Defaulting Generic Java to Ownership

Alex Potanin, James Noble, Dave Clarke[1], and Robert Biddle[2]
{alex, kjx}@mcs.vuw.ac.nz, dave@cwi.nl, and robert_biddle@carleton.ca

School of Mathematical and Computing Sciences,
Victoria University of Wellington, New Zealand
[1] Centrum voor Wiskunde en Informatice, Amsterdam, Netherlands
[2] Human-Oriented Technology Laboratory, Carleton University, Canada

Abstract. Generic ownership is a mechanism for seamlessly combining ownership and genericity. Every class in Ownership Generic Java (OGJ) has an owner parameter as the last type argument. Defaulting is a way to take any Generic Java (GJ) program and translate it to an OGJ program thus making it possible to provide full backwards compatibility of OGJ with GJ programs. Since standard Java now has support for generics, defaulting lays down a path for making vanilla Java programs ownership parametric with no change to the source code.

1 Introduction

Modern programming languages provide little support for object encapsulation and ownership. Escaped aliases to private objects can compromise both security and reliability of code in reference-abundant languages such as Java. Object ownership is a widely accepted approach to controlling aliasing in programming languages. Generic Ownership is a unified approach to providing generics and ownership [10, 11]. Generic Ownership imposes no more syntactic or runtime overheads than traditional generic types. We have extended Java 1.5 (with generics) to support Generic Ownership. We call the resulting language: Ownership Generic Java (OGJ). We ground the formal side of this work within the Featherweight Generic Java framework.

Why would we want ownership and generic types? Consider for example a *box* as a kind of object. In any object-oriented language we are allowed to say: “this is a box” (meaning any box of any things). In a language with generics, we are allowed to say: “this is a box of books” (denoting a box of books, but not containing birds). In a language with ownership parameterisation, we are allowed to say: “this is my box” or “these are library books”. Combining ownership and generics naturally allows us to say: “this is my box of library books”, not birds and not my personal books. Ownership turns out to work exceptionally well with genericity, both in theory, practice, and implementation.

The problem with OGJ is that to guarantee object containment, it requires annotation of every class declaration with an owner parameter and every class type needs an appropriate instantiation of that owner parameter. This means that programmers must do more work to write OGJ programs than GJ programs and plain Java programs that don’t use these parameters will not compile with OGJ compiler. Defaulting solves this problem — allowing programs to leave out explicit ownership parameters, and supply default values implicitly.

In Java 1.5 a programmer is allowed to say:

```
List listOfObjects = new List();
```

As it has been the case for years, this list can store `Object`s, but we are also allowed to say:

```
List<String> listOfStrings = new List<String>();
```

The compiler will ensure that only `String`s can be put in and taken out of this list. Both examples above will work with the same, generic, implementation of `List` since Java uses *raw types* [9] to provide full backwards compatibility with the original Java code.

OGJ allows a programmer to declare:

```
package myPackage;

class Foo {
  List<String, Package> listOfStringsConfinedToMyPackage =
    new List<String, Package>();
  ...
}
```

This makes it possible for the compiler to check not only that `String`s are the only elements in our list, but also that this instance of list is *confined* to package `myPackage` [2]. The contribution of *defaulting* is that the code above can be written and will work without a single change to the implementation of the `List` class provided with Java 1.5. Furthermore, although class `Foo` above is not parameterised by any ownership parameters, *defaulting* will still allow instances of `Foo` to be parameterised to be confined to either packages, classes, or particular instances, using OGJ.

Outline. Section 2 briefly surveys the Ownership Generic Java and its formalisation that allows the proof of the confinement invariant[1] in FGJ+c (an extension to FGJ [8]). Section 3 describes the defaulting mechanism itself. Section 4 discusses the implications of our mechanism, and section 5 concludes our paper.

2 Background

Generic Ownership [11] is a new linguistic mechanism that successfully combines genericity and ownership into a single simple language. Generic Ownership provides the benefits of both type and ownership parameterisation: catching all the errors and avoiding all the bugs that the generic and ownership languages do individually. Generic Ownership treats ownership and genericity as one single aspect of language design, and so code using Generic Ownership is no more complex than code that is either type-parametric or ownership-parametric.

The key technical contribution of Generic Ownership is that it treats ownership as an additional kind of generic type information. This means that existing generic type systems can be extended to carry ownership information with only minimal changes. This paper presents a novel defaulting mechanism which permits ownership information to be omitted in common circumstances, so that Generic Ownership programs are almost indistinguishable from their ownership-free counterparts, while gaining the benefits of ownership.

In this section, we overview Generic Ownership in terms of a language that we have implemented called Ownership Generic Java (OGJ), a seamless extension to Java 1.5 with support for generic types [12]. We also outline the formal basis of our work.

2.1 Ownership Generic Java (OGJ)

Figure 1 presents a `SimpleMap` class written in Ownership Generic Java. Note that the code in figure 1 is type-generic: definitions of fields in `MapNode` and methods everywhere use generic types such as `Key` and `Value` rather than plain class types such as `Object` or `Comparable`. But this code also supports ownership: each class has an extra generic parameter called `Owner` which represents its owner. As with other ownership type systems, it is not possible for owned objects to be exposed.

As you can see from the figure, OGJ is a language that is completely syntactically compatible with GJ — any valid OGJ program is also a valid GJ program. OGJ provides ownership by requiring every single class declaration to have at least one type parameter that is declared as a subclass of `World`. This parameter should always come *last* and we call it the *owner parameter*. For example, instead of declaring:

```
  public class Foo { ... }
```

OGJ requires the following declaration:

[1] To prove the containment invariant for ownership, a more complete formalism than FGJ is required (for example to support object identity). This is the topic of our current work.

```
public class SimpleMap<Key extends Comparable, Value, Owner extends World> {
  private Vector<MapNode<Key, Value, This>, This> mapNodes;

  public void put(Key key, Value value) {
    mapNodes.add(new MapNode<Key, Value, This>(key, value));
  }

  public Value get(Key key) {
    Iterator<MapNode<Key, Value, This>, This> i = mapNodes.iterator();
    while (i.hasNext()) {
      MapNode<Key, Value, This> mn = i.next();
        if (mn.key.equals(key)) {
          return mn.value;
        }
      }
    return null;
  }
}
class MapNode<Key extends Comparable, Value, Owner extends World> {
  public Key key;
  public Value value;
  MapNode(Key key, Value value) {
    this.key = key; this.value = value;
  }
}
```

Fig. 1. Ownership Generic Java implementation of a `SimpleMap` class

```
public class Foo<Owner extends World> { ... }
```

Owner parameters are used to record the ownership of individual objects of a particular class. OGJ provides four *ownership domains* that programmers can use to instantiate owner parameters: `World`, `Package`, `Class`, and `This`. These domains represent different ownership scopes to which objects can belong, and were chosen to parallel the existing static access structures in Java. Any instance of a class whose owner parameter is instantiated with `World` can be referred to by any other object. References to an object instantiated with a `Package` owner are confined within the package where the instance's type is instantiated, and an instance marked with `Class` is confined within that class. Finally, an object marked with `This` is encapsulated within the object where the type containing `This` is instantiated — that is, it is owned directly by that object. In this way, OGJ provides both confined types (`Package` and `Class`) and shallow ownership (`This`).

For example, if a `Foo` instance is created with its ownership parameter bound to `Package`, as in:

```
public Foo<Package> f = new Foo<Package>();
```

then the new instance of `Foo` can only be accessed via the package containing the instantiation. Similarly, if it is created using `This` ownership:

```
public Foo<This> f = new Foo<This>();
```

then it can only be accessed via the current instance of the class creating the new `Foo` object.

2.2 Featherweight Generic Java with Confinement (FGJ+c)

Our initial formalisation of OGJ is called FGJ+c [10]. FGJ+c embodies a minimalist confinement scheme which leverages parametrically polymorphic types to enforce static confinement, that is, for a statically known collection of protection domains. Our formalism is based on an extension to Featherweight Generic Java (FGJ), which, with Featherweight Java (FJ), provides simple models of Java and Generic Java's type systems [8]. The current implementation of OGJ also implements *shallow ownership*, though we have not treated this in this formalism.

Generic Ownership uses generic type parameters to carry ownership information as well as type information. Following the traditional approach of Ownership Types [6] 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. 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. We are careful to ensure that FGJ+c programs remain FGJ programs. The consequence is that we can leverage FGJ's type soundness result, and use it when establishing the desired confinement property. The syntax of FGJ from Igarashi et al. [8], adapted only slightly for presentation reasons, is given in figure 2. Note that we do not account for statics, which are not present in FGJ. We have thus omitted manifest ownership [5] from our formalism.

$$\begin{array}{lll}
\texttt{T} & ::= & \texttt{X} \mid \texttt{N}\\
\texttt{N} & ::= & \texttt{C<}\bar{\texttt{T}},\texttt{O>}\\
\texttt{L} & ::= & \texttt{class C<}\bar{\texttt{X}} \triangleleft \bar{\texttt{N}},\ \texttt{Owner} \triangleleft \texttt{Domain>} \triangleleft \texttt{N}\ \{\bar{\texttt{T}}\ \bar{\texttt{f}};\ \texttt{K}\ \bar{\texttt{M}}\}\\
\texttt{K} & ::= & \texttt{C(}\bar{\texttt{T}}\ \bar{\texttt{f}}\texttt{)}\ \{\ \texttt{super(}\bar{\texttt{f}}\texttt{); this.}\bar{\texttt{f}}\texttt{=}\bar{\texttt{f}}\texttt{;}\ \}\\
\texttt{M} & ::= & \texttt{<}\bar{\texttt{X}} \triangleleft \bar{\texttt{N}}\texttt{> T m(}\bar{\texttt{T}}\ \bar{\texttt{x}}\texttt{)}\ \{\ \texttt{return e;}\ \}\\
\texttt{e} & ::= & \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m<}\bar{\texttt{T}}\texttt{>(}\bar{\texttt{e}}\texttt{)} \mid \texttt{new N(}\bar{\texttt{e}}\texttt{)} \mid \texttt{(}\bar{\texttt{N}}\texttt{)e}
\end{array}$$

For presentation reasons, we reserve `O` to refer to owner-classes, `Domain` for concrete owner-classes, `Owner` for owner variables. (`O ::= Domain | Owner`.)

Fig. 2. FGJ syntax, adapted from Igarashi et al

Every FGJ+c program must satisfy the FGJ type rules [8], along with some additional constraints described in detail elsewhere [10, 11]. The rules deal with three concerns: (a) every FGJ+c type has an owner; (b) the owners are preserved over inheritance/subtyping; and (c) the owners are visible within the context in which they occur. These rules are recursively propagated through expressions, method and class declarations to ensure that they hold for a whole FGJ+c program.

3 Defaulting

To further reduce the syntactic overhead of Generic Ownership, owner parameters in OGJ can be elided from class declarations and instantiations — we call this *ownership defaulting*. Essentially, if a class declaration does not declare an owner parameter, the OGJ language will provide a default `Owner` extending `World`. We choose `World` so that ordinary Java and GJ code can be used unchanged in OGJ. For example, the following class declaration:

```
public class Athlete<Event> { ... }
```

when compiled using OGJ will be treated as:

```
public class Athlete<Event, Owner extends World> { ... }
```

Similarly, if a class has an owner parameter in its declaration (possibly added by defaulting), this parameter may be omitted and OGJ will instantiate the parameter to its bound by default. For example, code using `Athlete`, such as:

```
Athlete<Discus> a = new Athlete<Discus>();
```

will be taken to mean:

```
Athlete<Discus<World>, World> a =
  new Athlete<Discus<World>, World>();
```

The main effect of defaulting is that programmers are able to write Generic Ownership code with very little syntactic overhead, providing owner parameters only when absolutely required. Because the rules for defaulting are quite straightforward, they have no effect on the modularity of OGJ code — as with defaults in other generic type systems, but in contrast to more complex type inference schemes used to support other ownership types systems [1, 7].

We have implemented this defaulting scheme within OGJ compiler (an extension of Java 1.5 compiler [12]) and can successfully check Java's *collect.jar*, and compile the compiler source itself. The rest of this section describes the formal defaulting mechanism.

3.1 Defaulting for FGJ+c

Formally, we aim to model defaulting so that we can take any FGJ program (that doesn't have owner parameters) and produce a valid FGJ+c program (that has the appropriate owner parameters). When defaulting an FGJ program we replace every class declaration with a defaulted version, as well as replacing every nonvariable type occurrence in all expressions, method and class declarations with defaulted versions.

Defaulting `Object` (D-FGJ+C-OBJECT):

$$[\![\texttt{Object}]\!] = \texttt{CObject<World>} \qquad \text{O1}$$

Defaulting Types (D-FGJ+C-TYPE):

$$\frac{\begin{array}{c}\Delta_{\texttt{FGJ}} \vdash \texttt{T} = \texttt{C<}\overline{\texttt{N}}\texttt{>} \qquad \texttt{D}' = \mathit{bound}(\texttt{D}) \\ \texttt{class C<}\overline{\texttt{N}}\texttt{, O} \triangleleft \texttt{D>} \triangleleft \texttt{N}\{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}\ \texttt{FGJ+c}\end{array}}{[\![\texttt{C<}\overline{\texttt{N}}\texttt{>}]\!] = \texttt{C<}\overline{\texttt{N}}\texttt{, D}'\texttt{>}} \qquad \text{T1}$$

$$\frac{\begin{array}{c}\Delta_{\texttt{FGJ}} \vdash \texttt{T} = \texttt{C<}\overline{\texttt{N}}\texttt{>} \qquad \texttt{D}' = \mathit{bound}(\texttt{D}) \\ [\![\texttt{class C<}\overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N}\{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}]\!] = \texttt{class C<}\overline{\texttt{N}}\texttt{, O} \triangleleft \texttt{D>} \triangleleft \texttt{N}\{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}\end{array}}{[\![\texttt{C<}\overline{\texttt{N}}\texttt{>}]\!] = \texttt{C<}\overline{\texttt{N}}\texttt{, D}'\texttt{>}} \qquad \text{T2}$$

Defaulting Class Declarations (D-FGJ+C-CLASS):

$$\frac{\begin{array}{c}\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{, lastX} \triangleleft \texttt{lastN>} \triangleleft \texttt{N}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}\ \texttt{OK} \\ [\![\texttt{N}]\!] = \texttt{C}_{\texttt{N}}\texttt{<}\overline{\texttt{T}_{\texttt{N}}}\texttt{, Owner}_{\texttt{N}}\texttt{>} \qquad \neg\ (\mathit{bound}(\texttt{lastN}) \triangleleft \texttt{World}) \qquad \texttt{Owner}\ \mathit{free\ in}\ \texttt{C}\end{array}}{\begin{array}{c}[\![\texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{, lastX} \triangleleft \texttt{lastN>} \triangleleft \texttt{N}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}]\!] = \\ \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{, lastX} \triangleleft \texttt{lastN, Owner} \triangleleft \texttt{Owner}_{\texttt{N}}\texttt{>} \triangleleft \texttt{C}_{\texttt{N}}\texttt{<}\overline{\texttt{T}_{\texttt{N}}}\texttt{, Owner>}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}\end{array}} \qquad \text{C1}$$

$$\frac{\texttt{class C} \triangleleft \texttt{N}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}\ \texttt{OK} \qquad [\![\texttt{N}]\!] = \texttt{C}_{\texttt{N}}\texttt{<}\overline{\texttt{T}_{\texttt{N}}}\texttt{, Owner}_{\texttt{N}}\texttt{>} \qquad \texttt{Owner}\ \mathit{free\ in}\ \texttt{C}}{\begin{array}{c}[\![\texttt{class C} \triangleleft \texttt{N}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}]\!] = \\ \texttt{class C<Owner} \triangleleft \texttt{Owner}_{\texttt{N}}\texttt{>} \triangleleft \texttt{C}_{\texttt{N}}\texttt{<}\overline{\texttt{T}_{\texttt{N}}}\texttt{, Owner>}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}\end{array}} \qquad \text{C2}$$

Fig. 3. FGJ+c Defaulting Rules

Figure 3 shows[2] how to default any occurrence of a nonvariable type (O1, T1, and T2) and class declaration (C1 and C2). Rule O1 defaults every occurrence of `Object` type with `CObject<World>`, since every class in FGJ+c has to have an owner parameter — even the root of the class hierarchy [10]. Rules T1 and T2 replace the class instantiations that do not have owner parameters. The original class declaration may come from a valid FGJ+c class (T1) or may be defaulted by our class rules (T2). The class rules C1 and C2 appropriately insert a default owner parameter (that is bound by the owner bound of a superclass that is going to be `World` in case of plain FGJ programs) into every class declaration that does not already have one. The presence of the owner parameter is checked by testing the bound of the last type parameter

[2] Please note that FGJ and FGJ+c types are marked as valid using `OK` and `OK+c` correspondingly if they meet a number of type validity requirements [8, 10]. A class declaration is marked `FGJ+c` if it is a valid FGJ+c class.

(C1), unless there are none (C2). The rules for recursively propagating the class declaration and instantiation replacements through expressions, method and class declarations are straightforward and are omitted from this paper due to space constraints.

For vanilla FGJ programs, our defaulting guarantees that the resulting FGJ+c programs are valid (i.e. the translation is going to be sound). Indeed, in the resulting FGJ+c programs (see the end of section 2): (a) every class declaration and every type has an owner; (b) the owners are preserved over the inheritance, since the C1 and C2 rules explicitly ensure that superclasses have the same owner parameter; and (c) the owners are visible within the current context since for plain FGJ programs they are going to be `World` (propagated from the root of the hierarchy: `CObject<World>`).

For “mixed” FGJ and FGJ+c programs, our defaulting rules can no longer guarantee that the translation is sound, since the FGJ+c part of the program can use owner parameters in a way that invalidates the meaning of the program after the defaulting. Consider the following valid FGJ program (where $\pi_{\mathtt{p}}$ stands for the owner class of package `p`):

```
class util.List<T, Owner extends World> { ... }


class p1.Foo<πp1> {
  p1.Foo() {
    util.List<T,πp2> = new util.List<T,πp2>;
  }
}
```

This program is valid in FGJ, but it is invalid in FGJ+c and will be invalid after defaulting (since it accesses a class confined to package p2 outside package p2). These errors are only going to arise in the FGJ+c part of the program, not in vanilla FGJ.

FGJ utilises an *erasure* mechanism to translate FGJ programs to FJ (Featherweight Java without generics) [8]. Erasure removes the type parameters from type instantiations and adjusts the rest of a program (expressions, method and class declarations) so that the result preserves typing and reflects the original execution (unfortunately, the execution is not preserved). We can observe that any FGJ program and a defaulted version of it will reduce to the same FJ program under the erasure conditions. This means that typing and execution are not affected by the defaulting.

We have formalised defaulting within the context of static per-package confinement FGJ+c. We expect that defaulting can be expanded to cover shallow per-object ownership.

4 Discussion

Ownership defaulting is much weaker than the mechanisms provided by inference schemes [1, 3] since we only use *one* owner parameter and default it to `World`. Inferring only one owner parameter is less restrictive than it may seem, as all generic parameters can carry ownership information in OGJ. On the other hand, defaulting undeclared owners to `World` provides no encapsulation. However, type inference schemes can only infer types to describe the implicit encapsulation structures latent in the code. They do not enforce any particular encapsulation discipline: rather they will infer the equivalent of `World` ownership for any type which escapes the unit of analysis. This means that neither inference nor our defaulting can enforce encapsulation guarantees that the programmer has not specified, rather that inference will attempt to accurately identify latent encapsulation where defaulting currently will not.

We consider that simple defaulting rules are more appropriate for a general-purpose language such as OGJ — they are certainly simpler to explain and to understand. Thus, a programmer can easily omit unnecessary owner annotations in their code, while knowing exactly the consequences of their action. However, we plan to pursue ownership inference for OGJ programs as a programming tool outside the language, to support programmers adding explicit ownership declarations to programs to reflect their intentions.

Ownership defaulting described in this paper would also work for other ownership type systems [1, 4, 5]. A minor (but still important) language design feature makes defaulting in OGJ considerably easier than in the other systems: in OGJ, an object's primary ownership is carried by the *last* parameter, whereas in most other systems (following Clarke et al. [6]) it is carried by the *first*. Having ownership in the last position means that owners can be defaulted simply by leaving them off (e.g. "`Foo`") while Boyapati et.al., for example, require a placeholder (e.g. "`Foo<->`") which is referred to as an *anonymous owner*.

Defaulting provides backwards compatibility for OGJ with Generic Java in the same way that raw types provide backwards compatibility for GJ programs with plain Java. Generic ownership [11] makes it possible to reuse the formal foundations provided by FGJ, thus allowing to obtain such a compatibility easily.

Ownership defaulting only guarantees the validity of resulting FGJ+c programs if the original FGJ program is naked. In practice, not making such a guarantee for mixed programs means that the OGJ will show the errors in the mixed code that were not detected by the Java 1.5 compiler. But the errors are going to be in the OGJ part of the code and any vanilla Java 1.5 code will compile correctly if taken on its own.

5 Conclusion

In this paper, we described ownership defaulting mechanism for making any Java 1.5 program fully ownership parametric. Defaulting makes it possible to utilise ownership in day to day programming. A programmer can take OGJ, a prototype of which is available, and start using ownership from day one. We hope that ownership defaulting will be yet another step towards making ownership usable in the practical world of object-oriented programming.

Acknowledgments

This work is supported in part by the Royal Society of New Zealand Marsden Fund.

References

1. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)*, November 2002.
2. Boris Bokowski and Jan Vitek. Confined types. In *Proceedings of Conference on Object-Oriented Programming, Languages, and Applications*. ACM Press, 1999.
3. Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, February 2004.
4. Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)*, 2001.
5. Dave Clarke. *Object ownership and containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Australia, 2002.
6. David Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of Conference on Object-Oriented Programming, Languages, and Applications*. ACM Press, 1998.
7. Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with Confined Types. In *Proceedings of Conference on Object-Oriented Programming, Languages, and Applications*. ACM Press, 2001.
8. 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.
9. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. A recipe for raw types. In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2001.
10. Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Featherweight generic confinement. In *Foundations of Object-Oriented Programming (FOOL11)*, Venice, Italy, January 2004.
11. Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)*, 2004. Submitted for publication.
12. Sun Microsystems. JSR14 prototype implementation. `http://developer.java.sun.com/developer/earlyAccess/adding_generics/ind%ex.html`, 2003.