Eine Methode der effizienten und verifizierbaren Programmannotation für den Transport von Escape-Informationen

Dissertation
zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt dem Rat der Fakultät für Mathematik und Informatik
der Friedrich-Schiller-Universität Jena

von Dipl.-Inf. Andreas Hartmann

geboren am 22.11.1974 in Erfurt

Gutachter

1. .....

2. .....

3. .....

Tag der letzten Prüfung des Rigorosiums: .....

Tag der öffentlichen Verteidigung: .....

Widmung

Diese Arbeit ist dem Gedenken der Brüder Ulrich († 24.12.1986) und Hans-Günther Hartmann († 11.01.2006) gewidmet.

Danksagung
Ich möchte mich an dieser Stelle herzlich bei meinem Betreuer Dr. habil. Wolfram Amme bedanken, welcher mir diese Arbeit möglich gemacht hat und mir bei schwierigen Fragen und Problemen immer zur Seite stand. Ebenso bedanke ich mich bei Prof. Jens Knoop für sein intensives Engagement und seine hilfreichen Hinweise.
Weiterhin danke ich Prof. Wilhelm Rossak und dem Lehrstuhl für Softwaretechnik an der FSU Jena für die Unterstützung in den letzten Jahren, so dass ich das notwendige Arbeitsumfeld für die Anfertigung einer Dissertation fand.
Letzlich und insbesondere bedanke ich mich bei meiner Familie für alles, was mich zu diesem Punkt im Leben gebracht hat, sowie bei meinen Freunden, welche immer geeignete Wege der Motivation und die richtigen Worte für mich fanden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Für Java-Programme, die vor ihrer Ausführung auf einem Zielrechner in Maschinensprache übersetzt werden, können Optimierungen, die auf den Ergebnissen einer komplexen Escape-Analyse basieren, effizienter angewendet werden, wenn die entsprechende Analyse bereits vor der Übertragung des Programmes durchgeführt wird und die Ergebnisse im Zwischencode auf eine verifizierbare Art annotiert werden.

Abbildung 1 zeigt ein Quellprogramm in der Programmiersprache Java mit den Klassen `Value` (die Repräsentation eines Wertes), `Calc` (ein Rechner) und `Perform` (eine Klasse zur Durchführung einer Berechnung). Ziel dieses Programmes ist die Auswertung und Ausgabe von Werten aus speziellen Messreihen, wobei offensichtlich Möglichkeiten der Optimierung zur effizienteren Ausführung existieren.

Im Gegensatz zu einer Programmiersprache wie C++, handelt es sich bei Java und der dazugehörigen *Java virtuellen Maschine (JVM)* um ein System zur Programmübertragung von mobilem Code. *Mobiler Code*, der über ein Netz geladen und dann auf ganz unterschiedlichen Systemen heterogener Architektur ausgeführt werden kann, wird heute vor allem in Zusammenhang mit dem Internet eingesetzt. Ein Java-System zur Übertragung von mobilem Code besteht aus einer Produzentenseite (Java-Übersetzer), der so genannten Zwischencoderepräsentation (Java-Bytecode) und einer Konsumentenseite (JVM mit Interpreter oder dynamischem Übersetzer auf der Zielmaschine, letzteres wird auch als Just-In-Time-Compiler oder *JIT-Compiler* bzw. *JIT-Übersetzer* bezeichnet) [12].

Ein Java-Quellprogramm wird auf der Produzentenseite in Java-Bytecode übersetzt und kann dann von einer Konsumentenseite geladen werden. Auf der Konsumentenseite wird das Programm im Zwischencodeformat in die JVM geladen, entsprechend dekodiert und gemäß der Sicherheitsanforderungen des Java-Bytecodeformates überprüft (verifiziert). Nach erfolgreicher Verifikation kann das Programm dann ausgeführt werden, wobei ein optimierender JIT-Übersetzer während der Übersetzung noch wertvolle Optimierungen zur schnelleren Ausführung durchführen kann.

Eine Eigenschaft der JVM ist, dass in einem Programm Objekte nicht explizit durch eine Anweisung aus dem Speicher entfernt werden können (Deallokation). Anstelle dessen übernimmt ein so genannter Speicherbereiniger (*Garbage Collector*) die Entfernung nicht benutzter Objekte aus dem gemeinsamen Speicher. Die Speicherbereinigung ist jedoch insbesondere bei einer hohen Anzahl von Objekten sehr komplex und kostet Ausführungszeit. Weiterhin bedeutet die Erzeugung eines jeden Objektes einen entsprechenden Aufwand, welcher durch die Reduzierung der Objektanzahl eines Programmes vermindert werden kann.

Im zuvor genannten Beispielprogramm werden in der eigentlichen Ausführungsklasse (`Perform`) bzw. deren Methode `perform` eine möglicherweise hohe Anzahl von Objekten des Typs `Value` angelegt. Die Anzahl ist von der Charakteristik der Messreihen abhängig und könnte eine negative Auswirkung auf die Ausführungsgeschwindigkeit der Auswertung haben. Für eine Optimierung des Programmes wäre es also sinnvoll, die Anzahl der zur Berechnung erzeugten Objekte zu reduzieren. Bei einer genauen Betrachtung der Klasse `Calc` wird deutlich, dass lediglich auf die Zahlenwerte der Berechnungsobjekte zugegriffen wird. Würde eine Optimierung die Berechnung der Zahlenwerte in die ursprüngliche Methode `perform` übertragen, so könnte sogar auf die Erzeugung der Berechnungsobjekte vom Typ `Value` und den Rechner verzichtet sowie die Berechnung und Ausgabe auf den

```
class Value{
 int val=0;
 public Value(int a){
   val=a;
 }
}
```

```
class Calc{
 public int calc(Value a, Value b){
   int result=0;
   result = (a.val*a.val) + (b.val*b.val);
   return result;
 }
}
```

```
class Perform{
 public void perform(){
   int[] aa = getMeasurements(“a”);
   int[] bb = getMeasurements(“b”);
   int result = 0;
   Calc c = getCalc();
   for(int i=0; i<aa.length;i++){
     result = c.calc(new Value(aa[i]), new Value(bb[i]));
     System.out.println(“Result[i]: “ + i + “ - "+ result);
   }
 }
}
```

```
class RegCalc extends Calc{
 public int calc(Value a, Value b){
   int result=0;
   //register a,b in global table
   result = (a.val*a.val) + (b.val*b.val);
   return result;
 }
}
```

Abbildung 1: Java-Quellprogramm

Zahlenwerten durchgeführt werden. Die entsprechenden Optimierungen bezeichnet man als *Objektauflösung in skalare Felder* und *Methodeninlining* ([45]). Abbildung 2 zeigt eine in diesem Zusammenhang optimierte Version der Methode `perform`.

```
public void perform(){
  int[] aa = getMeasurements(“a”);
  int[] bb = getMeasurements(“b”);
  int result = 0;
  //Calc c = getCalc();
  for(int i=0; i<aa.length;i++){
    //inline of calc(Value a, Value b)
    result = (aa[i]*aa[i]) + (bb[i]*bb[i]);
    System.out.println(“Result[i]: “ + i + “ - "+ result);
  }
}
```

Abbildung 2: Optimierung von `perform`

Obgleich diese Optimierung sehr vielversprechend scheint, ist sie in einem Java-System nicht ohne Weiteres durchführbar. Grund hierfür ist die Art und Weise, in der die Klassen übersetzt und ausgeführt werden. Zu dem Zeitpunkt, wenn die Klasse `Perform` auf der Produzentenseite übersetzt wird, steht noch nicht endgültig fest, welche Implementierung des Rechners zur Laufzeit tatsächlich verwendet wird. Es existiert eine zweite Implementierung mit dem Namen `RegCalc` und die Verwendung hängt von der Methode `getCalc` ab. Der zweite Rechner unterscheidet sich jedoch hinsichtlich der Berechnung, da hier nicht nur die Werte berechnet werden, sondern noch eine Speicherung der Berechnungsobjekte in einer globalen Tabelle zur weiteren Verwaltung stattfindet. Es kann leicht nachvoll-

zogen werden, dass bei der Verwendung von `RegCalc` zur Laufzeit des Programmes die oben genannten Optimierungen nicht durchführbar sind. Üblicherweise kann auf der Produzentenseite nicht entschieden werden, welcher der beiden Rechner letztlich zum Einsatz kommt, was die Optimierung an dieser Stelle unmöglich macht.

Wenn das Programm auf der Konsumentenseite durch einen optimierenden JIT-Übersetzer ausgeführt wird, ist der verwendete Rechner bekannt und die Optimierung könnte bei Verwendung der Klasse `Calc` durchgeführt werden. Zuvor muss aber festgestellt werden, ob die Berechnungsobjekte nicht an einer anderen Stelle gespeichert bzw. benutzt werden. Die entsprechende Analyse bezeichnet man als *Escape-Analyse*. Die Anwendung der Escape-Analyse auf der Konsumentenseite führt jedoch aufgrund ihrer Komplexität und dem hohen Aufwand zu einer Erhöhung der Übersetzungszeit des JIT-Übersetzers.

Als Problem stellt sich heraus, dass sich eine Analyse und Optimierung auf der Konsumentenseite durch erhöhten Aufwand und längere Übersetzungszeiten negativ auf die Ausführungsgeschwindigkeit auswirkt. Auf der Produzentenseite ist zwar kein Einfluss auf die Laufzeit des Programmes zu befürchten, doch fehlen einige zur Optimierung wichtige und notwendige Informationen. Eine Lösung für dieses Problem kann die Technik der Beifügung (*Annotation*) bieten. Bei einer Annotation werden dem mobilen Code zusätzliche Informationen hinzugefügt, welche auf der Konsumentenseite z.B. durch einen optimierenden JIT-Übersetzer zur Erzeugung von effizienterem Maschinencode verwendet werden können. Im Zusammenhang mit der Escape-Analyse könnten also dem Zwischencode Informationen über optimierbare und nicht optimierbare Objekte hinzugefügt werden. Der JIT-Übersetzer der Konsumentenseite kann dann diese Informationen auswerten, um die Basis für eine entsprechende Optimierung zu schaffen, ohne die komplexe Analyse selbst durchführen zu müssen.

Bisherige Techniken der Annotation haben jedoch den Nachteil, dass die annotierten Informationen nicht verifizierbar sind. Im schlechtesten Fall könnten daher während der Übertragung des Programmes Änderungen im Code auftreten, welche zu einer fehlerhaften Ausführung des Programmes aufgrund der angewendeten Optimierung führen. In dieser Arbeit wird eine sichere und verifizierbare Annotationstechnik auf der Basis von *SafeTSA* vorgestellt. Bei SafeTSA handelt es sich ebenfalls um ein System zur Programmübertragung mittels mobilen Codes, wobei als Programmiersprache Java unterstützt wird. Durch eine Erweiterung des SafeTSA-Systems ist es möglich, Informationen einer auf der Produzentenseite durchgeführten Escape-Analyse im Zwischencode zu annotieren, den mobilen Code sicher und verifizierbar zur Konsumentenseite zu transportieren und die Analyse-Ergebnisse dort einem optimierenden JIT-Übersetzer für SafeTSA-Code zur Verfügung zu stellen.

Die vorliegende Dissertation beschreibt die Erweiterung des SafeTSA-Systems zur Unterstützung der Annotation von Ergebnissen einer auf der Produzentenseite durchgeführten Escape-Analyse und deren Auswertung auf der Konsumentenseite des Systems. Dabei wird auf die Grundlagen von Java und JVM eingegangen (Kapitel 2), verwandte Arbeiten der Escape-Analyse und Annotation vergleichend vorgestellt (Kapitel 3,4) und die Annotation mittels des erweiterten SafeTSA-Systems dargestellt (Kapitel 6). Weiterhin beschäftigt sich die Arbeit mit der Unterstützung Java-spezifischer Eigenschaften (Kapitel 7), beschreibt die Implementierung des erweiterten Systems (Kapitel 8) und wertet abschließend die Ergebnisse aus, welche den gewonnenen Geschwindigkeitsvorteil verdeutlichen (Kapitel 9).

2 Grundlagen

2.1 Java Virtuelle Maschine

Der folgende Abschnitt gibt eine Übersicht über einige Verfahrensweisen und Eigenschaften der Java Virtuellen Maschine (im Folgenden die *JVM* genannt) bzw. der Java Sprachdefinition. Für ein besseres Verständnis der in dieser Arbeit betrachteten Analysetechnik ist es notwendig, auf einige Eigenschaften von Java und der JVM näher einzugehen.

Bei der Entwicklung der Programmiersprache Java hatten die zu dieser Zeit weit verbreiteten Sprachen `C` und `C++` einen bedeutenden Einfluss. Insbesondere bemühte man sich, spezielle Schwächen und Fehlerquellen dieser Sprachen zu vermeiden und ein Mittel zu schaffen, mit dem möglichst sichere Programme geschrieben werden können. Ein Beispiel für häufige Fehlerursachen war der Speicherzugriff in `C`, bei dem über einen Zeiger direkt in einem Speicherbereich Werte geändert werden können. Bei diesem Zugriff werden weder Referenz-, noch Typüberprüfungen durchgeführt und im schlimmsten Fall kann der Zugriff zu einem schweren Fehler sowie einem Programmabsturz führen (z.B. Speicherschutzverletzung unter MS Windows). In Java wurde auf die Verwendung expliziter Zeiger verzichtet, wodurch Objekte nur mittels referenzierender Variablen im Speicher erreichbar und veränderbar sind. Weiterhin werden bei allen Zugriffen implizite Referenz- und Typüberprüfungen durchgeführt, welche mögliche Fehler an das Fehlerbehandlungsmodell (*Ausnahmemodell oder Exceptionmodell*) weiterleiten. Durch diese Einschränkung kann die Kontrolle des verwendeten Speichers vollständig an die JVM übergeben werden, welche je nach Bedarf eines Programmes physischen Speicher für Objekte bereitstellen und wieder freigeben kann. Der verwaltete Speicher wird als Haldenspeicher (*Heap*) bezeichnet.

Bei der Speicherverwaltung benutzt Java das Modell der automatischen Speicherrückgewinnung, welches bereits in funktionalen Programmiersprachen sowie Smalltalk Anwendung fand. Ein Nachteil bei der Verwendung einer automatischen Speicherrückgewinnung ist der damit verbundene Aufwand. Während der Laufzeit eines Programmes werden immer wieder neue Objekte erzeugt, von denen eine hohe Anzahl bereits nach kurzer Zeit nicht mehr benötigt wird. Da der Programmierer in seinem Programm keine explizite Speicherfreigabe definiert, ist der zur Verfügung stehende Speicher nach einer bestimmten Zeit erschöpft. Dann muss die JVM die Aufgabe übernehmen, den Speicher nach nicht mehr verwendeten Objekten zu untersuchen und entsprechend Speicher freizugeben. Dieser sehr zeitaufwendige Vorgang wird als *Garbage-Collection* bezeichnet, wobei die Analyse des Speichers und die Freigabe durch den *Garbage-Collector* durchgeführt wird. Die als *Stack-Allocation* bezeichnete Optimierungstechnik kann auf der Basis von Ergebnissen einer Escape-Analyse die Anzahl der im Heap zu speichernden Objekte verringern und somit den Zeitaufwand des Garbage-Collectors erheblich reduzieren.

Eine weitere Eigenschaft der Programmiersprache Java ist die Unterstützung von nebenläufigen Programmen durch die Verwendung so genannter *Threads*. Hierbei können unterschiedliche Threads erzeugt und gestartet werden, die ihre Aufgaben dann parallel ausführen. An bestimmten Stellen eines Programmes kann oder muss eine Synchronisation erfolgen, wenn z.B. von mehreren Threads auf gemeinsam genutzte Objekte zugegriffen wird. Instanzen der Klasse `java.util.Vector` sind häufig verwendete Objekte, die bei Programmen mit mehreren Threads (Multithreading) synchronisiert werden müssen. Die eigentliche Synchronisation unter Verwendung von *Monitoren* übernimmt ebenfalls die

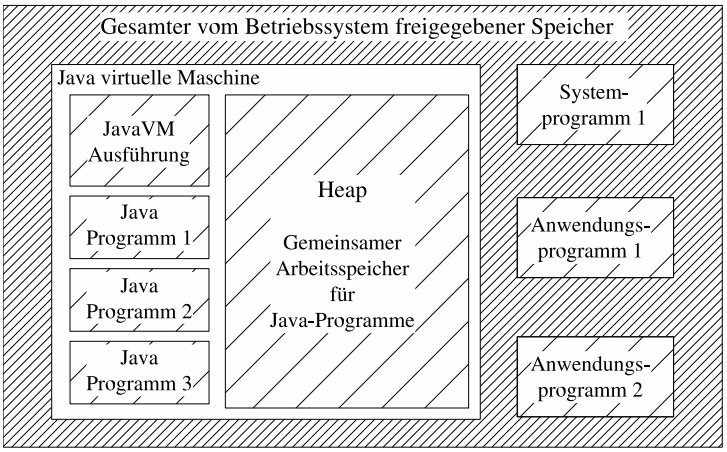


Abbildung 3: Vereinfachte Darstellung des Java-Speichermodells

JVM. Manchmal ergibt es sich, dass zur Laufzeit eines Programmes nur ein einzelner Thread arbeitet oder auf verschiedene Objekte nur innerhalb eines Threads zugegriffen wird. Eine hier durchgeführte Synchronisierung ist offensichtlich überflüssig und könnte vermieden werden. Die als *Synchronization Elimination* bezeichnete Optimierungstechnik entfernt auf der Basis von Ergebnissen einer Escape-Analyse unnötige Synchronisationen und verringert somit den Aufwand für die JVM.

2.1.1 Java Speichermodell

Die JVM stellt Speicher zur Verfügung, der durch alle Threads und Programme gemeinsam genutzt wird. Die Zuteilung von freiem Speicher erfolgt je nach Bedarf eines Programmes unter Berücksichtigung der gesamten Speichergröße. In diesem Speicher (Heap) werden alle benötigten Daten für die Ausführung der Programme gespeichert, unter anderem sind auch alle erzeugten Objekte und Felder enthalten. Die Aufteilung des Heap in spezielle Bereiche, z.B. für verschiedene Programme, die Programmausführung und -verwaltung, obliegt der jeweiligen Implementierung der JVM. Auch die Verwaltung des Heap und der Speicherbereiniger (Garbage Collector) können sich je nach implementierter JVM unterscheiden. Abbildung 3 zeigt vereinfacht eine mögliche Aufteilung des Heap in Sichtweise des gesamten einem Rechnersystem zur Verfügung stehenden Speichers.

Werden zur Laufzeit Objekte angelegt, so wird im Heap entsprechend der benötigten Größe des Objektes Platz angefordert und dieses dort gespeichert. Die Gesamtgröße des Heap kann fix oder dynamisch angelegt sein, wobei je nach Implementierung der JVM dem Benutzer Kontrolle über die maximale und die initiale Größe gegeben werden kann. Bei einem fixen Heap entspricht bereits die initiale Speichergröße dem maximalen zur Verfügung stehenden Speicher, wogegen bei einem dynamisch ausgelegten Heap zu Beginn eines Programmstartes nur der initiale Speicher zur Verfügung steht. Benötigt das Programm mehr Platz, kann die JVM weiteren Speicher bereitstellen, bis die maximale Speichergröße erreicht ist. Überschreitet ein Programm in seinen Anforderungen den maximalen zur Verfügung stehenden Speicher, wird ein `OutOfMemoryError` ausgelöst.

2.1.2 Methodenrahmen

Ein Methodenrahmen findet hauptsächlich Verwendung für das Speichern von lokalen Daten und partiellen Ergebnissen, sowie Rückgabewerten von Methoden. Mit jedem neuen Methodenaufruf wird ein entsprechender Methodenrahmen angelegt, der nach Beendigung des Aufrufes wieder freigegeben werden kann. Dabei ist es gleichgültig, ob die Methode regulär oder irregulär (z.B. abbrechend durch eine nicht abgefangene Ausnahme) schließt. Die Speicherung eines Methodenrahmens erfolgt jeweils in dem Stapel, der dem Thread zugeordnet ist, welcher die Methode aufruft.

Jeder Methodenrahmen verfügt über ein Feld, in welchem alle lokalen Variablen und Parameter gespeichert sind (*local variable array*). Weiterhin ist ein eigener Operandenstapel (*operand stack*) und eine Referenz auf den Konstantenpool (*constant pool*) enthalten. Letzterer ist der Klasse zugeordnet, welche die aktuelle Methode definiert. Der strukturelle Aufbau des Methodenrahmens im Speicher ist der Implementierung der JVM überlassen.

Bei der Ausführung eines Programmes ist zu jedem Zeitpunkt ausschließlich der Methodenrahmen für die auszuführende Methode eines gegebenen Threads aktiv. Dieser Methodenrahmen wird mit *current frame* bezeichnet, die Methode mit *current method* und die dazugehörige Klasse mit *current class*. Ein Methodenrahmen wird inaktiv, wenn die auszuführende Methode eine weitere Methode aufruft oder selbst endet. Mit dem Aufruf einer weiteren Methode wird ein entsprechender neuer Methodenrahmen erzeugt und dieser bekommt den Status *current frame*. Nach Beendigung der aufgerufenen Methode wird der Status wieder an den vorhergehenden Methodenrahmen zurückgegeben.

2.1.3 Stapel

Ein Stapel (*Java virtual machine stack*) wird zu jedem Thread bei dessen Erzeugung erzeugt. Ein Stapel enthält hauptsächlich Informationen zum Erzeugen, Löschen und Verwalten dieser Methodenrahmen und wird üblicherweise ebenfalls im Heap angelegt. Je nach Implementierung der JVM kann der Stapel eine feste oder dynamische Größe besitzen. Bei einer dynamischen Größe kann dem Benutzer die Kontrolle über die initiale und die maximale Größe des Stapels übergeben werden.

Erfordert ein Programm mehr Stapelspeicher, als bereitgestellt werden kann, so treten folgende Ausnahmen auf. Bei Überschreitung der zulässigen Größe für einen anzulegenden Stapel tritt ein `StackOverFlowError` auf. Reicht der gesamte Heap der JVM nicht aus, um einen weiteren Stapel anzulegen oder einen existierenden Stapel zu vergrößern, wird ein `OutOfMemoryError` erzeugt.

Abbildung 4 zeigt schematisch die Zuordnung der Stapel zu einem zugehörigen Thread und der Methodenrahmen zu den entsprechenden Methoden, wobei die Stapel und Methodenrahmen getrennt vom gemeinsam genutzten Speicher für Programmdaten verwaltet werden.

2.1.4 Threads und Sperren

Die JVM unterstützt bei der Ausführung von Programmen die parallele Abarbeitung mehrerer Threads, welche von einander unabhängig auf Werten und Objekten in gemeinsam genutzten Bereichen des Speichers operieren. In der Realität kann eine parallele Ausführung

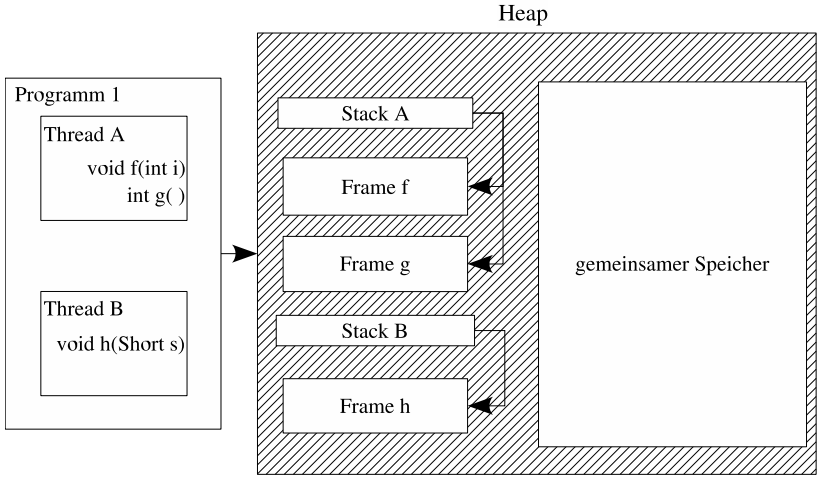


Abbildung 4: Aufteilung des Speichers mit Stapel und Methodenrahmen (schematisch)

von Threads auf mehreren Prozessoren oder eine pseudoparallele Ausführung auf einem einzelnen Prozessor stattfinden.

Die Repräsentation von Threads übernimmt in Java die Klasse `Thread`. Da ein Thread ausschließlich durch die Instanziierung eines Objektes dieser Klasse erzeugt werden kann, wird letztlich jeder Thread durch ein Objekt der Klasse `Thread` dargestellt. Der Start des Threads erfolgt über die `start()`–Methode, welche automatisch die ebenfalls in `Thread` definierte `run()`–Methode aufruft.

Java stellt verschiedene Mechanismen für die Kommunikation zwischen verschiedenen Threads zur Verfügung, wobei die Synchronisation von grundlegener Bedeutung ist. Die Synchronisation zwischen Threads basiert auf der Verwendung von *Monitoren*, wobei ein Monitor ein Objekt repräsentiert, mit dem Threads blockiert und wiederbelebt werden können ([7]). Wenn ein Thread weiterarbeiten möchte, ist unter Umständen ein bestimmter Zustand eines Objektes notwendig. Befindet sich dieses Objekt nicht in dem gewünschten Zustand, wartet der Thread. Erst wenn das Objekt eine Zustandsänderung signalisiert, prüft der Thread, ob er das Objekt verwenden kann und fährt dann mit seiner Arbeit fort. Jedem in Java erzeugten Objekt ist ein Monitor zugeordnet, der durch einen Thread gesperrt und wieder entsperrt werden kann. Jeder Monitor kann nur von einem einzigen Thread zu einer bestimmten Zeit gesperrt werden und jeder andere Thread, der versucht eine Sperre auf diesen Monitor zu erhalten, wird in seiner Ausführung gestoppt, bis er seinerseits die gewünschte Sperre erhält.

Folgende Synchronisationen werden durch die Sprachspezifikation definiert:

> Der *synchronisierte* Anweisungsblock enthält *synchronisierte* Anweisungen, die jeweils eine Referenz auf das zu synchronisierende Objekt ermitteln. Danach wird versucht eine Sperre auf dem mit diesem Objekt verbundenen Monitor durchzuführen, wobei die Anweisung in ihrer Ausführung solange wartet, bis der Monitor erfolgreich gesperrt werden konnte. Mit Erhalt der Sperre wird der Inhalt der Anweisung ausgeführt und nach deren Beendigung der Monitor wieder entsperrt.
>
> Eine *synchronisierte* Methode fordert eine entsprechende Sperre an, wenn sie auf-

Methode:

```
void onlyMe(Foo f) {
   synchronized(f) {
      doSomething();
   }
}
```

Programmcode:

```
Method void onlyMe(Foo)
  0  aload_1          // lade f
  1  astore_2         // speichere f in lokaler Variable 2
  2  aload_2          // lade lokale Variable 2 (f)
  3  monitorenter     // sperre Monitor von f
  4  aload_0          // mit gesperrtem Monitor, lade this
  5  invokevirtual #5 // rufe doSomething() auf
  8  aload_2          // lade lokale Variable 2 (f)
  9  monitorexit      // entsperre Monitor von f
 10  return           // beende normal
 11  aload_2          // Im Falle einer Ausnahme ...
 12  monitorexit      // ...entsperre ebenfalls den Monitor...
 13  athrow           // ...dann gib die Ausnahme weiter
```

Abbildung 5: Synchronisierte Methode und Bytecode

gerufen wird. Die Methode kommt erst mit erfolgreicher Sperre zur Ausführung. Das für die Sperre benötigte Objekt richtet sich nach Art der Methode. Handelt es sich um eine Instanzmethode, so wird die Sperre für den Monitor des Objektes angefordert, auf dem diese Methode aufgerufen wird. Handelt es sich um eine statische Methode, so bezieht sich die Sperre auf das Klassenobjekt (`class`), welches der Klasse mit der Definition der statischen Methode zugeordnet ist. Die Entsperrung des jeweiligen Monitors erfolgt nach Beendigung der Methode, wobei nicht zwischen der Art der Beendigung unterschieden wird (z.B. normal oder durch eine Ausnahme).

Die Verwendung von `monitorenter` und `monitorexit` Die JVM stellt spezielle Instruktionen `monitorenter` und `monitorexit` zur Verfügung, die das Sperren und Entsperren von Objekten (d.h. des Monitors, der mit einem Objekt verbunden ist) auf einer höheren Sprachebene implementieren. Für eine synchronisierte Anweisung würde ein Java-Übersetzer also dem Inhalt der Anweisung eine Sperre, implementiert durch `monitorenter`, voransetzen und dafür Sorge tragen, dass nach Beendigung der Anweisung eine entsprechende Entsperrung, implementiert durch `monitorexit`, erfolgt. Dabei bezieht sich jede Entsperrung (`monitorexit`) auf eine zuvor durchgeführte Sperre (`monitorenter`).

Abbildung 5 zeigt eine synchronisierte Methode und ihre Repräsentation im Zwischencodeformat Java-Bytecode ([4]).

2.2 SSA-Form

Die *Static Single Assignment*-Form (*SSA*-Form, [24], [8]) wurde 1988 von Wegman, Zadeck, Alpern und Rosen als Darstellungsform von Programmen vorgestellt, die heute bei vielen optimierenden Übersetzern Verwendung findet. Eine herausragende Eigenschaft dieser Darstellungsform ist, dass jede in einem Programm verwendete Variable nur ein einziges Mal definiert sein darf. Für bestimmte Programmanalysen bedeutet das die Möglichkeit, eine komplexe und zeitintensive Datenflussanalyse umgehen zu können. Hiermit eignet sich die SSA-Form als Basis für eine Vielzahl von Programmanalysen und -optimierungen, welche wesentlich effizienter durchführbar sind, als unter Verwendung einer nicht der SSA-Form entsprechenden Programmrepräsentation.

Im Folgenden ist der grundlegene Aufbau der SSA-Form dargestellt. Um die SSA-Form eines Programmes zu erhalten, wird bei jeder Zuweisung zu einer Variablen der linke Teil der Zuweisung mit einem eindeutigen und einmaligen Namen versehen. Im weiteren Verlauf der Programmdarstellung werden dann die Verwendungen dieser Variablen, z.B. als Operand einer Instruktion, entsprechend dieses Namens angepasst. Im Allgemeinen werden anstelle von jeweils neuen Variablennamen als Umbenennung Indizes eingeführt. Die so gekennzeichneten einmaligen Variablen werden auch als Werte bezeichnet (*values oder value instances*). Abbildung 6 stellt ein einfaches Programm dar, wobei der linke Teil das originale Quellprogramm in einer Hochsprache zeigt und der rechte Teil die entsprechende SSA-Form präsentiert.

```
i := 0;                 i1:= 0;
x := i + 1;             x1:= i1 + 1;
i := 2;                 i2 := 2;
y := i - 2;             y1:= i2 - 2;
```

Abbildung 6: Beispiel für Repräsentation in SSA-Form

Im Unterschied zu diesem einfachen Beispiel existieren in komplexeren Programmen Verzweigungen und Vereinigungsknoten, so genannte *Join-Nodes*. Das sind Knotenpunkte, an denen sich unterschiedliche Programmverzweigungen wieder in einem einzelnen Ausführungspfad vereinen. In diesen Knotenpunkten kann der Wert einer Variablen durch die verschiedenen Programmzweige definiert werden (eine Zuweisung erfolgen), es gibt jedoch keine exakte Aussage, welcher dieser möglichen Werte in Zukunft verwendet werden soll. Um das Problem zu lösen, wurde eine so genannte *Phi-Funktion* (*ϕ-Funktion*) eingeführt.

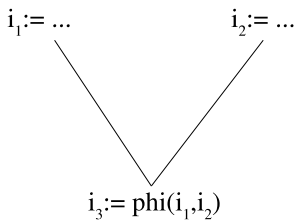


Abbildung 7: Beispiel für die ϕ-Funktion

Eine ϕ-Funktion existiert an jedem Vereinigungsknoten und hat genau so viele Operanden, wie es eingehende Kanten für diesen Knoten gibt. Ihre konzeptionelle Aufgabe ist es, zur Laufzeit des Programmes aus allen Werten den Einen auszuwählen, der in dem aktuell ausgeführten Programmzweig definiert wurde. Die Auswahl ist im Sinne einer Zuweisung realisiert; wird der Knoten durch den i-ten Zweig erreicht, so erhält der linke Teil der Zuweisung den Wert des i-ten Operanden der Funktion. Bei der zugewiesenen Variable handelt es sich wiederum um einen Wert, der nunmehr für den folgenden Programmverlauf eingesetzt werden kann. Abbildung 7 zeigt die Verwendung der ϕ-Funktion beim Zusammentreffen zweier Verzweigungen in einem Vereinigungsknoten. Hier beschreibt $phi\,(i_1, i_2)$ die ϕ-Funktion, welche die Werte der beiden Ausführungspfade zusammenführt.

3 Escape-Analyse

3.1 Einleitung

Die Escape-Analyse beschäftigt sich mit der Ermittlung von Informationen, die für Optimierungen zur Reduzierung der im Speicher anzulegenden Objekte genutzt werden können. Eine Anwendung der Ergebnisse einer Escape-Analyse ist z.B. die Stapel-Allokation (*Stack-Allocation*), bei der ein Objekt nicht im Heap, sondern im zugehörigen Stapel des Methodenrahmens der das Objekt erzeugenden Methode abgelegt wird. Die JVM erlaubt für die Stapel einer Methode eine separate Speicherverwaltung, in der insbesondere der durch den Stapel genutzte Speicher nach Beendigung der Methode ohne zusätzliche Untersuchungen wieder freigegeben werden kann. Ein Objekt kann allerdings nur dann auf einem Methodenstapel gespeichert werden, wenn nach Beendigung der Methode keine Referenz im Programm mehr auf dieses Objekt verweist. Die Escape-Analyse stellt in diesem Zusammenhang fest, ob ein Objekt nur innerhalb einer Methode referenziert wird und somit auf dem entsprechenden Stapel gespeichert werden könnte. Die Bezeichnung "fliehen" (escape) ist aus der Tatsache abgeleitet, dass ein von außerhalb der erzeugenden Methode referenziertes Objekt "aus der Methode fliehend" genannt wird. Im Gegensatz dazu heißt ein nicht fliehendes Objekt "an die Methode gebunden" (bound).

Eine weitere Anwendung findet die Escape-Analyse für Programme mit der Unterstützung multipler Threads. Hier kann generell von mehreren Threads parallel auf ein Objekt zugegriffen werden und es sind spezielle Mechanismen zur Synchronisation notwendig. Kann allerdings vor Programmausführung festgestellt werden, dass ein Objekt nur innerhalb des erzeugenden Threads referenziert wird oder das gesamte Programm nur aus einem einzelnen Thread besteht (*single Thread*), so können entsprechende Synchronisationsstrukturen für dieses Objekt entfernt werden. Die Optimierung wird Entfernung von Synchronisationsmechanismen (*Synchronization Elimination*) genannt und basiert auf den Ergebnissen einer Escape-Analyse, welche das Verhalten von Objekten bezüglich des erzeugenden Threads untersucht. Einen weiteren Vorteil dieser Optimierung kann die Möglichkeit einer vereinfachten Speicherverwaltung für Threads in Bezug auf nicht fliehende Objekte bieten. Werden alle nicht aus einem Thread fliehenden Objekte in einer speziellen Speicherregion (*Region based Heap, Thread specific Heap* - [49], [55]) abgelegt, kann diese nach Beendigung des Threads ohne aufwendige Untersuchungen des Garbage-Collectors wieder freigegeben werden.

Definition – Escape-Analyse

Die Escape-Analyse kann die Eigenschaft 'fliehen' von Objekten hinsichtlich ihrer Erreichbarkeit aus Methoden und Threads wie folgt definieren (nach [23]):

(*Methoden-Fliehen*) Sei O eine Objektinstanz und sei M eine aufgerufene Methode. O *flieht* aus M, wenn die Lebenszeit von O länger sein kann als die Lebenszeit von M. Die Eigenschaft *fliehen* kann als $Escapes\,(O; M)$ notiert werden.

(*Thread-Fliehen*) Sei O eine Objektinstanz und sei T ein Thread. O *flieht* aus T, wenn ein anderer Thread $\acute{T}$ mit $T \neq \acute{T}$ existiert, von dem aus möglicherweise auf O zugegriffen wird. Die Eigenschaft *fliehen* kann hier wieder als $Escapes\,(O; T)$ notiert werden.

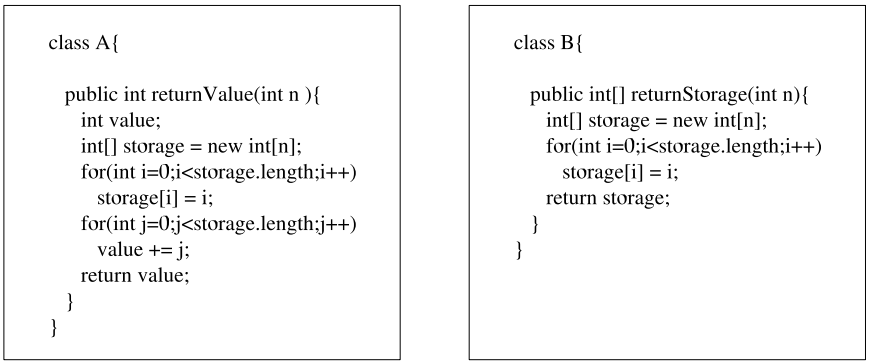


Abbildung 8: Beispiel von Methoden-Fliehen

Beispiel Das Beispiel in Abbildung 8 zeigt zwei Klassen mit einer Methode zur Berechnung von simplen Zahlen in einem Feld. Klasse `A` liefert über die Methode `returnValue` einen Wert in Form des primitiven Integerdatentyps zurück. Das Feld zur Berechnung jedoch 'lebt' nur für den eigentlichen Zeitraum der Ausführung der Methode. Mit der Beendigung von `returnValue` endet auch die Lebensdauer von dem Feld `storage`, da keine weitere Referenz auf dieses Objekt außerhalb von `returnValue` existiert. Anders verhält es sich in Klasse `B`. Dort findet ebenfalls eine primitive Berechnung statt, aber anstelle des Ergebnisses, wird hier das gesamte Feld mit allen Werten zur Berechnung zurückgegeben. Eine weitere Klasse könnte die Methode `returnStorage` aufrufen und den zurückgegebenen Wert nach Belieben abspeichern. In diesem Fall kann die Lebensdauer von dem Feld `storage` die Lebensdauer der erzeugenden Methode `returnStorage` überschreiten, da nach Beendigung der Methode eine Referenz auf das Objekt existieren kann. Das Feld *flieht* in `B` aus der erzeugenden Methode `returnStorage`.

Abbildung 9 zeigt eine Klasse `Collector`, in der über eine Methode `gatherObjects` Objekte gesammelt und gespeichert werden. Zum Sammeln werden verschiedene Threads erzeugt, welche alle parallel Objekte liefern können. Damit der Zugriff auf den Speicher `Storage` geordnet verläuft, wurde die Speichermethode synchronisiert. Das bedeutet, dass immer nur ein Thread zu einer Zeit ein neues Objekt abspeichern kann, andere Threads müssen warten. Auf das Speicherobjekt `storage` wird hierbei also durch mehrere Threads `T` zugegriffen, es *flieht* aus dem erzeugenden Thread. Abbildung 10 zeigt dasselbe Programm unter Verwendung eines einzelnen Threads.

3.2 Vergleich verwandter Arbeiten

Auf dem Forschungsgebiet der Escape-Analyse wurden in den letzten Jahren viele Techniken vorgestellt, die sowohl quantitativ als auch qualitativ hervorragende Ergebnisse liefern. Eine sehr frühe Arbeit wurde von Park und Goldberg [47] entwickelt. Die Autoren führen eine Escape-Analyse für funktionale Programmiersprachen ein, deren Grundlage das Referenzzählen bildet und welche auf Listen operiert. Im Rahmen schnellerer und effizienterer Verfahren entwickelte Steensgaard in [54] eine interprozedurale und datenfluss-unabhängige Zeigeranalyse, deren Komplexität nahezu linear zur Programmgröße ist (*linear time-complexity*). Obwohl das Verfahren weniger genau als andere Analysen arbeitet, zeigt der Autor, dass in vielen Fällen dennoch eine hinreichende Grundlage zur Optimierung ge-

```
class Storage{
  private Vector v;
  public Storage( ){
    v = new Vector();
  }

  public synchronized void store(Object o){
    v.add(o);
  }
}
```

```
class T extends Thread{
  private Storage s;
  public T(Storage s){
    this.s = s;
  }

  public void run( ){
    Object o = SomeSource.getObject();
    s.store(o);
  }
}
```

```
class Collector{
  private Storage s;
  private short sources = 1;
  public Collector(short sources){
    s = new Storage();
    this.sources = sources;
  }

  public void gatherObjects( ){
    for(int i=0;i<sources;i++){
      T t_n = new T(s);
      t_n.start();
    }
  }
}
```

Abbildung 9: Beispiel von Thread-Fliehen

```
class SingleCollector{
  private Storage s;
  public SingleCollector( ){
    s = new Storage();
  }

  public void gatherObjects( ){
    T t = new T(s);
    t.start();
  }
}
```

Abbildung 10: Beispiel mit nur einem Thread

schaffen wird. Genauere Ergebnisse aber auch eine bedeutend höhere Komplexität bietet das Verfahren aus der Arbeit von Whaley [60]. Whaley kombiniert die Zeiger- und Escape-Analyse unter Verwendung eines so genannten *Points-To-Escape-Graphen* und benutzt diese Datenstruktur zur Abbildung von Escape-Informationen. Die Ergebnisse der Analyse werden verwendet, um überflüssige Synchronisationsmechanismen von Objekten zu entfernen, die an einen bestimmten Thread gebunden sind. In einer weiteren Optimierung werden an Methoden gebundene Objekte nicht im Heap, sondern auf dem zugehörigen Methodenstapel angelegt. Threads werden ebenfalls in [52] behandelt, wo Salcianu eine Tech-

nik zur Bestimmung präziser Zeiger- und Escape-Informationen für Objekte beschreibt, auf die durch multiple Threads zugegriffen wird. In [57] stellt Vivien einen Algorithmus für eine inkrementelle Zeiger- und Aliasanalyse vor. Ruf [51] entwickelt eine kontextsensitive Escape-Analyse und wendet diese für die Entfernung von nicht benötigten Synchronisationsmechanismen in der Entwicklungsumgebung *MARMOT* ([26]) an.

Choi [23] wendet die Escape-Analyse für die Allokation von Objekten auf dem Methodenstapel sowie die Reduzierung von Synchronisationsmechanismen an, wobei er so genannte *Connection-Graphs* benutzt. Das Verfahren ist vergleichbar mit anderen Techniken auf dem Gebiet der Aliasanalyse und Points-To-Graphen, jedoch können die abgebildeten Informationen leichter zusammengefasst und bereitgestellt werden, wenn eine Methode an unterschiedlichen Aufrufstellen aufgerufen wird. Durch diese *kontext-sensitive* Analyse müssen die Escape-Informationen nicht für jede Aufrufstelle erneut berechnet werden. Eine weitere Anwendung zeigt [31]. Gay und Steensgaard beschreiben hier einen Ansatz, in dem die Ergebnisse der Escape-Analyse für die Allokation von Objekten auf dem Methodenstapel verwendet werden. Der Algorithmus arbeitet zwar sehr schnell (in linearer Zeit), kann jedoch nur eine Teilmenge aller optimierbaren Objekte identifizieren. Weiterhin macht die Technik konservative Annahmen, was die Speicherung von Objekten in Feldern anderer Objekte betrifft (diese Objekte fliehen generell).

Die bislang genannten Arbeiten können nicht unter Berücksichtigung dynamischer Aspekte der Programmiersprache Java, wie z.B. das dynamische Laden von Klassen, angewendet werden. Eine Technik, die auf diese besonderen Eigenschaften zum Teil eingeht, wird durch Kotzmann und Mössenböck in [41] vorgestellt. Die in der Hotspot JVM integrierte Escape-Analyse und Optimierung kann unter bestimmten Umständen rückgängig gemacht werden, wobei dann anstelle des optimierten Maschinencodes eine interpretative Ausführung der entsprechenden Methode angestoßen wird. Dieser Vorgang, bei dem alle lokalen Variablen auf einem anzulegenden Methodenrahmen kopiert werden, wird als Deoptimierung bezeichnet. Hirzel, Diwan und Hind beschreiben in [35] das generelle Problem einer Zeigeranalyse unter Berücksichtigung dynamischen Klassenladens. Es werden alle Schwierigkeiten einer Anwendung der Analyse von Anderson ([13]) für Java identifiziert, eine Lösung und deren Implementierung in einer JVM dargestellt. Die vorgestellte Technik ist vollständig dynamisch orientiert und effizient in Hinsicht auf stabile sowie langläufige Applikationen.

Im Folgenden werden einige der oben genannten Techniken detaillierter beschrieben und die unterschiedlichen Ansätze zur Bestimmung von Escape-Informationen dargestellt. Obwohl in den Arbeiten verschiedene Definitionen des Begriffes “fliehen” (escape) eingeführt werden, zeigt sich die analoge Verwendung von Analyse-Ergebnissen für die Optimierungen der Stapel-Allokation und der Entfernung unbenötigter Synchronisationsmechanismen. Ausführlicher beschrieben werden die Arbeiten von Kotzmann et al. und Hirzel et al., welche im Unterschied zu anderen Verfahren dynamische Aspekte der Programmiersprache Java berücksichtigt.

3.3 Reference Escape Analysis

Eine frühe Arbeit der Escape-Analyse auf Basis des Referenzzählens (*Reference Counting*) wurde durch die Autoren Park und Goldberg vorgestellt ([47]). Ziel der Technik sind höhere funktionale Programmiersprachen, wie LISP und Scheme, in denen die Speicherver-

waltung einen wichtigen Aspekt darstellt. Typisch für funktionale Sprachen ist, dass alle Programme und deren Prozeduren durch Funktionen repräsentiert werden. Die eigentliche Programmausführung entspricht dann der Auswertung von Ausdrücken, wobei Variablen im klassischen Sinn nicht existieren, sondern nur Parameter von Funktionen. Man unterscheidet Eingabewerte (*Inputparameter*) und Ausgabewerte (*Outputparameter*).

Die Speicherverwaltung erfolgt automatisch, wobei eine Methode der Rückgewinnung von Speicher durch die Technik des Referenzzählens bestimmt wird. Beim Referenzzählen bekommt jeder Speicherblock im gemeinsam genutzten Programmspeicher einen Referenzzähler, der die Anzahl der auf diesen Block verweisenden Programmteile zählt. Fällt der Zähler auf Null, so wird der entsprechende Speicherblock nicht mehr verwendet und kann freigegeben werden. Der Verwaltungsaufwand des Referenzzählens ist jedoch sehr hoch und belastet die Ausführungszeit des Programmes. Ziel der Autoren ist die Identifizierung von Referenzen, deren Lebensdauer nicht die der sie erzeugenden Umgebung übersteigt, wodurch Referenzzähler optimiert werden können.

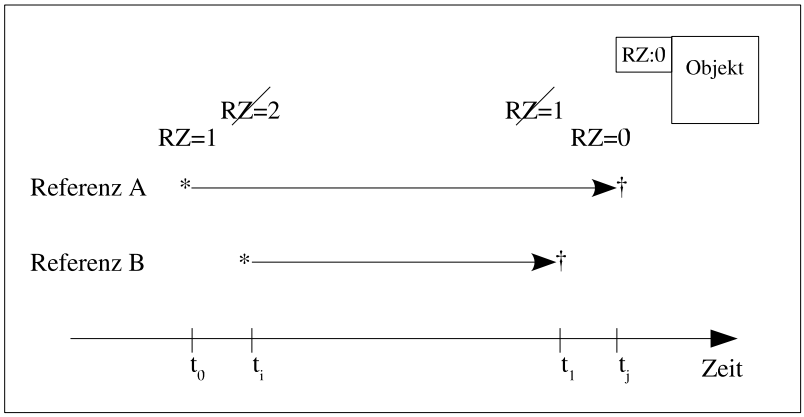


Abbildung 11: Beispiel für Referenzzählen

Abbildung 11 verdeutlicht das Prinzip des Verfahrens. Die Referenzen A und B zeigen auf ein Objekt mit dem Referenzzähler RZ. Die Linien von ($\star$) bis (†) stellen die Lebensdauer der Referenzen dar, wobei zu erkennen ist, dass die Referenz A früher als B erstellt und später zerstört wird. Könnte man zur Übersetzungszeit des Programmes feststellen, dass A zum Zeitpunkt t_1 noch aktiv ist, wäre für die Referenz B kein Referenzzähler notwendig, da RZ auch nach Zerstörung von B noch ungleich Null ist.

Die Autoren untersuchen in der vorgestellten *Referenz-Escape-Analyse* das Verhalten von Referenzen bzgl. deren Lebenszeiten und führen folgende Definition für das *Fliehen (Escape)* von Referenzen ein:

Definition:

"Gegeben ist eine Funktion f mit n formalen Parametern $x_1, ..., x_n$ und m lokalen Variablen $l_1, ..., l_m$. Das j-te Vorkommen des i-ten Parameters x_i (oder der Variablen l_i), bezeichnet als x_{ij} oder l_{ij} innerhalb von f, nennt man:

Global-Referenz-Fliehend aus der Funktionsdefinition von f, falls in einer möglichen Anwendung von f die mit x_{ij} oder l_{ij} verbundene Referenz im Rückgabewert der Funktion enthalten ist.

Lokal-Referenz-Fliehend aus einem Funktionsaufruf von $f\ e_1, ..., e_n$, falls die mit x_{ij} oder l_{ij} verbundene Referenz im Rückgabewert einer bestimmten Funktionsanwendung von $f\ e_1, ..., e_n$ enthalten ist. "

Die Autoren führen weiter eine exakte, jedoch nicht berechenbare und nicht standardisierte Semantik ein, welche vollständig das Escape-Verhalten von Referenzen in einem Programm beschreibt. Zur Bestimmung dieses Verhaltens wird jede Referenz einzeln untersucht, wobei relevante Informationen in so genannten *Referenz-Escape-Paaren* festgehalten werden. Die Paare entstammen der *Referenz-Escape-Semantik-Menge* (*RES-Menge* oder D_r). Das erste Element eines Referenz-Escape-Paares gibt an, ob es relevante Referenzen gibt oder nicht. Das zweite Element bezeichnet das funktionale Verhalten eines Ausdrucks, wenn er von einem anderen Ausdruck verwendet wird. Es handelt sich also um eine Funktion über D_r. Die Autoren definieren weiterhin entsprechende *Referenz-Escape-Semantik-Funktionen*.

Da die vorgestellte Semantik nicht direkt für einen Algorithmus angewendet werden kann, führen die Autoren eine Annäherung ein, in der sie RES-Untermengen für Listenausdrücke abstrahieren und eine Annäherung an die RES-Funktionen einführen. Wird nun eine Referenz hinsichtlich des Fliehens anhand der Semantik geprüft, so wird immer nur ein formaler Parameter getestet (*eine* Referenz, die auf *einen* formalen Parameter zeigt). Kommt ein entsprechender formaler Parameter mehrfach in einer Funktion vor, so muss die Analyse entsprechend oft wiederholt werden. Für den Test wird eine Funktion G_r (Globales-Referenz-Fliehen) eingeführt, die als Parameter die Funktion f, die Vorkommen des Parameters und die Umgebung, in der die Funktion operiert, enthält. Als Ergebnis der Analyse liefert G_r eine 0, falls r aus der Funktion f nicht flieht und 1 sonst. Analog gibt es eine Funktion L_r für die Analyse des Lokalen-Referenz-Fliehens.

3.4 Fast Escape Analysis and Stack Allocation

Die Arbeit [31] von Bjarne Steensgaard und David Gay beschreibt eine Technik der Escape-Analyse zur Übersetzungszeit auf der Produzentenseite, die im Unterschied zu anderen Methoden in linearer Zeit arbeitet. Dabei untersucht die Analyse das Verhalten von Objekten innerhalb von Methoden und prüft, ob diese in Felder anderer Objekte gespeichert oder durch die Methode zurückgegeben werden. Zur Bestimmung des Verhaltens der Objekte wird ein einfaches Typsystem mit Bedingungen abgeleitet, welches dann in linearer Zeit ($O(n)$) gelöst wird. Das Eingabeprogramm wird in einer auf der SSA-Form basierenden Zwischenrepräsentation vorausgesetzt.

Die Autoren führen so genannte *frische Methoden* und *frische Variablen* ein, wobei folgende Definition gilt:

Definition

Eine *frische Methode (Fresh Method)* ist eine Methode, die ein Objekt zurück gibt, das während der Ausführung der Methode angelegt wurde.

Eine *frische Variable (Fresh Variable)* ist eine Variable, deren Definition entweder direkt ein Objekt durch den *new*-Operator anlegt, oder indirekt durch den Aufruf und die Zuweisung des Rückgabewertes einer frischen Methode.

Für jede frische Variable stellt die Analyse dann fest, ob der zugewiesene Wert "irgendwie" flieht. Weiterhin wird für jede Variable ermittelt, ob der einer Variablen zugewiesene Wert durch eine Methode zurückgegeben wird. Hierdurch berücksichtigt die Analyse auch Objekte, die als Argumente an Methoden gegeben und durch diese wieder zurückgegeben werden. Später errechnet die Analyse für jede lokale Variable v mit einem Referenzdatentyp die folgenden booleschen Zustände:

> *Escaped(v)* ist wahr, genau dann wenn die Variable Referenzen hält, die durch eine Zuweisung oder eine Ausnahme (*throw*) fliehen könnten.
>
> *Returned(v)* ist wahr, genau dann wenn die Variable Referenzen hält, die durch die Rückgabe der sie definierenden Methode fliehen.

Weiter wird die Menge der in dem Programm verwendeten Referenztypen um die Elemente *top* und *bottom* erweitert, wodurch eine Halbordnung τ entsteht (die kleinste obere Schranke zweier Elemente von τ ist *top*). Nun wird ein Typ *vfresh(v)* für eine Variable v eingeführt, wobei v eine frische Variable ungleich *top* oder *bottom* ist. Das Element *bottom* bedeutet, dass der Zustand einer Variablen in der Analyse unbekannt ist und *top* heißt, die Variable ist definitiv nicht frisch. Für alle formalen Parameter p ist *vfresh(p)=top*. Der neue Typ *mfresh(m)* steht für Methoden m, die eine frische Variable ungleich *top* oder *bottom* zurückgeben.

Unter diesen Voraussetzungen wird ein System von Bedingungen abgeleitet, dessen Größe sich linear zur Programmgröße verhält und in linearer Zeit gelöst werden kann. Beispielsweise gilt für die Zuweisung einer lokalen Variablen v an ein statisches Feld s die Bedingung $s = v : true \Rightarrow Escaped(v)$.

Am Ende steht für jede Variable v fest, ob das durch v referenzierte Objekt aus der definierenden Methode flieht (*Escape(v)*). Für unbekannte und nicht analysierte Methoden nimmt der Algorithmus an, dass alle formalen Parameter fliehen. Die Ergebnisse der Analyse werden anschließend für eine Optimierung der Stack-Allokation genutzt.

3.5 Dynamic Escape Analysis

Kotzmann und Mössenböck präsentieren in [41] eine Escape-Analyse im Kontext der dynamischen Übersetzung, wobei diese Arbeit erweiterte Umstände wie dynamisches Klassenladen und Deoptimierung berücksichtigt. Das in eine intra- und interprozedurale Analyse aufgeteilte Verfahren wurde in Sun Microsystems' Java Hotspot JVM implementiert und arbeitet mit einer auf der SSA-Form basierenden Zwischencoderepräsentation. Die Ergebnisse der Analyse werden für verschiedene Optimierungen angewendet (Stapel-Allokation, Entfernung von Synchronisationsmechanismen und Ersetzen von skalaren Feldern). Eine Deoptimierung kommt in der HotSpot JVM zur Anwendung, wenn die JVM gezwungen ist, die Ausführung des optimierten Maschinencodes zu unterbrechen und sie die Ausführung an den Interpreter übergibt.

Intraprozedurale Analyse

Die intraprozedurale Analyse und die darauf folgende Optimierung mit dem Ersetzen skalarer Felder wird parallel zu der Erzeugung der auf der SSA-Form basierenden HIR

(*High Level Intermediate Representation*) durchgeführt. Die HIR wird durch das Parsen des Bytecodes und einer darauf folgenden, abstrakten Interpretation gewonnen. Zuerst werden alle Basisblöcke bestimmt und deren Grenzen ermittelt. Im weiteren Verlauf werden die Blöcke mit entsprechenden HIR-Instruktionen gefüllt, wobei die Operanden als Zeiger zu vorangehenden Instruktionen dargestellt werden.

Kotzmann und Mössenböck führen für ihre Analyse einen so genannten *Escape-Status* ein, der je nach Eigenschaft des untersuchten Objektes in folgende Stufen klassifiziert wird:

- *NoEscape* Auf das Objekt kann nur innerhalb der erzeugenden Methode zugegriffen werden. Das Objekt wird auch als *method-local* bezeichnet.
- *MethodeEscape* Das Objekt flieht aus der aktuellen Methode, aber nicht aus dem aktuellen Thread, indem es z.B. an einen Methodenaufruf als Argument weitergegeben wird, deren Parameter nicht fliehen. Objekte dieser Klasse werden als *thread-local* bezeichnet.
- *GlobalEscape* Das Objekt flieht global, d.h. es wird außerhalb der aktuellen Methode und des aktuellen Thread referenziert.

Da in der verwendeten HIR keine entsprechenden Anweisungen zum Laden und Speichern von lokalen Variablen einer Klasse existieren, verwaltet der Übersetzer während der Analyse diese Informationen in einem zusätzlichen Objekt (*state object*). Dieses enthält ein Array, in welchem die einer lokalen Variablen zugewiesenen Werte abgespeichert werden (*locals array*). Beim Ersetzen der skalaren Felder wird das Objekt um ein in der Größe veränderbares Array erweitert, das die aktuellen Werte aller Felder enthält, auf die durch eine Methode zugegriffen wurde (*fields array*). Nach der Erzeugung der HIR können dann entsprechend der Optimierung die Felder von nicht fliehenden Objekten durch die gespeicherten Werte substituiert werden.

Analyse der Basisblöcke

In der HIR wird ein Objekt durch die das Objekt erzeugende Instruktion repräsentiert. Dieser Instruktion wird ein Feld hinzugefügt, in dem der entsprechende Escape-Status des Objektes abgespeichert werden kann. Für jedes Objekt ist das Feld mit *NoEscape* initialisiert und wird im weiteren Verlauf entsprechend der Analyse aktualisiert, wobei die Übergänge nur von *NoEscape* in Richtung *GlobalEscape* definiert sind.

Der Escape-Status eines Objektes kann durch die folgenden Instruktionen beeinflusst werden:

- `p = new T()`: `p` startet mit dem Status *NoEscape*, außer die Klasse `T` definiert einen Finalizer, welcher vor dem Garbage Collector ausgeführt wird. In diesem Fall und wenn die Klasse `T` noch nicht geladen wurde, bekommt `p` den Status *GlobalEscape*.
- `T.sf = p`: Jedes Objekt `p`, das in einem statischen Feld gespeichert wird, bekommt den Status *GlobalEscape*.
- `q.f = p`: Wird `p` in einer Instanzvariablen von `q` gespeichert, erbt `p` den Escape-Status von `q` unter der Voraussetzung, dass `q` eine höhere Klassifizierung als `p` selbst besitzt. Weiterhin wird `p` in die Liste der durch `q` referenzierten Objekte eingefügt,

um `p` bei einem späteren Fliehen von `q` zu aktualisieren. In *fields array* wird ein zusätzlicher Eintrag für `q.f` erzeugt und entsprechend `p` abgespeichert.

- `x = p.f`: Für das Laden des Feldes `p.f` wird eine entsprechende HIR-Instruktion erzeugt. Der Übersetzer ermittelt den aktuellen Wert `v` von `p.f` aus *fields array* und speichert ihn für die eingefügte Ladeinstruktion. Wird `p` optimiert, so kann `p.f` durch `v` ersetzt werden.

- `p == q`: Werden zwei Objekte miteinander oder mit *null* verglichen, so müssen sie zumindest auf dem Stack existieren. Der Escape Status beider Objekte wird auf *MethodEscape* erhöht.

- `(T)p`: Ein Cast auf `p` kann fehlschlagen und eine Exception auslösen. Deshalb bekommt `p` den Status *MethodEscape*.

- `p.foo(q)`: Innerhalb der intraprozeduralen Analyse ist eine Aussage zum Verhalten der Argumente einer Methode nicht möglich. Aus diesem Grund bekommen alle aktuellen Argumente den Status *GlobalEscape*. Der Empfänger der Methode wird dabei wie ein Parameter behandelt.

- `return p, throw p`: Rückgabeanweisungen und Ausnahmeobjekte werden grundsätzlich als *GlobalEscape* gekennzeichnet.

Analyse des Kontrollflusses

Während der Erzeugung der HIR müssen entsprechend der SSA-Form spezielle ϕ-Instruktionen für alle lokalen Variablen und Felder eingefügt werden. Der Escape-Status für die Operanden einer ϕ-Instruktionen hängt von allen anderen Operanden ab. In diesem Zusammenhang wird ein so genannter *equi-escape-set* (EES) eingeführt, dessen Elemente die Operanden einer ϕ-Instruktion und die Anweisung selbst bilden. Alle Elemente einer solchen Menge bekommen den gleichen Escape-Status, der dem maximalen Status der Elemente entspricht.

Interprozedurale Analyse

In der interprozeduralen Analyse wird die Escape-Analyse auf jene Objekte ausgeweitet, die als Argumente an Methoden übergeben werden. Zu diesem Zweck werden beim Übersetzen einer Methode neben den lokalen Objekten auch Escape-Informationen zu den formalen Parametern ermittelt. Der Escape-Status aller formalen Parameter wird im Deskriptor der Methode gespeichert und kann dann während der interprozeduralen Analyse, also wenn die Methode aufgerufen wird, herangezogen werden. Hierfür werden die entsprechenden Informationen nach Erzeugung der HIR in zwei Bit-Vektoren kodiert, wobei der erste Vektor die Menge der *method-local*-Objekte und der zweite Vektor die Menge der *thread-local*-Objekte enthält. Bei dem in Abbildung 12 dargestellten Beispiel flieht das erste Argument an eine statische Variable, wogegen `a1` mit *null* verglichen wird und somit den Status *MethodEscape* erhält. Das Argument `a2` wird nicht verwendet und flieht nicht aus der Methode. Der Escape-Status der Parameter wird durch die Vektoren `001` und `011` entsprechend dem Status *method-local* und *thread-local* kodiert.

```
static boolean foo(Obj a0, Obj a1, Obj a2){
  sf = a0;
  return (a1 != null);
}
```

Abbildung 12: Beispiel zur Analyse von Argumenten

Da in der Java HotSpot JVM eine Methode vor dem Übersetzen mehrmals interpretiert wird, kann die Analyse auf einen Methodenaufruf treffen, dessen Methode noch nicht geladen ist. In diesem Fall wird eine schnelle und konservative Escape-Analyse der formalen Parameter der Methode durchgeführt, um eine Aussage über das Verhalten der Argumente zu erhalten. Die Ergebnisse dieser Analyse werden entsprechend aktualisiert, wenn die Methode durch den Übersetzer übersetzt wird. Trifft die Analyse auf einen rekursiven Methodenaufruf, werden Argumente und Parameter als fliehend angenommen (*GlobalEscape*).

Unterstützung zur Laufzeit

In dem vorgestellten Verfahren werden Methoden nur dann optimiert, wenn sich die auszuführende Methode eines Methodenaufrufes statisch bestimmen lässt, wobei Polymorphismen und dynamisches Binden berücksichtigt werden. Zum Auffinden einer passenden Methode wird an dieser Stelle eine Analyse der Klassenhierarchie (entsprechend [25]) durchgeführt. Stellt sich zur Laufzeit heraus, dass die falsche Methode optimiert wurde, indem z.B. eine passendere Methode durch das Nachladen einer Klasse gefunden wird, so ist der Maschinencode der optimierten Methode ungültig und die Ausführung wird an den Interpreter übergeben. Damit der Interpreter auf alle Werte lokaler Variablen zugreifen kann, muss ein entsprechener Methodenrahmen angelegt werden. Dieser Prozess wird als Deoptimierung bezeichnet ([36]), wobei auch auf die durch den Übersetzer erzeugten Debugging-Informationen zugegriffen wird.

3.6 Pointer Analysis in the Presence of Dynamic Class Loading

Hirzel, Diwan und Hind präsentieren in [35] eine Technik zur Zeigeranalyse, die sich mit Problemen bei Verwendung der Programmiersprache Java, wie z.B. dynamisches Klassenladen, befasst und eine Lösung anbietet. Die Autoren transformieren hierbei die Analyse von Anderson ([13]) für die Verwendung auf der Konsumentenseite und ermitteln alle Konflikte mit bestehenden Java-Eigenschaften. Die entsprechende Lösung wird anhand einer Implementierung in einer JVM dargestellt und validiert.

Konflikte bei Verwendung von Java

Die Technik von Hirzel et al. beleuchtet insbesondere das dynamische Laden von Klassen in Java[1] und stellt eine Reihe von Problemen vor, die eine statisch auf der Produzentenseite durchgeführte Analyse betreffen:

[1] Kapitel 7.2 stellt den Prozess des Klassenladens in Java ausführlich vor.

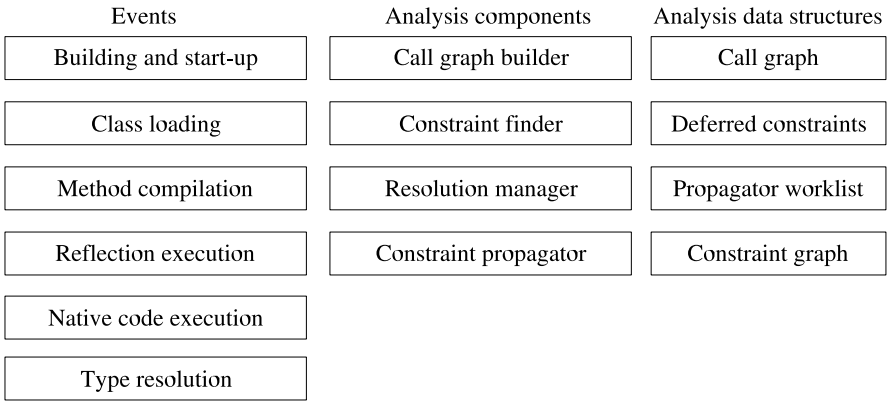


Abbildung 13: Übersicht der modifizierten Zeigeranalyse

1. Es ist nicht bekannt, von *wo* eine Klasse geladen wird. Somit kann eine statische Analyse nicht alle in einem Programm verwendeten Klassen untersuchen, sondern nur das, was bekannt ist.

2. Es ist nicht bekannt, *welche* Klasse geladen wird. Der Name einer zu ladenden Klasse kann eventuell erst zur Laufzeit bestimmt werden. Hierdurch können einige Klassen nicht auf der Produzentenseite untersucht werden, z.B. wenn sie noch nicht verfügbar sind. Entsprechende konservative Annahmen reduzieren jedoch die Qualität der Analyseergebnisse.

3. Es ist nicht bekannt, *wann* eine Klasse geladen wird. Selbst wenn eine Analyse auf der Produzentenseite ausgeführt wird, stehen nicht zwingend bei Start der JVM alle Klassen zur Verfügung. Durch spätes Laden können Klassen auch zu einem späteren Zeitpunkt geladen werden.

4. Es ist nicht bekannt, *ob* eine Klasse überhaupt geladen wird. Hängt die Verwendung einer Klasse von äußeren Umständen ab, so wird sie eventuell nicht geladen und muss in diesem Fall nicht untersucht werden. Eine statische Analyse nimmt an, dass jede Klasse geladen wird und hat somit einen höheren Analyseaufwand. Weiterhin kann eine nicht geladene Klasse die Ergebnisse der Analyse beeinflussen, obwohl sie zur Laufzeit überhaupt nicht ausgeführt wird.

Neben diesen Punkten untersuchen die Autoren die Verwendung der *Reflektion* und der Java nativen Schnittstelle (*JNI - Java Native Interface*), mittels derer Teilprogramme in unbekannten Maschinencode ausgeführt werden können.

Analyse

Die Analyse von Anderson bezieht sich auf eine Verwendung auf der Produzentenseite. Zur Behandlung der oben genannten Probleme wurde diese Technik, welche aus einem Schritt zum Bestimmen eines Bedingungsmodells und dessen Lösung durch Fixpunktbestimmung besteht, für die Konsumentenseite angepasst. Hierbei erweitern Hirzel et al. die Analy-

se um das dynamische Aufbauen des Aufrufgraphen[2] für Methoden zur Laufzeit, einen Mechanismus zur Aktualisierung des Bedingungsmodells bei bestimmten Ereignissen, eine Unterstützung für nicht aufgelöste Typen (*unresolved Types*) und die Berücksichtigung zusätzlicher Ereignisse, wie z.B. die Reflektion.

Abbildung 13 stellt die Erweiterungen und den Aufbau der Zeigeranalyse in einer Übersicht dar.

3.7 Compositional Pointer and Escape Analysis for Java

Whaley und Rinard entwickeln in [61] eine Escape-Analyse für Java, in der die Escape-Eigenschaft von lokalen Objekten anhand eines Graphen abgebildet wird. Eine besondere Eigenschaft dieser Technik ist der inkrementelle Aufbau von Analyse-Informationen in Abhängigkeit untersuchter Programmteile. Der Algorithmus ermittelt seine Daten zunächst auf der Basis der zur Verfügung stehenden Programmteile und bildet diese in einem so genannten *Points-To-Escape-Graphen* ab. Der Points-To-Escape-Graph kann nun separat für verschiedene Teile eines Programmes ermittelt werden. Zur Qualitätssteigerung führen die Autoren die ermittelten Graphen in einem speziellen Schritt zusammen. Der Algorithmus arbeitet interprozedural und stellt die Escape-Eigenschaft formaler Parameter einmal untersuchter Methoden auf eine einfache Art zur Verfügung, so dass diese an den entsprechenden Aufrufstellen im Programm schnell bereit steht.

Die Knoten des Graphen repräsentieren die untersuchten Objekte und die knotenverbindenden Kanten die Abhängigkeiten untereinander. Der Points-To-Escape-Graph wird für jede untersuchte Methode erstellt und beinhaltet sowohl Informationen über das Fliehen von Objekten aus der definierenden Methode, als auch über das Fliehen aus dem erzeugenden Thread. Die Qualität der Analyseergebnisse nimmt hierbei mit der Untersuchung jeder weiteren im Programm aufgerufenen Methode durch die Zusammenführung der Graphen sukzessive zu, aber auch ohne das Betrachten aller vorkommenden Methoden liefert die Technik verwertbare Ergebnisse.

Die vorgestellte Analyse der Autoren arbeitet sowohl intra- als auch interprozedural und lässt sich sehr gut in entsprechende Teilanalysen trennen, wodurch sich der Algorithmus für eine Anwendung der in dieser Dissertation dargestellten Technik der Annotation von Escape-Informationen eignet. Aus diesem Grund und hinsichtlich ihrer ausgezeichneten Analysequalität wurde die Escape-Analyse von Whaley und Rinard für die in Kapitel 8 beschriebene Referenzimplementierung verwendet. Eine ausführliche Beschreibung des Algorithmus sowie der Anpassungen für SafeTSA kann dem Anhang entnommen werden.

[2] Der Aufrufgraph wird für die interprozedurale Analyse verwendet.

4 Annotationstechniken

4.1 Einleitung

Unter Programmannotation versteht man das Beifügen von zusätzlichen Informationen (Metadaten) zur Programmrepräsentation, welche die Semantik des Programmes jedoch nicht beeinflussen ([6], [2]). Die Informationen können aber durch entsprechende Unterstützung in einem Hilfsprogramm verwendet werden, um entsprechende Optimierungen oder Anpassungen des Programmes auszuführen. Bei der Annotation von Ergebnissen einer Analyse werden diese Informationen als Basis von Optimierungen benutzt, welche z.B. durch einen JIT-Übersetzer durchgeführt werden können, um die Ausführungsgeschwindigkeit bzw. die Effizienz des Programmes zu erhöhen.

4.2 Vergleich verwandter Arbeiten

Die Techniken zur Programmannotation wurden in der vergangenen Zeit nicht so umfangreich untersucht, wie es für die Escape-Analyse der Fall ist. Wissenschaftliche Publikationen, die sich mit entsprechenden Verfahren auseinandersetzen, z.B. in Zusammenhang mit JIT-Übersetzern und der Verwendung von Java-Bytecode oder anderen Zwischencoderepräsentationen, sind [16], [44], [48]. Krintz stellt in [42] ein Verfahren für die Annotation von Java-Bytecode vor, um die Ausführungsgeschwindigkeit des Programmes in der JVM zu erhöhen. Franz [28] annotiert die Syntaxbäume von Programmen mit Escape-Informationen auf der Produzentenseite und kodiert diese zur sicheren Übertragung. Beide Techniken garantieren die Sicherheit ihrer Annotationen gegenüber Manipulation oder Übertragungsfehlern. Ebenfalls eine sichere und verifizierbare Annotation von Escape-Informationen stellt Beers [17] vor. Beers führt eine Escape-Analyse auf der Produzentenseite durch und annotiert deren Ergebnisse in der Zwischencoderepräsentation. Auf der Konsumentenseite werden die annotierten Ergebnisse durch die Überprüfung entsprechender Bedingungen verifiziert, wobei davon ausgegangen wird, dass fliehende Objekte nur in bestimmten Anweisungen bzw. Zuweisungen auftreten dürfen.

Ein weiteres Verfahren beschreibt Hannan in [33], wobei die Grundlage ein funktionaler Ansatz zur Annotation von Escape-Informations-Typen bildet. Hannan schließt allerdings dynamisches Laden von Klassen aus und gibt keinen Algorithmus zur Berechnung der benötigten Typen. Eine andere Arbeit umfasst die passive Annotation von möglichen Feldzugriffsoptimierungen. Yessick und Jones geben in [62] einen Weg zur Annotation von Schleifen an, in denen auf Felder zugegriffen wird. Es werden jene Schleifen identifiziert, für die eine Optimierung ausgeschlossen ist, so dass eine auf der Konsumentenseite durchgeführte Analyse weniger komplex gestaltet werden kann. Hummel [38] präsentiert ein Verfahren zur Annotation von Register-Informationen zur späteren Optimierung der Registerzugriffe (Register-Allokation). Zuvor stellt er fest, dass das Java-Bytecodeformat für die Extraktion einiger für Optimierungen benötigter Informationen ungeeignet ist und eine entsprechende Annotation diesen Mangel ausgleichen kann. Jones schließlich zeigt in [40] ebenfalls eine Annotation für Registerzugriffe in Java-Bytecode unter der Verwendung von so genannten virtuellen Registern, wobei diese Technik, ebenso wie die zuvor genannte Annotation, keine Möglichkeit der Verifikation einschließt.

Im Folgenden wird auf eine Auswahl dieser Verfahren ausführlicher eingegangen und der jeweilige Ansatz der Annotation dargestellt.

4.3 Removal of Bounds Checks in an Annotation Aware JVM

Die Arbeit von Yessick und Jones [62] befasst sich mit der Entfernung unnötiger Überprüfungen von Feldern (*Array Bounds Checks*) innerhalb von Schleifen. Die Überprüfung der Feldgrenzen wird durch Java implizit vorgeschrieben, obwohl an einigen Stellen eines Programmes keine Überschreitung der Grenze auftreten kann. Dieser Fall trifft insbesondere auf Schleifen zu, deren Schleifenvariable für den Zugriff auf das Feld verwendet wird (siehe Abbildung 14). Üblicherweise läuft in diesen Fällen die Schleife von Beginn des Feldes bis zum letztmöglichen Index. Eine Grenzüberschreitung ist also unter diesen Umständen auszuschließen.

Um alle Schleifen auf das entsprechende Verhalten zu überprüfen, ist jedoch eine relativ aufwendige Analyse erforderlich, die die Übersetzungszeit des verwendeten JIT-Übersetzers beeinflusst. Die Autoren befassen sich mit der Möglichkeit, Informationen über die Optimierbarkeit von Feldzugriffen im Bytecode zu annotieren. Implizite Instruktionen zur Feldüberprüfung werden jedoch nicht verändert.

```
void foo( ){
        int[] A = new int[ N ];
            int K;
            for (K=0; K < A.length; K++)
                    A[ K ] = K;
}
```

Abbildung 14: Beispiel einer zu optimierenden Schleife

Für die Realisierung der Annotation verwenden die Autoren ein zuvor entwickeltes Framework (siehe *Sable*[3]), das über einen für Annotationen erweiterten JIT-Übersetzer (*Annotation Aware JIT*) verfügt. Die Idee von Yessik ist es, eine bestimmte Untermenge aller Schleifen im gegebenen Programm zu untersuchen, da eine vollständige Analyse auch zur Übersetzungszeit zu komplex erscheint. Für diese Schleifenuntermenge ermittelt eine schnelle Analyse, ob sich Felder mit Feldzugriffen innerhalb der Schleifen befinden und der zugegriffene Index die Feldlänge nicht unter- oder überschreitet. Folgende Schleifen werden untersucht:

- Die Schleifenvariable wird direkt vor der Schleife initialisiert.
- Die Veränderung der Schleifenvariable wird durch ein Inkrement mit einer positiven konstanten Zahl beschrieben und findet vor der Überprüfung der Abbruchbedingung statt.
- Es handelt sich bei der Schleifenvariable um eine für die zugehörige Methode lokale Variable.
- Das zugegriffene Feld wird während der Schleife nicht verändert (im Sinne seiner Dimension und Begrenzung).

[3] [48], [30], [56].

Sind alle Schleifen analysiert, wird dem Bytecode eine Information über die Optimierbarkeit innerhalb dieser Schleife hinzugefügt. Die Annotation beschränkt sich hierbei auf eine Größe von 2 Byte je Schleife. Der JIT-Übersetzer kann nun die durch die Annotation gegebene Information nutzen, um die in den gekennzeichneten Schleifen vorhandenen Felder zu analysieren und entsprechende Routinen für die Grenzüberprüfung zu entfernen. Der Mehrwert des Verfahrens liegt in dem geringeren Aufwand des JIT-Übersetzers, der auf diese Weise nicht alle Schleifen zu überprüfen braucht, sondern lediglich relevante Kandidaten. Yessick gibt an, dass die Anzahl der zu analysierenden Schleifen um 90% reduziert wird.

Das Verfahren der Autoren bietet jedoch keine wirkliche Sicherheit und Verifikation einer Annotation. Diese wird lediglich auf eine Menge von Informationen eingeschränkt, deren Änderung die Semantik eines Programmes nicht verändern kann. Im schlechtesten Fall könnte eine böswillige Manipulation dazu führen, dass der JIT-Übersetzer durch die Analyse einer hohen Anzahl von Schleifen einen erhöhten Aufwand zu bewältigen hat. Eine tatsächliche Verschiebung von Analyseaufwand von der Laufzeit zur Übersetzungszeit findet jedoch nicht statt. Sinnvoll wäre hier eine sichere und verifizierbare Annotation von Feldzugriffen, deren implizite und nicht benötigte Indexüberprüfungen vor Ausführung des Programmes entfernt werden können. Die entsprechende Analyse könnte während der Übersetzung des Programmes auf der Produzentenseite durchgeführt werden.

4.4 Annotating Java Bytecodes in Support of Optimization

Hummel, Azevedo, Kolson und Nicolau präsentieren in [38] eine Arbeit zur Annotation von Java-Bytecode. Um Programme in Java signifikant zu beschleunigen, so stellen die Autoren fest, kommen derzeit drei verschiedene Ansätze verschiedener Optimierungsverfahren zur Anwendung:

- *On-the-fly*-Übersetzung zu Maschinencode (*JIT - Just in Time Compilation*)
- *Ahead-of-time*-Übersetzung von Java-Bytecode in eine höhere Zwischencoderepräsentation, welche dann in Maschinencode übersetzt wird
- Übersetzung von Java-Bytecode in eine höhere Sprache, wobei dann ein existierender Übersetzer zur Erzeugung von Maschinencode verwendet wird.

Bei allen Herangehensweisen wird auf der Basis von Java-Bytecode gearbeitet. Die Schwäche liegt demnach in der sehr aufwendigen Extraktion von Analyseinformationen aus dem Java-Bytecode, sowie weiteren gänzlich fehlenden Informationen, welche im Quellprogramm vorhanden waren. Insbesondere weisen die Ansätze die folgenden Schwächen auf:

1. *on-the-fly*: Der Nachteil von JIT-Übersetzern liegt darin, dass sie aufgrund des Zeitpunktes ihrer Arbeit, nämlich zur Laufzeit des Programmes, nur einen erheblich geringen Zeitanteil für die Optimierung des Programmcodes aufwenden können. Aus diesem Grund sind Optimierungen durch JIT-Übersetzer meist auf lokale Blöcke beschränkt und weisen sehr einfache Analysetechniken auf. Ausgeschlossen sind somit vor allem auch gewinnbringende Datenflussanalysen und Optimierungen, welche über den Bereich einer Methode hinausgehen.

2. *ahead-of-time*: Der bei einem ahead-of-time Übersetzer als Sprache betrachtete Bytecode wird in eine Zwischencoderepräsentation übersetzt, wobei diverse Analysen und Optimierungen durchgeführt werden. Der gesamte Prozess geschieht jedoch vor Ausführungszeit des Programmes, insbesondere bevor das Programm auf dem Zielrechner geladen wurde. Obwohl bei dieser Methode offensichtlich Zeit für Optimierungen gewonnen wird, welche nicht zu Lasten der Laufzeit geht, weist dieser Ansatz ebenfalls erhebliche Nachteile auf. Zum einen geht die direkte Portabilität verloren, zum Anderen müssen Analyseinformationen erneut aus der Zwischencoderepräsentation berechnet werden. Diese Berechnungen sind aufwendiger, als wenn die Analyse direkt auf dem Quellprogramm durchgeführt würde.

3. Ein weiterer Ansatz, das Verwenden einer alternativen Zwischencoderepräsentation[4] mit mehr Informationskapazität, hat den Nachteil, dass diese Programme nicht auf einer herkömmlichen JVM lauffähig sind.

Hummel stellt eine Technik der Annotation von Bytecode vor, mittels derer wertvolle Analyseinformationen im Java-Bytecode festgehalten werden und zusammen mit diesem verbreitet werden können. Auf dieser Basis können Konsumentenseiten, z.B. ein JIT-Übersetzer, von den annotierten Informationen profitieren. Die Kompatibilität bleibt von der Annotation unberührt, das Programm kann also auch durch eine herkömmliche JVM ohne die Möglichkeit der Auswertung von annotierten Informationen ausgeführt werden. Wie auch bei anderen Techniken der Annotation von Zwischencode gehen die gewonnenen Vorteile zu Lasten der Größe der zu übertragenden Programme.

Ansatz

Die Autoren stellen als Schwäche von Java-Bytecode heraus, dass dieser nicht auf die Fähigkeiten moderner Prozessoren abgestimmt ist. So entspricht das Stackmodell nicht Optimierungen, welche stark auf die Arbeit mit Registern, Cache und *Instruction Scheduling* abzielen. In erster Linie bietet die JVM keine Register für die Nutzung von Operanden, sondern setzt dagegen die Existenz eines entsprechenden Stacks voraus. Weiterhin sequenzialisiert das Stackmodell Berechnungsfolgen und verhindert das Wiederverwenden von Werten, da Operanden immer auf die Spitze des Stacks gelegt werden müssen. Als Folge, so argumentieren die Autoren, sind 40% aller in Java-Bytecode vorkommenden Instruktionen Lade- und Speicheranweisungen zu und vom Stack ([59]). Hummel folgert, dass die im Nachfolgenden genannten Optimierungen, aufgrund des Fehlens entsprechender Ausdrucksmöglichkeiten im Java-Bytecode, nicht durchgeführt werden können:

- *Umordnung von Instruktionen*: Da nur auf die Spitze des Stacks zugegriffen werden kann, wird eine Umordnung verhindert.

- *Entfernung von Laufzeitüberprüfungen*: In Java-Bytecode sind alle Überprüfungen implizit.

- *Arithmetische Komplexitätsverringerung*: Da in Java-Bytecode z.B. keine Shiftoperation definiert ist, können Multiplikationen nicht optimiert werden.

[4] ANDF[43], Slim binaries[27].

```
public void foo(int a[], int b[], int i)
{
  a[i] = (2 * a[i] + b[i]);
}
```

Abbildung 15: Java-Methode als Beispiel für zusätzlichen Aufwand

BYTECODE	COMMENT	LOADS	STORES	CHECKS
aload a	push ref to a	1	1	
iload i	push i	1	1	
iconst 2	push const 2		1	
aload a	push ref to a	1	1	
iload i	push i	1	1	
iaload	pop, pop, push a[i]	3	1	2
imul	pop, pop, push 2*a[i]	2	1	
aload b	push ref to b	1	1	
iload i	push i	1	1	
iaload	pop, pop, push b[i]	3	1	2
iadd	pop, pop, push (2*a[i])+b[i]	2	1	
iastore	pop, pop, pop, store into a[i]	3	1	2

Abbildung 16: Bytecode zur Methode `foo` in Bild 15

- *Automatische Speicherfreigabe*: Java-Bytecode stellt keine explizite Operation zur Freigabe von Speicher zur Verfügung.

Abbildung 15 zeigt ein Beispiel, in dem zwei Feldelemente geladen, eine Berechnung durchgeführt und das Ergebnis gespeichert werden. Im zugehörigen Bytecode (Abbildung 16) ist deutlich zu erkennen, dass zahlreiche Instruktionen vorhanden sind, welche sich mit Lade- und Speicherzugriffen befassen. Weiterhin werden häufig Überprüfungen ausgeführt, z.B. ob die Feldreferenz ungleich *null* ist. Zusammenfassend werden durch eine Zeile Programmcode insgesamt 12 Instruktionen mit 22 Lade- und 12 Speicheranweisungen erzeugt. Entsprechende Optimierungsansätze gingen zu Lasten der Laufzeit des Programmes.

Hummel bietet einen Ansatz zur Verbesserung des Laufzeitverhaltens mittels Optimierung, welcher auf der Annotation von Bytecode basiert. Dabei agiert das spezielle Übersetzungsprogramm zu Beginn der Übersetzung wie ein traditioneller Übersetzer. Nach der Erzeugung einer Zwischencoderepräsentation, verschiedener Analysen und Optimierungen, werden die Instruktionen geschrieben. In diesem Schritt wird zusätzlich zu jeder generierten Bytecode-Instruktion eine zugehörige Annotation erzeugt und parallel zum Bytecode ausgegeben. Dieses Prinzip der Parallelität wird bei der Verbreitung des Programmcodes beibehalten. Somit ist auch eine traditionelle virtuelle Maschine für Java in der Lage, das Programm unabhängig von den Annotationen und den möglichen Optimierungen auszuführen. Ein entsprechend angepasster JIT-Übersetzer jedoch kann die gegebenen Informationen auswerten und besseren sowie effizienteren Maschinencode erzeugen, als der nicht angepasster JIT-Übersetzer .

BYTECODE	src	inter	dest	last use	r-t check
aload a			v0		
iload i			v1		
iconst 2					
aload a	v0		v0		
iload i	v1		v1		
iaload	v0, v1	v2	v3		111
imul	2, v3		v3		
aload b	v4				
iload i	v1		v1		
iaload	v4, v1	v2	v4		101
iadd	v3, v4		v3	v4	
iastore	v0, v1, v3	v2	v3	v3	000

Abbildung 17: Annotierter Bytecode zur Methode `foo`

Beispiel

In dem von Hummel entwickelten Ansatz der Java-Bytecode Annotation wird für jede Methode `foo` eines Programmes das Attribut *$$annotations* zu der Liste der Attribute hinzugefügt. Jeder Bytecode von `foo` hat einen entsprechenden Eintrag in dem Datenfeld von *$$annotations*, welcher die Annotation repräsentiert. Die virtuelle Maschine kann zur Ausführungszeit diese Annotationen auswerten oder ignorieren. Somit wird die Kompatibilität zu existierenden Laufzeitumgebungen gewahrt.

In Abbildung 17 werden die Annotationen für die Optimierung der Registerallokation aufgezeigt. Das Beispiel bezieht sich wieder auf die Methode `foo`. In den Spalten `src`, `inter`, `dest` und `last use` werden Informationen für eine virtuelle Registerallokation abgelegt, wie sie zum Zeitpunkt der Übersetzung ermittelt wurden. Entsprechend des Schemas wird später das virtuelle Register `v0` auf das physische Register `R0` abgebildet, `v1` auf `R1` usw. Ist die maximale Anzahl physischer Register erreicht, werden die verbleibenden virtuellen Register auf Speicherplätze im gemeinsamen Speicher gelegt. Weitere Informationen werden durch die Spalte `r-t check` annotiert. Für einen Feldzugriff werden implizit bis zu drei verschiedene Überprüfungen durchgeführt:

1. Ist die Feldreferenz ungleich *null*?
2. Liegt ein negativer Index vor ($i < 0$)?
3. Liegt ein Index vor, der größer ist als die Feldlänge?

Für jede dieser möglichen Überprüfungen steht in der Spalte `r-t check` ein entsprechendes Bit, wobei `1` eine Ausführung bedeutet.

Ergebnisse und Zusammenfassung

Für die Evaluierung ihres Verfahrens verwenden die Autoren keinen JIT-Übersetzer, der die Annotationen auswertet. Anstelle dessen überführen sie den übermittelten Bytecode

"von Hand" in das von der Produzentenseite benötigte Format, wobei sie die Informationen der Annotation nutzen, um selbst die möglichen Optimierungen einzuarbeiten. Die Ergebnisse wurden in Vergleich zu in der Programmiersprache C geschriebenen Programmen mit einem durchschnittlichen Zeitverlust des Faktors 1,1 ermittelt. Eine reine Ausführung durch eine JIT-Übersetzer ohne Annotation wird mit 2,0 angegeben. Ein Nachteil des vorgestellten Verfahrens liegt in der mit jeder Annotationstechnik verbundenen Vergrößerung des zu übertragenden Programmcodes. Über deren Ausmaße wird leider keine Auskunft gegeben, kann aber aufgrund der verwendeten Technik als nicht unerheblich eingeschätzt werden[5]. Ein weiterer erheblicher Nachteil dieser Art der Annotation begründet sich in den fehlenden Sicherheitsmechanismen. Es ist für den von den Autoren angestrebten *AJIT - Annotation Aware JIT* Übersetzer in keiner Weise möglich, die angebotenen Informationen zu verifizieren. Somit können Analyseinformationen durch fehlerhafte Übertragung oder manipulativen Eingriff verändert werden. Hummel verweist zwar auf zukünftige Techniken der Codesignierung durch digitale Zertifikate (z.B. Microsoft), geht jedoch nicht auf ein konkretes Beispiel ein, wie es z.B. im Bereich der mobilen Anwendungen (*Java-MIDlets*, [53]) praktiziert wird.

4.5 Annotating Java Class Files with Virtual Registers

In [40] stellen Jones und Kamin eine maschinenunabhängige Annotationstechnik für Java-Bytecode vor, die speziell auf die Verbesserung der Laufzeit eines JIT-Übersetzers im Vergleich zu einem maschinenabhängigen, optimierenden Übersetzer zielt. Die vorgestellten Annotationen sollen den JIT-Übersetzer bei verschiedenen Optimierungen unterstützen, z.B. der Registerzuweisung (*register assignment*) und der Entfernung von unnötigen Lade-Speicher-Anweisungen (*load-store elimination*).

Ein Produzent für portablen Code ist zumeist nicht in der Lage, maschinenabhängige Optimierungen durchzuführen, weshalb diese Aufgabe dem JIT-Übersetzer zufällt. Offensichtlich kann beim Konsumenten lediglich ein geringer Zeitaufwand in Optimierungstechniken investiert werden, da sich dieser zur Laufzeit des Programmes addiert. Somit ist der vom JIT-Übersetzer produzierte Maschinencode nicht so effizient, wie der von einem optimierenden Übersetzer erzeugte Code für dieselbe Maschine. Prinzipiell gehen Jones und Kamin davon aus, dass sich das Problem in eine aufwendige maschinenunabhängige Optimierung, die Bereitstellung ermittelter Ergebnisse in einer kompakten Art und deren effiziente Nutzung im Konsumentensystem trennen lässt. Sie nennen diese Trennung "super-linear analysis and linear exploitation".

Beispiel

In Abbildung 18 wird eine einfache Schleife vorgestellt, in der drei Werte (*sum*, *i* und *3*) definiert sind. Im Bytecode des Programmausschnittes werden so genannte *virtuelle Register*[6] (*Virtual Registers (VRs)*) zugewiesen, welche zur Laufzeit auf physische Register abgebildet werden können. Im Beispiel kann jedem VR ein physisches Register zugeordnet werden und somit entsteht durch die Annotation der VR kein zusätzlicher Aufwand beim JIT-Übersetzer für die Registerzuweisung. Im Normalfall ist die Anzahl der vorhandenen

[5] 5-6 Informationen pro Instruktion.

[6] Die Registerannotationen stehen in der dritten Spalte der Bytecode-Darstellung.

Register der Konsumentenseite kleiner als die verwendete Anzahl virtueller Register. Die vorgestellte Technik eignet sich somit auch zur schnellen Bestimmung von Stellen im Programm, an denen eine Aufteilung auf die vorhandenen Register notwendig wird (*register spilling*).

```
int sum = 0;
for (int i = 0; i < 3; i++){
      sum +=i;
}
```

```
0  iconst_0      1
1  istore_0      1
2  iconst_0      0
3  istore_1      0
4  goto 13       -
7  iinc 0,1      1
10 iinc 1,1      0
13 iload_1       0
14 iconst_3      2
15 if_icmplt 7   0,2
```

Abbildung 18: Schleife und annotierter Java-Bytecode

Codegenerierung

Jones und Kamin konzentrieren sich auf die Annotation von virtuellen Registern (VR-Annotation) und setzen dabei die Anzahl von virtuellen Registern für einen bestimmten Bytecode voraus, die der Menge der im Bytecode verwendeten Operanden entspricht. Die VR-Annotation wird verwendet, um Registerzuweisungen für die Maschineninstruktionen zu erzeugen, wobei alle VR mit einer Priorisierung versehen werden. Sind auf der Konsumentenseite weniger Register vorhanden als benötigt, so werden bestimmte physische Register als temporäre Register reserviert.

Im folgenden sind einige Richtlinien zur Codegenerierung dargestellt. Der Begriff *physical register* steht für reale Maschinenregister und *physical location* verweist auf eine Position auf der Maschine, die entweder ein Maschinenregister sein oder im Speicher liegen kann. Weiterhin bezeichnet *primarily stored* jene Position, unter welcher der einem virtuellen Register zugeordnete Wert üblicherweise gefunden werden kann.

- Alle virtuellen Register haben mindestens eine physische Position (*physical location*) auf dem Stapel, wobei VR, die primär auf physische Register verweisen, eine weitere Position besitzen - das physische Register.
- Konstanten werden entweder durch Maschinenbefehle ausgedrückt, oder statisch zu jeder Methode auf dem Heap angelegt.
- Der Operandenstapel ist so angelegt, dass alle virtuellen Register, die nicht physischen Registern zugeordnet sind, bei Bedarf in temporäre physische Register geladen werden.
- Die Zuordnungen der virtuellen Register auf physische Positionen (*physical locations*) werden in einer VR-Positionstabelle (*VR location table*) verwaltet.

Bei der Erzeugung des entsprechenden Maschinencodes wird nach folgendem Schema vorgegangen:

1. Für alle eingehenden VR des Bytecodes, die keinem physischen Register zugeordnet sind, wird entsprechend der *VR location table* eine Ladeanweisung vom Stapel in ein temporäres physisches Register erzeugt.

2. Besteht für ein ausgehendes VR im Bytecode keine Zuordnung zu einem physischen Register, so wird es auf ein temporäres physisches Register abgebildet.

3. Der Maschinencode wird unter Verwendung der temporären und permanenten physischen Register erzeugt.

4. Ist ein ausgehendens VR im Bytecode keinem permanenten physischen Register zugeordnet, wird eine entsprechende Speicheranweisung von dem verwendeten temporärem Register auf den Stapel erzeugt.

Die VR-Positionstabelle wird bei der Codeerzeugung unter Verwendung von Informationen ermittelt, die während der Verifikation von Bytecode und VR-Annotation gesammelt werden. Die Verifikation dient der JVM zur Sicherheit, dass eine Instruktion im Bytecode eines Programmes nicht die Virtuelle Maschine gefährdet. Dieser Schritt wurde modifiziert, um zusätzlich die Annotation der mit jeder Methode assoziierten VR-Annotation zu verifizieren. Als Seiteneffekt dieses Prozesses wird der Typ eines jeden virtuellen Registers festgestellt.

Annotationen

Eine VR-Annotation besteht aus einer Zuweisung von einer Menge von virtuellen Registernummern für jede Instruktion des Java-Bytecodes, die mit den Operanden in der jeweiligen Instruktion korrespondiert. Die Anzahl der virtuellen Register variiert je nach Instruktion. Weiterhin sind VR *monotyp*, in dem Sinne, dass in jedem VR über den Lebenszeitraum einer Methode nur Werte eines Typs abgespeichert werden können. Das schließt Referenztypen ein, deren obere Schranke, definiert durch die erste gemeinsame Superklasse in der Vererbungshierarchie, durch einen Datenflussalgorithmus während der Bytecode-Verifikation ermittelt wird. Für die Annotation jedes VR wird zusätzlich ein Byte Speicherplatz benötigt.

Die Erzeugung der VR-Annotation ähnelt im Prinzip der einer herkömmlichen Registerallokation, wobei die Autoren einen einfachen Graph-Coloring-Algorithmus einsetzen, dessen Einschränkung darin besteht, dass die Anzahl der tatsächlichen physischen Register nicht bekannt ist. Der veränderte Allokator basiert auf dem *Chaitin-Graph-Coloring-Register-Allokator*, berechnet aber anstelle eines entsprechendem *k-Coloring*, das Minimum k, für das ein *k-Coloring* des Graphen existiert. Zur Unterstützung des Verifikationsprozesses werden Kanten zwischen Knoten unterschiedlichen Typs in den Graphen eingefügt, sodass schließlich jedes virtuelle Register nur einen Typ repräsentiert (*monotyp*). Weiterhin werden die Farben des Algorithmus im abstrakten Sinne von Graustufen priorisiert. Die Priorisierung orientiert sich an so genannten “Haifa Heuristiken” ([18]).

4.6 Efficiently Verifiable Escape Analysis

Beers [17] stellt die Annotation von Ergebnissen einer Escape-Analyse vor, welche als eine bedeutende Eigenschaft Verifizierbarkeit aufweist. Die Escape-Analyse wird auf der

Laufzeittypen	
$v = new\ C$	$rtt\,(v) \geq C$
$C_0\ m\,(C_1\ p_1, ...)$	$\forall p_i :\ rtt\,(p_i) = \top$
$v = s$	$rtt\,(v) = \top$
$v_0 = v_1.f$	$rtt\,(v_0) = \top$
$v_0 = v_1[...]$	$rtt\,(v_0) = \top$
$v_0 = v_1$	$rtt\,(v_0) \geq rtt\,(v_1)$
$v_0 = m\,(v_1, v_2, ...v_n)$	$rtt\,(v_0) = \top$
Escape-Eigenschaft	
return v	$esc\,(v) = \top$
throw v	$esc\,(v) = \top$
$s = v$	$esc\,(v) = \top$
$v_0.f = v_1$	$esc\,(v_1) = \top$
$v_0[...] = v_1$	$esc\,(v_1) = \top$
$v_0 = v_1$	$esc\,(v_0) \rightarrow esc\,(v_1)$
$v_0 = m\,(v_1, v_2, ...v_n)$	$esc\,(v_0) = \top \wedge$
	$\forall$ Parameter $p_i^{m'}$ von Methoden m',
	aufrufbar als m: $esc\left(p_i^{m'}\right) = esc\,(v_i)$

Tabelle 1: Bedingungen für die Bestimmung des Laufzeittyps und der Escape-Eigenschaft

Produzentenseite eines Systems durchgeführt und kann in linearer Zeit durchgeführt werden, wobei die Qualität der Analyse von Whaley und Rinard nicht erreicht wird. In Zusammenhang mit der Escape-Analyse werden die Ergebnisse bei vergleichsweise geringer zusätzlicher Belastung der Zwischencoderepräsentation annotiert und können auf der Konsumentenseite verifiziert und für Zwecke der Optimierung verwendet werden.

Escape-Analyse

Prinzipiell kann die vorgestellte Analyse in zwei Phasen unterteilt werden, in denen je die Eigenschaft $rtt\,(v)$ und $esc\,(v)$ einer Variablen v bestimmt wird. Die erste Phase ermittelt den so genannten Laufzeittyp $rtt\,(v)$, welcher entweder eine Klasse C sein kann oder nicht initialisiert ($\perp$) bzw. initialisiert, jedoch unbekannt ($\top$). Die Bedeutung von $rtt\,(v) = \top$ kann mit "der Laufzeittyp von v entspricht dem deklarierten Typ oder eines seiner abgeleiteten Typen" beschrieben werden. Diese Elemente bilden einen Halbverband mit der partiellen Ordnung $\leq$ und größtem Element $\top$.

Die zweite Phase bestimmt die eigentliche Escape-Eigenschaft $esc\,(v)$ für jede Variable v, wobei lokale Objekte und Parameter untersucht werden. Die Escape-Eigenschaft wird durch die beiden Werte $\perp$ (bound) und $\top$ (escape) definiert. Zu beachten ist, dass die Technik keine Escape-Informationen für Objekte analysiert, sondern nur Variablen, welche auf Objekte verweisen. Hierbei steht die Escape-Eigenschaft $esc\,(v)$ für "ein Objekt könnte durch v fliehen". Daraus folgt, dass ein Objekt dann nicht flieht, also gebunden ist, wenn es durch keine fliehende Variable referenziert wird.

Die Autoren verwenden nun diese Eigenschaften zur Aufstellung eines Bedingungs-

modells (siehe Tabelle 1)[7], welches in linearer Zeit gelöst werden kann. Hierbei wird das dynamische Klassenladen in Java durch eine zusätzliche Annahme berücksichtigt. In einer Version des Verfahrens wird von einer *geschlossenen* Welt ausgegangen, in der alle zu untersuchenden Klassen und Typen bekannt sind. In dieser Welt sind für den Typ einer deklarierten Variable v auch alle möglichen abgeleiteten Typen v_i bekannt. In der *offenen* Welt hingegen kann ein abgeleiteter Typ nicht immer abschließend genau bestimmt werden. Die Analyse trifft in diesem Fall die Annahme, dass genau die Variablen fliehen, die als Argumente an eine Methode gegeben werden und deren korrespondierende Parameter keinen eindeutig bestimmbaren Typ aufweisen.

```
public Map transformMap(
  MyClass this,          //Annotation: esc(this)=⊥, rtt(this)=⊤
  Map m,                 //Annotation: esc(m)=⊤, rtt(m)=⊤
  MyToken token)         //Annotation: esc(token)=⊥, rtt(token)=⊤
{
  Iterator iter;         //Annotation: esc(iter)=⊤, rtt(iter)=⊤
  Object key;            //Annotation: esc(key)=⊤, rtt(key)=⊤
  Object val;            //Annotation: esc(val)=⊤, rtt(val)=⊤

  ...
}
```

Abbildung 19: Auszug der Annotation für eine Methode

Annotation und Verifikation

Zur Übertragung der Analyseinformationen werden für jede lokale Variable und jeden formalen Parameter v an dessen Deklarationsstelle die Escape-Information $esc(v)$ und der Laufzeittyp $rtt(v)$ annotiert. Abbildung 19 zeigt beispielhaft und auszugsweise die Annotation für ein untersuchtes Programm. Die Größe der Annotationen berechnet sich aus je 6 Byte pro Methode, ein Bit für die Escape-Information $esc(v)$, sowie für den Laufzeittyp $rtt(v)$ je eine 4 Byte große Referenz auf die Konstantentabelle der Klasse. Insgesamt bewegt sich die Vergrößerung einer mit dieser Technik annotierten Klasse zwischen 1% und 11%, wobei keine Komprimierung (JAR-Archive) untersucht wird.

Die Verifikation der Annotation erfolgt während des Einlesens einer Klasse über die Erzeugung und Überprüfung der in Tabelle 1 dargestellten Bedingungen. Wird hier eine Bedingung gefunden, welche nicht erfüllt werden kann, so liegt eine Manipulation des Zwischencodes vor. Zu beachten ist, dass durch diese Art der Verifikation niemals die ursprüngliche Information garantiert werden kann, sondern nur eine mögliche Lösung, welche im ungünstigen Fall schlechter sein kann bzw. keine Optimierung unterstützt (d.h. wenn alle Variablen fliehen). Die Verifikationszeiten im JIT-Übersetzer werden mit 1,90 - 137,06 Millisekunden angegeben.

[7] Hier stehen v, v_i für lokale Variablen oder formale Parameter, s für statische Felder, f für Instanzfelder, m für Methoden und C für Klassen und deren abgeleitete Typen C_i.

Zu bemerken ist, dass die Implementierung der Technik nicht für Java-Bytecode, sondern ein baumorientiertes Zwischencodeformat erfolgte und dass bei Verwendung von Bibliotheken, welche die Annotation nicht unterstützen, keine vollständige Kompatibilität gewährleistet wird. Weiterhin beschränkt sich die Unterstützung dynamischen Klassenladens auf die Annahme, dass im schlechtesten Fall alle formalen Parameter fliehen, also immer dann, wenn der Laufzeittyp nicht eindeutig feststellbar ist.

5 Thesen

In den vorangehenden Kapiteln wurden einige verschiedene Techniken zur Escape-Analyse vorgestellt, welche durch Anwendung komplexer Algorithmen sehr gute Ergebnisse erzielen können und somit als Basis einer Optimierung dienen. Ein erheblicher Nachteil dieser Arbeiten ist jedoch die fehlende Unterstützung für dynamische Aspekte eines Systems mit Programmübertragung durch mobilen Code, wie es z.B. bei Java angewendet wird. Obwohl einige Techniken auf die Anwendung in einem Java-System gerichtet sind, wird vorausgesetzt, dass alle zu untersuchenden Klassen bereits auf der Produzentenseite des Systems vollständig bekannt sind. Nachträgliche Anpassungen der Analyse-Ergebnisse, wie sie z.B. durch das dynamische Laden von Klassen notwendig werden könnten, werden explizit ausgeschlossen (z.B. [31], [23]). Vielmehr müssen bei diesen Techniken für eine vollständige Analyse immer alle Klassen eines Programmes bereits auf der Produzentenseite bekannt sein ([47], [61]).

Auf der anderen Seite könnten die vorgestellten Algorithmen auch auf der Konsumentenseite in einem optimierenden JIT-Übersetzer eingesetzt werden. An dieser Stelle beeinträchtigen die komplexen Algorithmen mit zum Teil aufwendigen internen Datenstrukturen allerdings die Übersetzungszeit des JIT-Übersetzers und somit die Ausführungszeit des Programmes. In Zusammenhang mit der im Anschluss durchzuführenden Optimierung ist dieser Aufwand zumeist nicht mehr praktikabel.

Eine Lösung, die auf der einen Seite eine gute Basis für Optimierungsalgorithmen bietet und auf der anderen Seite verhältnismäßig geringen Einfluss auf die Ausführungszeit des Programmes nimmt, bietet eine Analyse auf der Produzentenseite verbunden mit einer Annotation entsprechender Analyse-Informationen ([41]). Auf dem Gebiet der Annotation wurden einige Arbeiten vorgestellt, die potenziell eine gute Basis für die Optimierung mit mobilem Code übertragener Programme bilden. Ein entscheidender Nachteil vieler Verfahren ist jedoch die fehlende Verifizierbarkeit der übertragenen Informationen. Ob durch Übertragungsfehler oder absichtliche Manipulation, die Korrektheit der Analyse-Ergebnisse ist auf der Konsumentenseite nicht verifizierbar und entsprechende Optimierungen könnten im schlechtesten Fall zu einer fehlerhaften Ausführung des Programmes führen ([38], [40], [32], [50]). Eine Ausnahme bilden Techniken, welche lediglich Informationen zu nicht optimierbaren Programmteilen liefern, da hier im schlechtesten Fall lediglich eine mögliche Optimierung ausgeschlossen wird ([62]). Allerdings verlagern diese Techniken nicht den Aufwand einer Analyse auf die Produzentenseite, sondern sparen nur einen Teil der Arbeit auf der Konsumentenseite ein.

Die in dieser Dissertation vorgestellte Methode der effizienten und verifizierbaren Programmannotation für den Transport von Escape-Informationen verknüpft die Vorteile einer auf der Produzentenseite durchgeführten Escape-Analyse mit der Sicherheit einer vollständig verifizierbaren Form der Annotation, die inhärent sichere Annotation. Insbesondere können durch diese Methode folgende Ziele erreicht werden:

1. Es wird eine inhärent sichere Annotation geschaffen, die das Zwischencodeformat *SafeTSA* erweitert und vollständig verifizierbar durch Konstruktion ist.

2. Dynamische Aspekte der Programmiersprache Java und der JVM werden durch Trennung der Analyse für die Produzenten- und Konsumentenseite des Systems sowie Anpassungen der nachfolgenden Optimierung zur Laufzeit unterstützt.

3. Die Übersetzungszeit eines optimierenden JIT-Übersetzers wird im Vergleich zu einer auf der Konsumentenseite durchgeführten Escape-Analyse nur gering beeinflusst.

4. Die durch die Annotation verursachte Vergrößerung des Zwischencodes ist im Vergleich zur Programmgröße sehr gering.

5. Das Verfahren ist für weitere Analysetechniken einsetzbar, deren Ergebnisse sich als Typen in einem erweiterten Maschinenmodell von SafeTSA formulieren lassen.

6 Annotation von Escape-Informationen mit SafeTSA

Die Techniken zur Annotation von Analyseergebnissen in der Zwischencoderepräsentation stellen in Zusammenhang mit einem JIT-Übersetzer ein effizientes Verfahren zur Übertragung von Informationen dar, wovon eine zur Laufzeit durchgeführte Optimierung profitieren kann. Mit dieser Technik ist es möglich, komplexe Analysen bereits auf der Produzentenseite durchzuführen und die Ergebnisse zusammen mit dem Zwischencode zum JIT-Übersetzer zu übertragen. Dieser kann nach Auswertung der annotierten Informationen eine entsprechende Optimierung starten. Während der Übertragung des annotierten Zwischencodes können jedoch Fehler oder absichtliche Manipulationen des Codes die Annotationen verändern, ohne dass der JIT-Übersetzer diese Änderungen nachweislich feststellen könnte.

In diesem Kapitel wird die Escape-Analyse mit einer Annotationstechnik kombiniert, die auf dem Zwischencodeformat SafeTSA ([10],[11]) basiert. Mit SafeTSA dargestellte Programme besitzen bestimmte, im Folgenden noch ausführlich erläuterte Eigenschaften hinsichtlich der Typ- und Referenzsicherheit, die als Basis für eine sichere Annotation von Escape-Informationen verwendet werden. Die auf diese Weise annotierten Informationen sind insbesondere verifizierbar und können auf der Konsumentenseite eines SafeTSA-Systems für entsprechende Optimierungen verwendet werden.

Zur Unterstützung weiterer spezieller dynamischer Eigenschaften der Programmiersprache Java und der JVM wird die Escape-Analyse in dieser Arbeit in zwei Teile getrennt. Damit wird jenen Aspekten Rechnung getragen, wo bestimmte, das Ergebnis einer Analyse beeinflussende Informationen erst zur Laufzeit eines Programmes verfügbar sind. Eine intraprozedurale, vorbereitende Analyse wird statisch auf der Produzentenseite eines Systems durchgeführt, wobei lediglich einzelne Methoden ohne Abhängigkeiten auf fremde Methoden und Klassen untersucht werden. Eine abschließende interprozedurale Analyse findet dynamisch auf der Konsumentenseite im JIT-Übersetzer Anwendung, wenn alle zur Untersuchung notwendigen Informationen wirklich vorliegen.

Im Weiteren wird das Zwischencodeformat SafeTSA vorgestellt, die Aufteilung der Escape-Analyse beschrieben und schließlich die Erweiterung des SafeTSA-Maschinenmodells für die Unterstützung einer Annotationstechnik eingehend erläutert.

6.1 SafeTSA

6.1.1 Einleitung

SafeTSA ist eine typsichere mobile Zwischencoderepräsentation, welche auf der Static Single Assignment Form (SSA) aufbaut und als Alternative zum Java-Bytecode der Firma *Sun Microsystems* entwickelt wurde. SafeTSA bietet dabei mehrere entscheidende Vorteile, wie Typ- und Referenzsicherheit durch Konstruktion, auch als inhärent sicherer Code bezeichnet. Weiterhin liefert SafeTSA die Grundlage für eine Reihe effizienter Optimierungstechniken. So können unter anderem Ergebnisse einer Nullreferenz- und Reihungszugriffsüberprüfungs-Eliminierung sicher mit diesem Code übertragen werden. *Sicher* bedeutet hier in erster Linie, dass ein durch SafeTSA repräsentiertes Programm mit enthaltenen Nullreferenz- und Reihungszugriffsüberprüfungs-Eliminierungen seine Typsicherheit behält, selbst wenn der Zwischencode böswillig von Hand manipuliert wurde. SafeTSA

eignet sich jedoch aufgrund seiner Struktur auch für weitere Übertragungstechniken. Im folgenden Abschnitt wird die SafeTSA-Zwischencoderepräsentation im Detail vorgestellt.

6.1.2 Maschinenmodell

Abbildung 20 enthält ein einfaches Java-Programm und das dazu äquivalente Programm in SSA-Form. Jede Zeile in der Darstellung steht für eine Instruktion, die einen Wert produziert, gekennzeichnet durch die so genannte *Instruktionsnummer*. Wird ein Wert einer Instruktion im folgenden Verlauf des Programmes als Operand einer weiteren Instruktion verwendet, so wird die entsprechende Instruktionsnummer in Klammern dargestellt.

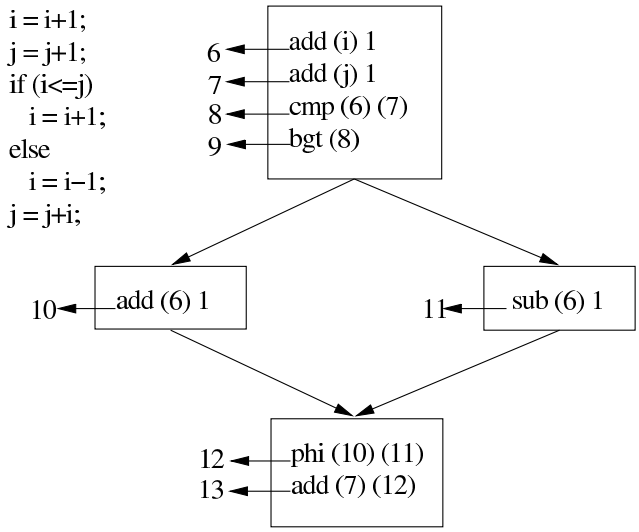


Abbildung 20: Einfaches Java-Quellprogramm und zugehörige SSA-Form

Eine Schwachstelle dieser Darstellung liegt in der fehlenden referenziellen Integrität des so repräsentierten Programmes. So könnte zum Beispiel durch einen Übertragungsfehler oder böswillige Manipulation das Programm aus Abbildung 20 so geändert werden, dass die Instruktion **13** auf den Wert **(11)** zeigt. Dieser Wert ist jedoch nur dann definiert, wenn der Kontrollfluss den Else-Zweig der Bedingungsanweisung durchläuft, was zwangsläufig zu einem Fehler führen würde.

In SafeTSA hingegen wird eine referenzielle Integrität auf der Basis der Dominator-Relation eingeführt. Das bedeutet für jede Instruktion eine Einschränkung der Referenzen auf ausschließlich diejenigen Werte, die diese Instruktion *dominieren*. Ein Wert *a dominiert* einen Wert *b*, wenn *a* auf jedem Ausführungspfad des Programmes vor *b* ausgeführt wird. Diese Erweiterung führt zu einer Darstellung, in der jede Referenz zu einer dominierenden Instruktion durch ein Paar $(l - r)$ repräsentiert wird, wobei l für einen Basisblock und r für die den Wert erzeugende Instruktion innerhalb dieses Blockes steht. Der Basisblock l entspricht der Anzahl von Schritten, die im Dominatorbaum vom aktuellen Block bis zum definierenden Block zurückgegangen werden muss. Handelt es sich um eine phi-Instruktion und ist $l=0$, so wird der der phi-Instruktion vorausgehende Basisblock im durch die Operandenreihenfolge gegebenen Kontrollfluss referenziert. Ist $l \neq 0$, steht l ebenfalls für eine

entsprechende Anzahl von Schritten. Jedoch wird nun bei dem aus der Operandenfolge ableitbaren und im Kontrollfluss vorausgehenden Block mit dem Zählen begonnen. Abbildung 21 zeigt das genannte Beispielprogramm in referenzsicherer SSA-Form.

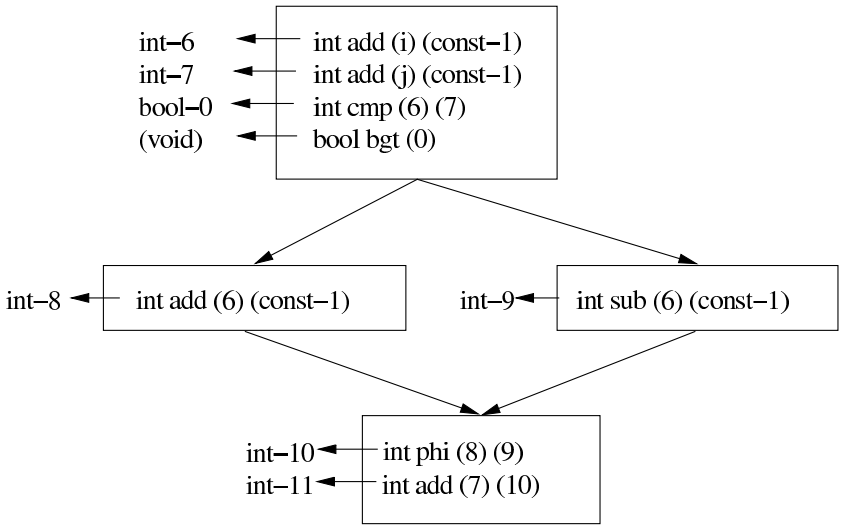


Abbildung 21: Beispielprogramm aus Abbildung 20 in referenzsicherer SSA-Form

Typentrennung ist ein weiterer grundlegender Aspekt von SafeTSA. Im Gegensatz zu SSA, dessen Maschinenmodell einen einzigen unbeschränkten Registersatz enthält, definiert das von SafeTSA verwendete Maschinenmodell für jeden Typ einen entsprechenden Registersatz. In diesem Registersatz können ausschließlich Werte des zugehörigen Typs abgelegt werden. Die Registersätze werden implizit während der Übersetzung des Programmes erzeugt, wobei jedem im Programm verwendeten Typ genau ein korrespondierender Registersatz entspricht. Abbildung 22 zeigt die schematische Darstellung des von SafeTSA verwendeten Maschinenmodells, der Typentabelle und des typisierten Konstantenpools.

Die Verwendung des korrespondierenden Registersatzes, bzw. dessen Auswahl durch die entsprechende Instruktion, ist implizit als Teil der Operationen gegeben. Jede Instruktion wählt automatisch die Registersätze der Quell- und Zielregister aus, wobei SafeTSA-Instruktionen als Operanden lediglich Registernummern erhalten. Die eigentliche Abbildung auf die Registersätze wird von jeder Instruktion selbst durchgeführt. So hat eine Integer-Addition in SafeTSA als Operanden die beiden Registernummern *r1* und *r2*. Die Operandenwerte holt sie sich entsprechend automatisch aus den Registern *r1* und *r2* des Integer-Registersatzes und speichert nach Ausführung ihr Ergebnis im nächsten freien Integer-Register *r3* dieses Registersatzes ab. Vergleichbar würde eine Integervergleichsoperation ihre Operanden aus dem Integer-Registersatz beziehen, jedoch ihr Ergebnis im nächsten freien Boolean-Register des Boolean-Registersatzes ablegen, da ihr Ergebniswert vom Typ Boolean ist. Abbildung 23 zeigt das oben vorgestellte Beispielprogramm aus Abbildung 20 in typ- und referenzsicherer SSA-Form (SafeTSA).

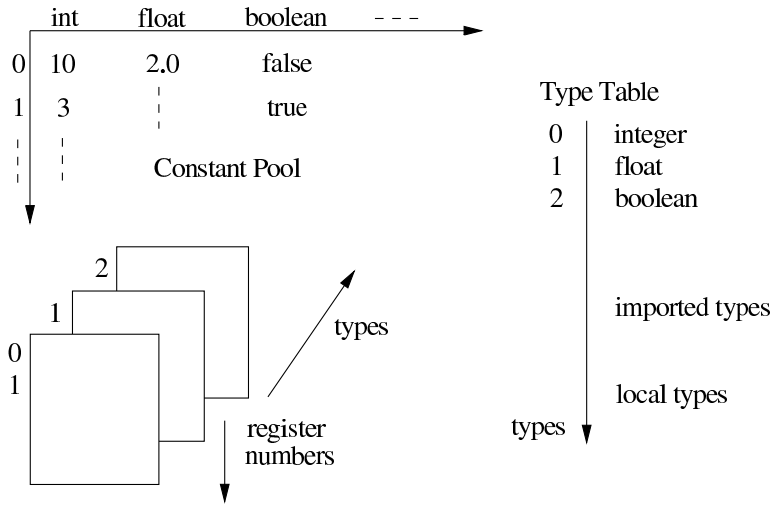


Abbildung 22: Typenmodell von SafeTSA

6.1.3 Typ- und Referenzsicherheit

Die SafeTSA-Form definiert spezielle Operationen zur Verwendung für Nullreferenz- und Reihungszugriffsüberprüfungen. Damit können unbenötigte Nullreferenz- und Reihungszugriffsüberprüfungen bereits bei der Übersetzung des Programmes auf der Produzentenseite aus dem Zwischencode entfernt werden. Zur Verwendung dieser Operationen wird für jeden Referenztyp R ein sicherer Referenztyp R_{safe} und für jedes Array A ein sicherer Indextyp A_{safe} eingeführt. Die diesen Typen zugeordneten Registersätze und Register können nur Referenzen ungleich der Nullreferenz beziehungsweise im Indexbereich eines Arrays liegende Integerwerte enthalten. Wird ein Wert für ein sicheres Register eingeführt, so muss er durch eine spezielle Instruktion bestimmt sein, die explizit zur Nullreferenz- oder Reihungszugriffsüberprüfung in SafeTSA definiert ist. Speichersicherheit in SafeTSA wird durch spezielle Restriktionen der SafeTSA-Operationen erreicht. So dürfen alle Operationen, mit denen auf den Speicher oder ein Array zugegriffen werden kann, ihre Operanden ausschließlich aus den ihrem Typ entsprechenden sicheren Registersätzen beziehen. Die im Folgenden vorgestellten SafeTSA-Operationen verdeutlichen die genannten Eigenschaften von SafeTSA.

6.1.4 SafeTSA-Operationen

Primitive Operationen Die Ausführung von Operationen, die für primitive Datentypen definiert sind, werden in SafeTSA durch die primitiven Operationen *primitive* und *xprimitive* formuliert:

primitive *type op* $op_1 ... op_n$, $\quad type \in Type,\ op \in Symbol,\ op_1 ... op_n \in N$
xprimitive *type op* $op_1 ... op_n$, $\quad type \in Type,\ op \in Symbol,\ op_1 ... op_n \in N$

Hierbei spezifiziert *type* den primitiven Datentyp, für den die Operation auszuführen ist, *op* den Namen der Operation und $op_1 ... op_n$ die Register, in denen die benötigten Ope-

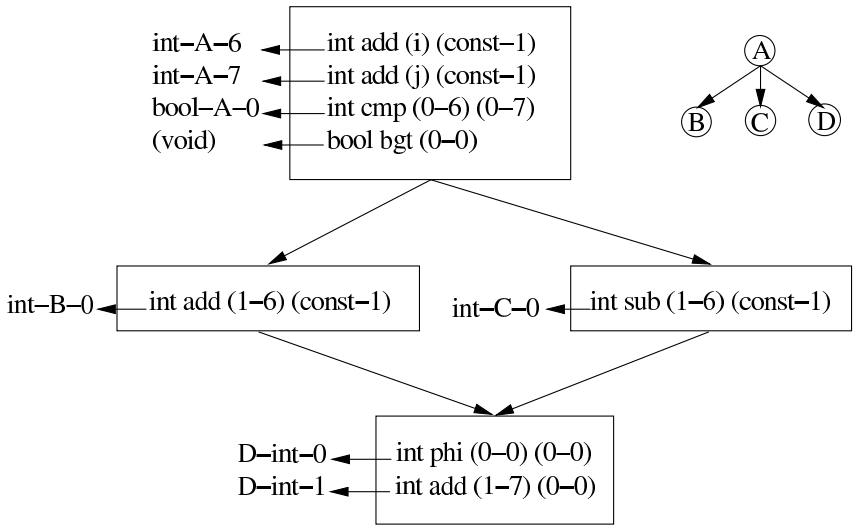


Abbildung 23: Beispiel in typ- und referenzsicherer SSA-Form

randen gespeichert sind. Weiterhin legt die Operation durch *op* implizit fest, aus welchem Registersatz die Operanden zu beziehen sind und in welchen Registersatz das Ergebnis der Berechnung nach Ausführung gespeichert wird. Der Unterschied zwischen *primitive* und *xprimitive* liegt in der Behandlung von Ausnahmefällen bzw. Fehlerfällen, wobei *xprimitive* eine Ausnahme auslöst, *primitive* jedoch mit einem fehlerhaften Wert weiter arbeitet.

Speicheroperationen Die Nullreferenz- und Indexüberprüfung wird durch eine Erweiterung des SafeTSA-Maschinenmodells realisiert. Für jeden Referenztyp *ref* wird ein sicherer Referenztyp *ref-safe* und für jede Reihung ein sicherer Indextyp *safe-arr-index* eingeführt. Weiterhin können Register des sicheren Referenztyps nur Werte ungleich der Nullreferenz und Register des sicheren Indextyps nur Integerwerte aus dem Indexbereich der jeweiligen Reihung enthalten. Es gilt ferner die Einschränkung, dass Werte sicherer Register ausschließlich durch die Instruktionen *xnullcheck* und *xindexcheck* zur expliziten Null- und Indexüberprüfung definierbar sind:

xnullcheck *ref object*, $ref \in Type,\ object \in N$
xindexcheck *arr index*, $arr \in Type,\ index \in N$

Hierbei bezeichnet *ref* den Typ der zu überprüfenden Referenz und $object$ das Register, in dem die Referenz abgespeichert ist. Entspricht bei Ausführung der Operation der angegebene Referenzwert der Nullreferenz, so wird eine Ausnahme ausgelöst. Anderenfalls wird eine Kopie des Wertes im nächsten verfügbaren Register des zugehörigen sicheren Registersatzes *ref-safe* abgelegt. Analog steht *arr* für den Reihungstyp und $index$ bezeichnet das Integer-Register des den Index repräsentierenden Operanden. Liegt bei Ausführung der Operation der Index außerhalb des Indexbereiches der Reihung, so wird eine Ausnahme ausgelöst. Anderenfalls wird der Integerwert im sicheren Registersatz *safe-arr-index* abgelegt.

Die Speichersicherheit ergibt sich nunmehr durch eine Restriktion aller in SafeTSA definierten Operationen mit Speicher- bzw. Indexzugriff. Für diese Operationen gilt, dass sie ihre Operanden ausschließlich aus den sicheren Registersätzen *ref-safe* und *safe-arr-index* beziehen dürfen. Die folgenden Operationen $getfield$ und $setfield$ stehen für Feldzugriffe (Klassenattribute in Java) zur Verfügung:

getfield *ref object field*,	$ref \in Type,\ object \in N, field \in Symbol$
setfield *ref object field value*,	$ref \in Type,\ object,\ value \in N,\ field \in Symbol$

Für beide Operationen bezeichnet *ref* den Typ des Referenzwertes, auf dem die Operation ausgeführt werden soll, und *object* das Register des sicheren Registersatzes *ref-safe*, aus dem der Operand für diesen Referenzwert bezogen werden kann. Ferner steht *field* für den Namen des zu lesenden oder schreibenden Feldes. Die Operation *getfield* speichert den Wert des gelesenen Feldes in dem Registersatz des *field* zugehörigen Datentyps ab. Aus dem ebenfalls durch den Typ von *field* bestimmten Registersatz liest *setfield* das Register *value* und speichert den Wert in das Feld *field*.

Die Operationen *getelt* und *setelt* können für den Zugriff auf Reihungen verwendet werden:

getelt *arr object index*,	$arr \in Type,\ object,\ index \in N$
setelt *arr object index value*,	$arr \in Type,\ object,\ index,\ value \in N$

Hier steht *arr* für den entsprechenden Reihungstyp und *object* für das Register des sicheren Registersatzes *arr-safe*, in dem die Basisadresse der Reihung zu finden ist. Weiterhin bezeichnet *index* das Register des sicheren Registersatzes *safe-arr-index*, der den Index für diese Operation enthält, und *value* das Register des dem Elementtyp der Reihung zugeordneten Registersatzes, dessen Wert dann im entsprechenden Element der Reihung gespeichert wird. Das Ergebnis von *getelt* wird im nächsten verfügbaren Register des dem Elementtyp der Reihung zugeordneten Registersatzes abgelegt.

Operationen zur Konvertierung von Referenzwerten Die folgenden Operationen *xupcast* und *downcast* stehen in SafeTSA zur Typkonvertierung von Referenzwerten zur Verfügung.

xupcast *src target object*	$src,\ target \in Type,\ object \in N$
downcast *src target object*	$src,\ target \in Type,\ object \in N$

Durch die in SafeTSA eingeführte strikte Typentrennung, welche durch das Maschinenmodell in getrennten Registersätzen dargestellt wird, entspricht eine Typkonvertierung dem Kopieren eines Wertes zwischen zwei Registersätzen. Bei der Operation *xupcast* wird von dem durch *src* gegebenen Typ der Ursprungsregistersatz abgeleitet und *object* steht für das den Wert enthaltende Register. Der Zielregistersatz, in dessen nächstem verfügbaren Register der Wert bei möglicher Konvertierung kopiert wird, ist dem Typ von *target* zugeordnet. Ist keine Konvertierung möglich, so wird eine Ausnahme ausgelöst. Die Operation *downcast* arbeitet analog, jedoch wird immer von einer möglichen Konvertierung ausgegangen.

Die Operationen können in einer speziellen Anwendung für Null- und Indexüberprüfungen eingesetzt werden (*xnullcheck* und *xindexcheck*), indem die zu prüfenden Werte von einem unsicheren in einen sicheren Typ konvertiert werden.

Methodenaufrufe und ϕ-Funktion Die Operationen *xdispatch* und *xcall* stehen in SafeTSA für die Durchführung der Methodenaufrufe zur Verfügung:

xcall *base rec meth* $op_1...op_n$,	$base \in Type,\ meth \in Symbol, rec,\ op_i \in N$
xdispatch *base rec meth* $op_1...op_n$,	$base \in Type,\ meth \in Symbol,\ rec,\ op_i \in N$

Das Register mit dem Empfängerobjekt wird in beiden Operationen durch *rec* bezeichnet, wobei der zugehörige sichere Registersatz dem in *base* bestimmten Typ entspricht. Die aufzurufende Methode wird durch das Symbol *meth* festgelegt und *op* steht für die Register, aus denen die Operandenwerte des Methodenaufrufes bezogen werden können. Der Rückgabewert wird im nächsten freien Register des dem Resultattyps zugeordneten Registersatz abgelegt.

Der Unterschied zwischen beiden Operationen ergibt sich aus der Art und Weise, wie die Methoden an den Aufruf gebunden werden. Bei *xcall* findet eine statische Bindung, bei *xdispatch* hingegen eine dynamische Bindung statt. Bei letzterer wird auf der Produzentenseite lediglich die Methodensignatur festgelegt und die eigentliche Bindung erst auf der Konsumentenseite durchgeführt.

Zur Darstellung von ϕ-Funktionen kann in SafeTSA folgende Operation verwendet werden:

phi *type* $op_1...op_n$, $type \in Type,\ op_1...op_n \in N$

Die phi-Operation ist auf Operanden gleichen Datentyps beschränkt, der durch *type* gekennzeichnet ist. Die möglichen Operandenwerte stehen in den durch *op* bezeichneten Registern. Bei Ausführung wird der Operand i des aktuellen Kontrollflusses berechnet und dessen Operandenwert op_i in das nächste verfügbare Register des durch *type* bestimmten Registersatzes abgelegt. Die phi-Operation in SafeTSA ist rein auf die Zwischencoderepräsentation ausgelegt. Auf der Konsumentenseite können die vorhandenen phi-Operationen aufgelöst werden.

6.2 Aufteilung der Escape-Analyse

Die Verlagerung der Escape-Analyse von der Laufzeit in die Übersetzungszeit bietet einen enormen Vorteil. Mit der Auswertung annotierter Analyseergebnisse kann der JIT-Übersetzer, ohne durch den hohen Aufwand einer kompletten Analyse belastet zu werden, profitable Optimierungen durchführen. Die ersparte Zeit kann direkt für die Implementierung besserer Optimierungstechniken verwendet werden. Dennoch besitzt diese Herangehensweise einen bedeutenden Nachteil, der in den Eigenschaften der Programmiersprache Java und der JVM begründet ist. Die Spezifikation von Java erlaubt so z.B. das dynamische Laden von Klassen[8]. Durch diese Technik können Klassen zur Laufzeit eines Programmes, also nach der Übersetzung auf einer Produzentenseite, in die JVM geladen und ausgeführt werden. Das besondere hierbei ist, dass sich ein Programm bereits in der Ausführung befinden kann, bevor alle benötigten Klassen vollständig geladen und verfügbar sind.

In Bezug auf die Optimierung, welche sich die Ergebnisse einer Escape-Analyse zu Nutze macht, ergibt sich ein Problem bezüglich der Interaktion zwischen verschiedenen

[8]Kapitel 7.2 erläutert dieses Verfahren.

```
class FrameCheck{
      boolean checkIntersect(FRect upper){
          FRect lower = new FRect(10,15,30,30);
          return upper.intersects(lower);
      }



final class FrameBorder{
   static void incDefBorder(FRect r){
      r.x -= 2; r.y -= 2;
      r.width += 2;
      r.height += 2;
   }
}
```

```
final class FRect{
   int x, y, width, height;
   public FRect(int x, int y, int width, int height){
      this.x = x; this.y = y;
      this.width = width;
      this.height = height;
   }
   boolean intersects(FRect r){
      FrameBorder.incDefBorder(r);
      return!((r.x + r.width <= x)||
        (r.y + r.height >= y)||
        (r.x >= x + width)||
        (r.y >= y + height));
   }
}
```

Abbildung 24: Java-Programm mit mehreren Klassen

Klassen (interprozedurale Analyse). Die bisher vorgestellten, statisch zur Übersetzungszeit durchgeführten Escape-Analysen basieren alle auf der Annahme, dass alle in dem zu untersuchenden Programm benutzten Klassen bekannt sind. Die verwendeten Techniken führen eine interprozedurale Analyse für Klassen durch, die zur Analysezeit zur Verfügung stehen und sich bis zur und während der Ausführung des Programmes in der JVM nicht mehr ändern. In der Realität verhalten sich Programme jedoch anders. Allein vom Zeitpunkt der Analyse bis zur Ausführung eines Programmes können sich Klassen und damit die untersuchten Ergebnisse ändern, indem einzelne Teile eines Programmes neu entwickelt und übersetzt werden. Die bereits durchgeführte Escape-Analyse berücksichtigt dies jedoch nicht und kann hierdurch zu fehlerhaften Optimierungen führen.

Beispiel

Abbildung 24 zeigt die Klasse `FrameCheck` zur Berechnung sich möglicherweise überlappender Fenster auf einem Desktop. Sie implementiert die Methode `checkIntersect()` zur Überprüfung, ob ein zweites neu gestaltetes Fenster ein bereits bestehendes Fenster überlagert. Das neue Fenster `lower` wird der Methode `intersects()` übergeben, die eine mögliche Überschneidung mit dem gegebenen Fenster `upper` berechnet. Eine statische Escape-Analyse könnte in diesem Beispiel nur verwertbare Ergebnisse für eine Optimierung liefern, wenn sich die Klassen `FRect` und `FrameBorder` bis zur Laufzeit des Programmes nicht mehr verändern würden. Das kann jedoch nicht garantiert werden, da bis zur Ausführung in der JVM neue Implementierungen dieser Klassen zur Verfügung stehen könnten. Zum Beispiel könnte sich die Klasse `FrameBorder` in der Hinsicht ändern, dass das übergebene Rechteck zu Verwaltungszwecken in einer externen Tabelle gespeichert wird. Damit würde das Objekt, auf das die Variable `lower` verweist, im Gegensatz zu dem Ergebnis einer auf der Produzentenseite durchgeführten Analyse aus der entsprechenden Methode entkommen.

Trennung der Analyse

Die Lösung des hier geschilderten Problems kann durch eine Trennung der Analyse in zwei separate Phasen erreicht werden, die sich in folgende Aufgabenbereiche gliedern:

> Phase 1: Die erste Phase beinhaltet eine vorbereitende, auf der Produzentenseite durchgeführte *intraprozedurale* Escape-Analyse. Zu diesem Zeitpunkt sind einzelne Klassen und deren Implementierungen bekannt, jedoch nicht die Interaktion zwischen verschiedenen Klassen. Die intraprozedurale Analyse beschränkt sich auf einzelne Klassen und die dort implementierten Methoden.
>
> Phase 2: Im zweiten Schritt der getrennten Analyse wird das Verhalten zwischen den verschiedenen Klassen untersucht. Die dynamische, auf der Konsumentenseite durchgeführte *interprozedurale* Escape-Analyse arbeitet mit den übermittelten Ergebnissen der intraprozeduralen Analyse. Zum Zeitpunkt der Ausführung eines Programmes sind der Konsumentenseite alle Klassen in ihrer aktuellen Implementierung bekannt. Nachdem eine Klasse geladen und vom Zwischencode in ein ausführbares Format übersetzt wurde, finden keine Änderungen mehr statt[9]. Ein JIT-Übersetzer könnte demnach ohne Gefahr die entsprechenden Optimierungen ausführen.

Bei der Aufteilung der zu verwendenden Escape-Analyse ist es wichtig festzustellen, an welchen Punkten die intraprozedurale in eine interprozedurale Analyse übergeht. Die Zwischenergebnisse der intraprozeduralen Analyse sind in einem Zustand festzuhalten und als Modell zu formulieren, welches mittels einer Annotation dem Zwischencode als Information hinzugefügt werden kann. Im weiteren Verlauf muss der Zustand der intraprozeduralen Analyse zur Laufzeit wieder hergestellt werden und auf Basis der ermittelten Zwischenergebnisse die interprozedurale Analyse gestartet werden, die an den zuvor definierten Punkten einsetzt.

Der folgende Abschnitt erläutert, wie eine Escape-Analyse in zwei Phasen zerlegt, die Ergebnisse der intraprozeduralen Analyse in einem Modell formuliert und mittels einer sicheren Annotationstechnik zur Konsumentenseite übertragen werden können. Weiterhin wird ein dynamisch auf der Konsumentenseite durchführbarer interprozeduraler Algorithmus vorgestellt, der auf den annotierten Ergebnissen der ersten Phase basiert und die Escape-Analyse effizient abschließt.

6.3 Escape-Annotationen in SafeTSA

6.3.1 Escape-Analyse und erweiterte Definition des Begriffes

Die in [61] vorgestellte Escape-Analyse von Whaley und Rinard bestimmt für jedes lokale Objekt und jeden formalen Parameter einer Methode, ob darauf zeigende Referenzen den Lebensraum der Methode verlassen (fliehen, escape) oder nicht (gebunden, bound). Es besitzen demnach alle lokalen Objekte und formale Parameter eine Escape-Eigenschaft *escape* (), die durch *escape* und *bound* bestimmt wird.

[9] Ausnahme: explizites Klassennachladen, wie in Kapitel 7.4 beschrieben.

Definition: Ein lokales Objekt oder formaler Parameter o *flieht* aus einer Methode m, genau dann wenn gilt: das Objekt ist über eine Referenz von außerhalb der Methode, nach Ablauf dieser Methode erreichbar. Jedes Objekt hat die Escape-Eigenschaft

$$escape_m(o) = \begin{cases} bound & \text{falls } o \text{ in } m \text{ nicht flieht} \\ escape & \text{sonst} \end{cases}$$

Mit der Aufteilung der Escape-Analyse in einen intra- und einen interprozeduralen Teil, die jeweils auf der Produzenten- und der Konsumentenseite ausgeführt werden, ergeben sich für die Definition der Escape-Eigenschaft neue Anforderungen. Während der intraprozeduralen Analyse werden Zusammenhänge, die über den Bereich der aktuell zu analysierenden Methode hinausgehen, nicht berücksichtigt. Insbesondere trifft das auf Objekte zu, die als Argument an andere Methoden weitergegeben werden und somit fliehen können. Diese Methoden sind zumeist nicht bekannt oder es kann nicht dafür garantiert werden, dass eine auf der Produzentenseite bekannte Methode in unveränderter Form auf der Konsumentenseite vorliegt. Obwohl ein Objekt, welches auf diese Art an eine fremde Methode übergeben wird, durchaus innerhalb der definierenden Methode gebunden sein kann, muss aufgrund fehlender Informationen immer die Escape-Eigenschaft *fliehend* angenommen werden.

Würden durch eine Annotation lediglich die Escape-Eigenschaft *fliehend* und *gebunden* zur Konsumentenseite übertragen, so müsste die interprozedurale Escape-Analyse unter erheblichem Aufwand alle fliehenden Objekte erneut untersuchen oder auf wertvolle Optimierungsmöglichkeiten verzichten. Eine Lösung für dieses Problem bietet die Erweiterung der Escape-Eigenschaft, welche alle Objekte beschreibt, die über die Weitergabe als Argument an andere Methoden fliehen können.

Definition: Gilt für ein lokales Objekt oder formalen Parameter o einer Methode m $escape_m(o) = escape$ genau deswegen, weil o an eine Methode g als Parameter übergeben wird und g ist zur Übersetzungszeit unbekannt oder kann nicht analysiert werden, dann erhält o die Escape-Eigenschaft $escape_m(o) = may$ und wird somit als möglicherweise fliehend gekennzeichnet. Die erweiterte Escape-Eigenschaft ergibt sich also zu:

$$escape_m(o) = \begin{cases} bound & \text{falls } o \text{ in } m \text{ nicht flieht} \\ may & \text{falls } o \text{ flieht möglicherweise in } m \\ escape & \text{sonst} \end{cases}$$

In dem verwendeten SafeTSA-System wird die so genannte *this*-Referenz[10] als Parameter an eine auf einem Objekt aufgerufene Methode übergeben. Das Verhalten von *this* innerhalb einer Methode kann also durch die Escape-Eigenschaft dieses speziellen Parameters beschrieben werdem. Betrachtet man das Beispiel aus Abbildung 25, so ist leicht zu erkennen, dass das Objekt *myObject* der Klasse B durch den Aufruf der Methode $foo()$ fliehen kann, wenn in $foo()$ entsprechend zur Laufzeit die Referenz auf *this* flieht, also gilt $escape_{foo}(this) = escape$. Für die Klasse A kann zur Laufzeit jedoch eine andere Implementierung vorliegen als diejenige, die auf dem Produzentensystem zum Zeitpunkt der Übersetzung bekannt war.

[10] Die Variable *this* zeigt auf die eigene Instanz eines Objektes.

```
class A{
  static java.util.Hashtable escapeList;

  public void foo(){
    escapeList = StaticList.getList();
    escapeList.put("AObject", this);
  }
}
```

```
class B extends A{
}

class Test{
  public void test(){
    B myObject = new B();
    myObject.foo();
  }
}
```

Abbildung 25: Fliehen durch Implizite Übergabe an Methode

Der für ein neu erzeugtes Objekt aufrufbare Konstruktor wird als eine spezielle Methode repräsentiert, an die ebenfalls eine Referenz auf das Objekt selbst übergeben wird (*this*), wodurch ein Objekt auch durch den Aufruf des Konstruktors fliehen kann. Im Unterschied zu anderen Methoden weist der Konstruktor eine zusätzliche Eigenschaft auf. Die Sprachdefinition von Java erfordert, dass sich Konstruktoren in der umgekehrten Reihenfolge der Vererbungshierarchie, nach einem fest definierten Schema aufrufen. Im Beispiel ruft also der Konstruktor der Klasse B den Default-Konstruktor der Klasse A auf. Die Aufrufe erfolgen bis zum Erreichen der Klasse `java.lang.Object`, welche die Superklasse aller Klassen darstellt. Offensichtlich kann auch hier der Fall auftreten, dass in einem Konstruktor einer beliebigen Klasse innerhalb der Vererbungshierarchie der *this* repräsentierende Parameter flieht und damit alle erzeugten Objekte aller abgeleiteten Klassen fliehen, für die der Konstruktor aufgerufen wurde. Demzufolge kann also jedes Objekt bereits durch den Aufruf eines Konstruktors fliehen, sofern das Objekt nicht eine Instanz von `java.lang.Object` ist oder alle in Frage kommenden Konstruktoren aller Superklassen bekannt sind.

Um eine effiziente Escape-Analyse in Bezug auf *this* für Methoden und Klassen (beim Aufruf von Konstruktoren) zu unterstützen werden folgende Escape-Eigenschaften definiert:

Definition: Eine Methode m besitzt die Escape-Eigenschaft *escape*, genau dann wenn gilt: eine Referenz auf das m aufrufende Objekt (*this*) flieht aus m. Flieht eine Referenz auf *this* ausschließlich aufgrund der Weitergabe an eine weitere Methode g oder ist g zu diesem Zeitpunkt nicht bekannt oder kann nicht analysiert werden, so wird m entsprechend als *may* bestimmt. Flieht *this* nicht, so erhält m die Escape-Eigenschaft *bound*.

$$escape\,(m) = \begin{cases} bound & \text{falls } this \text{ in } m \text{ nicht flieht} \\ may & \text{falls } this \text{ flieht möglicherweise in } m \\ escape & \text{sonst} \end{cases}$$

6.3.2 Erweitertes SafeTSA-Maschinenmodell

Die grundlegende Idee der hier vorgestellten Annotationstechnik ist die Integration von Escape-Eigenschaften in das Maschinenmodell und die Operationen von SafeTSA und somit in die Zwischencoderepräsentation zur Übertragung. Ausgangspunkt für die Integration ist die Typtrennung, welche durch die Trennung der Registersätze erreicht wird und

die folgende Typsicherheit eines mittels des SafeTSA-Formats übertragenen Programmes. Indem die Escape-Eigenschaft eines Objektes oder formalen Parameters als übergeordneter Typ formuliert wird, kann SafeTSA mit einer entsprechenden Erweiterung die sichere Übertragung dieses Typs garantieren und somit die sichere Übertragung der Ergebnisse einer Escape-Analyse in Form von Escape-Eigenschaften.

Da in einem Quellprogramm jedem Objekt bereits ein entsprechender Typ zugeordnet ist, muss das Maschinenmodell von SafeTSA um eine neue Dimension von Typen erweitert werden, welche orthogonal zu den bestehenden Typen angeordnet werden. Die hinzuzufügende Dimension definiert genau drei verschiedene Typen, welche den definierten Escape-Eigenschaften entsprechen (*escape*, *may*, *bound*). Für jeden Typ der neuen Dimension existiert wiederum ein getrennter Registersatz, der nur Werte dieses Typs bzw. dieser Escape-Eigenschaft enthalten kann. Für jeden Referenztyp R existieren also drei mögliche Typen der zusätzlichen Dimension, welche die Escape-Eigenschaft des R zugehörigen Objektes repräsentieren - R_{escape}, R_{may} und R_{bound}. Diese drei Typen werden als *Escape-Typen* bezeichnet. Abbildung 26 zeigt das erweiterte Maschinenmodell und die Registersätze der neuen Dimension.

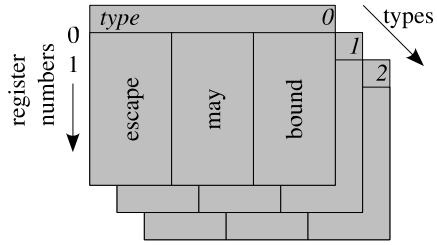


Abbildung 26: Erweitertes SafeTSA-Maschinenmodell

Die Erweiterung des Maschinenmodells, welche mit der Einführung sicherer und unsicherer Typen verglichen werden kann, verletzt nicht die Typsicherheit eines Programmes. Insbesondere können die Escape-Typen durch eine Erweiterung der SafeTSA-Operationen sicher von der Produzenten- zur Konsumentenseite transportiert und dort durch einen JIT-Übersetzer verifiziert werden.

6.3.3 Modifizierte SafeTSA-Operationen

Mit der Erweiterung des SafeTSA-Maschinenmodells müssen einige Operationen (Instruktionen) des SafeTSA-Instruktionssatzes ebenfalls erweitert und angepasst werden. Die Modifizierung der Operationen erfolgt so, dass die sichere Übertragung der Escape-Eigenschaft und somit die Richtigkeit der auf der Produzentenseite ermittelten intraprozeduralen Analyseergebnisse garantiert werden können. Im Anschluss werden notwendige Modifikationen und die erweiterten Operationen vorgestellt.

Rückgabewerte Referenziert eine Variable ein lokales Objekt o in der Methode m und es gilt $escape_m(o) = bound$, so ist o und somit diese Variable von der Verwendung als Rückgabewert von m ausgeschlossen, da o sonst fliehen würde. Aus diesem Grund muss die *return*-Instruktion in der Art modifiziert werden, dass sie ihre Operanden ausschließlich aus solchen Registersätzen bezieht, die dem Typ und der Escape-Eigenschaft *escape*

entsprechen. Somit ist garantiert, dass niemals ein Operand von dieser Instruktion als Rückgabewert zurückgeliefert wird, der mit einem gebundenen oder möglicherweise fliehenden Objekt korrespondiert.

Registersätze Eine Instruktion des SafeTSA-Instruktionssatzes bezieht ihre Operanden implizit von den SSA-Variablen der korrespondierenden Registersätze und speichert ihr Ergebnis automatisch in dem nächsten, frei verfügbaren Register des dem Typ dieser Variable entsprechenden Registersatzes ab. Diese Verfahrensweise gilt auch für das erweiterte Maschinenmodell, in welchem aber durch Einfügen einer zusätzlichen Dimension einige Instruktionen gleichermaßen auf Referenztypen mit unterschiedlichen Escape-Eigenschaften operieren. Die zu erweiternden Instruktionen benötigen einen zusätzlichen Modifikator, welcher den Typ hinsichtlich der Escape-Eigenschaften *may*, *bound* und *escape* kennzeichnet und anweist, aus welchem Registersatz die entsprechenden Operanden zu beziehen sind. Ist kein Modifikator angegeben, handelt es sich also um eine nicht-modifizierte Instruktion, so dass diese Instruktion ihre Operanden ausschließlich aus dem Registersatz für Typen der Escape-Eigenschaft *escape* bezieht.

Weiterhin ist es für Instruktionen nicht möglich, Werte zwischen zwei Registersätzen hinsichtlich des Escape-Typs zu kopieren. Es ist also z.B. ausgeschlossen, den Inhalt eines Registers dessen Registersatz der Escape-Eigenschaft *bound* entspricht, in ein Register zu kopieren, dessen Registersatz mit dem Typ *escape* verbunden ist. Tabelle 2 zeigt die gültigen und ungültigen Kopieroperationen.

Quell-/Zielregistersatz	bound	may	escape
bound	x	-	-
may	-	x	-
escape	-	-	x

Tabelle 2: Mögliche Kopieroperationen für SafeTSA-Instruktionen (z.B. *xupcast*)

Parameterübergabe Da eine Methode in einem Programm mehrere Aufrufstellen haben kann (*call sites*), ist die Escape-Eigenschaft des aktuellen Parameters einer Aufrufstelle nicht immer identisch mit der des formalen Parameters dieser Methode. In dem erweiterten Maschinenmodell muss jedoch garantiert werden, dass bei der Verwendung eines Parameters immer auf den korrekten Registersatz zugegriffen wird, also jenen Registersatz, der die Werte entsprechend der Escape-Eigenschaft des jeweiligen aktuellen Parameters speichert. Die in SafeTSA für die Repräsentation der aktuellen Parameter verwendeten Operandenlisten einer Aufrufstelle (Instruktionen *xcall* und *xdispatch*) werden aus diesem Grund um zusätzliche Modifikatoren erweitert. Ein Modifikator bezeichnet die Escape-Eigenschaft des entsprechenden aktuellen Parameters und wird dem jeweiligen Operanden hinzugefügt. Somit ergeben sich die Operandenlisten zu Listen von Paaren der Form (op, mod), wobei *op* für den Operanden des an die Methode übergebenen aktuellen Parameters steht und *mod* den Registersatz hinsichtlich des Escape-Typs bezeichnet, aus dem dieser Operand bezogen wird.

Erweiterte Operationen Sei E die Menge der Escape-Eigenschaften {*may, bound, escape*}. Im Folgenden sind die durch die Erweiterung betroffenen Operationen in einer Übersicht dargestellt:

new: Die *new*-Operation steht in SafeTSA für die Erzeugung eines neuen Objektes aus der angegebenen Klasse *class*. Das Objekt wird in dem nächsten freien Register des sicheren Registersatzes des dem Objekt entsprechenden Typs abgespeichert und anschließend der mit *con* bezeichnete Konstruktor aufgerufen. Um die Escape-Eigenschaft des durch *new* erzeugten Objektes und somit den zu verwendenden Registersatz hinsichtlich des Escape-Typs zu kennzeichnen, wird der Operation ein Modifikator *mod* hinzugefügt. Weiterhin werden $op_1 ... op_n$, welche die Register der Operandenwerte der an den Konstruktor *con* zu übergebenden aktuellen Parameter kennzeichnen, hier als Liste von Paaren der Form (op, mod) dargestellt.

$$\textbf{new}\ class\ con\ (op_1, mod_1) \ldots (op_n, mod_n)\ mod, \quad \begin{array}{l} mod,\ mod_i \in E \\ class,\ con \in Symbol \end{array}$$

xupcast, downcast: Die Operationen *xupcast* und *downcast* können Werte von einem Registersatz (*src*) zu einem anderen Registersatz (*target*) kopieren. So wird *xupcast*, verbunden mit einer Überprüfung, auch zum Kopieren von Werten eines *unsicheren* in einen *sicheren* Registersatz verwendet (Nullreferenzüberprüfung). In dem erweiterten Modell werden diese Operationen hinsichtlich des Escape-Typs beschränkt, indem das Kopieren nur innerhalb eines Escape-Typs erlaubt ist. Somit kann eine *cast*-Operation zwar Werte zwischen den Registersätzen *may* und *safe-may* kopieren, nicht jedoch zwischen *may* und *safe-bound*.

$$\begin{array}{ll} \textbf{xupcast}\ src\ target\ object\ mod, & mod \in E \\ \textbf{downcast}\ src\ target\ object\ mod, & mod \in E \end{array}$$

getfield, setfield, getelem, setelem: Die Operationen für Speicherzugriffe beziehen ihre Operanden aus dem sicheren Registersatz (*safe-ref*, *safe-arr*), der dem in *ref* bzw. *arr* bezeichneten Typ entspricht. Die Erweiterung des Maschinenmodells macht es erforderlich, dass diese Operationen den Registersatz auch hinsichtlich des Escape-Typs bestimmen, damit der Operand aus dem richtigen Register bezogen werden kann. Deshalb werden die Operationen mit einem Modifikator *mod* erweitert, der den Registersatz bezüglich des Escape-Typs festlegt. Die durch *value* bezeichneten Register können Werte ausschließlich aus einem Registersatz beziehen, der dem Escape-Typ *escape* entspricht. Das ergibt sich aus der Bedeutung dieser Speicheroperationen hinsichtlich der Escape-Analyse, da Objekte generell durch eine entsprechende Speicherung fliehen.

$$\begin{array}{ll} \textbf{getfield}\ ref\ object\ field\ mod, & mod \in E \\ \textbf{setfield}\ ref\ object\ field\ value\ mod, & mod \in E \\ \textbf{getelem}\ arr\ object\ index\ mod, & mod \in E \\ \textbf{setelem}\ arr\ object\ index\ value\ mod, & mod \in E \end{array}$$

xdispatch, xcall: Die Operationen für Methodenaufrufe beziehen den Operanden *rec*, der das Empfängerobjekt repräsentiert, aus dem entsprechenden Registersatz

zu dem in *base* angegebenen Typ. In dem erweiterten Modell wird analog zu den Speicheroperationen ein Modifikator *mod* eingeführt, der den Registersatz hinsichtlich des Escape-Typs bezeichnet. Weiterhin werden $op_1 ... op_n$, welche die Register der Operandenwerte der an die Methode *meth* zu übergebenden aktuellen Parameter kennzeichnen, in diesen Operationen als Liste von Paaren der Form (op, mod) dargestellt.

xdispatch *base rec method* $(op_i,\ mod_i)\, mod$, $\quad mod,\ mod_i \in E$
xcall *base rec method* $(op_i,\ mod_i)\, mod$, $\quad mod,\ mod_i \in E$

phi: Die phi-Operation ist auf einen Typ beschränkt, der ebenfalls in Bezug auf den Escape-Typ mit einem Modifikator *mod* gekennzeichnet wird. Der für den Bezug des Operanden und das Ergebnis der Operation zu verwendende Registersatz bestimmt sich aus dem Modifikator, wobei für alle Operanden und das Ergebnis der gleiche Escape-Typ gilt:

phi *type* $op_1 ... op_n$ *mod*, $\quad mod \in E$

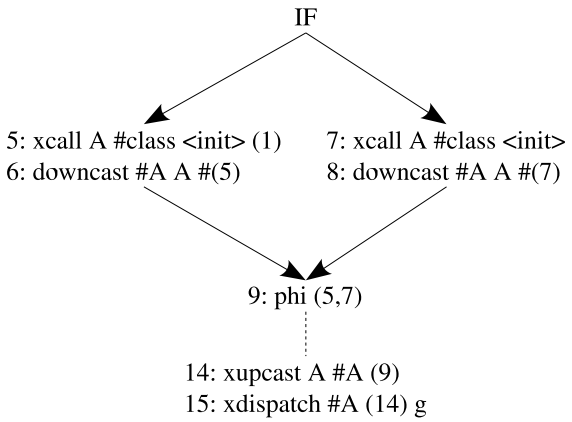
6.4 Interprozedurale Escape-Analyse zur Laufzeit

In den vorhergehenden Abschnitten wurde eine Technik zur Aufspaltung der Escape-Analyse vorgestellt, mit dem Ziel, bestimmte dynamische Eigenschaften der Sprache Java und der JVM zu unterstützen. Weiterhin wurde dargestellt, wie auf Basis der Zwischencoderepräsentation SafeTSA Ergebnisse einer intraprozeduralen Escape-Analyse auf der Produzentenseite sicher annotiert werden können. Im folgenden Schritt wird eine schnelle interprozedurale Escape-Analyse beschrieben, die auf der Konsumentenseite (z.B. einem JIT-Übersetzer) Anwendung findet und auf den annotierten Ergebnissen der intraprozeduralen Analyse basiert.

Durch die vorgestellte Annotationstechnik können die Zwischenergebnisse der intraprozeduralen Analyse auf der Konsumentenseite sicher wieder hergestellt werden. Die Escape-Eigenschaft *may* beschreibt hierbei jene Objekte, deren Verhalten bezüglich der Weitergabe als Parameter an Methodenaufrufe zu überprüfen ist, über die sie möglicherweise fliehen können. Objekte mit den Escape-Eigenschaften *bound* und *escape* bedürfen keiner weiteren Untersuchung, da deren Verhalten bereits bekannt ist. Die verwendete interprozedurale Analyse beschränkt sich somit ausschließlich auf möglicherweise fliehende Objekte und ermittelt, ob diese Objekte durch die Weitergabe an eine Methode tatsächlich fliehen.

Für ein möglicherweise fliehendes lokales Objekt oder Parameter o mit der umgebenden Methode m und $escape_m(o) = may$ ergibt sich die Escape-Eigenschaft $escape_m(o)$ nach der interprozeduralen Escape-Analyse zu:

$$escape_m(o) := \begin{cases} bound & \text{falls } o \text{ wird nur in weiteren Methodenaufrufen } g \text{ verwendet,} \\ & \text{die eindeutig zuordbar sind und deren korrespondierende} \\ & \text{formale Parameter nach der interprozeduralen Analyse die} \\ & \text{Escape-Eigenschaft } bound \text{ aufweisen} \\ escape & \text{sonst.} \end{cases}$$




```
public class A{
    static boolean b=true;
    public A(){
    }
    public A(int i){
        super(i);
    }
    public void foo(){
        A a;
        if (b){
            a = new A();
        }else{
            a = new A(1);
        }
        if (a != null)
            a.g();
    }
    public void g(){
    }
}
```

Abbildung 27: Indirekter Zugriff über Werte in SafeTSA

Interprozedurale Analyse

Der folgende Algorithmus zur Berechnung von Escape-Informationen unter Einbeziehung interprozeduraler Zusammenhänge ergänzt die auf der Produzentenseite durchgeführte intraprozedurale Escape-Analyse. Er wird auf der Konsumentenseite eines SafeTSA-Systems angewendet, welche die Ergebnisse einer intraprozeduralen Analyse zur Verfügung stellt, in diesem Fall auf Basis der Annotation. Der Aufruf des Algorithmus erfolgt über die Prozedur *checkMethod*, die eine zu untersuchende Methode m im SafeTSA-Format als Eingabe erhält und für jede zu übersetzende Methode aufgerufen wird. Die Methode wird zu diesem Zeitpunkt durch einen Syntaxbaum mit Basisblöcken und Instruktionen dargestellt, der durch *checkMethod* mittels einer Tiefensuche in einem Durchlauf traversiert wird.

Bei der Berechnung der endgültigen Escape-Informationen müssen jedoch die besonderen Eigenschaften der SSA- bzw. SafeTSA-Darstellung berücksichtigt werden. Da in der vorliegenden Programmrepräsentation keine Variablen mehr existieren, sondern nur noch über die Verwendung von Werten zugegriffen wird, kann eine Escape-Eigenschaft nicht ohne weiteres einem bestimmten Objekt zugeordnet werden. Vielmehr müssen während der Analyse alle Aliase eines Wertes, der für die Erzeugung eines Objektes steht, identifiziert und entsprechend behandelt werden. Abbildung 27 zeigt ein Beispielprogramm und die schematische Darstellung in SafeTSA, wo das Objekt `a` in Abhängigkeit einer Bedingung erzeugt wird ((5) und (7)). Es könnte nun sein, dass die neu erzeugte Instanz über einen der beiden Konstruktoren aus der Methode `foo` flieht. Bei der späteren Verwendung des Objektes (15) wird aber nicht auf die Werte aus (5) und (7) direkt zugegriffen, da der zu verwendende Wert der erzeugenden Instruktion erst durch die Phi-Instruktion (9) bestimmt wird. Weiterhin kann ein Wert erst Operand einer Cast-Operation sein, z.B. indem er mit einer Nullreferenzüberprüfung von einem unsicheren in einen sicheren

Registersatz kopiert wird (14). Entsprechend müssen die Ergebniswerte und die Operanden der Phi-Instruktion sowie der Ergebniswert der Cast-Operation bei der Bestimmung der Escape-Informationen mit berücksichtigt werden. Das geschieht, indem während der Untersuchung eines Programmes alle Aliase eines Wertes bestimmt werden und diese hinsichtlich der Escape-Analyse gleich behandelt werden. Im Beispiel sind (5),(6),(7),(8),(9) und (14) Aliase und haben die gleiche Escape-Eigenschaft.

Definition: Sei $ref \in R$ ein Referenzwert aus der Menge der Referenzwerte in der SafeTSA-Darstellung eines Programmes. Dann ist die Relation $alias : R \rightarrow R$ wie folgt definiert:

$$alias\,(ref_1, ref_2) \Longleftrightarrow ref_1 \text{ ist ein Alias von } ref_2.$$

Definition: Sei $ref \in R$ ein Referenzwert. Dann bezeichnet $\langle ref \rangle$ die Menge aller Aliase von ref. $\langle ref \rangle$ ist eine Äquivalenzklasse bezüglich der Relation $alias$ und zerlegt die Menge R.

$$\langle ref \rangle := \{ref_1 ... ref_n\} \text{ mit } alias\,(ref_i, ref_j) \text{ für alle i,j} \in \{1..n\}$$

Während der interprozeduralen Analyse werden die Äquivalenzklassen beim Durchlaufen der SafeTSA-Programmdarstellung dynamisch aufgebaut. Weiterhin werden nur die Äquivalenzklassen möglichweise fliehender Werte gebildet, welche letzlich für die Analyse von Bedeutung und zu untersuchen sind. Bestimmt die Analyse die Escape-Eigenschaft der Werte, so muss immer die Aliasmenge betrachtet werden, denn wenn während der Ausführung auch nur ein Alias aus einer Methode flieht, so kann das referenzierte Objekt ebenfalls fliehen. Folglich ist die Escape-Eigenschaft für die Aliasmenge zu erweitern.

Definition: Sei $\langle ref \rangle$ die Aliasmenge zu ref und *object* das durch ref referenzierte Objekt. Dann bezeichnet $escape\,(\langle ref \rangle)$ die Escape-Eigenschaft der Aliasmenge, welche sich durch die Escape-Eigenschaft zu jedem ref_i, welches in der Äquivalenzklasse $\langle ref \rangle$ enthalten ist, ergibt.

Da ein Objekt fliehen kann, wenn mindestens eine Referenz auf dieses Objekt flieht, gilt für die Escape-Eigenschaft der Aliasmenge:

$$escape\,(\langle ref \rangle) = \begin{cases} bound & \text{falls } \forall\ ref_i \in \langle ref \rangle \text{ mit } i = 1...n : escape\,(ref_i) = bound \\ escape & \text{sonst} \end{cases}$$

Die Escape-Eigenschaft des referenzierten Objektes *object* entspricht daher der Escape-Eigenschaft der Äquivalenzklasse $\langle ref \rangle$:

$$escape\,(object) = escape\,(\langle ref \rangle)\,.$$

Für den dynamischen Aufbau der Äquivalenzklassen wird eine Funktion *updateAlias* wie folgt eingeführt:

Definition: Für eine Äquivalenzklasse $\langle ref \rangle$ und einen Referenzwert ref_1 fügt die Funktion $updateAlias\,(\langle ref \rangle, ref_1)$ den Referenzwert ref_1 zu der Äquivalenzklasse $\langle ref \rangle$ hinzu und notiert für jedes $ref_i, ref_i \in \langle ref \rangle$, ref_1 ist ein Alias von ref_i.

Für Konstruktoren wird analog die Escape-Eigenschaft festgelegt, wobei diese auf eine Menge von Konstruktoren erweitert wird.

Definition: Bezeichne CON die Menge der Konstruktoren $con_i, i = 1...n$, die für ein zu erzeugendes Objekt *object* aufgerufen werden, beginnend mit dem Konstruktor con_1 (der *object* zugehörigen Klasse) entsprechend der umgekehrten Vererbungshierarchie bis zu con_n (dem Konstruktor der Klasse `java.lang.Object`).

Da ein Objekt fliehen wird, wenn ein beliebiger in der Vererbungshierarchie aufgerufener Konstruktur einer Superklasse flieht, gilt für die Escape-Eigenschaft von CON:

$$escape\,(CON) = \begin{cases} bound & \text{falls } \forall\ con_i \text{ mit } i = 1...n : escape\,(con_i) = bound \\ escape & \text{sonst} \end{cases}$$

Sei nun M die Menge der zu untersuchenden Objekte und Parameter, die möglicherweise fliehen, *this* eine Referenz auf das Empfängerobjekt der zu untersuchenden Methode m und $p_i \in P$, $i = 1...n$ die formalen Parameter der Methode. Bezeichne $inst_{current}$ eine Referenz auf die aktuell untersuchte Instruktion und sei *this* durch den nullten Parameter p_0 repräsentiert.

procedure checkMethod
Eingabe: zu untersuchende Methode m
M:=$\emptyset$

```
begin checkMethod
   M = {p_i | p_i ∈ P and escape(p_i) = may}
   foreach p ∈ M
      set escape(p) = bound
   end for


   foreach inst_current in Tiefensuche von m
      switch(inst_current)
         case(xupcast, downcast)
            ref = xupcast src target object mod
            if (⟨object⟩ ∩ M ≠ ∅)
               updateAlias (⟨object⟩, ref)
               M = {ref} ∪ M
            end if
```

```
            case(phi)
                ref = phi type op_1...op_n mod
                if (⋃_{i=1..n} ⟨op_i⟩ ∩ M ≠ ∅)
                    ∀i = 1...n updateAlias (⟨op_i⟩, ref)
                    M = {ref} ∪ M \{op_1...op_n}
                end if
            case(new, xdispatch, xcall)
                if (inst_current = new)
                    //ref = new class con (op_1, mod_1) ... (op_n, mod_n) mod
                    if (escape(ref) = may)
                        checkMethod(CON)
                        if (escape(CON) = bound)
                            set escape(ref) = bound
                            M = {ref} ∪ M
                        end if
                    end if
                end if
                if(inst_current = xdispatch || inst_current = xcall)
                    // xdispatch base rec meth (op_i, mod_i) mod
                    if (⟨base⟩ ∩ M ≠ ∅)
                        if (escape(meth)= may)
                            checkMethod(meth)}endif
                        if (escape(meth)= escape)
                            set escape(⟨base⟩) = escape
                            M = M \⟨base⟩
                        end if
                    end if
                end if
                for each op_i with ⟨op_i⟩ ∩ M ≠ ∅
                    let p̂_i be the corresponding formal parameter of meth
                    if (escape_meth(p̂_i)= may)
                        checkMethod(meth)
                    if (escape_meth(p̂_i)= escape)
                        set escape(⟨op_i⟩) = escape_meth(p̂_i)
                        M = M \⟨op_i⟩}
                    end if
                end for
        end switch
    end for
    escape(m) := escape(⟨p_0⟩)
end checkMethod
```

Im Verlauf des Algorithmus wird allen möglichweise fliehenden formalen Parametern $p_1...p_n$ der Methode m nach Aufruf von $checkMethod\,(m)$ entweder die Escape-Eigenschaft *escape* oder *bound* zugeordnet (Zwischenspeicherung). Weiterhin ist $escape\,(m) = escape$

oder $escape\,(m) = bound$ (entsprechend der Escape-Eigenschaft von $this$).

Der Algorithmus arbeitet rekursiv für alle weiteren Aufrufstellen von Methoden in der zu untersuchenden Methode, an die möglicherweise fliehende Objekte als aktuelle Parameter übergeben werden und deren korrespondierende formale Parameter die Escape-Eigenschaft *may* besitzen. Um eine korrekte Arbeitsweise von $checkMethod()$ auch bei im Quellprogramm vorkommender Rekursion zu gewährleisten, wird ein Kontrollmechanismus eingeführt.

Sei $R = \{m_0 ... m_n\}$ die Menge der aktuell in Untersuchung befindlichen Methoden. Für jede neue zu analysierende Methode $m_i,\ i \in N$ wird geprüft, ob die Methode bereits in R enthalten ist ($m_i \in R$), also analysiert wird. In diesem Fall liegt möglicherweise eine Rekursion vor und $checkMethod\,(m)$ bricht die aktuelle Untersuchung ab, wobei alle Elemente aus M, die der aktuell untersuchten Methode m zugeordnet werden können, die Escape-Eigenschaft *escape* erhalten. Ist m_i noch nicht in R enthalten, dann wird diese Methode hinzugefügt ($\{m_i\} \cup R$).

Beispiel

Bezogen auf das zuvor genannte Beispiel der Klasse `FrameCheck` arbeitet der Algorithmus wie folgt (Konstruktoren CON besitzen die Escape-Eigenschaft *bound*).

Die Methode `checkIntersect` definiert eine Variable `lower` mit $escape\,(lower) = may$. Das durch `lower` referenzierte Objekt könnte möglicherweise aus der Methode fliehen. Die Analyse stellt fest, dass `lower` als Argument an die Methode `intersects` der Klasse `FRect` übergeben wird. Steht die Klasse `FRect` zur Verfügung[11], so kann festgestellt werden, dass der formale Parameter von `intersects` nicht innerhalb der Methode gebunden ist ($escape\,(r) \neq bound$). Die Methode muss also untersucht werden und es stellt sich heraus, dass das Verhalten der Methode `incDefBorder` der Klasse `FrameBorders` entscheidend ist. Steht diese Klasse zur Verfügung, so ergibt sich für den formalen Parameter von `incDefBorder` die Escape-Eigenschaft $escape\,(r) = bound$, er flieht also nicht aus der Methode. Als Folgerung verzeichnet die Analyse, dass der formale Parameter der Methode `intersects` ebenfalls nicht flieht.[12] Weiterhin kann gefolgert werden, dass das an die Methode `intersects` übergebene und durch `lower` referenzierte Objekt nicht fliehen wird.

[11] Eine Klasse steht zur Verfügung, wenn sie bereits geladen ist. Ist das nicht der Fall, so muss davon ausgegangen werden, dass die entsprechenden formalen Parameter fliehen. In einem erweiterten System können Klassen bei Bedarf geladen werden, ohne diese jedoch zu übersetzen.

[12] An dieser Stelle kann die Escape-Eigenschaft dieses formalen Parameters zwischengespeichert werden.

7 Java-spezifische Eigenschaften

Wie zu Beginn erwähnt wurde, besitzt die Programmiersprache Java Eigenschaften, die das Verhalten eines Programmes hinsichtlich einer Escape-Analyse nachhaltig beeinflussen, jedoch erst auf der Konsumentenseite abschließend untersucht werden können. In diesem Kapitel werden die Eigenschaften ausführlich vorgestellt und der Einfluss auf die Durchführung der Escape-Analyse und einer anschließenden Optimierung herausgearbeitet. Betrachtet werden die folgenden Eigenschaften hinsichtlich der vorgestellten Escape-Analyse:[13]

- Reflektion
- Dynamisches Binden/Laden von Klassen
- Java native Schnittstelle (*JNI*)

7.1 Reflektion

Java bietet durch den Mechanismus der Reflektion die Möglichkeit, dynamisch zur Laufzeit eines Programmes einen Blick auf das Aussehen bereits geladener Klassen zu werfen. Prinzipiell besteht die zugehörige Klassenbibliothek (`Reflection API`) aus Komponenten für Objekte, die bestimmte Teile einer Klasse repräsentieren und Mechanismen, die das Arbeiten mit diesen Objekten auf eine sichere Art und Weise erlauben. Der Sicherheitsaspekt berücksichtigt alle Einschränkungen definierter Klassen bezüglich der Zugriffsberechtigungen, so dass durch die Reflektion keine so genannten unerlaubten Zugriffe (*Backdoors*) entstehen, mittels derer auf geschützte Informationen zugegriffen werden könnte. Grundlage für die Klassenkomponenten bildet die Klasse `Class`, die einen universellen Typ für die ein Objekt beschreibenden Meta-Informationen repräsentiert.

Über das so genannte Klassenobjekt einer geladenen Klasse, welches auch von jedem instanziierten Objekt dieser Klasse referenziert werden kann, ist ein Programm in der Lage, weitere Informationen über die Klasse zu beziehen. Die Meta-Informationen beinhalten öffentlich sichtbare Felder, Konstruktoren sowie statische und Instanzmethoden. Die Repräsentation dieser Daten erfolgt durch Objekte der Reflektionskomponenten `Field`, `Constructor` und `Method`.

Neben der Beschaffung von Meta-Informationen zu einer Klasse, können diese auch dazu verwendet werden, um mit Klassen unbekannten Typs zu arbeiten.[14] So ist es möglich, neue Objekte implizit oder explizit geladener Klassen zu instanziieren, Inhalte von Feldern zu manipulieren und Methoden auf instanziierten Objekten aufzurufen. Die `Reflection API` stellt für diese Techniken Methoden wie `newInstance()` und `invoke()` zur Verfügung.

Weiterhin bietet die Reflektion eine Möglichkeit, Klassen in Java erst während der Laufzeit eines Programmes explizit zu laden. Mit dieser Technik können Java Programme dynamisch erweitert werden, indem eine der in `java.lang.Class (forName())` oder `java.lang.ClassLoader (loadClass())` implementierten Methoden unter Angabe des

[13] Die hier vorgestellten Techniken beziehen sich auf die Implementierung der *JikesRVM*, für andere virtuelle Maschinen sei auf deren Spezifikation verwiesen. Die JikesRVM wird in Kapitel 8.3 im Detail vorgestellt.

[14] In diesem Zusammenhang *unbekannt* auf der Produzentenseite.

Namens des zu ladenden Typs aufgerufen wird. Ein Beispiel wäre z.B. ein in Java implementierter Webbrowser, in welchem durch spezielle HTML-Tags spezifizierte Applets von einer beliebigen Netzwerkadresse in das Programm geladen und daraufhin ausgeführt werden.

Während `loadClass()` Teil eines benutzerspezifischen Klassenladers ist, findet `forName()` den häufigeren Gebrauch bei der Programmierung entsprechender Applikationen (z.B. Arbeiten mit Datenbanken, wo der Datenbanktreiber mittels `forName()` geladen wird). Diese Methode liefert außerdem einen Typ zurück, der definitiv gebunden und initialisiert ist, was bei `loadClass()` nicht unbedingt der Fall sein muss.

Die JikesRVM unterstützt vollständig die Mechanismen der Reflektion.

7.2 Dynamisches Klassenladen

Java-Applikationen können mit entsprechenden benutzerdefinierten Klassenladern (*Class Loader*) Klassen und Schnittstellen laden, bzw. dynamisch binden, die zur Zeit der Erstellung des Programmes nicht bekannt sind oder noch nicht existieren. Verantwortlich für den Umgang der JVM mit zu ladenden Klassen ist das *Java Linking Model*, welches durch den Prozess der Klassenauflösung (*Class Resolution*) bestimmt wird.

Beim Erzeugen einer Java-Applikation werden für jede Klasse und Schnittstelle entsprechende Dateien (*Class Files*) einer Zwischencoderepräsentation erzeugt, z.B. Bytecode- oder SafeTSA-Dateien, eingeschlossen innere und anonyme Klassen. Obwohl die Klassen selbst unabhängig vorliegen, enthalten sie symbolische Verbindungen untereinander und zu den Klassen der Java API. Bei der Ausführung einer Applikation durch die JVM werden dann alle symbolischen Verbindungen zusammengeführt. Diesen Prozess nennt man dynamisches Binden (*Dynamic Linking*).

Die symbolischen Referenzen einer Klasse werden im Konstantenpool (*Constant Pool*) bei Bytecode-Dateien beziehungsweise in der Symboltabelle (*Symbol Table*) von SafeTSA-Dateien abgelegt, welche für jede Klasse unabhängig existieren und weiterhin Informationen zu Typen, Klassen, Methoden, Feldern und Konstanten enthalten. Während der Übersetzung werden diese in SafeTSA durch die Symboltabelle repräsentierten Informationen trotz unterschiedlicher Übersetzungsmethode in der JVM auf die gleiche Art verwaltet, wie die im Bytecode enthaltenen Daten. Auch werden die Referenzen auf die vollständig übersetzten Methoden an derselben Stelle gespeichert, wodurch eine Unterscheidung von unterschiedlich übersetzten Methoden nach Übersetzung nicht erforderlich ist.

Für jede geladene Klasse und Schnittstelle legt die JVM diese Informationen in Tabellen ab, die das Modell des Laufzeit-Konstantenpools (*Runtime Constant Pool*) verkörpern. Auf der einen Seite werden Referenzen auf Instanzfelder und virtuelle Methoden im *TIB* (*Type Information Block*) abgelegt, der für jede Klasse (jeden Typ) existiert. Die *JTOC* (*JikesRVM Table of Contents*) hingegen enthält unter anderem die Referenzen auf alle statischen Felder und Methoden sowie eine Referenz auf jeden im System vorkommenden TIB. Die JTOC und der TIB werden für die Auflösung (*Resolve*) von dynamischen Referenzen verwendet.

Während der Ausführung des Programmes muss eine spezielle symbolische Referenz, bevor sie verwendet werden kann, aufgelöst werden. Mit Auflösung wird der Prozess des Suchens einer Entität entsprechend der Identifikation durch die symbolische Referenz und der Austausch der symbolischen gegen eine direkte Referenz bezeichnet. Der Vorgang wird

auch *Constant Pool Resolution* genannt, da jeder symbolische Link auf ein bestimmtes Element im Konstantenpool (TIB oder JTOC) verweist.

Bei der verzögerten Übersetzung von Methoden in der JikesRVM kann es vorkommen, dass nicht alle Methoden zu Beginn des Programmes zur Laufzeit übersetzt werden, sondern die Übersetzung erst beim tatsächlichen Aufruf erfolgt (*Lazy Method Compilation*). In diesem Fall enthalten die Tabellen keine Referenzen auf die übersetzten Methoden, sondern auf einen so genannten *Lazy Method Invocation Stub*. Wird eine symbolische Referenz auf einen solchen Eintrag aufgelöst, veranlasst der Übersetzer zuerst die Übersetzung der Methode und dann wird der Eintrag der Referenz entsprechend aktualisiert. Der Vorgang entspricht dem Modell der *späten Auflösung* (*Late Resolution*).[15]

Alle Instruktionen, die eine symbolische Referenz benutzen, müssen einen entsprechenden Eintrag im Konstantenpool spezifizieren. So verweist die Operation `getfield` über die Werte *field* und *ref* auf Namen und Typ des zu ladenden Feldes sowie die hierfür benötigte Klasse. Die Auflösung für einen Eintrag im Konstantenpool erfolgt jeweils jedoch nur einmal. Das impliziert für alle folgenden Instruktionen die Verwendung der durch das Auflösen ermittelten direkten Referenz. Weiterhin werden beim Auflösen verschiedene Zugriffs- und Korrektheitsüberprüfungen durchgeführt.

7.3 JNI - Java Native Interface

Die native Schnittstelle von Java (`Java Native Interface - JNI`) stellt eine Programmierschnittstelle dar, mit deren Hilfe ein in der JVM laufendes Programm mit Anwendungen und Bibliotheken interagieren kann, die in plattformabhängigen Programmiersprachen geschrieben wurden (z.B. `C++`). Die Schnittstelle ist dann sehr hilfreich, wenn eine Anwendung nicht vollständig in Java implementiert werden kann, z.B. weil auf spezielle Funktionen der Zielplattform nicht durch die Java Klassenbibliotheken zugegriffen werden kann. Weiterhin eröffnet die JNI Möglichkeiten, bereits implementierte Teile einer Anwendung mit zu nutzen, sofern das auch hinsichtlich Funktion und Architektur machbar ist. Die Schnittstelle bietet sowohl die Möglichkeit für den Austausch von Informationen von Java zu nativem Code als auch in der Gegenrichtung von nativem Code zu Java (*native* Methoden).

Hierbei kann eine native Methode (in einer plattformabhängigen Sprache geschriebene Methode):

- Java Objekte erzeugen (Arrays und Strings eingeschlossen)
- auf diese Objekte zugreifen und Änderungen machen
- auf Objekte zugreifen, die innerhalb der JVM erzeugt wurden (unter Verwendung der `Invocation API`)
- Objekte gemeinsam mit der Java-Applikation benutzen.

Weiterhin können native Methoden entsprechende Java-Methoden aufrufen, die Bestandteil einer Klassenbibliothek sind und mit deren Rückgabewert arbeiten (Parameter

[15] Die eigentliche Adressberechnung erfolgt über Offsetwerte, die über die Tabellen *OffsetTableField* und *OffsetTableMethod* verwaltet werden.

übergeben). Als eine zusätzliche Funktionalität können native Methoden Ausnahmen definieren und auslösen, neue Klassen laden und inspizieren, sowie Laufzeit-Typüberprüfungen durchführen.

Die JikesRVM unterstützt JNI in der Version 1.1 fast vollständig, lediglich drei Funktionen wurden nicht implementiert: `DefineClass`, `RegisterNatives` und `UnregisterNatives`. Die entsprechenden JNI-Funktionen sind in `VM_JNIFunctions` definiert. Methoden dieser Klasse werden von der JikesRVM unter Hinzufügen spezieller Prologe/Epiloge übersetzt, welche die Übertragung von nativen zu Java-Aufrufkonventionen übernehmen sowie weitere Koordinationen steuern. Für die Übersetzung wird der plattformabhängige Übersetzer `VM_JNICompiler` eingesetzt, der jedoch nicht bei einer optimierenden Übersetzung verwendet werden kann. Das Thread-System der JikesRVM berücksichtigt spezielle Wiederherstellungsmechanismen, falls eine native Methode nicht ordnungsgemäß beendet werden kann oder blockiert ist. JNI-Methoden ist es in der JikesRVM nicht gestattet, beliebig auf den Java-Speicher (Heap) zuzugreifen.

Die JikesRVM definiert weiterhin so genannte *VM syscalls*, wobei es sich entweder um nicht blockierende Systemaufrufe oder spezielle Dienste aus C-Bibliotheken handelt. Diese Methoden können uneingeschränkt auf den Java-Speicher zugreifen.

7.4 Probleme für interprozedurale Escape-Analyse

Die in diesem Kapitel genannten Eigenschaften der Programmiersprache Java und der JikesRVM führen auf der Konsumentenseite eines Systems zu besonderen Situationen, auf die im Unterschied zu einer auf der Produzentenseite durchgeführten Escape-Analyse speziell eingegangen werden muss. Jede genannte Eigenschaft stellt eigene Anforderungen an die entwickelte interprozedurale Analyse, damit diese erfolgreich in der JikesRVM eingesetzt werden kann. Im folgenden werden die Probleme einzeln an Beispielen dargestellt und im Anschluss eine Lösungsmöglichkeit diskutiert.

Dynamisches Laden, Explizites Laden und Spätes Binden

Der in der JikesRVM verwendete Mechanismus des verzögerten Bindens und das explizite Laden von Klassen führen dazu, dass einige Klassen und deren Methoden während einer aktuellen Escape-Analyse nicht geladen sind bzw. zur Verfügung stehen. Da die in dieser Arbeit vorgestellte interprozedurale Escape-Analyse nicht nur von Informationen der aktuell untersuchten Klasse und deren Methoden abhängig ist, sondern auch das Verhalten zu weitereren Klassen analysiert, muss die Analyse mögliche Probleme berücksichtigen. Wichtig ist z.B. das korrekte Bestimmen einer zu untersuchenden Methode an einer Aufrufstelle. Hierfür muss gewährleistet sein, dass die statischen Typen der entsprechenden Empfängerobjekte ermittelt werden können. Nur durch den richtigen Typ kann eine passende Implementierung der Methode gefunden werden, welche auch die Escape-Eigenschaft der an sie übergebenen Parameter festlegt.

Handelt es sich bei dem Empfänger einer Methode g um ein Parameterobjekt einer Methode m, kann die Ermittlung des entsprechenden statischen Typs unter Umständen problematisch sein, wenn die Aufrufstelle von m noch nicht bekannt ist. Ist g nun polymorph, d.h. es gibt weitere Implementierungen von g, so könnte sich das Verhalten des Parameters je nach Aufrufstelle unterscheiden. Es ist demnach wichtig zu wissen, wel-

chen statischen Typ ein Objekt besitzt und ob eine aufgerufene Methode in mehreren Implementierungen vorliegt (Polymorphie).[16]

Ein anderes Problem ergibt sich aus zum Zeitpunkt der Analyse nicht geladenen Klassen. Neben Klassen, die explizit durch den Benutzer geladen werden (Reflektion) und im schlechtesten Fall überhaupt nicht untersucht werden können, gibt es eine Reihe von Klassen, die zwar zur Verfügung stehen, jedoch erst weit nach der Analyse geladen werden. Hier muss eine Möglichkeit gefunden werden, auf die entsprechenden Informationen zuzugreifen, soweit diese vorliegen.

```
public class Test{

 public void loadClass(className) throws Exception{
  Class c = Class.forName(className);
  A a = (A)c.newInstance();

  testClass(a);
 }

 public void testClass(A a){
  B b = new B();
  a.test(b);
 }
}
```

Abbildung 28: Beispiel eines expliziten Ladens einer Klasse

In der Klasse `Test` (Abbildung 28) wird eine Implementierung der Klasse `A` oder einer ihrer abgeleiteten Klassen durch einen benutzerdefinierten Klassenlader geladen. Die genaue Implementierung wird erst bei Aufruf der Methode `loadClass()` bekannt und kann sich dann für jeden weiteren Aufruf ändern. Das Objekt, auf das die Variable *a* verweist, könnte durch die interprozedurale Escape-Analyse als *bound* identifiziert werden, soweit das Verhalten des Objektes in der aufgerufenen Methode `test()` bekannt wäre. Da sich diese Methode je nach Implementierung von Klasse zu Klasse unterscheiden kann, muss ohne weitere Untersuchungen angenommen werden, dass das Objekt aus der Methode flieht. Ebenso ist das Objekt, auf welches die Variable *b* verweist, nicht ohne weitere Überlegungen untersuchbar. Die Methode `test()` kann durchaus in mehreren Implementierungen vorliegen und der Typ des Empfängerobjektes *a* von der entsprechenden Aufrufstelle von `test()` abhängen.

Reflektion

Durch die Mechanismen der Reflektion kann die interprozedurale Escape-Analyse auf unterschiedliche Art beeinflusst werden (neben dem expliziten Laden von Klassen). Zum einen ist es möglich, ein Objekt nicht über die Operation *new* zu erzeugen, sondern hierfür die Methode `newInstance()` der Reflektion zu verwenden. Es ist jedoch im schlechtesten Fall nicht ohne Weiteres möglich, den entsprechenden Typ und somit die Escape-

[16] Im Gegensatz zu einer polymorphen Methode kommt eine monomorphe Methode nur in einer Implementierung vor ([41]).

Eigenschaft des erzeugten Objektes zweifelsfrei zu bestimmen. Weiterhin können durch die Reflektion Methoden aufgerufen werden und insbesondere Objekte an diese Methoden weitergereicht werden. Da im schlechtesten Fall das Empfängerobjekt der Methode überhaupt nicht bestimmt werden kann, ist auch das Verhalten der an die Methode übergebenen Objekte nicht einschätzbar.

```
public void foo(Class cls, A a, String methodName){

try {
        B b = (B)cls.newInstance();
    Method method = cls.getMethod(methodName, new Class[]
                                        {a.class});
    Object result = method.invoke(b, new Object[] {a});
  } catch (ClassCastException e){
  } catch (InstantiationException e) {
  } catch (NoSuchMethodException e) {
  } catch (IllegalAccessException e) {
  } catch (InvocationTargetException e) {
  } catch (Exception e){
  }
}
```

Abbildung 29: Beispiel eines Methodenaufrufs mittels Reflektion

Abbildung 29 zeigt auszugsweise ein Quellprogramm in Java unter Verwendung der Reflektion. Hier wird ein lokales Objekt b von dem Typ erzeugt, der durch `cls` bestimmt ist. Weiterhin werden sowohl b (implizit) und a (explizit) an eine Methode übergeben, deren Namen zum Zeitpunkt der Analyse nicht bekannt ist. Weder für das lokale Objekt (b), noch für den formalen Parameter (a) kann eine Aussage zum Escape-Verhalten getroffen werden.

JNI

Sobald ein Programm mit der JNI arbeitet, ist eine Escape-Analyse auf der Konsumentenseite nur noch bedingt durchführbar. Im schlechtesten Fall kann eine native Methode (*syscall*) direkt auf Objekte zugreifen, wobei Zeitpunkt und Art des Zugriffs aus Sicht des Java-Quellprogrammes nicht ersichtlich sind. Eine in der JikesRVM verwendete Escape-Analyse kann nicht das Verhalten einer nativen Methode berücksichtigen. Dies gilt auch für Objekte, die an JNI-Methoden übergeben werden. Eine Analyse des Verhaltens formaler Parameter ist ohne Kenntnis des Verhaltens der nativen Methode nicht möglich.

Abbildung 30 zeigt ein Beispielprogramm ([1]) mit einer nativen Methode zum Zugriff auf ein Feld. Im linken Teil ist das entsprechende Java-Programm zu erkennen, welches das Feld und eine native Methode zum Zugriff definiert. Das Feld befindet sich zur Laufzeit entsprechend im Heap der JikesRVM. Der rechte Teil zeigt den Quellcode des zugehörigen maschinenabhängigen Programmes in `C` mit der eigentlichen Implementierung der Methode. Im vorliegenden Fall wird eine Referenz auf das Feldobjekt direkt in die native Methode übergeben und diese kann entsprechend darauf zugreifen. Je nach vorliegender Implementierung kann hier nicht entschieden werden, ob ein Zugriff auf das Objekt vorliegt oder

```
class IntArray {
  private native int sumArray(int arr[]);
  public static void main(String args[]) {
    IntArray p = new IntArray();
    int arr[] = new int [10];
    for (int i = 0; i < 10; i++)
      arr[i] = i;
    int sum = p.sumArray(arr);
    System.out.println("sum = " + sum);
  }
  static {
    System.loadLibrary("MyImpOfIntArray");
  }
}
```

```
#include <jni.h>
#include "IntArray.h"

JNIEXPORT jint JNICALL
Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr)
{
  jsize len = (*env)->GetArrayLength(env, arr);
  int i, sum = 0;
  jint *body = (*env)->GetIntArrayElements(env, arr, 0);
  for (i=0; i<len; i++) {
    sum += body[i];
  }
  (*env)->ReleaseIntArrayElements(env, arr, body, 0);
  return sum;
}
```

Abbildung 30: Beispielprogramm JNI-Feldzugriff in Java und C

nicht. Zum Beispiel könnte die Implementierung in `C` mit anderen nativen Programmen interagieren.[17] Weiterhin kann ein natives Programm über die entsprechenden Bibliotheken der JNI (`jni.h`) auf Attribute, Werte und Methoden von Java-Objekten zugreifen, ohne dass der Zugriff über eine Analyse des Java-Quellprogrammes identifiziert werden kann.

7.5 Lösung

7.5.1 Escape-Kontrolleinheit und Wächter

Die oben genannten Eigenschaften und die damit zusammenhängenden Probleme für die interprozedurale Escape-Analyse können im schlechtesten Fall dazu führen, dass eine Analyse nicht möglich ist oder bereits gewonnene Ergebnisse ungültig werden. Entsprechende pessimistische Annahmen bei der Untersuchung können einerseits helfen, den Problemen entgegenzuwirken, verschlechtern jedoch die Qualität der Analyseergebnisse. Auf der anderen Seite kann bei sich nachträglich ändernden Bedingungen eine gänzlich neue Untersuchung erforderlich werden. Die in diesem Fall auf der bisher durchgeführten Escape-Analyse basierenden Optimierungen könnten zu einer Fehlfunktion des übersetzten Programmes führen. Um die korrekte Arbeitsweise eines Programmes auch nach erfolgter Escape-Analyse und darauf basierender Optimierung zu gewährleisten sowie Fehlfunktionen auszuschließen, muss die JikesRVM um entsprechende Mechanismen erweitert werden, mittels derer eine neue Analyse oder eine entsprechende Re-Optimierung durchgeführt werden können.

Abbildung 31 zeigt schematisch die in dieser Arbeit verwendete Lösung. Basis der Erweiterung für die JikesRVM ist eine zentrale Kontrolleinheit (*Escape-Controller*), welcher mit so genannten Wächtern (*Guards*) zusammenarbeitet. Bei der Übersetzung eines Programmes mit dem optimierenden Übersetzer der JikesRVM wird immer eine Methode für sich bearbeitet (siehe Kapitel 8.3.3), wobei auch die interprozedurale Analyse stattfindet. Um einen Überblick über die bereits analysierten Methoden zu bekommen, diese zu verwalten und Informationen für auf der Escape-Analyse basierende Optimierungen bereitzustellen, ist eine übergeordnete Kontrolleinheit notwendig. Die Kontrolleinheit muss

[17] Das maschinenabhängige Programm liegt nur in bereits übersetzter Form vor.

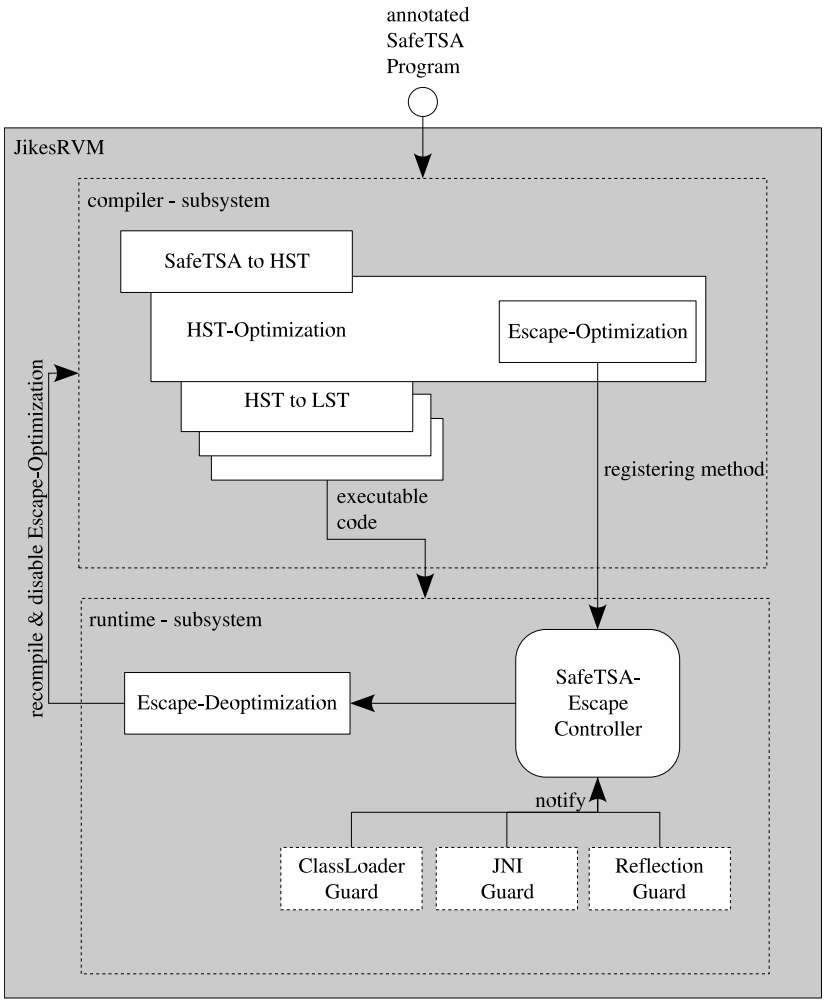


Abbildung 31: Laufzeitsystem mit Wächter (*Guard*) und Kontrolleinheit (*Escape-Controller*)

vor allem in der Lage sein, Informationen zu analysierten Methoden auch nach Abschluss der Analyse und der Übersetzung in Maschinencode noch weiter zu verwalten. Die Escape-Kontrolleinheit hat folgende Aufgaben:

- Verwaltung der analysierten Methoden
- Auflisten aller auf Basis der Escape-Analyse durchgeführten Optimierungen
- Verwaltung der Wächter
- Veranlassung erforderlicher neuer Analysen bei veränderten Bedingungen
- Initiierung einer erforderlichen Re-Optimierung

Ein Wächter hat die Aufgabe, bestimmte Ereignisse zu überwachen und die Escape-Kontrolleinheit bei Eintreten eines solchen zu benachrichtigen. Die Kontrolleinheit übernimmt dann die weitere Steuerung, es erfolgt kein aktiver Eingriff durch den Wächter.

Diese Methode erlaubt eine einfache Erweiterung des Systems auf neue Eigenschaften und Techniken, indem jeweils ein neuer Wächter eingeführt und die Kontrolleinheit um die entsprechende Steuerung erweitert wird. Die in dieser Arbeit implementierten Wächter werden für die zuvor genannten Eigenschaften eingesetzt und arbeiten wie folgt:

ClassloaderGuard: Dieser Wächter überwacht das Laden von Klassen. Sobald eine neue Klasse geladen wird, entweder explizit durch einen Klassenlader[18] oder implizit durch spätes Binden, benachrichtigt er die Escape-Kontrolleinheit und teilt den Typ der neu geladenen Klasse mit.

JNIGuard: Der JNI-Wächter prüft, ob auf Funktionen der JNI zugegriffen wird, welche in `VM_JNIFunctions` definiert sind und unter Verwendung von `VM_JNICompiler` übersetzt werden. Sobald ein Programm die Schnittstelle verwendet, gibt der Wächter eine entsprechende Information an die Escape-Kontrolleinheit weiter.

ReflectionGuard: Der Wächter der Reflektion schließlich wacht über die Verwendung von Funktionen der `Reflection API`, ausgenommen das explizite Laden von Klassen durch einen Klassenlader (hierfür ist der ClassloaderGuard zuständig). Insbesondere prüft der Wächter, ob in einem Programm das Klassenobjekt einer Klasse, referenziert durch das statische Feld *class*, verwendet wird. Erfolgt ein Zugriff auf ein Klassenobjekt, meldet er den der Klasse zugehörigen Typ an die Escape-Kontrolleinheit.

Nachdem ein Wächter seine Informationen an die Escape-Kontrolleinheit gegeben hat, muss diese eine der Situation entsprechende Aktion einleiten. Zum Beispiel könnte ein Wächter melden, dass eine neue Klasse geladen wurde, deren Methoden die Ergebnisse einer vorangegangenen Escape-Analyse beeinflussen, indem sie eine aktuellere Implementierung einer Methode bietet (eine weniger spezielle überschreibt). In diesem Fall muss die Analyse unter Einbeziehung der Escape-Eigenschaft der formalen Parameter der aktuelleren Methode wiederholt werden. Ändert sich hierdurch die Escape-Eigenschaft eines lokalen Objektes, so sind alle auf Basis der Analyse durchgeführten Optimierungen der untersuchten Methode rückgängig zu machen. Betrifft die Änderung jedoch einen formalen Parameter dieser Methode, so müssen alle weiteren Methoden überprüft werden, die eine relevante Aufrufstelle der untersuchten Methode besitzen. Ebenso sind die gemachten Optimierungen der weiteren Methoden neu zu bewerten und gegebenenfalls rückgängig zu machen. Der Escape-Kontrolleinheit obliegt die Verwaltung und Steuerung dieser Aufgaben.

Zu bemerken ist an dieser Stelle, dass die Arbeit der Escape-Kontrolleinheit in den in dieser Arbeit verwendeten Benchmarkprogrammen nur minimal ausfiel, da keine der zuvor genannten Probleme auftraten bzw. bereits ermittelte Escape-Eigenschaften keine nachträgliche Anpassung benötigten.

7.5.2 Arbeitsweise

Um die erforderlichen Aufgaben zu bewältigen, benötigt die Escape-Kontrolleinheit in erster Linie Informationen zu den untersuchten Methoden. Zu diesem Zweck registriert der optimierende Übersetzer der JikesRVM jede mit dem SafeTSA-Modul übersetzte Methode

[18] Verwendung von `Class.forName()` oder eines benutzerdefinierten Klassenladers.

bei der Escape-Kontrolleinheit, welcher eine Liste der bereits analysierten und übersetzten Methoden für jeden auftretenden Typ (Klasse) führt. Die Registrierung enthält auch Informationen darüber, ob Optimierungen auf Basis der Escape-Analyse durchgeführt wurden. Zusätzlich meldet der Übersetzer alle relevanten Aufrufstellen von Methoden (die im SafeTSA-Format vorliegen) an die Kontrolleinheit, welche diese der aufgerufenen Methode zuordnet. Befindet sich die aufgerufene Methode noch nicht in der entsprechenden Liste der Kontrolleinheit, so wird sie hinzugefügt.

Weiterhin überprüft die Kontrolleinheit für jede hinzugefügte Methode m, ob bereits eine Implementierung dieser Methode registriert wurde. Ist das der Fall, so werden alle Implementierungen von m als polymorph gekennzeichnet und die Kontrolleinheit veranlasst eine neue Analyse jener Methoden, die m aufrufen. Eine neue Analyse einer Methode wird veranlasst, indem die Escape-Kontrolleinheit die Übersetzung der Methode neu initiiert.

Die von den Wächtern gemeldeten Ereignisse werden von der Escape-Kontrolleinheit konservativ und pessimistisch betrachtet. Im konkreten Fall einer Verwendung der Schnittstelle JNI führt dies zu der Annahme, alle Objekte und Parameter könnten theoretisch fliehen. Entsprechend veranlasst die Escape-Kontrolleinheit bei Verwendung der JNI eine Neuübersetzung aller registrierten Methoden, wobei die Escape-Analyse und darauf basierende Optimierungen im optimierenden Übersetzer der JikesRVM deaktiviert werden.

Meldet der *ReflectionGuard* ein Ereignis, so überprüft die Kontrolleinheit zunächst, welche Methoden der gemeldeten Klasse registriert wurden. Auch hier wird angenommen, dass alle mit dem Typ in Verbindung stehenden Objekte und Parameter fliehen. Beinhaltet sind alle Instanzen des genannten Typs und weiterhin alle Objekte und Parameter, die an entsprechende Methoden $f_1 ... f_n$ der Klasse als Argumente übergeben werden (Aufrufstellen). Die Escape-Kontrolleinheit veranlasst eine Neuübersetzung für die Methoden der Klasse sowie aller Methoden, mit einer Aufrufstelle von $f_1 ... f_n$.[19] Hierbei erhalten alle formalen Parameter p_i von $f_1 ... f_n$ die Escape-Eigenschaft $escape_f(p_i) = escape$ und alle Methoden $escape(f) = escape$.

Wurde eine bereits analysierte Methode mit einer auf der Escape-Analyse basierenden Optimierung übersetzt, so muss diese Optimierung unter Umständen rückgängig gemacht werden. Hierfür startet die Escape-Kontrolleinheit für alle registrierten Methodenoptimierungen einen so genannten *Re-Optimierer*, der wiederum eine neue Übersetzung der spezifizierten Methode unter Ausschluss der Optimierung durchführt. Für alle Methoden, welche sowohl eine neue Escape-Analyse als auch eine Re-Optimierung erfordern, ist die Abarbeitung in einem Schritt durchführbar. Alternativ könnte hier ein Mechanismus eingesetzt werden, bei welchem jede optimierte Methode durch die JikesRVM in einer zweiten, nicht optimierten Version gespeichert wird. Im Fall einer Re-Optimierung müsste dann lediglich die nicht optimierte Variante der Methode aktiviert und ausgeführt werden.

[19] Ändert sich die Escape-Eigenschaft der formalen Parameter der Methode nicht, so ist eine Bearbeitung der Methoden mit einer Aufrufstelle nicht notwendig.

8 Implementierung

In diesem Kapitel wird die Implementierung der vorgestellten Analyse und Annotation für ein SafeTSA-basierendes Produzenten- und Konsumentensystem dargestellt. Auf der Produzentenseite wurde ein bestehender SafeTSA-Übersetzer erweitert, auf der Konsumentenseite kam eine für SafeTSA erweiterte Version der *JikesRVM* (Jikes - Research Virtual Machine) zum Einsatz. Die einzelnen Abschnitte behandeln den Aufbau der verwendeten Systeme, die Erweiterungen für die Unterstützung des Zwischencodeformats SafeTSA sowie die Annotation und die verwendete Escape-Analyse.

8.1 SafeTSA-Übersetzer Produzentenseite

8.1.1 Der Java-Übersetzer Pizza

Pizza[20] ist ein Java-zu-Bytecode Übersetzer, welcher selbst in der Programmiersprache Java durch Martin Odersky und Philip Wadler entwickelt wurde ([46]). Neben der reinen Übersetzung erweitert das Programm die Sprache Java um zusätzliche Mechanismen, z.B. durch die Einführung generischer Konstrukte (*Generics*). Der Pizza-Übersetzer bietet sich aufgrund seiner hohen Leistungsfähigkeit und seiner leichten Anpassungsfähigkeit für die Entwicklung eines SafeTSA-Übersetzers an, der Quellprogramme in Java übersetzt und als Ausgabeformat anstelle Bytecode den SafeTSA-Zwischencode verwendet.

8.1.2 SafeTSA-Erweiterung für Pizza

Nachdem Pizza das Eingabeprogramm in Java eingelesen und analysiert hat, wird mit den gewonnenen Informationen die interne SafeTSA-Darstellung des Programmes aufgebaut. Diese Darstellung basiert auf einem Kontrollstrukturbaum und Basisblöcken, welche die SSA-Form und die Dominatorrelation abbilden. Im nächsten Schritt werden dann diverse plattformunabhängige Optimierungen durchgeführt, welche durch die SSA-Form besonders effizient durchführbar sind. Unter anderem kann zu diesem Zeitpunkt eine Entfernung gemeinsamer Teilausdrücke (*Common Subexpression Elimination*), die Entfernung nicht verwendeten Codes (*Dead Code Elimination*) und eine Konstantenfaltung/ -weitergabe (*Constant Propagation*) erfolgen. Nach Abschluss der Optimierungsphase wird die optimierte Programmdarstellung an den SafeTSA-Kodierer weitergegeben, der anstelle von Bytecode eine Ausgabe im SafeTSA-Format erzeugt. Derzeit unterstützt der SafeTSA-Kodierer die beiden Formate *SafeTSA-ASCII*($SafeTSA\alpha$) und *SafeTSA-Binary*($SafeTSA\beta$). Bei *SafeTSA-ASCII* handelt es sich um ein Arbeitsformat, welches zu Entwicklungszwecken eingesetzt wird und die Typ- und Referenzsicherheit nicht unterstützt. In dieser Schrift werden Beispiele mit Rücksicht auf die Lesbarkeit in SafeTSA-ASCII dargestellt.

Beispiel - SafeTSA ASCII-Ausgabe Das Beispiel zeigt ein Java-Eingabeprogramm für den SafeTSA-Übersetzer (Abbildung 32), die erzeugte SafeTSA-Darstellung (Abbildung 33) und einen Auszug des im SafeTSA-ASCII-Format ausgegebenen Zwischencodes (Abbildung 34).

[20] http://pizzacompiler.sourceforge.net/

```
class B{
 int i=1;
  public int foo(int j){
    i = i + j;
    return i;
  }
}
```

Abbildung 32: Java-Eingabeprogramm für den Übersetzer

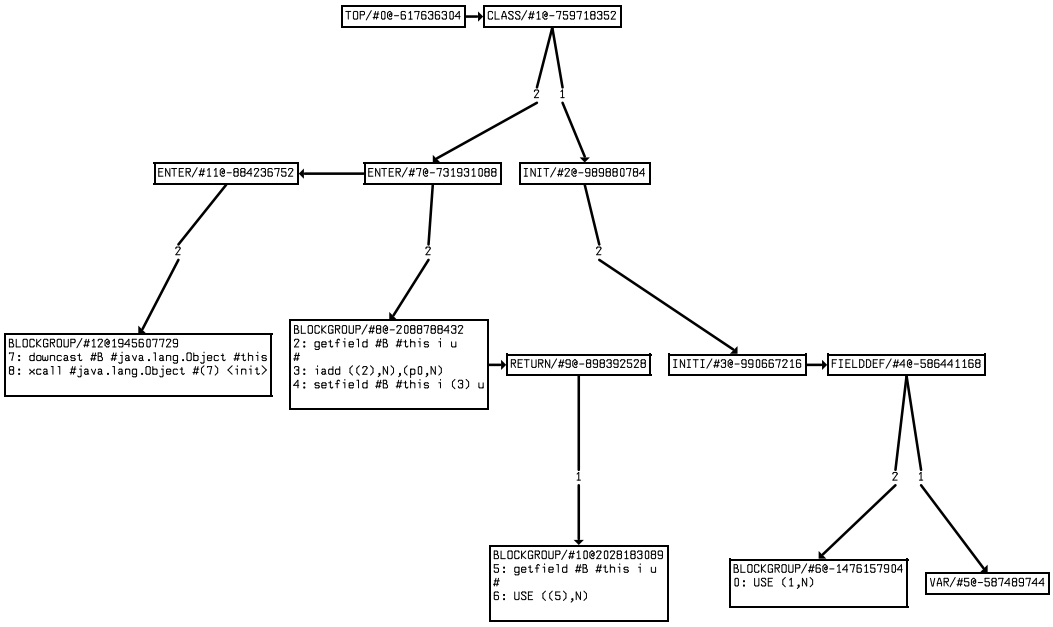


Abbildung 33: Beispiel einer SafeTSA-Programmdarstellung

8.2 Erweiterung des Übersetzers der Produzentenseite

8.2.1 Intraprozedurale Escape-Analyse

Im folgenden werden die Anpassungen der Escape-Analyse von Whaley und Rinard für den Einsatz im SafeTSA-Übersetzer betrachtet[21]. Bei der Übersetzung eines Programmes mit Pizza wird ein auf der SSA-Form basierender Syntaxbaum erzeugt, in dem keine Variablen im ursprünglichen Sinne existieren. Vielmehr werden anstelle dessen Instruktionsnummern betrachtet[22]. Jede Anweisung eines Programmes kann über die eindeutige Instruktionsnummer identifiziert werden, wobei diese gleichzeitig den entsprechenden aktuellen Wert repräsentiert. Tabelle 3 enthält die Abbildungen auf die Zuweisungen der ursprünglichen Analyse (siehe Seite 97), mit $C1$ – Symboltabelleneintrag der Klasse, M – Symboltabelleneintrag der Methode, $aType$ – den Eintrag des Arraytyps in der Typentabelle, i die eindeutige Instruktionsnummer, über welche im weiteren Verlauf des Programmes auf den resultierenden Wert der Anweisung zugegriffen werden kann; *elements* bezeichnet das

[21] Die Darstellung bezieht sich auf den in Kapitel A vorgestellten Algorithmus.
[22] Siehe Kapitel 6.1.

```
(safetsa
  (constants
    (constant :0: null null)
    (constant :1: boolean false)
    (constant :2: boolean true)
    (constant :3: int 1)
  (types
    (primitive-type boolean :4:)
    (primitive-type byte :5:)
    (primitive-type char :6:)
    (primitive-type double :7:)
    (primitive-type float :8:)
    (primitive-type int :9:)
    (class java.lang.Class :10: :11: :12:)
    (class java.lang.Object :13: :14: :15:)
    (class java.lang.String :16: :17: :18:)
    (primitive-type long :19:)
    (primitive-type null :20:)
    (primitive-type short :21:)
    (class B :22: :23: :24:))
  (methods
    (dynamic-method :25: :13: <init> (arg-types) (ret-type))
    (statically-called-dynamic-method :26: :13: <init> (arg-types)(ret-type)))
  (fields
    (object-field :27: :22: i :9:))
  (class :22:
    (modifiers public)
    (direct-supertypes :13: (interfaces ))
    (fields (nonstatic
               (field (modifiers public) :9: :27:
                 (expression-with-value
                   (blockgroup :28: (no-phimove))
                  :3:))))
    (methods (nonstatic-method :29:
                foo (modifiers public)
                 (arguments
                   (this-argument :24: :30:)
                   (this-argument :23:  :31:)
                   (argument :9: :32:))
                 (return-type :9:)
                 (statements
                   (blockgroup :33:
                     (block
                       (op-getfld :34: :24: :27: :30:)
                       (op-prim :35: int add(:34: :32:))
                       (op-setfld :36: :24: :27: :30: :35:))
                     (no-phimove))
                   (return
                     (expression-with-value
                       (blockgroup :37:
                         (block
                           (op-getfld :38: :24: :27: :30:))
                       (no-phimove))
                     :38:))))
```

Abbildung 34: ASCII-Ausgabe des SafeTSA-Kodierers zur Klasse B (Auszug)

ausgezeichnete Feld für einen Arrayzugriff. Methodenaufrufe werden in *xcall* (keine dynamische Bindung) und *xdispatch* (dynamische Bindung) unterteilt. Zuweisungen der Form $l = v$ sind durch die Verwendung der SSA-Form ausgeschlossen.

Anweisung	SafeTSA-Instruktion
$l = v$	–
$l_1.f = l_2$	i:setfield #C1 object field value
$l_1.elements = l_2$	i:setelem aType object index value
$l_1 = l_2.f$	i:getfield #C1 object field
$l_1 = l_2.elements$	i:getelem aType object index
$l = newClassName(p_1, \ldots, p_k$	i:xcall CL #C1 $\langle init \rangle (p_1, \ldots, p_k)$
$l = l_0.op(l_1, \ldots, l_k)$	i:xcall CL #Class M $(l_1, \ldots, l_k)$
	oder
	i:xdispatch #C1 M $(l_1, \ldots, l_k)$
$return\ l$	i:return l

Tabelle 3: Anweisungsrepräsentation in SafeTSA

Im folgenden wird die Anpassung der Übergangsfunktionen auf die Operationen von SafeTSA dargestellt. Sei dazu $(I, O, e, r) = \alpha(\bullet st)$ der Points-To-Escape-Graph vor Ausführung der Anweisung st:

1. Ist st eine xcall-Anweisung, so wird überprüft, ob es sich hierbei um einen Konstruktoraufruf handelt. Ein Konstruktor wird hier durch die spezielle statische Methode mit dem Namen $\langle init \rangle$ repräsentiert.

   (a) Es handelt sich bei st um die Instanziierung eines Objektes mittels eines Konstruktors. Alle Konstruktoren sind bereits analysiert. Es wird eine innere Kante $E : temp \rightarrow n$ erzeugt, wobei $temp$ eine neue temporäre Variable und n den erzeugten inneren Knoten, welcher das Objekt repräsentiert, bezeichnen. Diese Kante wird zur Menge der inneren Kanten I hinzugefügt. Anschließend werden die Analyseresultate des Konstruktors auf den Graphen abgebildet. Der neue Points-To-Escape-Graph $(I', O', e', r') = \alpha(st\bullet)$ nach der Ausführung von st ist definiert als

   $$(I', O', e', r') = map(st, (I \cup \{E\}, O, e, r), op)$$

   wobei op den aufgerufenen Konstruktor bezeichnet. Da alle Thread-Objekte grundsätzlich fliehen, muss anschließend noch überprüft werden, ob es sich um eine von `java.lang.Thread` abgeleitete Klasse handelt. Objekte dieser Art werden als fliehend gekennzeichnet.

   (b) Ist st kein Konstruktoraufruf, so kann diese Instruktion wie ein *xdispatch* behandelt werden, da die Analyse nicht zwischen statischen und virtuellen Methodenaufrufen unterscheidet.

2. Handelt es sich bei st um eine *xdispatch*-Anweisung, wird überprüft, ob die aufgerufene Methode M einen Rückgabewert liefert.

(a) Liefert M einen Referenzwert als Rückgabewert, wird eine neue innere Kante $E : temp \rightarrow n$ erzeugt, wobei $temp$ eine neue temporäre Variable und n einen den entsprechenden Rückgabewert repräsentierenden Knoten darstellt. Ist M bereits analysiert, so entspricht n einer Kopie des Knotens, der mit der Rücksprunganweisung zurückgegeben wurde. Die Escapefunktion $e(n)$ des Points-To-Graphen von M wird ebenfalls kopiert. Ist M nicht analysiert, so entspricht n dem Rückgabewert der nicht analysierten Methode.

(b) Liefert M keinen Rückgabewert, ist diesbezüglich nichts zu tun.

Seien im folgenden die Parameter der Methode M betrachtet. Ist M bereits analysiert, ergibt sich der neue Points-To-Escape-Graph zu:

$$(I', O', e', r') = map(st, (I, O, e, r), M).$$

Ist M noch nicht analysiert oder liegt als fertig übersetzte Klasse vor, werden alle übergebenen Parameter als durch die Methode M fliehend angenommen. Seien $n_1, \ldots, n_k$ die Knoten, welche die Argumente repräsentieren. Dann gilt:

$$e'(n_i) = e(n_i) \cup \{M\}, 1 \leq i \leq k.$$

3. Handelt es sich bei *st* um eine *getfield*-Anweisung, wird zunächst der das Objekt *obj* repräsentierende Knoten n bestimmt, das der Instruktion zugrunde liegt. Ist n gefunden, wird die Menge S_E aller fliehenden Knoten bestimmt, auf die n zeigt.

   (a) $S_E = \emptyset$ – eine neue innere Kante der Form $E : temp \rightarrow n'$ wird generiert, wobei *temp* eine neue temporäre Variable und n' einen Knoten bezeichnet, der von *obj* aus mit einer inneren, mit *field* beschrifteten Kante erreichbar ist. Sei M die Menge dieser Knoten. Dann ergibt sich der neue Points-To-Escape-Graph zu:

   $$(I', O', e', r') = (I \cup M, O, e, r).$$

   (b) $S_E \neq \emptyset$ – *object* zeigt auf mindestens einen fliehenden Knoten. Zusätzlich zu den Schritten aus a) werden ein neuer Ladeknoten n_L erzeugt und äußere Kanten von allen Knoten aus S_E ausgehend nach n_L eingefügt. Sei die Menge dieser neuen äußeren Kanten mit N bezeichnet. Dann ergibt sich der neue Points-To-Escape-Graph zu

   $$(I', O', e', r') = (I \cup M, O \cup N, e, r).$$

4. Ist *st* eine *setfield*-Anweisung, so werden zunächst die Knoten v und n bestimmt, die die über *value* beziehungsweise *obj* erreichbaren Objekte repräsentieren. Weiter werden alle Kanten mit der Beschriftung *field* gelöscht (M - Menge dieser Kanten) und eine neue innere Kante eingefügt

$$E : n \longrightarrow^{field} v.$$

Der neue Points-To-Escape-Graph ergibt sich damit zu

$$(I', O', e', r') = (I \setminus M_I \cup \{E\}, O \setminus M_O, e, r).$$

Dabei bezeichnet M_I die Menge aller inneren Kanten von M und M_O die Menge aller äußeren Kanten von M, so dass gilt: $M_I \subseteq M \supseteq M_O, M_I \cap M_O = \emptyset$ und $M_I \cup M_O = M$.

5. Handelt es sich bei *st* um eine *setelem*- oder *getelem*-Anweisung, kann entsprechend der Anweisungen *setfield* bzw. *getfield* vorgegangen werden, wobei das Feld *field* durch *elements* ersetzt wird.

6. Ist *st* schließlich eine *return*-Anweisung, so wird der Knoten n bestimmt, der das über *op* erreichbare Objekt repräsentiert und der Menge der Returnknoten r hinzugefügt. Damit ergibt sich folgender neuer Points-To-Escape-Graph

$$(I', O', e', r') = (I, O, e, r \cup R)\, mit$$

$$R = \begin{cases} \{n\} & \text{falls } st \text{ ist } return\text{-Anweisung} \\ \emptyset & \text{sonst} \end{cases}$$

Nach Terminierung des Algorithmus werden die ermittelten Escape-Eigenschaften im Kontrollstrukturbaum (CST) abgespeichert und stehen einer weiteren Bearbeitung zur Verfügung, in der die möglicherweise fliehenden Objekte aus der Menge der fliehenden Objekte zu identifizieren sind. Um alle Objekte mit der speziellen Escape-Eigenschaft *may* zu finden, werden alle jene Elemente bestimmt, die ausschließlich durch die implizite oder explizite Übergabe an eine Methode als Argument fliehen (eingeschlossen die als spezielle Methode repräsentierten Konstruktoren). Jedes so bestimmte Objekt erhält in der SafeTSA-Darstellung die Escape-Eigenschaft *may* und wird aus der Menge entfernt. Alle übrigen Elemente der Menge behalten die Eigenschaft *escape*.

8.2.2 Erweiterter SafeTSA-Kodierer

Die intraprozedurale Escape-Analyse im SafeTSA-Übersetzer liefert die für die Annotation notwendigen Informationen in Form der Escape-Eigenschaft, die durch das Programm

im Kontrollstrukturbaum gespeichert sind. Im abschließenden Schritt der Zwischencodeerzeugung setzt ein entsprechend modifizierter Kodierer unter Verwendung des erweiterten Instruktionssatzes und Maschinenmodells von SafeTSA die Escape-Annotationen für das SafeTSA-Zwischencodeformat um. Die Escape-Eigenschaft einer Methode wird hierbei durch die Escape-Eigenschaft des nullten formalen Parameters dieser Methode dargestellt. Mit dem nullten Parameter wird im SafeTSA-Format die Referenz auf `this` repräsentiert, wodurch die Escape-Eigenschaft dieses Parameters genau der Escape-Eigenschaft der Methode entspricht.

Der für die Generierung von SafeTSA-Zwischencode entwickelte SafeTSA-Kodierer ([58]) bezieht seine Informationen aus der internen SafeTSA-Darstellung des Programmes. Im ersten Schritt erzeugt der Kodierer zunächst einen eigenen Kontrollstrukturbaum mit Basisblöcken, wobei die ermittelten Escape-Eigenschaften mit übernommen werden. Danach startet der Kodierer einen Tiefensuchenalgorithmus auf diesem Baum und überträgt das Programm sukzessive in einen Ausgabestrom. Die Art des Ausgabestroms (SafeTSA-ASCII oder SafeTSA-Binary) wird zuvor über einen Schalter festgelegt. Abbildung 35 zeigt einen vereinfachten Überblick über den Prozess der Übersetzung und Kodierung (von links nach rechts).

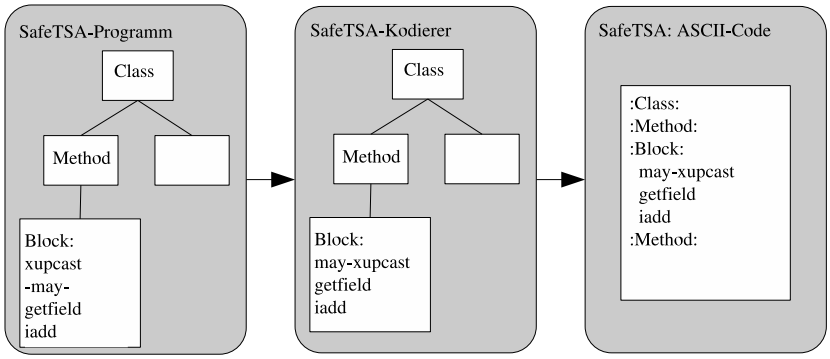


Abbildung 35: Arbeitsweise SafeTSA-Kodierer

Jede mit dem Kodierer erzeugte Ausgabedatei des SafeTSA-Formats erhält zusätzlich eine Startsequenz am Beginn der Datei, die das verwendete Format und die Art der Annotation kennzeichnet. Hierdurch kann ein Programm auf der Konsumentenseite des SafeTSA-Systems entscheiden, was für die Dekodierung der Datei notwendig ist. Entscheidend ist, dass jedes Einleseprogramm auf einer Konsumentenseite zumindest die vereinheitlichte Startsequenz dekodieren kann, um z.B. bei Nicht-Unterstützung eine mit dem erweiterten Instruktionssatz kodierte und annotierte Datei abweisen zu können.

Beispiel Abbildung 36 zeigt die ASCII-Ausgabe für einen erweiterten Kodierer mit annotierten Escape-Informationen. Der Kodierer verwendet hierbei den erweiterten Instruktionssatz von SafeTSA, wobei die Escape-Informationen nicht in Form von Hinzufügungen realisiert sind, sondern durch spezielle Instruktionen repräsentiert werden. Für jede Escape-Eigenschaft und den zugehörigen Modifikator wurde eine spezielle Variante der betroffenen Instruktion erzeugt. So können beispielsweise die Instruktionen $may - xupcast$ und $bound - xupcast$ entsprechend ihrer Modifikatoren *may* und *bound* verwendet werden, wobei abweichend an der Stelle von $escape - xupcast$ die ursprüngliche Variante *xupcast*

genutzt wird (in Anlehnung an die nicht für die Annotation zu verwendenden Instruktionen, z.B. *iadd*). Weiterhin wurden formale und aktuelle Parameter als Tupel beschrieben, deren zweiter Wert die Escape-Information enthält, z.B. der Parameter `FRect` der Methode `checkIntersect`.

8.3 JikesRVM - Aufbau

8.3.1 Übersicht

Die *JikesRVM* ([5], [39]) wurde für Forschungszwecke am IBM T.J. Watson Forschungszentrum entwickelt. Sie basiert auf dem früheren Projekt *Jalapeno* derselben Einrichtung. Die JikesRVM richtet sich speziell an Entwickler und bietet eine Reihe von Möglichkeiten, neue Ansätze und Optimierungen in einem Gesamtsystem zu integrieren. Die wesentlichen Eigenschaften der JikesRVM sind:

- Die JikesRVM ist selbst in der Programmiersprache Java geschrieben, ausgenommen ca. 1000 Zeilen in der Sprache `C` für den Zugriff auf Systemressourcen.
- Es sind mehrere Übersetzungsprogramme integriert, der so genannte *Baseline Compiler*, ein optimierender Übersetzer (*Optimizing Compiler*) und ein adaptiver Übersetzer (Adaptive Compiler). Eine Interpretation von Java-Programmen wird in der zur Verfügung stehenden Version nicht unterstützt.
- Die JikesRVM verfügt über verschiedene Speicherbereiniger (Garbage Collectors).
- Ein Paket für die effiziente Verwaltung von Threads wurde hinzugefügt.
- Der optimierende Compiler führt bei Bedarf sehr weitreichende Optimierungen durch.
- Es wurde eine flexible Optimierungsinfrastruktur integriert, das *Adaptive Optimization System*.

Obwohl die JikesRVM selbst in der Programmiersprache Java geschrieben ist, benötigt sie keine virtuelle Maschine im herkömmlichen Sinne zur Ausführung, wie es für übliche Java-Programme notwendig ist. Anstelle dessen werden in einem komprimierten Startprozess (*Bootstrapping*) alle benötigten Grundelemente der JikesRVM durch eine unterstützende virtuelle Maschine geladen. Sind die Elemente einmal geladen, so übernimmt die JikesRVM selbst die Ausführung. Für den speziellen Startprozess werden die benötigten Grundelemente als Maschinencode in ein so genanntes *Bootimage* geschrieben. Durch die offene Architektur dieses Systems werden basierend auf einem klaren und überschaubaren Softwaremodell Schnittstellen für die Integration von zusätzlichen Programmteilen geschaffen. So wurden in der Vergangenheit neue Speicherbereiniger entwickelt und durch die JikesRVM getestet (z.B. [22], [34], [37] und [19]), sowie neue Formen der Optimierung eingearbeitet (z.B. [14], [35] und [15]).

8.3.2 JikesRVM - Komponenten

Die JikesRVM setzt sich aus vier verschiedenen Komponenten zusammen:

- dem Laufzeitsystem (*Runtime Subsystem*)

```
enter method boolean checkIntersect(FRect (p0,bound) (p1,bound))

FRect-0: may-new FRect
void: may-xcall <init> FRect (0) (10),(15),(30),(30)
#FRect-0: bound-xupcast FRect #FRect (p1)
boolean-0: bound-xdispatch #FRect (0) intersects (0,may)
boolean-1: return boolean 0

enter method boolean intersects(FRect (p0,bound) (p1,may))

void: xcall FrameBorders #class includeSmallBorders (p1,may)
#FRect-0: may-xupcast FRect #FRect p1
int-0: may-getfield #FRect (0) x
int-1: may-getfield #FRect (0) width
int-2: iadd (0) (1)
int-3: bound-getfield #FRect p0 x
boolean-0: ilte (2),(3)
int-4: may-getfield #FRect (0) y
int-5: may-getfield #FRect (0) height
int-6: iadd (4),(5)
int-7: bound-getfield #FRect p0 y
boolean-1: ilte (6),(7)
boolean-2: bbor (0) (1)
int-7: bound-getfield #FRect p0 width
int-8: iadd (3),(7)
boolean-3: igte (0),(8)
boolean-4: bbor (2),(3)
int-9: bound-getfield #FRect p0 height
int-10: iadd (7),(9)
boolean-5: igte (4),(10)
boolean-6: bbor (4) (5)
boolean-7: bbnot (6)
boolean-8: bbreturn (7)

enter method void incDefBorder(FRect (p0,bound) (p1,bound))

#FRect-0: bound-xupcast FRect #FRect p1
int-0: bound-getfield #FRect (0) x
int-1: isub (0),const-2
void: bound-setfield #FRect (0) x (1)
int-2: bound-getfield #FRect (0) y
int-3: isub (2),const-2
void: bound-setfield #FRect (0) y (3)
int-4: bound-getfield #FRect (0) width
int-5: iadd (4),const-2
void: bound-setfield #FRect (0) width (5)
int-6: bound-getfield #FRect (0) height
int-7: iadd (6),const-2
void: bound-setfield #FRect (0) height (7)
```

Abbildung 36: Annotierte ASCII-Ausgabe mit erweiterten Instruktionen

- dem Thread- und Synchronisierungssystem (*Thread- and Synchronization Subsystem*)
- dem Speicherverwaltungssystem (*Memory Management Subsystem*)
- dem Übersetzersystem (*Compiler Subsystem*)

Für die Erweiterung der JikesRVM bezüglich der Verarbeitung von Programmen, die in dem Zwischencodeformat SafeTSA übertragen werden, ist insbesondere das System des Übersetzers von Bedeutung. Im folgenden Unterabschnitt wird die Arbeitsweise des optimierenden Übersetzers näher beleuchtet.

8.3.3 JikesRVM - optimierender Übersetzer

Eine Übersicht des Übersetzungsvorganges ist in Abbildung 37 dargestellt. Das Programm, welches in Java-Bytecode vorliegt, wird in die JikesRVM geladen und in mehreren Stufen dynamisch zur Laufzeit übersetzt. Dabei werden die Daten in drei unterschiedliche Zwischencoderepräsentationen transferiert und in jeder Stufe die für dieses Format geeigneten Optimierungen durchgeführt. Die registerbasierten Zwischencodeformate sind:

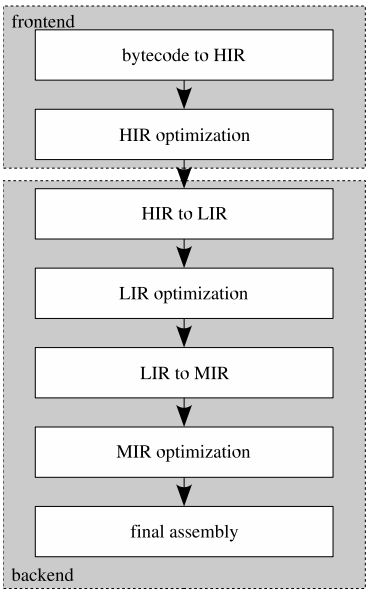


Abbildung 37: Stufen des JikesRVM optimierenden Übersetzers [21]

- *HIR* - High Level Intermediate Representation
- *LIR* - Low Level Intermediate Representation
- *MIR* - Machine-Specific Intermediate Representation.

In der ersten Phase der Übersetzung wird die HIR-Form aus dem vorliegenden Bytecode gebildet. Bei diesem Vorgang wird eine erweiterte Blockstruktur für eine Methode ermittelt, eine Ausnahmetabelle erzeugt (*exception table*) und HIR-Instruktionen entsprechend

des Bytecodes angelegt. Weiterhin bestimmt der Übersetzer in dieser Phase die Typinformationen. Parallel zur Erzeugung dieser Zwischencoderepräsentation führt der Übersetzer diverse Optimierungen durch, wozu z.B. *Copy Propagation*, *Constant Propagation* und *Dead Code Elimination* zählen. Die genannten Optimierungen werden in dieser Phase in einer vergleichsweise einfachen Art durchgeführt, unter anderem um die HIR-Darstellung möglichst platzsparend zu gestalten. Es kann durchaus vorkommen, dass diese Optimierungen in einer späteren Phase erneut zur Anwendung kommen.

Die resultierende HIR-Form ist in der Art ihrer Instruktionen nah an dem Java-Bytecode orientiert [39], wobei zwei nennenswerte Unterschiede genannt werden können. Zum einen arbeiten HIR-Instruktionen auf symbolischen Registeroperanden, nicht auf dem impliziten Stack. Zum Anderen enthalten die HIR-Instruktionen spezielle Operatoren, die explizite Laufzeitüberprüfungen (z.B. `ArrayIndexOutOfBound-Checks`) repräsentieren. Auf der HIR-Form werden wiederum Optimierungen durchgeführt, welche in drei Klassen zu unterteilen sind - *lokale Optimierungen*, *datenflussunabhängige Optimierungen* und *In-line-Erweiterung von Methodenaufrufen*. Als Beispiele können in diesem Zusammenhang die *Common Subexpression Elimination* und eine vom Datenfluss unabhängige, stark vereinfachte Escape-Analyse ([45]) genannt werden.

Nach Abschluss der Optimierungen wird die LIR-Form aus der HIR-Form abgeleitet. Dabei werden die allgemeineren HIR-Instruktionen in spezielle JikesRVM - Operationen überführt. Auf der LIR-Darstellung werden nun weitere Optimierungen durchgeführt, wobei aufgrund der ähnlichen Struktur beider Repräsentationen auch Algorithmen zur Anwendung kommen können, die bereits auf der HIR-Form angewendet wurden. Da die LIR-Form jedoch bis zu einem Faktor 2-3 größer als die entsprechende HIR-Form sein kann (Grund ist die höhere Anzahl von speziellen Instruktionen der JikesRVM), wird bei der Verteilung der Optimierungsverfahren auf Ausgewogenheit geachtet, um einen höchstmöglichen Nutzen aus den Verfahren zu ziehen. Beispielsweise wird die Eliminierung gemeinsamer Teilausdrücke auch auf der LIR durchgeführt. Zum Abschluss dieser Phase wird ein Abhängigkeitsgraph gebildet, dessen Knoten entsprechende LIR-Instruktionen darstellen. Eine Kante repräsentiert eine Abhängigkeit zwischen zwei Instruktionen.

In der letzten Phase der Übersetzung wird die auf eine bestimmte Zielarchitektur ausgerichtete MIR-Form erzeugt, wobei Information über Aufbau und Architektur der Plattform benötigt werden. Der in der vorhergehenden Phase erzeugte Abhängigkeitsgraph einer Methode wird nun in eine Gruppe von Bäumen überführt. Die Bäume dienen als Eingabe für das *BURS – Bottom-Up Rewriting System*, welches aus der Arbeit von [29] abgeleitet wurde. BURS ist ein System zur Erzeugung spezifischer und an eine spezielle Architektur angepasster Code-Generierungsprogramme. Die Abhängigkeit zur Architektur repräsentiert eine Grammatik mit entsprechenden Regeln. Eine Grammatik für die Erzeugung eines PowerPC kompatiblen Generierungsprogrammes benötigt beispielsweise etwa 300 Regeln. Nach Erzeugung der MIR werden die symbolischen Register auf physikalische Register abgebildet und für jede Methode ein *Prolog* (zum Anlegen eines Methodenrahmens) und ein *Epilog* (zum Wiederherstellen gespeicherter Register sowie Entfernen des Methodenrahmens) hinzugefügt. Diese Bereiche sind bei synchronisierten Methoden auch für das Setzen und Löschen der Sperren (*Locks*) zuständig.

8.4 SafeTSA-Übersetzer Konsumentenseite

Das SafeTSA-Übersetzungsprogramm der Konsumentenseite wurde als alternativer Teil des optimierenden Übersetzers in die JikesRVM integriert. Anwendungen, die im SafeTSA-Format übertragen werden, nutzen also das gleiche Laufzeitsystem wie ein in Java-Bytecode übermitteltes Programm in der JikesRVM. Im Unterschied werden die Klassen im SafeTSA-Format jedoch durch ein spezielles Einleseprogramm geladen und durch den SafeTSA-Übersetzer bis in die LIR-Form übersetzt. Die Weiterbearbeitung der LIR- zur MIR-Form und die Ausgabe des Programmes in Maschinensprache führt wieder der optimierende Übersetzer der JikesRVM durch.

In diesem Abschnitt werden die Komponenten und die Funktionsweise in der Übersicht dargestellt, einen Überblick stellt Abbildung 38 dar. Die ausführliche Beschreibung des SafeTSA-Übersetzers und dessen Integration kann aus [9] entnommen werden.

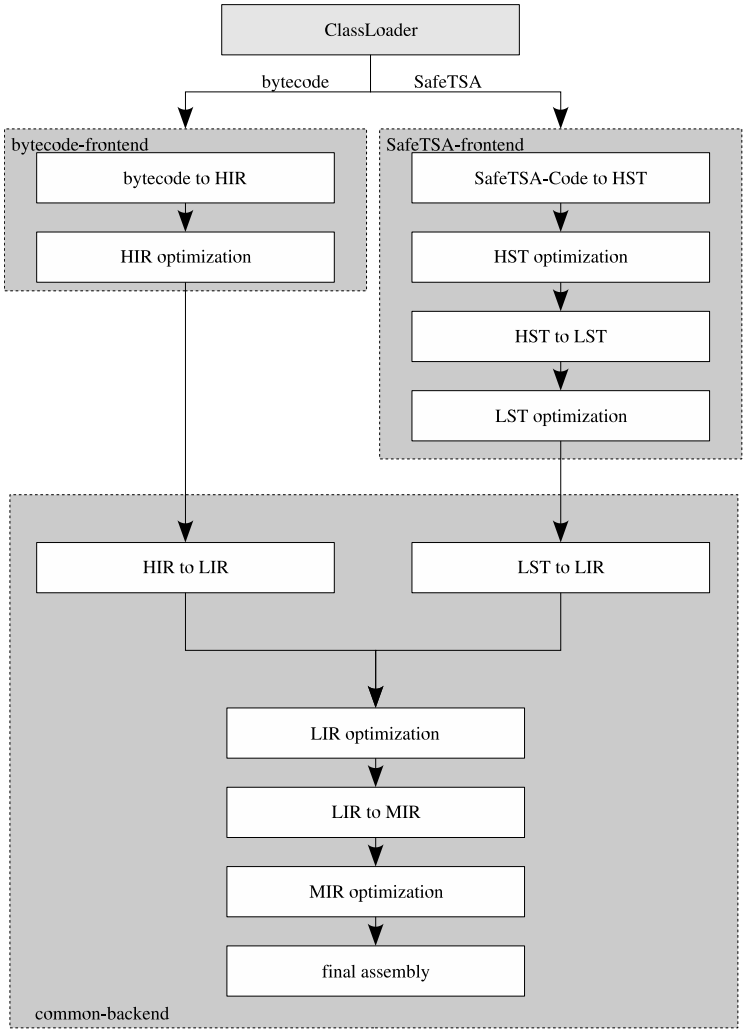


Abbildung 38: Anpassung SafeTSA-Übersetzer

Die mit dem SafeTSA-Übersetzer erweiterte JikesRVM (im folgenden bezeichnet Jikes-

RVM immer die erweiterte Version) kann mit Klassendateien in beiden Formaten arbeiten, Klassendateien in der Java-Bytecode-Repräsentation und Klassendateien im SafeTSA-Format. Der Vorteil dieser Fähigkeit liegt darin, dass Programme auch dann ausgeführt werden können, wenn sie nur zum Teil im SafeTSA-Format übertragen werden. Insbesondere bei Verwendung von Standardklassen der Java API kann somit auf bereits vorliegende Bytecode-Klassen zugegriffen werden. Für eine bessere Unterscheidung der SafeTSA- und der ursprünglichen JikesRVM Komponenten wurden die Module des SafeTSA-Übersetzers mit dem Prefix `ST` gekennzeichnet (im Unterschied zu `VM` bei dem optimierenden Übersetzer).

Wird nun eine Klasse in die JikesRVM geladen, so überprüft der Klassenlader (`VM_ClassLoader`) zunächst, ob im Klassenpfad eine entsprechende Klasse im SafeTSA-Format zu finden ist. Liegt diese anhand der Dateinamenerweiterung[23] zu unterscheidende Klasse nicht vor, so lädt der Klassenlader die Klasse im Java-Bytecode. Kann die Klasse auch in diesem Format nicht gefunden werden, folgt eine entsprechende Ausnahme (`NoClassDefFoundError`). Nachdem eine Klasse erfolgreich im SafeTSA-Format geladen wurde, beginnt das Einleseprogramm (`ST_Parser` - SafeTSA-Parser) mit dem Wiederaufbau des SafeTSA-Kontrollstrukturbaumes. Wird das SafeTSA-Binary-Format verwendet, so überprüft der SafeTSA-Parser hierbei die Typ- und Referenzsicherheit der vorliegenden Klasse.

Analysiert der SafeTSA-Parser eine Klasse, so legt er eine Typentabelle an, in der alle von der Klasse verwendeten primitiven Daten- und Referenztypen eingetragen werden. Weitere Informationen über die Klasse werden in entsprechende Tabellen des JikesRVM Übersetzungssystems eingetragen (z.B. Modifikatoren). Für alle Elemente der Klassen, Typen und die Klassen selbst werden gleiche Darstellungen verwendet, wie für Java-Bytecode. Allerdings enthalten die entsprechenden Darstellungen ein Kennzeichen, dass sie als Elemente einer SafeTSA-Darstellung auszeichnet. Bei Methoden existiert eine Referenz auf die SafeTSA-Repräsentation der Methode.

Im folgenden Schritt bildet der SafeTSA-Übersetzer für jede zu übersetzende Methode die so genannte High-Level-Form des SafeTSA-Formates (*HST* - High Level SafeTSA), welche SafeTSA insbesondere um Instruktionen erweitert, mit denen Zugriffe auf Felder und Methoden nicht geladener Klassen aufgelöst werden (*resolve*-Operationen, siehe auch Kapitel 7.2). Weiterhin werden in dieser Darstellung nicht notwendige Typkonvertierungen entfernt (z.B. *downcast*-Operationen) und Zugriffe auf statische Elemente einer Klasse durch spezielle Instruktionen ersetzt. Im Anschluss führt der Übersetzer einige Optimierungen auf der HST-Darstellung durch, unter anderem die Entfernung gemeinsamer Teilausdrücke.

Nach der High-Level-Form wird für die zu übersetzende Methode die Low-Level-Form erzeugt (*LST* - Low Level SafeTSA). In dieser Darstellung wird das SafeTSA-Format näher an die Struktur der JikesRVM gebracht, indem z.B. die Adressberechnung von Feld- und Methodenzugriffen entsprechend der JikesRVM in das Format eingefügt werden (Adressberechnung anhand von *Offsetwerten*). Weiterhin werden Indexüberprüfungen, Typkonvertierungen und Objekterzeugungen in die von der JikesRVM verwendeten Instruktionen überführt sowie die Speicherzugriffe des HST-Formates durch entsprechende *load*- und *store*-Befehle ersetzt. Auch auf der LST-Darstellung können weitere Optimie-

[23] *sta* für SafeTSA-ASCII und *stb* für SafeTSA-Binary.

rungen durchgeführt werden.

In der letzten Phase der SafeTSA-Übersetzung wird schließlich die LST-Form in die LIR-Darstellung der Methode überführt, wobei im wesentlichen die Kontrollstrukturen durch Sprungbefehle der LIR-Form ersetzt, LST-Instruktionen auf entsprechende LIR-Befehle und verwendete Werte auf entsprechende virtuelle Register abgebildet werden. Weiterhin werden in dieser Darstellung die in der SSA-Darstellung verwendeten Phi-Instruktionen endgültig aufgelöst. Aus der LIR-Repräsentation der Methode kann nun der optimierende Übersetzer der JikesRVM den für die Zielarchitektur passenden Maschinencode erzeugen.[24]

8.4.1 Steuerung der SafeTSA Komponenten

Die Aktivierung des SafeTSA-Übersetzers im Übersetzersystem der JikesRVM erfolgt über Optionen der Kommandozeile. Die JikesRVM bietet während der Entwicklung hierfür einen Mechanismus, bei dem alle möglichen Optionen in einem festgelegten Format in speziellen Dateien abgelegt werden (`BooleanOptions.dat` und `ValueOptions.dat`). Die Inhalte dieser Dateien werden beim Übersetzungsprozess der JikesRVM in entsprechende Java-Objekte umgewandelt, übersetzt und stehen später dem Laufzeitsystem zur Verfügung. Die Aktivierung des SafeTSA-Übersetzers erfolgt automatisch.

8.5 Erweiterung der JikesRVM

8.5.1 Einlesen annotierter SafeTSA-Klassen

In der JikesRVM werden Klassen immer zum Zeitpunkt ihrer ersten Verwendung geladen (*Late Binding*). Wird eine Klasse im SafeTSA-Format geladen, so ist der `ST_Parser` für das Einlesen und den Wiederaufbau der SafeTSA-Struktur verantwortlich. In Bezug auf die Arbeit mit annotierten SafeTSA-Klassen ist also an erster Stelle der Parser anzupassen, um die annotierten Informationen wieder korrekt in die SafeTSA-Struktur einzuarbeiten. Da neben der vorgestellten Annotationstechnik prinzipiell eine Reihe weiterer Annotationsformen einsetzbar sind, wurde bei der Erweiterung auf zukünftige Techniken Rücksicht genommen. Insbesondere wurde die Arbeit des `ST_Parser`, die Analyse und der Wiederaufbau, in den Annotationen zugeordnete Bereiche aufgeteilt. Bei diesem Modell übernimmt ein so genannter `Pre_Parser` die Analyse des SafeTSA-Dateikopfes (*Header*) und startet mit den gegebenen Informationen einen auf die vorliegende Annotationstechnik spezialisierten Parser. Somit können verschiedene Formen der Annotation sowie nicht annotierte SafeTSA-Klassen gleichzeitig verwendet werden. Liegt eine SafeTSA-Datei ohne Annotation vor, so wird der ursprüngliche `ST_Parser` gestartet. Für die hier behandelte Annotation von Escape-Informationen wurde der spezialisierte `ST_Escape_Parser`[25] entwickelt. Abbildung 39 zeigt eine Übersicht über das Modell.

Erweiterter Parser für Escape-Annotationen Der `ST_Escape_Parser` ist von seiner Arbeitsweise direkt aus dem ursprünglich in der JikesRVM verwendeten `ST_Parser` abgeleitet. Eine Klasse im SafeTSA-ASCII-Format wird eingelesen, analysiert und der Kontrollstrukturbaum von SafeTSA (mit Basisblöcken und Instruktionen) aufgebaut. Im Unter-

[24] Derzeit werden PowerPC und Intel Pentium unterstützt.

[25] Unterstützt SafeTSA-ASCII-Format.

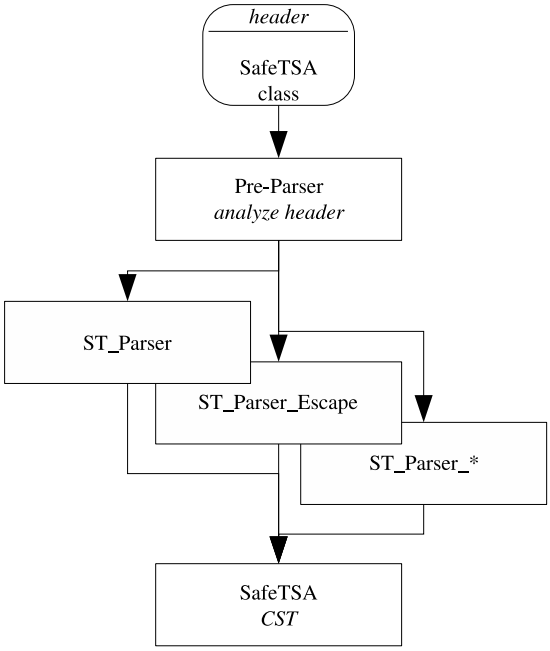


Abbildung 39: Anpassung ST_Parser

schied zum herkömmlichen Parser unterstützt der `ST_Escape_Parser` das erweiterte Typmodell und die erweiterten Instruktionen, welche für die Modellierung der Annotation als Grundlage dienen. Während des Einlesens kann der Parser somit die Escape-Eigenschaft aus den Instruktionen und den verwendeten Typen ableiten und wieder den Objekten und Parametern zuordnen. Die ermittelte Escape-Eigenschaft wird im Kontrollstrukturbaum, speziell in den für die HIR-Darstellung verwendeten Instruktionen abgespeichert. Hierbei besitzt jede Instruktion die Escape-Eigenschaft des der Instruktion zugehörigen Objektes bzw. Parameters. Weiterhin wird die Escape-Eigenschaft der formalen Parameter in der Repräsentation der entsprechenden Methode (`ST_Method`) gespeichert, wobei auch hier der nullte formale Parameter die Referenz auf *this* darstellt.

Da die Escape-Eigenschaft *escape* auf das ursprüngliche SafeTSA-Maschinenmodell[26] abgebildet wurde, kann der `ST_Escape_Parser` in diesem Fall auch nicht-annotierte SafeTSA-Dateien verarbeiten. Hier wird lediglich jedem Objekt und jedem Parameter die Escape-Eigenschaft *escape* zugewiesen. Bei anderen Annotationstechniken kann diese Möglichkeit unter Umständen nicht zur Verfügung stehen.

Zusätzlich zum herkömmlichen Parser trägt der `ST_Escape_Parser` in der ebenfalls angelegten Typentabelle alle sich durch die Erweiterung des Maschinenmodells ergebenden Typen ein. Somit wird für jeden neuen Typ, welcher standardmäßig die Escape-Eigenschaft *escape* darstellt, auch der entsprechende Typ *may* und *bound* registriert. Für alle Objekte und Parameter, die einem der Escape-Eigenschaft *may* zugeordneten Typ angehören, ist zu bemerken, dass diese Escape-Eigenschaft in der folgenden interprozeduralen Escape-Analyse entsprechend durch *bound* oder *escape* ersetzt wird. Folglich werden die Typen

[26]Der Escape-Typ *escape* für sich betrachtet entspricht dem ursprünglichen, nicht erweiterten SafeTSA-Maschinenmodell.

der betroffenen Objekte und Parameter neu zugeordnet und die Einträge der Tabelle im Verlauf der Analyse aktualisiert.

8.5.2 Interprozedurale Escape-Analyse

Die interprozedurale Escape-Analyse wird durch den SafeTSA-Übersetzer auf der HST-Darstellung einer zu übersetzenden Methode ausgeführt und die Ergebnisse in dem Kontrollstrukturbaum abgespeichert. Für formale Parameter erfolgt die Speicherung abweichend in der Repräsentation der Methode (`VM_Method` bzw. `ST_Method`). Im weiteren Verlauf der Übersetzung wird die Escape-Eigenschaft von Objekten und Parametern mit in die folgenden Darstellungen übertragen, wodurch eine auf der Escape-Analyse basierende Optimierung an der jeweils geeigneten Stelle (z.B. auf der LST-Darstellung) erfolgen kann. Der Start der Analyse ist hierbei nicht von annotierten Escape-Informationen abhängig, so dass die interprozedurale Escape-Analyse beliebig gegen einen anderen Algorithmus ausgetauscht werden kann. Für den Zweck des Vergleichs des Laufzeitverhaltens wurde tatsächlich eine vollständige intra- und interprozedurale Analyse an dieser Stelle integriert, die alternativ aktiviert werden kann.

Während die Aktivierung der interprozeduralen Escape-Analyse durch eine Option der JikesRVM mittels `BooleanOptions.dat` erfolgt, finden bei der Übersetzung weitere Steuerungsmechanismen statt. An erster Stelle wird die zu untersuchende Methode mit einem Status gekennzeichnet, wobei als Werte *nicht analysiert*, *in Analyse* und *vollständig analysiert* zur Verfügung stehen. Jede Methode ist zunächst *nicht analysiert* und tritt zu Beginn in den Status *in Analyse*. Im weiteren Verlauf wird dieser Status zur Identifizierung von Rekursionen im interprozeduralen Algorithmus verwendet, wenn die Escape-Eigenschaft formaler Parameter an Aufrufstellen von Methoden festzustellen ist. Nach Abschluss der Analyse erhält die Methode den Status *vollständig analysiert.*

Escape-Kontrolleinheit Ist die interprozedurale Escape-Analyse abgeschlossen, wird die untersuchte Methode bei einer in das JikesRVM Laufzeitsystem integrierten Escape-Kontrolleinheit (`ST_EscapeController`) registriert. Die Kontrolleinheit ordnet nun die Methode dem Typ der Klasse zu, welche die Methode implementiert und bietet eine Möglichkeit zur Registrierung entsprechender Methodenoptimierungen. Hierbei werden jene Optimierungen an die Kontrolleinheit gemeldet, die von der Escape-Eigenschaft der formalen Parameter der untersuchten Methode abhängig sind. Trifft nun ein Ereignis ein, welches die Escape-Eigenschaft der formalen Parameter bezüglich einer Aufrufstelle nachträglich ändern könnte, z.B. indem eine neuere Implementierung der aufgerufenen Methode vorliegt, so reagiert die Kontrolleinheit entsprechend. Das Ereignis wird durch die speziell im Laufzeitsystem implementierten Wächter (`ST_Guard`) gemeldet, wobei es im besten Fall keine Änderung der Escape-Eigenschaft formaler Parameter nach sich zieht. In dieser Situation hat das Ereignis keinen Einfluss auf die durchgeführten Optimierungen und die Escape-Kontrolleinheit braucht in dem Zusammenhang nichts weiter zu tun. Im ungünstigen Fall allerdings ändert sich die Escape-Eigenschaft eines formalen Parameters einer registrierten Methode und die Kontrolleinheit muss entsprechend reagieren. Sie prüft zunächst die Liste der zugehörigen Methodenoptimierungen und veranlasst für diese Methoden eine neue Übersetzung einschließlich einer neuen interprozeduralen Escape-Analyse unter Einbezug der veränderten Escape-Informationen. Bei Ereignissen, die eine Optimie-

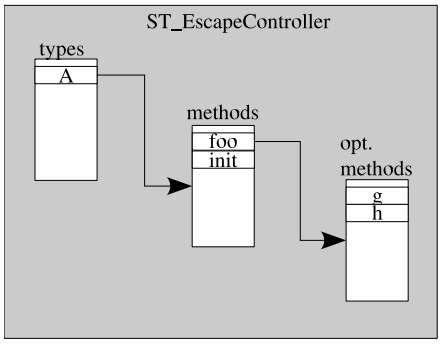


Abbildung 40: Verwaltung von Methoden in der Escape-Kontrolleinheit

rung verhindern (z.B. die Verwendung der Reflektion), erhalten alle formalen Parameter und lokalen Objekte betroffener Methoden die Escape-Eigenschaft *escape*, wodurch die Optimierung ausgeschlossen wird.

Der Prozess der neuen Übersetzung durch den SafeTSA-Übersetzer der JikesRVM kann auf zwei Wegen realisiert werden. Zum Einen kann beim ersten Übersetzen einer Methode das Original dieser Methode in der HST-Darstellung gespeichert und für den weiteren Vorgang eine Kopie verwendet werden. Ist zu einem späteren Zeitpunkt eine erneute Übersetzung der Methode notwendig, so kann hierfür eine neue Kopie angelegt bzw. bei auszuschließender Optimierung das Original selbst übersetzt werden. Zum Anderen kann, wie in dieser Arbeit implementiert, die Methode durch einen unabhängig arbeitenden Klassenlader erneut geladen werden. Der Klassenlader lädt in diesem Fall tatsächlich nur die Implementierung der Methode, ohne jedoch die Klasse selbst zu initialisieren. Die Implementierung der Methode steht in der HST-Darstellung zur Verfügung und kann folgend durch den SafeTSA-Übersetzer übersetzt werden.

Abbildung 40 zeigt den Verwaltungsaufbau der Escape-Kontrolleinheit. Beim Registrieren einer Methode *foo* wird zunächst der Typ *type* der *foo* definierenden Klasse ermittelt. Existiert für *type* noch kein Eintrag in der internen Typentabelle, so wird ein Eintrag und eine leere Methodentabelle für *type* erzeugt. Weiterhin wird die zu registrierende Methode *foo* in die Methodentabelle von *type* eingetragen und eine leere Tabelle für die Registrierung weiterer in Abhängigkeit von *foo* optimierter Methoden erzeugt. Diese Tabelle wird dem Eintrag *foo* zugeordnet.

Wächter Die Wächter, welche die Escape-Kontrolleinheit über bestimmte Ereignisse während der Ausführung eines Programmes informieren, arbeiten nach den folgenden Richtlinien:

> `ST_ClassLoaderGuard`**:** Dieser Wächter reagiert sobald der Klassenlader der JikesRVM (`VM_ClassLoader`) mit dem Laden einer neuen Klasse beauftragt wurde. Er meldet den Typ `VM_Type` der Klasse an die Escape-Kontrolleinheit. Wurde der Ladevorgang durch die Kontrolleinheit ausgelöst, erfolgt keine Benachrichtigung.
>
> `ST_JNIGuard`**:** Der JNI-Wächter meldet Ereignisse an die Kontrolleinheit, sobald eine Klasse die Funktionen der JNI-Schnittstelle der JikesRVM benutzt. Dies ist zum einen der Fall, wenn eine Methode mit dem speziellen Übersetzer `VM_JNICompiler`

übersetzt wird, da die Methode auf Funktionen der `VM_JNIFunctions` zugreift. Findet dieser Übersetzer Verwendung, so meldet der Wächter den betroffenen Typ `VM_Type` der Klasse, welche die übersetzte Methode definiert. Zum anderen reagiert der Wächter auf die Verwendung der so genannten *VM syscalls*. Sobald ein *VM syscall* während der Ausführung eines Programmes verwendet wird, meldet der Wächter dies der Kontrolleinheit. Hier wird auf die Übermittlung des betroffenen Typs der zugehörigen Klasse verzichtet, da die Verwendung von *VM syscalls* zu einer globalen pessimistischen Annahme führt (alle Objekte können prinzipiell fliehen, da ein uneingeschränkter Zugriff auf den Speicher der JikesRVM möglich ist).

`ST_ReflectionGuard`: Der Wächter der Reflektion überwacht die Verwendung der Klassen der *Reflection API*, abgesehen von dem expliziten Laden von Klassen durch benutzerdefinierte Klassenlader oder `Class.forName()`. Deren Ereignisse werden bereits durch den `ST_ClassLoaderGuard` gemeldet. Für die weiteren Funktionen der Reflektion gilt, dass für jede Verwendung zuvor ein Zugriff auf eine der Klassen `Class`, `Method` oder `Field` der *Reflection API* erfolgt. Üblicherweise beginnt ein Programm mit dem Zugriff auf das statische Feld `class` eines Objektes oder einer Klasse. Der Wächter der Reflektion meldet jede Verwendung der zuvor genannten drei Klassen der *Reflection API* an die Kontrolleinheit unter Angabe des Typs `VM_Type` der Klasse, auf die zugegriffen wird. Eine Überprüfung auf entsprechende Zugriffe während der Übersetzung durch den optimierenden SafeTSA-Übersetzer ist hier zwar möglich, jedoch aufgrund der parallelen Verwendung von Klassen im Java-Bytecodeformat nicht sinnvoll.

9 Ergebnisse

9.1 Messumgebung

Die Messungen wurden auf einem *PowerMac* der Firma Macintosh durchgeführt. Das System verfügt über einen *PowerPC G4* Prozessor, 1,5 GB Hauptspeicher (RAM) und 256 KB Level 2 Cache. Bei dem verwendeten Betriebssystem handelte es sich um *Mandrake Linux 7.1* mit dem Kernel *2.4.17-1mdk*. Um eine Beeinflussung der Messungen durch andere Benutzer zu vermeiden, wurde das System im Einzelbenutzermodus betrieben.

Die aktuelle Version der verwendeten JikesRVM wurde mit dem Profil `FullOptNoGC` erzeugt. Bei diesem Vorgang werden die Klassen der virtuellen Maschine durch die JikesRVM unter Verwendung des optimierenden Übersetzers übersetzt und in das *Bootimage* eingebettet. Weiterhin ist bei diesem Profil der Speicherbereiniger deaktiviert (*no **g**arbage **c**ollector*), welcher einen Einfluss auf die Laufzeit des Übersetzers bzw. der Analyse nehmen könnte.

Als Testprogramme dienten die Sektionen 2 und 3 der *Java Grande Benchmark Suite – JGF*[20], einige in der Programmiersprache Java entwickelte, rechenintensive und frei verfügbare Anwendungen [3]. Die Sektion 1 dieser Programmgruppe beinhaltet hauptsächlich einfache Berechnungen und Tests, welche für eine Escape-Analyse keine wertvollen Erkenntnisse liefern können. Alle Anwendungen wurden vierfach gestartet und aus der Gruppe von Messergebnissen der Mittelwert ermittelt.

Die Produzenten- und Konsumentenseite des SafeTSA-Systems wurde in einzelne Module unterteilt, welche über jeweils unterschiedliche Profile mit/ohne Analyse bzw. Annotation verfügen. Für die Messung wurden die Module in der entsprechenden für die Messung erforderlichen Zusammensetzung kombiniert. Folgende Module standen zur Verfügung:

- SafeTSA-Übersetzer der Produzentenseite ohne Escape-Analyse und Annotation
- SafeTSA-Übersetzer der Produzentenseite mit vollständiger Escape-Analyse (Whaley) ohne Annotation
- SafeTSA-Übersetzer der Produzentenseite mit erweitertem Maschinenmodell, intraprozeduraler Escape-Analyse (Whaley) und Annotation
- Java-Bytecode Übersetzer
- JikesRVM nur mit Ausführung von Java-Bytecode
- JikesRVM mit SafeTSA-Integration
- JikesRVM mit SafeTSA-Integration, Unterstützung von Annotationen (`ST_Escape_Parser`) und interprozeduraler Escape-Analyse
- JikesRVM mit SafeTSA-Integration und vollständiger Escape-Analyse (Whaley)[27]

Im folgenden werden die Ergebnisse für die Übersetzung der Benchmarks mit dem Produzentensystem, der Einfluss auf die zu übertragenden Dateien im Zwischencodeformat SafeTSA und die Ausführung unter Verwendung des optimierenden Übersetzers der JikesRVM dargestellt.

[27] Der vollständige Algorithmus von Whaley ist im Anhang dargestellt.

Klasse	Intra- und Interprozedurale Analyse	Intraprozedurale Analyse und Annotation	Keine Escape-Analyse
JGFEuler	5,91	5,87	5,85
Tunnel	5,90	5,87	5,83
JGFMoldyn	5,34	5,31	5,28
md	38,59	15,15	15,00
RayTracer	10,85	9,86	9,68
SearchGame	10,27	8,71	8,58
TransGame	7,17	7,09	7,01
AppDemo	18,93	18,87	18,79
RatePath	8,68	8,49	8,14
ReturnPath	7,04	7,02	6,95
Universal	6,06	6,04	5,97

Tabelle 4: Übersetzungszeiten (sek) der Produzentenseite (Auszug)

9.2 Produzentenseite und SafeTSA-Dateien

In Zusammenhang mit dem SafeTSA-Übersetzer der Produzentenseite wurden das Zeitverhalten und mit besonderer Aufmerksamkeit die Vergrößerung der ausgegebenen Zwischencodedateien im SafeTSA-Format betrachtet. Für diesen Zweck wurden auf der Produzentenseite alle Klassen der JGF Benchmark Suite mit und ohne Annotation von Ergebnissen der Escape-Analyse erzeugt, wobei als Implementierung die in Kapitel 8.2.1 beschriebene intraprozedurale Escape-Analyse von Whaley diente. Weiterhin wurden qualitative Messungen mit der vollständig auf der Produzentenseite implementierten Escape-Analyse (intra- und interprozedural) durchgeführt. Tabelle 4 präsentiert einen Auszug der Übersetzungszeiten der Produzentenseite für Klassen der JGF-Sektion 3 jeweils mit vollständiger Escape-Analyse (intra- und interprozedural), Escape-Analyse mit Annotation (intraprozedural) und ohne Escape-Analyse oder Annotation.

Die Überprüfung des erzeugten Zischencodes zeigt, dass die SafeTSA-Klassendateien im Vergleich zu anderen Annotationstechniken (z.B. [17]) nur minimal wachsen. Für die Klassen der JGF-Benchmarks gerechnet, beträgt der Zuwachs nur zwischen **0.03%** und **0.28%**, im Durchschnitt vergrößern sich die Dateien um **0.16%**.

Tabelle 5 stellt einen Überblick über die relative Vergrößerung der Klassen der JGF-Benchmarks dar. Dabei bezeichnet `op` die Anzahl der in der Klasse vorkommenden Operanden, `par` steht für die Summe der formalen Parameter aller Methoden `meth` einer zu übersetzenden Klasse. Weiterhin bezeichnen `bit` und `byte` den zusätzlichen Platz, den die Annotation insgesamt für eine Klasse beansprucht. Der Wert berechnet sich aus den vorhergehenden Zahlen. Abschließend stellt die Spalte `rel` den relativen Platzbedarf der Annotation dar, wobei sich diese Größe auf das platzsparendere SafeTSA-Binary-Format bezieht. Für das Arbeitsformat SafeTSA-ASCII ergeben sich bedeutend kleinere Werte, da eine Datei in diesem Format wesentlich mehr Platz in Anspruch nimmt.

class	op	par	meth	bit	byte	rel
JGFEuler	34	0	7	41	6	0.22%
Tunnel	11	14	11	36	6	0.03%
Statev.	0	4	6	10	2	0.18%
Vector2	0	1	3	4	1	0.21%
AppDemo	14	6	19	39	5	0.11%
JGFMonte.	25	0	7	32	4	0.20%
RatePath	38	12	16	66	9	0.17%
JGFSearch	12	0	7	19	3	0.12%
TransGame	13	0	11	24	3	0.10%
JGFMolDyn	25	0	7	32	4	0.21%
md	8	0	3	11	2	0.04%
JGFRayTr.	29	0	7	36	5	0.17%
RayTracer	69	16	11	96	12	0.19%
Vec	4	19	17	40	5	0.21%
JGFAll	7	1	2	10	2	0.28%

Tabelle 5: Übersicht des zusätzlichen Platzbedarfes der Annotation

9.3 Konsumentenseite JikesRVM

Durch die Integration der vorgestellten Annotationstechnik in die JikesRVM war es möglich, den Zeitaufwand für den erweiterten Parser und die auf der Annotation basierende interprozedurale Escape-Analyse zu messen. In diesem Zusammenhang konnte der zeitliche Vorteil ermittelt werden, der mit diesem Verfahren im Vergleich zu einer vollständig zur Laufzeit durchgeführten Escape-Analyse gewonnen wird. Diese zusätzliche Zeit steht entsprechend Optimierungsalgorithmen zur Verfügung, die auf Ergebnissen der Analyse arbeiten.

Für die Messungen wurde zum Einen die vollständige Escape-Analyse von Whaley und Rinard mit intra- und interprozeduralem Algorithmus in der JikesRVM implementiert. Für diese Variante wurden die auszuführenden Programme im SafeTSA-Zwischencodeformat ohne entsprechende Annotationen übertragen und durch den herkömmlichen `ST_Parser` eingelesen. Anschließend führte der optimierende Übersetzer die vollständige Escape-Analyse durch. Zur Feststellung der Analyseergebnisse wurde ein Protokoll erzeugt und gespeichert, wobei für die Ausgabe die `debug`-Funktion der JikesRVM Verwendung fand. Die Zeitmessung und die Protokollausgabe fand in getrennten Durchläufen statt, um die Ausführungszeiten nicht durch die Protokollierung zu beeinflussen.

Zum Anderen wurden die Klassen lediglich mit der intraprozeduralen Analyse übersetzt und die Ergebnisse annotiert. In dieser Variante bearbeitete der `ST_Escape_Parser` die annotierten SafeTSA-Klassen. Der optimierende Übersetzer der JikesRVM führte hier die in Kapitel 6.4 beschriebene interprozedurale Escape-Analyse durch, wobei die annotierten Informationen der intraprozeduralen Analyse entsprechend Verwendung fanden. Auch bei diesem Verfahren wurden die gewonnenen Ergebnisse in einem Protokoll ausgegeben und festgehalten. Die Messung war ebenfalls in einem getrennten Durchlauf durchgeführt worden. Die Zeiten, die der erweiterte Parser für die Auswertung der annotierten Analyseinformationen aufwenden musste, wurden separat erfasst und in der Bewertung

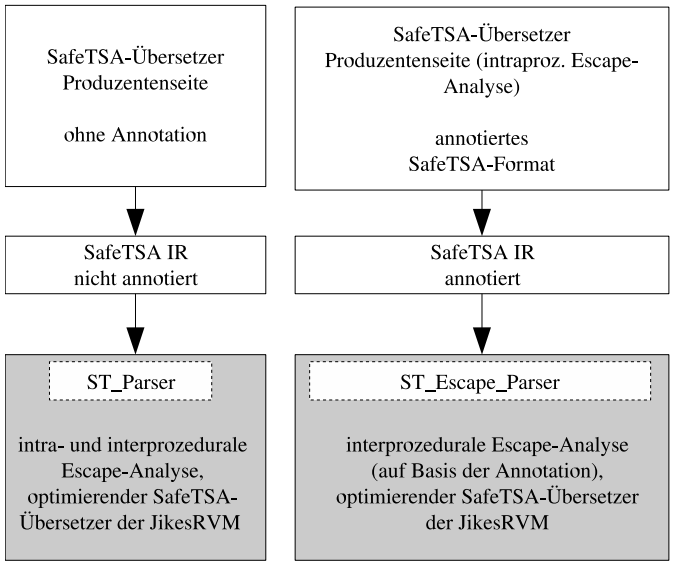


Abbildung 41: Übersicht der Konfigurationen

entsprechend berücksichtigt. Eine Gegenüberstellung beider Verfahren ist in Abbildung 41 dargestellt.

Abbildung 42 zeigt die absoluten Übersetzungszeiten des optimierenden Übersetzers für die Varianten der auf der Konsumentenseite durchgeführten Escape-Analyse, die jeweils mit annotierten und nicht-annotierten Klassen im SafeTSA-Zwischencodeformat arbeiten (*annotierte interprozedurale Escape-Analyse* und *nicht annotierte vollständige Escape-Analyse*). Die Zeiten der annotierten interprozeduralen Escape-Analyse sind aufgeteilt in die zusätzliche Arbeitszeit des `ST_Escape_Parser` für die annotierten SafeTSA-Klassen und die Zeit der eigentlichen interprozeduralen Escape-Analyse. Zu bemerken ist, dass der Aufwand des erweiterten Parsers für die verschiedenen SafeTSA-Formate ähnlich ausfällt, da die Annotationen während des Einlesens mit einem Mechanismus gleicher Komplexität ausgewertet werden.

Die Ergebnisse zeigen, dass das vorgestellte Verfahren in jedem Fall zu einem Zeitgewinn führt, wobei die Analyseergebnisse bezogen auf die annotierten Klassen annähernd denen der vollständigen Escape-Analyse entsprechen. Im Durchschnitt identifizierte die annotierte interprozedurale Analyse über 93% der durch Whaleys Analyse gefundenen Objekte. Die gewonnene Zeit steht Optimierungen wie der Stack-Allokation oder der Synchronisationsentfernung zur Verfügung. Gerade für komplexe und große Klassen wird die vollständige Escape-Analyse sehr zeitintensiv. In diesen Klassen zeigt sich der Vorteil der Trennung von intra- und interprozeduraler Analyse verstärkt, da ein hoher Anteil der Arbeit bereits zur Übersetzungszeit durchgeführt wird. Besonders für das Programm `Euler` konnte ein Zeitgewinn mit der vorgestellten Methode erzielt werden. Die Übersetzungszeit der annotierten Escape-Analyse war hier um 0,448 s (bzw. 95,5%) schneller als die der vollständigen Escape-Analyse, was im Wesentlichen auf die hohe Anzahl zu untersuchender Reihungen zurückgeführt werden kann, deren Escape-Eigenschaft beim Annotationsverfahren bereits auf der Produzentenseite als *escape* feststellbar war.

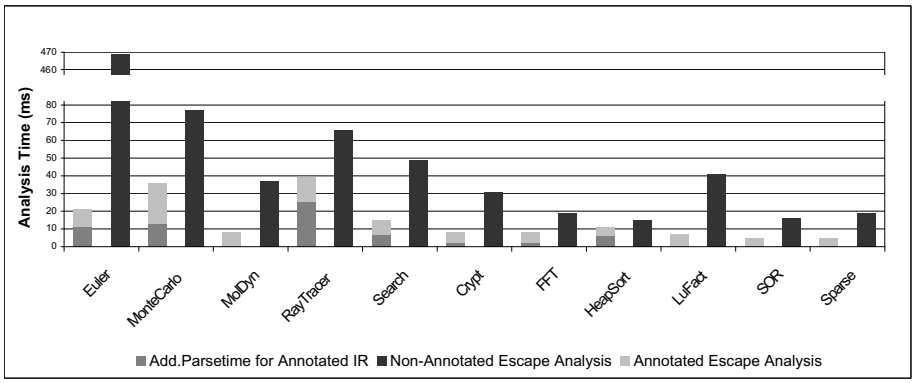


Abbildung 42: Analyse-Ergebnisse auf einem PowerPC

Zusammenfassung

In dieser Dissertation wurde eine sichere Annotation von Ergebnissen einer auf der Produzentenseite durchgeführten Escape-Analyse vorgestellt. Die Annotation basiert auf dem Zwischencodeformat SafeTSA und erweitert dessen Maschinenmodell durch die Einführung von neuen Typen und Registersätzen. Durch die Konstruktion des Zwischencodeformates und des Maschinenmodells sind die zu übertragenden Informationen vollständig verifizierbar. Eine vergleichbare Erweiterung kann auch für weitere Analysetechniken eingesetzt werden, soweit sich deren Ergebnisse durch entsprechende Typmodelle ausdrücken und somit annotieren lassen. Werden mehrere Analysetechniken unterstützt, ist jedoch mit einem höheren Aufwand und einer weiteren Vergrößerung des Zwischencodes zu rechnen. Die Vergrößerung des Zwischencodes durch die Annotation von Escape-Informationen beträgt durchschnittlich 0.16% und fällt somit im Vergleich zur gesamten Programmgröße gering aus. Ebenfalls gering ist der zusätzliche Aufwand eines optimierenden JIT-Übersetzers, welcher die annotierten Escape-Informationen auswertet und eine interprozedurale Escape-Analyse auf Basis der Annotation durchführt. Durch einen Vergleich mit einer vollständig auf der Konsumentenseite durchgeführten Escape-Analyse konnte die Effizienz des Systems gezeigt werden. Die vorgestellte Methode unterstützt weiterhin alle dynamischen Aspekte der Programmiersprache Java und der JVM (z.B. dynamisches Laden von Klassen) und stellt einen entsprechenden Mechanismus für Optimierungen bereit, durch welchen auch zur Laufzeit eines Programmes neue Informationen berücksichtigt und gegebenenfalls Optimierungen angepasst werden können.

Literatur

[1] *Advanced Programming for the Java 2 Platform, Chapter 5: JNI.* http://java.sun.com/developer/onlineTraining/Programming/JDCBook/jni.html.

[2] *Annotation.* http://en.wikipedia.org/wiki/Annotation.

[3] *Java Grande Forum Benchmark Suite, Edinburgh Parallel Computing Centre.* http://www.epcc.ed.ac.uk/computing/research_activities/java_grande/.

[4] *The Java^{TM} Virtual Machine Specification.* http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html.

[5] *Jikes^{TM} Research Virtual Machine (RVM).* http://jikesrvm.sourceforge.net/index.shtml.

[6] *JSR 175: A Metadata Facility for the Java^{TM} Programming Language.* http://www.jcp.org/en/jsr/detail?id=175.

[7] *Threads in Java, Einführung in Konzept und Anwendung.* http://www.javamagazin.de/itr/online_artikel/psecom,id,228,nodeid,11.html.

[8] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL*, pages 1–11, 1988.

[9] W. Amme. Effiziente und sichere Codegenerierung für mobilen Code. Habilitationsschrift, FSU-Jena, 2005.

[10] W. Amme, N. Dalton, M. Franz, and J. von Ronne. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. Technical report, University of California, Irvine, Information and Computer Science, Nov. 11, 2000.

[11] W. Amme, N. Dalton, M. Franz, and J. von Ronne. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'2001)*, volume 36 of *ACM SIGPLAN Notices*, pages 137–147, Snowbird, Utah, USA, June 2001. ACM Press.

[12] W. Amme and M. Franz. Effiziente Codegenerierung für mobilen Code. *Informatik Spektrum*, 26(4):237–246, 2003.

[13] L. O. Andersen. Program analysis and specialization for the C programming language. PHD-Thesis, University of Copenhagen, 1994.

[14] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2), 2005. special issue on Program Generation, Optimization, and Adaptation".

[15] M. Arnold and B. G. Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In *ECOOP 2002 - Object-Oriented Programming, 16th European Conference, Malaga, Spain, Proceedings*, volume 2374 of *Lecture Notes in Computer Science*, pages 498–524, June 2002.

[16] A. Azevedo, A. Nicolau, and J. Hummel. An annotation-aware Java virtual machine implementation. *Concurrency - Practice and Experience*, 12(6):423–444, 2000.

[17] M. Q. Beers, C. Stork, and M. Franz. Efficiently verifiable escape analysis. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 75–95. Springer, June 2004.

[18] D. Bernstein, D. Q. Goldin, M. C. Golumbic, H. Krawczyk, Y. Mansour, I. Nahshon, and R. Y. Pinter. Spill code minimization techniques for optimizing compilers. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'1989)*, pages 258–263, 1989.

[19] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. In *SIGMETRICS*, pages 25–36, 2004.

[20] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, May 2000.

[21] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In *Proceedings ACM 1999 Java Conference*, pages 129–141, San Francisco, CA, United States, June 1999. ACM.

[22] D. Buytaert, K. Venstermans, L. Eeckhout, and K. D. Bosschere. Garbage collection hints. In *HiPEAC*, pages 233–248, 2005.

[23] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'1999)*, volume 34 of *ACM SIGPLAN Notices*, pages 1–19, New York, Oct. 1999. ACM Press.

[24] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[25] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, London, UK, 1995. Springer-Verlag.

[26] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: an optimizing compiler for Java. *Software – Practice and Experience*, 30(3):199–232, Mar. 2000. Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard,

David Tarditi, Marmot: An Optimizing Compiler for Java, *Software - Practice and Experience*, Vol. 30, No. 3, March 2000, pp.199-232.

[27] M. Franz and T. Kistler. Slim Binaries. *Communications of the ACM*, 40(12):87–94, Dec. 1997.

[28] M. Franz, C. Krintz, V. Haldar, and C. H. Stork. Tamper proof annotations. Technical Report 02-10, Department of Information and Computer Science, University of California, Irvine, Mar. 2002.

[29] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[30] E. M. Gagnon and L. J. Hendren. SableVM: A research framework for the efficient execution of Java Bytecode. In *Java Virtual Machine Research and Technology Symposium*, pages 27–40. USENIX, 2001.

[31] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Proceedings of the International Conference on Compiler Construction (CC'2000)*, volume 1781 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[32] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science*, 248(1–2):147–199, 2000.

[33] J. Hannan. A type-based escape analysis for functional languages. *Journal of Functional Programming*, 8(3):239–273, May 1998.

[34] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 143–153, New York, NY, USA, 2005. ACM Press.

[35] M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in the presence of dynamic class loading. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 96–122. Springer, June 2004.

[36] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 32–43, New York, NY, USA, 1992. ACM Press.

[37] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 69–80, New York, NY, USA, 2004. ACM Press.

[38] J. Hummel, A. Azevedo, D. Kolson, and A. Nicolau. Annotating the Java bytecodes in support of optimization. *Concurrency: Practice and Experience*, 9(11):1003–1016, 1997.

[39] IBM Research. *Jikes RVM User's Manual*, v2.0.3 edition, Mar. 2002.

[40] J. Jones and S. Kamin. Annotating Java class files with virtual registers for performance. *Concurrency: Practice and Experience*, 12(6):389–406, 2000.

[41] T. Kotzmann and H. Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 111–120, New York, NY, USA, 2005. ACM Press.

[42] C. Krintz and B. Calder. Using annotation to reduce dynamic optimization time. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–167, 2001.

[43] S. Macrakis. From Uncol to ANDF: Progress in standard intermediate languages. Technical Report, Open Software Foundation Research Institute, 1993.

[44] G. C. Necula. Proof-carrying code. In N. D. Jones, editor, *Proceedings of the Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, Jan. 1997. ACM Press.

[45] U. H. Nielson, T. Riisbjerg, M. Danquah, T. Krogh, and J. Gallagher. Escape analysis in the Jikes RVM. Technical Report, Roskilde University Denmark, 2003.

[46] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'1997)*, pages 146–159, 1997.

[47] Y. G. Park and B. Goldberg. Reference escape analysis: Optimizing reference counting based on the lifetime of references. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'1991)*, volume 26, 9 of *ACM SIGPLAN Notices*, pages 178–189, New York, June 1991. ACM Press.

[48] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *CC 2001*, volume 2027 of *Lecture Notes in Computer Science*, pages 334+, 2001.

[49] F. Qian and L. Hendren. An adaptive, region-based allocator for java. In *Proceedings of the third International Symposium on Memory Management*, pages 127 – 138. ACM Press, 2002.

[50] F. Reig. Annotations for portable intermediate languages. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001. First Workshop on Multi-Language Infrastructure and Interoperability (BABEL 01).

[51] E. Ruf. Effective synchronization removal for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'2000)*, pages 208–218, 2000.

[52] A. Salcianu and M. C. Rinard. Pointer and escape analysis for multithreaded programs. In *Principles Practice of Parallel Programming*, pages 12–23, 2001.

[53] K.-D. Schmatz. *Java 2 Micro Edition*. dpunkt.verlag, 2004.

[54] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'1996)*, pages 32–41, 1996.

[55] B. Steensgaard. Thread-specific heaps for multi-threaded programs. In *ISMM '00: Proceedings of the 2nd international symposium on Memory management*, pages 18–24, New York, NY, USA, 2000. ACM Press.

[56] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[57] F. Vivien and M. C. Rinard. Incrementalized pointer and escape analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 35–46, 2001.

[58] J. von Ronne. A safe and efficient machine-independent code transportation format based on static single assignment form and applied to just-in-time compilation. PHD-Thesis, University of California, Irvine, 2005.

[59] P. Wayner. Sun gambles on Java chips: Are Java chips better than general purpose CPUs? or will new compilers make them obsolete? *j-BYTE*, 21(11):79–86, Nov. 1996.

[60] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. *ACM SIGPLAN Notices*, 34(10):187–206, 1999.

[61] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'1999)*, volume 34 of *ACM SIGPLAN Notices*, pages 187–206, Denver, CO, Oct. 1999. ACM Press.

[62] D. Yessick. Removal of bounds checks in an annotation aware JVM. In *Proceedings of the IEEE, SoutheastCon*, pages 226–228, 2002.

A Compositional Pointer and Escape Analysis for Java - Whaley, Rinard

Whaley und Rinard stellen in [61] eine Technik der Escape-Analyse für Javaprogramme vor, die sich speziell auf die Untersuchung von Objektreferenzen konzentriert. Um aus dem Verhalten von Referenzen Escape-Informationen bezüglich der referenzierten Objekte ableiten zu können, wird eine graphenbasierte Datenstruktur angelegt, der so genannte *Points-To-Escape-Graph*, welcher die Zusammenhänge zwischen den Referenzen und Objekten im Kontext der umgebenden Methoden abbildet. Die Autoren haben ihre Technik in einen lokalen und einen globalen Teil gegliedert. Die intraprozedurale Analyse (lokal) bezieht sich auf das Verhalten innerhalb einer Methode und die interprozedurale Analyse (global) untersucht Abhängigkeiten zwischen Methoden.

Definition: Sei m diejenige Methode, die aktuell analysiert wird, und I_m sei die Menge aller Methoden, die von m aufgerufen werden. Dann sei der *aktuelle Analyse-Bereich* definiert als $S_m = \{m\} \cup I_m$.

Das Verfahren setzt voraus, dass das Eingabeprogramm in der folgenden aufbereiteten Form vorliegt und sämtliche Anweisungen das folgende Format haben:

- Kopieranweisung: *k=v;*
- Speicheranweisung: $l_1.f = l_2$;
- Ladeanweisung: $l_1 = l_2.f$;
- Methodenaufruf: $l_0.op(p_1, ..., p_n)$;
- Instanziierung: $l = new\ Object(p_1, ..., p_n)$;
- Returnanweisung: *return l;*

Die Variablen l_1 und l_2 zeigen auf entsprechende Objekte. Ist l_1 bei einer Speicheranweisung eine globale Variable, spricht man von einer *globalen Speicheranweisung*. Ist l_2 bei einer Ladeanweisung eine globale Variable, spricht man von einer *globalen Ladeanweisung*. Zugriffe auf Array-Objekte erfolgen über entsprechende Lade- und Speicheranweisungen; Zugriffe auf die einzelnen Array-Elemente erfolgen über ein ausgezeichnetes Feld *elements*. Ist also a ein Array-Objekt, so kann mittels *a.elements* auf die Elemente von a zugegriffen werden. Methodenaufrufe innerhalb einer zu analysierenden Methode werden bei dem Verfahren mittels eines Kontrollflussgraphen repräsentiert. Zur Ermittlung der Menge aller aufgerufenen Methoden werden bei der interprozeduralen Analyse dann Informationen aus diesem Aufrufgraphen (*Call Graph*) verwendet.

Definition: Sei im folgenden o ein beliebiges Objekt. Wir sagen, o "flieht direkt", genau dann, wenn eine der folgenden Situationen vorliegt:

1. Eine Referenz auf o wurde als Parameter an die aktuell zu analysierende Methode übergeben.

2. Eine Referenz auf o wurde in eine statische Klassenvariable gespeichert.

3. Eine Referenz auf o wurde als Parameter an eine aufgerufene Methode übergeben und es liegen keine Infomationen darüber vor, wie diese Methode die Parameterreferenz verwendet.

4. o ist ein Thread-Objekt.

5. Eine Referenz auf o wurde mittels einer Returnanweisung an eine aufrufende Methode übergeben.

Desweiteren sagen wir, dass o genau dann flieht, wenn es über eine Sequenz von Referenzen erreichbar ist, die von einem direkt fliehendem Objekt ausgeht. D.h. es existieren ein direkt fliehendes Objekt d und Referenzen der Form $d \rightarrow o_1 \rightarrow o_2 \rightarrow \ldots \rightarrow o_k \rightarrow o$, wobei o_i beliebige Objekte sind, für $1 \leq i \leq k$.

Die Escape-Information wird aus dem aktuellen *Points-To-Escape-Graph* abgeleitet. Der *Points-To-Escape-Graph* ist definiert als:

Definition: Ein *Points-To-Escape-Graph* G ist ein Tupel der Form G=(I,O,e,r), wobei folgende Bezeichnungen gelten:

I . . . Menge der inneren Kanten

O . . . Menge der äußeren Kanten

e . . . Escape Funktion

r . . . Menge der Return-Knoten

Die Elemente des Points-To-Escape-Graphen erläutern sich wie folgt: Kanten - bezeichnet als die Menge $E = I \cup O$ - entsprechen dabei gerichteten Verbindungen zwischen Knoten, wobei N in der Folge die Menge der Knoten bezeichnet. Für jeden Knoten $n \in N$ gilt entweder $n \in N_I$ oder $n \in N_O$ ($N_I \subseteq N \supseteq N_O, N_I \cap N_O = \oslash, N_I \cup N_O = N$), wobei N_I die Menge aller inneren Knoten und N_O die Menge aller äußeren Knoten bezeichnet. Innere Knoten werden verwendet, um Objekte zu modellieren, die innerhalb des aktuellen Analyse-Bereiches erstellt wurden und auf die mittels Referenzen zugegriffen wird, die ebenfalls innerhalb des aktuellen Analyse-Bereiches erstellt wurden. Diese Referenzen werden als innere Kanten bezeichnet. Analog werden äußere Kanten definiert, um Objekte zu modellieren, die außerhalb des aktuellen Analyse-Bereiches erstellt wurden und auf die mittels außerhalb des aktuellen Analyse-Bereiches erstellter Referenzen zugegriffen wird. Diese Referenzen werden entsprechend als äußere Kanten bezeichnet. Für jede Kante $e \in E$ ist ferner eine Kennzeichnung mit einem Feld $f \in F$ möglich, wobei F die Menge aller Objektfelder bezeichnet.

Thread-Knoten sind eine Spezialform der inneren Knoten, die innere Objekte repräsentieren, welche Instanzen einer von `Java.lang.Thread` abgeleiteten Klasse sind oder als Parameter an einen Konstruktor einer von `Java.lang.Thread` abgeleiteten Klasse übergeben werden. Äußere Knoten werden ferner in globale Knoten (N_G), Return-Knoten (N_R), Klassen-Knoten (N_{CL}), Lade-Knoten (N_L) und Parameter-Knoten (N_P) unterteilt.

Globale Knoten repräsentieren Objekte, auf die eine entsprechende statische Klassenvariable während der Ausführung der derzeitig analysierenden Methode zeigt. Es existiert ein globaler Knoten für jede statische Klassenvariable. Return-Knoten repräsentieren Rückgabewerte von nicht analysierten Methoden. Für jede Rücksprunganweisung existiert genau ein Return-Knoten. Klassen-Knoten repräsentieren die statisch allokierten Klassenobjekte. Für jede Klasse existiert genau ein Klassen-Knoten, dessen Felder genau den statischen Klassenvariablen dieser Klasse entsprechen. Die Lade-Knoten werden dazu verwendet, um diejenigen Objekte darzustellen, deren Referenzen mittels einer entsprechenden Ladeanweisung geladen werden. Es existiert für jede Ladeanweisung genau ein Lade-Knoten. Parameter-Knoten repräsentieren diejenigen Objekte, auf die die aktuellen Parameter während der Ausführung der aktuell analysierten Methode zeigen. Es existiert für jeden formalen Parameter genau ein Parameter-Knoten und das Empfänger-Objekt wird dabei als der nullte Parameter behandelt.

Die Escape-Funktion e ist definiert als

$$e : N \rightarrow 2^M, e(n) = \{m_1, \ldots, m_l\}, |e(n)| \geq 0,$$

wobei M die Menge aller Methoden bezeichnet. Die Escape-Funktion zeichnet alle nicht analysierten Methoden auf, in die ein Knoten flieht. Die Menge $r \subseteq N$ der Return-Knoten beinhaltet alle Knoten, die mittels einer Rücksprunganweisung aus der aktuell analysierten Methode fliehen.

Wird die Definition von *fliehen* auf diesen Points-To-Escape-Graphen abstrahiert, so erhält man folgende Aussagen (Sei dazu $n \in N$ ein beliebiger Knoten):

1. n flieht direkt $(directlyescapes(n)) \Leftrightarrow n \in N_R \cup N_P \cup N_T \cup N_{CL} \vee e(n) \neq \emptyset$.
2. n flieht $(escaped(n)) \Leftrightarrow \exists n_0, \ldots, n_k \in N : d = n_0 \rightarrow n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_{k-1} \rightarrow n_k = n \wedge directlyescapes(d)$.
3. n ist gefangen $(captured(n)) \Leftrightarrow \neg escaped(n)$.

A.1 Intraprozedurale Analyse

In diesem Abschnitt werden die Veränderungen des Points-To-Escape-Graphen während der intraprozeduralen Analyse beschrieben. Das Verfahren nutzt eine Datenflussanalyse, um den Points-To-Escape-Graphen zu jedem Punkt in der aktuell analysierten Methode zu berechnen. Dabei wird der initiale Points-To-Escape-Graph $G_0 = (I_0, O_0, e_0, r_0)$ wie folgt definiert:

- Die Menge O_0 enthält alle äußeren Kanten, welche von den statischen Klassenvariablen zu den entsprechenden globalen Knoten zeigen.
- Die Menge I_0 enthält alle inneren Kanten, welche von den Parametern zu den entsprechenden Parameterknoten zeigen.
- Die Escape-Funktion e_0 liefert zunächst für alle Knoten $n \in N : e(n) = \emptyset$.
- Die Menge der Return-Knoten r_0 ist zum Startzeitpunkt der Analyse leer: $r_0 = \emptyset$.

Der Algorithmus setzt voraus, dass alle statischen Klassenvariablen und Parameter zu unterschiedlichen Objekten verweisen. Diese Einschränkung wird später mit Hilfe eines Mapping-Algorithmus [28] aufgelöst, indem alle globale Knoten (bzw. Parameter-Knoten), die dasselbe Objekt repräsentieren, zu einem globalen Knoten (Parameter-Knoten) verschmolzen werden.

Der Algorithmus benutzt eine Transferfunktion der Form $(I', O', e', r') = [st]\,(I, O, e, r)$, um die Auswirkungen einer Anweisung st auf den aktuellen Points-To-Escape-Graphen (I, O, e, r) zu beschreiben:

1. Hat st die Form $l = v$, so zeigt l nach der Ausführung von st auf alle Objekte, auf die v zeigt. Um diesen Effekt darzustellen, werden alle Kanten gelöscht, die von l ausgehen und zusätzliche innere Kanten hinzugefügt, welche von l ausgehen und auf alle Knoten zeigen, auf die v zeigt.

2. Ist st eine Ladeanweisung in der Form $l_1 = l_2.f$, so zeigt nach der Ausführung von st l_1 auf alle Knoten, auf die $l_2.f$ zeigt. Zunächst werden alle Kanten gelöscht, die von l_1 ausgehen. Dann werden zusätzliche innere Kanten hinzugefügt, welche von l_1 ausgehen und auf alle Knoten zeigen, auf die $l_2.f$ zeigt. Dabei wird zwischen folgenden zwei Fällen unterschieden. Zeigt l_2 mindestens auf einen fliehenden Knoten, so wird ein Lade-Knoten eingefügt, auf den l_1 mit einer neu erzeugten inneren Kante zeigt. Weiterhin werden für alle fliehenden Knoten äußere Kanten erzeugt, welche mit dem Feld f gekennzeichnet sind und ebenfalls auf diesen Ladeknoten zeigen. Zeigt l_2 auf keinen fliehenden Knoten, sind keine Änderungen durchzuführen. Die Fallunterscheidung ist notwendig, da andere Programmteile außerhalb der analysierenden Methode oder ein Thread, welcher parallel abläuft, auf das entsprechende durch den fliehenden Knoten repräsentierte Objekt zugreifen und in seinen Feldern abspeichern könnten. Somit könnte $l_2.f$ unter Umständen eine Referenz beinhalten, die außerhalb des aktuellen Analyse-Bereiches erstellt wurde. Diese Referenz wird mittels der äußeren Kante modelliert.

3. Hat st die Form $l_1.f = l_2$, so zeigt l_1 nach der Ausführung von st auf alle Objekte, auf die l_2 zeigt. Dazu werden neue innere Kanten erzeugt und hinzugefügt, welche von l_1 ausgehen und auf alle Knoten zeigen, auf die l_2 zeigt und die mit dem Feld f bezeichnet sind.

4. Handelt es sich bei st um eine globale Ladeanweisung der Form $l = cl.f$, so werden zunächst alle von l ausgehenden Kanten gelöscht. Anschließend werden neue innere Kanten hinzugefügt, welche von l ausgehen und auf alle inneren und äußeren Knoten zeigen, die von cl über das Feld f erreichbar sind.

5. Ist st eine globale Speicheranweisung der Form $cd.f$, so werden neue innere Kanten von cl zu allen Knoten, auf die l zeigt, mit der Bezeichnung f hinzugefügt.

6. Hat st die Form $l = newObject\,(p_1, \ldots, p_n)$, so wird ein neues Objekt instanziiert und l eine Referenz darauf zugewiesen. Alle Kanten, die von l ausgehen, werden nun gelöscht und eine neue innere Kante hinzugefügt, welche von l ausgeht und auf einen neu erzeugten inneren Knoten n zeigt, der das neu instanziierte Objekt repräsentiert.

[28] Siehe Abschnitt A.3.

7. Ist *st* eine Return-Anweisung der Form *returnl*, so spezifiziert *st* den Rückgabewert und ist der unmittelbare Vorgänger der *exit*-Anweisung der Methode. Es werden entsprechend alle Knoten, auf die l zeigt, der Menge der Return-Knoten hinzugefügt.

8. Die Übergangsfunktion produziert für eine *exit*-Anweisung den endgültigen Points-To-Escape-Graphen der zu analysierenden Methode und damit das intraprozedurale Analyseergebnis, welches nun für jeden Aufruf der Methode verwendet werden kann. Abschließend werden noch alle Informationen aus dem Graphen entfernt, die für die aufrufenden Methoden nicht sichtbar sein sollen. Alle Knoten, die weder von den globalen Knoten, den Parameter-Knoten oder der Menge der Return-Knoten erreichbar sind, werden gelöscht.

A.2 Interprozedurale Analyse

In der interprozeduralen Analyse werden Wechselwirkungen von Methoden untersucht und mit in das Ergebnis einbezogen. Hierfür werden Points-To-Escape-Graphen verschiedener analysierter Methoden durch eine *meet*-Operation zusammengeführt.

Definition: Sei J ein Join-Knoten mit Vorgängerknoten P_i, $1 \leq i \leq k, k \geq 2, k \in N$. Seien weiterhin $(I_1, O_1, e_1, r_1), \ldots, (I_k, O_k, e_k, r_k)$ die Points-To-Escape-Graphen nach der Analyse der Vorgängerknoten $P_1, \ldots, P_k$. Der Points-To-Escape-Graph (I_J, O_J, e_J, r_J) nach der Bearbeitung des Join-Knoten J ist definiert als:

$$(I_J, O_J, e_J, r_J) = (I_1, O_1, e_1, r_1) \sqcup \ldots \sqcup (I_k, O_k, e_k, r_k) \tag{1}$$

$$= \left(\bigcup_j I_j, \bigcup_j O_j, \bigcup_j e_j, \bigcup_j r_j \right); 1 \leq j \leq k \tag{2}$$

Dabei ist $e_j \cup e_m = e_l(n) \cup e_m(n) \, \forall n \in N = N_l \cup N_m$, mit N_l-Knotenmenge des Graphen (I_l, O_l, e_l, r_l) und N_m-Knotenmenge des Graphen (I_m, O_m, e_m, r_m).

Mit der Meet-Operation wird eine partielle Ordnung $\leq$ auf der Menge der Points-To-Escape-Graphen definiert. Seien (I_1, O_1, e_1, r_1) und (I_2, O_2, e_2, r_2) zwei Points-To-Escape-Graphen. Dann gilt:

$$(I_1, O_1, e_1, r_1) \leq (I_2, O_2, e_2, r_2) \Leftrightarrow I_1 \subseteq I_2 \wedge O_1 \subseteq O_2 \wedge r_1 \subseteq r_2 \\ \wedge e_1(n) \subseteq e_2(n) \, \forall n \in N = N_1 \cup N_2$$

Im Sinne einer Datenflussanalyse erhält die Relation $\leq$ ein $Null$-Element der Form

$$(I_\perp, O_\perp, e_\perp, r_\perp) = (\emptyset, \emptyset, e(n) = \emptyset, \emptyset) \, \forall n \in N.$$

Für eine Anweisung *st* produziert die Analyse Ergebnisse $\alpha(\bullet st)$ und $\alpha(st\bullet)$ nach der Anweisung *st*. Diese erfüllen die folgenden Gleichungen:

$$\alpha(\bullet enter_{op}) = (I_0, O_0, e_0, r_0) \tag{3}$$

$$\alpha(\bullet st) = \sqcup \{\alpha(st'\bullet) \mid st' \in pred(st)\} \tag{4}$$

$$\alpha(st\bullet) = [st]\,\alpha(\bullet st) \tag{5}$$

Dabei bezeichnet $pred(st)$ die Menge aller Anweisungen, die unmittelbar vor st ausgeführt werden sollen. Die endgültigen Analyseergebnisse einer Methode op werden mit $\alpha(exit_{op})$ bezeichnet.

Wann immer die Analyse einen Methodenaufruf findet, unterscheidet sie zwischen den folgenden Möglichkeiten: (Sei dazu $l = l_0.op(p_1, \ldots, p_k)$ ein solcher Methodenaufruf m.)

1. Die aufgerufene Methode wird analysiert und der Mapping-Algorithmus wird angewendet, um die Analyseergebnisse in den Points-To-Escape-Graphen der aufrufenden Methode abzubilden. Der daraus resultierende Points-To-Escape-Graph (I', O', e', r') ist definiert als

$$(I', O', e', r') = [m](I, O, e, r) \tag{6}$$
$$= \sqcup\{map(m, (I, O, e, r), op) \mid op \in callees(m)\}. \tag{7}$$

   Dabei bezeichnet (I, O, e, r) den Points-To-Escape-Graphen vor dem Methodenaufruf, map den Mapping-Algorithmus und $callees : M \to \Re(M)$, *mit M ist Menge aller Methoden*, eine Funktion, die für ein gegebenes Argument alle aufgerufenen Methoden bestimmt.

2. Die Analyse der aufgerufenen Methode wird übersprungen. Dann trifft die Analyse die Annahme, dass alle Parameter, die dem Methodenaufruf übergeben werden, in diesen Aufruf fliehen. Die Übergangsfunktion löscht nun alle inneren Kanten, die von l ausgehen und fügt eine zusätzliche innere Kante ein, die von l ausgeht und auf n_R zeigt (mit n_R ist der Return-Knoten, welcher den Rückgabewert der nicht analysierten Methode repräsentiert). Anschließend wird die Escape-Funktion des aktuellen Graphen wie folgt angepasst:

$$e'(n) = \{e\}$$

   Dabei liefert die Funktion $I : N \to \Re(N)$ für ein gegebenes Argument alle Knoten, zu denen dieses eine innere Kante besitzt.

Beim Abbilden der Analyseresultate einer aufgerufenen Methode op sind folgende Points-To-Escape-Graphen von Interesse:

1. Der ursprüngliche Points-To-Escape-Graph (I, O, e, r) vor dem Methodenaufruf.
2. Der neue Points-To-Escape-Graph $(I_m, O_m, e_m, r_m) = \alpha(exit_{op}\bullet)$ nach Abschluss der Analyse von Methode op, der die endgültigen Analyseresultate dieser Methode repräsentiert.
3. Der resultierende Points-To-Escape-Graph $(I', O', e', r') = map(m, (I, O, e, r), op)$ nach dem Aufruf der Methode op.

A.3 Der Mapping Algorithmus

Der Algorithmus erstellt zuerst eine Initialabbildung $\mu : N \to \Re(N)$ von den Knoten des eingehenden Points-To-Escape-Graphen zu den Knoten des ursprünglichen Graphen. Für jeden äußeren Knoten n des eingehenden Graphen bezeichnet $\mu(n)$ die Menge aller Knoten,

die n während der Analyse der aufgerufenen Methode repräsentiert. Der Startwert von μ bildet dabei globale Knoten, Parameterknoten und Ladeknoten auf die entsprechenden Knoten im neuen Graphen ab. Um Ladeknoten korrekt abzubilden, werden die zugehörigen Pfade des eingehenden Graphen mit dem ursprünglichen Graphen verglichen. Jeder Pfad des alten Graphen besteht aus einer Sequenz von inneren Kanten und der entsprechende Pfad des eingehenden Graphen aus einer Sequenz von äußeren Kanten, die die zugehörigen inneren Kanten während der Analyse der Methode repräsentieren.

Anschließend erzeugt der Algorithmus den neuen Points-To-Escape-Graphen $(I', O', e', r') = map\,(m, (I, O, e, r)\,, op)$. Zuerst wird der neue Graph mit dem alten Graphen (I, O, e, r) initialisiert und danach die Abbildung μ genutzt, um Knoten und Kanten des eingehenden Graphen auf den alten Graphen abzubilden. Die abgebildeten Knoten repräsentieren dabei Objekte, die entweder in der aufgerufenen Methode erzeugt wurden oder auf die von der aufgerufenen Methode zugegriffen wurde. Abgebildete Kanten repräsentieren Referenzen, die entweder von der aufgerufenen Methode erstellt wurden oder auf die die aufgerufene Methode zugegriffen hat.

Als Teil des Abbildungsprozesses wird μ derart erweitert, dass gilt: Wenn ein Knoten n des eingehenden Graphen auf den neuen Graphen abgebildet wird, dann ist $n \in \mu\,(n)$. In diesem Fall ist n in dem neuen Graphen *vorhanden*. Die Abbildung ist dabei von der Art des Knotens abhängig, welcher abgebildet werden soll. Sei n ein abzubildender Knoten.

- $n \in N_P$ – n repräsentiert Knoten, auf die aktuelle Parameter zeigen und es werden die Kanten von n auf die entsprechenden Kanten dieser Knoten abgebildet, wobei n selbst nicht in diesem Graphen vorhanden ist. $\mu\,(n)$ ist dann die Menge jener Knoten im aktuellen Graphen, auf die die aktuellen Parameter zeigen.

- $n \in N_G$ – n repräsentiert Objekte, auf die eine entsprechende statische Klassenvariable zeigt und es werden alle Kanten von n auf die zugehörigen Kanten im alten Graphen abgebildet, wobei n im neuen Graphen vorhanden ist. $\mu\,(n)$ ist dann die Menge der Knoten, auf die die entsprechende Klassenvariable zeigt und es gilt $n \in \mu\,(n)$. (n flieht im neuen Graphen)

- $n \in N_L$ – alle Kanten der Knoten, welche n repräsentiert, werden in dem neuen Graphen abgebildet. Repräsentiert n ein Objekt, das außerhalb des aktuellen Analysebereiches der aufrufenden Methode erstellt wurde oder auf das mittels Referenzen zugegriffen wird, die außerhalb dieses Analysebereiches erstellt wurden, wird n zusammen mit seinen Kanten in den neuen Graphen abgebildet. Die zugrundeliegende Idee ist, dass n nur dann im neuen Graphen vorhanden sein sollte, wenn mindestens ein fliehender Knoten auf n zeigt. Eine Untersuchung jener Knoten des eingehenden Graphen, die auf n zeigen, liefert folgende zwei Fälle:

  1. Wenn mindestens einer dieser Knoten – n' – im neuen Graphen vorhanden ist und flieht, dann sollte n ebenfalls vorhanden sein und es wird eine äußere Kante von n' nach n im neuen Graphen existieren.
  2. Wenn mindestens einer dieser Knoten – n'' – auf einen fliehenden Knoten des alten Graphen abgebildet wird, sollte n im neuen Graphen vorhanden sein und es wird eine äußere Kante von n'' nach n im neuen Graphen existieren.

In beiden Fällen flieht n. $\mu(n)$ beinhaltet die Menge der Knoten des alten Graphen, die n während der Analyse der Methode repräsentierte, und es gilt: $n \in \mu(n)$, wenn n im neuen Graphen vorhanden ist.

- $n \in N_I$ – innere Knoten mit ihren Kanten sind nur dann im neuen Graphen vorhanden, wenn die Objekte, die sie repräsentieren, in der aufrufenden Methode erreichbar sind. Eine Überprüfung der Knoten des eingehenden Graphen, die auf n zeigen, entscheidet über das Vorhandensein von n im neuen Graphen.

  1. Wenn mindestens einer dieser Knoten – n' – im neuen Graphen vorhanden ist, dann ist n im neuen Graphen vorhanden und es existiert eine Kante von n' nach n.
  2. Wenn mindestens einer dieser Knoten – n'' – einen Knoten des alten Graphen repräsentiert, dann ist n im neuen Graphen enthalten und es existiert eine Kante von n'' nach n.

  Im Unterschied zu Ladeknoten, welche nur dann im neuen Graphen vorhanden sind, wenn sie darin fliehen, sind innere Knoten enthalten, wenn sie in diesem Graphen erreichbar sind, gleich ob sie fliehen. Es gilt: $\mu(n) = \{n\}$, falls n im neuen Graphen vorhanden ist, $\mu(n) = \emptyset$ sonst.

- $n \in N_R$ – für Return-Knoten gelten die Bedingungen der inneren Knoten.

Selbständigkeitserklärung:
Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Hilfsmittel und Literatur angefertigt habe.

Jena, den 21. April 2006

.....

Lebenslauf

Name:	Andreas Hartmann
Geburtsdatum:	22.11.1974
Geburtsort:	Erfurt
Anschrift:	Schrödinger Strasse 59 07745 Jena
Schulbildung:	1981 - 1989 POS 56, Erfurt 1989 - 1993 Integrierte Gesamtschule Johannesplatz, Erfurt 1993 Abschluss Abitur
Wehrersatzdienst:	Juli 1993 - Februar 1994
Studium:	1994 - 1995 Rechtswissenschaften 1995 - 2001 Informatik Diplom, Nebenfach Betriebswirtschaft 2001 Abschluss Diplom
Wissenschaftliche Tätigkeit:	Januar 2002 - März 2005 FSU Jena (wissenschaftlicher Mitarbeiter im Projekt SafeTSA) seit April 2005 FSU Jena (wissenschaftlicher Mitarbeiter zu je 50% für Koordination HISPOS und Konzeption Infrastruktur der FSU in Unterstützung von Dr. Hinz und Prof. Rossak, weiterhin Unterstützung der Einführung thoska an der FSU) 2002 FH Jena (Lehrauftrag) 2003 - 2006 FH Erfurt (Lehrauftrag)

Jena, den 21. April 2006