{T_0},\ \mathtt{C})}{\Delta;\mathtt{C}\vdash \mathit{visible}(\mathtt{e_0.f_i})} \quad \text{(V-FIELD)}$$

(V-FIELD-SET):

$$\frac{\Delta;\mathtt{C}\vdash \mathit{visible}(\mathtt{e_0}) \quad \Delta;\mathtt{C}\vdash \mathtt{e_0.f_i} : \mathtt{T_0} \quad \mathit{visible}_{\Delta}(\mathtt{T_0},\ \mathtt{C}) \quad \Delta;\mathtt{C}\vdash \mathtt{e} : \mathtt{T} \quad \mathit{visible}_{\Delta}(\mathtt{T},\ \mathtt{C})}{\Delta;\mathtt{C}\vdash \mathit{visible}(\mathtt{e_0.f_i} = \mathtt{e})}$$

(V-METHOD):

$$\frac{\Delta;\mathtt{C}\vdash \mathtt{e.m}(\overline{\mathtt{e_0}}) : \mathtt{T_0} \qquad \mathit{visible}_{\Delta}(\mathtt{T_0},\ \mathtt{C}) \qquad \Delta;\mathtt{C}\vdash \mathit{visible}(\mathtt{e_0}) \qquad \Delta;\mathtt{C}\vdash \mathit{visible}(\overline{\mathtt{e}})}{\Delta;\mathtt{C}\vdash \mathit{visible}(\mathtt{e_0.m}(\overline{\mathtt{e}}))}$$

$$\frac{\Delta;\mathtt{C}\vdash \mathit{visible}(\mathtt{e_0}) \qquad \mathit{visible}_{\Delta}(\mathtt{N},\ \mathtt{C})}{\Delta;\mathtt{C}\vdash \mathit{visible}((\mathtt{N})\ \mathtt{e_0})} \quad \text{(V-CAST)}$$

$$\frac{\begin{array}{c}\Delta;\mathtt{C}\vdash \mathtt{e_0} : \mathtt{T_0} \qquad \mathit{visible}_{\Delta}(\mathtt{T_0},\ \mathtt{C}) \qquad \Delta;\mathtt{C}\vdash \mathtt{x} : \mathtt{T_x} \\ \mathit{visible}_{\Delta}(\mathtt{T_x},\ \mathtt{C}) \qquad \Delta;\mathtt{C}\vdash \mathtt{e} : \mathtt{T} \qquad \mathit{visible}_{\Delta}(\mathtt{T},\ \mathtt{C})\end{array}}{\Delta;\mathtt{C}\vdash \mathit{visible}(\mathtt{let\ x} = \mathtt{e_0\ in\ e})} \quad \text{(V-LET)}$$

$$\frac{\Delta;\mathtt{C}\vdash \mathtt{x} : \mathtt{T} \qquad \mathit{visible}_{\Delta}(\mathtt{T},\ \mathtt{C})}{\Delta;\mathtt{C}\vdash \mathit{visible}(\mathtt{x})} \quad \text{(V-VAR)} \qquad \frac{\mathit{visible}_{\Delta}(\mathtt{N},\ \mathtt{C})}{\Delta;\mathtt{C}\vdash \mathit{visible}(\mathtt{new\ N}())} \quad \text{(V-NEW)}$$

$$\frac{}{\Delta;\mathtt{C}\vdash \mathit{visible}(\mathtt{error})} \quad \text{(V-ERROR)} \qquad \frac{}{\Delta;\mathtt{C}\vdash \mathit{visible}(\mathtt{null})} \quad \text{(V-NULL)}$$

Figure 12: FGO Term Visibility Rules

with a fixed owner `World`. This rule is very similar to the previous "pure" rule, except that it takes into account that the owner is looked up from the superclass and is not present explicitly in the current class declaration. It is interesting to note that although FGO uses syntax to distinguish *the* classes' owner, the other type parameters of both classes and methods *can be* either types or owners without any special treatment from the type system — a benefit of Generic Ownership merging type parameters and ownership types.

The first line of the FGO-CLASS-MANIFEST rule starts by initialising the FGO type environment (Δ) with the subtype information and initialising placeholder owners as described in the end of this section. The second line enforces the owner nesting required of any ownership type system providing deep ownership and checks that the types involved are well formed FGO types. The third line checks the method declarations and the final, fourth line checks that all the types involved are visible in the current class.

To summarise, both of the class rules check that (1) all the types involved (types of fields and type parameters) are *visible* within the domain of the owner of the class being declared or its superclass; (2) all the types are *well formed* FGO types; and (3) all the method declarations are valid. The grey clauses ensure that the owner nesting (the distinguished owner is *inside* the owners of the other type parameters) is preserved for the purposes of the deep ownership invariant theorem proven below in the Section 13. The difference between the two class rules

FGO Class Definition (FGO-CLASS-PURE):

$$\dfrac{\begin{array}{c}\Delta = \{\overline{\mathtt{X}} <: \overline{\mathtt{N}}, \mathtt{X}^{\mathtt{O}} <: \mathtt{N}^{\mathtt{O}}, \mathtt{This} <: \mathtt{X}^{\mathtt{O}}\} \cup \mathit{placeholderowners}_{\Delta}(\overline{\mathtt{N}}) \\ \forall \mathtt{X}' \in \overline{\mathtt{X}} : \Delta = \Delta \cup \{\mathtt{X}^{\mathtt{O}} <: \mathit{owner}_{\Delta}(\mathtt{X}')\} \qquad \Delta \vdash \mathtt{N}, \overline{\mathtt{T}}\ \mathsf{OK} \\ \forall \mathtt{N}' \in \overline{\mathtt{N}} : \Delta \vdash \mathtt{N}'\ \mathsf{OK} \vee \Delta \vdash \mathtt{N}' <: \mathtt{World} \qquad \Delta \vdash \overline{\mathtt{M}}\ \mathtt{FGO\ IN\ C,\ X}^{\mathtt{O}} \\ \mathtt{N} = \mathtt{D} < \overline{\mathtt{T}'}, \mathtt{X}^{\mathtt{O}} > \qquad \mathit{visible}_{\Delta}(\mathtt{N}^{\mathtt{O}},\ \mathtt{C}) \qquad \mathit{visible}_{\Delta}(\overline{\mathtt{T}},\ \mathtt{C}) \qquad \mathit{visible}_{\Delta}(\overline{\mathtt{N}},\ \mathtt{C})\end{array}}{\mathtt{class\ C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}},\ \mathtt{X}^{\mathtt{O}} \lhd \mathtt{N}^{\mathtt{O}} > \lhd \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}\ \mathtt{FGO}}$$

FGO (Manifest) Class Definition (FGO-CLASS-MANIFEST):

$$\dfrac{\begin{array}{c}\Delta = \{\overline{\mathtt{X}} <: \overline{\mathtt{N}}, \mathtt{This} <: \mathit{owner}_{\Delta}(\mathtt{N})\} \cup \mathit{placeholderowners}_{\Delta}(\overline{\mathtt{N}}) \\ \forall \mathtt{X}' \in \overline{\mathtt{X}} : \Delta = \Delta \cup \{\mathit{owner}_{\Delta}(\mathtt{N}) <: \mathit{owner}_{\Delta}(\mathtt{X}')\} \qquad \Delta \vdash \mathtt{N}, \overline{\mathtt{T}}, \overline{\mathtt{N}}\ \mathsf{OK} \\ \Delta \vdash \overline{\mathtt{M}}\ \mathtt{FGO\ IN\ C},\ \mathit{owner}_{\Delta}(\mathtt{N}) \\ \mathit{visible}_{\Delta}(\mathtt{N},\ \mathtt{C}) \qquad \mathit{visible}_{\Delta}(\overline{\mathtt{T}},\ \mathtt{C}) \qquad \mathit{visible}_{\Delta}(\overline{\mathtt{N}},\ \mathtt{C})\end{array}}{\mathtt{class\ C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \lhd \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}\ \mathtt{FGO}}$$

FGO Method Definition (FGO-METHOD):

$$\dfrac{\begin{array}{c}\Delta = \Delta \cup \{\overline{\mathtt{Y}} <: \overline{\mathtt{P}}\} \cup \mathit{placeholderowners}_{\Delta}(\overline{\mathtt{N}}) \\ \forall \mathtt{Y}' \in \overline{\mathtt{Y}} : \Delta = \Delta \cup \{\mathtt{C}^{\mathtt{O}} <: \mathit{owner}_{\Delta}(\mathtt{Y}')\} \qquad \Delta \vdash \overline{\mathtt{T}}, \mathtt{T}\ \mathsf{OK} \\ \Delta \vdash \forall \mathtt{P}' \in \overline{\mathtt{P}} : (\mathtt{P}'\ \mathsf{OK}) \vee (\mathtt{P}' <: \mathtt{World}) \qquad \mathtt{class\ C} < \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} > \ \lhd\ \mathtt{N}\ \{\ldots\} \\ \mathit{visible}_{\Delta}(\overline{\mathtt{T}},\ \mathtt{C}) \qquad \mathit{visible}_{\Delta}(\mathtt{T},\ \mathtt{C}) \qquad \mathit{visible}_{\Delta}(\overline{\mathtt{P}},\ \mathtt{C}) \\ \Delta,\ \overline{\mathtt{x}}\ : \overline{\mathtt{T}},\ \mathtt{this}\ :\ \mathtt{C} < \overline{\mathtt{X}} > ; \mathtt{C} \vdash \mathit{visible}(\mathtt{e}_0) \\ \Delta,\ \overline{\mathtt{x}}\ : \overline{\mathtt{T}},\ \mathtt{this}\ :\ \mathtt{C} < \overline{\mathtt{X}} > ; \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{S} \\ \Delta \vdash \mathtt{S} <: \mathtt{T} \qquad \mathit{override}(\mathtt{m},\ \mathtt{N},\ < \overline{\mathtt{Y}} \lhd \overline{\mathtt{P}} > \overline{\mathtt{T}} \to \mathtt{T})\end{array}}{\Delta \vdash < \overline{\mathtt{Y}}\ \lhd\ \overline{\mathtt{P}} >\ \mathtt{T\ m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\{\ \mathtt{return\ e}_0;\ \}\ \mathtt{FGO\ IN\ C},\ \mathtt{C}^{\mathtt{O}}}$$

Figure 13: FGO Class and Method Rules

lies with the fact that the owner of the class has to be looked up for a manifest version and is provided explicitly for the pure version.

The method typing rule starts by adding additional subtyping relationships and additional placeholder owners to the environment Δ supplied by the class declaration rule. The second line checks the owner nesting required for the deep ownership with respect the the owner of the class inside which the method being checked is declared. The second line also checks the well-formedness of the types of method arguments and method's return type. The third line allows the method type parameters to be owner classes on their own and specifies the class declaration used for the class inside which the current method is declared so that the method rule can refer to the type parameters used in the class declaration. The fourth line checks the visibility of all the types involved in method declaration. The fifth and sixth lines check the visibility and well-formedness of the type of the method expression. To perform this checks on the expression, the assumptions on the left about the environment include the types for `this` and the method parameters. The checks also specify the permission to be the current class `C` so that this context can be used to detect any illegal accesses to package owner classes. Finally, the sixth line, in

Placeholder Owners Function:

$$\begin{array}{lcl}
placeholderowners_{\Delta}(\mathtt{C}\mathtt{<}\bar{\mathtt{T}}\mathtt{>}) & = & \{owner_{\Delta}(\mathtt{C}\mathtt{<}\bar{\mathtt{T}}\mathtt{>})\mathtt{<:}\ \mathtt{World}\} \cup \\
 & & \cup\ placeholderowners_{\Delta}(\bar{\mathtt{T}}) \\
 & & \text{if } owner_{\Delta}(\mathtt{C}\mathtt{<}\bar{\mathtt{T}}\mathtt{>}) \notin dom(\Delta) \\
placeholderowners_{\Delta}(\mathtt{C}\mathtt{<}\bar{\mathtt{T}}\mathtt{>}) & = & placeholderowners_{\Delta}(\bar{\mathtt{T}}) \\
 & & \text{otherwise} \\
placeholderowners_{\Delta}(\mathtt{X}) & = & \{\}
\end{array}$$

Figure 14: FGO Placeholder Owners Function

exactly the same manner as FGJ, checks the type of the expression with respect to the method return type and verifies the validity of method overriding (if applicable) — this preserves Java's requirement that the method with the same name and arguments has to return the result that is a proper subtype of the super class's method result type it overrides. The contravariance of method arguments is omitted for simplicity.

Additionally, FGO allows class declarations to use implicit *placeholder owner parameters* in formal type parameter bounds. Consider:

```
class List<E extends Foo<FO>, Owner extends World> { ... }
```

Here, `List`'s formal parameter `E` can only be bound by the actual type parameters that are subclasses of `Foo`; the owner of that type parameter (`FO`) can be different from the owner of the list (`Owner`) — although `FO` has to be "‘inside"’ `Owner`. Since FGO (like FGJ) requires every type variable to be bound, one needs to make sure that the FGO type environment contains an appropriate mapping for implicit placeholder parameters like `FO`. The function *placeholderowners* in Figure 14 does exactly that, by making sure that any (placeholder) owner used in the type bounds is recorded in the FGO environment Δ as being a subtype of `World` (unless the owner already has a bound declared for it explicitly, in which case it is used instead of `World`). This ensures that every owner present in the class declaration is bound, whether implicitly or explicitly. The FGO well formed types that are not owners are not affected by the presence of owners, since all of the owners are subtypes of `World` and not subtypes of `Object<O>` — these two hierarchies are shown in Figure 1 in Section 2, Chapter 2. `Any` type bound mechanism in FGJ+c described in the previous chapter achieves similar goal to that of *placeholderowners* function by allowing any owner to be used in a particular type parameter in an FGJ+c class.

Figure 14 shows the *placeholderowners* function definition. *placeholderowners* accepts a type. For every nonvariable type *placeholderowners* adds its owner (the last type variable in a pure FGO class) to the type environment Δ unless it is already present. Then, for every type parameter, *placeholderowners* recursively calls itself to add any additional owners than might not be recorded in Δ. The *placeholderowners* function recurses into the subexpressions for complex type expressions until everything has been expanded.

Store Well-Formedness (FGO-STORE-WF):

$$\frac{\forall l \in \mathit{dom}(\Delta) : \Delta \vdash \ \Delta(l)\ \mathsf{OK}}{\Delta\ \mathsf{OK}}$$

Store Typing (FGO-STORE):

$$\frac{\begin{array}{c}\Delta\ \mathsf{OK} \quad \mathit{dom}(\Delta) = \mathit{dom}(S) \quad S[l] = \mathtt{N}(\overline{v}) \iff \Delta(l) = \mathtt{N} \\ (S[l,i] = l') \wedge (\mathit{fields}(\Delta(l)) = \overline{\mathtt{T}}\,\overline{\mathtt{f}}) \implies \Delta \vdash \Delta(l') <: [\mathtt{This}_l/\mathtt{This}]\mathtt{T}_i \\ (S[l,\ i] = l') \implies \Delta \vdash \ \Delta(l')\ \mathsf{OK}\end{array}}{\Delta \vdash S}$$

Figure 15: FGO Store

10 Representing the Heap

This section addresses the representation of the heap in the FGO type system. The store typing rules shown in Figure 15 are mostly standard [6, 1]. The mapping Δ contains the types for each location and the FGO-STORE-WF rule in Figure 15 ensures that every one of the types is well formed. The mapping S maps the locations to the types of classes that are instantiated at these locations together with further location values for each field in these instances. The main FGO-STORE rule ensures that not only the types are well formed, but also that each field location's type is well formed FGO type and is a correct subtype of the declared field type. It is interesting to note that the FGO type system does not have any explicit ownership constraints in the store rule — the benefit of ownership information being part of the type is that subtyping ensures that none of the ownership constraints are broken. Finally, in this rule FGO only considers the part of the environment Δ that maps locations l to their types — this is used to validate the domain of S. The difference between $S[l]$ and $\Delta(l)$ lies in the fact that $S[l]$ returns the type of the location together with location values stored in each field (e.g., $\mathtt{N}(\overline{v})$) and $\Delta(l)$ returns the type of the location (e.g., $\mathtt{N}$).

11 Reduction Rules

Figure 16 shows the small step semantics reduction rules. Again, these are standard given the expressions that FGO supports. The notation shows how and expression $\mathtt{e}$ and store S together reduce to a new expression $\mathtt{e}'$ with possibly an updated store S': $\mathtt{e}, S \rightarrow \mathtt{e}', S'$. l denotes locations and v denotes values of particular fields (which can be `null`).

R-NEW reduces a newly created instance to the newly created location l, at first storing only `null` for each of the classes' fields. R-FIELD uses the *fields* lookup function and store S to reduce to the value stored in a particular field. R-FIELD-SET modifies the store S by replacing the value stored in a particular field $\mathtt{f}_i$ and similarly to the behaviour or R-FIELD this rule reduces to the new value that was just stored. R-METHOD looks up the appropriate method body using the *mbody* function and reduces to an expression of the form $l > \mathtt{e}$ where the actual

(R-NEW):

$$\frac{l \notin \mathit{dom}(S) \qquad S' = S[l \mapsto \mathtt{N}(\overline{\mathtt{null}})] \qquad |\overline{\mathtt{null}}| = |\mathit{fields}(\mathtt{N})|}{\mathtt{new}\ \mathtt{N}(), S \longrightarrow l, S'}$$

(R-FIELD):

$$\frac{S[l] = \mathtt{N}(\overline{v}) \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{l.\mathtt{f}_i, S \longrightarrow v_i, S}$$

(R-FIELD-SET):

$$\frac{S[l] = \mathtt{N}(\overline{v}) \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad S' = S[l \mapsto [v/v_i]\mathtt{N}(\overline{v})]}{l.\mathtt{f}_i = v, S \longrightarrow v, S'}$$

(R-METHOD):

$$\frac{S[l] = \mathtt{N}(\overline{v_l}) \qquad \mathit{mbody}(\mathtt{m}<\overline{\mathtt{V}}>, \mathtt{N}) = \overline{\mathtt{x}}.\mathsf{e}_0}{l.\mathtt{m}<\overline{\mathtt{V}}>(\overline{v}), S \longrightarrow l \ > \ [\overline{v}/\overline{\mathtt{x}}, l/\mathtt{this}, \mathtt{This}_l/\mathtt{This}]\mathsf{e}_0, S}$$

(R-*-NULL):

$$\frac{}{\mathtt{null}.\mathtt{m}<\overline{\mathtt{V}}>(\overline{v}), S \longrightarrow \mathtt{error}, S}$$

$$\frac{}{\mathtt{null}.\mathtt{f}_i, S \longrightarrow \mathtt{error}, S} \qquad \frac{}{\mathtt{null}.\mathtt{f}_i = v, S \longrightarrow \mathtt{error}, S}$$

$$\frac{S[l] = \mathtt{N}(\overline{v}) \qquad \mathtt{N} <: \mathtt{P}}{(\mathtt{P})l, S \longrightarrow l, S}\ \text{(R-CAST)} \qquad \frac{S[l] = \mathtt{N}(\overline{v}) \qquad \mathtt{N} \not<: \mathtt{P}}{(\mathtt{P})l, S \longrightarrow \mathtt{error}, S}\ \text{(R-BAD-CAST)}$$

$$\frac{}{l \ > \ v, S \longrightarrow v, S}\ \text{(R-CONTEXT)} \qquad \frac{}{\mathtt{let}\ \mathtt{x}\ =\ v\ \mathtt{in}\ \mathsf{e}_0, S \longrightarrow [v/\mathtt{x}]\mathsf{e}_0, S}\ \text{(R-LET)}$$

Figure 16: FGO Reduction Rules

arguments are substituted in place of formal parameters, current location l is substituted in place of `this`, and current instance's owner $\mathtt{This}_l$ is substituted in place of owner `This`.

Neither field nor method accesses are allowed on `null` — which is dealt with by R-*-NULL reduction rules reducing to `error`. FGO does not have to explicitly prohibit l from being `null` in the field and method access rules due to the FGO syntax only allowing values (v) to be `null` and not locations (l). The FGO's `error` captures non-type errors that cannot be captured easily by a decidable type system, on the other hand `error` is well-typed to allow type preservation proof for the FGO type system to deal only with well typed expressions.

R-CAST and R-BAD-CAST deal with type casts by allowing them if the type of the location is a subtype of the cast target type and reducing to `error` otherwise. R-CONTEXT removes the no longer necessary information about the receiver of the method call, once the reduction has evaluated the expression's right hand side to a value v. Finally R-LET substitutes every occurrence of variable `x` in the expression e_0 with its value v.

Figure 17 presents the FGO's context reduction rule that defines the evaluation order for the FGO programs. An evaluation context E is an expression with a hole ($[\,]$) so that the expression $E[\mathsf{e}]$ results from placing expression `e` into the hole of E. The rule uses the evaluation context E to define which subexpression of an FGO expression `e` should be reduced first. Evaluation

Reduction Context Expression:

$E ::=$

$[\,]$
$E.\mathtt{f}$
$E.\mathtt{f} = \mathtt{e}$
$l.\mathtt{f} = E$
$E.\mathtt{m}<\overline{\mathtt{T}}>(\overline{\mathtt{e}})$
$\mathtt{e}.\mathtt{m}<\overline{\mathtt{T}}>(\overline{l}, E, \overline{\mathtt{e}'})$
$(\mathtt{N})E$
$l > E$
$\mathtt{let}\ \mathtt{x} = E\ \mathtt{in}\ \mathtt{e}$

Context Reduction Rule:

$$\frac{\mathtt{e}, S \rightarrow \mathtt{e}', S'}{E[\mathtt{e}], S \rightarrow E[\mathtt{e}'], S'}$$

Figure 17: FGO Context Reduction Rule

context E takes expression $\mathtt{e}$ as its argument and replaces it with another expression with its argument expression $\mathtt{e}$ placed in the appropriate position that needs to be reduced first. For example, $E[\mathtt{e}_0]$ can be replaced with $\mathtt{e}_0.\mathtt{f} = \mathtt{e}$ which means that the expression on the left hand side of the field assignment has to be reduced before the expression being assigned to the field.

In the method context reduction the arguments of the method call are evaluated left to right as shown by the location replacing the ones on the left of the current argument (denoted by E) being evaluated. The context reduction follows small step semantics since the notation in Figure 17 is no more than a shorthand for a large number of context reduction rules.

12 Type Soundness

In this section I present the proofs of the *Type Preservation* Theorem and the *Progress* Theorem. Together they prove type soundness (as defined by Wright and Felleisen [12]) of the FGO type system. In contrast, FGC as described in Chapter 2 does not need a full type soundness proof because FGJ+c is a subset of FGJ. FGJ+c relies on the proven type soundness of FGJ for a guarantee of a safe and reliable execution of any FGJ+c (and thus FGJ) program. The type soundness result proves the absence of ordinary type errors — it does not prove the ownership guarantees. The next section presents the proofs of the ownership guarantees provided by the FGO.

The type preservation theorem (also known as subject reduction theorem) proves that for every reduction possible for any FGO expression, the resulting expression is always going to be the subtype of the original expression. The progress theorem shows that any FGO expression will always reduce to a value or produce an error — in other words, FGO programs will never get “stuck” not being able to apply any of the reduction rules.

The type soundness shows that the FGO language is a safe and predictable tool for writing programs. One cannot and need not conclude more than this from the type soundness. On the other hand, one can prove a variety of other properties and invariants that FGO preserves for all of its programs. I show such ownership invariants in the next section.

12.1 Type Preservation Theorem

The type preservation theorem proves that if any FGO expression reduces to another FGO expression then the latter is always a subtype of the former. Before stating the theorem, lets define a shorthand for a well typed expression in a well typed store.

Definition 1. $\Delta; P \vdash \texttt{e}, S \ : \ \texttt{T} \equiv (\Delta; P \vdash \texttt{e} \ : \ \texttt{T}) \ \wedge \ (\Delta \vdash S)$

Theorem 1. ***(Type Preservation)*** *If* $\Delta; P \vdash \texttt{e}, S \ : \ \texttt{T}$ *and* $\texttt{e}, S \longrightarrow \texttt{e}', S'$*, then* $\exists \Delta' \supseteq \Delta$ *and* $\exists \texttt{T}' <: \texttt{T}$ *such that* $\Delta'; P \vdash \texttt{e}', S' \ : \ \texttt{T}'$.

Proof. Using structural induction on the reduction rules in Figure 16, as follows.

R-NEW

$$\boxed{\dfrac{l \notin \mathit{dom}(S) \qquad S' = S[l \mapsto \texttt{N}(\overline{\texttt{null}})] \qquad |\overline{\texttt{null}}| = |\mathit{fields}(\texttt{N})|}{\texttt{new N}(), S \longrightarrow l, S'} \quad \text{(R-NEW)}}$$

Expression: By T-NEW one has $\texttt{e} = \texttt{new N}() : \texttt{T}$ where $\texttt{T} = \texttt{N}$. By T-LOC $\texttt{e}' = l : \texttt{T}'$ where $\texttt{T}' = \Delta'(l)$. By FGO-STORE, $\Delta'(l) = \texttt{N}$ if and only if $S'[l] = \texttt{N}(\overline{v})$, but the latter holds by definition in R-NEW since the correct number of $\mathit{fields}(\texttt{N})$ are initialised to `null`. Therefore $\texttt{T}' = \texttt{N}$ and $\texttt{T} = \texttt{N} = \texttt{T}'$. Hence by S-REFL $\texttt{T}' <: \texttt{T}$ as required.

Store: Define $\Delta' = \Delta \cup \{l \rightarrow \texttt{N}\}$. I show that $\Delta' \vdash S'$ using the FGO-STORE rule.

To satisfy the first line of the FGO-STORE rule. By definition of Δ' and S' ($S' = S[l \mapsto \texttt{N}(\overline{\texttt{null}})]$), $\mathit{dom}(S') = \mathit{dom}(S) \cup \{l\} = \mathit{dom}(\Delta) \cup \{l\} = \mathit{dom}(\Delta')$ — where Δ is restricted to the mapping of locations to their types only. $S'[l] = \texttt{N}(\overline{v})$ — where all values have been set to `null`. $\Delta'[l] = \texttt{N}$ by definition of Δ'. By T-NEW $\Delta \vdash \texttt{N}$ OK and by definition $\Delta'(l) = \texttt{N}$, and hence $\Delta \vdash \Delta'(l)$ OK and by FGO-STORE-WF $\Delta \vdash \Delta'$ OK.

To satisfy the second line of the FGO-STORE rule. Consider any field i in $\mathit{fields}(\Delta'(l)) = \mathit{fields}(\texttt{N}) = \overline{\texttt{T}}\ \overline{\texttt{f}}$. All the newly added fields are set to `null` and, by T-NULL, `null` is a subtype of any type, including $\texttt{T}_i$.

Finally, to satisfy the third line of the FGO-STORE rule. By FGO-STORE-WF, for any location l' in the new store S', $\Delta' \vdash \Delta'(l')$ OK since if $l' \neq l$, then $l' \in \Delta$ and $\Delta \vdash S$ makes $\Delta(l')$ OK and $\Delta'(l') = \Delta(l')$; and for $l' = l$, it is already established that $\Delta'(l)$ OK. Therefore $\Delta' \vdash S'$ as required.

R-FIELD

$$\boxed{\frac{S[l] = \mathtt{N}(\overline{v}) \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\,\overline{\mathtt{f}}}{l.\mathtt{f}_i, S \rightarrow v_i, S} \quad \text{(R-FIELD)}}$$

Expression: By T-FIELD one has $\mathtt{e} = l.\mathtt{f}_i : \mathtt{T}$ where $\mathtt{T} = [\mathit{this}_P(l)/\mathtt{This}]\mathtt{T}_i$. By T-LOC, $\mathtt{e}' = v_i : \mathtt{T}'$ where $\mathtt{T}' = \Delta'(v_i)$ if $v_i \neq \mathtt{null}$. By FGO-STORE and since $\Delta' = \Delta$, one has $\Delta'(v_i) <: [\mathtt{This}_l/\mathtt{This}]\mathtt{T}_i$ or if $v_i = \mathtt{null}$ then by T-NULL $\mathtt{null} <: \mathtt{T}_i$. Furthermore, before v_i is stored in the store with R-FIELD-SET, T-FIELD-SET ensures that $\mathtt{This}$ is substituted by $\mathit{this}_P(l)$, hence $\mathtt{T}' <: [\mathit{this}_P(l)/\mathtt{This}]\mathtt{T}_i$ and not just $\mathtt{T}' <: \mathtt{T}_i$. Therefore $\mathtt{T}' <: \mathtt{T}$ as required. Note that $\mathit{this}_P(l)$ will return $\mathtt{This}_l$ since at run-time the only value of permission P allowing the types to be well-formed is l.

Store: Define $\Delta' = \Delta$ so that $\Delta \subseteq \Delta'$, to show that $\Delta' \vdash S'$, one needs $\Delta \vdash S$, which already holds.

R-FIELD-SET

$$\boxed{\frac{S[l] = \mathtt{N}(\overline{v}) \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\,\overline{\mathtt{f}} \qquad S' = S[l \mapsto [v/v_i]\mathtt{N}(\overline{v})]}{l.\mathtt{f}_i = v, S \rightarrow v, S'} \quad \text{(R-FIELD-SET)}}$$

Expression: By T-FIELD-SET, have both $\mathtt{e} : \mathtt{T}$ and $\Delta(v) : \mathtt{T}$. Since $\mathtt{e}' = v$, $\mathtt{T}' <: \mathtt{T}$ as required.

Store: Define $\Delta' = \Delta$. The only change to S is an update that ith field of object at location l is now pointing at v instead of v_i. Futhermore, v is chosen by T-FIELD-SET to be such that $\Delta(v) <: \Delta(v_i)$. Thus, the store well-formedness is preserved.

R-METHOD

$$\boxed{\frac{S[l] = \mathtt{N}(\overline{v_l}) \qquad \mathit{mbody}(\mathtt{m}<\overline{\mathtt{V}}>, \mathtt{N}) = \overline{\mathtt{x}}.\mathtt{e}_0}{l.\mathtt{m}<\overline{\mathtt{V}}>(\overline{v}), S \rightarrow l \ > \ [\overline{v}/\overline{\mathtt{x}}, l/\mathtt{this}, \mathtt{This}_l/\mathtt{This}]\mathtt{e}_0, S} \quad \text{(R-METHOD)}}$$

Expression [1]**:** By T-METHOD one has $\mathtt{e} = l.\mathtt{m}<\overline{\mathtt{V}}>(\overline{v}) : \mathtt{T}$ where $\mathtt{T} = [\overline{\mathtt{V}}/\overline{\mathtt{Y}}, \mathit{this}_P(l)\ /\mathtt{This}]\mathtt{U}$ and $\mathit{mtype}(\mathtt{m},\ \mathit{bound}_\Delta(\Delta(l)) = <\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}>\overline{\mathtt{U}} \rightarrow \mathtt{U}$ and for some class $\mathtt{C}$ such that $\mathtt{N} <: \mathtt{C}$ method $\mathtt{m}$ is declared in it (by MT-CLASS). By method typing rule (T-METHOD), $\mathtt{e}_0 : \mathtt{U}$ and hence by T-CONTEXT one has $\Delta; l \vdash [\overline{v}/\overline{\mathtt{x}}, l/\mathtt{this}, \mathtt{This}_l/\mathtt{This}]\mathtt{e}_0 : [l/\mathtt{this}, \mathtt{This}_l/\mathtt{This}]\mathtt{U} = \mathtt{T}'$. Finally, since $P = l$, the FGO *this* function expands $\mathit{this}_P(l)$ into $[l/\mathtt{this}, \mathtt{This}_l/\mathtt{This}]$ substitution and $\mathtt{T}' <: \mathtt{T}$ as required.

Store: Define $\Delta' = \Delta$ so that $\Delta \subseteq \Delta'$, to show that $\Delta' \vdash S'$, one needs $\Delta \vdash S$, which already holds.

[1]The notation for $\overline{\mathtt{x}}.\mathtt{e}_0$ is taken from FGJ and lists all the arguments of the method followed by its body.

R-METHOD/FIELD/FIELD-SET-NULL

(R-METHOD/FIELD/FIELD-SET-NULL): $$\frac{}{\texttt{null.m}\,\texttt{<}\,\overline{\texttt{V}}\,\texttt{>}(\overline{v}), S \rightarrow \texttt{error}, S}$$

$$\frac{}{\texttt{null.f}_i, S \rightarrow \texttt{error}, S} \qquad \frac{}{\texttt{null.f}_i = v, S \rightarrow \texttt{error}, S}$$

NB! The same proof applies to all three of these rules.

Expression: By T-ERROR, `error` can be any well formed FGO type and thus will be a well formed FGO subtype of `T` for any `e : T`.

Store: Define $\Delta' = \Delta$ so that $\Delta \subseteq \Delta'$, to show that $\Delta' \vdash S'$, one needs $\Delta \vdash S$, which already holds.

R-CAST

$$\frac{S[l] = \texttt{N}(\overline{v}) \qquad \texttt{N} <: \texttt{P}}{(\texttt{P})l, S \rightarrow l, S} \ \text{(R-CAST)}$$

Expression: By T-CAST $\texttt{e} = (\texttt{P})l : \texttt{P}$ and, hence, $\texttt{P} = \texttt{T}$. By T-LOC $\texttt{e}' = l : \Delta(l)$ and by R-CAST $\Delta(l) = \texttt{N} = \texttt{T}'$. But by R-CAST one has $\texttt{T}' = \texttt{N} <: \texttt{P} = \texttt{T}$ as required. Basically the reduction rule enforces the preservation of the type during casting, otherwise the R-BAD-CAST applies and raises an `error`.

Store: Define $\Delta' = \Delta$ so that $\Delta \subseteq \Delta'$, to show that $\Delta' \vdash S'$, one needs $\Delta \vdash S$, which already holds.

R-BAD-CAST

$$\frac{S[l] = \texttt{N}(\overline{v}) \qquad \texttt{N} \not<: \texttt{P}}{(\texttt{P})l, S \rightarrow \texttt{error}, S} \ \text{(R-BAD-CAST)}$$

Expression: By T-ERROR, $\texttt{T}' = \texttt{error} <: \texttt{P} = \texttt{T}$ as required. Since `error` is made a subtype of all the well-formed types for the type preservation proof to be simple.

Store: Define $\Delta' = \Delta$ so that $\Delta \subseteq \Delta'$, to show that $\Delta' \vdash S'$, one needs $\Delta \vdash S$, which already holds.

R-CONTEXT

$$\frac{}{l \ > \ v, S \rightarrow v, S} \ \text{(R-CONTEXT)}$$

Expression: Given that $\mathtt{e} = l > v : \mathtt{T}$ if and only if $\Delta; l \vdash v : \mathtt{T}$ by T-CONTEXT, one has $\mathtt{e}' = v : \mathtt{T}$ as required.

Store: Define $\Delta' = \Delta$ so that $\Delta \subseteq \Delta'$, to show that $\Delta' \vdash S'$, one needs $\Delta \vdash S$, which already holds.

R-LET

$$\frac{}{\mathtt{let\ x} = v\ \mathtt{in\ e_0}, S \rightarrow [v/\mathtt{x}]\mathtt{e_0}, S} \quad \text{(R-LET)}$$

Expression: By T-LET one has $\mathtt{e} = \mathtt{let\ x} = v\ \mathtt{in\ e_0} : \mathtt{T}$ where $\mathtt{x} : \Delta(l), \Delta; P \vdash \mathtt{e_0} : \mathtt{T}$. Since in $\mathtt{e}' = [v/\mathtt{x}]\mathtt{e_0}$, one has $\mathtt{x} : \Delta(v)$ by T-LET, one also has $\mathtt{e}' : \mathtt{T}$ as required.

Store: Define $\Delta' = \Delta$ so that $\Delta \subseteq \Delta'$, to show that $\Delta' \vdash S'$, one needs $\Delta \vdash S$, which already holds.

□

12.2 Progress Theorem

The progress theorem shows that FGO programs do not get "stuck" and any well typed FGO expression that does not contain free variables (closed) can be reduced to some value or FGO's `error` (the latter includes failed downcasts due to R-BAD-CAST reducing them to `error`).

Theorem 2. ***(Progress)*** *Suppose* `e` *is a closed well-typed FGO expression. Then either* `e` *is a value (or* `error`*) or there is an applicable reduction rule that contains* `e` *on the left hand side.*

Proof. Since `e` is a closed expression it cannot be a variable. If `e` is already `error`, `null` or location l, then there is nothing to prove. In all other cases, either `e` is a redex and can be reduced using the context reduction rule or one of the other reduction rules applies. There are no additional requirements for the context reduction, R-LET, and R-CONTEXT rules and thus they are applicable if `e` matches their left hand side.

In case of expression `e` being a cast, one of R-CAST or R-BAD-CAST has to apply based on whether the two types involved are subtypes or not. In either case, the reduction result is a value.

If expression `e` is of any other type, one needs to make sure that the corresponding reduction rule having it on its left hand side has all of the additional conditions met.

In case of R-FIELD and R-FIELD-SET well-typedness of `N` ensures that $\mathit{fields}(\mathtt{N})$ is well defined and $\mathtt{f}_i$ appears in it. In case of R-METHOD, the fact that *mtype* looks up the type for `m` ensures that *mbody* will succeed too and will have the same number of arguments (since MT-CLASS and MB-CLASS are defined in the same way).

In case of $l = \mathtt{null}$, one of R-FIELD-NULL, R-METHOD-NULL, and R-FIELD-SET-NULL will ensure that the expression reduces to `error`.

Finally, in the case of a $\texttt{new N()}$ expression, it always reduces to a value by R-NEW.

$\square$

13 Ownership Guarantees

I start by presenting a lemma stating that in FGO all the types preserve their ownership information. I use this lemma to prove the confinement invariant that shows that FGO guarantees confinement in a way equivalent to other systems like CFGJ [13]. I then state two definitions of what it means for an object to *refer* to another object and what it means for an object to be *inside* another object. I utilise these definitions to state and prove an ownership invariant that shows that FGO also provides ownership in a way comparable to SafeJava and Ownership Types [3, 5].

Finally, in Section 13.3, I present a different version of an ownership invariant — called a *shallow* ownership invariant [8] and discussed in Chapter 2, Section 2 — that is not as strong as deep ownership invariant but provides for more flexibility [2].

The major difference between these Generic Ownership proofs and earlier non-type-generic ownership invariant proofs lies in a much simpler formulation. The key benefit comes from integrating ownership into a parametric polymorphic type system, rather than building an ownership-parametric type system on top of a non-generic typed language.

Lemma 1. ***(Ownership Invariance)*** *If* $\Delta \vdash \texttt{S} <: \texttt{T}$ *and* $\Delta \vdash \texttt{T} <: \texttt{Object}<\texttt{O}>$, *then* $\mathit{owner}_\Delta(\texttt{S}) = \mathit{owner}_\Delta(\texttt{T}) = \texttt{O}$.

Proof. By induction on the depth of the subtype hierarchy. By FGO class typing rules a FGO class has the same owner parameter as its superclass. $\square$

13.1 Confinement Invariant

This section presents a confinement invariant that is equivalent to the confinement invariant supported by FGC as described in Chapter 2. The difference is that now one is in an imperative language setting, rather than a simpler language like FGJ.

Theorem 3. ***(Confinement Invariant)*** *Let* $\texttt{e}$ *be a subexpression appearing in the body of a method of a well formed FGO class* $\texttt{C}$ *during program execution. If* $\texttt{e} \longrightarrow^* \texttt{new}\ \texttt{D}<\overline{\texttt{T}_\texttt{D}}>(\overline{\texttt{e}})$, *then* $\mathit{visible}_\Delta(\texttt{D}<\overline{\texttt{T}_\texttt{D}}>,\ \texttt{C})$.

Proof. Because the class is a well formed FGO class, its methods are well formed FGO methods. This, plus the standard subformula property [2], implies that, for appropriate $\Delta; P$: both $\Delta; P \vdash \texttt{e} : \texttt{T}$ and $\Delta; \texttt{C} \vdash \mathit{visible}(\texttt{e})$ hold. From this one can derive $\mathit{visible}_\Delta(\texttt{T},\ \texttt{C})$, and hence $\mathit{visible}_\Delta(\mathit{owner}_\Delta(\texttt{T}),\ \texttt{C})$. By FGO's type preservation property, there is a $\texttt{T}'$ such that $\Delta; P \vdash$

[2]Subformula property basically means that all the subexpressions of an expression have to be well-formed since the expression typing rules recurse into every possible subexpression and check its well-formedness.

$\texttt{new D<}\overline{\texttt{T}_\texttt{D}}\texttt{>(}\overline{\texttt{e}}\texttt{)} : \texttt{T}'$, where $\Delta \vdash \texttt{T}' <: \texttt{T}$. Furthermore, one has that $\Delta; P \vdash \texttt{new D<}\overline{\texttt{T}_\texttt{D}}\texttt{>(}\overline{\texttt{e}}\texttt{)} : \texttt{D<}\overline{\texttt{T}_\texttt{D}}\texttt{>}$, and hence clearly $\Delta \vdash \texttt{D<}\overline{\texttt{T}_\texttt{D}}\texttt{>} <: \texttt{T}'$, and $\Delta \vdash \texttt{D<}\overline{\texttt{T}_\texttt{D}}\texttt{>} <: \texttt{T}$. By Lemma 1, $owner_\Delta(\texttt{D<}\overline{\texttt{T}_\texttt{D}}\texttt{>}) = owner_\Delta(\texttt{T})$, from which one deduces $visible_\Delta(owner_\Delta(\texttt{D<}\overline{\texttt{T}_\texttt{D}}\texttt{>}), \texttt{C})$, and therefore $visible_\Delta(\texttt{D<}\overline{\texttt{T}_\texttt{D}}\texttt{>}, \texttt{C})$. □

13.2 Ownership Invariant

This section presents a deep ownership invariant equivalent to the other established ownership type systems [5, 3]. First, lets define what it means for one object to refer to another object in FGO:

Definition 2. ***(Refers To)*** *An object at location l refers to an object at location l' if and only if (1, fields) $\Delta(l) = \texttt{N}(\bar{l})$ and $l' \in \bar{l}$; or (2, locals) for some Δ, P one has $\Delta; P \vdash l > \texttt{e} : \texttt{T}$ and l' occurs as one of the subexpressions of* e.

Second, lets define an *inside* ($\prec$) relationship on owner classes for objects (e.g., $\texttt{This}_l$) in the same manner as the previous ownership type systems [7, 4]. During the execution of any FGO program with deep ownership, if an object l refers to object l', then $\texttt{This}_l \prec owner(\Delta(l'))$. Or more formally:

Definition 3. ***(Inside)*** *Owner class $\texttt{T}$ is inside ($\prec$) owner class $\texttt{T}'$ denoted $\texttt{T} \prec \texttt{T}'$ if and only if $\Delta \vdash \texttt{T} <: \texttt{T}' <: \texttt{World}$.*

At class declaration validation time, the $\prec$ relationship for owner classes is as follows: $\texttt{This} \prec \texttt{Owner} \prec \texttt{World}$ (note that the FGO owner classes' subtyping relationship is along the same lines: $\texttt{This}_l <: \texttt{This} <: \texttt{World}$). During reduction, both Owner and This will have appropriate location-specific owners (e.g., $\texttt{This}_l$) substituted for them. This allows us to prove a deep ownership invariant similar to that of Clarke (and as used by Boyapati).

Theorem 4. ***(Ownership Invariant)*** *l refers to l' only if $\texttt{This}_l \prec owner_\Delta(l')$ or $owner_\Delta(l') = \pi_\texttt{C}$ and $visible_\Delta(\pi_\texttt{C}, \Delta(l))$.*

Proof. For fields by FGO-STORE, $\Delta \vdash \Delta(l') <: [\texttt{This}/\texttt{This}_l]\ \texttt{T}_\texttt{i}$. If owner is World or This, then the theorem holds by the definition of $\prec$. If owner is anything else then since well-formedness preserves owner class nesting and $\texttt{This} <: \texttt{Owner} <: \texttt{O}$ (where O is the set of owners of type parameters) holds, one has $\texttt{This}_l <: \texttt{This}_{l'}$. The second part of the proof holds due to the confinement invariant. □

13.3 Shallow Ownership Invariant

This section assumes that the highlighted bits of WF-TYPE, T-METHOD, FGO-CLASS-MANIFEST and FGO-CLASS-PURE rules are omitted from the type system. Refer to Chapter 2, Section 2 for the discussion of deep vs shallow ownership.

Theorem 5. ***(Shallow Ownership Invariant)*** *l refers to l' only if l'* $\in$ *owners(*$\Delta(l)$*) or* $visible_\Delta$ $(\Delta(l'), \Delta(l))$.

Proof. By FGO type preservation, $\Delta'(v) <: \Delta(l)$ and $\Delta \subseteq \Delta'$. By ownership invariance, $owner_\Delta(\Delta(l)) = owner_\Delta(\Delta'(l)) = owner_\Delta(\Delta'(v))$. By V-OWNER, $visible_\Delta(\Delta'(v), \Delta(l))$. $\square$

14 Summary

In this chapter I presented the FGO type system together with type soundness and ownership invariance proofs. FGO is the first type system to provide support for ownership, confinement, and type polymorphism at the same time. The FGO type system is reasonably compact and easy to formulate. The only restrictions imposed by the FGO type system to provide generic ownership are owner preservation over subtyping, owner nesting, a function to handle `This` owners, and placeholder owner initialisation.

References

[1] Jonathan Aldrich and Craig Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 3086, pages 1–25, Oslo, Norway, June 2004. Springer-Verlag, Berlin, Heidelberg, Germany.

[2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias Annotations for Program Understanding. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 311–330, Seattle, WA, USA, November 2002. ACM Press, New York, NY, USA.

[3] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, EECS, MIT, February 2004.

[4] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership Types for Object Encapsulation. In *Proceedings of ACM Symposium on Principles of Programming Languages (POPL)*, pages 213–223, New Orleans, LA, USA, January 2003. ACM Press, New York, NY, USA. Invited talk by Barbara Liskov.

[5] Dave Clarke. *Object Ownership and Containment*. PhD thesis, School of CSE, UNSW, Australia, 2002.

[6] Dave Clarke and Sophia Drossopoulou. Ownership, Encapsulation, and the Disjointness of Type and Effect. In *Proceedings of ACM Conference on Object-Oriented Programming,*

Systems, Languages, and Applications (OOPSLA), pages 292–310, Seattle, WA, USA, November 2002. ACM Press, New York, NY, USA.

[7] David Clarke, John Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 48–64, Vancouver, Canada, October 1998. ACM Press, New York, NY, USA.

[8] David Clarke and Tobias Wrigstad. External Uniqueness is Unique Enough. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 2473 of *Lecture Notes in Computer Science (LNCS)*, pages 176–200, Darmstadt, Germany, July 2003. Springer-Verlag, Berlin, Heidelberg, Germany.

[9] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, May 2001.

[10] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[11] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Featherweight Generic Confinement. *Journal of Functional Programming*, 16(6):793–811, September 2006.

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

[13] Tian Zhao, Jens Palsberg, and Jan Vitek. Type-Based Confinement. *Journal of Functional Programming*, 16(1):83–128, 2006.