

# A Summary of *AliasJava* and some related works

Daniel Khodorkovsky

December 10, 2002

## AliasJava

AliasJava [AKC02] is a type annotation system for placing structural and temporal bounds on data sharing in Java programs.

In the annotation system, *unique* describes an unshared reference, *owned* objects are assigned an *owner* that controls access to the object, and a *shared* reference is globally-aliased. In addition, *owned* objects can be *lent* to a method call.

AliasJava combines uniqueness and ownership annotations to express idioms that neither system is capable of expressing in isolation. The AliasJava checker extends to the full Java language, including arrays, casts, inheritance, and inner classes.

Alias annotations can be automatically inferred by a constraint-based algorithm. The time to infer annotations is comparable to that of parsing and annotation checking, but currently the algorithm produces too many inferred parameters.

## Related Work

Several related works use the concept of unique or linear types to help with pointer-induced aliasing. Minsky [Min96] argues that *unique pointers* and *unshareable objects*, in addition to affording the standard benefits of aliasing reduction, make storage management safer and more efficient.

Ownership annotations used in AliasJava are closely related to *aliasing modes* from Flexible Alias Protection [NVP98]. In Flexible Alias Protection, static checking of code annotations makes guarantees about a program's aliasing properties. Alias modes include *rep* and *arg* to label a container's private *representation* and its public *arguments*.

## Kacheck/J

The ownership typing model of AliasJava is related to the concept of package-based confinement. Confinement bounds aliasing of encapsulated objects to the defining package of their class.

Kacheck/J [GPV01] is a fast and scalable system for inferring confined types. The tool discovers potential confinement violations and lists all confined types for each package analyzed. The analysis operates at the bytecode level and therefore allows use on library code.

The results of the paper show that about 25% of all package-scoped classes and interfaces from the PBS (Purdue Benchmark Suite) are confined.

## Cyclone

Cyclone [GMJ+02] is a type-sound variant of C that uses region-based memory management to insure soundness. Cyclone is designed to eliminate dangling-pointer dereferences and memory leaks.

The system is based on the notion of *regions*: each object lives in one region, and a region's objects are all deallocated simultaneously (except for the special heap region which may be garbage collected).

All pointers point into exactly one region. Pointer types are annotated with the *region name*. In the following example, `r_L` is the name of the region corresponding to the block labeled L.

```
int *r_L p;
L:{
int x = 0;
p = &x;
}
*p = 42;
```

Cyclone rejects this code because `r_L` is not in scope at the point of `p`'s declaration.

Although Cyclone is explicitly typed, the authors use a combination of inference and well-chosen defaults to reduce drastically the number of annotations needed in practice. As a result, authors find that the annotation burden is negligible for application code, but increases for library code.

The performance of networking programs written in Cyclone essentially equals that of their C counterparts, whereas CPU-bound applications are up to a factor of three slower due to run-time checks and the overhead of pointer representation as fat pointers.

## Parameterized Race-Free Java

Boyapati and Renard [BR01] present a static types system for multi-threaded programs; any well-typed program in this system is provably free of data races. Parameterized Race-Free Java (PRFJ) allows to write generic code to implement a class and then create different objects of the same class that have different protection mechanisms.

Every object in the system is associated with a *protection mechanism* that ensures that accesses to the object never create data races. The system enables programmers to specify the protection mechanism for each object as part of the

type of the variables that refer to that object. The type can specify either the mutual exclusion lock that protects the object from unsynchronized concurrent accesses, or that threads can safely access the object without synchronization because either the object is immutable, the object is accessible to a single thread and is not shared between threads, or the variable contains the unique reference of the object.

The key to the type system is the concept of object ownership. Every object in the system has an owner: another object, itself, or a special per-thread owner called `thisThread`. Objects owned by `thisThread` are local to the corresponding thread and cannot be accessed by any other thread.

To enable object migration from one thread to another, PRFJ uses the notion of unique pointers. In addition, the system supports read-only objects; hence they can be accessed without using any synchronization operations.

PRFJ does not provide a way to specify temporal properties (many threads accessing disjoint elements of the same array).

## JFlow

A final branch of similar research is secure information flow. A representative system is JFlow [Mye99], which implements a subset of Java with a statically-checked ownership and capabilities model.

Every value has a *labeled type* that consists of a regular Java type and a *label*. A label specifies a set of *owners* and their corresponding set of *readers*. For example, the declaration `int {o: r} z` specifies that integer variable `z` is owned by `o` and can be accessed by `r`, in addition to `o`. If a local variable is missing a label, JFlow will infer it automatically based on its use. In this system, classes and methods are also label-parameterized.

JFlow treats static checking of flow annotations as an extended form of type checking. The compiler is a source-to-source translator, so its output is a standard Java program that can be compiled by any Java compiler. The run-time, data and code space overheads are minimal. For the most part, the translation process involves removal of the static annotations in the JFlow language, with the exception of certain run-time constructs, e.g., dynamic testing labels.

## References

- [AKC02] Jonathan Aldrich, Valentin Konstadinov, and Craig Chambers. *Alias Annotations for Program Understanding*. Proc. Object-Oriented Programming, Systems, Languages, and Applications, November 2002, Seattle, Washington.
- [Min96] Naftaly Minsky. *Towards Alias-Free Pointers*. Proc. European Conference on Object-Oriented Programming, Linz, Austria, July 1996.
- [GMJ+02] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. *Region-Based Memory Management in Cy-*

*clone*. Proc. Programming Language Design and Implementation, Berlin, Germany, June 2002.

[GPV01] Christian Grothoff, Jens Palsberg, and Jan Vitek. *Encapsulating Objects with Confined Types*. Proc. Object-Oriented Programming Languages, Systems, and Applications, Tampa, Florida, November 2001.

[NVP98] James Noble, Jan Vitek, and John Potter. *Flexible Alias Protection*. Proc. European Conference on Object-Oriented Programming, Brussels, Belgium, 1998.

[BR01] Chandrasekhar Boyapati and Martin Renard. *A Parameterized Type System for Race-Free Java Programs*. Proc. Object-Oriented Programming Systems, Languages, and Applications, Tampa, Florida, October 2001.

[Mye99] Andrew C. Myers. *JFlow: Practical Most-Static Information Flow Control*. Proc. Principles of Programming Languages, San Antonio, Texas, January 1999.