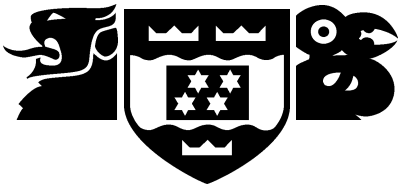
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Mathematical and Computing Sciences
Computer Science

Featherweight Generic Confinement

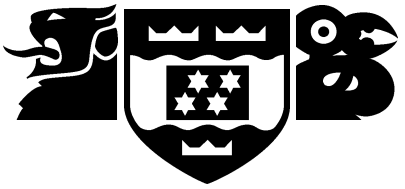
Alex Potanin, James Noble, Dave Clarke[1], Robert Biddle
School of Mathematical and Computing Sciences,
Victoria University of Wellington
New Zealand
{alex, kjx, robert}@mcs.vuw.ac.nz
[1] Institute of Information and Computing Sciences,
Utrecht University
The Netherlands
dave@cs.uu.nl

Technical Report CS-TR-03/17
December 2003

School of Mathematical and Computing Sciences
Victoria University
PO Box 600, Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Mathematical and Computing Sciences
Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341, Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Featherweight Generic Confinement

Alex Potanin, James Noble, Dave Clarke[1], Robert Biddle
School of Mathematical and Computing Sciences,
Victoria University of Wellington
New Zealand
{alex, kjx, robert}@mcs.vuw.ac.nz
[1] Institute of Information and Computing Sciences,
Utrecht University
The Netherlands
dave@cs.uu.nl

Technical Report CS-TR-03/17
December 2003

Abstract

Existing approaches to object encapsulation and confinement either rely on restrictions to programs or require the use of specialised ownership type systems. Syntactic restrictions are difficult to scale and to prove correct, while specialised type systems require extensive changes to programming languages. We demonstrate that confinement can be enforced cheaply in Featherweight Generic Java, with no essential change to the underlying language or type system. This result delineates the differences between parametric polymorphism and ownership type systems, demonstrates that polymorphic type parameters can simultaneously act as ownership parameters, and should facilitate the adoption of ownership and confinement type systems in general-purpose programming languages.

Publishing Information

Accepted for the Foundations of Object-Oriented Languages Workshop (FOOL'11)

Author Information

Alex Potanin is a PhD Student supervised by James Noble and Robert Biddle and Dave Clarke is based in University of Utrecht and completed his PhD under James Noble several years ago

Featherweight Generic Confinement

Alex Potanin, James Noble, Dave Clarke[1], Robert Biddle
School of Mathematical and Computing Sciences
Victoria University of Wellington, New Zealand
{alex, kjx, robert}@mcs.vuw.ac.nz

[1] Institute of Information and Computing Sciences
Utrecht University, Netherlands
dave@cs.uu.nl

ABSTRACT

Existing approaches to object encapsulation and confinement either rely on restrictions to programs or require the use of specialised ownership type systems. Syntactic restrictions are difficult to scale and to prove correct, while specialised type systems require extensive changes to programming languages. We demonstrate that confinement can be enforced cheaply in Featherweight Generic Java, with no essential change to the underlying language or type system. This result delineates the differences between parametric polymorphism and ownership type systems, demonstrates that polymorphic type parameters can simultaneously act as ownership parameters, and should facilitate the adoption of ownership and confinement type systems in general-purpose programming languages.

1. INTRODUCTION

Two main approaches to object instance encapsulation are under investigation in the literature. On one hand, programming conventions, such as Islands [13] and various kinds of Confined Types [4, 10] use tailored restrictions on programs to provide containment guarantees for programs in existing programming languages, but until recently had not been proven sound [22]. On the other hand, ownership type systems [9], originating from the formalisation of Flexible Alias Protection [19], require quite significant modifications to programming languages. In particular, languages like Joe, Universes, AliasJava, and SafeConcurrentJava, depend upon *ownership parameterisation* within the type system [8, 16, 1, 5]. All these type systems are distinct, but they only support ownership parameterisation, not type parameters.

This paper continues the efforts to provide effective object encapsulation within practical programming languages. The key insight behind this paper is that ownership and confinement type systems can readily be modelled within existing parametric polymorphic type systems: in fact, we demonstrate that ownership systems for object confinement within static protection domains can be subsumed completely within a basic generic type system. This is achieved by using a single parameter space to carry *both* generic type and ownership type information. As a result, we can enforce confinement in Featherweight Generic Java [14] "almost for free" — with no change to the underlying language or type system — by additionally enforcing some simple visibility rules and constraints on program structure.

We hope that this result will have several consequences. First, we aim to delineate the differences between parametric polymorphism and ownership and confinement type systems, isolating the small additions to generic type systems that are required to support confinement or ownership. Second, we hope to obtain a simpler formalism, with few new concepts. Third, we are developing an extension to Generic Java that will merge ownership and generic types: our Featherweight Generic Java [14] model will support soundness proofs for our language. Finally, we hope this approach will facilitate the adoption of ownership and confinement type systems by general-purpose programming languages.

The next section of this paper briefly introduces the notion of encapsulation or confinement, in particular the kind of confinement used in Confined Types [4], the primary topic of this paper. We then present FGJ+c, which leverages the generic type rules of FGJ to support a simple confinement invariant, ensuring that confined classes may not be accessed outside a static protection domain (effectively a Java package). We then present the extra constraints required of programs in FGJ+c, prove a confinement invariant, and conclude with a discussion of our prototype implementation and plans for future work.

2. CONFINEMENT

Islands, confinement, and ownership are all essentially forms of object encapsulation [18]. All these schemes are attempts to establish an encapsulation *boundary* that protects some objects *inside* the boundary from direct access by other objects *outside* that boundary. Where these proposals differ from earlier programming language encapsulation and module systems is that they restrict access to objects at runtime: that is, they constrain values of pointers or references to objects in object-oriented systems, rather than merely accesses to field and method names. These schemes enforce a *containment invariant* which simply states that objects outside a particular boundary may not access objects inside that boundary. For example, in Confined Types [4], the unit of confinement is a Java package: all the instances of public classes within that package form the encapsulation boundary; all the instances of private classes (known as *confined classes*) are inside the boundary, and instances of classes

in any other package are outside the boundary. This means that a class may access a public class belonging to any package, but may only access those confined classes belonging to its own package.

What confinement means in practice is that any code written in one protection domain (say one Java package) should, when executed, never *directly* refer to an instance of a class inside the boundary of another protection domain. Static references, such as those stored in object fields, must be restricted: a field of a class cannot hold an object that belongs inside a different package. The execution of a class's methods must also be restricted. Methods cannot access confined classes of other packages. Note, however, that this prohibition refers only to direct accesses: *indirect* access is permitted — indeed, is encouraged. Public classes (or instances of public classes) thus provide an interface to the private instances in their package.

Zhao et. al. [22] have formulated a containment invariant in terms of the expressions within methods. Basically, if an expression (or any of its subexpressions) can possibly evaluate to some object o, that object must be visible in the context of the method. In some more detail: if d_0 is a subexpression of d, and d_0 evaluates to e (denoted, $d_0 \rightarrow^* e$), then any object denoted by e must be visible in the class containing d.

3. FEATHERWEIGHT GENERIC JAVA + CONFINEMENT

Featherweight Generic Confinement is a minimalist confinement scheme that leverages parametrically polymorphic types to enforce static confinement. In this section we present *Featherweight Generic Java + Confinement* (FGJ+c hereafter) which embodies this confinement scheme. After outlining the main principles behind FGJ+c, we give a formal presentation and a proof of confinement in the following section.

3.1 Program Structure

The key idea behind Generic Confinement is to use generic type parameters to carry ownership information as well as type information. Following the traditional approach of Ownership Types [9] we require every FGJ+c class to have at least one type parameter to carry this ownership information. We use the *last* type parameter to record an object's owner, to promote upwards compatibility and because our implementation will allow ownership parameters to be defaulted or elided. To avoid changing the FGJ class `Object`, all FGJ+c classes descend from a new class `CObject` (for confinable object) that has just one parameter called `Owner`; all its subclasses must invariantly preserve this parameter to represent their owner.

The following declaration of a class called `Main` shows that the class is declared with an owner parameter `Owner` that is bound to `CObject`'s owner parameter.

```
class M.Main<Owner extends World>
    extends CObject<Owner> {
  M.Main() { super(); }
  S.Stack<CObject<World>,World> publicStack()
     { return new S.Stack<CObject<World>, World>; }
  S.Stack<M.Main<World>,M>  confinedStack()
     { return new S.Stack<M.Main<World>, M>; } }
```

Note that all class names are prefixed by a package identifier, thus `M.Main`. This is just a convention to indicate the package within which each class is defined. Note also that classes which extend class `World` are used to indicate ownership.

Within the `Main` class, two methods return two `Stack` objects; one of these is confined to the package. Each `Stack` has two type parameters, the first being the type of items to be stored into the stack, and the second being the ownership of that stack instance. The public stack stores `CObject<World>` instances that are accessible from anywhere (`World` is instantiating `CObject`'s owner parameter); the private stack stores `Main` instances that are also globally accessible. The stacks' second ownership parameter describes their ownership. The public stack has owner `World`, so it is universally accessible, however the private stack has owner `M`, meaning that it is only accessible within package `M`. These two stacks illustrate that FGJ+c provides both *type polymorphism* (the stacks hold different item types) and *ownership polymorphism* (the stacks belong in different protection contexts).

3.2 Packages and Owner Classes

FGJ+c types such as `M`, `S`, and `World` represent packages (or protection domains). In FGJ+c protection domains are static, but we need to represent them within the FGJ type system so that we can provide the owner parameters. For this reason, we use parameter-less FGJ classes to represent these domains. Because the classes that represent domains (again like Java packages) are not actually part of the program, they should not be instantiated during the execution of an FGJ+c program so we call them *owner classes*.

Figure 1 shows the relationships between these owner classes and program classes in FGJ+c. Owner classes inherit from the FGJ class `Object`, and there is one owner class corresponding each FGJ+c package.

Confinement in FGJ+c is enforced quite simply, by requiring that any owner class (other than `World`) can only appear within the body of classes within its own package. In other words, the owner class `M` can appear within the definition of classes such as `M.Main` but the owner class `S` cannot. (Note that class names themselves are not restricted *per se*; this is why a name like `S.Stack` can appear in package `M`).

This restriction facilitates the definition of fully confined classes, that is, classes which can never be used outside their defining package. Consider the definition of the `Link` class in package `L`:

```
class L.Link<Item extends CObject<ItemOwner>,
             Owner extends L>
      extends CObject<Owner>
// ...
}
```

`Link`'s owner parameter in its declaration is defined as `Owner extends L`. This means that a `Link` can only be instantiated with the `L` owner class as its actual owner parameter. This owner class is only visible within package `L`, however, thus ensuring all instances of `Link` will be confined within that package.

4. FGJ+c DEFINITION

FGJ+c is a strict subset of FGJ, that is, every FGJ+c program is an FGJ program. FGJ+c, however, adds some

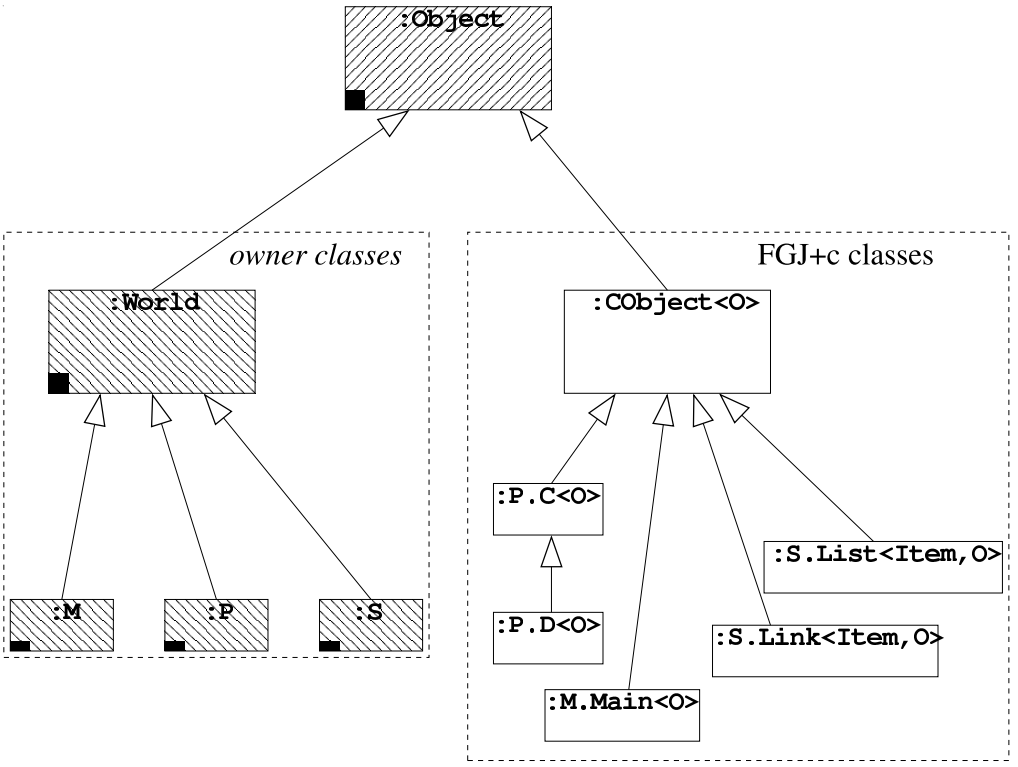


Figure 1: FGJ Classes separated into FGJ+c classes and owner classes.

extra restrictions that leverage FGJ's proven type soundness to provide confinement. Every FGJ+c program must meet the FGJ rules [14] along with additional rules presented in figure 2. For reference, figure 3 shows the FGJ syntax from Igarashi et al. [14].

4.1 FGJ+c Programs

Any FGJ+c program is an FGJ program that meets the following requirement: any class declared as part of it is FGJ+c-Class. A corresponding rule (FGJ+c-Type) ensures that the only types used in FGJ+c programs are subtypes of `COobject<Owner>` for some `Owner`.

Because `Object` doesn't have an owner parameter and cannot be part of FGJ+c programs, we declare `COobject<Owner>` as the root, `World`, and any of the owner classes corresponding to packages separately as valid FGJ classes. For example:

```
class COobject<Owner extends World> extends Object {
    COobject() { super(); }
}

class World extends Object {
    World() { super(); }
}

...
```

This ensures that every FGJ+c program is also an FGJ program. To be able to find a owner class corresponding to the package of a class `C`, we introduce a function π_C returning the owner class of the package to which `C` belongs.

4.2 FGJ+c Additional Rules

Figure 2 gives the following rules used to constrain FGJ programs: **FGJ+c Types** that specifies which types can be instantiated in FGJ+c programs — they are said to be `OK+c`, **Owner Visibility** that show when a particular owner (always a owner class) can be used within the context of the current class's package, **Type Visibility** that gives the types (that have to be `OK+c`) that can be used within the context of the current class's package, **Term Visibility** that shows how expressions don't break the visibility within the context of the current class' package, and finally **FGJ+c Methods** and **FGJ+c Classes** rules that state which method and class declarations are considered legal within FGJ+c— they are said to be `FGJ+c`.

FGJ+c Types. This is used by the rules validating class and method declarations. It states that the only types allowed are the subtypes of `COobject<Owner>`, as shown on the right in figure 1. These types in fact are exactly those that have an owner parameter. Note that the type variables will be classified as `OK+c` by the virtue of their bounds.

Owner Visibility. These rules state that within a context of the domain that is a package of a class `D`, the only owners allowed to be used are `World`, π_D, or those bounded by any of these two. Note that this only allows owner classes as owner parameters.

Type Visibility. Rule V-Type checks to see if the type `T` is legal within FGJ+c programs (by checking if it is `OK+c`, as enforced by **FGJ+c Types** rule, and that its owner is visible according to **Owner Visibility** rules). In case when `T` is a type variable, its *bound*, which is a non-variable type, is checked instead. A type from class `C` is only visible if its

FGJ+c Types:

$$\frac{\Delta \vdash \texttt{T} \texttt{<:} \texttt{CObject<O>} \quad \Delta \vdash \texttt{O} \texttt{<:} \texttt{World}}{\Delta \vdash \texttt{T OK+c}} \quad \text{(FGJ+c-Type)}$$

Owner Visibility:

$$\frac{}{\Delta \vdash \mathit{visibleowner}(\pi_{\texttt{D}},\ \texttt{D})} \qquad \frac{}{\Delta \vdash \mathit{visibleowner}(\texttt{World},\ \texttt{D})} \quad \text{(V-Owner)}$$

$$\frac{\Delta \vdash \mathit{visibleowner}(\mathit{bound}_{\Delta}(\texttt{Owner}),\ \texttt{D})}{\Delta \vdash \mathit{visibleowner}(\texttt{Owner},\ \texttt{D})}$$

Type Visibility:

$$\frac{\Delta \vdash \texttt{T OK+c} \quad \texttt{N} = \mathit{bound}_{\Delta}(\texttt{T}) \quad \texttt{N} = \texttt{C<}\bar{\texttt{T}},\ \texttt{O>} \quad \Delta \vdash \mathit{visibleowner}(\texttt{O},\ \texttt{D})}{\Delta \vdash \mathit{visible}(\texttt{T},\ \texttt{D})} \quad \text{(V-Type)}$$

Term Visibility:

$$\frac{\Delta; \Gamma \vdash \texttt{x} : \texttt{T} \quad \Delta \vdash \mathit{visible}(\texttt{T},\ \texttt{D})}{\Delta; \Gamma \vdash \mathit{visible}(\texttt{x},\ \texttt{D})} \quad \text{(V-Var)}$$

$$\frac{\Delta; \Gamma \vdash \mathit{visible}(\texttt{e},\ \texttt{D}) \quad \Delta; \Gamma \vdash \texttt{e.f}_{\texttt{i}} : \texttt{T} \quad \Delta \vdash \mathit{visible}(\texttt{T},\ \texttt{D})}{\Delta; \Gamma \vdash \mathit{visible}(\texttt{e.f}_{\texttt{i}},\ \texttt{D})} \quad \text{(V-Field)}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash \texttt{e.m}(\bar{\texttt{e}}) : \texttt{T} \quad \Delta \vdash \mathit{visible}(\texttt{N},\ \texttt{D}) \\ \Delta; \Gamma \vdash \mathit{visible}(\texttt{e},\ \texttt{D}) \quad \Delta; \Gamma \vdash \mathit{visible}(\bar{\texttt{e}},\ \texttt{D})\end{array}}{\Delta; \Gamma \vdash \mathit{visible}(\texttt{e.m}(\bar{\texttt{e}}),\ \texttt{D})} \quad \text{(V-Invk)}$$

$$\frac{\Delta; \Gamma \vdash \mathit{visible}(\bar{\texttt{e}},\ \texttt{D}) \quad \Delta \vdash \mathit{visible}(\texttt{N},\ \texttt{T})}{\Delta; \Gamma \vdash \mathit{visible}(\texttt{new N}(\bar{\texttt{e}}),\ \texttt{D})} \quad \text{(V-New)}$$

$$\frac{\Delta; \Gamma \vdash \mathit{visible}(\texttt{e},\ \texttt{D}) \quad \Delta \vdash \mathit{visible}(\texttt{N},\ \texttt{D})}{\Delta; \Gamma \vdash \mathit{visible}((\texttt{N})\ \texttt{e},\ \texttt{D})} \quad \text{(V-Cast)}$$

FGJ+c Methods:

$$\frac{\begin{array}{c}\Delta \vdash \bar{\texttt{T}}, \texttt{T OK+c} \quad \Delta \vdash \mathit{visible}(\bar{\texttt{T}},\ \texttt{C}) \\ \Delta \vdash \mathit{visible}(\texttt{T},\ \texttt{C}) \quad \Delta \vdash \mathit{visible}(\bar{\texttt{P}},\ \texttt{C}) \\ \Delta;\ \bar{\texttt{x}} : \bar{\texttt{T}},\ \texttt{this} : \texttt{C<}\bar{\texttt{X}},\ \texttt{Owner>} \vdash \mathit{visible}(\texttt{e}_0,\ \texttt{C})\end{array}}{\texttt{<}\bar{\texttt{Y}} \lhd \bar{\texttt{P}}\texttt{>}\ \texttt{T m}(\bar{\texttt{T}}\ \bar{\texttt{x}})\{\ \texttt{return e}_0\texttt{;}\ \}\ \texttt{FGJ+c IN C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}},\ \texttt{Owner>}} \quad \text{(FGJ+c-Method)}$$

FGJ+c Classes:

$$\frac{\begin{array}{c}\Delta = \bar{\texttt{X}} \texttt{<:} \bar{\texttt{N}}, \texttt{Owner <: Domain} \quad \Delta \vdash \texttt{N}, \bar{\texttt{T}}\ \texttt{OK+c} \quad \Delta \vdash \mathit{visible}(\bar{\texttt{N}},\ \texttt{C}) \\ \Delta \vdash \mathit{visibleowner}(\texttt{Domain},\ \texttt{C}) \quad \bar{\texttt{M}}\ \texttt{FGJ+c IN C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}},\ \texttt{Owner} \lhd \texttt{Domain>} \\ \Delta \vdash \mathit{visible}(\bar{\texttt{T}},\ \texttt{C}) \quad \texttt{N} = \texttt{C}'\texttt{<}\bar{\texttt{T}}',\ \texttt{Owner>}\end{array}}{\texttt{class C<}\bar{\texttt{X}} \lhd \bar{\texttt{N}},\ \texttt{Owner} \lhd \texttt{Domain>} \lhd \texttt{N}\ \{\bar{\texttt{T}}\ \bar{\texttt{f}}\texttt{;}\ \texttt{K}\ \bar{\texttt{M}}\}\ \texttt{FGJ+c}} \quad \text{(FGJ+c-Class)}$$

Figure 2: FGJ+c Additional Rules

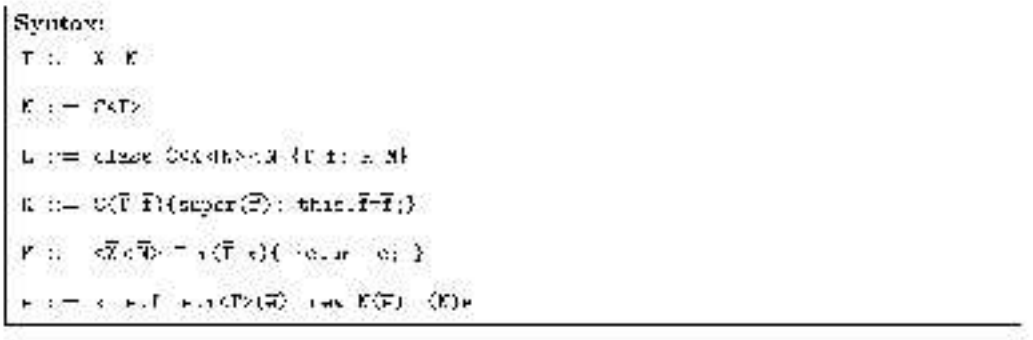


Figure 3: FGJ syntax, from Igarashi et al.

owner parameter is visible within the context of D.

Term Visibility. These structural rules take any expression and recursively check if it is legal within the context of the domain that is a package of a class D. Five rules recurse into five kinds of expression available in FGJ and check if the types involved are *visible* (these can only be types that are OK+c — not owner classes).

FGJ+c Methods. The method is considered to be FGJ+c if and only if all the parameter types and the return type are valid (OK+c) and *visible*, if all the generic method parameters are *visible*, and finally if the expression involved in the definition of the method meets the **Term Visibility** requirements.

FGJ+c Classes. Most importantly, a class is considered to be FGJ+c if all the methods involved in its declaration are FGJ+c, if all the field types are *visible*, if the owner parameter is preserved as we extend another class, and if all the generic parameters involved are *visible* or, in the case when their bounds are owner classes, they are *visibleowner*.

These rules guarantee that FGJ+c programs don't break confinement, allowing us to prove a confinement invariant.

4.3 Confinement Invariant Proof

The confinement invariant for well-typed FGJ+c programs shows that types that are not visible within the current package are not reachable. We assume that we only deal with FGJ+c programs: that is FGJ programs where all the classes are FGJ+c as given by (FGJ+c-Type) rule in figure 2. We prove that during execution, we cannot get to an instance of a class that is not *visible* within the current package. This result relies on the fact that the owner parameter is preserved in the class hierarchy. Let $owner_\Delta(\mathtt{T})$ be O for types $\mathtt{C}\langle\overline{\mathtt{T}},\mathtt{O}\rangle$ and the owner of the bound in Δ for type variables. Then, the following two results that are trivial to prove can be observed:

Lemma (Ownership Invariance over Subtyping). *If* $\Delta \vdash \mathtt{S} <: \mathtt{T}$ *and* $\Delta \vdash \mathtt{T} <: \mathtt{CObject}\langle\mathtt{O}\rangle$, *then* $owner_\Delta(\mathtt{S}) = owner_\Delta(\mathtt{T}) = \mathtt{O}$.

Proof. By induction on subtypes below CObject. S and T must eventually be bound to classes as these are the only ground types in FGJ. Base case: they are bound to CObject so this is trivially true. Inductive case: By FGJ subtyping they must be bound to some transitive subclass of CObject. By FGJ+c-Class a FGJ+c class has the same owner parameter as its superclass. □

Theorem (Confinement Invariant). *Given any subexpression* e *of an expression* d *in a method* m *of class* C *which is* FGJ+c.
If $\mathtt{e} \rightarrow^* \mathtt{new}\ \mathtt{D}\langle\overline{\mathtt{T}_\mathtt{D}}, \mathtt{O}\rangle(\overline{\mathtt{e}})$, *then* $visible(\mathtt{D}\langle\overline{\mathtt{T}_\mathtt{D}}, \mathtt{O}\rangle, \mathtt{C})$.

Proof. By FGJ subject reduction: $\mathtt{e} : \mathtt{T} \rightarrow^* \mathtt{e}' : \mathtt{T}' \Rightarrow \mathtt{T}' <: \mathtt{T}$. Since $visible(\mathtt{d}, \mathtt{C})$ by FGJ+c-Method, by visibility rules we also get $visible(\mathtt{e}, \mathtt{C})$. By rule V-Type T is OK+c and thus by FGJ+c-Type $\mathtt{T} = \mathtt{B}\langle\overline{\mathtt{T}_\mathtt{B}}, \mathtt{O}\rangle$ where $\mathtt{B} <: \mathtt{CObject}\langle\mathtt{O}\rangle$. By the ownership invariance over subtyping lemma: $\mathtt{T}' = \mathtt{D}\langle\overline{\mathtt{T}_\mathtt{D}}, \mathtt{O}\rangle$ where O is the same in all three. But to get $visible(\mathtt{B}\langle\overline{\mathtt{T}_\mathtt{B}}, \mathtt{O}\rangle, \mathtt{C})$ we must have $visibleowner(\mathtt{O}, \mathtt{C})$. By FGJ+c-Type, $\mathtt{D}\langle\overline{\mathtt{T}_\mathtt{D}}, \mathtt{O}\rangle$ will be OK+c.
Then by V-Type: $visible(\mathtt{D}\langle\overline{\mathtt{T}_\mathtt{D}}, \mathtt{O}\rangle, \mathtt{C})$. □

4.4 FGJ+c and FGJ

Every FGJ+c program is also an FGJ program. We expect that every FGJ program can be translated into an FGJ+c program as follows: (1) every class gets an extra parameter, and every type instantiates it as World; (2) every extension of Object is now replaced with an extension of CObject<O>; and (3) in every class declaration, the owner parameter of the class and the type it extends is matched. This will ensure every FGJ type is now OK+c and every class FGJ+c-Class.

5. RELATED WORK

Object encapsulation has been recognised as a means for addressing aliasing, security, concurrency, and memory management problems, with the merit of smoothly aligning with the way many object-oriented programs are designed. Two complementary threads of research have evolved. On one hand are expressive but weighty type systems based on ownership types [9]. On the other hand are lightweight but limited systems based on confined types [4].

The systems based on ownership types differ essentially in only one characteristic, which Clarke and Wrigstad distinguish as *shallow* vs. *deep* ownership [11]. A deep ownership type permits only a single object as entry point to the collection of objects it owns, whereas a shallow ownership type permits multiple entry points into the confined collection.

Clarke and Drossopoulou [8] and Boyapati et al. [5] describe how to exploit the useful properties of deep ownership, but there is also a general concern about whether it might be too restrictive in practice. Ownership types require ad-

ditional annotations to use them, raising issues about their role in programming. Some authors argue that, with appropriate defaults, this need not be a problem in practice [1, 5].

Confined type systems have achieved their more limited goals while keeping the amount of annotations low. Vitek and Bokowski's original system [4], which had security as its application, required certain classes to be annotated as confined to indicate classes confined within the present package, and certain methods to be annotated as anonymous, to indicate that such methods do not reveal "`this`". Grothoff, Palsberg, and Vitek [12] show how type inference can be used to avoid the need for annotation, making a system that can provide per-package encapsulation in practical programs. Clarke, Richmond and Noble [10] apply these ideas in the context of Enterprise Java Beans, and by exploiting special architecture specific constraints, provide per-object encapsulation without annotations or inference.

Recent work by Zhao, Palsberg and Vitek [22] has formalised Vitek and Bokowski's approach to per-package confinement, with an operational semantics and a static type system based on Featherweight Java, and augmented by a number of specific rules that restrict programs. This work also proposes a notion of generic confined types, allowing, for example, a collection to be confined or not, depending upon the specifications of the contained elements. This proposal is then supported by further development of the static type system.

Our approach is essentially the opposite. Rather than starting from a language without generic types, and then adding a special form of genericity to better support confinement, we start from a language with generic types (GJ, or rather its formal core FGJ) and then ensure per-package confinement. This approach has led to a simpler formal system, requiring few new concepts. We do not need to distinguish anonymous methods, because "`this`" is parameterised to record its ownership.

Banerjee and Naumann prove a per-object representation independence result for Java [2, 3]. They adopt a confinement discipline resembling ownership types, except that they apply the confinement only at the point they wish to reason about. They require that confined classes extend a special class called Rep, and that the boundary classes extend a special class called Own. Neither Rep nor Own can be forgotten from a type.

A bit further afield, we find that the implementation of the State Monad in Haskell [15] adopts similar mechanisms. In the State Monad, a type variable is assigned to the encapsulated state, and an appropriate hiding of the type (via rank-2 polymorphism) ensures that the state doesn't escape and thus behaves correctly. Interestingly, this design resembles an encoding of existential types in terms of universal types, while Clarke's thesis formalises the confinement provided by ownership types as existential over owners [7].

6. IMPLEMENTATION AND FUTURE WORK

6.1 Object Ownership

Object ownership is essentially the same as confinement, but works at a finer granularity: ownership allows objects to be encapsulated with a *dynamic* protection domain, typically another object, where confinement is limited to *static* protection domains, such as as Java-like packages.

We plan to extend FGJ+c to provide object ownership as well as per-package confinement. We will introduce an additional owner class called "`This`" to model objects owned by the current object (i.e., owned by "`this`"). Then, we need to ensure that we can only access objects owned by `This` from the instance to which they belong — that is, only when derefencing `this`. Following [1], we sketch such an ownership rule:

$$\frac{\Delta;\Gamma \vdash \mathtt{e} : \mathtt{T} \qquad \mathtt{This} \in \mathit{owners}(\mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{T}))) \Rightarrow \mathtt{e} \equiv \mathtt{this}}{\Delta;\Gamma \vdash \mathit{ownership}(\mathtt{e.m}(\bar{\mathtt{e}}),\ \mathtt{D})}$$

Where *owners* finds the set of all ownership parameters or bounds in a type. To provide deep ownership, we will also need to introduce an ordering on owner classes to ensure the proper object containment relationships.

We can state this rule within FGJ, but FGJ's functional substrate is not strong enough to support a proof of an ownership invariant. For this, we plan to adopt an imperative Java-like calculus, such as that underpinning $\mathtt{Joe}_1$ [8], although that calculus will first have to be extended with FGJ-like genericity.

6.2 Capabilities

We also plan to extend FGJ+c to model capability-like systems [6, 20] where types control invocations of individual methods, rather than access to whole objects. Again, we have to alter the method invocation rule:

$$\frac{\Delta;\Gamma \vdash \mathit{visible}(\mathtt{e},\ \mathtt{D}) \qquad \Delta;\Gamma \vdash \mathit{visible}(\bar{\mathtt{e}},\ \mathtt{D}) \qquad \Delta;\Gamma \vdash \mathtt{e} : \mathtt{N}' \qquad \Delta;\Gamma \vdash \mathit{visiblemethod}(\mathtt{m},\ \mathtt{T},\ \mathtt{D})}{\Delta;\Gamma \vdash \mathit{visible}(\mathtt{e.m}(\bar{\mathtt{e}}),\ \mathtt{D})}$$

including a ternary *visiblemethod* check that tests whether a particular method `m` may be invoked on an object bounded by type `T` in domain `D`. This will allow capabilities to be encoded via specialised owner classes in `T`'s ownership: note that this check would need to be monotonic over subtyping.

6.3 OGJ: Ownership Generic Java

We have implemented an extension to the JSR14 prototype implementation of the Java Compiler [21] that we call OGJ (for "Oh! Gee! Java!" [17]). OGJ is the first language implementation that supports both confinement and genericity. OGJ programs are essentially Generic Java programs with the addition of owner parameters that can be: `World`, `Class`, `Package`, or `This`. These are real Java interfaces defined in a package `ogj.ownership`. Any OGJ program will compile as long as the (blank) definitions of these interfaces are present in the class path, making OGJ backwards compatible with GJ compilers.

On the other hand, if the program is compiled using our extension to the JSR14 prototype, these four interfaces are treated specially so that any class parameterised by `World` and `Package` behaves in a similar way to FGJ+c classes parameterised by `World` and the package owner classes. Thus, a class defined as follows:

```
import ogj.ownership.*;
```

```
package my.util;

public class Link<Item, Package> {
   ...
}
```

will be guaranteed to have all of its instances contained within the `my.util` package as long as the code is compiled using our OGJ compiler extension. While in FGJ+c we used a separate owner class for each owner parameter corresponding to a package, OGJ will make appropriate replacements of every occurence of parameter `Package` with an appropriate owner class. This reduces a programmer load and hides the owner classes from view.

Furthermore, we have also implemented full support for `Class` visibility (per-class confinement similar to Class Universes [16]: objects can only be used by the *class* within which they are declared) and `This` visibility (per-object ownership). In this paper, we have formalised the part of OGJ that supports confinement. We plan to finish the formalisation of the ownership support in the near future, as described above.

OGJ compiler extension is the first implementation that has support for both confinement and genericity.

7. CONCLUSION

In this paper we have demonstrated that generic type systems are capable of expressing class confinement. In particular, we have demonstrated that the FGJ type system, combined with a series of visibilty rules, is strong enough to provide a confinement invariant comparable to that of Confined Types.

This result shows that ownership and generic type information can be expressed within the same system, and carried around the program as binding to the same parameters. We have proved this is possible for static package and class confinement, and hope to extend this to more discriminating systems such as ownership types. This may provide a lightweight route for ownership types to become applicable in practice, with genericity carrying ownership into popular object-oriented programming languages.

Acknowledgments

This work is supported in part by the Royal Society of New Zealand Marsden Fund. The second author would like to thank the staff of Ward 18, Wellington Hospital.

8. REFERENCES

[1] Aldrich, J., Kostadinov, V., and Chambers, C. Alias annotations for program understanding. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)* (Nov. 2002).

[2] Banerjee, A., and Naumann, D. A. Ownership confinement ensures representation independence for object-oriented programs. Journal version of POPL 2002 paper, submitted., 2002.

[3] Banerjee, A., and Naumann, D. A. Representation independence, confinement and access control. In *POPL* (2002), pp. 166–177.

[4] Bokowski, B., and Vitek, J. Confined types. In *Proceedings of Conference on Object-Oriented Programming, Languages, and Applications* (1999), ACM Press.

[5] Boyapati, C., Liskov, B., and Shrira, L. Ownership types for object encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL)* (Jan. 2003).

[6] Boyland, J., Noble, J., and Retert, W. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In *European Conference on Object-Oriented Programming (ECOOP)* (June 2001), Springer-Verlag.

[7] Clarke, D. *Object ownership and containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Australia, 2002.

[8] Clarke, D., and Drossopoulou, S. Ownership, encapsulation, and the disjointness of type and effect. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)* (2002).

[9] Clarke, D., Potter, J., and Noble, J. Ownership types for flexible alias protection. In *Proceedings of Conference on Object-Oriented Programming, Languages, and Applications* (1998), ACM Press.

[10] Clarke, D., Richmond, M., and Noble, J. Saving the world from bad beans: Deployment-time confinement checking. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)* (October 2003).

[11] Clarke, D., and Wrigstad, T. External uniqueness is unique enough. In *European Conference on Object-Oriented Programming (ECOOP)* (Darmstadt, Germany, July 2003), L. Cardelli, Ed., vol. 2473 of *Lecture Notes In Computer Science*, Springer-Verlag, pp. 176–200.

[12] Grothoff, C., Palsberg, J., and Vitek, J. Encapsulating objects with confined types. In *Proceedings of Conference on Object-Oriented Programming, Languages, and Applications* (2001), ACM Press.

[13] Hogg, J. Islands: Aliasing protection in object-oriented languages. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)* (New York, Nov. 1991), vol. 26, ACM Press, pp. 271–285.

[14] Igarashi, A., Pierce, B., and Wadler, P. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)* (N. Y., 1999), L. Meissner, Ed., vol. 34(10), pp. 132–146.

[15] Launchbury, J., and Peyton Jones, S. L. State in Haskell. *Lisp and Symbolic Computation 8*, 4 (dec 1995), 293–341.

[16] Müller, P., and Poetzsch-Heffter, A. *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999, ch. Universes: a type system for controlling representation exposure.

[17] Noble, J., and Biddle, R. Oh! Gee! Java! — ownership types (almost) for free. Tech. Rep.

VUW-CS-TR-03/9, Computer Science, Victoria University of Wellington, New Zealand, May 2003.

[18] Noble, J., Biddle, R., Tempero, E., Potanin, A., and Clarke, D. Towards a model of encapsulation, 2003. Presented at the ECOOP 2003 IWACO Workshop on Aliasing, Confinement, and Ownership. Available at: http://www.mcs.vuw.ac.nz/comp/Publications.

[19] Noble, J., Vitek, J., and Potter, J. Flexible alias protection. In *European Conference on Object-Oriented Programming (ECOOP)* (1998), Lecture Notes in Computer Science, Springer-Verlag.

[20] Skalka, C., and Smith, S. Static use-based object confinement. *Springer International Journal of Information Security* (2003).

[21] Sun Microsystems. Java development kit. http://java.sun.com/j2se/, 2002.

[22] Zhao, T., Palsberg, J., and Vitek, J. Lightweight confinement for featherweight java. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)* (October 2003).

APPENDIX

Featherweight Generic Java

For the convenience of our readers, and by the grace of `pdflatex`, in this appendix we present additional figures describing FGJ from Igarashi et al. [14].

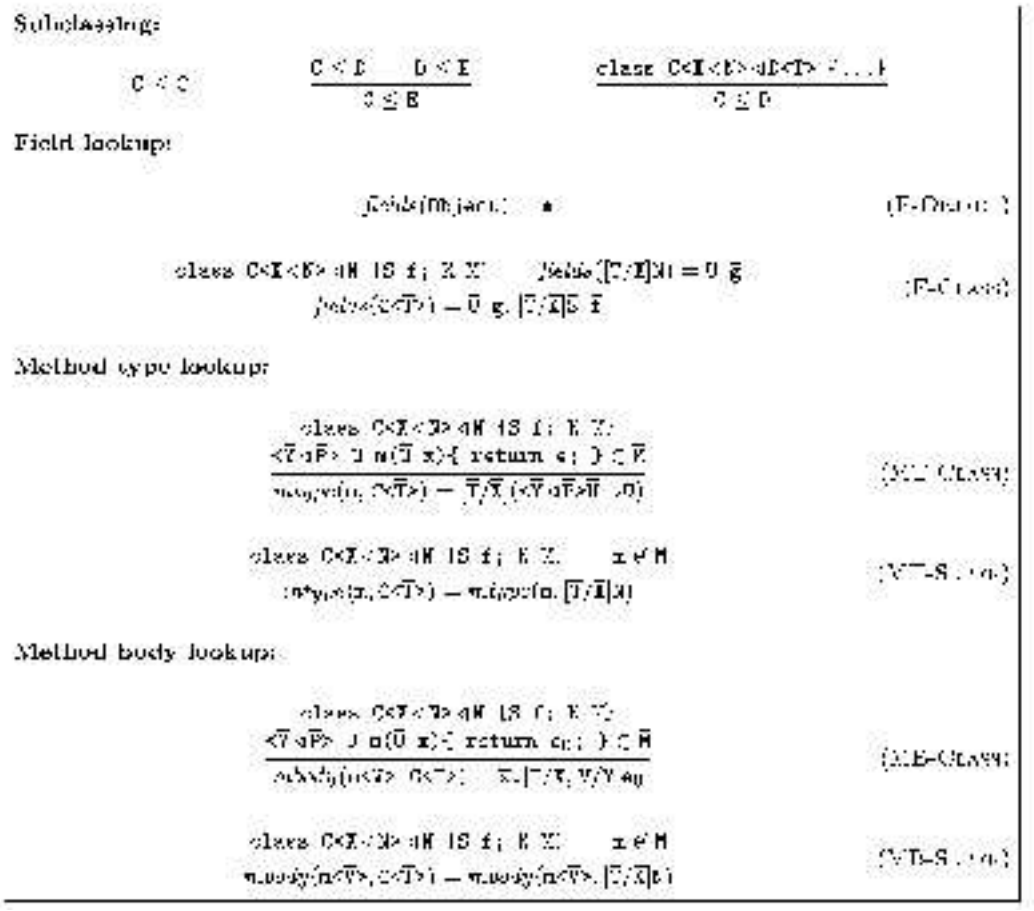


Figure 4: FGJ auxiliary functions, from Igarashi et al.

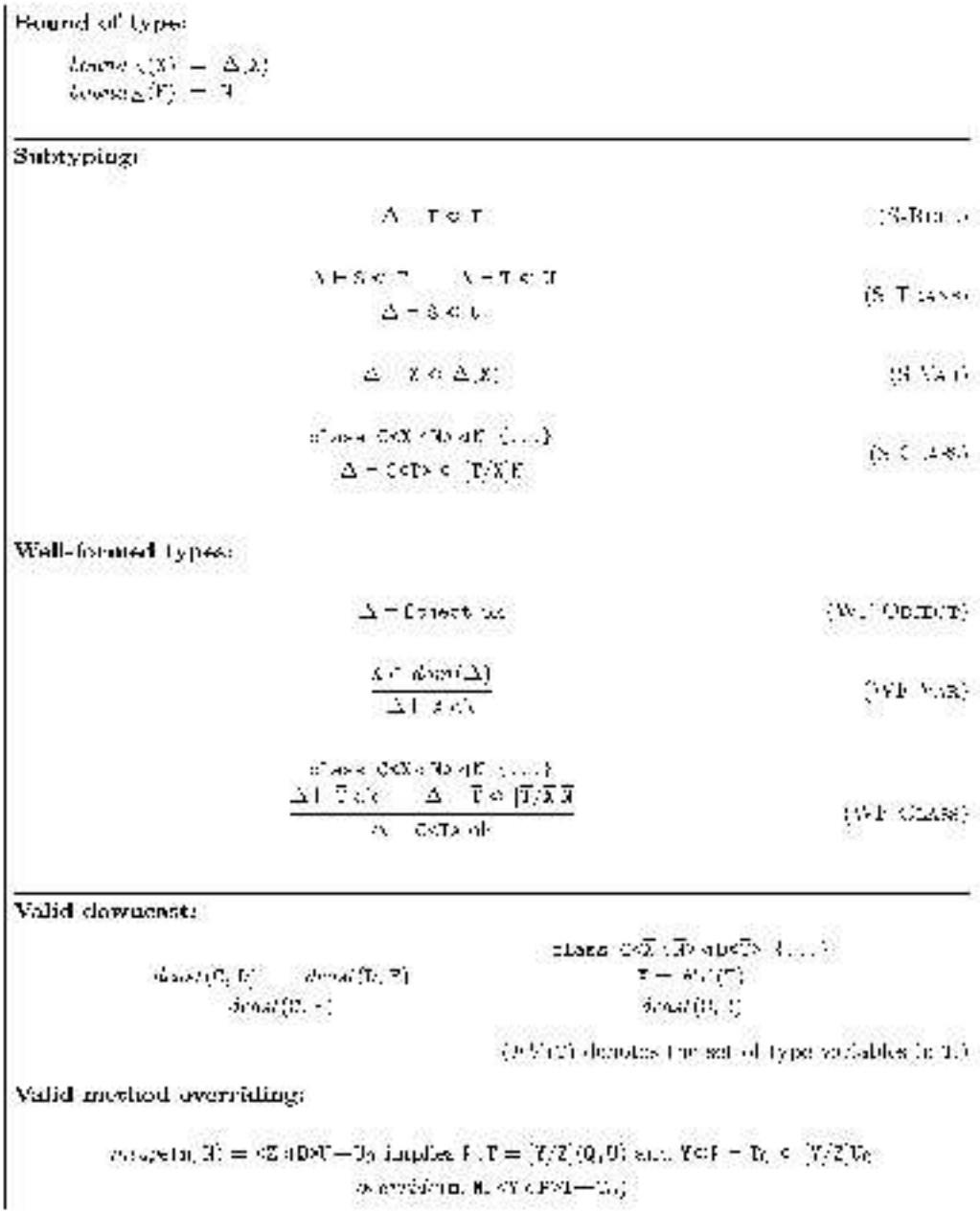


Figure 5: FGJ subtyping and type well-formedness rules, from Igarashi et al.

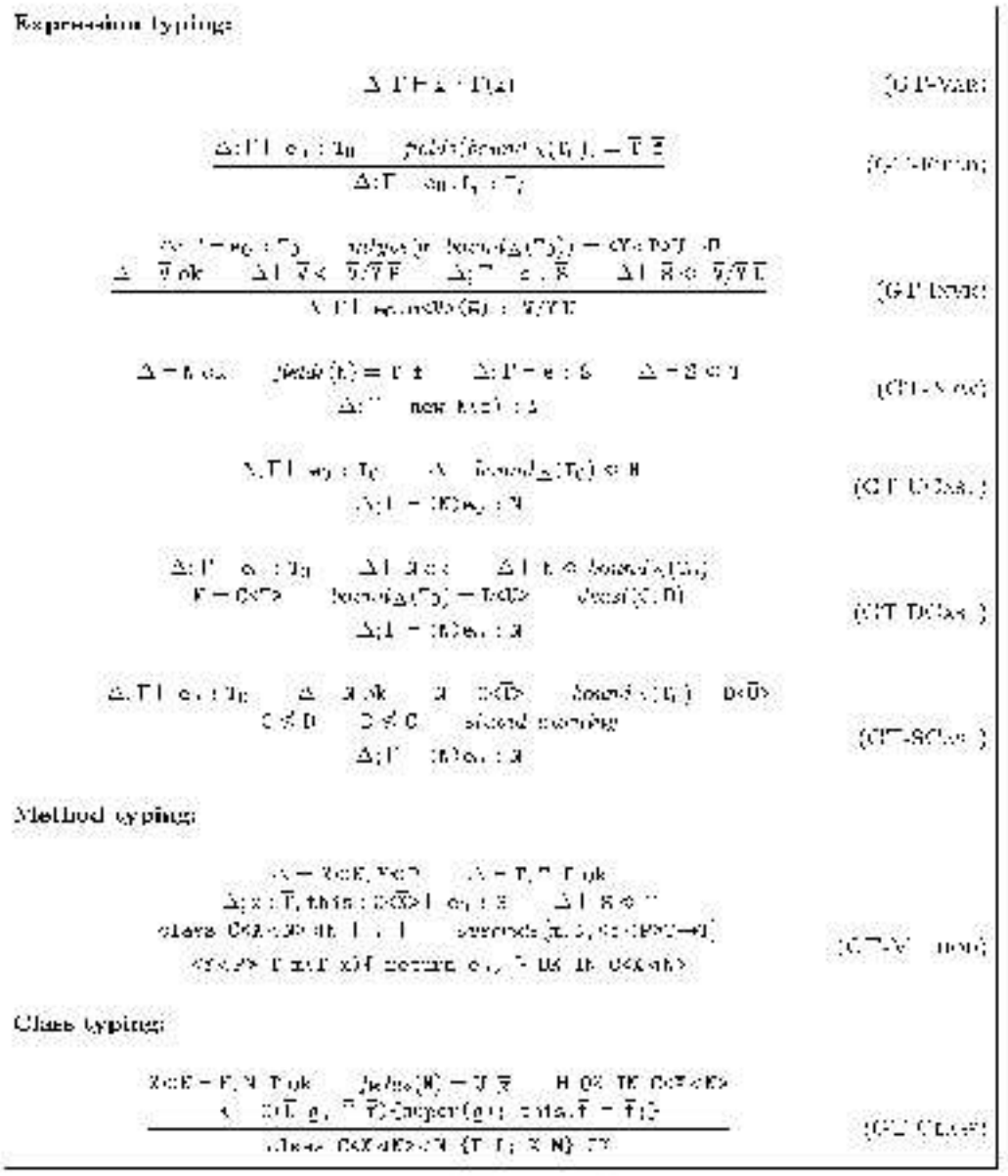


Figure 6: FGJ typing rules, from Igarashi et al.