Universe Types

Topology, encapsulation, genericity, and tools

Doctoral Thesis ETH No. 18522

Universe Types
Topology, Encapsulation, Genericity, and Tools

A dissertation submitted to the

Swiss Federal Institute of Technology Zurich
(ETH Zurich, Switzerland)

for the degree of

Doctor of Sciences

presented by

Werner Michael Dietl

Diplom-Ingenieur, Universität Salzburg
born December 7^{th}, 1976
citizen of Austria

accepted on the recommendation of

Prof. Dr. Peter Müller, examiner
Prof. Dr. Michael D. Ernst, co-examiner
Prof. Dr. Martin Odersky, co-examiner

2009

Abstract

We present *Generic Universe Types*, a sound, lightweight ownership type system for type-generic object-oriented programming languages, which cleanly separates the ownership topology from the owner-as-modifier encapsulation discipline and is supported by a comprehensive set of tools.

Mutable references give object-oriented programming languages the power to build complex object structures and to efficiently modify them. However, this power comes at the cost of *aliasing*: two or more references to the same object can exist, and the modifications performed through one reference are visible through all other references. In main-stream object-oriented languages, like Java and C♯, objects and references build a complicated mesh and there is no support to structure the heap. Visibility modifiers (such as `private` and `protected` in Java) only deal with information hiding, for example, ensuring that a field is only accessible from within its declaring class. However, the object referenced by that field might still be aliased and modified by other objects at runtime. There is no support to encapsulate object structures and to ensure that objects at runtime are only accessed in a controlled fashion.

Aliasing and the unstructured nature of the heap lead to problems with, for example, understanding the behavior of a program, adapting a consistent locking discipline to ensure correct concurrent behavior, exchanging the implementation of an interface, and the formal verification of properties.

The concept of *object ownership* has been proposed as a mechanism to structure the heap hierarchically and to provide encapsulation of object structures. Each object is assigned at most one other object as its owning object, and restrictions are enforced on the references. Ownership type systems provide static type annotations to enforce an ownership topology and encapsulation.

For maintaining invariants, the existence of aliases is no problem, as long as these aliases are not used to modify the internal representation of a different object. We call this encapsulation system the owner-as-modifier discipline, because it guarantees that the owner object is always in control of modifications. The Universe type system is a lightweight ownership type system that enforces the owner-as-modifier discipline and supports the modular verification of object-oriented programs.

We define Generic Universe Types (GUT), a combination of type genericity with Universe types. GUT subsumes the previous non-generic Universe type system and cleanly separates the topology from the encapsulation system. We give a complete formalization of the GUT system and prove it sound.

Usually, ownership type systems entangle the enforcement of the ownership topology with the enforcement of an encapsulation system; that is, the structuring of the heap and the restrictions on the use of references are enforced together. In this thesis, we cleanly separate the ownership topology from the encapsulation system, giving a cleaner formalization and allowing the separate reuse.

Finally, we discuss the integration of a Generic Universe Types checker into the Java Modeling Language (JML) compiler suite, the support for Java 7 (JSR 308) annotations, and Scala compiler plug-ins. We also illustrate the automatic inference of ownership modifiers using a static and a runtime approach.

Zusammenfassung

Wir präsentieren *Generic Universe Types*, ein typsicheres, leichtgewichtiges *Ownership*-Typsystem für typ-generische objektorientierte Programmiersprachen, welches die Ownership-Topologie klar von der *Owner-as-Modifier* Kapselungsdisziplin trennt und von einer umfassenden Menge an Werkzeugen unterstützt wird.

Veränderbare Referenzen geben objektorientierten Programmiersprachen die Ausdrucksstärke, komplexe Objektstrukturen aufzubauen und sie effizient zu modifizieren. Für diese Ausdrucksstärke nimmt man jedoch *Aliasing* in kauf: zwei oder mehr Referenzen die das selbe Objekt referenzieren. Modifikationen die durch eine Referenz ausgeführt werden sind auch durch alle anderen Referenzen sichtbar. In geläufigen objektorientierten Programmiersprachen, wie Java und C♯, bilden Objekte und Referenzen ein komplexes Netzwerk und es gibt keine Unterstützung, um den Speicher zu strukturieren. Die Zugriffsmodifikatoren (wie zum Beispiel `private` und `protected` in Java) stellen nur das Geheimnisprinzip sicher, zum Beispiel, dass auf ein Feld nur innerhalb seiner deklarierenden Klasse zugegriffen werden kann. Jedoch kann das vom Feld referenzierte Objekt zur Laufzeit trotzdem mehrfach referenziert werden und über einen Alias modifiziert werden. Es gibt keine Unterstützung der Datenkapselung und keinen Mechanismus um sicherzustellen, dass Objekte zur Laufzeit nur in einer kontrollierten Art benutzt werden.

Aliasing und der unstrukturierte Speicher führen zu einigen Problemen, zum Beispiel beim Programmverständnis, beim Verwenden einer konsistenten Sperrdisziplin für Monitore um korrektes paralleles Verhalten sicherzustellen, beim Austauschen der Implementierung einer Schnittstelle und bei der formalen Verifikation von Programmeigenschaften.

Das Konzept der *Object Ownership* wurde als Mechanismus vorgeschlagen, um den Speicher hierarchisch zu strukturieren und um Datenkapselung für Objektstrukturen sicherzustellen. Jedes Objekt gehört zu maximal einem anderen Objekt und Beschränkungen auf die möglichen Referenzen werden eingehalten. Ownership-Typsysteme verwenden statische Typannotationen um eine Ownership-Topologie und Datenkapselung sicherzustellen.

Für den Erhalt von Invarianten sind verschiedene Referenzen auf das selbe Objekt kein Problem, solange ein Alias nicht zum Verändern der internen Repräsentation eines anderen Objekts verwendet wird. Wir bezeichnen diese Datenkapselungseigenschaft als die Owner-as-Modifier-Disziplin, weil sie sicherstellt, dass der Besitzer eines Objekts Veränderungen am Objekt kontrollieren kann. Das Universe Typsystem ist ein leichtgewichtiges Owernship-Typsystem das die Owner-as-Modifier-Disziplin erzwingt und dadurch die modulare Verifikation von objektorientierten Programmen ermöglicht.

Wir definieren Generic Universe Types (GUT), eine Kombination von Typgenerizität mit dem Universe Typsystem. GUT subsumiert das nicht-generische Universe Typsystem und trennt die Ownership-Topologie klar von der Datenkapselung. Wir geben eine komplette Formalisierung von GUT und zeigen die Fehlerfreiheit.

Normalerweise vermischen Ownership-Typsysteme das Sicherstellen einer Ownership-Topologie und der Datenkapselung, das heisst, die Strukturierung des Speichers und die Verwendung

von Referenzen werden gemeinsam beschränkt. In dieser Doktorarbeit trennen wir diese beiden Konzepte konsequent und bekommen dadurch eine klarere Formalisierung und können die Komponenten getrennt wiederverwenden.

Schlussendlich diskutieren wir die Integration von Generic Universe Types in den Kompiler der Java Modellierungssprache JML, die Unterstützung von Java 7 (JSR 308) Annotationen und die Verwendung von Erweiterungen für den Scala Übersetzer. Wir illustrieren auch wie Ownership Annotationen automatisch inferiert werden können und präsentieren dafür einen statischen und einen dynamischen Ansatz.