Universe Types
Topologie, Kapselung, Generizität und Tools

Werner M. Dietl
University of Washington, Seattle, WA, USA
http://www.cs.washington.edu/homes/wmdietl/

Abstract: Mehrere veränderbare Referenzen auf ein Objekt geben objektorientierten Programmiersprachen Ausdruckstärke, verkomplizieren allerdings das Programmverständniss und verunmöglichen verschiedenste formale Beweise von Programmen. Als Lösungsmöglichkeit für diese Probleme bieten sich *Ownership*-Typsysteme an, die den Speicher hierarchisch strukturieren und die möglichen Referenzen und deren Effekte beschränken.

Generic Universe Types sind ein typsicheres, leichtgewichtiges *Ownership*-Typsystem für typgenerische objektorientierte Programmiersprachen, das die Ownership-Topologie klar von der *Owner-as-Modifier*-Kapselungsdisziplin trennt und von einer umfassenden Menge an Tools unterstützt wird.

In meiner Doktorarbeit gebe ich zuerst einen ausführlichen Überblick über die Problemstellung und definiere dann eine Programmiersprache und eine entsprechende Laufzeitumgebung. Danach präsentiere ich das GUT Typsystem das die Ownership-Topologie sicherstellt und darauf aufbauend separate Regeln um die Owner-as-Modifier-Kapselungsdisziplin sicherzustellen. Die Doktorarbeit formalisiert die Eigenschaften des Systems und beweist ausführlich deren Gültigkeit. Weiters beschreibe ich die von mir entwickelten Tools, nämlich verschiedene Typchecker und Inferenz-Tools.

In dieser Kurzfassung meiner Doktorarbeit stelle ich den Problembereich dar und präsentiere anhand von Beispielen die Grundgedanken des Typsystems.

1 Einführung

In meiner Doktorarbeit definiere ich *Generic Universe Types* (GUT), eine Kombination von Typgenerizität mit dem leichtgewichtigen Ownership-Typsystem Universe Types. Die nicht-generischen Universe Types werden von GUT subsumiert und die Ownership-Topologie wird in GUT klar von der Datenkapselung getrennt.

Veränderbare Referenzen geben objektorientierten Programmiersprachen die Ausdrucksstärke, komplexe Objektstrukturen aufzubauen und sie effizient zu modifizieren. Für diese Ausdrucksstärke nimmt man jedoch *Aliasing* in Kauf: Dies sind 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. Betrachten wir die folgende Klasse:

```
class Aliasing {
  private Data internal;

  public void setData(Data p) {
    // capturing
    internal = p;
  }

  public Data getData() {
    // leaking
    return internal;
  }
}
```

Die Klasse `Aliasing` möchte ein Objekt der Klasse `Data` als interne Repräsentation im Feld `internal` verwalten. Jedoch wird durch den Zugriffsmodifikator `private` nur das Geheimnisprinzip sichergestellt, das heisst, nur die Sichtbarkeit des Feldes wird beeinflusst, das referenzierte Objekt kann trotzdem extern referenziert werden. Die Methode `setData` speichert direkt eine Referenz auf das übergebene `Data`-Objekt als interne Repräsentation ab. Dadurch wird eine zusätzliche Referenz auf das Objekt erzeugt und externe Modifikationen sind für `Aliasing` sichtbar. Ähnlich gibt Methode `getData` eine Referenz auf das vermeintlich gekapselte Objekt zurück und wird dadurch abhängig von externen Modifikationen. Diese als *capturing* und *leaking* bekannten Phänomene erzeugen sehr leicht mehrere Aliase für ein Objekt.

Aliasing und der unstrukturierte Speicher führen zu vielen 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 hat maximal ein Owner-Objekt. Die Menge aller Objekte mit dem gleichen Owner wird *Kontext* genannt. Der *Wurzelkontext* ist die Menge der Objekte ohne Owner-Objekt. Die Ownership-Beziehung bildet einen Baum. Beschränkungen auf die möglichen Referenzen und deren Effekte werden eingehalten. Ownership-Typsysteme verwenden Typannotationen um eine Ownership-Topologie und Datenkapselung statisch sicherzustellen.

Ownership wurde erfolgreich zur Lösung verschiedener Probleme genutzt, zum Beispiel der Programmverifikation, der Synchronisation von Threads, der Speicherverwaltung und um die Unabhängigkeit von Implementierungen zu zeigen.

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. Man bezeichnet diese Datenkapselungseigenschaft als die *Owner-as-Modifier*-Disziplin, weil sie sicherstellt, dass der Besitzer eines Objekts Veränderungen am Objekt kontrollieren kann. Universe Types sind ein leichtgewichtiges Ownership-Typsystem, das die Owner-as-Modifier-Disziplin erzwingt und dadurch die modulare Verifikation von objektorientierten Programmen ermöglicht.

Die in meiner Doktorarbeit definierten Generic Universe Types (GUT) sind ein leichtgewichtiges Ownership-Typsystem für objektorientierte Programmiersprachen mit Typgenerizität. 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 meiner Doktorarbeit trenne ich diese beiden Konzepte konsequent und erziele dadurch eine klarere Formalisierung und kann die Komponenten getrennt wiederverwenden. Ich gebe eine komplette Formalisierung von GUT und beweise die Fehlerfreiheit. Schlussendlich diskutiere ich die Integration von Generic Universe Types in den Compiler der Java Modellierungssprache JML, die Unterstützung von Java 7 (JSR 308) Annotationen und die Verwendung von Erweiterungen für den Scala Compiler. Ich illustriere auch wie Ownership-Annotationen automatisch inferiert werden können und präsentiere dafür einen statischen und einen dynamischen Ansatz.

In dieser Kurzfassung präsentiere ich anhand von Beispielen die Grundgedanken des Typsystems. Die Formalisierung, Beweise, Beispiele und Tools findet man in meiner Doktorarbeit [Die09].

2 Grundlegende Konzepte

In diesem Abschnitt erläutere ich die wichtigsten Konzepte von Generic Universe Types (GUT) informell anhand eines generischen `Map` Beispiels.

Klasse `Map` (Abb. 2) implementiert eine generische Abbildung von Schlüsseln auf Werte. Schlüssel-Wert-Paare sind in einfach-verknüpften `Node`-Objekten (Abb. 3) gespeichert. Die `main`-Methode der Klasse `Client` (Abb. 4) baut die Map-Struktur in Abb. 1 auf. Aus Gründen einer vereinfachten Darstellung lasse ich die Zugriffsmodifikatoren von allen Beispielen weg.

2.1 Ownership-Annotationen

Ein Typ in GUT ist entweder eine Typvariable oder besteht aus einer Ownership-Annotation, dem Namen einer Klasse und möglicherweise Typargumenten. Die *Ownership-Annotation* drückt die Ownership-Beziehung relativ zum aktuelle Receiver-Objekt `this` aus[1]. Programme können die Ownership-Annotationen `peer`, `rep` und `any` enthalten. Diese haben die folgenden Bedeutungen:

- `peer` drückt aus, dass das Objekt den gleichen Owner wie das `this`-Objekt hat. Das heißt, dass das aktuelle Objekt und das referenzierte Objekt den gleichen Owner haben und daher im gleichen Kontext sind.

[1] Statische Felder und Methoden ignoriere ich hier, aber eine Erweiterung ist möglich.

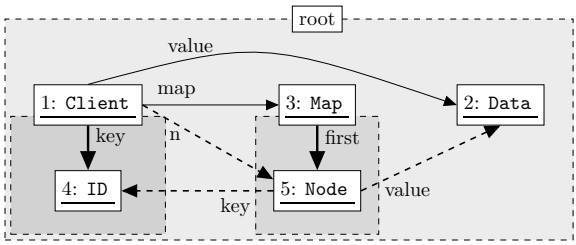

Abbildung 1: Objektstruktur einer Map von `ID`- zu `Data`-Objekten. Die Map wird von `Nodes` dargestellt. Das `Client`-Objekt hat eine direkte Referenz auf einen `Node`. Objekte, Referenzen und Kontexte werden jeweils durch Rechtecke, Pfeile und gestrichelte Rechtecke dargestellt. Owner-Objekte sitzen auf dem Rechteck um die Objekte, die sie besitzen. Die Pfeile sind mit den Namen der Variablen markiert, die die Referenz speichern. Gestrichelte Pfeile zeigen Referenzen, die Kontext-Grenzen überschreiten, ohne über den Owner zu gehen. Wenn die Owner-as-Modifier-Disziplin durchgesetzt wird, können solche Referenzen nicht den Zustand des referenzierten Objekts verändern. Der Quellcode für dieses Beispiel wird in Abb. 2, 3 und 4 gezeigt.

- `rep` drückt aus, dass das Objekt `this` als Owner hat. Das heißt, dass das aktuelle Objekt der Owner des referenzierten Objekts ist.
- `any` drückt aus, dass das Objekt einen beliebigen Owner haben kann. Die `any`-Annotation ist eine "don't care" Annotation und drückt aus, dass der Owner des referenzierten Objekts bewusst für diese Referenz nicht näher spezifiziert wird; `any`-Typen sind daher Supertypen der `rep` und `peer`-Typen mit der gleichen Klasse und gleichen Typargumenten, da `any`-Typen weniger spezifische Informationen über die Ownership-Beziehung vermitteln.

Die Nutzung der Ownership-Annotationen wird durch Klasse `Map` (Abb. 2) veranschaulicht. Ein `Map`-Objekt besitzt seine `Node`-Objekte, da sie die interne Repräsentation der `Map` bilden. Diese Ownership-Beziehung wird durch die `rep`-Annotation von `Map`'s Feld `first` ausgedrückt, welches auf den ersten `Node` der `Map` zeigt.

Intern verwendet das Typsystem zwei zusätzliche Ownership-Annotationen, `self` und `lost`:

- `self` wird nur als Annotation für das aktuelle Objekt `this` verwendet und unterscheidet das aktuelle Objekt von anderen Objekten, die den gleichen Owner haben. Die Verwendung einer separaten `self`-Annotation hebt die besondere Rolle, die das aktuelle Objekt in Ownership-Systemen spielt, hervor und vereinfacht das Gesamtsystem durch die Beseitigung von Sonderfällen für Zugriffe über `this`.
- `lost` bedeutet, dass die Ownership-Beziehung statisch nicht mit einer der anderen Ownership-Annotationen ausgedrückt werden kann. Es ist eine "don't know" Annotation, die anzeigt, dass Ownership-Informationen "verloren" wurden; im Gegensatz zur `any`-Annotation könnten für die Referenz konkrete Ownership-Informationen benötigt werden.

```
class Map<K, V> {
  rep Node<K, V> first;

  void put(K key, V value) {
    rep Node<K, V> newfirst = new rep Node<K, V>();
    newfirst.init(key, value, first);
    first = newfirst;
  }

  pure V get(any Object key) {
    rep Node<K, V> n = getNode(key);
    return n != null ? n.value : null;
  }

  pure rep Node<K, V> getNode(any Object key) {
    rep Node<K, V> n = first;
    while (n != null) {
      if (n.key.equals(key)) return n;
      n = n.next;
    }
    return null;
  }
}
```

Abbildung 2: Eine Implementierung einer generischen Map. `Map`-Objekte besitzen ihre `Node`-Objekte, was durch die `rep`-Annotationen in allen Vorkommnissen der Klasse `Node` erwirkt wird.

Die Owner-as-Modifier-Kapselungdisziplin wird von GUT separat erzwungen, indem Veränderungen von Objekten auf `self`, `peer` und `rep` Receiver eingeschränkt werden. Ein Ausdruck mit `lost` oder `any` Ownership kann als Receiver für einen Lesezugriff auf ein Feld oder einen Aufruf einer nebeneffekt-freien (*puren*) Methode verwendet werden, aber nicht für einen Schreibzugriff oder einen Aufruf einer nicht-puren Methode. Um diese Eigenschaft zu überprüfen, erfordert das Kapselungssystem das nebeneffekt-freie Methoden mit dem Schlüsselwort `pure` annotiert werden. Diese Unterscheidung zwischen puren und nicht-puren Methoden ist nicht relevant für das topologische System und kann optional erzwungen werden.

2.2 Viewpoint-Anpassung

Da die Ownership-Annotationen Ownership relativ zum aktuellen `this`-Objekt ausdrücken, müssen sie angepasst werden, wenn sich dieser "Viewpoint" ändert. Betrachten wir `Node`'s Methode `init` (Abb. 3). Der dritte Parameter hat den Typ `peer Node<K, V>` und wird verwendet, um das `next` Feld zu initialisieren. Die `peer`-Annotation drückt aus, dass das Parameterobjekt den gleichen Owner wie der Receiver der Methode haben muss. Andererseites ruft `Map`'s Methode `put` Methode `init` auf einem `rep Node` Receiver auf, das heißt, einem Objekt, dessen Owner `this` ist. Daher muss das dritte Argument des Aufrufs von `init` auch `this` als Owner haben. Dies bedeutet, dass aus diesem Viewpoint das dritte Argument eine `rep`-Annotation haben muss, obwohl der Parameter mit einer `peer`-Annotation deklariert ist. Im Typsystem wird diese *Viewpoint-Anpassung*

```
class Node<K, V> {
  K key; V value;
  peer Node<K, V> next;

  void init(K k, V v, peer Node<K, V> n) {
    key = k; value = v; next = n;
  }
}
```

Abbildung 3: `Node`s bilden die interne Repräsentation der Map. Klasse `Node` implementiert eine einfach-verkettete Liste von Schlüsseln und Werten.

durch die Kombination des Typs des Receivers eines Aufrufs (hier `rep Node<K, V>`) mit dem Typ des formalen Parameters (hier `peer Node<K, V>`) erzielt. Diese Kombination liefert den Argumenttyp aus der Sicht des Aufrufers (hier `rep Node<K, V>`).

Viewpoint-Anpassung kann zur Folge haben, dass Ownership-Information verloren geht, wenn die Ownership-Beziehung nicht vom neuen Viewpoint ausdrückbar ist. Als Beispiel kann man sich vorstellen, dass es einen zusätzlichen Feldzugriff `map.first` in Abb. 4 gibt; die Viewpoint-Anpassung des Feldtyps, `rep Node<K, V>`, ergibt eine `lost`-Annotation, weil es keine Ownership-Annotation gibt um genau auszudrücken, dass es sich um eine Referenz in die Repräsentation des Objekts `map` handelt. Als eine Konsequenz, um die Korrektheit des topologischen Systems sicherzustellen, können Methoden nicht direkt ein `rep`-Feld eines Objekts ändern, das nicht `this` ist.

Wenn jedoch nur das topologische System eingesetzt wird, können Referenzen mit verlorener oder beliebiger Ownership-Information weiterhin als Receiver verwendet werden. Betrachten wir das Hauptprogramm in Abb. 4. Lokale Variable `n` speichert eine Referenz in die Repräsentation eines anderen Objekts, da Methode `getNode` eine Referenz auf die internen Nodes der `peer Map` zurückgibt. Das Update `n.key` ist gültig, da es die Topologie des Speichers aufrechterhält. Das System ist in voller Kenntnis des Feldtyps nach Viewpoint-Anpassung und keine Ownership-Information geht verloren. Andererseits ist das Update des Feldes `next` untersagt. Nach der Viewpoint-Anpassung enthält der Typ der linken Seite eine `lost` Ownership-Annotation und somit kann die Topologie des Speichers statisch nicht gewährleistet werden.

Viewpoint-Anpassung und die Owner-as-Modifier-Disziplin sorgen für Kapselung der internen Repräsentation von Objekten. Betrachten wir wieder Methode `getNode` der Klasse `Map`. Durch die Viewpoint-Anpassung des Rückgabetyps, `rep Node<K, V>`, können Benutzer der Map nur eine `lost` Referenz auf den `Node` erhalten. Die Owner-as-Modifier-Disziplin erfordert einen `self`, `peer` oder `rep` Receiver-Typ für Veränderungen und gewährleistet damit, dass Benutzer nicht direkt die Node-Struktur ändern können. Dies ermöglicht der Map die Aufrechterhaltung von Invarianten über die Nodes, zum Beispiel, dass die Node-Struktur azyklisch ist.

2.3 Typparameter

Ownership-Annotationen werden auch in Typargumenten verwendet. Zum Beispiel instanziiert `Client`'s Methode `main` ein `Map`-Objekt mit den Typargumenten `rep ID` und `any Data`. Daher hat Feld `map` den Typ `peer Map<rep ID, any Data>`, welcher drei Ownership-Annotationen hat. Die *Hauptannotation* `peer` drückt aus, dass das `Map`-Objekt den gleichen Owner wie `this` hat, während die *Argumentannotationen* `rep` und `any` die Ownership von Schlüsseln und Werten bezogen auf das `this`-Objekt ausdrücken, in diesem Fall, dass die Schlüssel `ID`-Objekte mit Owner `this` sind und dass die Werte `Data`-Objekte in einem beliebigen Kontext sind. Besonders hervorzuheben ist, dass die Argumentannotationen wieder Ownership relativ zum `this`-Objekt (hier dem `Client`-Objekt) ausdrückt und nicht relativ zur Instanz der generischen Klasse ist, die die Argumentannotationen enthält (hier dem `Map`-Objekt `map`).

Typvariablen unterliegen nicht der Viewpoint-Anpassung, die für Nicht-Variable-Typen durchgeführt wird. Wenn Typvariablen verwendet werden, z.B. in Felddeklarationen, bleibt die Ownership-Information implizit und muss daher nicht angepasst werden. Die Substitution von Typvariablen durch ihre tatsächlichen Typargumente geschieht wenn die Typvariable instanziiert wird. Daher ist der Viewpoint der gleiche wie für die Instanziierung und es ist keine Viewpoint-Anpassung erforderlich. Zum Beispiel können wir uns vorstellen, es gäbe einen Lesezugriff `n.key` in Methode `main` (Abb. 4). Der deklarierte Typ des Feldes ist die Typvariable `K`. Das Lesen durch die Referenz `n` ersetzt die Typvariable durch das tatsächliche Typargument `rep ID`, und führt keine Viewpoint-Anpassung aus.

Obwohl die Klasse `Map` nicht die Owner der Schlüssel und Werte kennt (wegen der impliziten `any` oberen Schranke für `K` und `V`, siehe unten), können Benutzer der Map die genauen Informationen über die Ownership-Beziehung aus den Typargumenten wiederherstellen. Dies verdeutlicht, dass Generic Universe Types starke statische Garantien gibt, ähnlich denen von owner-parametrischen Systemen, auch in Gegenwart von `any`-Typen. Die entsprechende Umsetzung in nicht-generischen Universe Types erfordert einen Cast von einem `any`-Typen zu einem `rep`-Typen und der entsprechenden Laufzeitüberprüfung.

Typvariablen haben obere Schranken, die standardmäßig `any Object` sind. In einer Klasse C drückt die Ownership-Annotation einer oberen Schranke die Ownership gegenüber der C-Instanz `this` aus. Aber wenn C-Typvariablen instanziiert werden, sind die Annotationen der Typargumente bezogen auf den Receiver der Methode. Daher benötigt die Überprüfung der Konformität eines Typarguments mit seiner oberen Schranke eine Viewpoint-Anpassung. Ebenso haben Methoden-Typvariablen obere Schranken, die relativ zur aktuellen Instanz der Klasse sind.

2.4 Die `lost` und `any`-Annotationen

Es gibt einen fundamentalen Unterschied zwischen einem generischen Typ, der einen beliebigen Owner als Typargument verwendet und einem generischen Typ, der einen unbekannten Owner als Typargument verwendet.

```
class ID { /* ... */ }
class Data { /* ... */ }

class Client {
  peer Map<rep ID, any Data> map;

  void main() {
    map = new peer Map<rep ID, any Data>();
    peer Data value = new peer Data();
    rep ID key = new rep ID();
    map.put(key, value);

    any Node<rep ID, any Data> n = map.getNode(key);
    n.key = new rep ID(); // OK
    n.next = new rep Node<rep ID, any Data>(); // Error
  }
}
```

Abbildung 4: Hauptprogramm für unser Map Beispiel. Die Ausführung der Methode `main` erzeugt die Objektstruktur in Abb. 1.

```
class ClientUser {
  void useMap(peer Client client) {
    client.map.put(new rep ID(), new peer MyData()); // Error
  }
}
```

Abbildung 5: Viewpoint-Anpassung der Map ergibt verlorene Ownership-Information.

Zum Beispiel hat die Map aus Abb. 4 den Typ `peer Map<rep ID, any Data>` und drückt damit aus, dass das `Map`-Objekt den gleichen Owner wie das aktuelle Objekt hat und dass die Schlüssel `ID`-Objekte sind deren Owner das aktuelle Objekt ist, sowie dass die Werte `Data`-Objekte sind, die beliebige Owner haben.

Die Anpassung eines Typarguments ergibt eine `lost`-Annotation, wenn Ownership-Informationen nicht vom neuen Viewpoint ausgedrückt werden konnten, wie es in Abb. 5 veranschaulicht wird. Der Typ des Lesezugriffs `client.map` ist `peer Map<lost ID, any Data>`. Es kann statisch ausgedrückt werden, dass die Map selbst im gleichen Kontext wie das aktuelle Objekt ist, und auch, dass die Werte in einem beliebigen Kontext sind. Aber von diesem neuen Viewpoint aus kann nicht ausgedrückt werden, dass der Schlüssel als Owner die Client-Instanz `client` haben muss; es gibt keine spezifischere Ownership-Annotation für diese Beziehung und daher wird die `lost`-Annotation verwendet. Es wäre nicht typsicher, den Aufruf der Methode `put` auf einen Receiver dieses Typs zu erlauben. Die Signatur für Methode `put` enthält nach Viewpoint-Anpassung und Substitution `lost` und das topologische System kann die genaue Ownership-Beziehung, die für das erste Argument erforderlich ist, nicht ausdrücken. Andererseites enthält die Signatur der Methode `get` kein `lost` und kann noch immer aufgerufen werden. Die Methoden `get` und `getNode` verwenden `any Object` als Parametertypen und nicht die Typvariable `K`. Das Verwenden der oberen Schranke einer Typvariable anstelle der Typvariable ermöglicht es, eine Methode aufzurufen, selbst wenn die tatsächliche Ownership eines Typarguments `lost` ist. Dies ist besonders nützlich für Methoden, die den Speicher nicht verändern, insbesonders also für pure Methoden.

```
class Cast {
  void m(any Object obj) {
    peer Map<rep ID, any Data> map = (peer Map<rep ID, any Data>) obj;
    map.put(new rep ID(), new peer Data());
  }
}
```

Abbildung 6: Demonstration eines Cast.

2.5 Laufzeitmodell

Die Ownership-Informationen und die Laufzeit-Typargumente, einschließlich ihrer zugehörigen Ownership-Informationen, werden von GUT explizit gespeichert, weil diese Informationen in der Laufzeitumgebung für Prüfungen von Casts und für Instanziierungen von Typvariablen erforderlich sind. In dieser Hinsicht ist unser Laufzeitmodell ähnlich dem der .NET CLR, wo zur Laufzeit Informationen über Generizität gespeichert wird und "`new` constraints" die Instanziierung von Typvariablen ermöglichen.

Zum Beispiel nimmt Methode `m` in Abb. 6 ein Objekt mit einem beliebigen Owner als Parameter und verwendet einen Cast um Ownership-Informationen sicherzustellen. Um den Cast zu überprüfen speichern wir zur Laufzeit eine Referenz auf seinen Owner und ebenso die Ownership- und Typinformationen für die Typargumente.

Das Speichern der Ownership-Informationen zur Laufzeit ermöglicht uns auch, Instanzen von Typvariablen zu erstellen, wenn die Hauptannotation der entsprechenden oberen Schranke `peer` oder `rep` ist. In unserer Sprache kann jede Klasse mittels eines einheitlichen `new` Ausdrucks instanziiert werden, der alle Felder auf `null` initialisiert. Typevariablen mit `any` als obere Schranke können nicht instanziiert werden, da nicht gewährleistet werden kann, dass die tatsächlichen Typargument konkrete Ownership-Informationen liefern, die notwendig sind, um das neue Objekt richtig in der Ownership-Topologie zu platzieren.

2.6 Eigenschaften

In meiner Doktorarbeit definiere ich eine Programmiersprache und entsprechende Laufzeitumgebung und präsentiere dann das GUT Typsystem das die Ownership-Topologie sicherstellt und darauf aufbauend separate Regeln, um die Owner-as-Modifier-Kapselungsdisziplin sicherzustellen. Die Doktorarbeit formalisiert die Eigenschaften des Systems und beweist ausführlich deren Gültigkeit. Die beiden Haupteigenschaften sind die Typsicherheit des Systems und die Einhaltung der Owner-as-Modifier-Disziplin. Typsicherheit stellt sicher, dass die statischen Ownership-Annotationen korrekt die Ownership-Beziehung zur Laufzeit reflektieren. Die Owner-as-Modifier-Disziplin stellt sicher, dass der Owner eines Objekts Veränderungen kontrollieren kann.

3 Fazit

Das korrekte Verwalten von Objektstrukturen ist eine wichtige Herausforderung in der Programmiersprachenforschung mit vielen praktischen Anwendungen. In dieser Kurzfassung meiner Doktorarbeit habe ich die Grundgedanken von Generic Universe Types (GUT), einem leichtgewichtigen Ownership-System mit Typgenerizität, erklärt. Das GUT System unterscheidet sich von früheren Arbeiten auf drei Arten. Zuerst trennt es die topologische Struktur von der Kapselungsdisziplin. Diese Trennung ermöglicht die separate Entwicklung und Wiederverwendung dieser Teile. Zweitens integriert es Ownership und Typgenerizität, was mir die Untersuchung von Zusammenhängen, wie dem Unterschied zwischen den existentiellen Annotationen `any` und `lost`, erlaubt hat. Drittens erzwingt es optional die Owner-as-Modifier-Disziplin, einer flexiblen Kapselungsdisziplin die die Verifikation von Objektinvarianten ermöglicht. Das GUT System ist streng formalisiert, hat sich als ausgereift erwiesen und ist für verschiedene Programmiersprachen implementiert.

Als zukünftige Arbeit plane ich eine weitere Verbesserung der Aussagekraft und Tool-Unterstützung für GUT. Die Aussagekraft könnte durch die Kombination von GUT mit bestehenden Lösungen für Ownership-Transfer und Objektunveränderlichkeit verbessert werden. Die Tool-Unterstützung könnte verbessert werden durch die Erweiterung unserer Inferenz-Tools für nicht-generische Universe Types auf GUT.

Literatur

[Die09] W. Dietl. *Universe Types: Topology, Encapsulation, Genericity, and Tools*. Dissertation, Department of Computer Science, ETH Zurich, Dezember 2009. Doctoral Thesis ETH No. 18522.

Werner M. Dietl wurde im Dezember 1976 in Salzburg, Österreich geboren. Nach der Ausbildung an der HTBLA Salzburg im Bereich Elektronik und Informatik, begann er 1996 das Informatik-Studium an der Universität Salzburg. In einem Austauschprogramm mit der Bowling Green State University in Ohio, USA, erlangte er im Jahr 2000 einen Master und arbeitete danach für ein Jahr im Silicon Valley bei einer Internet Start-up Firma. Nachdem er das Studium 2003 mit dem Diplom abschloss, entschied er sich für eine wissenschaftliche Laufbahn und begann das Doktorat an der ETH Zürich in der Schweiz. Seine Doktorarbeit konnte er 2009 erfolgreich abschließen. Derzeit ist er, unterstützt durch den Schweizerischen Nationalfonds (SNF), als Post-doctoral Research Associate an der University of Washington in Seattle, USA.