Speicherbereinigung in rein funktionalen Sprachen

Christoph Müllner[1] und Harald Steinlechner[2]

[1] e0525067@student.tuwien.ac.at
[2] e0825851@student.tuwien.ac.at

Zusammenfassung. Funktionale Sprachen stellen hohe Anforderungen an die Speicherverwaltung. Durch Einsatz von unveränderbaren Datenstrukturen werden Objekte häufig kopiert, neu angelegt. Weiters erzeugen Programme oft sehr viele temporäre Objekte. Gerade aber *Unveränderbarkeit* scheint positive Auswirkungen auf mögliche Optimierungen für automatische Speicherbereinigung zu haben. Es stellt sich heraus, dass die Auswertungsstrategie von rein funktionalen Sprachen intern sehr wohl Datenstrukturen modifiziert. In diesem Artikel behandeln wir Speicherbereinigung in Haskell, einer nicht strikten rein funktionalen Programmiersprache. Fokus liegt dabei auf die parallele Garbage Collection und Optimierungen für funktionale Sprachen.

1 Einführung

Funktionale Programmierung bevorzugt unveränderbare Datenstrukturen gegenüber destruktiven Zuweisungen und Modifikationen. Rein funktionale Sprachen wie Miranda, Haskell, Clean lassen im Gegensatz zu ML, Lisp,.. grundsätzlich keine Seiteneffekte zu. Variable sind also Unveränderbar - ist erstmal ein Wert zu gewiesen, kann dieser nicht nochmal verändert werden. Eine Konsequenz davon ist, dass es keine destruktive Zuweisung gibt.

Grundsätzlich gibt es zwei Auswertungsstrategien für funktionale Programme:

1. Strikte Auswertung (*strict-evaluation*). Hierbei werden die Argumente einer Funktion ausgewertet, bevor die Funktion reduziert wird (engl.. auch *Call-by-value*).
2. Nicht strikt Auswertung. Hier werden Funktionen reduziert, bevor ihre Argumente ausgewertet wurden. Argumente werden erst dann ausgewertet, wenn sie auch wirklich benötigt werden (engl. auch *Call-by-name*).

Eine Optimierung stellt *Call-by-need* dar. Hier werden Variable höchstens einmal ausgewertet. Ihr Wert wird danach zwischengespeichert (*caching*). Haskell verwendet eine *Lazy evaluation*, eine Realisierung von *call-by-need*.

Haskell wertet also Variable kein, oder genau einmal aus. Unveränderbarkeit scheint Garbage Collection auf den ersten Blick erheblich zu simplifizieren, weil alte Werte per Definition nicht beachtet werden müssen.

Ein Blick auf Implementierungen zeigt allerdings, dass der Schein der Unveränderbarkeit trügt:

Unausgewertete Zellen werden in Haskell Implementierungen auch *Thunks* genannt. Thunks sind demnach nicht, wie Variable in Haskell Unveränderbar, da schon existierende Thunks im nachhinein ausgewertet werden können, falls der Wert des Thunks angefordert wird. Hier spricht man von der *Update-Once-Property*: Thunks werden höchstens genau einmal während der Programmausführung modifiziert - also mit dem Wert des zugrunde liegenden Ausdrucks bestückt.

Haskell Implementierungen scheinen also keinen direkten Nutzen aus *purity* ziehen zu können, gerade aber funktionale Programme stellen andere hohe Anforderungen an Speicherverwaltung.

Folgendes Haskell Programm berechnet Faktorielle unter der Verwendung von Endrekursion.

```
factorial 0 acc = acc
factorial n acc = factorial (n-1) $! (acc*n)
```

Hier ist zu bemerken: Um die Funktion auch wirklich Endrekursiv auszuführen ist es notwendig das Akkumulatorargument *string* (engl. *strict*) auszuwerten. Dadurch wird die Auswertung in jedem Schritt erzwungen - das heißt pro Rekursionsschritt auch wirklich ein Wert erzeugt, welcher in der jeweilig nächsten Iteration unbenutzt wird. Hier werden also zur Ausführung viele kurzlebige Objekte erzeugt.

Konträr dazu erzeugt das nächste Beispiel laufend Objekte (Cons-Zellen), welche potentiell später noch benötigt werden.

```
upto i n | i<=n      = i : upto (i+1) n
         | otherwise = []
```

Die bisherigen Beispiele verleiten dazu, zu Werte in Haskell als azyklisch anzusehen. Dieses Faktum würde Referenzzähler als äußerst attraktiven Implementierungsmechanismus erscheinen lassen - abgesehen von den Vorteilen, die ein Kopieralgorihmen (Speichereffizienz bzw. Reduktion der Fragmentierung). Rekursive Werte sind allerdings wie man sich sehr schnell überzeugen kann *sehr* häufig.

```
recVal = 1 : recVal
```

Moderne Haskell-Übersetzer überzeugen allerdings durch hohe Effizient, woran auch die Garbage Kollektoren einen hohen Beitrag leisten. In diesem Artikel werden wir die Vorteile funktionaler Sprachen hinsichtlich automatischer Speicherbereinigung untersuchen.

1.1 Generationelle Speicherbereinigung

Generationelle Speicherbereinigung [1] ist eine Technik, welche die folgenden Eigenschaften nutzt um die Speicherbereinigung mittels Kopieralorithmen zu optimieren.

- Die Dauer eines Speicherbereinigungsdurchlaufes ist abhängig von der Menge an zu kopierenden (lebenden) Objekte im Speicher.
- Die meisten Objekte haben eine kurze Lebensdauer. 80 % bis 98% aller Objekte ([2]) haben eine Laufzeit von ein paar wenigen Millionen Instruktionen.
- Objekte die einen Speicherbereinigungslauf überstehen, haben einen hohe Wahrscheinlichkeit mehrere Läufe zu überstehen.

Bei der generationelle Speicherbereinigung werden Objekte in $n + 1$ Generationen gruppiert. Neu allozierte Objekte sind Teil der Generation 0. Objekte, welche eine bestimmte Anzahl an Speicherbereinigungsdurchläufen überleben, kommen in die nächst höhere Generation. Wir nun eine Generation von Objekten gereinigt, werden alle niedrigeren Generationen ebenfalls gereinigt. Höhere Generationen werden prinzipiell seltener gereinigt, da deren Objekte einen hohe Wahrscheinlichkeit mehrere Läufe zu überstehen.

Die Speicherbereinigung läuft formal in zwei Schritten ab: evakuieren (*evacuate*) und reinigen (*scavenge*). Im ersten Schritt werden Zeiger evakuiert. Dabei werden referenzierte Objekte an eine neue Speicherstelle (*to-space*) kopiert. An der alten Speicherstelle (*from-space*) wird ein Vorwärtszeiger zur neuen Adresse hinterlassen. Im zweiten Schritt werden die kopieren (lebenden) Objekte gereinigt. Dabei werden die Objekte nach Zeigern durchsucht, welche dann entweder aktualisiert (Zeiger auf bereits kopiertes Objekt) oder evakuiert werden.

Ein wichtiger Begriff im Kontext der generationellen Speicherbereinigung ist die gemerkte Menge (*remembered set*). Diese Menge enthält alle Referenzen von älteren Objekten zu neueren Objekten. Diese Menge ist deshalb von Bedeutung, da bei einem Speicherbereinigungsdurchlauf nicht unbedingt alle Generationen bereinigt werden. Damit junge Objekte dennoch als lebend erkannt werden, wenn sie lediglich von älteren Objekten referenziert werden, ist diese Menge notwendig.

Generationelle Speicherbereinigung wurde bereits 1993 im Glasgow Haskell Compiler verwendet. In [3] wird der Vorteil dieser Speicherbereinigungstechnik für den GHC genauer analysiert.

2 Implementierung und Analyse der aktuellen Realisierung im Glasgow Haskell Compiler

In diesem Kapitel wird näher auf die Implementierung des Speicherbereinigungsverfahrens im Glasgow Haskell Compiler eingegangen. Es werden grundlegende Techniken und spezielle Optimierungen beschrieben.

2.1 Blockstruktur des Speichers

Eine wichtige Technik, welche bei der Implementierung des Speicherbereinigungsverfahrens im Glasgow Haskell Compiler verwendet wird, ist die blockweise Strukturierung des Speichers (*block-structured heap*). Diese Technik ist bereits in [4] und älteren Publikationen beschrieben.

Die Idee ist, dass der gesamte Speicher in Blöcke mit fixer Größe aufgeteilt. Mehrere Blöcke werden zu einem Megablock zusammengefasst. Ein Megablock enthält zusätzlich noch einen Blockdeskriptor für jeden Block, dessen Speicheranwendung leicht berechnet werden kann. Abbildung 1 zeigt den Aufbau eines Megablocks.

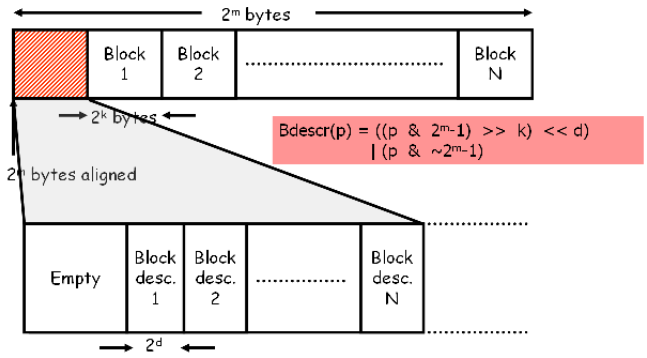

Abb. 1: Blockstruktur des Speichers: Megablock und dazugehörige Blöcke und Deskriptoren

Durch die Blockstruktur ist es möglich Speicher flexibel zu verwalten. So können mehrere Blöcke zu einem Bereich (*area*) zusammengefasst werden.

Objekte liegen aneinander gereiht in den Blöcken. Das genaue Speicherlayout eines Objekts ist in Abbildung 2 dargestellt. Das erste Wort eines Objekts ist ein Zeiger zur Informationstabelle (info table), welcher unter anderem die Typinformation enthält. Zeiger auf Objekte zeigen immer auf das erste Wort des referenzierten Objekts.

Objekte, welche größer als ein Block sind, werden in einer Blockgruppe (mehrere aufeinander folgende Blöcke) alloziert. Der eventuell frei bleibende Speicher am Ende des letzten Blocks wird nicht verwendet. Dies hat den Vorteil, dass diese Objekte während der Speicherbereinigung niemals kopiert werden.

2.2 Parallelisierung der Speicherbereinigung

Im diesem Kapitel behandeln wir den derzeit im Glasgow Haskell Compiler (GHC) eingesetzten Garbage Collector [5].

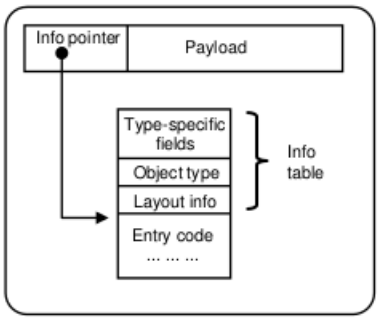

Abb. 2: Speicherlayout eines Objekts im Speicher

Die Verfügbarkeit von Mehrkern-Prozessoren in handelsüblichen Rechnern macht die Parallelisierung der Garbage Collection sehr attraktiv - besonders da man mit hoher Effizienzsteigerung rechnen kann, ohne dass Code des Programmierers angepasst werden muss. Wir behandeln ausschließlich Verfahren welche die Speicherbereinigung parallel ausführen - das Programm wird dazu gestoppt. Es handelt sich also um kein nebenläufiges Verfahren.

Der zugrunde liegende Kern für den Kopierenden Algorithmus kann wie folgt beschrieben werden:

```
while (pending set non-empty) {
      remove an object p from the pending set
      scavenge(p)
      add any newly-evacuated objects to the pending set
}
```

Die Parallelisierung davon - das Abarbeiten des *Pending-Sets* auf mehreren CPUs scheint auf den ersten Blick offensichtlich. Hier liegt die Schwierigkeit allerdings im Detail. Schwierigkeiten ergeben sich durch gemeinsam verwendete Ressourcen (z.B.: *Pending-Set*), bzw. wenn es darum geht Arbeit gleichmäßig auf die Arbeiter-Thronrede aufzuteilen und dabei eine möglichst hohe Auslastung zu erzielen.

Potentiell werden gleiche Objekte von verschiedenen Collector Threads gleichzeitig traversiert. In [5] werden dazu Spinlocks verwendet. Wird ein Objekt traversiert, wird exklusiver Zugriff dafür angefordert - beinhaltet das Objekt schon einen *Forward-Pointer*, wird das Objekt ignoriert - ansonsten in den Zielspeicher *To-Space* kopiert. Diese Implementierung könnte mit optimistischem Locking optimiert werden, Kollisionen entstehen allerdings sehr selten [5].

Zielspeicher sind implizit wiederum eine gemeinsam benutzte Ressource unter Collector-Threads. Praktisch ist Verwendung eines Zielspeichers pro Thread eine effiziente Lösung. Geht einem Collector Thread sein Zielspeicher aus, fordert

er einen neuen Block vom Allokator an.

Die Verwendung einer expliziten Datenstruktur für das *Pending-Set* erzeugt gezwungenermaßen zusätzlichen Synchronisationsaufwand. Cheney [6] verwendet anstatt einer expliziten Datenstruktur den Zielspeicher selbst als *Pending Set*. Dabei werden Grenzen zwischen schon bearbeiteten und ausstehenden Speicher durch Pointer markiert. Marlow et al. verwenden dieses Prinzip. Allerdings stellt sich weiterhin die Frage wie man Arbeit unter den Threads aufteilt. Imai und Tick [7] verwenden dazu eine Menge an Blöcken, von diesen Collektor Threads neue Arbeit abholen. Dies ist besonders praktikabel weil das bisher beschriebene Verfahren bereits auf Blöcken arbeitet.

2.3 Optimierungen des Verfahrens

Referenzen von ältere auf jüngere Objekte ist problematisch, da derartige Referenzen im *Remembered Set* abgespeichert werden müssen. Junge Objekte, welche von alten referenziert werden, können allerdings aufgrund des Konzeptes der Generational Garbage Collection niemals vor den alten Objekten bereinigt werden. Auf den ersten Blick sind *Alt-Neu* Referenzen in rein funktionalen Sprachen allerdings gar nicht möglich - ein altes Objekt müsste mutiert werden um auf ein neueres zu zeigen. Gerade aber die Implementierung von Lazy-evaluation mutiert virtuell Zellen, wie in Abbildung 3 demonstriert wird.

```
let plus5 = (+) val
    ...
    val = 5

in plus5 5
```

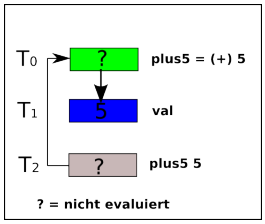

Abb. 3: Zustand bei Auswertung von Thunks und deren Referenzen. Die ausstehende Auswertung (suspended evaluation) von `plus5` verweist durch Auswertung und teilweiser Auswertung auf eine neuere Zelle.

Dieses Modifikation wird in rein funktionalen Sprachen niemals (der Wert nie benötigt) oder genau ein mal durchgeführt. In der Literatur wird diese Eigenschaft als *Update-Once-Property* bezeichnet.

Aus Sicht des Algorithmus ist es korrekt, Zellen welche durch alte Objekte referenziert werden in die Generation der referenzierenden Zelle zu verschieben. Collections niedriger Generation müssen sich dann nicht mit Objekten aufhalten, welche ohnehin nicht wegfallen bereinigt werden können. Diese Optimierung wird von Marlow et al. als *Eager Promotion* bezeichnet. Abbildung 4 veranschaulicht das Prinzip - neuere Objekte werden direkt in die ältere Generation verschoben. Diese Optimierung setzt die bereits angesprochene *Update-Once-Property*

voraus, da sich verändernde Zellen Objekte in ältere Generationen verschieben würde, auch wenn die Referenz später gelöscht oder verändert wird - so würden potentiell viele Objekte in hohe Generationen verschoben und damit unnötig Speicher verbraucht.

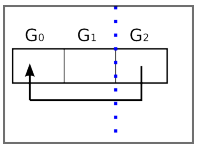

Abb. 4: Darstellung des Heaps bei Alt-Neu Referenzen. Objekte in G_0, welche von einem Objekt in G_2 referenziert werden, können nach G_2 verschoben werden.

2.4 Resultate

Die Autoren hätten sich einen größeren Vorteil bei der Optimierung von rein funktionalen Sprachen erwartet. Die Parallelisierung schlägt sich auf Quad Core Maschinen um Faktor 1.5 bis 3.2 nieder. Dem gegenüber steht ein Synchronisationsoverhead von ca. 20%. Auf Dual Core Maschinen schlägt sich die Parallelisierung auf 20% der Wall Clock Time nieder. Quad Core Prozessoren erreichen einen Speedup von 45%.

Die angesprochene Optimierung *Eager Promotion* trägt dabei einen Anteil von 7% und erscheint unerwartet effektiv.

3 Zusammenfassung

Die Eigenschaften von nicht-strikt ausgewerteten funktionalen Sprachen, wie Haskell, bietet eine Reihe von Herausforderungen als auch Optimierungsmöglichkeiten für die Speicherbereinigung.

Die Implementierung der Speicherbereinigung im GHC setzt dabei auf alt bewährte allgemeine Techniken wie generationelle Speicherbereinigung und Blockstrukturierung des Speichers. Weiters werden spezielle Optimierungen angewandt, welche spezielle Eigenschaften der Sprache nutzen um einen positiven Laufzeiteffekt zu erzielen, wie etwa Eager Promotion.

Durch die Parallelisierung der Speicherbereinigung werden zusätzliche Möglichkeiten moderner Prozessoren ausgenutzt.

Durch die Optimierungen kann auf einem Quad Core Prozessor ein Speedup von 45% erreich werden.

Literatur

1. Ungar, D.: Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In: Proceedings of the first ACM SIGSOFT/SIGPLAN software

engineering symposium on Practical software development environments. SDE 1, New York, NY, USA, ACM (1984) 157–167
2. Wilson, P.R.: Uniprocessor garbage collection techniques. In: Proceedings of the International Workshop on Memory Management. IWMM '92, London, UK, UK, Springer-Verlag (1992) 1–42
3. Sansom, P.M., Peyton Jones, S.L.: Generational garbage collection for haskell. In: Proceedings of the conference on Functional programming languages and computer architecture. FPCA '93, New York, NY, USA, ACM (1993) 106–116
4. Dybvig, R.K., Eby, D., Bruggeman, C.: Dont stop the bibop: Flexible and efficient storage management for dynamically-typed languages. Technical report, Indiana University Computer Science Department (1994)
5. Marlow, S., Harris, T., James, R.P., Peyton Jones, S.: Parallel generational-copying garbage collection with a block-structured heap. In: Proceedings of the 7th international symposium on Memory management. ISMM '08, New York, NY, USA, ACM (2008) 11–20
6. Cheney, C.J.: A nonrecursive list compacting algorithm. Commun. ACM **13**(11) (November 1970) 677–678
7. Imai, A., Tick, E.: Evaluation of parallel copying garbage collection on a shared-memory multiprocessor. IEEE Trans. Parallel Distrib. Syst. **4**(9) (September 1993) 1030–1040