Ausgewählte Kapitel aus dem Übersetzerbau
Zusammenfassung des Wichtigsten

Wintersemester 2016/17

Prof. Dr. Philippsen et. al.

Autor:

- Christian Strate

Inhaltsverzeichnis

0 Organisatorisches **3**

1 Linker und Lader **4**

2 Exceptions **9**

3 Aspektorientierte Sprachen **14**

4 Garbage Collection **17**
4.1 mit Verwendungszähler 18
4.2 mark and sweep - Markieren und Auskehren 21
4.3 Kopierend 23
4.4 Generationskonzept 24
4.5 Inkrementelle und nebenläufige Speicherbereinigung 26

5 Statische Analyse und Symbolische Ausführung **29**

6 Virtuelle Maschinen, Just-in-Time-Compiler I **31**

7 Virtuelle Maschinen, Just-in-Time-Compiler II **37**

8 Wettlaufsituationen **44**

9 Software Watermarking **49**

10 Funktionale Sprachen 1 **51**

11 Funktionale Sprachen 2 **56**

12 LLVM **61**

13 Abbildungsverzeichnis **65**

14 Listingverzeichnis **66**

0 Organisatorisches

- Übung in den Semesterferien
- Anmeldung via EST

Note:

Bei dieser Zusammenfassung handelt es sich um eine inoffizielle Mitschrift von Studenten. Entsprechend sind weder Garantie auf Vollständigkeit noch auf Korrektheit gewährleistet.

1 Linker und Lader

Linker:

Programmmodule werden zu einem ausführbaren Programm zusammengefügt.

Lader:

Des ausführbaren Programm wird in den Arbeitsspeicher geladen und auf die Ausführung vorbereitet.

Gemeinsame Aufgaben von Linker und Lader:

Beide sind für Relokation von Code- und Datenspeicherlayout und für die Auflösung von Symbolen zuständig. Entsprechend können Linker und Lader jeweils vereinfacht werden, indem das entsprechende Gegenstück mehr Arbeit erledigt.

```
1  .file   "main.c"
2          .data
3          .type string.0 @object
4          .size string.0, 13
5  string.0:
6          .string "Hello, world"
7          .text
8
9  .global main
10         .type  main, @function
11 main:
12         pushl %ebp
13         movl  %esp, %ebp
14         subl  $8, %esp
15         andl  $-16, %esp
16         movl  $string.0, (%esp)
17         call  print
18         movl  %ebp, %esp
19         popl  %ebp
20         ret
21 .size   main, .-main
```

Listing 1: 32bit Assembler main-print - ruft print mit “Hello, world” auf. - wird in fig. 1 gebunden.

```
1  .file   "print.c"
2          .text
3
4  .global print
5          .type  print, @function
6  print:
7          pushl %ebp
8          movl  %esp, %ebp
9          pushl %ebx
10         subl  $16, %esp
11         movl  8(%ebp), %ebx
12         pushl %ebx
13         call  strlen
14         addl  $12, %esp
15         pushl %eax
16         pushl %ebx
17         pushl $1
18         call  write
19         movl  -4(%ebp), %ebx
20         leave
21         ret
22 .size   print, .-print
```

Listing 2: 32bit Assembler print - ruft write auf mit Parameter - wird in fig. 1 gebunden.

Binden:

Beim Binden werden Objektdateien eingelesen und das Speicherlayout wird erstellt, wobei etwaig notwendiger Speicher alloziert wird und Symbole mit den entsprechenden Adressen aufgelöst werden. Eventuell werden noch Symbole für verwendete Bibliotheken, "Glue-Code"[1] um dynamische Bibliotheken zu laden, der Startcode und Debug-Symbole hinzugefügt.

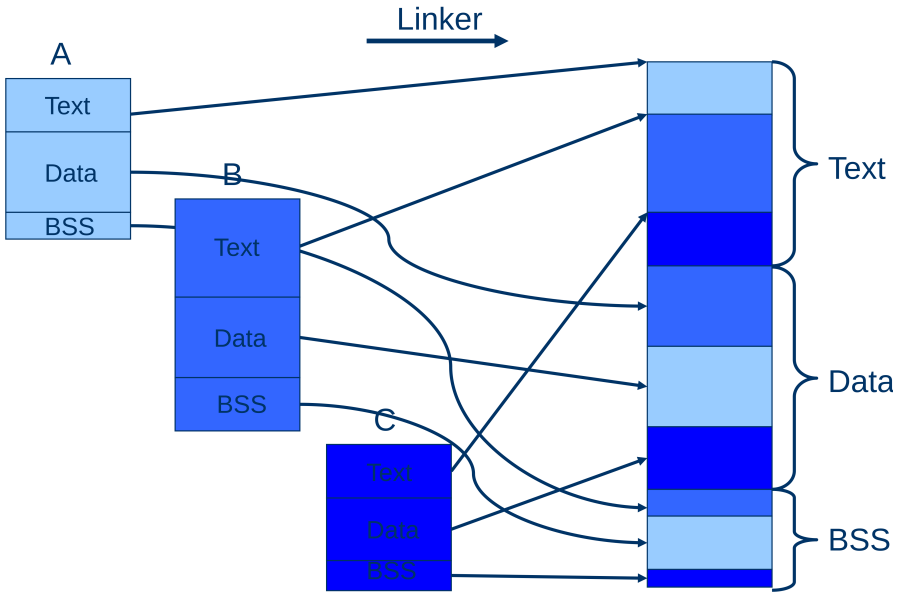

Abbildung 1: Reales Speicherlayout print - Binden des Assemblercodes von listings 1 and 2

Relokation:

Adressen werden angepasst und eingesetzt, nachdem das Speicher-Layout erstellt wurde, da auch relative Adressen aufzulösen sind. Die Informationen der relokierten Symbole werden in "Relocation Records' gespeichert und dienen zur späteren Auflösung. Dort enthalten sind Offset, Type, Value. Beispiel für listing 1 - die zwei aufeinander folgenden Instruktionen bei dem print-Aufruf *movl $string.0 (%esp); call print* Der erste Eintrag ist absolut, der zweite relativ zum Programmcounter. Das zweite ist also additiv zur Adresse des Befehls der Verwendung:

RELOCATION RECORDS FOR [.text]:

OFFSET	TYPE	VALUE
0x00000c	R_386_32	.data
0x000011	R_386_PC32	print

ELF als Erweiterung für a.out:

Mit dem alten Format a.out waren dynamisch gebundene Bibliotheken nur schwer möglich und C++ wird nicht sinnvoll unterstützt. Dafür liefert ELF eine Lösung.

[1] Der Code, der den Bibliothekscode mit dem eigentlichen Programmcode verklebt.

Zwei verschiedene Sichten, einmal für Linker und einmal für Loader

- Program Header Table - für Loader relevant
  benötigen große Teile (Segmente)
- Section Header Table - Relevant zur Compilezeit (Linker)
  Benötigen kleine Teile (Sections)

statische Shared Libraries:

- schnell, da feste Adressen
- kleiner als mit dyn. Bibliotheken
- unflexibel und statisch, Anpassung erfordert neues Binden.
- Sprungtabelle für Indirektionen um Aktualisierungen zu vereinfachen
  - bessere Wartbarkeit, aber immernoch schwierig
  - geringe Laufzeitkosten durch Indirektion
- Die Sprungtabelle, dessen Adressen sich nicht ändern, kann gecached werden.
- Bibliothekerzeugung ist komplex
  - Speicher-Layout festlegen
  - Sprungtabellen
  - Stub-Bibliotheken (Umsetzung zwischen Symbolen in Eingabebibliotheken und Sprungtabelle vor)erzeugen.
- Binden ist einfach
  - Layout der Bibliotheken ist bekannt
  - Relokation innerhalb der Bibliotheken fertig
  - Stub-Bibliotheken enthalten alle relevanten Informationen

dynamische Shared Libraries:

- Bindezeit: Linker durchsucht Bibliotheken nach unaufgelösten Symbolen und ordnet die Symbole den Bibliotheken zu.
- Ladezeit: benötigte Bibliotheken werden im Speicher abgebildet und zur Laufzeit erfolgt die Auflösung
- Code muss position independent sein. Beispiel 1-48
- Vor- Nachteile
  - einfacher erzeugbar

- Wesentlich bessere Wartbarkeit
- Laden und Entladen von Routinen zur Laufzeit
- Laufzeitkosten höher als bei statischen, da das Binden bei jedem Programmaufruf wiederholt werden muss
- Dynamisch gebundene Symbole sind in der Symboltabelle nachzuschlagen und aufzulösen
- Indirekter Zugriff auf Bibliotheksvariablen über GOT[2] Jede shared lib. besitzt ihre eigene GOT, die vom dynamischen Linker bei Programmstart relokiert wird. Lazy Binding sorgt dafür, dass die GOT mit der korrekten Adresse gepatcht wird, sobald die Bibliothek die Adresse sucht (indirekter Zugriff). s.a. PLT and GOT - the key to code sharing and dynamic libraries
- Indirekte Funktionsaufrufe über procedure linkage table (PLT)
  * Jedes dynamisch gebundene Programm/Bib besitzt eine solche Tabelle mit aufgerufenen externen Funktionen
  * lazy-evaluation sorgt für schnelleren Programmstart. Beim ersten Aufruf einer solchen Funktion wird der dyn. Linker aufgerufen, der die Adresse findet und sich selbst als Zieladresse ersetzt.

[2] GOT auf x86 finden: call ⇒ Programmcounter von dem pop auf den stack, diese wird nach ebx geschoben, dadrauf dann die GOT-Basisadresse draufschmeisen.

2 Exceptions

jmp_buf, setjmp:

Ist ein Sprungpuffer, der die zur Wiederherstellung des Zustands des Aufrufers notwendigen Informationen enthält.

- Stackpointer
- Framepointer
- Programmcounter
- Callee-Save-Register
- Rückgabewert ist per default 0

Dynamisches Registrieren - stack cutting:

Ein catch-Block wird beim Betreten eines try-Blocks registriert und der Stack wird bis auf Ebene des passenden catch-Blocks abgeschnitten. Dies ist notwendig, da bei einer Exception der Fehler bis zum Catch die einzelnen Stackframes "hochklettern" muss. Die Zustände von jmp_buf lassen sich mittels *setjmp* speichern. Ein *longjmp* nutzt die Informationen des Sprungpuffers. listing 3

longjmp:

- Stack- und Framepointer zurücksetzen
- setjmp-Rückgabewert auf ungleich 0 setzen
- Sprung an Programmcounter

```
1  jmp_buf handler;
2  int div(int a, int b){
3     if(!b)
4        longjmp(handler, 1);
5     return a/b;
6  }
7  int main(void){
8     if(!setjmp(handler)
9        std::cout<< div(5,0) << std::endl;
10    else
11       std::cerr<< "division by zero" << std::endl;
12 }
```

Listing 3: Beispiel stack cutting - setjmp setzt den Rückgabewert auf 0, merkt sich die wichtigen Register, wie Stackpointer, Basepointer etc. Dann wird die Funktion *div* betreten, dort tritt der Ausnahmefall auf und es wird entsprechend ein longjmp durchgeführt, der zur Adresse des vorherigen setjmps springt und dort den setjmp-Rückgabewert auf 1 setzt, weshalb dieses mal nicht die Funktion *div* aufgerufen wird, sondern die Fehlerbehandlung.

Probleme:

Probleme dabei sind, dass nur ein Behandlungsort unterstützt und auch nur ein Typ von einer Ausnahmen behandelt werden kann. Man kann zwar in der Ausnahmebehandlung selbst auch wieder longjmps ausführen, und weiter oben den zugehörigen setjmp ausführen, aber das kann auch recht lästig und unübersichtlich werden. Und dafür benötigte man mehr als eine *jmp_buf*-Instanz. Da man aber nicht weiß wie viele maximal auftreten können ist das auch eher unschön, daher lieber das folgende tun.

Lösungen:

- Beim Betreten eines try-Blocks auf dem Stack eine neue jmp_buf-Instanz anlegen und diese im Fehlerfall anspringen
- Ausnahmen besitzen eine Typbezeichnung und suche im Stack nach einer Behandlung dieses geworfenen Typen

Vor-Nachteile Dynamisches Registrieren - stack cutting:

- Vorteile
  - Einfaches Verfahren
  - Transformationen im Compiler einfach
  - günstige Ausnahmebehandlung
- Nachteile
  - langsam
  - Allokation und Verkettung der jmp_buf-Instanzen auf dem Prozedurstack nicht trivial
  - Komplexes Laufzeitsystem zur Verwaltung der Ausnahmebehandlungen
  - So wie ich das sehe werden allocs nicht beachtet und auch keine Destruktoren aufgerufen, weshalb die Referenzcounter von bspw. Shared Pointern nicht mehr aktuell sind.

Die dynamisch registrierten Informationen sind statisch im Compiler zur Compile-Zeit

bestimmbar.

Statische Tabellen - stack unwinding:

Es werden drei Label erzeugt L1: try{, L2: }, L3: catch{ Für jeden try-Except-Block wird ein Tabellen-Eintrag angelegt mit der Form *Intervall, exceptiontype, handler*, wobei das Intervall der Code zwischen L1 und L2 ist, der type der Ausnahme-Typ und der handler die Adresse von L3

Vor-Nachteile Statische Tabellen:

- Vorteile
  - Transformationsschablone im Compiler einfach
  - Keine Kosten im ausnahmfreien Fall
- Nachteile
  - sehr teuer im Ausnahmefall (Abbau des Stacks)
  - Intervallsuche in der Ausnahme-Tabelle
  - Ausnahme-Tabellen können groß werden

Automatische Variablen:

Sind Variablen, die auf dem Stack abgelegt werden. Nach dem Auftreten einer Exception müssen sämtliche Variablen des Stack-Frames aufgeräumt werden, also insbesondere auch Destruktoren aufgerufen werden. Entweder man merkt sich diese entsprechend in verketteten Listen und durchläuft diese Rückwärts oder man fängt die Ausnahme bei jedem Stackframe ab, ruft die Destruktoren auf und wirft die Ausnahme weiter. Die Auflösung des Stacks erfolgt also Stackframe-weise (bzw. streng genommen Scopeweise, da die Lebenszeit in C(++) ja Scope-basiert ist).

Locks:

z.B. von synchronized Blöcken sind in Ausnahmefällen freizugeben.

Probleme durch Ausnahmen:

Ausnahmen können den Kontrollfluss verändern wodurch die Kontrollflusskanten-Anzahl stark ansteigt und die Grundblock-Größe verkleinert wird. Das ist für Optimierungen kontraproduktiv, da hier große Grundblöcke wünschenswert sind.

Die Idee ist es nun einen faktorisierten KFG (FKFG) zu erzeugen (fig. 2 and algorithm 1), der die Ausnahme-Kanten zusammenfasst. Dadurch wird anstelle eines Single-Entry-Single-Exit-Prinzips, in dem alle vorhergehenden Grundblock-Instruktionen postdominiert werden, ein Single-Entry-Multiple-Exit-Prinzip eingeführt, bei dem der Grundblock vorzeitig verlassen werden kann und nicht notwendigerweise alle vorhergehenden Instruktionen eines Grundblocks postdominiert werden. Nachteil am FKFG ist, dass viele Algorithmen, wie z.B. der DFA-Fixpunkt-Algorithmus aus CB2 nicht funktionieren.

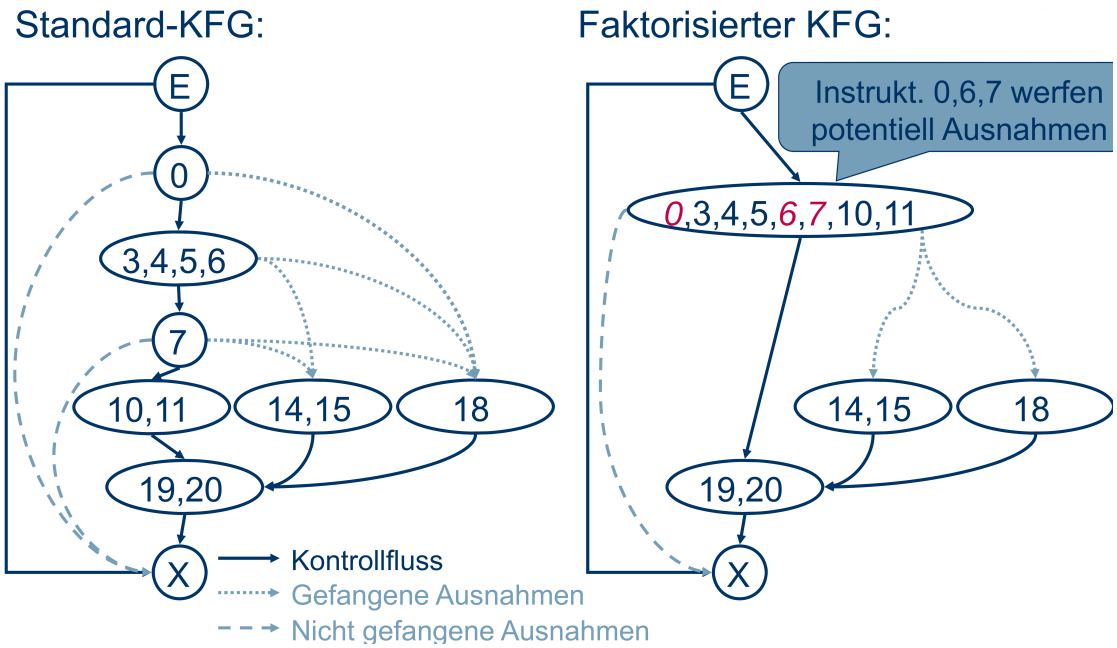

Abbildung 2: FKFG - potentielle Ausnahmen der Instruktionen
0: RuntimeException, Error
6: NullPointerException, ArrayIndexOutOfBoundsException
7: Exception, Error

Ausnahmetreue:

Ausnahmen müssen auch im optimierten Code an den gleichen Stellen auftreten (relativ zum Rest) - das schränkt beliebige Vertauschung von Instruktionen ein, da weitere Abhängigkeiten entstehen.

Algorithm 1 FKFG-Erzeugung

▷erzeugt FKFG für Funktion f
Konstruktion KFG für f (s. CB2)
Einträge der Ausnahme-Tabelle für f nach Intervall und Behandlungsadressen sortieren
for $i :=$ alle Instruktionen inf, die eine Ausnahme werfen können (PEI) **do**
  B ist Grundblock von i
  if $i \in$ Behandlungsintervall I **and** die Ausnahmetypen verträglich sind **then**
    Füge Kante von B zum ersten Grundblock der entsprechenden Behandlung ein
  else
    Füge Kante von B zum $Exit - Knoten$
  end if
end for

Lösung Ausnahmetreue:

- aggressive Optimierung ausschalten, sofern Ausnahmen möglich
- Verzicht auf Ausnahme-Treue durch Semantik-Änderung
- Versionierung - es werden zwei Versionen gebaut - jeweils eine in der (k)eine Ausnahmen auftreten dürfen ⇒ Code-Explosion
  - optimierte Variante - hier können/dürfen keine Ausnahmen auftreten
  - kaum optimierte Variante - Ausnahmen sind möglich

  Da Ausnahmen meist selten auftreten, kann das Programm so angeordnet werden, dass der normale Fluss den Instruktionscache möglichst effizient nutzt und die Ausnahmebehandlungen (catch) an andere Stelle verschoben werden und angesprungen werden müssen.

3 Aspektorientierte Sprachen

Kernbelang:
: Eigentliche Applikations-Funktionalität

Systembelang:
: orthogonal zu Kernbelang - zur Implementierung der Applikation notwendige Funktionalität, z.B. Protokolle, Caching, Thread-Sicherheit, Authentifizierung

Für tiefere Einblicke in AOP empfehle ich meine KSS-Zusammenfassung, bzw. die originalen KSS Folien. Hier wird im Folgenden nur sehr grob auf die Konzepte eingegangen und die i2-Vorlesungsslides sind wie immer sehr Java-spezifisch.

Join Points:
: Ein Programmpunkt im Kontrollfluss, den der Übersetzer identifizieren kann. z.B. Methoden{aufruf, ausführung, rückkehr} {Variablen, Attribut}zugriff, Objekterzeugung, Initialisierungen, Ausnahmebehandlungen ....

Pointcuts:
: Regeln, die festlegen an welchen Join Points Code eingewoben werden sollen. Beispiel in Java:

$$\underbrace{\texttt{public}}_{\text{Sichtbarkeit}}\ \texttt{pointcut}\ \underbrace{\texttt{accountOperations}()}_{\text{Pointcut-Name}}:\ \underbrace{\texttt{call}}_{\text{Join Points-Typ}}\ \underbrace{(\texttt{* Account.*(..)})}_{\substack{\text{Signatur: *: Platzhalter für Ident.;}\\ \text{..: beliebige Argumente}}}$$

Advice:
: Welcher Programmcode bei einem Pointcut eingewoben werden soll. Hierbei unterscheidet man *before advice, after advice, around advice*, die den Code gemäß ihrer Bezeichnung selbsterklärend platzieren.

Aspekt:
: Ist eine Menge von Pointcuts und Advices.

dynamische Parameter im Advice: 3-25

```
1 public aspect AsyncAspect{
2    pointcut callMethod() : call(* Database.*(..));
3    void araound() : callMethod(){
4       //Das hier drinnen ist betroffen
5       Thread t = new Thread(new Runnable(){
6          public void run() { proceed(); }
7       });
```

```
8         t.start()
9     }
10 }
```

Listing 4: Threads Ausführung kann stattfinden, nachdem man schon lange aus der Funktion rausgelaufen ist, der Aspekt-Kontext also nicht mehr vorliegt. Lösung ist ein closure - das ist aber viel zu langsam weil für jedes Thread-Objekt angelegt werden muss. Besser ist es für jede Klasse und Funktion, für die Advices getriggert werden listing 5 zu machen - also werte direkt reinkopieren

```
1  class AspectClass
2     void advMeth$1(Around$1 closure, int jpID, int clID, args){
3        if(clID == 0)
4           closure.proceed(jpID, args);
5        else if(clID == 1)
6           Class1.proceed$1(jpID, args);
7        else if(clID == 2)
8           Class2.proceed$1(jpID, args);
9        ...
10    }
11 }
12
13 class Class1{
14    static void proceed$1(int jpID, args){
15       if(jpID == 0)
16          ;//do what 1st JP did
17       else if(jpID == 1)
18          ;//do what 2nd JP did
19       ...
20    }
21 }
```

Listing 5: Lösung für listing 4 - Werte werden direkt in die Objekte hineinkopiert, anstatt sich auf die Umgebung des Aspektes zu verlassen.

cflow, cflowbelow:

cflow: Pointcut-Bedingung - ist erfüllt, wenn der Kontrollfluss sich in einem durch die Pointcut-Expression beschriebenen Join Point befindet.
cflowbelow: Der Kontrollfluss muss sich unterhalb des Join Points befinden, sonst äquivalent.

Interprozedurale Optimierung, mayCflow, mustCflow, necessaryShadows:

Bei Optimierungen wird versucht eine möglichst große Menge dynamischer Tests (cflow etc.) durch statische zu ersetzen, indem Kontrollflussanalyse betrieben wird. Hierfür ist ein Aufrufgraph für die einzelnen Methoden von Nöten. Man unterscheidet hierfür in *mayCflow(sh)* - Menge von Instruktionen i, des Programms, die sich bei Ausführung in der dynamischen Umgebung von *sh* befinden kann. *mustCflow(st)* - Menge von Instruktionen i des Programms, die sich bei Ausführung sicher in der dynamischen Umgebung eines Schattens *sh* befinden, der den Keller *st* aktualisiert. *necessaryShadows(st)* - Teilmenge der aktualisierenden Schatten vom Keller *st*, deren Effekt an der Position von abfragenden Schatten beobachtet werden kann. Die Berechnungen der Mengen finden sich auf 3-51ff

4 Garbage Collection

Garbage Collection ist in meinen Augen für shared Pointer etc. (vgl C++) sinnvoll, jedoch nicht alle hier vorgestellten Praktiken, da diese Pointer vergleichsweise länger leben und einige Verfahren die Lebendigkeitsdauer der Objekte nicht hinreichend berücksichtigen. Für alles andere sollte man sich meiner Meinung nach selbst Gedanken machen wann die Objekte zu sterben haben anstelle dies von einer teuren GC erledigen zu lassen.

Wird bei der Halden-Freispeichersuche ein Block gesucht und wurde einer gefunden, so ist es sinnvoll den neuen als belegt markierten Block ans Ende des Blocks zu packen, der den freien Speicher repräsentierte. Somit muss lediglich die Größe des weiterhin freien Blocks geändert werden und nicht die Adresse der Zeiger der verkettete Liste manipuliert werden.

Externe Fragmentierung:
: Freie Speicherbereiche zwischen zugeteilten Blöcken

Interne Fragmentierung:
: Freie Speicherbereiche, die Zugeteilt worden, obwohl sie nicht angefordert wurden.

Halden-Verwaltungsstrategien:

- First-Fit 4-12ff
- Buddy 4-19f
- Boundary-Tag 4-21f
- Best-Fit 4-23
- Auffrischung Vor-Nachteile Speicher - Strategien der Speicherzuteilung
- Auffrischung Speicherverwaltung Speicherverwaltung - Zuteilungsverfahren

Es folgen eine triviale Verschmelzungsstrategie beim Freigeben (4-15ff, die in 4-21 noch durch doppelte Metainformationen optimiert werden kann - dieses Boundary-Tag ist etwas schneller als das dereferenzieren der Nachbar-Zeiger) und Buddy-Verfahren (Reduktion der externen Fragmentierung durch Zunahme der internen Fragmentierung, da nicht beliebige Blockggrößen erlaubt werden - 4-19f)

Garbage:
: Strenggenommen ist all das *Garbage*, was für weitere Berechnungen irrelevant ist. Praktisch beschränkt man sich hierbei auf die Blöcke, die von der *Wurzelmenge* nicht (mehr) erreichbar sind.

Wurzelmenge:

- Globale Variablen
- Lokale Variablen im Aufrufstapel
- Registerwerte
- Prozedurparameter

Speicherbereinigungsstrategien - Zeitpunkte:

- Objekt als tot erkannt
- periodisch
- Fehlschlag von Speicherallokation
- Freispeicher unterschreitet prozentualle Grenze

Speicherbereinigungsfamilien:

- mit Verwendungszähler
- mark and sweep - Markieren und Auskehren
- Kopierend
- mit Generationskonzept
- Inkrementell und nebenläufig

4.1 mit Verwendungszähler

Evaluation sofortiges Freigeben bei reference counter == 0:

- Vorteile
  - im allgemeinen kurze Speicherbereinigungspausen
  - gute Lokalität durch sofortige Freigabe und Wiederverwendung
  - gute Lokalitäten für das Freigeben eines Blocks (nur betroffene Blöcke sind zu betrachten)
  - Laufzeit unabhängig vom Speicherverbrauch - Fragmentierung stört dennoch
  - Zählerstände können für Laufzeitstatistiken relevant sein.
- Nachteile

  - Werden von einem Element das es zu löschen gilt Kinder referenziert, die jeweils selbst Kinder referenzieren und all diese Kinder werden nirgendwo sonst referenziert, dann können Aufräumarbeiten länger dauern und die Laufzeit erhöhen. Faules freigeben schafft Abhilfe.
  - Programm und Speicherverwaltung sind eng verwoben
  - Zähler erhöht Platzbedarf (ist m.E. in OO irrelevant)
  - zyklischer Müll (also sich gegenseitig referenzierender Müll) wird weder entdeckt noch freigegeben. - Abhilfe: Verwende andere Strategie sobald Speicher verbraucht

Faules Freigeben:

Stackartige Strategie. Blöcke werden nicht rekursiv freigegeben, sondern lediglich der zu löschende Block, der sofort in die Freispeicherliste eingetragen wird. Die referenzierten Elemente werden erst bei Entnahme des gelöschten Blockes betrachtet und deren reference counter dekrementiert. Nimmt dieser den Wert null an, so füge ihn der FS-Liste hinzu.

- Vorteile
  - begrenzter Overhead bei Freigabe
- Nachteile
  - Verlust der Lokalität
  - (große) Blöcke können lange als Müll verbleiben

Verzögertes Verwendungszählen:

Empirisch scheint ein Großteil der Zeigermodifikationen an lokalen Variablen und Argumenten und temporären Variablen zu erfolgen. Entsprechend werden beim reference counting lediglich diejenigen Referenzen betrachtet, die von Halde-Blöcken anderer Art ausgehen. Erreicht der RC den Wert null, so wird er in einer Null-Tabelle aufbewahrt und nicht direkt gelöscht, sondern erst sobald ein Halden-Obkjekt einen neuen Zeiger auf das Objekt der Null-Tabelle erhält. Die Null-Tabelle wird regelmäßig bereinigt, hierbei werden diejenigen Einträge bereinigt, die zur Zeit nicht mehr vom Stack referenziert werden (listing 6).

- Vorteile
  - die meisten Zeigeroperationen werden aufgrund der RC-Einsparung viel günstiger
- Nachteile
  - Null-Tabelle kann überlaufen
  - keine sofortige Freigabe mehr
  - reduzierte Lokalität

```
1  void reconcile(){
2     for N in Stack
3        N.RC++;
4     for N in zeroCountTable
5        if(!N.RC){
6           for M in Children(N)
7              delete(*M);
8           FreeBlock(N);
9        }
10    for N in Stack
11       N.RC--
12 }
```

Listing 6: Bereinigung der Null-Tabelle - Für die Dauer der Funktion werden die RC der Variablen des Stacks erhöht.

Schlanke Zähler:

Reference Counter erfordern normalerweise so viel Speicherplatz wie die Adressgröße, können in der Praxis jedoch mit einer Obergrenze ausgestattet werden, indem sie als *sticky* markiert werden und lediglich bis zu einer bestimmten Größe funktionieren. Was bei einem Überlauf erfolgen muss wird nicht erläutert.

Erkennen und Entfernen von zyklischen Mülls - Beispiel 4-57ff - code listing 7:

Hierfür werden den Blöcken verschiedene Zustände zugeordnet:
schwarz - In Benutzung
weiß - Müll bzw. freigegeben
grau - bereits während der Speicherbereinigung besucht
lila - potentiell Bestandteil eines Müll-Zyklus

- Erster Durchlauf - RC wird bei Nachfolgern von lila Blöcken dekrementiert, durch Zyklen entstandene Effekte werden entfernt (RC dekrementiert) und die lila Blöcke werden nach grau umgefärbt. - 4-58ff
- Zweiter Durchlauf - Graue Nachfolger eines lila Blocks, bei denen der RC der grauen Nachfolger null ist, werden weiß markiert. Sobald der RC bei einem Nachfolger größer null ist werden die RC wieder inkrementiert. - 4-64ff
- Dritter Durchlauf - Weiße Nachfolger des lila Blocks werden freigegeben. - 4-69

```
1  delete(T){
2     T.RC--;
3     if(!T.RC)
4        ... //wie bisher
5     else if(T.color != purple){
```

```
6         T.color = purple;
7         purpleList.push(T);
8      }
9  }
10
11 update(R,S){
12     delete(*R);
13     S.RC++;
14     R* = S;
15     R.color = black;
16     S.color = black;
17 }
18
19 collect(){
20     S = purpleList.pop();
21     while(S){
22         if(S.color == purple){
23             S.markGrey() //phase 1
24             S.scan() //phase 2
25             S.collectWhite() //phase 3
26         }
27         S = purpleList.pop();
28     }
29 }
```

Listing 7: Entfernung zyklischen Mülls

4.2 mark and sweep - Markieren und Auskehren

Mark and sweep - Markieren und Auskehren:

Müll bleibt bis zur nächsten Speicherbereinigung liegen. Die normale Berechnung pausiert während der Speicherbereinigung und es wird alles gemäß Tiefensuche markiert, was kein Müll ist, und der Rest wird entfernt. Also zwei Phasen:

- *erste Phase* - markiere alle erreichbaren Blöcke - Tiefensuche von Wurzel aus. - Kosten proportional zur Zahl lebendiger Objekte - das zweifel ich an, denn es kann Gruppierungen geben, die von vielen Objekten referenziert werden. In diesem Fall gilt es die Referenz durchaus zu verfolgen, nur weil das nicht eine neue Lebendigkeitsentscheidung zur Folge hat, heißt das für die Laufzeit nicht automatisch selbiges.
- *zweite Phase* - Block markiert? $\Rightarrow$ entferne Markierung, sonst gebe Block frei - Entlang des Heaps - Kosten proportional zur Haldengröße.
- Vorteile
  - Zyklischer Müll wird beseitigt

  - Zeiger-Aktualisierungen verursachen keine zusätzlichen Laufzeitkosten

- Nachteile
  - sehr lange Berechnungspause
  - schlechte Lokalität, da die Blöcke im Speicher verteilt werden, virtueller Speicher leidet unter Seitenflattern
  - darum auch externe Fragmentierung problematisch
  - Je voller der Speicher, desto öfter und länger läuft das Programm
  - Möglicherweise Speicherprobleme durch massive Rekursion bei langen Verzeigerungspfaden. - kann reduziert werden durch einen TODO-Stack und Reduktion der gleichzeitigen Abarbeitung auf ein Element

Verbesserungsmöglichkeiten für Mark and sweep:

- nicht-rekursiv
- Zeigerinvertierung (pointer reversal)
- Markierungstabellen anstelle der Objektmodifikation
- faules Auskehren

Markierungstabellen statt Objektmodifikation:

Jedes Bit der Tabelle repräsentiert eine möglichen Objektadresse.

- Vorteile
  - Ausschließlich die Tabelle ist zu modifizieren, diese passt evtl. sogar komplett in den HS
  - aus dem virtuellen HS verdrängte übrige Seiten müssen nicht auf die Platte swaped werden
- Nachteile
  - Zugriffe auf Objekte und Tabellen sind aufwendiger

Faules Auskehren:

Die Halde muss nicht komplett auf einmal ausgefegt werden, sondern man kann sich bei jeder Allokation auf eine feste Menge Ausfegearbeit beschränken.

4.3 Kopierend

Zustände bei mark and sweep und kopierenden Bereinigern:

- *schwarz* - Objekte und alle Kinder wurden besucht
- *weiß* - noch unbesucht - Objekte sind Müll, falls am Ende noch weiß
- *grau* - Objekt besucht, aber Kinder noch nicht oder die Kanten haben sich seit dem letzten Besuch geändert und müssen erneut untersucht werden

Kopierende Speicherbereiniger - Beispiel 4-100ff:

Halbieren die Halde, wobei die eine Hälfte die aktuellen Daten und die andere Hälfte obsolete Daten enthält. Bei einer Speicherbereinigung werden die Daten der aktiven Hälfte in die andere Hälfte kopiert, Zeiger angepasst und im Anschluss tauschen die Rollen der Hälften. Zur Korrektur von weiteren Zeigern, die ein Objekt referenzieren ist eine Weiterleitungsadresse, die auf die neue Kopie verweist, notwendig - 4-103ff. Dadurch merk man sich wo die Objekte hinkopiert werden, um bei einer ursprünglichen doppelten Referenz das entsprechende Objekte nicht mehrfach zu kopieren.

- Vorteile
  - Lebendige Daten werden kompatkifiziert ⇒ Reduktion der Fragmentierung
  - Dadurch kann u.U. auch Lokalität der Objekte verbessert werden, da die Objekte neu geordnet werden
  - bessere asymptotische Komplexität als naive Formen von mark and sweep, da der Aufwand proportional von der Größe der lebendigen Datenstrukturen abhängt und nicht von der Größe des Speicherplatzes.
  - Leicht zu implementieren
- Nachteile
  - Kopierende Speicherbereiniger beginnen früher zu flattern als naive mark and sweep Ansätze
  - Teures kopieren (Schreibzugriffe) statt setzen von Bits
  - Berechnungspause während Speicherbereinigungsphase
  - Halbierung des Speicherplatzes
  - Vernichtung sämtliche virtueller Speicherverwaltung durch Kopieren und Umschalten, dadurch schlechte Lokalität (Cachlines invalidiert)
  - Bei vollem Speicher steigt die Bereinigungs-Frequenz und der Slow-Down.
  - Korrekte Zeigererkennung ist von massiver Wichtigkeit, da sonst Nicht-Zeiger verändert werden

– Basis-Algorithmus rekurisv - besser Zeigerinvertierung - es gibt iterative Varianten mit Breitensuche - unterschiedliche Speicheranordnung Beispiel 4-114f

Traversierungsstrategie kann an das Speicherlayout der Datenstrukturen der Sprache angepasst werden, um häufig gemeinsam verwendete Objekte in einer Seite liegen. Die Erzeugungsreihenfolge lassen häufig Rückschlüsse auf die Lokalitätsbeziehungen, da top-down erzeugte Objekte auch in dieser Form verwendet werden.

Cheney's iterativer Bereiniger - Beispiel 4-119ff:

Kopiert nach inkrementeller Breitensuche, wodurch nicht nur die kopierten Objekte in der neuen Hälfte gespeichert werden, sondern auch die Liste der noch zu bearbeitenden Objekte. Die Pointer werden ganz normal rüber kopiert (die alten Pointer werden beibehalten) - erst bei der Abarbeitung des kopierten Objekts werden diese Pointer korrigiert und werden auf die neuen Objekte umgebogen.

4.4 Generationskonzept

Hypothesen:

- die meisten Objekte sterben jung (schwache Generationen-Hypothese)
- je älter desto geringer(!) die Sterblichkeit, alte Objekte bleiben mit hoher Wahrscheinlichkeit bis zum Programmende (negierte starke Generationen-Hypothese)

Generationskonzept - 4-135ff:

Man beschränkt sich also bei der Speicherbereinigung auf junge Objekte, wodurch der Aufwand für überlebende Objekte reduziert wird bei weiterhin relativ guter Speicherfreigabe. Die Halde wird also in drei Blöcke unterteilt: Junge Objekte (oft Bereinigt - kopierende Speicherbereinigung), Ältere Objekte (selten bereinigt), Bereinigungsblock. Ab einem bestimmten Alter werden Objekte in Bereiche für ältere Objekte verschoben. Hierbei kann das Alter über die Anzahl der überlebten Bereinigungen bestimmt werden. Entweder man führt einen Counter ein oder aber mehrere Stufen.

Sobald alte Generations-Objekte jüngere referenzieren wird dieses Verfahren komplexer (4-143f), da neue Wurzeln entstehen. Dafür versucht man mitzuschneiden wann diese Pointer erzeugt werden. Möglichkeiten dafür können sein: Entweder "passiert durch Garbage Collection vorher junges (nun altes) zeigt auf weiterhin junges" oder "wenn das Programm das selber tut: das ist etwas teurer, da hier eine neue Wurzel entsteht". Man merkt sich dafür in einem Puffer alle Adressen, in die ein Pointer geschrieben wird. Ist der Puffer voll, werden alle referenzierte Objekte betrachtet. Befindet sich dort eine Referenz von alt nach jung, wird die hash-Tabelle angefasst.

Referenzen von der jungen in die alte Generation sind eher selten und zwar entweder bei der Bereinigung, wenn Objekte verschoben werden (Bereiniger erkennt diese) oder beim expliziten Setzen von Referenzen, wobei etwas Zusatzaufwand (Referenzverwaltung s.u.) notwendig wird. Außerdem ist dies nicht zu berücksichtigen, sollte die neue Gerneration zusammen mit der alten bereinigt werden.

- Vorteile
  - Nur neue Daten werden besucht und kopiert, nicht mehr alle, denn die alten Daten bleiben auf angestammten Seiten
  - Dadurch wird die Lokalität für ältere Daten u.U. besser
  - kürzere GC-Pausen
- Nachteile
  - Laufzeitpausen während der GC (bei Inkrementellen und nebenläufigen nicht der Fall)
  - Sind nur dann hilfreich, wenn junge Objekte gemäß der schwachen Generationen-Hypothese früh sterben. (bei Inkrementellen und nebenläufigen nicht der Fall)

Referenzverwaltungsansätze:

- Indirektionslisten - entry sets - 4-147ff (unpraktisch)
  - Nachteile
    - Indirektionslisten können Einträge für dasselbe Objekt mehrfach enthalten wodurch die Laufzeit von der Zeigerzuweisungsanzahl und nicht von der verweisenden Objektanzahl abhängt
    - Indirektion erhöht Laufzeitkosten
- Erinnerungsmengen - remembered sets - 4-150ff
- Markierungskarten - card marking - 4-154ff

Erinnerungsmengen - Beispiel 4-150ff:

Zeigerzuweisungen von einem alten Objekt ausgehend in die neue Generation werden in einem sequentiellen Puffer notiert (ggf. mehrfach). Sobald dieser unmittelbar vor der Bereinigung voll ist, so werden die Einträge mit Zeigern zwischen Generationen in eine Erinnerungsmenge (Hashtable) übertragen, wobei etwaige Dubletten entfernt werden. Die Einträge der Erinnerungsmenge fungieren als zusätzliche Wurzeln. Wichtig ist, dass die Puffer schnell beschrieben werden können, um die Kosten der Zeigermodifikationen gering zu halten.

- Vorteile

  - Bereinigungskosten Abhängig von der Zeigeranzahl, die von der alten Generation in die neue verweisen, nicht von der gesamten Anzahl von Zeigerzuweisungen

- Nachteile
  - Irrelevante alte Objekte sind auch dann zu untersuchen, wenn die Referenz bereits nicht mehr in die junge Generation verweist.

Markierungskarten - 4-154ff:

Die alte Generation wird in mehrere Adressbereiche unterteilt, wobei für jeden dieser ein Bit auf den Markierungskarten existiert. Ein solches Bit wird markiert, sobald in dem entsprechenden Abschnitt ein Zeiger auf ein junges Objekt eingetragen wird. Die Objekte der Abschnitte werden bei der Bereinigung auf Zeiger untersucht. Einmal gesetzte Markierungen werden nicht entfernt, auch dann nicht, wenn die Referenzen nicht mehr in die junge Generation verweisen.

- Vorteile
  - extrem wenig Zusatzaufwand bei der Zeigermodifikation

- Nachteile
  - Sequentielles Adressbereich-Ablaufen bei der Bereinigung erhöht den Aufwand
  - Wissen über Datentypen ist aufgrund des seq. Ablaufens notwendig, was bei Objekten, die sich in mehreren Adressbereichen aufhalten, etwas problematisch sein kann.

Trade-Off Generationsgröße:

Mehr Platz für die junge Generation erhöht den Arbeitsaufwand bei der Bereinigung, verringert dafür jedoch ihre Frequenz. Dadurch erlangen Objekte mehr Zeit um zu altern, wodurch jedoch auch mehr Zeiger zwischen den Generationen entstehen. Heuristiken können z.B. mehrstufiges Altern betrachten.

4.5 Inkrementelle und nebenläufige Speicherbereinigung

Problem bei Generationen sind die Laufzeitpausen während der GC und die Notwendigkeit der Erfülltheit der Generationen-Hypothese. Sowohl Inkrementelle als auch nebenläufige Speicherbereinigung besitzen diese Problematik nicht. Bei beiden ist das Ziel stets mehr Speicher frei zu halten, als die Anwendung benötigt. Die Färbetechnik der mark and sweep sowie kopierenden Bereinigern ist auch für diese Arten hier nützlich.

Inkrementell:

Anwendung ruft die Speicherbereinigung immer wieder auf, wobei ein Teil des Speichers aufgeräumt wird. Verwendungszähler mit fauler Freigabe sind inkrementell, aber haben viele Nachteile. Besser ist eine Erweiterung der zeigerverfolgenden Verfahren.

Nebenläufig:

Bereinigung läuft parallel zur Anwendung (anderer Kontrollfluss auf anderem Prozessor). Problem der Nebenläufigkeit ist aufgrund einer schwarz-weiß-Referenz offensichtlich 4-162ff. Deswegen nimmt man lieber in Kauf, dass nicht erreichbare Objekte als aktiv eingestuft werden.

Entstehung Schwarz-weiß-Refernzen:

1. In einem schwarzen Objekt wird ein Zeiger auf ein weißes eingetragen. Das weiße Objekt ist aus der Wurzelmenge erreichbar, was der Bereiniger noch nicht bemerkt hat, was er jedoch tut solange nicht
2. der ursprüngliche Zeiger auf das weiße Objekt gelöscht wird und dieser die letzte Referenz auf das weiße Objekt war.

Lösungsansätze für das Nebenläufigkeitsproblem:

- Lese-Barriere - Der Zugriff auf das weiße Objekt wird abgefangen und der weiße Block wird grau gefärbt (teuer)
- Schreib-Barriere - Blockade der Schreibzugriffe und Markierung der Objekte, so dass das weiße Objekt vor den anderen besucht wird. Realisierbar über zwei Möglichkeiten
  - Anlegen eines Schnappschusses mit nachträglichem Einfärben, wodurch eben das entfernen der Referenz auf das weiße Objekt verhindert wird - 4-172ff. Dadurch wird Müll erst in der übernächsten Bereinigung entfernt (da im nächsten grau markiert). Dies widerspricht der schwachen Generationenhypothese. - verhindert den ersten Punkt der schwarz-weiß-Referenz
  - nachträgliches Einfärben nach Dijkstra oder Steele - 4-180ff. Das Objekt, das jetzt neu referenziert wird, wird erneut als grau markiert (oder das Objekt was das neu referenzierte Objekt referenziert), was dazu führt, dass der GC sich das Objekt nochmal anschauen. Weiße jung gestorbene Objekte werden in der nächsten Bereinigung entfernt. Dadurch das junge überlebende Objekte jedoch teuer über Zeigerverfolgung besucht werden müssen, ist dies nur bei hoher Sterblichkeit junger Objekte gut. Graue oder schwarze jung gestorbene Objekte werden auch hier erst in der übernächsten Bereinigung entfernt.

grau-Marken:

Es können für die drei Farben auch nur ein Bit statt zwei Bit verwenden, wenn markierte graue Objekte zur Unterscheidung von schwarzen Objekten zusätzlich in einer grau-Liste landen. Dadurch müssen graue Objekte nicht gesucht werden.

Nicht-harte Refernztypen:

- *Soft-Reference* - es kann weg, wäre aber schön wenn es bliebe
- *Weak-Reference* - Der GC darf das Objekt löschen, wenn ihm der Speicher volläuft. Beispiel gecachete Webseiten, das ist unproblematisch wenn es weg kommt
- *Phantom-Reference* - Keine Reaktivierung des Objekts mehr möglich, da bereits aufgeräumt. Es ist lediglich feststellbar, dass es sich in einer Queue befand.

5 Statische Analyse und Symbolische Ausführung

Satz von Rice:

Jede nichttriviale Eigenschaft eines Programms ist unentscheidbar.

Vollständigkeit - completeness:

Ist ein Programmfragment frei von Fehlern, so kann eine vollständige Analyse dies feststellen, es werden jedoch nicht zwingend alle Fehler gefunden. ⇒ false negatives.

Korrektheit - soundness:

Ist ein Programmfragment von einer korrekten Analyse für fehlerfrei befunden worden, so ist es das auch, es werden jedoch nicht zwingend alle fehlerfreien Stellen gefunden. ⇒ false positives.

Intervallabschätzung, Weitungsoperator ∇, Verengungsoperator Δ - 5-9ff:

Um die Werte von Variablen abzuschätzen werden Intervalle verwendet und diese können durch den Weitungsoperator ∇ vereinfacht werden.

$$(l_1, h_1)\nabla(l_2, h_2) = (l_0, h_0), \qquad l_0 = \begin{cases} -\infty, & l_2 < l_1 \\ l_1, & sonst \end{cases}, h_0 = \begin{cases} \infty, & h_2 > h_1 \\ h_1, & sonst \end{cases}$$

Ebenso kann eine durch den Weitungsoperator erreichte Lösung durch den Verengungsoperator Δ verbessert werden.

$$(l_1, h_1)\Delta(l_2,h_2) = (l_0, h_0),$$
$$l_0 = \begin{cases} l_2, & l_1 = -\infty \\ \min(l_1, l_2), & sonst \end{cases}, h_0 = \begin{cases} h_2, & h_1 = \infty \\ \max(h_1, h_2), & sonst \end{cases}$$

Abstrakte Interpretation:

Diese gezeigten Möglichkeiten sind Beispiele für eine abstrakte Interpretation. Hierbei wird die gesammte Programmausführung analysiert.

Im Folgenden wurden einige SMT-Ansätze (Satisfiability Modulo Theories) gezeigt, die mit SMT Solvern gelöst werden. Die Grundidee ist, dass man für unbekannte Werte Symbole einführt und diese als logische Ausdrücke mitschleift, wodurch sich sämtliche Pfad-Optionen (und Endzustände) herleiten lassen. Da die Anzahl der Symbole hierbei sehr groß werden kann gibt es Ansätze diese auf das relevante Maß zu beschränken. Ein Beispiel hierfür ist Concolic Execution.

Symbolische Ausführung:

Es wird i.A. nur ein Teil der Ausführung analysiert.

Concolic Execution, dynamische symbolische Ausführung - 5-42:

Durch Concolic Execution soll die Symbole auf ein relevantes Maß reduzieren. Dadurch können automatisiert Laufzeitfehler gefunden werden, Testfälle erzeugt werden und Exploits generiert werden. Dies ist eine Mischung von konkreten und symbolischen Ausführungen und ist primär zur Fehlerüberprüfung in großen Anwendungen sinnvoll.

1. Programmausführung mit konkreter Eingabe
2. Aufbau der entsprechenden Pfadbedingungen pc
3. pc' Bilde eine Invertierung der Pfadbedingung
4. Suche eine erfüllende Belegung für die pc' mit SMT Solver
5. Verwende diese Belegung als neue konkrete Eingabe

Verbesserung der Concolic Execution für eigenen Anwendungsfall:

Es können auch mehrere Teile der Pfadbedingung invertiert werden, um relevante Programmteile zu erreichen. Diese Stellen kann man Finden, indem man minimale Distanz im KFG betrachtet.

1. Ordne jedem Ausdruck der Pfadbedingung einen Knoten im KFG zu
2. bestimme minimale Distanz zum nächsten interessanten Knoten
3. Füge (Pfadbedingung, Mutation, Distanz) nach Distanz sortiert in eine Warteschlange.

6 Virtuelle Maschinen, Just-in-Time-Compiler I

Naiver Ansatz - Simple Interpretation:

- Schleife mit Switch-Case über Instruktionscode. Für jeden Operanden wird in die Abarbeitungsfunktion gesprungen und am Ende zurück.
- Datencache wird überfüllt, da die Instruktionen dort stehen
- per default extrem viele Sprünge notwendig, da Sprung in die Instruktionen, zurück in die Schleife und vom Schleifenende an den Schleifenanfang.
- Die Sprungvorhersagen werden dadurch ziemlich schlecht.
- Vorteile
  - leicht zu schreiben
  - gute Portabilität, da der Interpreter nur einmalig geschrieben werden muss.
  - Sehr geringer Speicherverbrauch
  - Schneller Interpreterstart, da keine Initialisierung notwendig
- Nachteile
  - Extrem langsam - stetige Neuübersetzung des selben Codes, Cache wird überschwemmt und die Sprungparty steigert nicht die Produktivität.

Naiver Ansatz mit Inlining - Indirekt durchgefädelte Interpretation:

- besser ist es ein gewissen Inlining in den aufgerufenen Funktionen, die die Entscheidung für die folgenden Instruktionen vornimmt
- Der Code wird also nicht mehr in einer Schleife ausgeführt, sondern in jeder Operationsabarbeitungsfunktion dupliziert, wodurch nur noch ein Sprung zur nächsten Instruktion notwendig ist und nicht mehr zwei.

Meine Vermutung warum kein normales Inlining betrieben wird ist, dass der Code, der die tatsächliche Ausführung/Interpretation vornimmt, nachdem entschieden wurde was zu tun ist, schlicht zu groß werden kann, so dupliziert man lediglich ein paar kleine if-else-Kaskaden. Außerdem hat man sowieso schon einen Sprung aufgrund der Switch-Kaskade, der zweite Sprung zur Funktion ist dann sowieso unbedingt, der sich wegoptimieren lässt. Um den einen Sprung kommt man eben auch bei echtem Inlining nicht herum.

- Vorteile
  - Schneller Start
  - leicht zu schreiben
  - gute Portabilität, da der Interpreter nur einmalig geschrieben werden muss.
  - geringer Speicherverbrauch
  - Schneller Interpreterstart, da keine Initialisierung notwendig
- Nachteile
  - langsam - stetige Neuübersetzung des selben Codes, Cache wird überschwemmt und die Sprungparty steigert nicht die Produktivität.

Verbesserungsansätze - (teilweise Vorübersetzung):

- dekodierte Instruktion werden (etwas manipuliert) gecached und für folgende Ausführungen gespeichert
- Instruktions-Code durch die Adresse der auszuführende Instruktionsroutine ersetzen, wodurch in der dispatch-table nicht mehr nachzuschlagen ist.
- Instruktionen zu Grundblöcken zusammenpacken 6-23f

Vorübersetzung - Vorübersetzte Interpretation:

Vorübersetzung erlaubt Optimierungen, aber Start wird langsamer. Verbrauchte Größe wächst aber stark proportional zur Programmgröße. Es wird also das Ergebnis einer bestimmten Instruktion vorübersetzt und abgespeichert. Auch ist direktes Durchfädeln ein sinnvoller Ansatz, indem bestimmte Instruktionsroutinen vorübersetzt werden, dann spart man sich das Nachschlagen in der dispatch-Tabelle, da die Adresse den Instruktionscode ersetzten kann.

- Vorteile
  - Erzeugter Code ist schneller
  - Sprungzielvorhersage etwas besser
- Nachteile

- Portabilität leidet etwas, da Zwischencode Systemspezifisch sein kann
- Speicherverbrauch steigt
- langsamer Interpreterstart, da Vorübersetzung

Allgemeine Anmerkung:

VMs arbeiten häufig auf Bytecode und nicht auf Maschinencode, da bei CISC die Instruktionslänge nicht fest ist. Den Bytecode-Instruktionen hingegen kann man eine feste Länge zuweisen.

Nach der Vorübersetzung sind die gespeicherten Adressen nicht identisch mit denen die sich im Quellcode befinden. Hier ist also ein Mapping der Adressen notwendig. Auch hier wird die Berechnung der neuen Adressen durch eine feste Instruktionslänge einfacher.

Zusammenfassen von Instruktionen:

Mehrere Instruktionen können zu Blöcken zusammengefasst werden, die mit der selben Routine bearbeitet werden, wodurch der Code nur einmal im Cache liegen muss. Sehr hoher Platzverbrauch und langsamer Start, aber schnelle Ausführung.

Code-Cache Grundblöcke Optimierung:

Bei der Interpretation der Grundblöcke sind ebenfalls unglaublich viele Sprünge notwendig, aber auch hier funktioniert die selbe Idee wie oben. Es wird die fortlaufende Suche nach folgenden Instruktionen inlined. Es werden also direkte Sprünge durchgereicht. Dies ist dann möglich wenn der nachfolgende Block bereits übersetzt ist und sich noch im Cache befindet. Bei der Verdrängung müssen diese direkten Sprünge entfernt werden.

Einsparen von Nachschlagen in Sprungtabellen - lazy evaluation, Sprungzielvorhersage:

Viele Verzweigungen mit vielen Sprüngen erfordern viel Buchhaltung mit Nachschlagen in Tabellen. Ein Ansatz ist lazy evaluation. Falls der Code noch nicht übersetzt wurde, so kann für diesen auch noch kein Eintrag in einer Tabelle vorliegen. Idee ist es also in einem solchen Falle den Code zu übersetzen und die Adresse in die Tabelle einzutragen und anzuspringen. Dadurch muss nicht erst in der Tabelle nachgeschlagen werden. Ansonsten wird in der Tabelle nachgesehen und diese Adresse angesprungen.

Ebenfalls können in Software bestimmte Vorhersagen getroffen werden, bevor das Nachschlagen in der Tabelle erfolgt. Z.B. wenn die Variable von einem bestimmten Typen ist und dann offensichtlich ist welche Funktion anzuspringen ist, dann wird dies gemacht, ohne in der Tabelle nachzuschlagen.

Neuübersetzung und Austausch von Code:

Wird in ein als Read-Only markierten Code-Bereich geschrieben, so wird eine Ausnahme geworfen, die abgefangen werden kann, der alte Code kann entfernt und durch neuen ersetzt werden.

Cache-Verwaltungsstrategien:

- *LRU* - starke Fragementierung, hoher Zähleraufwand, Zeigerkorrekturen bei Referenzen innerhalb des Caches
- *flush when full* - keine Fragmentierung und keine Zeigerkorrekturen. Die Cachefüllung passt sich automatisch verschiedenen Programmphasen[3] an. Allerdings wird Arbeit mehrfach verrichtet da Übersetzungen wiederholt werden müssen.
- *preemptive flush* - Annahme: ein Phasenwechsel wird durch viele Neuübersetzungen signalisiert. Wird dies bemerkt, so wird der Cache komplett geleert.
- *FIFO* - Fast keine Fragmentierung, da der Cache zyklisch zugewiesen wird. Auch muss nicht gezählt werden. Allerdings können älteste Blöcke durchaus noch verwendet werden und darüber hinaus sind weiterhin Zeigerkorekturen notwendig.
- *Grobgranulares FIFO* - der Cache wird in ein paar wenige relativ große Blöcke unterteilt. Wird neuer Platz benötigt, so wird ein kompletter Block auf einmal aufgeräumt. Vorteile von FIFO bleiben bestehen, aber die notwendigen Zeigerkorrekturen werden weniger, da Korrekturen von Blöcken sinnlos sind, wenn diese ohnehin gleich entfernt werden. Einzig die alten verwendeten Objekte gehen weiterhin verloren.

Neuübersetzung mit mehr Optimierungen:

Neuübersetzung von bereits übersetztem Code kann sinnvoll sein. Weil z.B. zur Laufzeit bemerkt wird: diese Funktion wird häufig aufgerufen, wird erneut mit Optimierungen (O3 o.ä.) übersetzt. Ebenso müssen bereits aus dem Instruktionscache verdrängte übersetzte Teile neu übersetzt werden, falls man dort erneut hinspringen möchte. Dadurch werden auch die Sprünge, die auf diese verdrängten Bereiche passieren, ungültig. Entsprechend muss Buch geführt werden, um bei Verdrängung die Sprünge auf Default setzen zu können.

Problem bei Mehrfachübersetzung:

Ein großes Problem bei der Füllung des Instruktionscaches sind viele unterschiedlich übersetzte Varianten von einzelnen Codestücken, die häufig übersetzt wurden, weil der JIT viel rumprobierte, um die Ausführungszeit nach unten zu pressen.

[3] Der zur Zeit ausgeführte Code kann je nach aktueller Laufzeit des Programms variieren und sich über die Zeit anderen Funktionalitäten zuwenden.

Heißer Code:

Heißer Code ist solcher, der häufig ausgeführt wird und am besten mit höheren Optimierungsgraden optimiert werden sollte. Es reicht meistens die Eingangsknoten, die Ziele von Rückwärtskanten und die Nachfolger heißer Knoten zu messen, um zu überprüfen ob diese heiß sind. Man unterscheidet verschiedene Ansätze. Die Kosten von Messungen können reduziert werden, indem nur für eine kurze Zeit gemessen wird, nicht permanent. D.h. der Code wird kurzzeitig mit Messungen versehen und anschließend wiederhergestellt. Dadurch sind die bedingten Sprünge zum Zählen nur kurzzeitig aktiv - Problem ist aber wenn in diesem Code Endlosschleifen sind, die man nicht trivial wieder verlassen kann. Dann muss man das zurückschalten über die Rückwärtskanten vornehmen.

- Blockprofile - Zählen der Block-Besuche
- Kantenprofile - Zählen der Kanten-Besuche - hierbei sind mehr Zähler notwendig als bei Blockprofilen, aber diese können anscheinend mit Algorithmen reduziert werden
- Pfadprofile - Betrachtung der kompletten Pfade - eher ungeeignet für dynamische Optimierung, da sich diese erst mit der Zeit entwickeln.
- Methodenprofile - Zählen der Funktionsaufrufe - daraus wird die darin verbrachte Zeit abgeschätzt
- Stack - periodische Betrachtung von welchen Funktionen Elemente auf dem Stack liegen - Messung dauert häufig etwas länger

Verwendung der Messergebnisse:

Alte wichtige Elemente können mit der Zeit in einer anderen Programmphase irrelevant werden. Weiterhin sollten die Zähler nicht überlaufen. Daher ist es sinnvoll eine Halbwertzeit zu verwenden, so dass nach bestimmten Intervallen die Zähler halbiert werden. Diese Messergebnisse können unter anderem für Folgendes verwendet werden.

- Schleifenausrollen von heißen Schleifen oder bekannter Wdh-Zahl
- Registerallokation - für kalte Pfade wird ggf. Spill-Code erzeugt.
- Ausnahmebehandlung - effiziente Behandlung häufiger Fehler
- Prefetching von Sprungzielen

Trampolinfunktion:

Um den Aufruf unabhängig davon zu machen welche Funktion wie aufzurufen ist - mit welchem Optimierungslevel diese aufgerufen werden soll, ob diese zu übersetzen

oder lediglich zu interpretieren ist, ob es eine native Funktion ist, sprich bereits übersetzt wurde oder vielleicht sogar eine bereitgestellte Bibliotheksfunktion ist - ist eine Trampolinfunktion sinnvoll, die all dies wegabstrahiert.

- Vorteile
  - Einheitlicher Methodenaufruf im Code
  - Zentrale Stelle für Optimerungsentscheidungen.
    - Seltene Methoden interpretieren
    - Häufige Methoden übersetzen
    - Heiße Methoden optimieren
- Nachteile
  - Indirektion bei Aufruf von nativen Funktionen unnötig. Kann aber durch vorherige Fallunterscheidung vermeiden.

7 Virtuelle Maschinen, Just-in-Time-Compiler II

Code-Umordnung ⇒ Code-Lokalität:

Verbesserung der Optimierbarkeit und der Instruktionscache-Verwendung durch Vergrößerung der Blöcke. Inlining von Methoden reduziert Sprünge und Aufruf-Overhead. Seltener Code kann angesprungen werden, wohingegen der Rest nah beieinander liegen sollte. Diesen Rest kann man über heiße Pfade bestimmen.

Finden heißer Pfade - Ablaufspur - 7-10, fig. 3:

1. Der heißeste Block zum Anfang einer Ablaufspur
2. Füge den Knoten zur Ablaufspur hinzu, der anhand des Kantenprofils über die heißeste Kante erreichbar ist, bis
   - Block gehört bereits zu einer anderen Ablaufspur
   - Erreichen eines Prozeduraufrufs oder eines Returns
   - maximale Länge der Ablaufspur erreicht
3. Wiederhole das Vorgehen bis alle Blöcke in einer Ablaufspur enthalten sind

Finden heißer Pfade - Superblöcke - 7-12, fig. 3:

1. Überschreitet ein Block einen Starschwellwert und ist bisher kein Teil eines Superblocks, so wird er zum Anfang eines Superblocks
2. Aufnahme von Nachfolgekanten, die zuerst einen Schwellwert überschreiten werden in den Suberblock aufgenommen, bis
   - Erreichen eines Startblocks eines Superblocks
   - Superblock erreicht maximale Länge
   - keine Nachfolgekante erreicht den Schwellenwert
   - Erreichen eines Sprungbefehls oder Prozeduraufrufs
3. Wiederhole das Vorgehen

Ablaufspur vs. Superblock:

Superblöcke besitzten nur einen Eingang, Ablaufspuren nicht. Bei Ablaufspuren werden Blöcke betrachtet, wohingegen bei Superblöcken Kanten betrachtet werden, weshalb Blöcke auch mehrfach Teil verschiedener Superblöcke sein können. Ablaufspuren bilden solche Sub-Pfade die in ihrem Elter-Pfad am Wahrscheinlichsten genommen werden, müssen jedoch nicht zwingend warm geschweige denn heiß werden. Superblöcke hingegen orientieren sich stets an Schwellwerten, wodurch garantiert ist, dass ausschließlich heiße Kombinationen erfasst werden.

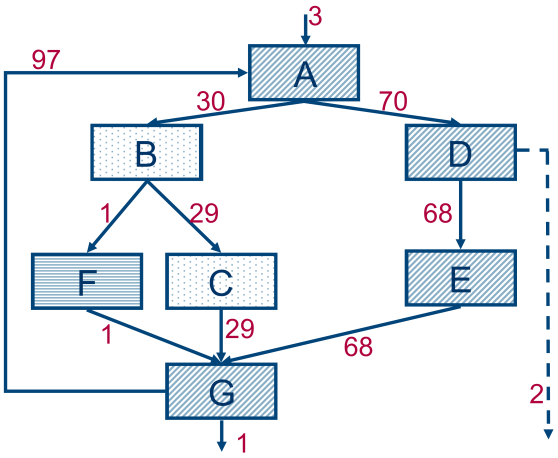

Abbildung 3: Ablaufspuren - Vorhandene Superblöcke:
A, D, E; B,C; F; G
Superblöcke - Vorhandene Superblöcke bei Startschwellwert: 100, Folgeschwellwert: 50:
A, D, E, G; B,C,G

Optimierungsmöglichkeiten:

Es kann sinnvoll sein Spezialfälle gesondert abzufragen und diese mit optimierten Instruktionen zu übersetzen und im unspezialisierten Fall den ganz normalen Code auszuführen. Dadurch lassen sich u.U. teure Instruktionen durch günstigere ersetzen. Insbesondere kann es sinnvoll sein häufig verwendete Blöcke zu duplizieren, um Optimierungsmöglichkeiten für verschiedene Szenarien zu erhöhen. Dabei kann man den unspezialisierte Fall auch erst on demand übersetzen. Dieses Verfahren ist auch für Methoden-Inlining nach Typ-Analyse sinnvoll. Es wird anhand von Datenflussanalyse und Klassenhierarchien eine Abschätzung bzgl. der möglichen Typen einer Variable getroffen.

Der Compiler sollte allerdings stets (stichprobenartig) überprüfen, ob diese Optimierung, die er für günstig hielt, auch tatsächlich positive Effekte mit sich bringt, indem er die Laufzeit misst und bei schlechten Resultaten den ursprünglichen Code wiederherstellt. Optimierungen sind hier häufig mit anderer Herangehensweise vorgenommen als bei statischen Übersetzern, da der Compiler hier denkt: “Wenn das hier so wäre, dann könnten wir optimieren. Deswegen untersuche doch mal dieses kleine Stück Code und analysiere diese Gegebenheit”

Probleme bei Ersetzung des Runtime Stacks:

Angenommen eine Methode, die sich gerade in Ausführung befindet soll ersetzt werden, z.B. weil festgestellt wird, dass ein vorgenommenes Inlining eine schlechte Idee war und der Compiler möchte dies rückgängig machen. So muss nachträglich eine weitere Methodenschachtel in den Stack eingefügt werden, wobei spezialisierte Methoden u.U. andere Methodenschachteln benötigen. Die Idee hierbei ist es die optimierte Methodenschachtel vom Stack zu entfernen, durch eine zu erzeugende konzeptuelle zu ersetzen, die nahe an der ursprünglichen abstrakten Vorstellung ist, also ohne Spezialisierungen etc. Bei einer etwaigen Neuübersetzung kann erneut eine optimierte, implementierungsabhängige Methodenschachtel erzeugt werden.

Wenn Code während der Ausführung gerade inlined wird, so passt der Code nicht mehr zu dem aktuellen Stack. Also müssen die Methodenschachteln angefasst werden. Das Austauschen dieser Schachteln erfolgt, indem man die spezialisierte Version an anderer Stelle im Code anlegt, dort optimiert und anschließend wieder an den Stack klebt 7-27f

Darüber hinaus müssen Statusänderungen durch den neue Code auch für nebenläufige Kontrollflüsse sichtbar sein. Auch kann leicht nicht-deterministisches Verhalten auftreten.

dynamisch einmalige Zuweisungen:

Wird einer Variable einmalig ein Wert zugewiesen, der statisch nicht bekannt ist, so können hier viele in CB2 besprochene Optimierungen, wie z.B. Konstantenfortschreibung angewendet werden.

Optimierungsstufen JikesRVM:

JikesRVM verfügt über verschiedene Optimierungsstufen, wobei die aggressivste sogar SSA und Graphfärben durchführt.

Registerallokation - Linearer Scan:

Registerzuteilung kann dynamisch per default mit *linearem Scan* erfolgen, da Registerfärben nur für sehr heiße Code-Teile lohnend. Beim *linearem Scan* werden Registerzuteilungen beim durchlaufen vorgenommen, ohne im weiteren Verlauf getroffene Entscheidungen zu überarbeiten.

Merkwürdigerweise ist das Lebendigkeitsintervall hier der komplette Bereich vom ersten beschreiben bis zum letzten Lesen, inklusive der Bereiche in denen ein neuer Wert geschrieben wird. Lebendigkeitslöcher bleiben aus Kostengründen unbeachtet. Es wird also einmal durchgelaufen, Lebendigkeitsinvertalle gebildet und beim zweiten Durchlauf erfolgt die Zuteilung gemäß der Heuristik zu den Registern. Die in 7-47ff gezeigte Heuristik ist ziemlicher mist, aber das Verfahren wird klar. Die Sortierung der Variablen ist relevant.

Der Aufwand ist zur Anzahl der symbolischen Register proportional. Unschön sind große Lebendigkeitslöcher, Ignorieren von Schleifen und Kontrollflüssen.

Die Lebendigkeitsintervalle betrachten keine Löcher, da Löcherbestimmung bei komplexeren Kontrollflussgraphen komplex werden kann und unter Umständen nicht mehr trivial ist. Weiterhin entstünden dadurch relativ viele Schiebeoperationen, die man sich evtl. sparen kann. Für Schleifen ist der Algorithmus ziemlich unglücklich, aber die Registerverteilung ist sehr viel schneller als unter Beachtung der Lebenedigkeits-Löcher. Schleifen werden als ein Knoten betrachtet (starke Zusammenhangskomponenten), weil dies Dinge vereinfacht. Unter anderem wird Branching innerhalb von Schleifen stark vereinfacht, weil es dadurch nicht notwendig wird den Kontrollfluss zu überwachen und zu prüfen, ob die Variable nun wirklich vor dem Lesen beschrieben wurde.

Ist es innerhalb von Schleifen nicht möglich alle Register, sondern lediglich einen Teil der Schleifenvariablen zuzuweisen, so wird die Zuweisung zufällig getroffen, da diese Variablen die selbe Lebendigkeitsspane vorweisen.

Solange keine Kollisionen entstehen werden Register zugeteilt. Kommt es zu Kollisionen, so wird derjenigen Variable, die erst am spätesten ihre Lebendigkeit verliert niemals ein Register zugewiesen. Also auch nicht weiter vorne, wo es dafür keine Kollision gäbe (das kann man natürlich schlauer machen, ist allerdings auch aufwändiger). Wichtig ist also nicht zwingend die Länge der Lebendigkeit, sondern vielmehr die verbleibende Lebendigkeit pro Programmpunkt. Die Heuristik dahinter ist schlüssig: Je länger diese Variable noch von dem aktuellen Punkt ein Register belegt, desto eher treten hier Kollisionen mit folgenden Variablen auf. Das beachtet zwar in keinsterweise die Lesehäufigkeit der Variablen, ist dafür allerdings trivial zu implementieren und in der Ausführung recht zügig.

Eine Verbesserung die sich meiner Meinung nach relativ leicht integrieren ließe wäre eine Beachtung von Schleifenvariablen. Sprich aufgrund des Mappings der SCC (starken Zusammenhangskomponenten) zu den zugehörigen Programminstruktionen sollte es möglich sein Schleifen zu detektieren. Denn jene starke Zusammenhangskomponenten, die mehr als eine Instruktion beinhalten sollten eine Schleife bilden. Instruktionen die sich innerhalb dieser Schleife befinden sind wichtiger als äußere - man kann also versuchen die dort gelesenen Operanden mit einer höheren Gewichtung zu belegen und ihnen präferiert die Register zuzuweisen.

bin packing - Integrierte Code-Umschreibung - Beispiel 7-61ff:

Bei jeder Verwendung eines symbolischen Registers wird überprüft, ob es eine höhere Priorität vorweist als die in den realen Registern befindliche. Ist dies der Fall, so wird Spill-Code für eines dieser anderen symbolischen Register erzeugt und im nachfolgenden Code wird das höherpriore symbolische Register durch das reale Register ersetzt. Bei Zusammenfluss mehrerer Blöcke mit verschiedenen Registerzuordnungen wird Spill- oder Kopiercode eingefügt.

Inlining:

Lohnt sich besonders

- bei kleinen Funktionen
  - da dort der Funktionsaufruf verhältnismäßig viel Overhead bedeutet
  - Registerdruck wird nicht zu groß
  - Codegröße wächst in einem akzeptablen Ausmaß
- bei heißen Funktionen
  - Funktionsaufruf aus einem Schleifenrumpf heraus
- Vorteile
  - geringerer Aufruf-Overhead
  - größere Grundblöcke, wodurch u.U. bessere Optimierung ermöglicht wird
  - Konstantenweitergabe, Dead-Code-Elimination
- Nachteile
  - Code-Wachstum ⇒ negativer Einfluss auf den Instruktionscache
  - Höherer Registerdruck

Inlining im Quelltext:

- Nameclash durch Inlining ist zu vermeiden ⇒ Umbenennung vor dem Inlining. Die Variablen der inlinten Funktion umzubenennen sollte meistens reichen, wenn der Programmierer allerdings die Variablen in der inlinenden Funktion nach genau diesem Umbenennungs-Schema wählte, dann muss man das auch in der inlinenden Funktion tun oder es dem Programmierer verbieten diesen Clash zu provozieren.
- Der Stackframe der inlinenden Funktion wird größer
- Instruktionen verschieben sich
- Sprünge BEIDER Funktionen verschieben sich.
- Vorzeitige Returns sind schwierig.
- Vorteile
  - Parameterübergabe durch Zuweisung möglich
  - Rückgabewertübergabe durch Zuweisung möglich
- Nachteile
  - frühzeitiges Return verkompliziert den Prozess

  - Variablenumbenennung notwendig

Inlining im AST:

- Vorteile
  - Parameterübergabe durch Zuweisung möglich
  - Rückgabewertübergabe durch Zuweisung möglich
  - Variablen sind eindeutig, müssen also nicht umbenannt werden, sondern nur die Einträge der Namenstabelle angepasst werden
- Nachteile
  - frühzeitiges Return verkompliziert den Prozess

Inlining im Kontrollflussgraphen:

- Vorteile
  - Parameterübergabe durch Zuweisung möglich
  - Rückgabewertübergabe durch Zuweisung möglich
  - frühzeitiges Return sind leicht zu handhaben
- Nachteile
  - Codeerzeugung wird schwieriger, da Sprungkanten und fall-through-Kanten berücksichtigt werden müssen
  - Variablenumbenennung nicht im selben Rutsch möglich

Inlining im Assembler-Code:

- Vorteile
  - Parameterübergabe durch Zuweisung möglich
  - Rückgabewertübergabe durch Zuweisung möglich
  - Return ist wie ein normaler Sprung behandelbar
- Nachteile
  - semantische Informationen nicht mehr vorhanden
  - Variablen-Offsets sind anzupassen
  - Registervergabe kann schwieriger werden
  - Funktionsprolog und -epilog dürfen nicht mit kopiert werden

Inlining im Zwischencode/Bytecode:

Zu beachten:

- Sprungziele verschieben sich in beiden Funktionen deswegen wird unter anderem die Behandlung von frühzeitige Return-Anweisungen schwierig
- Variablen-Offsets verändern sich. Die einfachste Lösung ist $offset_{neu} = offset_{alt} + \max\{offset\}_{caller}$
- Stackframe-Größe ändert sich
- Inlining muss bei rekursiven Funktionen begrenzt werden. Ist keine Kreuzrekursion möglich, so ist dies trivial über den Funktionsnamen möglich. Falls Kreuzrekursionen doch möglich sind, so muss man wohl jedesmal einen Zähler hochzählen.
- Vorteile
  - Parameterübergabe durch Zuweisung möglich
  - Rückgabewertübergabe durch Zuweisung möglich
  - Return ist wie ein normaler Sprung behandelbar
- Nachteile
  - Anpassung der Sprungziele notwendig
  - Anpassung der Variablen-Offsets notwendig
  - Zuordnung der Argumente zu den Funktionsaufrufen notwendig

8 Wettlaufsituationen

Fehlerauftrittshäufigkeit:

Test-Ausführungen triggern den Fehler...

- ... immer
- ... manchmal
- ... nie, tritt aber später im Produktivsystem auf

Arten von Wettlaufsituationen:

- Real Races - Tatsächlicher Fehler, der fehlerhafte Ausführung zur Folge haben kann
- False Races - Wettlaufsituation kann nicht auftreten - eigene Objekt-/Speicherverwaltung - Prüfung kann unsynchronisiertes Beschreiben der selben Speicherstelle ermitteln, die jedoch nicht Fehlerhaft sein muss.
- Benign Races - echte Race, die keine fehlerhafte Ausführung erzeugen - Beispiel zwei schreiben gleichzeitig den selben Wert

Lösungsansätze:

- Typsystem - durch Spracherweiterung
- Locksets
- Happened-Before
- Hybride Verfahren

Typsystem - Concurrent - Clang:

Ist eine nette Spracherweiterung für die meisten Sprachen, kann beschreiben “Beschreibung von Variablen erfordert beispielsweise das vorherige Locken von Mutexen”. Hierbei können false races getriggert werden.

```
1 "#include mutex.h"  //Wrapperklasse, die die Annotation enthält
2 //Übersetzen dieses Codes: clang++ -Wthread-safety
3 class Racing{
4    Mutex mu;
5    //signalisiert: Jedes beschreiben der Variable "val" muss
        mit "mu" abgesichert werden.
6    int val GUARDED_BY(mu);
7
```

```
8     //triggert zur Compilezeit eine Warning, da das Mutex nicht
          gelockt wird
9     void foo(int c){
10       val += c;
11    }
12
13    //Der Aufrufer der Funktion muss das Lock besitzen, sonst
          wirft das zur Laufzeit einen Fehler
14    void bar(int c) REQUIRES(mu){
15       val += c;
16    }
17
18    //wirft zur Compilezeit eine Warning, da der Compiler glaubt
          es gäbe einen Pfad, auf dem "val" beschrieben, aber "mu"
          nicht zuvor gelockt. False Race
19    void baz(int c){
20       if(c > 0)
21          mu.lock();
22       else if(c <= 0)
23          mu.lock();
24       val += c;
25       mu.unlock();
26    }
27 };
```

Listing 8: Concurrent-Clang

Bewertung Typsystem:

- Vorteile
  - Frei von Nebenläufigkeitsfehlern
  - Deadlockerkennung
- Nachteile
  - Ist eine Spracherweiterung
  - Manuelle Annotation erforderlich wodurch Fehler passieren können

Algorithm 2 Locksysteme - Lockset-Analyse
Für jede Variable und jede Verwendung dieser betrachte die Menge an Locks, die gesperrt sind. Ist die Schnittmenge für jede Variable leer, so existiert möglicherweise eine Wettlaufsituation.

```
curLockSet := []
for  do∀i ∈ Variables:
    varSet[i] := {all locks}
end for
while not done do
    if lock(lo) then
        curLockSet.append(lo)
    else if unlock lo then
        curLockSet.remove(lo)
    else if access variable v then
        varSet[v] := varSet[v] ∩ curLockSet
        if varSet[v] == {} then
            report possible race
        end if
    end if
end while
```

Bewertung Lockset-Analyse:

- Vorteile
  - Auch zur Deadlockerkennung nutzbar, wenn zusätzlich die Reihenfolge gespeichert wird
  - Schnell
- Nachteile
  - Semaphoren können sowohl als Condition Vars, als auch als Locks verwendet werden
  - Erfordert gute Testfälle
  - false positives

Happened Before:

Die Idee ist es eine strikte partielle Ordnung von Ereignissen herzustellen. Hierbei wird die Relationen $a > b$ gebildet, sofern a eine Nachricht an b sendet oder sich beide auf dem selben System befinden und a vor b ausgeführt wird. Offensichtlich ist diese Ordnung auch transitiv. Häufig wird dieser Ansatz verwendet um logische Uhren miteinander zu synchronisieren. Beispiel - fig. 4.

Darstellen lassen sich diese Relationen mit einem DAG, wobei diese Befehle frei von Wettlaufsituationen sind, in denen ein Pfad existiert, wohingegen in gegenteiligen Wettlaufsituationen auftreten können. Allerdings sind die Analysen von der konkreten Ausführung abhängig. Wenn man sagt ein *unlock()* ist zwingend vor dem nächsten *lock()* aufzurufen, so ist dennoch nicht garantiert, dass alles vor dem *unlock* vor dem anderen *lock* passierte, denn beispielsweise könnten die zwei Kontrollflüsse in anderer Reihenfolge ausführbar sein, wodurch die bisherige Beziehung falsifiziert werden würde.

Implementierungen finden sich in Valgrind, Helgrind, DRD, TSan

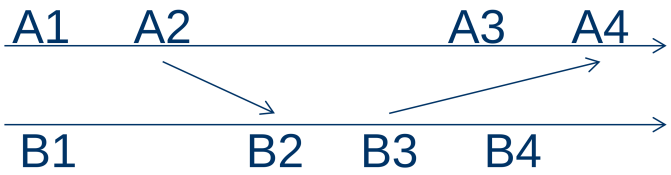

Abbildung 4: Beispiel Happened Before - Über A3 und B3 kann keine Aussage getroffen werden. Genau das ist die Stelle in der eine Wettlaufsituation auftreten kann.

Bewertung Happened Before:

- Vorteile
  - Keine false positives
- Nachteile
  - Aufwendig

TSan:

TSan wird von Google entwickelt und erzeugt eine relativ große Abdeckung. Beim Senden von Nachrichten wird die aktuelle Zeit des eigenen Threads und die Zeit des letzten Zugriffs auf eine Variable mit versendet, dadurch weiß der Empfänger wie weit der Sender beim Absenden der Nachricht war und wann dort auf Variablen zugegriffen wurde. Lässt sich am Ende keine eindeutige Happened Before-Ordnung, so können Wettlaufsituationen erkannt werden. Gemerkt werden sich die Funktions-Eintritt-, -Exit-Zeiten und Schreib und Lesezeiten. Daraus werden dann die Stack-Traces rekonstruiert, sobald Wettlaufsituationen vermutet werden, um den erforderlichen Speicherverbrauch geringer zu halten.

Es werden ShadowStates verwaltet, um Fehler zu erkennen, wobei jweils 8 Byte ein Shadow Word repräsentieren. Gespeichert wird Zugreifender Thread, aktuelle Zeit, Schreib-/Leseflag, Größe und Offset.

Einige Schreibzugriffe auf die ShadowStates nicht synchronisiert, wodurch durch Wettlaufsituationen im TSan selbst einige Wettlaufsituationen im untersuchten Programm verloren gehen könnten. Dafür erhöht TSan die Laufzeit "nur" auf ein bis zu 15-faches bei einer Verzehnfachung des Speicherverbrauchs. Compilerflag ist *-fsanitize=thread*

Hybrid-Ansätze:

Hybride Ansätze kombinieren die schnellen Lockset-Algorithmen zum erkennen möglicher Stellen und Happened-Before um false positives zu eliminieren.

Automatische Fehlerbehebung mit Satisfiability Modulo Theories (SMT):

Jeder Anweisung wird ein int-Literal o_i zugewiesen und a happened before b wird durch $o_a < o_b$ dargestellt.

Es werden viele Ablaufpläne erzeugt, in denen Wettlaufsituationen bestehen, dort wird die kleinste Menge gesucht in der der Fehler auftritt und anschließend die Wettlaufsituationen in den Abläufplänen. Das wird solange wiederholt bis keine Wettlaufsituationen mehr gefunden werden. Am Ende müssen aus all denen Schnittmengen gebildet werden, so dass sich diese nicht untereinander widersprechen. Lässt sich dies lösen, so findet man am Ende tatsächlich einen Ablaufplan, der keine Wettlaufsituationen mehr enthält.

Bewertung Automatische Behebung:

- Vorteile
  - Ablaufpläne
  - Automatische Fehlerbehebung
- Nachteile
  - Benötigt Bedingungen, die Korrektheit Prüfen (Asserts)
  - Auflösen der Formel ist schwer und das Lösen noch viel schwerer.
  - Ist entsprechend bisher nur für Programme der Größe von etwa 200 Zeilen.

9 Software Watermarking

Die vorgestellten Verfahren wurden lediglich so oberflächlich behandelt, dass ich es nicht für sinnvoll erachte diese in der Zusammenfassung aufzugreifen. Weder wurden sie mathematisch begründet noch in praktischem Umfang demonstriert. Letztendlich dienten sie wohl ausschließlich zur Inspiration existierender Möglichkeiten.

Watermarking soll geringe Modifikation an dem Original vornehmen, um das ursprüngliche Bild nicht zu zerstören oder Leuten zu signalisieren, dass offensichtliches Watermarking angewendet wurde.

Anwendungsfälle:

- Broadcast Monitoring - Warten auf ein bestimmtes Signal - Beispiel Werbung soll x-mal auf dem Sender y laufen. Dann kann man auf dieses Signal lauschen, um dies zu prüfen
- Eigentumsbeweis + Kopienverteilung mit eindeutigen Wasserzeichen, um zu verifizieren wer etwas verbreitet hat

Wichtige Eigenschaften:

- Robustheit
- Sichtbarkeit
- False-Positive-Rate

Menschliche Wahrnehmung:

Man Unterscheidet zwischen menschlicher Wahrnehmung und automatischer Erkennung. Die Erkennungsraten von Variationen im hochfrequenten Bereichen sind ziemlich schlecht. Deswegen beschränkt man sich für robustes Watermarking auf niedrige und mittlere Frequenzbereiche.

Kompressionsverfahren schmeißen häufig weniger relevante Informationen weg, wodurch implizit Watermarking entfernt werden kann, wenn diese nicht tiefgreifend sind.

Erkennung:

Die Entscheidung ob es sich bei einem Objekt um ein mit Watermarking versehenes handelt wird mithilfe von statistischen Testmethoden getroffen, die für eine gegebene Wahrscheinlichkeit entscheiden ob ein Watermark vorhanden ist.

Dynamische vs. statische Einbettungsmethoden:

Bei dynamischen Verfahren wird eine zusätzliche Funktionalität durch geheime Eingaben getriggert und kann nur durch die Ausführung oder im Debugger gefunden werden.

Statische Methoden basieren hingegen auf nicht-funktionaler Codeerweiterung in Form von Umordnung von Anweisungen, Umbenennung von Variablen-/Funktionsnamen, Anpassungen des Kontrollflussgraphen ....

Prinzipiell scheint man die Wahl zu haben: Entweder man veröffentlicht sein Schema, das mit einem private Key Code erzeugt (z.B. kann man diesen verwenden um die Watermark-Stelle im Code zu spezifizieren). Dieses Schema muss dann allerdings wirklich einen Großteil des Codes erzeugen um gewisse Constraints erzeugen zu können. Wird das Schema nicht veröffentlicht, so ist es nicht hinreichend plausibel zu belegen, dass dies ein Schema ist nach dem der Code entstand. Refactoring von Code scheint aber prinzipiell jedes dieser Verfahren zu brechen, solange nur hinreichend viel refactored wird. Was sich bei wirklich intelligenten Krypto-Verfahren durchaus lohnen kann.

Angriffe:

- Brute Force
- Additiv - Hinzufügen eigener Watermarks
- Subtraktiv - Entfernen von Watermarks
- Verzerrend - semantikerhaltende Transformation
- Protokoll - alternativer Schlüssel
- Kollusive - Differenz zwischen mehreren Kopien

10 Funktionale Sprachen 1

Relevanz sicherheitskritischer Systeme:

Funktionale Sprachen eignen sich aufgrund ihrer Typ-, Speichersicherheit und ausschließlich expliziter Seiteneffekte dafür.

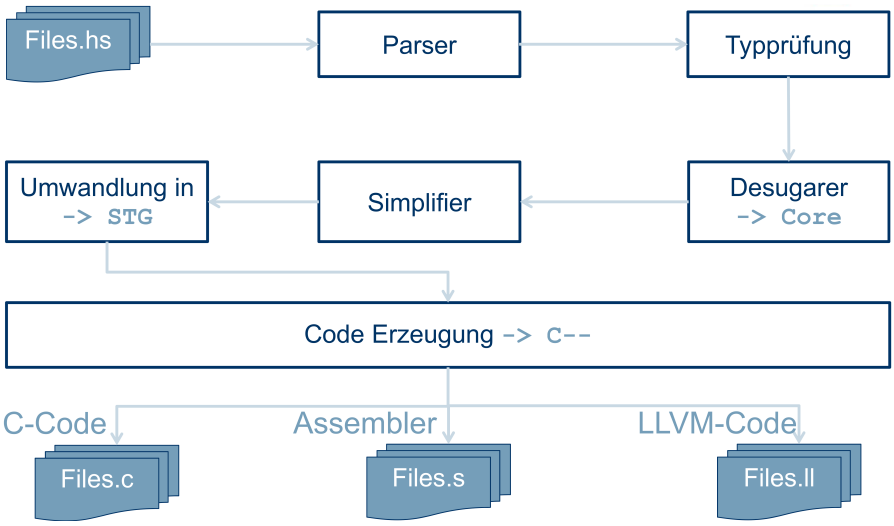

Abbildung 5: Glasgow Haskell Compiler(GHC)
Desugerer erzeugt Zwischencode (Core), Simplifier optimiert darauf und es erfolgt eine Umwandlung in neuen Zwischencode (STG), woraus dann C– Code gebaut werden kann.

Konzept und Eigenschaften von Haskell:

- Alles folgt der mathematischen Funktionsdefinitionen. Für identische Eingaben wird stets das selbe Ergebnis bestimmt.
- Typüberprüfung erfolgt zur Compilezeit, aber es müssen keine expliziten Typen angegeben werden. Der Compiler kann die Typen mittels Typinferenz bestimmen. Die explizite Angabe von Signaturen erleichtert jedoch die Fehlersuche, sollte der Compiler einen Missmatch vorfinden.
- Funktionsargumente können auch lazy ausgewertet werden. Dadurch ist auch arbeiten auf unendlich lange Listen (eingeschränkt) möglich - beim Versuch die Länge zu bestimmen geht es natürlich kaputt.
- Jeder Variable kann nur einmalig ein Wert zugewiesen werden

Spracheigenschaften:

- Tupel können unterschiedliche Datentypen fassen, wohingegen Listen lediglich Elemente gleichen Typs aufnehmen können
- Cons-Operator ":" - Prepending Elements zu einer Liste
- ++-Operator "++" - Verschmelzung zweier Listen
- List Comprehension - [n^2 |n <- [1..20], odd n], wobei "|" Eigenschaften einleitet und "<-" einem $\in$ ähnelt. Das Ergebnis ist eine Liste und keine Menge
- Funktionsdefinitionen mittels "=" - square x = x*x
- Funktionen können mit Argumenten unterversorgt sein, wobei diese unterversorgten Funktionen als Rückgabewert eine Funktion erzeugen, die wiederum selbst eine Funktion ist, die bei ihrer Auswertung weitere Parameter erwartet.
  my_add x y = x+y und add_one = my_add 1
  Hierbei bildet add_one eine neue Funktion, die eine Spezialisierung der my_add darstellt
- Funktionen können ebenfalls überversorgt sein, wobei das Ergebnis dann die restlichen Argumente aufzubrauchen hat
- Am Ende muss die Anzahl Parameter und Argumente übereinstimmen, was einen an Lambda-Ausdrücke erinnern lässt.
- Es gibt *if-then-else* oder für größere Unterscheidungen *case*
- Vergleiche erfolgen häufig über Mustervergliche, wobei die Termseiten übereinstimmen müssen
  f 0 = 1 oder auch fac 0 = 1 mit fac n = n * fac (n-1)
- Typeigenschaften werden durch Typklassen beschrieben:
  - Eq - mit "==" vergleichbare Typ-Klasse
  - Num - numerische Typ-Klasse
  - Show - ausgebbare Typ-Klasse
- Das coole ist: es ist möglich polymorphe Typangaben von Funktionen zu machen, wodurch diese für eine ganze Klasse definiert werden kann. (s.u.)

```
1 fac n = fac_iter 1 n
2         where fac_iter x 0 = x
3               fac_iter x n = fac_iter (x*n) (n-1)
4
5 -- oder auch
6 fac n = let fac_iter x 0 = x
7             fac_iter x n = fac_iter (x*n) (n-1)
8         in fac_iter 1 n
```

Listing 9: Fakultät in Haskell

Beschreibung von Funktions-Typen:

my_add: $(Num\ \alpha) => \alpha -> (\alpha -> \alpha)$ heißt *my_add* ist eine Funktion, die ein Argument vom Typ α erwartet und eine Funktion zurück liefert, die für sich wiederum ein Argument vom Typ α erwartet und ein α zurückgibt.

Lambda-Kalkühl und Currying:

Findet sich auf 10-22ff. Mithilfe von Currying ist es möglich Funktionen, die mehr als einen Parameter erwarten auf mehrere Funktionen abzubilden, die jeweils für sich nur einen Parameter entgegen nehmen, wodurch man im Compiler die vereinfachte Annahme treffen kann: Funktionen haben nie mehr als einen Parameter.

Typinferenz für polymorphe Typen unter Einsatz von Currying:

Ein Beispiel für die Anwendung des Aglorithmus in algorithm 3, bei dem die Unifikation der Typen vorgenommen werden ist 10-39ff zu sehen, wobei der linke @-Knoten eigentlich aus zwei Knoten bestehen müsste, aber das hätte nicht mehr auf die Folien gepasst.

Hier wird deutlich weshalb die Fehlermeldungen vom Compiler bei der Typprüfung zur Compilezeit nicht zwingend sonderlich hilfreich sind. Es ist möglich, dass man lange sucht bis man die falsche Stelle findet. Das Angeben von Funktionssignaturen hilft u.U.. Die Komplexität ist auch echt aufwendig.

Durch die Einführung neuer Sprachelemente (Seite 45) wird dieses Verfahren gebrochen.

Algorithm 3 Typinferenz für polymorphe Typen - Beispiel s.u.

while true do
Laufe bottom-up durch den AST
Betrachte in dem aktuellen Knoten n
if n ist eine Konstante **then**
Typ der Konstanten ist bekannt und Zuordnung zu einer Typklasse ist möglich
else if n ist eine Variable **then**
if n ist noch nicht typisiert **then**
erzeuge neue Typvariable T und ein Tupel (n, T), wobei n mit dem Typen T assoziiert wird. Hierbei entspricht (x, T) einem Eintrag der Definitions-Tabelle
end if
else if n ist eine Funktionsanwendung $fe1 \dots en$ **then**
if Typ der Funktion ist bekannt **then**
Unifiziere den Typ der Argumente mit dem der Funktionsparameter. Das Unifikations-Ergebnis bestimmt den Typ
else if Funktion ist noch nicht bekannt **then**
Behandlung der Funktion wie eine Variable und versuche Typbestimmung soweit möglich
end if
else if n ist ein if-Ausdruck **then**
Der Bedingungs-Typ muss zu Bool unifizierbar sein
else if n ist ein let-Ausdruck **then**
Durch Unifikation der linken und rechten Seite ergibt sich der Typ der neugebundenen Variable
else if n ist eine Funktionsdefinition **then**
Funktionstyp lässt sich durch Unifikation der formalen Parameter und des Funktionsrumpfes bestimmen
end if
end while

Funktionaler Kern:

- Ist eine Erweiterung des Lambda-Kalkühls, um an Effizienz zu gewinnen und die Verständlichkeit zu erhöhen.
- Deswegen werden grundlegende und benutzerdefinierte Datentypen eingeführt, wodurch Konsistenzprüfungen im Compiler möglich werden. Dadurch können beispielsweise Funktionen mit optimiertem Assemblercode verwendet werden, wenn man beispielsweise weiß: Diese Funktion wird nur mit Integern aufgerufen.
- Lazy Evaluation, wodurch nicht alle Argumente ausgewertet werden müssen und sich die Laufzeit verbessern kann.
- if-then-else-Ausdrücke (case), wodurch sich die Lazy Evaluation auch auf Fallunterscheidungen anwenden lässt, da die Erkennung solcher Stellen vereinfacht wird
- Lokale Bindungen mittels *let*, verhindert die Mehrfachauswertung von Termen
- Datentyp-Konstruktoren vereinfachen die Datentyp-Verwaltung im Compiler, denn diese Informationen werden auf der Halde benötigt
- Arithmetische Operationen, denn Rechnen im Lambda-Kalkühl ist nicht sinnvoll lesbar.

Auswertung des funktionalen Kerns durch Graphreduktion, Redix - Beispiel 10-74ff:

Ziel ist die Übersetzung in ausführbaren Maschinencode. Es werden schrittweie sämtliche @-Knoten reduziert, bis solche nicht mehr vorliegen. Hierbei bezeichnet *Redix* einen Teilgraphen, der sich vereinfachen lässt. Man hält sich bei der Suche nach @-Knoten stets in linken Teilbäumen auf. Zu beachten gilt es hierbei etwaige lazy Evaluation nicht zu verletzen.

Implementierung: Jede Knotenklasse besitzt eine Methode, die die Reduktion vornimmt.

11 Funktionale Sprachen 2

Closure - Abschluss - Kontextsicherung:

Ein Closure ist ein Ausdruck oder eine Funktion, die ihren eigenen Kontext sichert und wiederherstellt. Dies ist insbesondere dann nützlich, wenn Elementen vom Stack verschwinden, weil der darumliegende Stackframe entfernt wird, die Elemente aber noch zu verwenden sind. Mithilfe der Closures werden die Elemente auf dem Heap gesichert.

Weiterhin kann man dies verwenden um die Argumente für Über-/Unterversorgung von Funktionen zu verwalten.

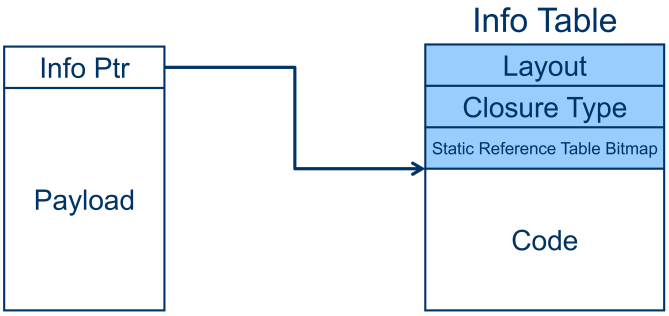

Abbildung 6: Closures im GHC - werden u.A. für Constructors verwenden

In dem Register des Code-Abschnitts steht die Constructor-Nummer, anhand der festgestellt werden kann welcher Typ das ist und dies kann für Pattern-Matching verwendet werden. Beispielsweise ist der letzte Typ bei Listen *NIL* und hat einen anderen Constructor. Hierzu sollte man sich die Beispiele der Vorlesungsfolien ansehen.
Im GHC liegen Closures immer auf dem Stack

Push/Enter von $f\ x\ y$:

Push/Enter kann verwendet werden um das Problem der unbekannten Parameteranzahl her zu werden, weil das Unterversorgen von Funktionen nicht unterschieden werden kann von einem normalen Aufruf. *Push* legt die Argumente $(x,\ y)$ auf den Stack, *Enter* führt f aus.

Eval/Apply von $f\ x\ y$:

Evaluate springt f an und wertet das Ergebnis aus hierbei prüft es ob das Ergebnis dieser Auswertung eine Funktion ist. Der Aufrufer legt x auf den Stack. *Apply* führt f mit dem neuen Stack aus. Dank dieses Verfahrens ist es möglich unbekannte Funktionen anhand der übermittelten Parameter auszuführen.

Das hier soll im GHC wohl leichter implementierbar sein als Push/Enter und ähnliche Performanz liefern.

Umsetzung der Lazy-Evaluation:

Variablen, die nicht nicht bekannte Werte enthalten, werden entfernt und durch null-stellige Funktionen ersetzt, die beim ersten Aufruf den zugehörigen Wert erzeugen und puffern.

Thunk - 11-39:

Erweiterte Abschlüsse für partiell ausgewertete Funktionen. Diese stellen eine Kopie von globalen Variablen bereit und können, anders als Funktionen, aktualisiert werden

Beispiel Twice Square - 11-44ff:

Hier wird recht detailliert dargestellt wie der Stack und der Heap sich bei Ausführung verhält.

CL auf dem Stack soll signalisieren dieses Closure zeigt auf den Heap, das *upd_closure* ist eine Zusammenfassung, die normalerweise aus zwei Befehlen besteht, die zusammengestampft wurden, um das auf die Folien packen zu können

unoptimiert wird alles in Closures gepackt, auch Zahlen. Das ist natürlich erstmal crap, aber dafür sind keine Sonderfälle notwendig. Das auspacken scheint günstiger zu sein als die vielen Varianten zu erzeugen. Die Lazy-Evaluation scheint das auspacken hinreichend verzögern zu können. Da muss man am Ende wohl noch eine Optimierung für vornehmen (Escape irgendwas) Der Code der mit Optimierungen erzegut wird: Funktion mit eingepackten Werten entpackt und ruft am Ende die Funktion auf, die ausgepackte Werte erwartet (siehe fac-Beispiel Seite 84) Auf jedenfall erleichtert die Angabe der Funktionssignaturen das. Die Analyse kann das anscheinend nicht, also es wird nicht für jeden Typen eine eigene Funktion erzeugt

11-56 - das square quadriert und verdoppelt in dem Fall die Funktion execute

11-57 - multiplication - Parameter ist execute (eine closure) $\Rightarrow$ muss erst aktualisiert werden

Optimierungen - Simplifier:

Sämtliche Optimierungen finden im Simplifier statt.

Cache Optimierung in Haskell gibt es vielleicht nicht, was man aber machen kann ist den Haskell Code nach C o.ä. zu transferieren und dort dann Cacheoptimierungen vorzunehmen Auch Branch-Prediktion wird hart gefordert durch viele indirekte Sprünge, da muss sehr aggressives Inlining stattfinden.

Haskell-Code ist automatisch SSA-Konform, was Optimierungen direkt möglich werden lässt.

Auspacken (Unboxing) - Normalerweise werden sowohl die Argumente, als auch

die Ergebnisse von Funktionsaufrufen in Closures gepackt. Idee ist es nun soweit möglich auf entpackten Werten zu arbeiten. Bei Lazy Evaluation darf dies nur dann erfolgen wenn garantiert werden kann, dass der entsprechende Wert immer berechnet wird. Dieses Unboxing beinhaltet implizit des vorherige Berechnen dieser Werte bevor der Wert ausgepackt werden kann. - Beispiel listing 10

Umwandlung der Endrekursionen in Schleifen.

Zum Aufrufer einer Endrekursion braucht man nicht zurückkehren, da dort lediglich das Ergebnis entgegen genommen wird und Cleanup vorgenommen wird. Stattdessen kann man den Activation Record (Stack Frame) des Aufrufers durch den des Aufgerufenen ersetzen.

Auch nicht endrekursive Funktionen können in Schleifen umgewandelt werden, wenn die eine Operation, die die Funktion um ihre Endrekursivität beraubt assoziativ ist. Ich verstehe die Einschränkung an der Stelle nicht ganz. Klar ist das dann möglich, aber im Grunde ist das immer möglich, nur womöglich nicht so trivial. Vermutlich zielt die Aussage der Folien darauf, dass Instruktionen einsparbar sind, ohne dafür Mehraufwand zu erhalten.

Speziell in Haskell ist wohl die Striktheit der Argumente notwendig um Endrekursionen eliminieren zu können. - 11-100f

Inlining von Funktionsaufrufen kann zum Name-Clash führen, die mit α-Konversionen zu beheben sind. Mehfachauswertung von Funktionsaufrufen ist unproblematisch, da das aufgrund der SSA-Eigenschaft wegoptimiert werden kann.

Entfernung der Berechnungen für einen Wert einer toten Variable ist nur dann erlaubt, wenn es sich um ein nicht-strikte Sprache handelt.

Einige Werte die in Closures gehalten werden, können auch auf dem Stack gehalten werden. Denn die Aufbewahrung in Closures ist nur dann notwendig, wenn die Variable länger lebt als der zugehörige Stackframe. Diese Überprüfung nimmt man anhand einer Escape-Analyse vor, die mittels abstrakter Programminterpretation die Lösung approximiert. Es wird jeder Ausdruck mit einer abstrakten Funktion ausgewertet, die der Compiler testweise ausführt. s.u.

```
1 -- ursprünglicher Funktionsaufruf, bei dem x, y, und das
     Ergebnis in Closures stecken
2 f x y = x + y
3
4 -- Hier wird nur noch der Rückgabewert in ein Closure gepackt
5 f x y = case x of
6         Int x# -> case y of
7                   Int y# -> case (x# +# y+)
8                             t# -> Int t#
```

Listing 10: Unboxing von Closures, das # signalisiert entpackte Typen

strikte Funktion f:

Funktion $f(x_1, \ldots, x_i, \ldots, x_n)$ heißt *strikt in* $x_i \Leftrightarrow$ (Auswertung eines Arguments a terminiert nicht $\Rightarrow$ Auswertung von $f(b_1, \ldots, b_{i-1}, a, b_{i+1}, \ldots, b_n)$ terminiert nicht)

Weiterhin ist eine Funktion f *strikt*, wenn sie in allen Argumenten x_i strikt ist. Also x_i immer benötigt wird um den zugehörigen Funktionswert zu bestimmen. Weshalb Call-by-Value für Argumente dann möglich wird, wenn die Striktheit für dieses Argument gegeben ist.

Eine exakte Striktheitsanalyse ist nicht möglich. Daher nimmt man das Fehlen der strikten Eigenschaft an, sofern sich die Striktheit nicht beweisen lässt.

In der Praxis ermittelt man dies mit Bitvektoren, die ermittelt werden, indem für jede Funktion jede mögliche Eingabe die Striktheit betrachtet wird. In dem gezeigten Fixpunkt-Algorithmus gilt die striktheit bei Terminierung für ein Argument x_i.

Hilfsfunktionen Striktheitanalyse:

Die folgenden Hilfsfunktionen dienen der Bestimmung ob etwas Strikt sein kann, NICHT muss. Warum hier die may und nicht must-strikt betrachtet wird, keine Ahnung. Weil ja die ës muss strikt sein, sonst nehmen wir das Gegenteil an"notwendig ist, um den Algorithmus anwenden zu können.

1. $M(7, \sigma) = true$
2. $M(x, \sigma) = x \in \sigma$
3. $M(E_1 \circ E_2, \sigma) = M(E_1, \sigma) \wedge M(E_2, \sigma)$
4. $M(E_1?E_2 : E_3, \sigma) = M(E_1, \sigma) \wedge (M(E_2, \sigma) \vee M(E_3, \sigma))$
5. $M(f(E_1, \ldots, E_n), \sigma) = (f, (M(E_1, \sigma), \ldots, M(E_n, \sigma))) \in H$
   Diese Regel überträgt Wissen über Striktheit anderer Funktionen auf die eigene

Beispiel Anwendung der Hilfsunktionen der Striktheitshanalyse für f:

$fxy = x + y$, σ ist die Menge der Argumente, für die gerade der Bitvektor eine Überprüfung anordnet. Für diese Bitvektoren werden sämtliche Möglichkeiten durchprobiert. $M(B, \sigma)$ sind die angewendeten Hilfsfunktionen zur Striktheitsanalyse.

- $b = (0,0), H = \{\} \Rightarrow \sigma = \{\}$
- $b = (0,1), H = \{\} \Rightarrow \sigma = \{y\}, M(B, \sigma) = false$ (Regel 3)
- $b = (1,0), H = \{\} \Rightarrow \sigma = \{x\}, M(B, \sigma) = false$ (Regel 3)
- $b = (1,1), H = \{\} \Rightarrow \sigma = \{x, y\}, M(B, \sigma) = true$ (Regel 3)
- $\Rightarrow H = \{(f, (1,1))\} \Rightarrow (f, (0,1)), (f, (1,0)) \notin H$

Beispiel 2:

$f\ x\ y = if\ x < 0\ then\ y\ else\ x$

- $b = (0,0), H = \{\} \Rightarrow \sigma = \{\}$
- $b = (0,1), H = \{\} \Rightarrow \sigma = \{y\}, M(B,\sigma) = false$ (Regel 4)
- $b = (1,0), H = \{\} \Rightarrow \sigma = \{x\}, M(B,\sigma) = true$ (Regel 4)
- $b = (1,1), H = \{\} \Rightarrow \sigma = \{x,y\}, M(B,\sigma) = true$ (Regel 4)
- $\Rightarrow H = \{(f,(1,0)),(f,(1,1))\} \Rightarrow (f,(0,1)) \notin H$

Regeln zur Escape-Analyse - Beispiel 11-123ff:

Die Entscheidung ob Argument i von f flüchtet wird bestimmt durch

$$L_i(f,e_1,\ldots,e_n,\varepsilon) = (E[\![fx_1\ldots x_n]\!]\varepsilon[x_j := y_j])_{(1)}, \quad y_j = \begin{cases}(false,(E[\![e_j]\!]\varepsilon)_{(2)}), & j \neq i\\ (true,(E[\![e_j]\!]\varepsilon)_{(2)}), & sonst\end{cases}$$

1. $E[\![c]\!]\varepsilon = (false,\bot)$ - Konstanten können nicht flüchten
2. $E[\![x]\!]\varepsilon = \varepsilon[\![x]\!]$ - x wird in Umgebung ε ausgewertet
3. $E[\![e_1 \circ e_2]\!]\varepsilon = (false,\bot)$ - Ausdrucks-Auswertung bleibt nicht-flüchtig
4. $E[\![e_1e_2]\!]\varepsilon = \underbrace{(E[\![e_1]\!]\varepsilon)_{(2)}}_{\text{zweiter Teil des Tupels}(b,c)} \quad (E[\![e_2]\!]\varepsilon)$
5. $E[\![e_1?e_2 : e_3]\!]\varepsilon = (E[\![e_2]\!]) \cup (E[\![e_3]\!]\varepsilon)$
6. $E[\![\lambda x.e]\!]\varepsilon = (v, \lambda y.E[\![e]\!]\varepsilon[x := y]), v = v_{z\in f}(\varepsilon[\![z]\!])_{(1)}$ und F die Menge der freien Variablen in $\lambda x.e$
7. $E[\![letx_1 = e_1 \ldots x_n = e_nine]\!]\varepsilon = E[\![e]\!]\varepsilon', \varepsilon' = \varepsilon[x_1 := (E[\![e_1]\!]\varepsilon),\ldots,x_n := (E[\![e_n]\!]\varepsilon)]$

12 LLVM

Mögliche Schnittstelle zwischen Frontend und Optimierer:

- Übersetzung nach C - und dann auf dem C-Code optimieren erschwert unter anderem die Möglichkeit Fehler einer Stelle des ursprünglichen Source-Codes zuzuordnen, da die Fehlermeldungen Informationen über den C-Code melden, nicht jedoch über den ursprünglichen Source-Code. Auch wird dies knifflig sobald Konzepte vorliegen, die von C nicht unterstützt werden.
- GCC als Compiler Collection - Die erzeugte Zwischensprache ist nicht unabhängig von den Frontend-Datenstrukturen, wodurch im Grunde keine Verwendung als Bibliothek möglich ist
- Virtuelle Maschinen von Java und .NET - Der Bytecode ist nah an den Hochsprache-Konzepten wodurch Unflexibilität entsteht

Bestandteile des LLVM:

- Assemblerartige Sprache - LLVM IR
- Bibliothekssammlung
- Werkzeugsammlung
- C++-Standardbibliothek
- ...

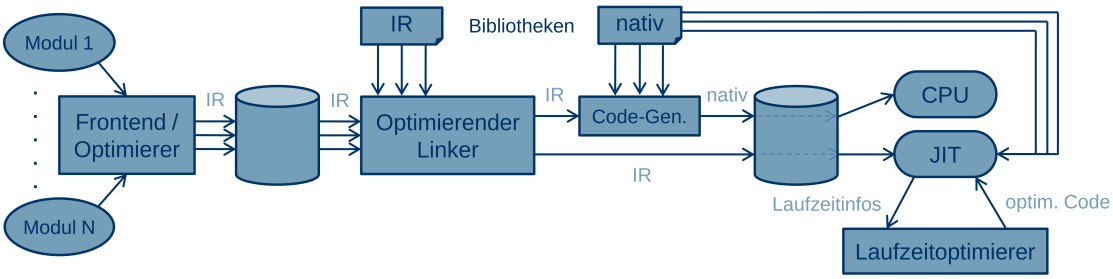

Abbildung 7: LLVM IR dient als Schnittstelle zwischen den Schritten und gehorcht wohldefinierter Syntax u. Semantik. Auf dieser Zwischensprache werden Optimierungen sowohl zur Linkzeit als auch zur Laufzeit möglich.

Modularität des LLVM:

Sämtliche Komponenten sind wiederverwendbar und großteils unabhängig von einander, so dass lediglich diese Module einzubinden sind, dessen Funktionalität man wünscht.

LLVM kann Code in drei verschiedenen, gleichmächtigen Repräsentationen darstellen:

- Textformat (.ll) - für den Menschen lesbar und auch schreibbar. In diesem Format können die Frontends für Sprachen geschrieben werden, ohne dabei auf eine LLVM-API-Schnittstelle zurückzugreifen. Vorteil hierbei ist die deutlich seltenere Repräsentation des Textes im Vergleich zur API.
- Binärformat (.bc) - Bitdarstellung
- In-Memory-Repräsentation - wird im Speicher gehalten in irgendeiner Darstellung

LLVM IR - Eigenschaften, Design:

Ist sowohl für den Menschen, als auch für die Maschine leicht zu schreiben und zu lesen und bietet eine Zwischenstufe, die unabhängig von den Sprachen beider Richtungen ist. D.h. die Quellsprache, als auch die Zielsprache nimmt keinerlei Einfluss.

Auch besitzt LLVM IR ein eigenes Typsystem mit expliziten Typangaben und sehr primitiven Datentypen. D.h. keine Klassen o.ä, um Unabhängig von Hochsprachenkonzepten sein zu können.

Um jedoch die Eigenschaften schwach typisierter Sprachen, wie C nachzubilden werden beliebige Typkonvertierungen erlaubt, erfordern hierbei jedoch explizite Cast-Instruktionen, wodurch cast-lose Programme implizit typsicher sind.

Jeder Grundblock erhält ein explizites Grundblockende, wodurch ein impliziter *Fall-through* unterbunden wird. Dadurch werden Optimierungen aufgrund der Verschiebung einzelner Grundblöcke möglich. Um diese Gegebenheit zu erfüllen erhalten bedingte Sprünge stets zwei Sprungziele, vor jeder Sprungmarke haben Sprunginstruktionen zu stehen und nach jeder Sprunginstruktion haben Sprungmarken zu stehen.

Eine ungemein praktische Eigenschaft ist die stetige Erfülltheit der SSA-Form, da Optimierungen dadurch möglich werden, ohne zuvor die SSA-Form bestimmen zu müssen.

- Vorteile
  - Dadurch werden beliebige Quellsprachen unterstützt.
  - Es ist möglich Programme aus verschiedenen Quellsprachen heraus zu binden. Bzw dieses Vorgehen wird vereinfacht, da Calling Conventions der verschiedenen Programmiersprachen nicht bei dem Binden beachtet werden müssen, sondern dies beim Übersetzen nach IR wegabstrahiert wird.
- Nachteile
  - Abbildung von Features der Hochsprachen müssen abgebildet werden

– Optimierungen, die Wissen über spezifische Eigenschaften der Hochsprache erfordern, machen es notwendig diese Optimierungen bereits in dem Frontend vorzunehmen (selber schreiben)

LLVM IR Namen und Typeigenschaften:

- globale Namen (beginnen mit @) beinhalten die Adresse der globalen Variable, der Inhalt der Adresse muss explizit mit einer Funktion geladen werden
- Lokale Namen beginnen mit %
- Deklarationen werden mittels *declare* eingeleitet
- Definitionen hingegen mittels *define*
- Aufrufe *call*
- Rücksprünge *ret*
- Sämtliche Funktionen sind frei von Overloading, auch Namespaces werden nicht unterstütz. D.h. diese Auflösung muss im Frontend vorgenommen werden und die unterschiedlichen Funktionen gleichen Namens sind umzubenennen.
- Vererbung erfordert die Verwendung von reinterpret cast, damit in Oberklassen gecastet werden kann
- Sämtliche Variablen sind vor der Verwendung in ein Register zu laden und anschließend wieder zu sichern
- Weitere Syntax findet sich in dem Foliensatz 12-19 - 12-30
- Debuginformationen sind explizit hinzuzufügen, um Breakpoints setzen zu können und Compiler-Fehlermeldungen sinnvoll werden zu lassen. 12-63

SSA-Form für LLVM IR - *mem2reg* - Beispiel 12-33f:

Die SSA-Form kann man erzeugen, indem man sämtliche Variablen auf den Stack legt wofür mit einer Funktion Speicher auf dem Stack angefordert wird, ja Stack - nicht Heap. Sämtliche Zugriffe ob lesend oder schreibend erfordern dann explizite Befehle. Die Auflösung des Mappings auf virtuelle Register etc. und Überführung in die SSA-Form macht das Tool *mem2reg* für einen, wobei auch ungemein viele der unnützen Lade- und Schreibinstruktionen entfernt werden. Dies reduziert den Übersetzungsaufwand nach LLVM IR erheblich.

Der Code der da selbst bei naiver Übersetzung rausfällt ist häufig gar nicht so schlecht, da die Registerzuteilung ziemlich gut optimiert wird.

Weitere nützliche LLVM-Tools:

- *llvm-as* - Ein Assemblierer, der die Übersetzung vom Text- in das Binärformat vornimmt
- *llvm-dis* - Ein Dissassemblierer, der die Übersetzung vom Binär- in das Textformat vornimmt
- *opt* - Unglaublich praktisches Tool, das
  - den *domtree* erzeugen
  - die Dominatoren dotten
  - das Mapping der Abbildung von Stackpositionen auf Register darstellen kann
  - Dead-Code-Eliminierung und Inlining vornehmen kann
  - Verschied aggressive Optimierungen vornehmen
  - ...
  - und sogar eigene Passes (Läufe) entgegennehmen und auf dem Code ausführen kann.
- *llvm-link* - Linker für die Module in LLVM IR, funktioniert z.Z. allerdings noch nur auf einen Rutsch und nicht wie in C modulweise. Dafür gibt es allerdings Ansätze zur Umsetzung: *Thin LTO*
- *llc* - Statischer Übersetzer für LLVM IR - Erzeugt Assembler oder Object-Files
- *lli* - Interpreter/JIT-Compiler für LLVM IR, Das Programm wird ohne initiale Übersetzung ausgeführt. Eingelesen werden können die Programme sowohl Text- als auch im Binärformat

13 Abbildungsverzeichnis

Abbildungsverzeichnis

1 Reales Speicherlayout print 5
2 FKFG 12
3 Ablaufspuren 38
4 Beispiel Happened Before 47
5 GHC 51
6 Closures im GHC 56
7 LLVM IR zur Optimierung 61

14 Listingverzeichnis

List of Listings

1 Assembler main-print 4
2 Assembler print 4
3 Beispiel stack cutting 9
4 advice-problem Threads 14
5 Lösung advice-problem Threads 15
6 Bereinigung der Null-Tabelle 20
7 Entfernung zyklischen Mülls 20
8 Concurrent-Clang 44
9 Fakultät in Haskell 52
10 Unboxing von Closures 58