Abstrakte C-Maschine und Stack

— Ausarbeitung —

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von: Julian Tobergte
E-Mail-Adresse: 2tobergt@informatik.uni-hamburg.de
Studiengang: Informatik

Betreuer: Nathanael Hübbe

Hamburg, den 05.07.2013

Inhaltsverzeichnis

1 Motivation / Thematik

In diesem Vortrag wird der Versuch gemacht, einen einigermaßen einfachen Einstieg in das Thema Abstrakte Maschinen allgemein, und speziell zu der Abstrakten C-Maschine zu liefern. Dabei wird zunächst wert darauf wert gelegt, dass der Unterschied einer Abstrakten Maschine zu einem Compiler, beziehungsweise einem Interpreter erkenntlich wird. Der Einstieg geschieht dadurch, dass zunächst die Funktionsweise erklärt wird und an einem Beispiel nachvollzogen wird. Gleichzeitig wird explizit darauf hingewiesen, welche Vorteile der Einsatz einer Abstrakten Maschine konkret für Entwickler und Nutzer beiderseitig bietet. Des weiteren spielt der Stack bei der Betrachtung grundlegenderer Funktionalitäten eine wichtige Rolle in der Hinsicht, als dass beispielsweise von angreifenden Personen ausgenutzte Risiken gerade im Stack vorhanden sind. Insofern ist es essentiell, das Verhalten eines Stacks und seine Arbeitsweise zu kennen, um die Chance zu haben Programme sicherer zu schreiben. Zu diesem Zweck wird der Stack in seiner Grundfunktion erklärt, sowie einige Beispiele zum besseren Verständnis und Veranschaulichung. Die C Language Specification wird ebenfalls ein Bestandteil dieses Vortrags, da dort unter anderem die Spezifikationen für die Abstrakte Maschine enthalten sind.

2 Abstrakte Maschine

2.1 Einleitung

Eine Abstrakte Maschine wird dort benötigt, wo ein Entwickler einer Software in einer Sprache x, diesen Code nicht nur für ein einziges System, vermutlich sein eigenes, kompilieren möchte, um das Programm dann darauf auszuführen, sondern das Programm auch noch verbreiten möchte, beispielsweise um seine eigene Reputation zu steigern und oder diese Software zu verkaufen und zu diesem Zweck ist es von Vorteil wenn das Programm auch auf nicht baugleichen Systemen zum Entwicklungs-system lauffähig ist, sprich möglichst viele Benutzer in der Lage sind die Software zu nutzen.

2.2 Was ist die Abstrakte Maschine

Die Abstrakte Maschine löst genau diese in der Einleitung beschriebene Art von Problem, indem sie eine Zwischenebene bereitstellt, die den Spalt zwischen Quelltext und ausführendem System überbrückt, wie es in der folgenden Grafik abstrakt als eine Art von Brücke dargestellt ist.

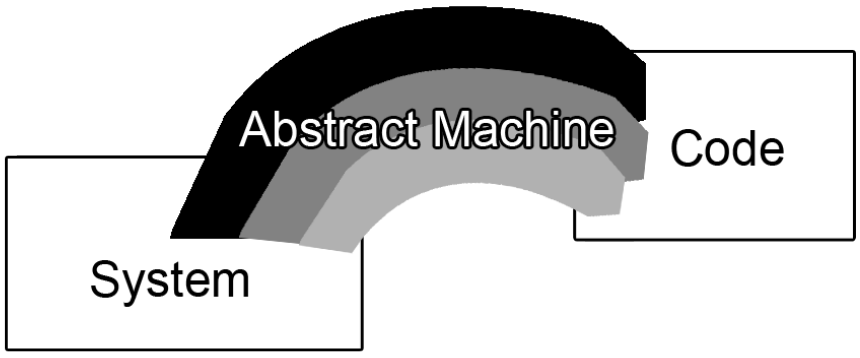

Abbildung 2.1: Abstrakte Darstellung einer Abstrakten Maschine

Diese Brücke ist eine „Abstrakte" Version der Hardware eines jeden Ausführenden Systems und aus Sicht des Quelltexts immer gleich, in der Hinsicht, dass man sie sich

als eine Art Interface vorstellen kann, auf deren angebotene Operationen vom Quelltext zugegriffen wird. Eine konkrete Abstrakte Maschine für ein spezifisches System auf dem der Quellcode ausgeführt werden soll muss dann die Befehle der Sprache, die im Quelltext aufgerufen werden, für dieses spezifische System implementieren, um zu funktionieren.

2.3 Compiler / Interpreter

Ein Interpreter tut genau das, was die Abstrakte Maschine können soll, er versteht den Quellcode und kann ihn für das jeweilige System, das ihn ausführt in Maschinencode übersetzen. Dabei ist der Interpreter gerade die Abstrakte Maschine geworden, denn er ist an keinen spezifischen Code der Sprache gebunden, sondern übersetzt jeden gültigen Code. Dies ist zum Beispiel bei Python der Fall, wobei zu beachten ist dass in der am häufigsten genutzten Implementation CPython der Quellcode erst zwischen-kompiliert wird und dann dem Interpreter übergeben wird, da die Ausführung anderenfalls zu langsam ablaufen würde.

Dies ist der Unterschied zum Compiler, der für einen Quelltext einen Maschinencode ausgibt, der auf ein konkretes System angepasst laufen wird. Das System des Compilers wird beispielsweise von Java unterstützt, denn dort wird der Quelltext zunächst zu Bytecode kompiliert um dann hinterher auf einer Implementation der Abstrakten Maschine, typischerweise die Java-VM, deren Namensgebung schon eindeutig erkennen lässt worum es sich handelt, ausgeführt wird.

Wie man an den beiden genannten Beispielen merkt, trifft häufig der Fall zu, dass in dem Übergang von Quellcode zur Ausführung Compiler und Interpreter hintereinander verwendet werden und so eine Intermediate Language, also eine Art „Zwischensprache" zu Stande kommt.

2.4 Nutzung

Die Nutzung der Abstrakten Maschine beläuft sich darauf, dass bei der Entwicklung einer Sprache am Anfang zunächst eine Abstrakte Maschine spezifiziert werden muss, auf die die Operationen der Sprache angewandt werden sollen. Auf diese Weise kann sich ein Software-Entwickler diese Spezifikation durchlesen und verstehen was die einzelnen Operation konkret auf dem Abstrakten System tun. So kann er seine Software optimieren, ohne sich vorher festlegen zu müssen auf welchem System das Programm am Ende ausgeführt werden wird. Auf der anderen Seite muss sich natürlich die Abstrakte Maschine oder genauer formuliert die jeweilige Implementation der Abstrakten Maschine an diese Sprachspezifikation halten, das heißt sie muss alle Operationen auf dem ausführenden System so ausführen, als würden sie tatsächlich auf der „imaginären" Abstrakten Maschine laufen, oder als ob. Diese ist gerade imaginär, weil sie nur in den Spezifikationen als Modell existiert. Das als ob erklärt sich dadurch, dass man allgemein sagt, solange die „Observable Effects" also die Beobachtbaren Effekte der Ausführung (Speicherbelegungen, usw.) genau so sind, als würde das Programm auf der Abstrakten Maschine laufen, gelten diese als

gleichwertig, das heißt die Implementation erfüllt die Anforderungen und kann verwendet werden. Solange dies eingehalten wird, darf der jeweilige Compiler oder Interpreter tun was er will. Insbesondere ist dadurch eine gewisse Freiheit bei der Entwicklung einer Implementation einer Abstrakten Maschine gegeben, die der Entwickler nutzen darf und soll. Deshalb ist es empfehlenswert in der Sprachspezifikation was diesen Punkt anbelangt nicht allzu spezifisch zu werden um dem Entwickler eine möglichst große Freiheit zu lassen. Die Nutzung eines solchen Sprach-Entwicklungs-System, also der Ansatz eine Abstrakte Maschine zu verwenden bringt immense Vorteile. Wie im Einleitungstext schon angesprochen muss ein Entwickler einer Software nun nichtmehr das Programm für jedes neue System auf dem das Programm ausgeführt werden soll neu schreiben, sondern kann diese Arbeit von einem weiteren Entwickler erledigen lassen, der einmalig für jedes System einen Compiler oder Interpreter schreibt, der jedoch alle Programme der Sprache, vorausgesetzt sie sind gültig, also halten die Anforderungen der Sprache ein, übersetzen kann. Ein weiterer Vorteil dabei ist, dass pro Programm nur ein Quellcode existiert und somit gegenüber mehreren Quellcodes zum gleichen Programm eindeutig leichter wartbar ist, denn logischerweise müsste der Programmierer die Änderungen an jedem der einzelnen Quellcodes vornehmen. Des Weiteren ist es auch vorteilhaft für ein Programm, wenn es in Zukunft, mit neuen Rechnerarchitekturen, immer noch ausführbar ist, ohne dass ein Entwickler daran irgendwelche Änderungen vornehmen müsste, der dazu womöglich auf Grund menschlicher Limitierungen gar nicht mehr in der Lage ist. Stattdessen wird für die neue Architektur einfach ein neuer Compiler/Interpreter geschrieben der Programme der Sprache auf diesem System ausführen kann, folglich auch den alten Code des Programms. Dazu kommt noch der nicht ganz so offensichtliche Vorteil des Gewinns an zusätzlichen Funktionalitäten, die man durch die Hilfe einer Abstrakten Maschine gewinnt. Denn keine heutige Architektur unterstützt Hardware-seitig Kontrollstrukturen wie if-Blöcke oder while-Schleifen, weswegen man ohne eine Abstrakte Maschine den gesamten Kontrollfluss mit z.B. goto- und jump-Befehlen programmieren müsste.

3 Stack

Ein Stack, zu deutsch Stapel, ist eine auf dem LIFO- (Last In First Out) Prinzip basierende Datenstruktur, die ihrem Namen sehr gerecht wird, da man sich das ganze in der Tat als einen Stapel vorstellen kann, bei dem nur das oberste Element runtergenommen werden kann und Elemente können auch nur oben eingefügt werden. Diese Datenstruktur wird häufig von Betriebssystemen und auch von Compilern genutzt, beispielsweise im Falle des Kompilierens bei einer Überprüfung auf Syntax-korrektheit würde für jeden geöffneten Block ein Element hinzugefügt und für jeden geschlossenen Block eines heruntergenommen, sodass man am Ende bei einer eingehaltenen Syntax einen leeren Stack vorliegen haben müsste. Also ganz ähnlich eines Kellerautomaten. Der Stack bietet grundsätzlich 3 Operationen an: push für das Hinzufügen, pop / pull für das Entfernen und oft auch peek um das oberste Element anzusehen, aber nicht zu entfernen. Für die Adressierung von Variablen im Stack gibt es verschiedene Arten von Pointern. Der Base-pointer %ebp zeigt auf den Start des Stack-Frames, also auf den Rand des Stacks. Der Stack-pointer %esp zeigt auf das Ende des Stacks, also auf das oberste Element. Die Adressierung von Variablen geschieht dann relativ zu einem Pointer mit einem Offset, beispielsweise 8(%ebp). Bei der Ausführung von C-Code wird unter x86-Architekturen der Stack intern genau auf diese Weise benutzt. Bei einem Funktionsaufruf wird quasi ein neuer kleiner Stack „oben" auf dem Hauptstack angelegt, indem der alte Base-pointer der pushen zwischengespeichert wird und danach der aktuelle Stack-pointer dem Framepointer zugewiesen wird. Daraus folgt, dass der neue Stack als Beginn den Punkt definiert, an dem der alte Hauptstack geendet hat. Dieses Vorgehen ist sehr gut geeignet für Rekursion, da sich verschachtelte Funktionsaufrufe so Stück für Stück nachvollziehen und auch wieder ordentlich auflösen lassen. Jedoch gilt auch zu beachten dass der zu beginn eines Funktionsaufrufs gepushte alte Base-pointer als Adresse interpretiert wird, die den Ort angibt, wo der ehemalige Stack begonnen hatte. Wenn nun ein Element zu viel oder zu wenig gepusht wird und somit ein verfälschter Stack-pointer zu Stande kommt, wird der Stack willkürlich abhängig vom Inhalt der aktuellen Zelle an irgendeinen Ort im Speicher gelegt, was mit an Sicherheit grenzender Wahrscheinlichkeit zu einem Fehler führen wird, der das ganze Programm in seiner Ausführung behindert. Schlimmer ist jedoch das Szenario, in dem ein Angreifer bewusst solche Fehler begeht um an eine Stelle im Speicher zu springen, an der er schädlichen Code platziert hat, um diesen so ausführen zu lassen.

3.1 Veranschaulichung

Listing 3.1: Beispiel-C-Code

```
1 int max(int x, int y)
2             {
3                   if (x > y)
4                      return x;
5                   else
6                      return y;
7             }
```

In diesem Beispiel geht es um eine Veranschaulichung einer Funktion und den dazugehörigen Operationen, die intern auf den Stack angewandt werden. Offensichtlich handelt es sich im Beispiel um eine Max-Funktion, die zu 2 int-Werten den größeren zurückgibt. Im Stack wird für dessen Ausführung (s. Listing 3.2) zunächst wie im Einleitungstext erklärt der aktuelle Base-pointer gepusht und damit gesichert. Danach wird der aktuelle Stackpointer mit dem movl Befehl als Framepointer gesetzt und die eigentlich Funktion kann beginnen zu operieren. Da mit einem positiven Offset zum Basepointer in Z.5 auf eine Variable zugegriffen wird könnte man hinterfragen ob dies Sinn ergäbe, denn nach dem Basepointer wurde doch gar nichts mehr auf den Stack gepusht. Dies erklärt sich dadurch, dass der Stack nicht wie man es vermuten würde nach „oben" hin zu einer größeren Speicheradresse wächst, sondern nach „unten" zu einer niedrigeren Speicheradresse. Dies ist historisch bedingt und auch in heutigen x86-Systemen präsent. Also erfolgt der Zugriff mit einem positiven Offset quasi nach unterhalb des Basepointers außerhalb des Stacks. Genau dort liegen nämlich die zuletzt hinzugefügten Elemente der aufrufenden Funktion und dies sind genau die übergebenen Parameter an die aufgerufene Funktion. Folglich werden in Z.5 und Z.6 die beiden Parameter x und y den Registern %edx und %eax zugewiesen. Darauf folgt ein cmpl Befehl, der ein Bitflag setzt, falls %eax größer %edx sein sollte. In der darauffolgenden Zeile wird dann abgefragt ob dieses Bitflag gesetzt wurde und wenn dem so ist soll zum Abschnitt „end" gesprungen werden. Falls dem nicht so ist wird normal im Code weiter ausgeführt und %edx nach %eax verschoben, sodass nun garantiert der größere Wert in %eax liegt. In der weiteren Ausführung wird die Abschnittsmarkierung „end" nicht weiter beachtet, da diese nur für jump-Befehle von Bedeutung ist und einfach die darauffolgende Zeile ausgeführt. In dieser wird der Stack angewiesen den aktuellen Basepointer als Stackpointer zu werten und danach das oberste Element als Basepointer zu setzten, also genau die Anweisungen des Anfangs rückgängig zu machen. Somit ist wieder der alte Stack der aufrufenden Funktion hergestellt, wobei das Ergebnis der aufgerufenen Funktion im Register %eax liegt.

Listing 3.2: Beispiel-Assembler-Code

```
1  _max:
2                  pushl %ebp
3                  movl %esp, %ebp
4
5                  movl 8(%ebp), %edx
6                  movl 12(%ebp),%eax
7                  cmpl %eax, %edx
8                  jle end
9                  movl %edx, %eax
10             end:
11                 movl %ebp, %esp
12                 popl %ebp
13                 ret
```

4 Abstrakte Maschine in C

In der Sprache C selbst wird die Abstrakte Maschine explizit in der Sprachspezifikation erwähnt:
C Language Specification [4]

- Section 5.1.2.3 3)
  - In the abstract machine, all expressions are evaluated as specified by the semantics. An actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no needed side effects are produced (including any caused by calling a function or accessing a volatile object).
- Section 5.1.2.3 5)
  - The least requirements on a conforming implementation are:
    At sequence points, volatile objects are stable in the sense that previous accesses are complete and subsequent accesses have not yet occurred.
    At program termination, all data written into files shall be identical to the result that execution of the program according to the abstract semantics would have produced.

  Zur Erläuterung: Die beschriebenen Sequence Points sind verschiedene „Sequenzpunkte" in der Ausführung des Codes, die beispielsweise durch das Ende von Blöcken oder auch das Ende von Zuweisungen definiert sind.

  Was dies nun konkret für den Code und die Ausführung bedeutet wird an einem Beispiel gezeigt.

Listing 4.1: C-Code

```
1              static int a, b;
2
3  a = 5;
4
5  b = 42;
6
7  if (a == 5)
8  {
9      <do stuff>
10 }
```

Laut Section 5.1.2.3 5 können bei der Kompilierung des Codes die Zuweisungen von a und b vertauscht werden, da dazwischen offensichtlich keine Zugriffe auf die Variablen stattfindet. Der Compiler könnte sich auch einen völlig anderen Ort im Speicher suchen um b zu speichern, als für a, obwohl die beiden scheinbar direkt hinter einander zugewiesen werden. Vorausgesetzt im if-Block findet kein Zugriff auf b statt, könnte die Zuweisung von eben dieser Variable sogar erst nach dem if-Block stattfinden. Rein logisch gesehen könnte man argumentieren, dass dies egal sei, vor allem weil der Compiler theoretisch wissen müsste was er tut und man ihm vertrauen könne. Allerdings trifft dies nicht auf die Fälle zu in denen man Verhalten implementiert, das von der C-Sprachspezifikation nicht abgedeckt ist. Beispielsweise das allseits beliebte Multi-threading. Dieses kann in C realisiert werden, die Abstrakte Maschine weiß davon jedoch nichts. Aus diesem Grund kann es leicht zu Fehlern kommen, wenn beispielsweise die Zuweisung von Variable b in Thread 1 vom Compiler sehr viel weiter nach hinten verlegt wird, als es der Quelltext vermuten lassen würde und in Thread 2 zwischen diesen beiden Zeitpunkten auf b zugegriffen werden soll, würde es unvermeidbar zu Fehlern kommen. Aus diesem Grund verwendet man für solche Probleme Mutex-Verfahren, die der Ausführung eine Konsistenz geben sollen. Dies geschieht dann natürlich auf Kosten der Performance, weswegen eine feste Verankerung im Sprachstandard wünschenswert wäre. Genau jenes ist im Sprachstandard von C++11 der Fall, worüber folgendes Zitat Auskunft gibt.

- „When you are talking about splitting [code] across different cores that's in the standard, we are talking about the memory model. We are going to optimize it without breaking the following assumptions people are going to make in the code," Sutter [3]

Dies bedeutet nun also, dass Multi-threading ohne besondere Änderungen möglich ist und vor allem dass bei Nutzung eben dieser Funktion die Abstrakte Maschine weiß, wie sie damit umzugehen hat und es hoffentlich nicht zu solch simpel vermeidbaren Fehlern wie im Beispiel angedeutet kommen wird.

5 Quellen

- http://en.wikibooks.org/wiki/LaTeX/Presentations
- http://www.cs.ut.ee/∼varmo/TM2008/slides/tm-cma.pdf [2]
- http://www.theregister.co.uk/2011/06/11/herb_sutter_next_c_plus_plus/page2.html [3]
- http://bradmajors.tumblr.com/post/51783938593/me-you
- http://bartoszmilewski.com/2008/12/01/c-atomics-and-memory-ordering/
- http://mortoray.com/2012/06/18/abstract-machines-interpreters-and-compilers/
- http://blogs.msdn.com/b/larryosterman/archive/2007/05/16/the-c-abstract-machine.aspx
- http://stackoverflow.com/questions/6319146/c11-introduced-a-standardized-memory-model-what-does-it-mean-and-how-is-it-g
- http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf [4]
- http://www.inf.ed.ac.uk/teaching/courses/lsi/diehl_abstract_machines.pdf
- http://tams.informatik.uni-hamburg.de/lectures/2012ws/vorlesung/rs/index.php?content=01-unterlagen