Entwicklung einer Implementationssprache für einen sicheren Mikrokern

Matthias Daum
<md11@inf.tu-dresden.de>

29. Januar 2002

Abriss

C++ hat sich zu einer Art Muttersprache in der Betriebssystemekonstruktion entwickelt. In C++ kann der Programmierer die volle Kontrolle über Speicherverwaltung und Variablentypen selbst übernehmen. Entwurfsziele wie Effizienz, Optimierbarkeit und eine möglichst hohe Kontrolle über die zugrundeliegende Maschine wurden über die Typsicherheit und eine wohldefinierte Semantik gestellt.

Dadurch gelingt es jedoch nicht, die Korrektheit von C++-Programmen formal zu beweisen. Die vorliegende Arbeit soll nun eine Sprache entwickeln, die C++ syntaktisch ähnelt, aber eine wohldefinierte Semantik besitzt und somit die Verifikation in ihr geschriebener Programme gestattet.

Aufgabe

Ziel ist die Entwicklung einer Programmiersprache für die Implementierung eines sicheren, verifizierten Mikrokerns, die einerseits gängige Arbeitsweisen der Betriebssystemprogrammierung gestattet (systemnahe Programmierung, objektorientierte Programmierung), andererseits aber eine leicht formalisierbare Semantik besitzt.

Anforderungen an die Sprache. Die Sprache sollte eine zu C++ ähnliche Syntax haben und um einige definierte Sprachkonstrukte erweitert worden sein, die der Implementation eines Betriebssystemkerns dienen. Sie muss semantisch wohldefiniert sein, sodass in ihr verfasste Programme von einem Semantikcompiler automatisch in die Logiksprache eines Beweisers übersetzt werden können. Zur Vereinfachung der Semantikspezifikation sollte die Implementationssprache auf die benötigten Ausdrucksmittel beschränkt sein.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Heutzutage wird die Korrektheit von Programmen durch sorgfältigen Entwurf und intensives Testen geprüft. Doch selbst die umfangreichsten Tests können Fehlerfreiheit nicht sicherstellen. Ein verlässlicher Beweis kann nur durch formale Methoden erbracht werden.

Für die meisten Anwendungen ist Fehlerfreiheit zwar erwünscht, aber nicht zwingend notwendig. Gerade die jüngsten Entwicklungen in der Netzwerktechnik lassen jedoch einen großen Zuwachs kryptographischer Anwendungen vermuten, die auf handelsüblichen PCs beispielsweise Dokumente zertifizieren oder Geld transferieren. Die Korrektheit dieser Anwendungen in einer Umgebung potentiell böswilliger Angreifer ist dabei von höchster Bedeutung.

Doch kann eine Anwendung nur dann eine Gewähr über seine Korrektheit geben, wenn es sich auf gleichermaßen starke Garantien des unterliegenden Betriebssystems stützen kann. Für die Validität einer elektronischen Signatur muss das Betriebssystem beispielsweise sicherstellen, dass niemand den Hauptspeicher auslesen kann, während das Signaturprogramm läuft. Sonst ließe sich der für die Signatur benutzte, geheime Schlüssel ausspähen.

In dem gemeinsamen Projekt „VFiasco" der Lehrstühle „Algebraische und logische Grundlagen der Informatik" und „Betriebssysteme" der Fakultät Informatik an der TU Dresden soll ein wesentlicher Schritt hin zu einem beweisbar korrekten Betriebssystem getan werden. Hierfür soll der zur Zeit in Entwicklung befindliche Betriebssystemkern Fiasco so angepasst werden, dass einige sicherheitsrelevante Eigenschaften des gesamten Betriebssystems verifiziert werden können. Dies bedingt zugleich eine Weiterentwicklung coalgebraischer Spezifikationstechniken, so dass sie sich auf reale Software anwenden lassen.

Derzeit ist Fiasco in C++ implementiert. Diese, in der maschinennahen Programmierung sehr beliebte Programmiersprache ist für eine Programmverifikation jedoch ungeeignet. Benötigt wird eine Sprache, die es einerseits erlaubt, einen möglichst großen Teil des bestehenden Programmtextes zu übernehmen, eine Verifizierung von Programmen jedoch wesentlich vereinfacht.

Die vorliegende Arbeit beschäftigt sich mit dem Entwurf von „Safe C++", einer neuen, typsicheren Programmiersprache, die den oben genannten Zielen gerecht wird. Im nächsten Kapitel wird der derzeitige Stand der Technik kurz vorgestellt. Der Hauptteil der Arbeit (Kapitel 3) beschäftigt sich mit problematischen Konstrukten in C++. Hier dienen die ersten zwei Abschnitte dazu, alle Probleme aufzudecken; anschließend werden Lösungsvorschläge diskutiert.

Kapitel 4 wird untersuchen, wie sich unvermeidliche Einbußen in der Ausdrucksmächtigkeit von Safe C++ ausgleichen lassen und welche Sprachmittel zusätzlich eingeführt werden, um Safe C++ auf den Einsatz in maschinennaher Umgebung vorzubereiten. Abschließen wird die Arbeit mit einer kurzen Übersicht über die wichtigsten Unterschiede zwischen C++ und Safe C++.

Kapitel 2

Stand der Technik

Die Idee eines sicheren Betriebssystems ist schon recht alt. Meist scheitert man jedoch an der Komplexität des Programmcodes. Ein erster Schritt zur besseren Gliederung der Programmteile ist der Entwurf eines erweiterbaren Betriebssystems. Hierfür sei beispielhaft SPIN genannt, das in Modula-3 geschrieben wurde[17][18][19]. Mikrokerne ziehen eine noch schärfere Grenze zwischen dem Kern, als zentralem Bestandteil des Betriebssystems, und angegliederten Systemfunktionen, wie zum Beispiel der Dateiverwaltung. Die strengere Unterteilung macht eine Programmverifikation wesentlich leichter; deswegen entschied man sich für die Reimplementierung von Fiasco[1][2] in einer Sprache mit wohldefinierter Semantik. Modula-3 ist dafür jedoch schlecht geeignet, da Fiasco derzeit in C++ geschrieben ist und deren Syntax sich stark von Modula-3s unterscheidet.

Die Schwächen von C++ sind ebenfalls schon lange bekannt und Gegenstand langjähriger Forschung. Nicht zuletzt sind die Programmiersprachen Java und C#[4] Versuche einer Verbesserung von C++. Diese sind jedoch nicht systemnah genug für die Implementation eines Betriebssystems. Zahlreiche Arbeiten beschäftigen sich daher mit der fehlenden Speichersicherheit (*memory safety*), dem wohl größten Problem von C++. Die Vorschläge reichen von allgemeinen Richtlinien zur Programmierung für sicherheitskritische Anwendungen[6] über Programmbibliotheken für eine automatische Speicherbereinigung[7][10] bis hin zur Sprachanpassung[5] an eine neue Zeigerphilosophie. Nicht zwangsläufig müssen sich die Techniken zur Erkennung von Zugriffsfehlern auf C/C++ beschränken[11]. Ein sehr junges Projekt ist Cyclone, ein sicherer C-Dialekt[12]. Die Autoren versprechen, durch statische Analyse und das Einfügen von Laufzeitprüfanweisungen die gleiche Sicherheit wie höhere Programiersprachen, z. B. Java, zu erreichen. Das Ziel der vorliegenden Arbeit ist es, gleiches für C++ zu garantieren.

Kapitel 3

Problematische Konstrukte in C++

Die systematische Suche nach Problemen in der C++-Semantik soll im folgenden auf 2 Säulen stehen: Zum Einen werden einige Faustregeln für leicht formalisierbare Semantiken vorgestellt. Ein Vergleich von C++ und Java soll spezifische Probleme genauer beleuchten und bei der Suche nach Alternativen helfen.

3.1 Faustregeln zur Semantikdefinition

Bei der Entwicklung formaler Semantiken für bestehende Programmiersprachen haben sich einige beliebte, aber weniger zentrale Eigenschaften häufig als Fallstricke für die Formalisierung herausgestellt. Daraus lassen sich leicht einige Faustregeln für neue Sprachen ableiten, deren Semantik sich leichter formalisieren lässt.

So empfehlen sich

- strukturierte Programme nach Wirth,
- eine klare Trennung zwischen Anweisungen und Ausdrücken und
- ausschließlich explizite Typumwandlungen.

Problematisch sind dagegen

- Überläufe (*roll overs*) in der Arithmetik und
- Zeigerarithmetik.

3.1.1 Programmstrukturierung nach Wirth

Nicht nur für die Formalisierung der Sprachsemantik, auch aus rein softwaretechnischer Sicht wird heute in der Regel strukturierte Programmierung bevorzugt. Wenn es in C++ dennoch einen `goto`-Befehl gibt, dann wohl eher aus historischen Gründen. Daher sollte die Streichung dieses Kommandos in Safe C++ keinerlei Probleme bereiten.

Ein Grenzfall ist das `switch`-Kommando von C++. Seine Semantik des „Durchfallens" läßt sich zwar prinzipiell formalisieren; doch ist der damit verbundene Aufwand recht hoch, verglichen mit der Pragmatik dieser Semantik.

3.1.2 Trennung zwischen Anweisungen und Ausdrücken

Das zentrale Anliegen einer klaren Unterscheidung zwischen Anweisungen und Ausdrücken liegt – neben der konzeptionellen Ästhetik – in der einfacheren Semantik von komplexen Ausdrücken. Angestrebt wird, den globalen Zustand des Programms während der Berechnung eines Ausdruckes nicht anzutasten. Dann darf es innerhalb einer Ausdrucksberechnung keine Seiteneffekte geben.

3.1.3 Typumwandlungen

Naheliegenderweise gestaltet sich die Konvertierung eines Typs als problematisch, sobald der Zieltyp den Wertevorrat[1] des Ausgangstyps nicht vollständig umfasst. Schon um Programmierfehler zu vermeiden, sollte eine solche Umwandlung stets explizit vorgenommen werden. Im Zusammenspiel mit der Überladung von Operatoren und Methoden führt das implizite Konvertieren in breitere Typen zu einer immens großen Anzahl von Regeln, deren Komplexität auch zahlreiche Quellen für Programmierfehler birgt.

3.1.4 Überläufe in der Arithmetik

Sind die Wertebereichsgrenzen für Datentypen fest definiert, lässt sich ein Überlauf (*roll over*) in der Semantik von Ausdrucksberechnungen berücksichtigen. Allerdings fordert eine solche formale Spezifikation einen hohen Aufwand, gemessen an der seltenen, realen Nutzbarkeit dieser Eigenschaft. Zum Übersetzungszeitpunkt

[1] Der Begriff Wertevorrat soll hier in strikt mathematischem Sinne verstanden werden und ist vom Werteumfang zu unterscheiden. Der Wertevorrat zweier Aufzählungstypen (1,2,3) und (A,B,C) ist keinesfalls gleich, sondern sogar disjunkt.

kann noch nicht auf Überläufe geprüft werden; die Einfügung spezieller Instruktionen für eine Überprüfung zur Laufzeit ist aber weitaus zu aufwendig. Statt dessen lässt sich die Freiheit von Überläufen aber bei der Programmverifikation in einem eigenen Beweis sicherstellen.

3.1.5 Zeigerarithmetik

Zeiger ermöglichen in C++ den indirekten Zugriff auf Datenobjekte im Speicher. Sie stellen somit ein konzeptionell unverzichtbares Sprachmittel dar. Doch verbinden sich mit dem sehr freien Umgang mit Zeigern auch zahlreiche Probleme. In besonderem Maße trifft dies für das Rechnen mit Zeigern zu. Dadurch ist es möglich, den Zeiger von dem Datenobjekt zu lösen, auf das er ursprünglich verwies.

Konzipiert wurde die Zeigerarithmetik als eine einfache und effiziente Implementierung von Feldern (*arrays*); doch ermöglicht sie letztlich den Zugriff auf beliebige Speicherbereiche, unabhängig vom Datentyp der dort gespeicherten Informationen. Dies ist möglich, weil bei der Adressrechnung nicht geprüft wird, ob an der neuen Speicheradresse tatsächlich ein Datenobjekt vom Referenztyp (*pointee type*) des Zeigers abgelegt ist.

3.2 Unterschiede zwischen Java und C++

Java lehnt sich syntaktisch stark an C und C++ an, hat jedoch eine wohldefinierte Semantik. Bei C++ wurde hoher Wert auf Effizienz zur Laufzeit und Optimierbarkeit gelegt. Deshalb hat der Programmierer umfangreichen und freien Zugriff auf die darunterliegende Maschine, durch diese Freiheit aber auch eine große Verantwortung. Die Entwickler von Java verfolgten dagegen das Ziel einer robusten, klaren und einfach zu beherrschenden Programmiersprache. Deshalb fehlen in Java viele Konzepte aus C++, die als unsicher oder zu anfällig für Programmierfehler befunden wurden. Safe C++ soll nun sowohl maschinennah als auch robust sein, gewissermaßen also die Annehmlichkeiten beider Sprachen verbinden. Daher empfiehlt sich der Blick auf die Unterschiede zwischen Java und C++.

3.2.1 Präprozessor

Java verfügt im Gegensatz zu C++ nicht über einen Präprozessor. Der C++-Präprozessor dient vor allem dazu, Deklarationen für extern definierte Programmteile einzubinden (`#include`) sowie Konstanten und Makros zu definieren

(`#define`). Darüber hinaus erlaubt der Präprozessor eine bedingte Übersetzung. Diese Vorübersetzung verlangt vom Programmierer aber das Denken in den zwei Verarbeitungsebenen und die Kenntnis beider Sprachen.

Deshalb wurden für Java alternative Konzepte zur Umgehung des Präprozessors entwickelt. Dazu zählt die Übersetzung in 2 Pässen, die beliebige Vorwärtsreferenzen ohne eine vorherige Deklaration erlaubt. Eine Deklaration externer Klassen ist überflüssig, denn der Zugriff auf andere Klassen erfolgt auch zur Laufzeit generell über deren vollständigen Namen. Da Java-Klassen in Paketen (*packages*) organisiert sind, können die Namen sehr lang sein. Lediglich zur kürzeren Notation dient daher die `import`-Anweisung; das Einlesen anderer Quelltextdateien ist nicht mehr möglich. Konstanten können durch das Schlüsselwort `final` definiert werden. Präprozessor-Makros sind durch die Einführung von `inline`-Funktionen schon in C++ nicht mehr notwendig; in Java wird die Entscheidung über das Einbetten allerdings ganz dem Compiler überlassen.

3.2.2 Variablen und Funktionen

In Java müssen alle Variablen und Funktionen einer Klasse angehören. Globale Variablen oder Funktionen, wie man sie aus C++ kennt, lassen sich nicht definieren. Dieser Umstand scheint jedoch nur dem Konzept des dynamischen Auffindens von Klassen geschuldet zu sein.

Für die Semantikdefinition von Safe C++ interessanter ist dagegen die in Java übliche implizite Initialisierung bereits bei der Variablendefinition, die in C++ nicht konsequent vorgenommen wird.

Die von C++ her bekannten Funktionen mit variabler Argumentenanzahl kann es in einer typsicheren Sprache wie Java und Safe C++ selbstverständlich nicht geben. Die Angabe von Vorgabewerten (*default values*) für Methodenargumente ist – obwohl prinzipiell formalisierbar – in Java nicht möglich.

3.2.3 Datentypen

Es gibt einige weniger relevante Unterschiede in Art und Wertumfang der ***primitiven Basistypen*** von Java und C++. So findet man in Java zusätzlich den Typ `byte`, vermisst dafür aber vorzeichenlose Datentypen. Von größerer Bedeutung ist die exakte Angabe des Wertebereichs und des Zeichenvorrates in Java. Gerade für eine typsichere Sprache etwas verwunderlich ist die Tatsache, dass sich in Java keine Aufzählungstypen definieren lassen.

Sehr interessant ist die Konzeption der ***komplexen Datentypen***. Java kennt neben primitiven Werten nur Objekte. Strukturen (`struct`) und Varianten (`union`), wie in C++ üblich, gibt es nicht. Die Adressierung von Objekten erfolgt ausschließlich über Referenzen; eine nicht vorhandene Referenz wird durch das Schlüsselwort `null` gekennzeichnet. Java trennt sich damit vollständig vom Konzept der Zeiger.

Felder werden in Java wie ein Objekt verwaltet. Die Bereichsgrenzen werden vor dem Zugriff geprüft.

Eine ***Definition neuer Typnamen***, wie mit `typedef` in C++, ist in Java nicht möglich.

Implizite Typumwandlungen sind sowohl in Java als auch in C++ bekannt, wobei Java hier etwas restriktiver ist als C++. Aus Gründen der Effizienz werden in C++ 3 Arten expliziter Typkonvertierungen unterschieden:

- `static_cast` für eine Typprüfung zur Übersetzungszeit,
- `dynamic_cast` für eine Typprüfung zur Laufzeit und
- `reinterpret_cast`, falls keine Typprüfung gewünscht wird.

Java kennt diese Unterscheidung nicht; die Typprüfung erfolgt bei allen Umwandlungen dynamisch.

3.2.4 Operatoren

Durch den Verzicht auf Zeiger sind Dereferenzierungs- und Referenzierungsoperator (`*`, `&`) in Java überflüssig geworden. Darüber hinaus fehlen der Komma- und der `sizeof`-Operator. Im Gegenzug dafür findet man den in C++ unbekannten `instanceof`-Operator.

Eine ***Überladung von Operatoren*** ist in Java nicht möglich. Doch gibt es für alle Objekte eine Methode zum Vergleichen (`equals()`) und eine zum Kopieren (`clone()`), da alle Klassen implizit von der Klasse `Object` abgeleitet werden.

3.2.5 Klassen

Java bietet nicht die Möglichkeit einer Parametrisierung von Klassen oder Funktionen (*templates*). Deren gebräuchlichster Anwendungsfall ist sicherlich die Implementierung universeller Klassen und Funktionen, die mit Daten eines beliebigen Typs operieren; beispielsweise Containerklassen wie Felder oder Listen. In Java wird dieses Problem durch die allen Klassen gemeinsame, implizite Superklasse `Object` gelöst.

Mehrfachvererbung ist in Java nicht erlaubt. Die Methodenbindung erfolgt ausschließlich dynamisch (*dynamic method lookup / late binding*).

Eine Zusammengehörigkeit und damit verbundene, enge Zusammenarbeit von Klassen wird in Java nicht länger durch eine Freundschaftsbeziehung (Schlüsselwort `friend`) ausgedrückt, sondern spiegelt sich in der Zugehörigkeit zu einem gemeinsamen Paket wider. Diesem Konzept ist die neu hinzu gekommene Paket-Sichtbarkeit geschuldet.

Durch einige syntaktische Ergänzungen gelingt Java eine höhere Übersichtlichkeit im Umgang mit Klassendefinitionen: So wurde das Schlüsselwort `abstract` für Klassen und Methoden eingeführt. Bei Klassen in aller Regel lediglich der Hinweis auf eine abstrakte Methode, für letztere nur eine eingängigere Syntax statt der in C++ üblichen Null-Zuweisung in der Methodendefinition (`abstract void method()` statt `virtual void method()=0`). Ebenfalls neu ist das Definieren von sogenannten Schnittstellen (`interface`s). Diese lassen sich als attributlose, abstrakte Klassen mit ausschließlich abstrakten Methoden auffassen. Des weiteren ist es in Java möglich, Klassen und Methoden als `final` zu deklarieren. Sie können dann nicht mehr durch Subklassendefinitionen überschrieben werden.

3.2.6 Dynamische Speicherverwaltung

Java erlaubt keine explizite Speicheranforderung und -freigabe. Die Erzeugung neuer Objekte erfolgt mittels des Schlüsselwortes `new`. Nicht mehr referenzierte Objekte werden bei Bedarf durch die automatische Speicherbereinigung (*garbage collection*) freigegeben.

3.2.7 Programmfluss-Steuerung

Ein Sprung an eine beliebige Marke (*label*) innerhalb des Programms, z. B. mit `goto`, ist in Java nicht vorgesehen. Statt dessen ist es möglich, in einer Schleifenstruktur mehrere Ebenen auf einmal zu überwinden (*labelled* `break`, `continue`).

3.2.8 Weitere Besonderheiten

Eine ***Ausnahmebehandlung*** bieten sowohl Java als auch C++ an. Java ist hier jedoch in seiner Konzeption etwas restriktiver. Zum Einen verlangt es die explizite Angabe der in einer Methode möglichen Ausnahmen bei der Deklaration; zum Anderen dürfen nur Objekten spezieller Klassen (`Throwable` und Subklassen) „geworfen" werden.

Im Gegensatz zu C++ lassen sich in Java keine ***Assembler-Anweisungen*** in den Quelltext einbetten. Es können lediglich plattformabhängig implementierte Methoden mit Hilfe des Schlüsselworts `native` eingebunden werden.

3.3 Diskussion zum Umgang mit problematischen Konstrukten

Die oben genannten Probleme und Besonderheiten von C++ bedürfen einer differenzierten Betrachtung unter dem Gesichtspunkt der Semantikdefinition. Sie werden nun ausführlich diskutiert und mögliche Vorschläge für den Umgang mit ihnen entwickelt. Am Ende sollen in Grundsatzfragen klare Entwurfsentscheidungen getroffen werden.

3.3.1 Präprozessor

Der Verzicht auf einen Präprozessor hat weit reichende Konsequenzen für die Sprachdefinition. Schon deshalb kommt dies für Safe C++ nicht in Frage. Zudem sind die Anweisungen an den Präprozessor kein integraler Sprachbestandteil und somit für die Semantik der eigentlichen Sprache irrelevant. Safe C++ selbst wird keine eigenen Präprozessor-Anweisungen umfassen; es empfiehlt sich aber die Verwendung des C++-Präprozessors vor der Übersetzung durch den Safe-C++-Compiler.

3.3.2 Variablen und Funktionen

Die Behandlung uninitialisierter Variablen erfordert einen hohen Aufwand bei der Formalisierung. Deshalb werden Variablen in Safe C++ bei ihrer Definition initialisiert.

Die Deklaration von Funktionen mit variabler Argumentenanzahl ist in Safe C++ nicht möglich, da dies den Verlust der Typsicherheit nach sich ziehen würde.

Die Angabe von Vorgabewerten (*default values*) für Methodenargumente wird in Safe C++ nicht unterstützt, da diese Möglichkeit – gemessen an dessen hohem Formalisierungsaufwand – zu selten genutzt wird.

3.3.3 Datentypen

Primitive Basistypen. Schlichtweg notwendig für eine wohldefinierte Semantik ist die ***exakte Angabe des Wertebereichs bzw. des Zeichenvorrats*** für die primitiven Datentypen. Hardware-nahe Programmierung verlangt darüber hinaus oft die genaue Kenntnis der Speichergröße eines Typs. Zum Teil ist sogar die maschineninterne Repräsentation relevant. – Ein populäres Beispiel dafür sind Zeichensätze; möglicherweise gibt es aber auch Anwendungsfälle, in denen die interne Darstellung numerischer Typen bekannt sein muss.

Jede Maschinenabhängigkeit erfordert bei einer Portierung von Programmtext erneut einen sehr hohen Verifikationsaufwand. Um diesen einzudämmen, sollte der Programmierer in der Lage sein, die gewünschte Verwendung einer Variable sehr genau anzugeben. Deshalb wird zum Beispiel ein Typ wie `char` in Safe C++ ausschließlich der Speicherung von Zeichen vorbehalten bleiben.

Die Zielsetzung des neuen Typsystems unterscheidet sich damit klar von dem in C++ gewählten Ansatz. Schon um Verwechslungen vorzubeugen, empfiehlt sich die Wahl neuer Typnamen. Der entscheidende Vorteil ist jedoch die einfachere Abbildung der neuen Typen auf die C++-Typen, die dann nur ein `typedef` erfordern. Vorzeichenbehaftete Ganzzahlen werden auf „int", gefolgt von der Speicherbreite in bit, getauft: `int8`, `int16`, `int32` und `int64`. Nicht-negative Ganzzahlen heißen entsprechend `uint8` bis `uint64`.

Gleitkommatypen werden zur Implementierung von Betriebssystemkernen in aller Regel nicht benötigt. Zur Vereinfachung der Semantikspezifikation wird daher ganz auf sie verzichtet. Bei Bedarf ließen sich die üblichen Typen `float` und `double` aber problemlos ergänzen (Definition entweder auf Basis der reellen Zahlen oder über den IEEE-Standard).

Überläufe (*roll overs*) in der Arithmetik werden zu selten genutzt, als dass der verhältnismäßig hohe Aufwand für die formale Spezifikation dieser Eigenschaft gerechtfertigt wäre. Sie sind in Safe C++ daher nicht definiert.

In C++ müssen ***Aufzählungstypen*** nicht benannt werden. Die vereinbarten Bezeichner dürfen überall dort im Programm verwendet werden, wo ganzzahlige Werte erlaubt sind. Dadurch verschwimmt der Unterschied zu Konstantendefinitionen. Safe C++ verlangt deshalb die Benennung von Aufzählungstypen. Die

Bezeichner sind dort und nur dort zu verwenden, wo ein Wert des betreffenden Typs zulässig ist. Etwaige Umwandlungen zwischen Aufzählungstypen und Ganzzahlen müssen explizit erfolgen.

Modifizierte Typen. Aus dem in C++ üblichen, sehr freien Umgang mit ***Zeigern*** ergeben sich zahlreiche Schwierigkeiten: Bereits kurz angeschnitten wurde die Problematik des Rechnens mit Zeigern, die in engem Zusammenhang mit der Verwaltung von Feldern steht. Ebenfalls in diesen Kontext gehören Typumwandlungen in Zeigertypen (siehe unten) und hängende Zeiger.

Zur typsicheren Verwendung der Zeigerarithmetik muss sichergestellt sein, dass Zeiger jederzeit auf ein Datenobjekt vom Typ des Zeigers verweisen. In dieser Allgemeinheit verlangt eine solche Prüfung die Speicherung von Typinformationen zu den Speicheradressen.

Alternativ könnte man Zeigern auch einen Gültigkeitsbereich zuordnen, denn die Zeigerarithmetik wurde ja durch Felder motiviert. Die Semantik eines Programms ist dann nur definiert, wenn die Zeiger die Bereichsgrenzen nicht überschreiten. Probleme bereitet jedoch eine klare Definition dieser Grenzen: Verweist ein Zeiger gerade auf ein Feld, so wäre Zeigerarithmetik in deren Grenzen erlaubt; zeigt er auf einen Basistyp, ist sie nicht erlaubt. Demnach handelt es sich um 2 verschiedene Typen. Soll es keine impliziten Typumwandlungen geben, müsste man diese Unterscheidung also bereits bei der Zeigerdeklaration treffen. Ein solcher Ansatz erscheint jedoch sehr unübersichtlich.

Statt dessen wird das Rechnen mit Zeigern ganz unterbunden. Dies scheint zunächst zu einem Problem bei der Übergabe von ***Feldern*** an Unterprogramme zu führen, doch schafft hier die Einführung einer eigenen Typkategorie für Felder Abhilfe. Sie lässt sich in C++ einfach durch ein Template implementieren. Auf diese Weise kann gleichzeitig auch die Prüfung der Bereichsgrenzen sichergestellt werden.

Als ***hängend*** wird ein Zeiger bezeichnet, der nach dem Verschwinden des ursprünglich referenzierten Datenobjekts weiterhin auf dessen Speicherbereich zeigt. Am auffälligsten geschieht dies bei der expliziten Freigabe dynamischer Datenstrukturen, auf die es mehrere Verweise gibt. Hier versprechen mehrere Lösungsansätze Abhilfe, vom Fehlschlagen der Freigabe bei mehreren Referenzen bis hin zur völligen Abschaffung expliziter Speicherfreigabe. Im Abschnitt 3.3.6 über dynamische Speicherverwaltung wird näher darauf eingegangen.

Es gibt aber auch eine weniger offensichtliche Gelegenheit zur Entstehung hängender Zeiger: das Verlassen des Gültigkeitsbereichs (*scopes*) einer referenzierten

Kellervariable (auch automatische Variable). Um dies zu verhindern, führt Nagle in [5] „temporäre Zeiger" ein. Auf Implementationsebene handelt es sich dabei schlicht um „rohe", d. h. einfache, C++-Zeiger ohne Referenzzähler. Sie werden mit dem Schlüsselwort `auto` kenntlich gemacht und können das referenzierte Objekt nicht überleben. Im Einzelnen bedeutet dies:

- `auto`-Datenstrukturen können keine Werte eines kleineren Gültigkeitsbereiches zugewiesen werden.
- `auto`-Datenstrukturen müssen mit einem gültigen Wert initialisiert werden. Zeigerarithmetik ist für `auto`-Zeiger nicht erlaubt.
- nicht-`auto`-Zeiger dürfen `auto`-Zeigern zugewiesen werden, jedoch nicht umgekehrt.
- `auto` ist als Attribut für Funktionsargumente erlaubt.

Nagle stellt heraus, dass ein etwaiger, dadurch auftretender Gültigkeitsbereichsfehler bereits zur Übersetzungszeit festgestellt werden kann. Es bedarf keiner zusätzlichen Laufzeitprüfung für `auto`-Variablen; das legt den Vergleich mit `const` nahe.

Sehr interessant u. a. im Zusammenhang mit der Parameterübergabe von Zeigern an Funktionen ist die Frage, ob und in wie weit sich dieses Konzept auch mit dynamischen Objekten verträgt. Dies ist in so fern problematisch, als diese keinen festen Gültigkeitsbereich besitzen. Nagles Beispielen zufolge hält er beide Konzepte für vereinbar, doch fehlen weitere Ausführungen dazu noch.

Zeiger auf globale Datenstrukturen können dagegen gar nicht hängen. Es empfiehlt sich daher für jeden der 3 Fälle (dynamische, automatische und globale Referenzobjekte) eine eigene Zeigerkategorie einzuführen.

Zusammengesetzte Datentypen. ***Strukturen*** gefährden in keiner Weise die Typsicherheit. Es gibt daher keinen Grund, sie in Safe C++ nicht zu erlauben.

Ganz anders verhält es sich dagegen mit ***Varianten*** (`unions`). Die in C++ übliche Implementierung bietet keine Möglichkeit, den Typ des in einer Variante gehaltenen Werts im Nachhinein festzustellen; die Kontrolle darüber verbleibt vollständig beim Programmierer. Für eine typsichere Sprache ist dies jedoch inakzeptabel.

Varianten erlauben, knapp ausgedrückt, die gemeinsame Nutzung von Speicher durch verschiedene Variablen. Für alternativ verwendete Variablen bedeutet dies lediglich eine effektivere Speichernutzung. Hierfür schlägt Binkley[6] eine typsichere Implementierung unter Verwendung von Vererbungsbeziehungen vor[2]. Zugleich bietet sich mit Varianten aber auch die Möglichkeit zur Reinterpretation von Werten, beispielsweise beim Füllen von Bit-Datenstrukturen. Gerade im Bereich der Kernprogrammierung sind dafür Anwendungen denkbar, wie das Einlesen von Daten aus Registern oder dem Nutzerbereich. Hier muss Safe C++ geeignete, alternative Mechanismen bereitstellen.

Typumwandlungen. Safe C++ geht ganz sicher einen Sonderweg, wenn es Typumwandlungen einer leicht überschaubaren Semantik zuliebe generell explizit vornehmen lässt. Doch scheint die unvermeidlich hohe Komplexität von Regeln für implizite Typkonvertierungen und die damit verbundene Anfälligkeit für Programmierfehler ein solches Experiment durchaus zu rechtferigen.

In konsequenter Weiterführung dieses Ansatzes muss sich auch die Semantik von ***Typdefinitionen*** ändern. Während in C++ lediglich alternative Bezeichnungen (*aliases*) für die schon bestehenden Typen eingeführt werden können, soll nun zusätzlich das Erzeugen eines eigenständigen, neuen Typs möglich sein. Statt des Schlüsselworts `typedef` gibt es in Safe C++ `typealias` zur Deklaration eines zusätzlichen Namens für den selben Typ und `typecreate`, um einen neuen Typ zu erstellen.

C++ erlaubt sehr freizügige Typkonvertierungen. In den meisten Fällen ist eine Umwandlung jedoch unkritisch und kann recht leicht mit einer Semantik unterlegt werden. Problematisch gestalten sich jedoch ***Konvertierungen in Zeigertypen***. Gerade in Betriebssystemkernen ist es allerdings in bestimmten Fällen erforderlich, konventionelle Werte in Speicheradressen umzuwandeln.

So liegt es nahe, diese Art von Konvertierungen zu erlauben und mit einer wohldefinierten Semantik zu unterlegen. Dazu muss man sicherstellen, dass so entstandene Zeiger tatsächlich auf ein Datum des postulierten Typs zeigen. Wie bereits bei der Besprechung der Zeigerarithmetik festgestellt, verlangte eine solche Prüfung die Mitführung von Typinformationen für alle Speicheradressen. Das bedeutet die vollständige Modellierung des Adressraumkonzeptes. Auf Maschinen mit Unterstützung mehrerer virtueller Adressräume wäre der ohnehin schon enorme Aufwand überdies für jeden Adressraum einzeln zu treiben. Es ist leicht

[2] Die Variante wird dabei durch eine abstrakte Basisklasse ersetzt, deren Schnittstelle durch typspezifische Unterklassen zu implementieren ist. Nagle[5] führt diese Idee genauer aus, indem er die Schnittstelle der Basisklasse mit einem Funktionspaar get / set pro Typ definiert und vorimplementiert (Erzeugen einer Ausnahme). In den Unterklassen wird dann jeweils nur das zutreffende Funktionspaar reimplementiert.

einzusehen, dass dies kaum praktikabel sein kann.

Auch erwähnt werden sollte ein damit verbundener Sicherheitsaspekt: Ohne die Möglichkeit zur Konvertierung in Zeigertypen kann ein Programm keine Referenzen auf „fremde" Speicherbereiche erzeugen. Ein System kann sich also bei der Ausführung von Unterprogrammen sicher sein, dass die eigenen Speicherbereiche davon unberührt bleiben. Zugreifbar sind nur jene Datenobjekte, für die vom betrachteten Programmtext aus (bereits) Referenzen existieren.

Beschneidet man C++ jedoch um diese Fähigkeit, beraubt man es um einen großen Anteil seiner Ausdrucksmächtigkeit. Den entstandenen Verlust müssen neue Sprachmittel kompensieren. Die dazu erforderliche, genaue Untersuchung der einzelnen Anwendungsfälle soll im folgenden Kapitel vorgenommen werden.

3.3.4 Ausdrücke und Operatoren

Seiteneffekte in Ausdrücken. Eine von Seiteneffekten freie Ausdrucksberechnung würde die Definition der Semantik stark vereinfachen; doch stellt sich diese Forderung als problematisch heraus: Einerseits liefern Funktionen Werte – sind somit also Ausdrücke. Andererseits können sie aber auch Anweisungen enthalten, die den globalen Programmzustand ändern. Letzteres zu verbieten, bedeutete eine starke Einschränkung in der Programmierung.

Seiteneffekte in Ausdrücken nur innerhalb von Funktionsaufrufen zu erlauben, erscheint bei genauerer Betrachtung ebenso wenig sinnvoll. Zum einen vereinfacht sich die Semantikdefinition dadurch nicht. Zum anderen lässt sich jeder Seiteneffekt letztlich auf einen Operator zurückführen; der wiederum kann auch als Funktionsaufruf geschrieben werden.

Sind Seiteneffekte in Ausdrücken zulässig, so hängt der Wert des Ausdrucks ohne Beschränkung der Allgemeinheit von der Reihenfolge der Auswertung ab. Daher muss für eine wohldefinierte Semantik die Auswertungsreihenfolge festgelegt werden.

Operatoren. Wenn in Java auf den Komma-Operator verzichtet wird, so lässt sich dies als ein Schritt zu einer klareren Trennung zwischen *Anweisung* und *Ausdruck* auffassen. Dieser Intention folgend, wird es den Operator auch in Safe C++ nicht geben.

3.3.5 Klassen

C++ stellt mit seinen Templates (Typschablonen) ein äußerst mächtiges Werkzeug zur Verfügung. Zugunsten einer leichteren Formalisierbarkeit wird Safe C++ sich auf den gebräuchlichsten Fall parametrisierter Klassen und Funktionen beschränken und spezialisierende Templates sowie Templates innerhalb von Klassen (*member templates*) nicht zulassen.

3.3.6 Dynamische Speicherverwaltung

Die Implementierung der expliziten Speicherfreigabe in C++ birgt zwei Risiken: Zum Einen kann es reservierte Speicherbereiche geben, die nicht mehr referenziert werden, und zum Anderen ist es möglich, dass Zeiger auf bereits freigegebene Bereiche verweisen (*hängende Zeiger / dangling pointers*). Ersteres mag unerwünscht sein, bezüglich der Typsicherheit bringt jedoch nur der zweite Fall Gefahren mit sich.

Java löst beide Probleme durch eine automatische Speicherbereinigung (*garbage collection*). Ebenfalls aus der Literatur bekannt sind vielfältige Implementierungen von eleganten Zeigern (*smart pointers*) zur Verhinderung hängender Zeiger. Dazu gehören:

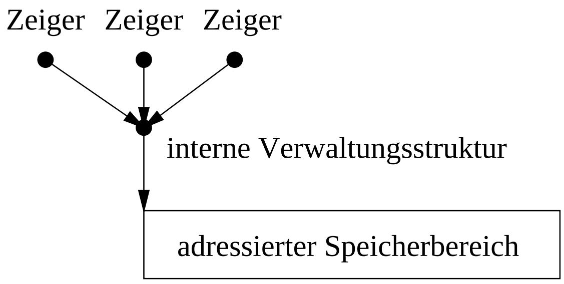
- die Einführung einer weiteren Indirektionsstufe,
- das Zählen der Referenzen auf Speicherbereiche oder
- Pseudonyme für Zeiger.

Im Folgenden sollen die Konzepte kurz skizziert werden.

Automatische Speicherbereinigung macht einen Befehl zur expliziten Freigabe eines bestimmten Speicherbereiches überflüssig. Statt dessen gibt es eine Routine, die alle nicht mehr referenzierten Speicherbereiche auffindet und freigibt. Diese Freigaberoutine kann dann beispielsweise bei Speicherknappheit aufgerufen werden. Deren Implementierung ist allerdings verhältnismäßig aufwendig; auch sind die Auffassungen zu diesem Ansatz durchaus geteilt[9][5].

Es gibt einige bekannte Implementierungen, die jedoch mit Problemen behaftet sind. Als Beispiel seien hier Safe C++[10] von Xerox PARC und der Conservative Garbage Collector von Hans Boehm[7] genannt. Die Arbeit von Xerox PARC enthüllt Differenzen zwischen der Destruktor-Semantik von C++ und dem Konzept der Speicherbereinigung; und der Conservative Garbage Collector stellt nicht sicher, dass alle unerreichbaren Speicherbereiche auch freigegeben werden[8]. Überdies verschlechtert sich bei einer derartigen Implementierung zwangsläufig die Vorhersagbarkeit für die Dauer einer Speicheranforderung.

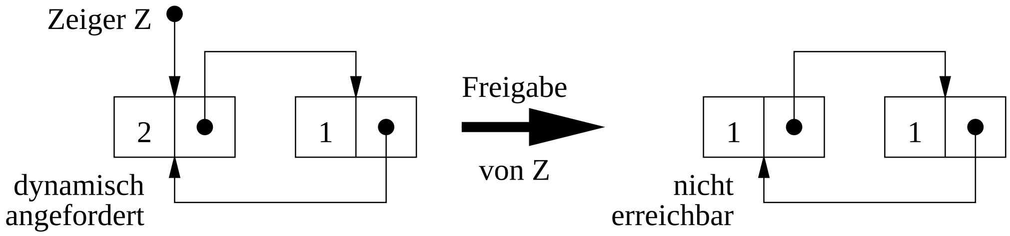
Einführung einer weiteren Indirektionsstufe. Hier verweist jeder Zeiger auf eine interne Verwaltungsstruktur, die den eigentlichen Speicherbereich adressiert. Bei der (expliziten) Speicherfreigabe wird die Adresse aus der Verwaltungsstruktur gelöscht, so dass alle zugehörigen Zeiger ungültig werden. Der Nachteil dieser Methode liegt im Zurückbleiben der Verwaltungsstruktur, die nicht mehr freigegeben werden darf. Für den konkreten Fall einer Kernimplementierung ist dieses Konzept daher ungeeignet.



Eleganter Zeiger mit weiterer Indirektionsstufe

Referenzzähler. Ein weitgehend intuitiver Ansatz ist das Speichern eines Referenzzählers zu jedem dynamisch angeforderten Speicherbereich. Der Zähler könnte beispielsweise direkt vor dem Bereich abgelegt werden, so wie die C-Standardbibliotheksfunktion `malloc()` die Länge eines angeforderten Bereichs kodiert. Mit Hilfe eines C++-Klassentemplates für Zeiger lässt sich der Zähler automatisch verwalten und Konsistenz sicherstellen. Diese Implementierung benötigt die oben eingeführte Unterscheidung zwischen Zeigern auf dynamisch angeforderte Speicherbereiche (mit Referenzzähler) und Zeigern auf globale oder automatische Variablen (ohne Zählerverwaltung).

Der Einsatz von Referenzzählern ist für zwei Konzepte denkbar. Zum Einen kann der Zählerstand lediglich benutzt werden, um die Zulässigkeit einer expliziten Freigabe zu prüfen bzw. beim Fallenlassen der letzten Referenz einen Fehler auszulösen. Ebenso kann der referenzierte Speicherbereich jedoch auch automatisch freigegeben werden, sobald der Zählerstand auf 0 zurückfällt. Gerade hier ist bei zyklischen Strukturen jedoch Vorsicht geboten, weil es sonst zu Speicherverlusten kommen kann:

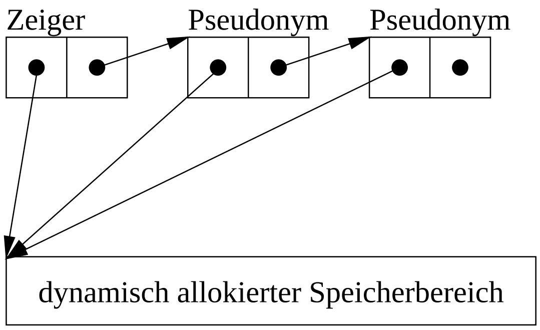


Eleganter Zeiger mit Referenzzähler

Nagle[5] verweist in diesem Zusammenhang auf ein in Perl 5 eingeführtes Modell „schwacher Zeiger" (weak pointers), ohne diesen Ansatz bis zum Ende auszuführen.

Zeiger und Pseudonyme. Hinter diesem Konzept steht die Idee einmaliger Zeiger, wodurch das Zählen der Referenzen erspart bleibt. Hier unterscheidet man Zeiger, mit denen ausschließlich dynamisch Speicher angefordert werden darf und Pseudonyme, die nur auf bereits zugeteilten Speicher verweisen können. Zu jedem Zeiger wird eine Liste der Pseudonyme verwaltet. Die Speicherfreigabe ist nur erlaubt, wenn diese Liste leer ist. Alternativ kann man bei der Freigabe auch einfach die aufgelisteten Pseudonyme annullieren.

Eine intelligente Implementierung der Listen erlaubt auch die Zuweisung eines Pseudonyms an ein Pseudonym. Sowohl Zeiger als auch Pseudonyme werden intern durch ein Zeigerpaar repräsentiert. Der jeweils Erste zeigt auf den Speicherbereich und der Zweite auf das nächste Pseudonym in der Liste:



Eleganter Zeiger mit Pseudonymen

Wie soll jedoch vorgegangen werden, wenn ein Pseudonym – beispielsweise durch eine neue Zuweisung oder Verlassen des Gültigkeitsbereichs – ungültig wird? Sicherlich wäre hier eine doppelt verkettete Liste möglich, viel interessanter erscheint aber eine Ringverkettung. Dann entfällt die Unterscheidung zwischen Zeigern und Pseudonymen.

3.3.7 Programmfluss-Steuerung

Es besteht kein Zweifel darin, dass der C++-Befehl `goto` nicht mehr dem Stand heutiger Programmiertechnik entspricht. Dennoch wird er in einigen modernen C++-Programmen verwendet. So läßt sich durch ihn das Ausbrechen (`break`) aus bzw. das erneute Aufsetzen (`continue`) in einer Schleifenstruktur über mehrere Ebenen hinweg realisieren. Java erlaubt dazu die Angabe von Marken (*labels*) bei den Schleifenkontrollanweisungen. Im Sinne einer minimalistischen Definition sollte eine Erweiterung der Syntax jedoch, wo möglich, vermieden werden. Die Semantik dieses Konstrukts lässt sich auch mit einfachen Schleifenstrukturen nachbilden, allerdings zu Lasten der Übersichtlichkeit. Es obliegt daher einer genaueren Untersuchung des vorliegenden Programmtextes, in wie weit diese Syntaxerweiterung lohnt.

Die Semantik des `switch`-Kommandos von C++ erfordert aufgrund des „Durchfallens" einen hohen Aufwand. Da praktisch in den meisten Fällen eine Befehlsfolge mit `break` abgeschlossen wird, bevor eine neue `case`-Anweisung folgt, wird dies in Safe C++ syntaktisch zwingend vorgeschrieben.

3.3.8 Weitere Besonderheiten

Zu Gunsten einer einfacheren Semantik wird in Safe C++ auf ein Konstrukt zur Ausnahmebehandlung (`try` / `catch`) verzichtet. Es ist allgemein fraglich, ob es in der Betriebssysteme-Konstruktion seiner hohen Kosten wegen überhaupt genutzt würde.

Trotz häufiger Verwendung ebenfalls entfallen muss die Möglichkeit zur Einbettung von Assembler-Quelltext. Als Ersatz dafür werden neue Sprachkonstrukte benötigt. Das folgende Kapitel wird sich diesem Thema intensiv widmen.

3.4 Fazit: Entwurfsentscheidungen für Safe C++

Zusammenfassend sollen hier noch einmal die wesentlichen Entscheidungen für den Entwurf von Safe C++ genannt werden.

Präprozessor-Anweisungen sind kein Bestandteil der Safe-C++-Sprachdefinition; der C++-Präprozessor kann vor der Übersetzung durch den Safe-C++-

Compiler zur Anwendung kommen.

Variablen werden in Safe C++ bei ihrer Definition initialisiert. Die Zahl der an eine ***Funktion*** zu übergebenden Argumente ist invariabel und in der Deklaration eindeutig festzulegen. Es können keine Vorgabewerte (*default values*) für Methodenargumente angegeben werden.

Wertebereich bzw. Zeichenvorrat und Speichergröße der primitiven ***Datentypen*** werden in der Semantikspezifikation exakt angegeben. Statt der in C++ verwendeten Typnamen werden neue Bezeichnungen eingeführt: `int8` bis `int64` für vorzeichenbehaftete Ganzzahlen, `uint8` bis `uint64` für vorzeichenlose. Der Zeichen-Typ `char` kann nur zur Speicherung von Zeichen, nicht aber von Zahlen verwendet werden. Die Gleitkommatypen `float` und `double` werden ersatzlos gestrichen. Überläufe (*roll overs*) in der Arithmetik sind unzulässig. Aufzählungstypen sind zu benennen; zwischen ihnen und Ganzzahlen wird strikt unterschieden. Zeigerarithmetik ist in Safe C++ nicht erlaubt. Für Felder gibt es eine eigene Typkategorie. Zeiger werden in 3 Kategorien eingeteilt: Zeiger auf globale, auf lokale und auf dynamische Datenstrukturen. Safe C++ trennt sich vom Konzept der Varianten.

Typumwandlungen erfolgen in Safe C++ generell explizit. `typedef` wird durch 2 neue Schlüsselwörter abgelöst: `typealias` zur Deklaration eines zusätzlichen Namens für den selben Typ und `typecreate`, um einen neuen Typ zu erstellen. Die Konvertierung von numerischen Werten in Adressen ist nicht gestattet.

Die Auswertungsreihenfolge von ***Ausdrücken*** ist festzulegen, da Seiteneffekte in Ausdrücken zugelassen werden. Auf den Komma-***Operator*** wird verzichtet.

Templates lassen sich nur zur Parametrisierung von Klassen und Funktionen verwenden; innerhalb von Klassen sind sie jedoch nicht erlaubt; spezialisierende Templates sind ebenfalls kein Sprachbestandteil.

Die ***dynamische Speicherverwaltung*** lässt sich durch eine automatische Speicherbereinigung oder elegante Zeiger typsicher implementieren. Letztgenannte Variante gestattet auch eine explizite Speicherfreigabe.

Die ***Programmfluss-Steuerung*** mittels `goto` ist nicht länger erlaubt. Ob ausgleichend eine Schleifenkontrolle über mehrere Ebenen hinweg eingeführt werden soll, muss erwogen werden. Innerhalb einer `switch`-Anweisung muss jede Befehlsfolge mit `break` abgeschlossen werden, bevor eine neue `case`-Anweisung zulässig ist.

Es gibt kein Konstukt zur Ausnahmebehandlung oder zum Einbetten von Assembler-Quelltext.

Im nächsten Kapitel bleibt zu klären, welche neuen Ausdrucksmittel für die Betriebssysteme-Konstruktion nach Streichung des Varianten-Konzepts, von Konvertierungen in Zeigertypen und eingebetteten Assembler-Bereichen einzuführen sind.

Kapitel 4

Notwendige Spracherweiterungen – Eine Analyse von Fiasco

Im folgenen sollen Spracherweiterungen von Safe C++ gegenüber C++ erarbeitet werden. Sie dienen dazu, einerseits Safe C++ optimal auf den Einsatz in einer systemnahen Programmierumgebung vorzubereiten, andererseits die notwendige Ausdrucksmächtigkeit zurück zu gewinnen, wo sie durch den Verzicht auf problematische Konstrukte von C++ verloren ging.

Um größtmögliche Praxistauglichkeit zu erreichen und den Aufwand für eine Migration nach Safe C++ besser abschätzen zu können, werden die Erweiterungen anhand der Fiasco-Quelltexte (Version „DD01" vom 10. Mai 2001) diskutiert.

4.1 Umwandlungen in Zeigertypen

Wie bereits skizziert ist der Verwaltungsaufwand für eine typsichere Konvertierung konventioneller numerischer Werte in einen Zeigertyp zu hoch. Es bleibt nur der Ausweg, solche Typkonvertierungen zu unterbinden. Doch müssen gerade in einer systemnahen Programmierumgebung in bestimmten Fällen Zeiger aus anderen Datenstrukturen gewonnen werden. Eine Ad-hoc-Anfrage liefert zunächst über 80 Stellen im Quelltext von Fiasco. Nicht selten sind die Adressen Ergebnis umfangreicher Berechnungen. Bei genauerer Betrachtung kristallisieren sich folgende Problemklassen heraus:

- Verwaltung von Seitentabellen

- Speicherzuteilung
- Grundinitialisierung des Rechners (*bootstrapping*)
- Interpretation von Zeigern aus Nutzerprogrammen beim Kerneinsprung (*trap*)
- Spezialisierende Typumwandlungen (*downcasts*)
- Hardware-Treiber
- Fehlersuche

Dazu kommen noch einige Besonderheiten in den Implementierungsdetails; unter ihnen solche, die sich durch kleine, kosmetische Eingriffe ebenso gut auch anders schreiben oder durch doppelte Datenhaltung vermeiden lassen. Ein Beispiel für den ersten Fall findet sich in `kmem_alloc::page_alloc (vm_offset_t, zero_fill_t)`. Diese Routine liefert im Erfolgsfall einen Zeiger auf die ihr übergebene Adresse und im Fehlerfall den Nullzeiger zurück. Da die Adresse bereits bekannt ist, genügt ein Boole'scher Rückgabewert. Als Beispiel für doppelte Datenhaltung kann `current_space()` dienen. Es liest die Adresse des aktuellen Seitenverzeichnisses aus dem zuständigen Register aus und gibt sie zurück. Der Registerwert ändert sich jedoch nur durch eine explizite Zuweisung. Wird der Wert zusätzlich im Speicher vorgehalten, lässt sich die Abfrage des Registers umgehen.

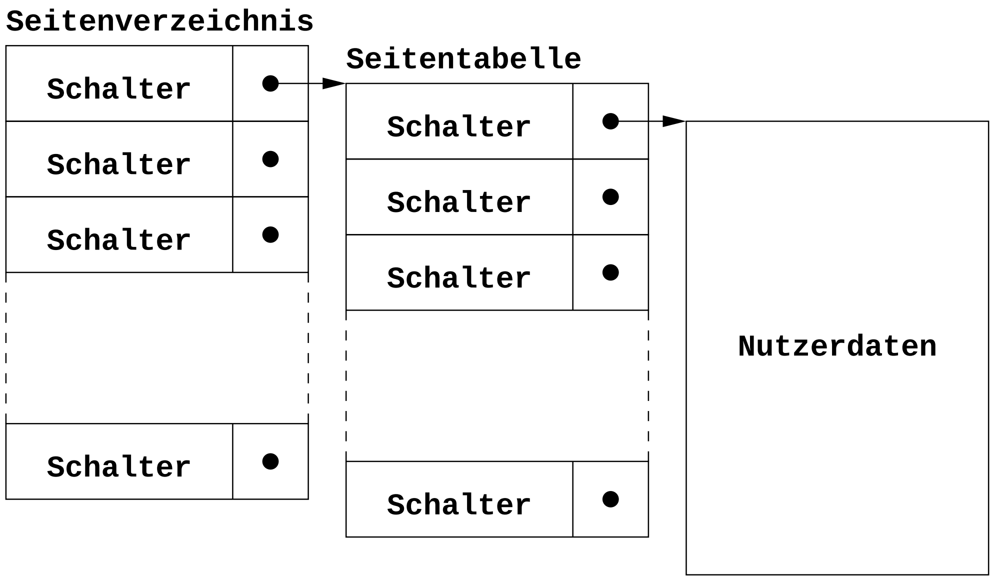
An einigen Stellen müssen die Veränderungen jedoch bereits auf konzeptioneller Ebene erfolgen. So wird es keine Routine geben können, die zu einer gegebenen, physischen Adresse einen Zeiger auf eine zugehörige virtuelle Adresse liefert. Auch werden einige Optimierungen im sicheren Kern nicht mehr vorgenommen werden können. Hier soll beispielhaft die TCB-Magie genannt werden, die unten näher erläutert wird.

Einige der oben genannten Problemklassen werden zunächst von der weiteren Betrachtung ausgeschlossen. Dies betrifft Routinen zur Fehlersuche, da sie im fertigen Kern nicht mehr verwendet werden. Ebenfalls unbeachtet bleiben Hardware-Treiber. Alle anderen Bereiche werden nun genauer diskutiert.

4.1.1 Verwaltung von Seitentabellen

Eine entscheidende Aufgabe moderner Betriebssysteme ist es, jeder im System laufenden Anwendung einen von unbefugten Beeinflussungen freien Ablauf zu gewährleisten. Zum Schutz der Anwendungen voreinander muss insbesondere der Zugriff auf gemeinsam genutzte Ressourcen wie den Speicher kontrolliert werden.

Die Intel-Architektur stellt als Speicherschutzkonzept virtuelle Adressräume und damit die Abkopplung der (virtuellen) Adressen vom (physischen) Speicher zur Verfügung. Das Betriebssystem wird damit betraut, die Adressen mit Speicher zu hinterlegen. Die notwendige Zuordnung wird in den Seitentabellen gehalten.



Jeder virtuelle Adressraum besitzt ein Seitenverzeichnis (*page directory*). Deren Speicheradresse wird in einem speziellen Register gehalten. Das Seitenverzeichnis ist eine Tabelle mit 1024 Zeilen. In jeder Zeile gibt es einige Schalter (*flags*) und einen Verweis auf die Seitentabelle (*page table*) der 2. Stufe. Diese ist in gleicher Weise aufgebaut, nur dass die Verweise hier direkt auf den Bereich der Nutzdaten gerichtet sind.

Abbildung 4.1: Organisation von Seitentabellen

Nachschlagen (*lookup*) in Seitentabellen. Der virtuelle Adressraum wird mit zweistufigen Seitentabellen verwaltet. Demzufolge steht im Seitenverzeichnis ein Verweis auf die Seitentabellen der 2. Stufe. Das Problem besteht nun darin, dass der Verweis nicht in Form eines Zeigers (also als virtuelle Adresse), sondern als physische Adresse abgelegt ist. Speicherzugriffe sind aber nur über virtuelle Adressen erlaubt. Daher wird die physische in eine virtuelle Adresse umgerechnet und in einen Zeiger umgewandelt.

Ein intuitiver Ansatz zur Lösung des Problems liegt in der Verwendung von Zeigern im Seitenverzeichnis. Dies hat aber eine doppelte Datenhaltung zur Folge, da die Speicherverwaltungseinheit (MMU – *memory management unit*) des Prozessors physische Adressen in den Verzeichniseinträgen erwartet.

Postuliert man jedoch eine in diesem Zusammenhang eineindeutige Zuordnung von physischen und virtuellen Adressen, so kann man die sehr speicheraufwendige, redundante Mitführung der virtuellen Adressen einsparen. Dieser Vorteil muss allerdings durch die Einbettung der Zuordnungsfunktion in die Sprache erkauft werden. Denkbar sind zwei Realisierungen: Eine recht offene Lösung ist eine Zeigerkategorie, die ihre Verweisadressen intern physisch ablegt. Auch eine spezielle Zeigerklasse für Seitentabelleneinträge ist vorstellbar.

Kontextwechsel. Um den Speicher der Anwendungen (*tasks*) vor unerwünschten Zugriffen anderer Anwendungen zu schützen, wird jeder Anwendung ein eigener Adressraum (Kontext) zur Verfügung gestellt. Soll der Prozessor einer anderen Anwendung überlassen werden, so muss auch der Adressraum gewechselt werden. Dies geschieht durch die Angabe eines anderen Seitenverzeichnisses im Seitenverzeichnis-Basisregister (PDBR – *page directory base register*).

Hierbei muss jedoch sichergestellt werden, dass die allen Anwendungen gemeinsamen Kernbereiche noch korrekt auf den physischen Speicher abgebildet werden. In Fiasco betrifft das alle Adressen oberhalb von 0xC0000000, ausgenommen das Bitmap mit den Ein-/Ausgaberechten (*I/O bitmap*) und die Prozesskommunikationsbereiche (*IPC windows*). Es ist dabei unproblematisch, wenn einige (benutzte) Kernbereiche in einem Kontext noch nicht zu dem betreffenden physischen Speicher zugeordnet sind, solange dies bei Bedarf durch die Seitenfehlerbehandlung nachgeholt werden kann. Unverzichtbar ist jedoch die Konsistenz der Zuordnungen. Das heißt, sobald es in einem Kontext eine Zuordnung gibt, darf kein anderer Kontext diese virtuelle Adresse einer anderen physischen Adresse zuordnen.

Aufgrund der Komplexität der Bedingungen und der kernspezifischen Organisation des Kernadressraumes lassen sich diese Bedingungen nicht durch die Sprache sicherstellen. Es kann lediglich ein Mechanismus zum Aufruf spezieller Kernroutinen bereitgestellt werden, deren Korrektheit im Zuge der allgemeinen Programmverifikation nachzuweisen ist.

Eintragen von Speicher in die Seitentabellen. Beim Einbetten von physischem Speicher in den virtuellen Adressraum muss durch die Hierarchie der Seitentabellen abgestiegen werden, um die zu verändernde Seitentabelle zu finden. Existiert noch kein Seitentabelleneintrag im Seitenverzeichnis, muss Speicher zur

Haltung der Seitentabelle angefordert werden. Das kann über die normalen Mechanismen zur Reservierung von Kernspeicher vonstatten gehen. Die Eintragung selbst erfolgt wie bereits beschrieben: entweder sowohl als physische als auch als virtuelle Adresse oder bei einer definierten Zuordnungsvorschrift nur physisch.

4.1.2 Dynamische Speicherzuteilung

C++ bietet 2 Befehle zur dynamischen Speicherverwaltung: `new` fordert neuen Speicher an und `delete` gibt ihn wieder frei. Welche Probleme sich mit der expliziten Speicherfreigabe verbinden, wurde bereits im vorangegegangenen Kapitel erläutert. Im Kontext von Fiasco stellt sich nun auch die Speicheranforderung als problematisch heraus, weil die Standardimplementierung von `new` nicht verwendet werden kann. Ebenso scheint eine Spezialimplementierung als Sprachmittel ungeeignet, da Fiasco eine umfassende Kontrolle über den Speicher benötigt.

Zur weiteren Erläuterung denke man sich den freien Speicher in einer Art „Halde" organisiert. Eine Speicherzuteilung (*memory allocation*) bedeutet nun, den geforderten Bereich der Halde zu entnehmen und ihn einer Verwendung zugänglich zu machen. Sie hat also zwei Aspekte: Die Entnahme und die Aufbereitung für die beabsichtigte Nutzung, oder genauer: die Assoziation mit einem Typ. Ein universales `new` scheitert zunächst an der Existenz mehrerer Speicherhalden und verschiedener Entnahmestrategien. Werden Routinen zur Speicherzuteilung jedoch in Safe C++ formuliert, bedürfen sie einer Sprachunterstützung: Sie müssen aus den ungetypten Adressen des von ihnen verwalteten Speichers typsichere Zeiger gewinnen können. Zudem muss sich durch die Sprache sicherstellen lassen, dass Speicher auf diese Weise nicht doppelt vergeben wird.

Fiasco kennt 7 Zuteilungsroutinen, die Speicher auf unterschiedlichen Ebenen verwalten. Nahezu alle Anforderungen werden letztlich durch den Speicherverwalter unterster Ebene befriedigt (`kmem_alloc::low_level_alloc`). Diese Beobachtung legt nahe, die Speicherverwaltung auf der niedrigsten Ebene in die Sprache einzubetten. Dadurch kann eine doppelte Speichervergabe sicher verhindert werden.

Doch kann die entgültige Typbindung erst auf der höheren Verwaltungsebene erfolgen. Nun dient ein Typ konzeptionell dazu, die Art der Verwendung zu charakterisieren. Das motiviert die Einführung eines speziellen Typs `blank`, der die Verwendung als Speicherreservoir anzeigt. Der Typ muss eine variable Länge haben[1]. Ein Speicherverwalter kann mit Speicherreservoirs operieren, in dem er

[1] Die jeweilige Länge eines `blank`-Datenobjektes muss im Speicherbereich selbst abgelegt werden. Die minimale Größe beträgt ein Byte; die Längenangaben sollen jedoch einen größeren

sie teilt (`split`) oder zwei nebeneinander liegende verbindet (`join`). Ein spezielles Sprachkonstrukt erlaubt ihm die Umwandlung eines `blank`-Zeigers in einen anderen Typ. Wie bei der Speicherfreigabe muss auch hier verhindert werden, dass es weitere Zeiger auf die betreffende Adresse gibt. Zusätzlich muss für eine Initialisierung gesorgt werden; andernfalls könnte das gespeicherte Bitmuster mit dem gewünschten, neuen Typ unvereinbar sein.

4.1.3 Grundinitialisierung des Rechners

Die Ausführung von Safe-C++-Programmen setzt einen gewissen Grundzustand des Rechners voraus. Im folgenden soll dieser näher umrissen werden. Die Korrektheit der Programmteile für die Initialisierung wird dann in einem getrennten Verfahren zu beweisen sein.

Die vorliegende Implementierung startet zunächst mit dem Ladeprozess (`bootstrap.cc`). Es wird eine 1:1-Abbildung des physischen Speichers in den virtuellen Adressraum vorgenommen und der L4-Kernprogrammcode an die Ladeadresse kopiert. Abgeschlossen wird der Vorgang mit dem Aufruf der Startroutine (`startup.cc`). Hier beginnt nun die eigentliche Initialisierung des Betriebssystems. Der gegenwärtige Prozessor wird erkannt, die Initialisierung des Kernspeicherobjekts (`kmem::init`) sowie des Speicherverwalters (`kmem_alloc::init`) angestoßen und schließlich das Hauptprogramm ausgeführt.

Im Kernspeicherobjekt sind folgende Daten einzurichten:

- Informationen zum Rechnerstart (*multiboot information*),
- die Start-Kommandozeile,
- die Hauptkopie des Kern-Seitenverzeichnisses (*master copy of kernel page directory*),
- Prozessor-Datenstrukturen
  - Interrupt-Deskriptortabelle (*IDT*),
  - Segment-Deskriptortabelle (*GDT*) und
  - Task-Zustandssegment (*TSS*)
- eine Kerninformationsseite (*kernel info page*).

Wertebereich abdecken können. Dem kann abgeholfen werden, indem in den obersten zwei Bits des ersten Bytes die Anzahl der zur Längenangabe verwendeten Bytes abgelegt wird. Auf diese Weise kann sich ein `blank`-Objekt über einen Bereich von 1 bis 2^{30} Bytes erstrecken.

Dem Speicherverwalter wird ein bestimmter Bereich des Kernspeichers zur Bedienung zukünftiger Anforderungen abgetreten.

Das Hauptprogramm ruft seinerseits die Startroutine des Kern-Threads (`kernel_thread_t::bootstrap`) auf. Sie richtet den Thread-Steuerblock und den Prozess-Kontext ein, initialisiert mathematischen Koprozessor und Interrupts und startet dann den Benutzerprozess zur Speicherseitenverwaltung (*Sigma0*) und den Boot-Prozess.

Einige Teile des geschilderten Prozesses könnten dabei bereits in Safe C++ formuliert werden. Ganz sicher gehört der Ladeprozess aber nicht dazu. Die Prozessorerkennung lässt sich entweder im Rahmen der Initialisierung des Rechners vornehmen oder muss als Sprachmittel zur Verfügung gestellt werden. Da sie jedoch nur einmal erfolgt, wird man sich für die erstgenannte Variante entscheiden.

Auch die Informationen zum Rechnerstart müssen verfügbar sein, bevor Safe C++ verwendet werden kann. Prinzipiell kann man dies einfach mittels einer globalen Variable erreichen; allerdings macht sich unter Umständen eine Anpassung der Informationsstruktur an die Safe-C++-Konventionen erforderlich.

Ebenfalls notwendig ist der Zugriff auf den physischen Speicher. Da C++ diesen nur über virtuelle Adressen erlaubt, bildet Fiasco den gesamten physischen Speicher auf virtuelle Adressen ab. Wie beschrieben erfolgt die Abbildung beim Laden aber zunächst 1:1. Erst mit dem Einrichten des Kern-Seitenverzeichnisses werden auch Adressen aus dem Kernadressbereich auf den physischen Speicher abgebildet. Die Seitenverwaltung wäre jedoch genauso in Safe C++ möglich, wenn auf den physischen Speicher bereits über die später zu verwendenden Adressen zugegriffen werden könnte. Dazu ließe sich eine globale Variable vom Typ `blank` einrichten, die den gesamten physischen Speicher ausfasst.

Ein genauer Blick auf die momentane Implementierung fördert weitere kleine Spezialprobleme zu Tage. So braucht Sigma0 die physikalische Adresse der Kerninformationsseite. Kann man dies nicht umgehen, müssen derartige Informationen ebenfalls noch vor dem Aufsetzen von Safe-C++-Programmen ermittelt und zugänglich gemacht werden.

4.1.4 Interpretation von Zeigern aus Nutzerprogrammen

Beim Kerneinsprung (*trap*) müssen häufig Zeiger aus Nutzerprogrammen im Kern interpretiert werden. An der bezeichneten Adresse stehen Daten des Nutzers für eine Weiterverarbeitung im Kern oder sollen Kerndaten für den Nutzer bereitgestellt werden. In jedem Fall ist zu prüfen, ob die angegebene Adresse tatsächlich auf Nutzerdaten zeigt. Sollen die dort befindlichen Nutzerdaten im Kern verwendet werden, müssen sie einem gültigen Bitmuster des gewünschten Typs entsprechen. Hier kann man entweder eigene Sprachmittel zur Verfügung stellen oder auf die für die Speicherverwaltung eingeführten Mechanismen zurückgreifen.

4.1.5 Spezialisierende Typumwandlungen

Fiasco ist nicht so implementiert, dass er sich auf das Laufzeitsystem von C++ stützen kann. Daraus ergeben sich einige Einschränkungen. Unter Anderem ist es nicht möglich, spezialisierende Typumwandlungen vorzunehmen. Hier kann die Typsicherheit nicht schon während der Übersetzung, sondern erst zur Laufzeit geprüft werden. In den Fiasco-Quellen tritt dieses Problem aber lediglich ein einziges Mal auf: Dort soll die Thread-Struktur zu einem gegebenen Adressraumkontext ermittelt werden. Das ließe sich durch eine zusätzliche Referenz vom Kontext auf den Thread realisieren. Es ist prinzipiell auch vorstellbar, Fiasco auf die Verwendung des Laufzeitsystems vorzubereiten.

4.1.6 TCB-Magie

Fiasco reserviert einen bestimmten Teil des virtuellen Adressraums für die Thread-Steuerblöcke (*thread control blocks*, TCBs), jedoch ohne ihn sofort mit physischem Speicher zu unterlegen. Aus dem internen Thread-Identifikator (*thread ID*) lässt sich die genaue Adresse des Steuerblocks berechnen. Beim Erzeugen eines Threads wird einfach auf diese Adresse zugegriffen. Im Regelfall ist der Adresse kein physischer Speicher zugeordnet, es kommt also zum Seitenfehler. Die allgemeine Seitenfehlerbehandlung (`thread_t::handle_page_fault`) erkennt an der Adresse, dass es sich um ein TCB handelt und hinterlegt physischen Speicher, so dass Daten dort abgelegt werden können. Diese Art der Verwaltung ist sehr platzsparend, da nicht vorhandene Threads keinen Speicherplatz benötigen, sondern lediglich einen kleinen Bereich virtuellen Adressraums beanspruchen.

Problematisch ist hier die Berechnung der Speicheradresse. Eine alternative Implementierung kann über eine Feldvariable erfolgen. Dies hat aber den Nachteil, dass die Variable bei ihrer Definition automatisch initialisiert wird. Dadurch würde sofort auch physischer Speicher hinterlegt, der mit Nullen gefüllt ist.

Man kann hier zumindest teilweise Abhilfe schaffen, indem man immer dieselbe physische Speicherseite hinterlegt. Da der gesamte Bereich den selben Inhalt hat, ist dies möglich, wenn man die Seite als nur lesbar markiert. Dann kommt es beim ersten Schreiben zu einem Seitenfehler, der durch das Hinterlegen einer anderen, exklusiv genutzten Seite aufgelöst werden kann.

Zu der gemeinsam genutzten Seite kommen noch die Kosten für den Aufbau der Seitentabellen zweiter Stufe. Der Aufwand dafür ist aber vertretbar; für den gesamten TCB-Bereich werden 128 Seitentabellen benötigt. Alternativ könnte man die verwendeten TCB's auch in einer Registrierliste führen.

Derzeit ist die TCB-Datenstruktur etwas kleiner als 2 kByte. Entscheidet man sich für die Feldvariable, sollte man sie auf exakt 2 kByte vergrößern. Andernfalls müssen beim Erzeugen eines Threads zwei neue Seiten angefordert werden, wenn sich die Seitengrenze innerhalb der TCB-Datenstruktur befindet. In der momentanen Implementierung ist die tatsächliche Länge dagegen irrelevant, da sie für die Adressberechnung nicht verwendet wird.

4.2 Eingebettete Assembler-Anweisungen

Derzeit enthalten die Fiasco-Quellen etwa 20 Stellen, an denen Assembler-Anweisungen in den Programmtext eingestreut sind. Klassifiziert man diese etwas genauer, so kann man folgende Problemzonen ausmachen, die einer Sprachunterstützung bedürfen:

Nicht-blockierende Synchronisation. Wo möglich, nutzt Fiasco nicht-blockierende Methoden zur Synchronisation. Dazu ist jedoch Hardware-Unterstützung notwendig. Zur Anwendung kommen die Befehle `compare and swap` (Vergleichen und Tauschen) und `test and set` (Testen und Setzen). Hierfür sollte Safe C++ eine Schnittstelle anbieten.

Warten auf Interrupts. Nicht immer lässt sich Synchronisation blockierungsfrei realisieren. Dann verwendet man zur Sicherung der kritischen Abschnitte auf Einprozessormaschinen gern Interruptsperren. Folgen nun mehrere solcher Abschnitte direkt hintereinander, könnte dies zu unnötig langem Blockieren der Interrupts führen. Um das zu verhindern, wird zwischen zwei direkt aufeinander folgenden kritischen Abschnitten gern zwei CPU-Takte auf Interrupts gewartet.

Thread-Erzeugung und Kontextwechsel. Bei einer Umschaltung zwischen Threads muss die jeweils alte Rechnerumgebung gespeichert und die neue geladen werden. Das betrifft in jedem Falle die Prozessorregister, bei einem Kontextwechsel kommt noch die Umschaltung des Seitenverzeichnisses hinzu.

Verwaltung der Register des mathematischen Koprozessors. Die Gleitkommaregister müssen beim Thread-Wechsel gesichert und beim Eintreten eines Interrupts für den Koprozessor entweder wieder hergestellt oder neu initialisiert werden.

Leerlauf. Befindet sich das Betriebssystem im absoluten Leerlauf (d. h. es ist kein Prozess rechenbereit oder aktiv), muss die CPU mit der `hlt`-Anweisung angehalten werden.

Speicherseitenverwaltung. In diesem Zusammenhang werden mehrere Sprachmittel benötigt. So muss bei einer Seitenfehler-Behandlung festgestellt werden können, ob der Fehler im Nutzer- oder im Kernmodus auftrat. Beim selektiven Aus-/Umtragen bestimmter Seiten aus den Seitentabellen (z. B. *Flexpages* oder Prozesskommunikationsbereiche (*IPC-Windows*)) muss der TLB invalidiert werden. In manchen kritischen Abschnitten dürfen keine Seitenfehler auftreten. Es muss sichergestellt werden, dass sich alle benötigten Seiten im physischen Speicher befinden. Im Zweifelsfall muss vor dem kritischen Abschnitt noch einmal auf die Seite zugegriffen werden, um einen eventuellen Seitenfehler zu provozieren. Dank moderner Übersetzungstechnik fallen augenscheinlich überflüssige Anweisungen jedoch der Optimierung zum Opfer. Mit Hilfe von Maschineninstruktionen kann man dennoch den gewünschten Effekt erzielen.

In der Aufstellung unberücksichtigt bleiben Routinen zur Fehlersuche und zur Rechnerinitialisierung. Darüber hinaus muss eine Stelle genannt werden, die der Umimplementierung bedarf; gleich wenn sich kein neues Sprachmittel erforderlich macht. Die globale Funktion `current()` ermittelt den aktuellen Kontext, indem sie sich den aktuellen Zeiger auf den Programmkeller (Register *esp*) holt und dessen Kontext ermittelt. Da ein Kontextwechsel jedoch immer explizit im Programm erfolgt, kann man statt des beschriebenen Vorgehens den Kontext auch extra speichern[2].

4.3 Fazit: Neue Sprachkonstrukte für Safe C++

Nachfolgend werden die für Safe C++ benötigten, zusätzlichen Sprachkonstrukte noch einmal kurz aufgelistet.

[2]Gleiches wurde im Zusammenhang mit der Funktion `current_space()` bereits für das PDBR vorgeschlagen (siehe Abschnitt 4.1 auf Seite 27).

Es wird eine Umwandlungsfunktion von physischen Adressen in Zeiger benötigt. Diese kann entweder in Form einer allgemeinen Zeigerkategorie oder als spezielle Seitentabellenzeiger implementiert werden.

Safe C++ muss es ermöglichen, das Seitenverzeichnis-Basisregister zu setzen. Die notwendige Konsistenzerhaltung des Kernadressraumes ist dabei Aufgabe des Betriebssystems, dem die Sprache eine Schnittstelle dafür zur Verfügung stellen muss.

Für die Verwendung als Speicherreservoir wird der Typ `blank` eingeführt. Dieser abstrakte Datentyp hat eine variable Länge; es sind lediglich 3 Operationen auf ihm definiert: `split` zum Teilen und `join` zum Verbinden zweier Speicherreservoirs sowie einem speziellen Sprachkonstrukt zur Umwandlung in einen anderen Typ.

Der Ladeprozess und einige Teile der Grundinitialisierung können nicht in Safe C++ vorgenommen werden.

Für die Interpretation von Zeigern aus Nutzerprogrammen können entweder eigene Sprachmittel zur Verfügung gestellt oder die für die Speicherverwaltung eingeführten Mechanismen benutzt werden.

Zu nachfolgenden Maschinenbefehlen werden Schnittstellen in Safe C++ benötigt:

- Unterstützung nicht-blockierender Synchronisation
- Warten auf Interrupts
- Umschaltung zwischen Threads (Speichern / Laden der Prozessorregister)
- Verwaltung der Register des mathematischen Koprozessors
- Leerlauf (`hlt`-Anweisung)
- Ermitteln des Orts eines Seitenfehlers (Kern oder Nutzer)
- Invalidieren des TLB
- expliziter Zugriff auf eine Speicheradresse (d. h. das Laden einer Seite in den physischen Speicher erzwingen)

Kurzreferenz Safe C++

Typsystem

Basistypen

Datentyp	Bezeichnung
Ganzzahlen mit Vorzeichen	`int8` .. `int64`
vorzeichenlose Ganzzahlen	`uint8` .. `uint64`
Zeichendatentyp	`char`
Gleitkommadatentypen	– keine –

Überläufe (*roll overs*) in der Arithmetik sind nicht definiert.

Modifizierte Typen

Zeiger.

- Unterscheidung nach Referenzierung eines globalen, automatischen oder dynamischen Objekts
- keine Zeigerarithmetik

Felder. Repräsentation als eigene Typkategorie (Implementierung als Template)

Varianten. nicht vorhanden

Typdefinitionen

Definition alternativer Typnamen mit `typealias` oder eigenständiger, neuer Typen mit `typecreate`

Aufzählungstypen.

- Benennung obligatorisch
- strikte Unterscheidung von numerischen Basistypen

Typumwandlungen

- generell explizit
- Konvertierung numerischer Werte in Adressen unzulässig

Dynamische Speicherverwaltung

typsichere Implementierung mittels automatischer Speicherbereinigung oder eleganten Zeigern

Programmfluss-Steuerung

- kein `goto`
- kein Durchfallen im `switch`-Kommando; `break` syntaktisch erforderlich
- keine Ausnahmebehandlung
- keine eingebetteten Assemblerbefehle

Spracherweiterungen und spezielle Sprachunterstützung

- Umwandlungsfunktion physischer Adressen in Zeiger
- Spezialtyp `blank`, der sich in anderen Typ umwandeln lässt
- Funktion zum Setzen des Kontexts mit Aufrufmechanismus für Kernroutinen, die Konsistenz des Kernadresraumes sicherstellen
- Anweisungen als Schnittstellen zur Maschine für:
  - nicht-blockierende Synchronisation
  - Warten auf Interrupts
  - Verwaltung der Maschinenregister
  - Leerlauf
  - Ermittlung des Orts eines Seitenfehlers
  - Invalidierung des TLB
  - expliziter Speicherzugriff (erzwungener Seitenfehler)

Literaturverzeichnis

[1] M. Hohmuth: *The Fiasco Kernel: System Architecture*, (unveröffentlicher Entwurf eines Technischen Berichtes)

[2] H. Tews, H. Härtig, M. Hohmuth: *VFiasco – Towards a Provably Correct μ–Kernel*, Januar 2001

[3] B. Ford u. a.: *The Flux Operating System Toolkit*, September 1996

[4] J. Surveyer: *C# Strikes a Chord*, Dr. Dobb's Journal, Januar 2000
URL: `http://www.ddj.com/Documents/s=875/ddj0065g/`

[5] J. Nagle: *Strict mode for C++ (Early draft proposal)*, Juli 2001
URL: `http://www.animats.com/papers/languages/index.html`

[6] D. W. Binkley: *C++ in Safety Critical Systems*, November 1995
URL: `http://hissa.ncsl.nist.gov/sw_develop/ir5769.ps`

[7] H. J. Boehm: *A garbage collector for C and C++*
URL: `http://http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

[8] Programmdokumentation (Readme) zu Hans Boehms Conservative Garbage Collector
URL: `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_source/gc.tar.gz`

[9] H. J. Boehm: *Advantages and Disadvantages of Conservative Garbage Collection*
URL: `http://www.hpl.hp.com/personal/Hans_Boehm/gc/issues.html`

[10] J. R. Ellis, D. L. Detlefs: *Safe, efficient garbage collection for C++*, Juni 1993
URL: `ftp://parcftp.xerox.com/pub/ellis/gc/gc.ps`

[11] T. M. Austin, S. E. Breach, G. S. Sohi: *Detection of All Pointer and Array Access Errors*, December 1993
URL: `ftp://ftp.cs.wisc.edu/sohi/papers/1994/pldi.safe-c.ps.gz`

[12] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, Y. Wang: *Cyclone: A safe dialect of C*, November 2001
URL: `http://www.cs.cornell.edu/projects/cyclone/papers/cyclone-safety.pdf`

[13] H. Hansen: *Vergleich Java – C++*, 2000

URL: http://www.f4.fhtw-berlin.de/people/hansen/Lehre/WS200001/OOP/Folien/Vergleich.pdf

[14] P. Pfahler: *Programmieren in C++*, 1997
URL: http://www.uni-paderborn.de/fachbereich/AG/agkastens/cplpl/folien/folien.html

[15] H. Burchardt, H. Duden: *C++/Java-Tutorium*, März 2001
URL: http://streisand.offis.uni-oldenburg.de/~amoeller/public/doc/zwischen/zwischenbericht/node16.html

[16] K. Dombrowski: *Überblick über die Sprache Java*, 1996
URL: http://www.fh-wedel.de/ si/seminare/ws96/ausarbeitung/java/java2.htm

[17] W. C. Hsieh, M. E. Fiuczynski, C. Garrett, S. Savage, D. Becker, B. N. Bershad: *Language Support for Extensible Operating Systems*

[18] E. G. Sirer, S. Savage, P. Pardyak, G. P. DeFouw, M. A. Alapat, B. N. Bershad: *Writing an Operation System with Modula-3*

[19] M. Fiuczynski, W. Hsieh, P. Pardyak, E. G. Sirer, B. N. Bershad: *Low-level Systems Programming with Modula-3*, September 1997

[20] G. Morrisett, K. Crary, N. Clew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, S. Zdancewic: *TALx86: A Realistic Typed Assembly Language*

[21] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, H. Tews: *Reasoning about Java Classes (Preliminary Report)*, April 1998