Diplomarbeit im Fach Informatik

64-Bit-Portierung des Alpha-Oberon-Systems und des Oberon-2-Compilers

vorgelegt von

Hartmut Goebel

angefertigt am

Institut für Mathematische Maschinen und Datenverarbeitung
Lehrstuhl für Programmier- und Dialogsprachen sowie ihre Compiler
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Günter Dotzel und Jörg Nilson

Beginn der Arbeit: 3. November 1997
Abgabe der Arbeit: 4. Mai 1998

Inhaltsverzeichnis

I. Aufgabenstellung **IV**

II. Eigenständigkeitserklärung, Rechtliches **IV**

III. Kurzfassung **V**

1. Einleitung **1**

2. Bildhafte Darstellung des 64-bit Adreßraums **3**
2.1. Schachbrett-Geschichte . . . 3
2.2. 123 leere Seiten . . . 3

3. Das Layout des 64-bit Adreßbereichs **4**
3.1. Das traditionelle Layout des *OpenVMS* 32-bit Adreßraums . . . 4
3.2. Das Layout des *OpenVMS Alpha* 64-bit Adreßraums . . . 5

4. Die Problemstellung **7**
4.1. Wie groß muß/darf LONGINT sein? . . . 7
4.2. 32-bit Adressen bei 64-bit Ganzzahlen . . . 8
4.3. 64-bit Adreß-Register bei 32-bit Betriebssystem . . . 8
4.4. 64-bit Betriebssystem mit einigen 32-bit Strukturen . . . 8

5. Ziele der Erweiterung **9**
5.1. Ziele der Sprach-Erweiterung . . . 9
5.2. Ziele der Compiler-Erweiterung . . . 9
5.3. Ziele der Erweiterung des Oberon-System . . . 10

6. Die Spracherweiterungen **11**
6.1. Adreß-Typen und Zeiger . . . 11
6.2. Ganzzahl-Typen . . . 12
6.3. Hexadezimal-Literale . . . 16
6.4. Set-Typen . . . 17
6.5. Sonstige Änderungen oder Erweiterungen . . . 21
6.6. Änderungen der Syntax . . . 21

7. Gemischte Verwendung von 32- und 64-bit Modulen **22**
7.1. Motivation . . . 22
7.2. Anforderungen seitens der Sprachdefinition . . . 23
7.3. Anforderungen seitens *OpenVMS* . . . 24
7.4. Variablenparameter und SYSTEM.ADR . . . 25
7.5. Zeiger als Variablenparameter . . . 25
7.6. Auswirkungen auf die Modulbibliothek . . . 26

8. Lösung in DEC C **28**

9. Erweiterungen am Compiler 30
9.1. Änderungen am Modula-2–Compiler 30
9.2. Neue Compiler-Option /PointerSize 31
9.3. Konstante Ganzzahlen 31
9.4. Handhabung der neuen Typen 31
9.5. Ganzzahl-Typen 32
9.6. Adreß-Typen und Zeiger 32
9.7. Dynamischer Speicher 33
9.8. Set-Typen 35
9.9. Behandlung von Variablenparameter 35
9.10. Symboldatei 35
9.11. Objektdateien und Oberon Load Files 38
9.12. *OpenVMS Alpha*-abhängige Erweiterungen 39
9.13. Sonstige Erweiterungen 41
9.14. Beschränkungen durch die Implementierung 41
9.15. Implementierung der geänderten Prozeduren 42
9.16. Änderung der Code-Qualität in 32-bit Modulen 45

10. Implementierung von AOS 46
10.1. Vorbereitungen bei der 32-bit Version 46
10.2. Vorgehensweise 46
10.3. Bootlader und Modullader 47
10.4. Speicherverwaltung 48
10.5. Ausnahmebehandlung 50
10.6. Ein-/Ausgabesystem (Record Management Service) 50
10.7. Schnittstelle zu X 11 51
10.8. Hausgemachte Probleme? 52
10.9. Untersuchung des Speichermehrbedarfs 52

11. Zusammenfassung 54

12. Ausblick 55
12.1. Verbesserte Speicherverwaltung 55
12.2. Erweiterung des Modula-2-Compilers 55

A. Anleitung zum Ändern von X-11-Klienten 56
A.1. Die X 11-Prozedur erwartet Variablenparameter 56
A.2. Adressen großer Datenblöcke werden übergeben 57
A.3. Die Schnittstelle ist betroffen 58

B. Glossar 59

Literaturverzeichnis 61

Abbildungsverzeichnis

1. Bisherige Ganzzahltypen in A2O und ihre Größe 1
2. Schichtenmodell für stand-alone Anwendungen und das Oberon-System . . 2
3. Layout des virtuellen 32-bit Adreßraums von *OpenVMS* 4
4. Layout des virtuellen 64-bit Adreßraums von *OpenVMS Alpha* 6
5. Die Zeiger-Typen aus SYSTEM 11
6. Die geänderten Größen der Ganzzahl-Typen 12
7. Die Ganzzahl-Typen nach der Erweiterung 13
8. Vorschlag mit einem neuen 64-bit Typ HUGEINT 13
9. Vorschlag mit Verschieben der Typ-Hierarchie 14
10. SHORT/LONG vs. SYSTEM.SHORT/SYSTEM.LONG 16
11. Die geänderten Syntax-Regeln 21
12. Kritische Stellen bei gemischter Verwendung (Beispiel) 27
13. Gemischte Verwendung in C mittels #pragma 28
14. Datenfluß der Schnittstellendefinition in C 29
15. Abbildung der Speicherverwaltungs-Routinen 33
16. Die Allokations-Routinen 34
17. Datenfluß der Schnittstellendefinition in Oberon 36
18. Speichermehrbedarf bei einigen Datenstrukturen 53

I. Aufgabenstellung

Im Rahmen der Portierung sind folgende Schritte durchzuführen:

- Entwurf und Implementierung der Spracherweiterungen und -änderungen für die 64-bit Adressierung, sowie Erweiterung des Oberon Load File Formats.

  Damit keine 32-bit Einschränkungen bei ganzzahligen Literalen und Adreßrechung (Offsets) verbleiben, muß dazu der in Modula-2 geschriebene ALPHA OBERON-2 COMPILER so geändert werden, daß intern 64-bit statt 32-bit Ganzzahlen verwendet werden. Der ALPHA MODULA-2 COMPILER ist entsprechend zu erweitern, z. B. zur Indizierung von Feldern mit 64-bit Ganzzahlen.

  Die Programmiersprache Oberon-2 soll so erweitert werden, daß möglichst keine Änderungen in bestehenden Anwendungsprogrammen beim Übergang von 32-bit zu 64-bit Adressierung notwendig sind.

- Portierung des AOS auf 64-bit Adressierung.

  Das umfaßt die 64-bit Erweiterung der *OpenVMS*-Datenstrukturen für die Eingabe/Ausgabe, die Speicherallokation, den Bootlader, den Modullader, den Garbage Collector, den Trap-/Exception-Handler und die Schnittstelle zu X11.

  Damit keinerlei 32-bit Beschränkungen verbleiben, sollen die globalen Daten, die Strings- und Literal-Programm-Sektion, der Programmcode, die Prozedur- und Typdeskriptoren im 64-bit Heap, d. h.: im sogenannten P2-Space von OpenVMS liegen.

- Untersuchung des Speichermehrbedarfs des 64-bit ALPHA OBERON SYSTEMS im Vergleich zur 32-bit Version.

II. Eigenständigkeitserklärung, Rechtliches

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Nürnberg, den 7. September 2009 Hartmut Goebel

III. Kurzfassung

Das Betriebssystem *OpenVMS Alpha* unterstützt seit Version 7.0 virtuellen Speicher mit einem Adressierungsbereich von 64 Bit – dieser wurde für Oberon/Oberon-2 und das Oberon-System nutzbar gemacht. Bislang gab es keine Implementierung von Oberon/Oberon-2 oder des Oberon-Systems, die 64-bit Zeiger oder 64-bit Ganzzahlen unterstützt.

Die Arbeit zeigt, daß der Ganzzahl-Typ LONGINT in Oberon immer die gleiche Größe haben muß wie Zeiger, um der Sprachdefinition zu genügen.

Für die Unterstützung von 64-bit Ganzzahlen und Zeigern war keine Syntax-Änderung nötig, auch die Definition der Sprache mußte hierfür nicht geändert werden.

Für die Unterstützung von 64-bit Literalen in hexadezimaler Notation war eine Syntax-Erweiterung nötig. Nur so kann unterschieden werden, ob bei vierstelligen Hexadezimalzahlen Bit 31 das Vorzeichenbit ist.

Zur Integration von 64 Bit großen Sets war ebenfalls eine Syntax-Erweiterung nötig: Bei Set-Konstruktoren kann nun der Set-Typ vorangestellt werden, vorgegeben ist der bisherige Set-Typ mit 32 Bit. Nur so kann entschieden werden, welchen Typ z. B. -{} hat.

Der Compiler wurde geändert, um per Kommandozeilen-Option in den 64-bit Modus zu wechseln. In diesem Modus haben alle neu deklarierten Zeigertypen 64 Bit, ebenso der Typ LONGINT. Die vordefinierten Prozeduren erwarten bzw. liefern weiterhin LONGINT, nun aber 64 Bit groß. Dadurch ist es möglich, die meisten Module durch lediglich neu Compilieren auf 64 Bit umzustellen.

Um 32-bit und 64-bit Modul kombinieren zu können, wurde die Sprachdefinition um einige Regeln ergänzt. Der neue Dialekt ist eine vollständige Obermenge der bisherigen Definition von Oberon/Oberon-2.

Als Praxistest für die Spracherweiterung wurde das ALPHA OBERON SYSTEM auf 64 Bit umgesetzt. Dies war größtenteils problemlos: lediglich die low-level Module mußten geändert werden, um weiterhin zur Schnittstelle von *OpenVMS* zu passen. Problematisch waren hierbei die verbliebenen Restriktionen von *OpenVMS*, insbesondere von X 11, die größere Eingriffe verursachten.

Insgesamt zeigt die Umsetzung von AOS auf 64 Bit, daß die gewählte Spracherweiterung der richtige Weg war. Insbesondere, da alle bisherigen Implementierung des Oberon-Systems inhärent davon ausgehen, daß Zeiger und Ganzzahlen 32 Bit groß sind.

Trotz kompletter Umstellung des ALPHA OBERON SYSTEMS auf 64-bit Zeiger und 64-bit Ganzzahlen, ist der Speicherbedarf nur um ca. 30% gestiegen.

SHORTINT	=	SYSTEM.SIGNED_8	8 bit
INTEGER	=	SYSTEM.SIGNED_16	16 bit
LONGINT	=	SYSTEM.SIGNED_32	32 bit
		SYSTEM.SIGNED_64	64 bit
SYSTEM.PTR	=	SYSTEM.ADDRESS_32	32 bit

Abbildung 1: Bisherige Ganzzahltypen in A2O und ihre Größe

1. Einleitung

Oberon bezeichnet sowohl eine Programmiersprache, als auch ein Betriebssystem – entwickelt von Niklaus Wirth und Jürg Gutknecht [WG92]. Als Nachfolger[1] von Pascal und Modula-2 wurde die Programmiersprache Oberon auf das Wesentliche reduziert. Hinzugekommen sind lediglich die erweiterbaren Datentypen, die für objekt-orientiertes Programmieren unerläßlich sind.

Von Hanspeter Mössenböck wurde Oberon zu Oberon-2 erweitert ([Mös91]): hinzu kamen insbesondere typgebundene Prozeduren (sog. Methoden) und dynamische Arrays. Durch diese kleine Ergänzung gewann Oberon-2 noch einmal an Mächtigkeit. Die Unterschiede zwischen den beiden Dialekten sind jedoch so gering, daß oft nur von Oberon gesprochen wird. Auch in dieser Arbeit wird nicht unterschieden, da die Unterschiede hier nicht relevant sind.

Funktionell ergänzt wird die Sprache durch das Oberon-System. Es ist sehr flexibel, mächtig und leicht zu erweitern, aber dennoch kompakt und sparsam mit Ressourcen. Die Stärke des Oberon-Systems ist jedoch nicht in der Objektorientierung im Sinne von Polymorphie zu suchen, sondern liegt in der dynamischen Erweiterbarkeit.

Die Sprachdefinition von Oberon legt die Größe der vordefinierten Typen nicht fest; aber alle bisher existierenden Compiler und Systeme gehen inhärent davon aus, daß SIZE(LONGINT) 4 Bytes groß ist, ebenso die Größe von Pointern.

Der ALPHA OBERON-2 COMPILER A2O und ALPHA OBERON SYSTEM (AOS) sollten nun erweitert werden, um 64-bit Adressen und Ganzzahlarithmetik voll zu unterstützen.

Hier noch ein kurzer Blick auf den Stand der zu erweiternden Komponenten zu Beginn der Arbeit:

A2O ist ein Oberon-2 Compiler für *OpenVMS Alpha*, entwickelt von ModulaWare, Deutschland/Frankreich. A2O unterstützt von Anfang an eingeschränkte 64-bit Ganzzahlarithmetik durch den SYSTEM-Typ SYSTEM.SIGNED_64, aber Zeiger und Adressen waren immer 32 Bit groß. Die Ganzzahltypen hatten die Größen entsprechend Abbildung 1. Die Typhierarchie der Ganzzahlen wurde um SYSTEM.SIGNED_64 erweitert:

$$\text{SHORTINT} \subseteq \text{INTEGER} \subseteq \text{LONGINT} \subseteq \text{SYSTEM.SIGNED_64}$$

[1] Das Lehrbuch zu Oberon heißt im englichen Original „Programming in Oberon", als Anspielung auf das bekannte(?) „Programming in Modula-2" (PIM) – die deutsche Übersetzung hat dagegen den Titel „Oberon – Das neue Pascal", um von der Popularität Pascals zu profitieren (und vielleicht auch um zu zeigen, daß Pascal endgültig ausgedient hat).

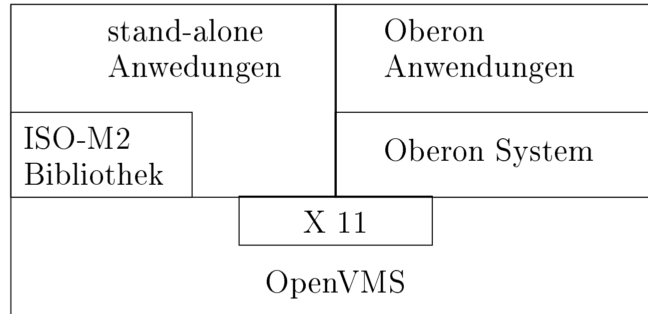

Abbildung 2: Schichtenmodell für stand-alone Anwendungen und das Oberon-System

Der Compiler kann zwei verschiedene Objektdatei-Formate erzeugen:

1. *OpenVMS*-Objektdatei-Format

   Damit können 'stand-alone' Anwendungen gebunden werden; diese benötigen das Oberon-System nicht und werden direkt aus der *OpenVMS*-Shell gestartet.

2. Oberon Load File-Format

   Der Modullader des Oberon-Systems kann dieses Format lesen und damit Module dynamisch in das Oberon-System nachladen.

Das ALPHA OBERON SYSTEM (AOS) ist die Portierung des Oberon-Systems Version 4 auf *OpenVMS Alpha*. Es handelt sich um ein komplettes, arbeitsfähiges System mit Garbage Collector, Ausnahmebehandlung, etc. Ergänzend dazu sind etliche Anwendungen verfügbar – z. B. ein Runtime-Debugger [Hof95] – die ebenfalls integriert wurden.

2. Bildhafte Darstellung des 64-bit Adreßraums

Noch vor wenigen Jahren schienen 16, später 32 Bit für die Adressierung völlig auszureichen. Inzwischen stößt sogar der „Heimanwender“ an die Grenzen dieses Bereichs, wenn sich bspw. die neue Festplatte nicht problemlos nutzen läßt. Große Datenbanken leiden schon länger unter dem Problem, daß ihr Inhalt nicht mehr komplett in den virtuellen Adreßraum von 32 Bit paßt.

Mit 32 Bit lassen sich $2^{32} \approx 4.2 \times 10^9$ Bytes = 4 GByte adressieren,[2] mit 64 Bit sind es $2^{64} = 2^{32} \times 2^{32} \approx 1.8 \times 10^{19}$ Bytes. Die gewaltige Größe des 64-bit Adreßraums läßt sich nur sehr schwer vorstellen und erfassen. Ebenso, um wieviel größer der 64-bit Adreraum im Vergleich zum 32-bit Adreßraum ist.

Darum wird hier mit zwei Analogien versucht, dies zu verdeutlichen.

2.1. Schachbrett-Geschichte

Eine bekannte Geschichte ist folgende: Ein Held hatte sich als Belohnung gewünscht, auf dem ersten Feld des Schachbretts ein Reiskorn zu bekommen und auf alle anderen jeweils die doppelte Anzahl derer, die auf dem vorherigen Feld liegen.[3] Der König willigte ein und mußte feststellen, daß es in seinem großen Reich gar nicht genug Reis gab – er mußte dem Helden das Reich übergeben.

Der 32-bit Adreßbereich entspräche dann lediglich den Reiskörnern, die auf der ersten Hälfte des Schachbretts zu liegen kämen.

2.2. 123 leere Seiten

Eine bildhafte Darstellung des Verhältnisses vom 32-bit Adreraum zum 64-bit Adreßraum ist vielleicht im Rahmen eines Kunstwerks möglich, aber nicht im Rahmen einer solchen Arbeit:

Mit einem Drucker, der 600 dpi beherrscht, soll der 32-bit Adreßraum einen Punkt groß dargestellt werden, der 64-bit Adreßraum maßstabsgerecht mit „weißen“ Punkten. Dann erhalten wir 123 *leere* Seiten und in der Mitte der 124. Seite befindet sich der Punkt für den 32-bit Adreßraum.

Der geneigte Leser möge selbst nachrechnen; der Quotient ist $2^{32} = \frac{2^{64}}{2^{32}}$, da ein Punkt 2^{32} Adreßen repräsentiert.

$$\frac{2^{32}}{(8.26 * 11.69)\, inch^2 * (600 \frac{dots}{inch})^2} \approx 123.4$$

[2] Festplatten in dieser Größe sind heute gang und gäbe

[3] Die Analogie zur Reihe der Zweier-Potenzen ist offensichtlich.

3. Das Layout des 64-bit Adreßbereichs

Hier nur ein kurzer Überblick über den *OpenVMS Alpha* virtuellen 64-bit Adreßraums [Dig96a], soweit er für diese Arbeit sinnvoll ist. Auch wenn im folgenden immer nur vom „Adreßraum“ die Rede ist, ist doch immer der *virtuelle* Adreßraum gemeint.

Vorauszuschicken ist, daß die beiden hier vorgestellten Layouts nicht die einzig richtigen sind; sie sind vielmehr die Folge der Programmierkonventionen für *OpenVMS Alpha* [Dig94]. Oder andersherum: Die Programmierkonventionen sind die Folge des gewünschten Layouts, das die Migration von 32 auf 64 Bit erleichtern soll.

3.1. Das traditionelle Layout des *OpenVMS* 32-bit Adreßraums

In *OpenVMS Alpha* vor Version 7.0 war das Layout des Adreßraums angelehnt an den 32-bit Adreßraum der VAX Prozessoren. Abbildung 3 zeigt die *OpenVMS Alpha*-Umsetzung des *OpenVMS VAX*-Layouts.

Die untere Hälfte des *OpenVMS VAX*-Adreßraums (von 0_{16} bis 7FFF FFFF_{16}) ist der 'process-private' Bereich. Dieser teilt sich zu gleichen Teilen in den 'P0'- und 'P1'-Bereich, jeweils 1 GB groß. Der 'P0'-Bereich beginnt bei Adresse 0_{16} und wächst in Richtung der höheren Adressen bis maximal 3FFF FFFF_{16}. Der 'P1'-Bereich beginnt bei Adresse 7FFF FFFF_{16} und wächst in Richtung der kleineren Adressen bis $4000\,0000_{16}$.

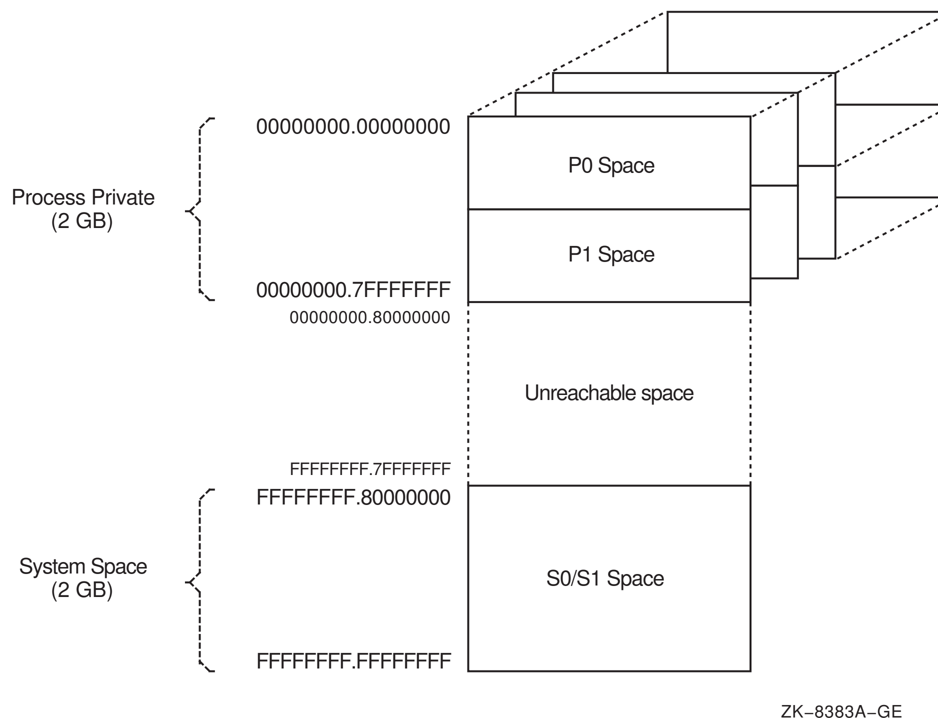

Abbildung 3: Das Layout des virtuellen 32-bit Adreßraums von *OpenVMS* bei 64-bit Adreßregistern (wie DEC Alpha) [Dig96a]

Die obere Hälfte des VAX Adreßraums ist der 'system' Bereich, den sich alle Prozesse teilen. Seine untere Hälfte (die Adressen von $8000\,0000_{16}$ bis BFFF FFFF_{16}) bildet den 'S0'-Bereich; er wächst von $8000\,0000_{16}$ in Richtung der höheren Adressen.

Dieses Layout „entsteht“ durch die Programmierkonvention, nach der 32-bit Adressen als vorzeichenbehaftete Ganzzahlen in die (64-bit) Register[4] zu laden sind. Damit wird das Vorzeichenbit (Bit 31) in die Bits 32...63 propagiert.

3.2. Das Layout des *OpenVMS Alpha* 64-bit Adreßraums

Der *OpenVMS Alpha* 64-bit Adreßraum ist eine Erweiterung des traditionellen *OpenVMS* 32-bit Adreßraums, siehe Abbildung 4.

Der neue Adreßraum besteht neben den bereits bekannten 'process-private' und 'system' Bereich noch aus dem 'page table' Bereich (PT). Von Letzerem soll hier nur erwähnt werden, daß er sich in einen prozeßeigenen und einen systemweiten Teil gliedert; er liegt auf der Grenze zwischen dem 'private' und dem 'shared' Adreßraum.

3.2.1. 'process-private' Adreßraum

Die Bereiche 'P0' und 'P1' sind analog den Bereichen 'P0' und 'P1' der VAX-Architektur definiert. Zusammen umfassen sie den traditionellen 32-bit Adreßraum von $0.0000\,0000_{16}$ bis 0.7FFF FFFF_{16}. Der 'P2' Bereich umfaßt den gesamten restlichen 'process' Bereich: Er beginnt direkt oberhalb des 'P1' Bereichs bei $0.8000\,0000_{16}$, und endet direkt unterhalb des 'process table' Bereichs.

Zu beachten ist, daß die Adreßen des 'P2' Bereichs positiv oder negativ sein können, wenn man sie als vorzeichenbehaftete 64-bit Ganzahlen betrachtet.

3.2.2. System Space

Die Bereiche 'S0' und 'S1' sind analog den Bereichen der VAX-Architektur definiert. Zusammen umfassen sie den traditionellen 32-bit Adreßraum von $\text{FFFF FFFF}.8000\,0000_{16}$ bis $\text{FFFF FFFF.FFFF FFFF}_{16}$. Der 'S2' Bereich umfaßt den gesamten restlichen 'system' Bereich zwischen der höchsten Adresse des 'process table' Bereichs und der niedrigsten Adresse des 'S1' Bereichs.

Die Bereiche 'S0', 'S1' und 'S2' werden von allen Prozessen gemeinsam benutzt; zusammen mit dem system-weiten Teil des 'process table' Bereichs bilden sie den 'shared' Bereich. Der 'S0'/'S1' Bereich dehnt sich in Richtung der höheren Adressen aus, wogegen der 'S2' Bereich grundsätzlich in Richtung der niedrigeren Adressen wächst.

Speicherbereiche innerhalb des 'system' Bereichs können nur von Programmteilen alloziert und freigegeben werden, die im Kernel-Modus laufen. Allerdings können Schreib- und/oder Lese-Zugriffe auch für weniger privilegierte Prozesse ermöglicht werden.

Die tatsächliche Größe des 'system' Bereichs wird mit einen System-Parameter festgelegt, der angibt, wieviele Megabytes für den 'S2' Bereich reserviert werden.

[4] Alle Register der Alpha-Prozessoren sind 64 Bit breit.

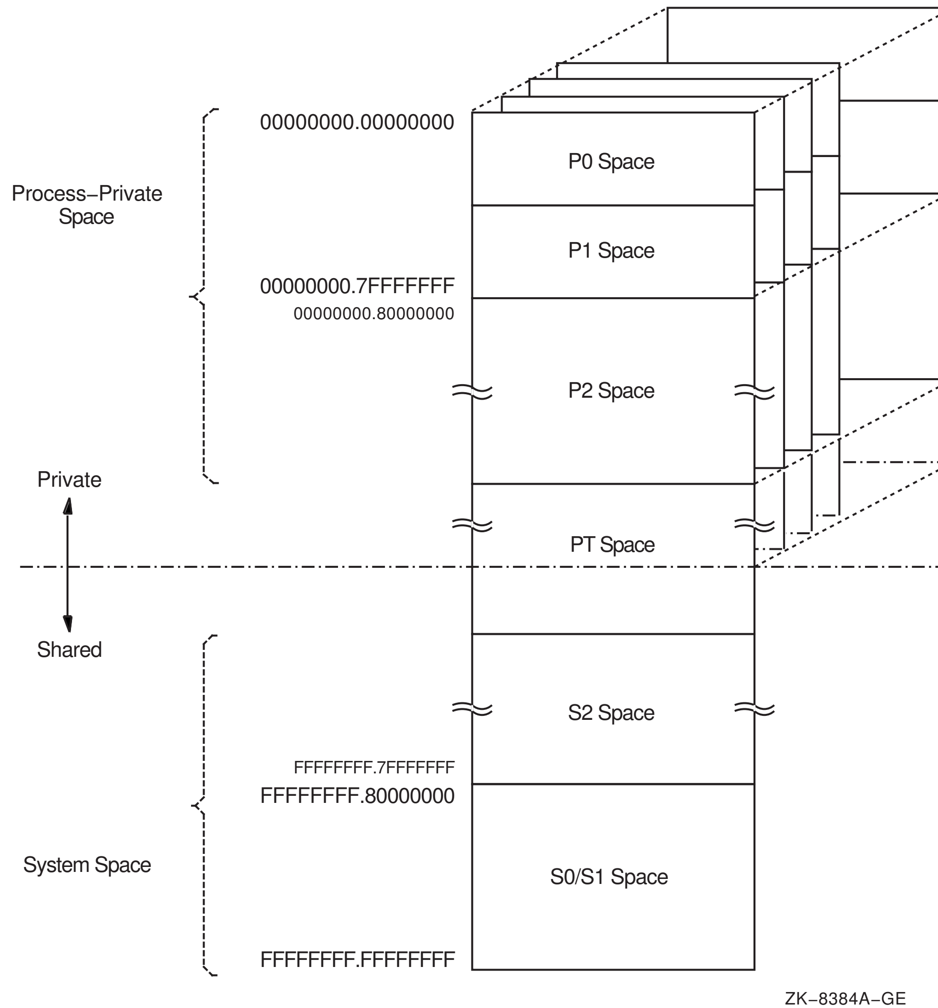

Abbildung 4: Das Layout des virtuellen 64-bit Adreßraums von *OpenVMS Alpha* [Dig96a] Die gestrichelte Linie markiert die Grenze zwischen dem 'process-private' Adreßraum und dem 'shared' Adreßbereich.

4. Die Problemstellung

OpenVMS unterstützt seit Version 7.0 einen Adressierungsbereich von 64 Bit. Dieser sollte für Oberon-2 nutzbar gemacht werden. Bislang gab es keine Implementierung des Oberon-Systems, das 64 Bit unterstützte.

Alle bisherigen Implementierungen von Oberon/Oberon-2 und des Oberon-Systems waren reine 32-bit Systeme: sie benutzten 32-bit Ganzzahlen und 32-bit Adressen. Entsprechend stand LONGINT für den 32-bit Ganzzahl-Typ, INTEGER und SHORTINT für die 16- bzw. 8-bit Ganzzahl-Typen.

Sowohl beim bei der Entwicklung der Compiler als auch des Oberon-Systems wurde diese Implikation angenommen: Konstanten wie 4, 8 oder 12, statt flexibleren Ausdrücken mit SIZE(LONGINT), finden sich an vielen low-level Stellen des Oberon-Systems, wenn auf die Größe von LONGINT Bezug genommen wird. Dies Konstanten werden verwendet, als wären die Größen der Ganzzahl-Typen auf Sprachebene definiert.

Die Sprachdefinition von Oberon macht keine Aussage über die Größe der Ganzzahl-Typen (siehe Abschnitt 4.1). Allerdings versäumt sie auch, Ganzzahl-Typen mit fester Größe im compiler-internen Modul SYSTEM zu definieren: diese könnten dann in low-level Modulen verwendet werden, wenn ein Typ einer bestimmten Größe verlangt wird.[5]

Die Problematik dieser Arbeit lag also nicht nur in der Definition einer geeigneten Spracherweiterung (siehe Abschnitt 5.1), sondern auch im Aufspüren aller kritischen Stellen im ALPHA OBERON SYSTEM.

4.1. Wie groß muß/darf LONGINT sein?

Diese Frage erscheint auf den ersten Blick verwirrend. Aber die Größe von LONGINT kann nicht beliebig gewählt werden, sondern ist eng verbunden mit der Größe der Zeiger:

Die Programmiersprache Oberon ist im Oberon-Report [Wir92] definiert. Dort wird keine Annahme über die Größe der Ganzzahl-Typen gemacht, bspw.:

> 5. LONGINT the integers between MIN(LONGINT) and MAX(LONGINT).

Allerdings definiert Abschnitt 12 des Oberon-Reports die Funktion SYSTEM.ADR: ihr Typ ist LONGINT und sie liefert die Adresse einer Variablen. Damit ist festgelegt, daß LONGINT mindestens so groß sein muß, daß eine Adresse der Zielplattform hineinpaßt.

Gemäß der Sprachdefinition wäre es erlaubt, auf einer 64-Bit-Adreß-Maschine LONGINT mit einer Größe von 128 Bit zu definieren.

Dem steht allerdings die Verwendung von LONGINT in Low-Level-Bereichen entgegen: Beim Aufbau von Laufzeitstrukturen wird häufig SYSTEM.PUT(addr, longint) verwendet. Die Prozedur SYSTEM.PUT schreibt die kompletten Daten des zweiten Parameters an die angegebene Adresse (addr im Beispiel). Wenn nun der Parameter longint eine Adresse darstellt und der Typ LONGINT wäre größer als der Adreß-Typ des Systems, dann würden zuviele Bytes geschrieben und damit evtl. andere Daten überschrieben.

[5]Dies könnte man als Indiz dafür deuten, daß auch beim Design der Sprache von 32 Bit ausgegangen wurde.

Daher ist es das einzig Sinnvolle, LONGINT genauso groß zu machen, wie der Adreß-Typ des jeweiligen Systems.

Im Folgenden werden noch einige Prozessor- und Betriebssystem-Modelle diskutiert, bei denen Ganzzahlen und Adressen unterschiedliche Größen haben.

4.2. 32-bit Adressen bei 64-bit Ganzzahlen

Leicht vorstellbar ist ein Prozessor, dessen Ganzzahl-Register zwar 64 Bit haben, dessen Adreß-Register aber nur 32 Bit groß sind. Hier hängt es von den Programmier-Konventionen des Betriebssystems und des Prozessors ab, ob LONGINT 32 oder 64 Bit groß sein muß:

Falls Adressen immer als 32-bit Werte betrachtet werden müssen, muß auch LONGINT 32 Bit groß sein. Andernfalls gäbe es die oben geschilderten Probleme mit SYSTEM.PUT. Der 64-bit Ganzzahl-Typ müßte dann die Hierarchie nach oben erweitern und sollte in SYSTEM definiert sein, um die System-Abhängigkeit zu zeigen.

Eine solche Erweiterung wurde bereits zu Anfang in A2O implementiert und ist in [Dot94] beschrieben.

Falls Adressen immer als 64-bit Werte betrachtet werden müssen, auch wenn der Prozessor nur 32 Bit unterstützt, so muß auch LONGINT 64 Bit groß sein. In diesem Fall sollten auch alle Betriebssystem-Stukturen entsprechend Platz vorsehen.

Dies entspricht dann der Erweiterung, wie sie in diesem Papier beschrieben wird.

4.3. 64-bit Adreß-Register bei 32-bit Betriebssystem

Dies entspricht der Situation von *OpenVMS Alpha* vor Version 7.0: Der Prozessor hat 64-bit Adreßregister, das Betriebssystem ist aber noch auf 32 Bit ausgelegt. Manche Strukturen sind bereits auf 64 Bit vorbereitet, es werden aber allein Adressen vergeben, die mit 32 Bit darstellbar sind.

Hier gilt das gleiche wie im ersten Fall im Abschnitt 4.2, wobei der Compiler dafür sorgen muß, daß die Adressen richtig geladen und geschrieben werden.

4.4. 64-bit Betriebssystem mit einigen 32-bit Strukturen

Dies entspricht der Situation von *OpenVMS Alpha* ab Version 7.0: Manche Strukturen enthalten Adressen mit 64 Bit, andere Adressen mit 32 Bit. Da nun auch Adressen vergeben werden, die mit 32 Bit nicht darstellbar sind, muß der Compiler (zumindest intern) zwei unterschiedlich große Adreß-/Zeiger-Typen kennen. LONGINT muß wegen SYSTEM.ADR aber 64 Bit sein.

Dies entspricht der Lösung, die in diesem Papier vorgestellt wird.

Für dieser Überlegung ist es irrelevant, ob der Prozessor 64-bit Adreßregister hat: Die Programmier-Konventionen schreiben vor, wie Adressen handzuhaben sind.

5. Ziele der Erweiterung

Soweit zur Analyse der Problemstellung. Hier nun die Ziele der Erweiterung auf 64 Bit, die sich in drei Bereiche gliedern lassen. In den folgenden Kapiteln werden die Erweiterungen dann in diesen drei Bereichen diskutiert.

5.1. Ziele der Sprach-Erweiterung

Die 64-bit Erweiterung der Sprache sollte folgenden Zielen gerecht werden:

- Keine Änderung der bisherigen Sprachdefinition:

  Bestehende 32-bit Programme müssen auch weiterhin im 32-bit Modus funktionieren.

- Möglichst wenig Erweiterungen der Sprache:

  Jede Erweiterung einer Sprache führt zu Inkompatibilitäten mit anderen Compilern.[6]

- Bestehende 32-bit Programme sollen ohne Änderungen im 64-bit Modus lauffähig sein; Änderungen in low-level Modulen sind möglichst gering zu halten.

  Damit können Anwendungen durch lediglich Neu-Kompilieren auf 64 Bit gebracht werden.

- Die Philosophie von Oberon soll erhalten bleiben, alle Änderungen müssen sich „natürlich“ in die Sprache einfügen. Insbesondere dürfen essentielle Eigenschaften von Oberon nicht verwässert werden, etwa die Typsicherheit.

- Es soll möglich sein, Module, die im 32-bit Modus kompiliert wurden, mit Modulen zu mischen, die im 64-bit Modus kompiliert wurden.

Nur wenn diese Ziele erreicht werden, können Module auf anderen Quellen leicht übernommen werden; wichtig ist dies insbesondere bei den Anwendungen des Oberon-Systems, da diese aus vielen verschiedenen Quellen gespeist werden. Müßten auch nur einige dieser Module bei jeder neuen Version für 64 Bit angepaßt werden, wäre der Wartungsaufwand immens.

5.2. Ziele der Compiler-Erweiterung

Bereits bei der Entwicklung des ALPHA OBERON-2 COMPILER A2O wurde darauf geachtet, daß er leicht auf 64 Bit erweiterbar ist [Goe96, Abschnitt 4.6]; die eigentliche Erweiterung stand aber noch aus.

[6] Turbo-Pascal ist das Parade-Beispiel hierfür.

- Volle Unterstützung von 64-bit Ganzzahl-Arithmetik.

  64-bit Ganzzahl-Arithmetik unterstützte A2O von Anfang an (mittels SYSTEM.SIGNED_64), es gab aber noch einige Einschränkungen, etwa bei MOD und DIV.

  In den Laufzeitstrukturen, etwa bei den Index-Grenzen von Arrays, waren die Felder bereits für 64 Bit vorgesehen.

- Unterstützung von 64-bit Ganzzahl-Konstanten und (dezimalen und hexadezimalen) Literalen.

- Unterstützung von 64-bit Zeigern und Adressen.

  OpenVMS Alpha verlangt schon seit der ersten Version 64-bit Adreßrechnung [Dig94], was der Compiler daher schon immer unterstützt hat: Für alle *internen* Adressen und Zeiger wurden bereits 64 Bit benutzt. Allerdings war der Typ POINTER TO Type nur 32 Bit groß, ebenso SYSTEM.PTR.[7]

Diese Punkte sollten sowohl für stand-alone Programme, als auch für die Module innerhalb von AOS erfüllt werden.

5.3. Ziele der Erweiterung des Oberon-System

- AOS soll im 64-Bit-Modus lauffähig sein, also den 64-bit Adreßbereich für Daten und Programme voll nutzen können;

- AOS soll keinerlei 32-bit Beschränkungen mehr enthalten. Alle globalen/statischen Daten, inklusive dem Programmcode, sollen im 64-bit Speicher liegen.

  Damit wäre AOS allen stand-alone Anwendungen überlegen, egal in welcher Sprache: Durch eine Beschänkung im *OpenVMS*-Linker müssen statische Daten (Programmcode, globale Variablen, Konstanten, etc.) im 32-bit Speicher liegen. Stand-Alone Anwendungen können damit den 64-bit Adreßbereich nur als dynamischen Speicher (Heap) benutzen[8] – was in der Praxis allerding keine wesentliche Einschränkung ist.

- Alle Anwendungen sollen weiterhin laufen; hierzu sollen möglichst keine Änderungen der Anwendungen nötig sein.

- Mit AOS als großer Anwendung soll die Tauglichkeit der Spracherweiterung getestet werden; insbesondere das Ziel, die meisten Module lediglich neu kompilieren zu müssen.

[7]Dies ist kein Problem, da *OpenVMS* vor Version 7.0 nur einen Speicherbereich von 32 Bit unterstützte. Entsprechend den Programmier-Konventionen wurden 32-bit Adressen vorzeichen-erweitert auf 64 Bit. [Goe96, Abschnitt 7.4]

[8]Diese Einschränkung betrifft auch AOS, allerdings nur den stand-alone gebundenen Bootloder. Die Module des Oberon-System werden von diesem dann in dynamisch allozierten Speicher geladen [Goe96]; Anwendungen innerhalb des Oberon-System unterliegen dieser Einschränkung damit nicht.

6. Die Spracherweiterungen

Nach den Überlegungen in Abschnitt 4.1 ist klar, daß die Festlegung der Größen der Ganzzahl-Typen der Ausgangspunkt der Erweiterung sein muß: die Größen der Zeiger-Typen ist implizit klar durch die Wahl des Adressierungsbereichs.

Mit der Compiler-Option /PointerSize (siehe Abschnitt 9.2) kann der Compiler in den 32-bit Modus oder den 64-bit Modus geschaltet werden. Wie der Name schon sagt, bestimmt die Option die Größe der Zeiger-Typen. Der 64-bit Modus ist nötig, da in einem 32-bit Zeiger natürlich keine 64-bit Adresse Platz hätte, der volle 64-bit Adreßbereich also nicht nutzbar wäre. Der 32-bit Modus garantiert, daß bestehende (stand-alone) Anwendungen unverändert übersetzt werden – als 32-bit Anwendungen selbstverständlich.

Zur gemischten Verwendung von 32- und 64-bit Modulen und den sich daraus ergebenden Problemstellungen siehe Kapitel 7.

In den folgenden Abschnitten werden die Erweiterungen im einzelnen diskutiert, die nötigen Änderungen am Compiler sind in Abschnitt 9.15 beschrieben.

6.1. Adreß-Typen und Zeiger

6.1.1. Größe von Adreß-Typen und Zeigern

Die Option /PointerSize bestimmt die Größe der Zeiger (siehe oben). Der generische Zeiger-Typ SYSTEM.PTR muß die entsprechende Größe haben, da an ihn jeder Zeiger zugewiesen werden kann.

Bei low-level Bereichen muß gelegentlich die Größe eines generischen Zeigers unabhängig von /PointerSize sein; dies ermöglichen die neuen Typen SYSTEM.ADDRESS_32 und SYSTEM.ADDRESS_64: Sie unterliegen den gleichen Kompatibilitäts-Regeln wie SYSTEM.PTR, haben aber fest die Größe 32 bzw. 64 Bit.[9]

Typ	Typgröße	
POINTER TO Typ	4 oder 8 Byte	abhängig von Option /PointerSize
SYSTEM.PTR	4 oder 8 Byte	abhängig von Option /PointerSize
SYSTEM.ADDRESS_32	4 Byte	
SYSTEM.ADDRESS_64	8 Byte	

Abbildung 5: Die Zeiger-Typen aus SYSTEM

6.1.2. Zusätzliche Kompatiblitätsregeln für Zeiger

Um Module kombinieren zu können, die in unterschiedlichen Modi (32- oder 64-bit) kompiliert wurden, wurden folgende Kompatiblitätsregeln für Zeiger ergänzt (siehe auch

[9] Die Namen sind Historisch entstanden: Beim Portieren von aos wurden in den Modulen entsprechende Typen als Aliase für Ganzzahl-Typen definiert, um Adressen zu kennzeichnen [Goe96, Abschnitt 7.4]. Bei der Erweiterung des Compilers wurden diese Namen einfach übernommen. Sie verdeutlichen zwar gut, daß man Variablen dieses Typs nicht dereferenzieren kann, passen aber nicht zum Bezeichner SYSTEM.PTR.

Typ	Typgröße	
SHORTINT	1 Byte	
INTEGER	2 Byte	
LONGINT	4 oder 8 Byte	abhängig von Option /POINTERSIZE

Abbildung 6: Die geänderten Größen der Ganzzahl-Typen

Abschnitt 7.2.1):

1. 32-bit Zeiger dürfen an 64-bit Zeiger zugewiesen werden, genau dann, wenn sie ungeachtet des Größenunterschieds zuweisungskompatibel sind.

2. 64-bit Zeiger dürfen *nie* an 32-bit Zeiger zugewiesen werden.

3. Bei Variablenparametern muß die Größe des aktuellen Zeiger-Typs mit der des formalen Zeiger-Typs übereinstimmen – in Ergänzung zu den anderen Regeln für Parametersubstitution.

Beim letzten Punkt muß beachtet werden, daß die Größe eines Zeigertyps nicht aus dem Programmtext ersichtlich ist: sie hängt von der Option /POINTERSIZE ab. Allerdings findet sich die Größe in der Symboldatei (siehe Abschnitt 9.10.1).

6.2. Ganzzahl-Typen

Da LONGINT die gleiche Größe haben muß wie SYSTEM.PTR (und damit alle Zeiger-Typen), ergeben sich die geänderte Typ-Größen gemäß Abbildung 6.

Wie in Abbildung 7 gut zu erkennen ist, entsteht mit /POINTERSIZE=64 ein Sprung bei den Größen der pervasiven[10] Ganzzahl-Typen: SYSTEM.SIGNED_32 fehlt. Trotzdem scheint dies die beste Lösung zu sein: eine Sprachänderung ist nicht nötig.

Zudem werden 32-bit Ganzzahlen nur benötigt,

1. für system-spezifische Datenstrukturen, die sowieso system-abhängig sind,

2. für andere Datenstrukturen, deren Ausbau nicht geändert werden kann, weil z. B. der Quelltext nicht verfügbar ist,

3. zum Lesen/Schreiben von 32-bit Ganzzahlen in Dateien, um die alte Dateistruktur beibehalten zu können,

4. um bei großen Ganzzahl-Feldern Speicher zu sparen.

Hier kann der Import des Modules SYSTEM, toleriert werden, auch wenn das Modul nicht im engeren Sinn systemabhängig ist.

Eine Möglichkeit, zu verhindern, daß SYSTEM überall importiert werden muß, wo 32-bit Ganzzahlen benötigt werden, wäre die Einführung eines neuen pervasiven Typs, etwa

[10] Die pervasiven Typen in Oberon sind: SHORTINT, INTEGER, LONGINT, REAL, LONGREAL, CHAR, BOOLEAN und SET; siehe auch Glossar.

Typ	Typgröße	pervasive Name	
		/PointerSize=32	/PointerSize=64
SYSTEM.SIGNED_8	1 Byte	SHORTINT	SHORTINT
SYSTEM.SIGNED_16	2 Byte	INTEGER	INTEGER
SYSTEM.SIGNED_32	4 Byte	LONGINT	—
SYSTEM.SIGNED_64	8 Byte	—	LONGINT

Abbildung 7: Die Ganzzahl-Typen nach der Erweiterung

MIDDLEINT = SYSTEM.SIGNED_32. Dadurch würde die Zahl der vorzunehmenden Änderungen aber auch nicht verringert, weshalb darauf verzichtet wurde, den Typ pervasiv in die Sprache aufzunehmen.

Lösungen, die keinen Sprung aufweisen, haben andere, gravierendere Nachteile (siehe unten) als dieses Problem.

Bleibt zu überlegen, ob nicht besser der 16-bit Ganzzahl-Typ ausgelassen werden sollte. Hiergegen spricht, daß INTEGER üblicherweise als der „kleine" Ganzzahl-Typ verwendet wird, für „große" Ganzzahlen wird üblicherweise LONGINT benutzt. Zudem ist kein klarer Vorteil dieser Änderung ersichtlich; da hiermit *zwei* Ganzzahltypen ihre Größe ändern würden, ist sogar mit zusätzlichen Schwierigkeiten zu rechnen.

6.2.1. Andere Vorschläge

Im folgenden werden zwei andere Vorschläge vorgestellt, wie 64-bit Ganzzahlen und 64-bit Adreßbereiche in Oberon integriert werden könnten. Die Nachteile dieser Vorschläge sind aber gewaltig, so daß die Lücke in der Typ-Hierarchie (siehe Abschnitt 6.2) als die bei weitem beste Lösung erscheint.

Neuer 64-bit Typ HUGEINT Bei diesem Vorschlag von Michael van Acken wird die Typ-Hierarchie der pervasiven Ganzzahlen nach oben erweitert; der neue, 64-bit Typ wird HUGEINT genannt, LONGINT bleibt 32 Bit. Um den unterschiedlichen Adreß-Größen gerecht zu werden, wird ein neuer Typ SYSTEM.ADDRESS eingeführt, dessen Größe von der verwendeten Compiler-Option abhängt [OOC], siehe Abbildung 8.

Dies ermöglich es, 64-bit Ganzzahlen zu nutzen, wo gewünscht, ohne bestehende 32-bit Anwendungen zu beeinträchtigen.

Typ	Typgröße	
SHORTINT	1 Byte	
INTEGER	2 Byte	
LONGINT	4 Byte	
HUGEINT	8 Byte	
SYSTEM.ADDRESS	4 oder 8 Byte	abhängig von Compiler-Option

Abbildung 8: Vorschlag mit einem neuen 64-bit Typ HUGEINT

Original-Typ	neuer Typ	Typgröße
SYSTEM.BYTE	—	1 Byte
SHORTINT	BYTE	1 Byte
INTEGER	SHORTINT	2 Byte
LONGINT	INTEGER	4 Byte
—	LONGINT	8 Byte

Abbildung 9: Vorschlag mit Verschieben der Typ-Hierarchie

Für Systeme mit 32-bit Adressen bringt die Einführung eines neuen pervasiven Typs aber keinerlei Vorteile gegenüber einem SYSTEM-Typ SYSTEM.SIGNED_64. Für Systeme mit 64-bit Adressen führt die unterschiedliche Größe von Zeigern/Adressen und LONGINT zu den in Abschnitt 4.1 beschriebenen Problemen.

Van Acken steht auf dem Standpunkt, daß das Modul SYSTEM nicht Teil der Sprachdefinition ist, sondern nur ein Anhang, und es darum für die Erweiterung uninteressant sei, wie SYSTEM.PUT und SYSTEM.GET verwendet werden [OOC]. Ich halte seinen Standpunkt aber nicht für haltbar: zum einen sind in [Mös91] auch wichtige Typ-Regeln im Anhang definiert, zum anderen sollte man bei Sprachänderungen auch die Praxis berücksichtigen.

Verschieben der Typ-Hierarchie Von Michael Grebeling stammt ein Vorschlag, alle Ganzzahl-Typen um eine „Position“ nach oben zu verschieben und dem damit freiwerdenden 1-Byte-Typ den Namen BYTE zu geben (siehe Abschnitt 9). Der Typ SYSTEM.BYTE sollte entfernt werden. [OOC]

Bei diesem Vorschlag kann zwar sehr leicht auf 64-bit Ganzzahlen umgestellt werden, aber es wurde die besondere Semantik übersehen, die SYSTEM.BYTE in Oberon spielt:

- Es ist nötig, die Sprachdefinition zu ändern.
- SYSTEM.BYTE ist *kein* Ganzzahl-Typ, sondern gehorcht speziellen Kompatibilitäts-Regeln [Wir92, Abschnitt 12]:
  - SYSTEM.BYTE ist kompatibel zu SHORTINT und CHAR.
  - Ein formaler Variablenparameter vom Typ ARRAY OF SYSTEM.BYTE ist kompatibel zu einem aktuellen Parameter beliebigen Typs.
- Schon alleine durch das Entfernen von SYSTEM.BYTE und die Einführung von BYTE wären viele Programme nicht mehr kompilierbar.
- Eine solche Änderung würde beim Benutzer für Verwirrung sorgen.

Der Vorschlag beschreibt nicht, wie die Typ-Hierarchie bei 32 Bit aussehen soll, eine sinnvolle Übertragung scheint aber nicht möglich. Man kann also davon ausgehen, daß es nicht möglich ist, Anwendungen in beiden Varianten zu nutzen.

Im Laufe der Diskussion räumte Grebeling auch ein, daß eine derart geänderte Sprache nicht mehr 'Oberon' wäre. Ihm schwebte damit eine neue Sprache 'Oberon-3' vor.

6.2.2. Weitere Konsequenzen aus der Umstellung von LONGINT

Im 64-bit Modus ist LONGINT 64 Bit groß. Als Folge dessen ändert sich die Semantik der Standard-Funktionen und -Prozeduren, die entweder ein Argument oder das Funktions-Ergebnis vom Typ LONGINT haben: Sie erwarten bzw. liefern weiterhin LONGINT, lediglich ist dieses im 64-bit Modus 64 Bit groß.

Anzumerken ist, daß diese Semantikänderung nicht an der Sprachdefintion erkenntlich ist. Dies soll sie auch nicht! Ein Programm, daß keine Annahme über die Byte-Größe der Typen macht, ist damit durch lediglich Neu-Kompilieren im 64-bit Modus lauffähig.[11]

Im folgenden werden die geänderte Typhierarchie und die Prozeduren im einzelnen beschrieben; alle Aussagen beziehen sich auf den 64-bit Modus (/PointerSize=64).

Geänderte Typhierarchie

Die Typ-Hierarchie der Ganzzahlen ist nun SHORTINT $\subseteq$ INTEGER $\subseteq$ LONGINT.[12]

Damit ergibt sich das Problem, daß LONG(longint)[13] nicht mehr zulässig ist, wofür es weder eine Lösung noch einen Workaround gibt. Da dieses Konstrukt aber *äußerst* selten ist[14], kann diese Einschränkung in Kauf genommen werden.

Entsprechend der Typ-Hierarchie und der Semantik von SHORT/LONG liefern diese:

allgemein	mit /PointerSize=32	mit /PointerSize=64
SHORT(longint): integer	SIGNED_16 ← SIGNED_32	SIGNED_16 ← SIGNED_64
LONG(integer): longint	SIGNED_16 → SIGNED_32	SIGNED_16 → SIGNED_64

Dies spiegelt den Sprung bei den Typ-Größen wieder, verbietet aber die Konversion von SYSTEM.SIGNED_64 nach SYSTEM.SIGNED_32 und umgekehrt. Darum wurden die Funktionen SYSTEM.SHORT und SYSTEM.LONG eingeführt, die im 32- und 64-bit Modus verfügbar sind.

Abbildung 10 zeigt die Konvertierungs-Hierarchie der Ganzzahl-Typen in Abhängigkeit vom Compiler-Modus:

Zusammenfassend: SHORT/LONG arbeitet auf der Typ-Hierarchie der Oberon-Ganzzahl-Typen, während SYSTEM.SHORT/SYSTEM.LONG auf der Hierarchie der Ganzzahl-Typen aus SYSTEM operiert.

Alle drei Argumentse von SYSTEM.MOVE bleiben LONGINT.

Das erste Argument (die Speicheradresse) von SYSTEM.GET, SYSTEM.PUT, und SYSTEM.BIT bleibt LONGINT.

[11] Module, die die Schnittstelle z. B. zum Betriebssystem definieren, sind hiervon natürlich ausgenommen: sie setzen ja bestimmte Typ-Größen voraus.

[12] Die bisherige Hierarchie beinhaltet noch SYSTEM.SIGNED_64, siehe Seite 2.

[13] LONG konvertiert eine Zahl in den (im Rahmen der Typhierarchie) nächstgrößeren Typ.

[14] Nur in Modulen, die den zusätzlichen Typen SYSTEM.SIGNED_64 nutzen. Und LONG wird generell kaum benutzt, da in den allermeisten Fällen die automatische Typkonvertierung genügt.

/PointerSize=32	SHORT/LONG:	SIGNED_8, _16, _32, _64
/PointerSize=64	SHORT/LONG:	SIGNED_8, _16, _64
		↖ _32 ↗
/PointerSize=32	SYSTEM.SHORT/LONG:	SIGNED_8, _16, _32, _64
/PointerSize=64	SYSTEM.SHORT/LONG:	SIGNED_8, _16, _32, _64

Abbildung 10: Die Semantik von SHORT/LONG und SYSTEM.SHORT/SYSTEM.LONG

Das Funktionsergebnis von SYSTEM.ADR bleibt LONGINT.

Im 32-bit Modus stellt ggf. ein Laufzeittest sicher, daß die Adresse in 32 Bit paßt; der Test ist z. B. nötig, wenn ein 64-bit Zeiger dereferenziert wird.[15]

Das Funktionsergebnis von ASH bleibt LONGINT.

Das Funktionsergebnis von ASH(signed64, x) ist immer SYSTEM.SIGNED_64, unabhängig von der Option /PointerSize.

Wenn beide Argumente von ASH konstante Ausdrücke sind, ist das Ergebnis eine generische Ganzzahl, deren Typ von den Wertebereichen der Typhierarchie bestimmt wird.[16]

Das Funktionsergebnis von ENTIER bleibt LONGINT.

Um von Realzahl-Typen auf SYSTEM.SIGNED_64 konverterien zu können, hatte A2O bereits eine Funktion SYSTEM.LENTIER(anyRealType): SYSTEM.SIGNED64 implementiert. Im 64-bit Modus ist ENTIER nun ein Synonym für LENTIER.

Das Funktionsergebnis von SYSTEM.ESTABLISH und SYSTEM.REVERT bleibt LONGINT.

Diese Funktionen dienen dem Installieren bzw. Entfernen von Ausnahme-Behandlungs-Prozeduren unter *OpenVMS Alpha*. Der Parameter ist die Behandlungsroutine, die installiert werden soll.

6.3. Hexadezimal-Literale

Für 64-bit Hexadezimalzahlen war es leider nötig, die Syntax zu erweitern: Der Scanner in allen 32-bit Oberon-Compilern berechnet Hexadezimal-Literale im Bereich von $8000\,0000_{16}$ bis $\mathrm{FFFF\,FFFF}_{16}$ als negative Zahlen, indem er annimmt, Bit 31 sei das Vorzeichen-Bit. 0FFFFFFFFH wird also als Konstante mit dem Wert -1 interpretiert.

Die maximale Anzahl der Hexadezimalziffern einfach von 8 auf 16 zu erhöhen scheitert: 0FFFFFFFFH würde als 4294967295 interpretiert, der Typ wäre SYSTEM.SIGNED_64. Dies würde die Verwendung bisheriger Module verhindern, bspw. bei:

```
VAR i: LONGINT; BEGIN i := 0FFFFFFFFH;
```

[15] 64-bit-Zeiger können auch im 32-bit Modus auftreten, siehe Abschnitt 7.

[16] Dies entspricht dem üblichen Verfahren bei konstanten Ausdrücken.

Daher mußte die Syntax für Hexadezimal-Literale geändert werden. Statt

```
integer ::= digit {digit} | digit {hexDigit} "H".
```

lautet sie nun

```
integer ::= digit {digit} | digit {hexDigit} ("H" | "S").
```

Das "H" kennzeichnet weiterhin 32-bit Hexadezimal-Literale, das "S" (für sedezimal) kennzeichnet 64-bit Hexadezimal-Literale (jeweils vorzeichenbehaftet).

Die Syntaxänderung ist unkritisch:

1. Sie ist verträglich mit bestehenden Anwendungen: Hexadezimalzahlen (der alten Syntax) haben weiterhin den gleichen Wert.
2. Ein gültiger Oberon-Quelltext kann (nach alter Syntax) kein Konstrukt gemäß der neuen Syntax enthalten.

Zwar können andere Compiler ein Modul, das diese Syntax-Erweiterung benutzt, nicht compilieren; dies ist aber kein Problem, da ein solches Modul 64-bit Ganzzahlen voraussetzen würde, die ein anderer Compiler ebenfalls nicht unterstützen würde.

Unter diesen Aspekten ist die Syntaxänderung hinnehmbar.

Hier noch einige Beispiele für Hexadezimal-Literale:

80000000H = MIN(SIGNED_32)	80000000S = MAX(SIGNED_32)+1
0FFFFFFFFH = -1	0FFFFFFFFS = 4294967295
	0FFFFFFFFFFFFFFFFS = -1
7FFFFFFFH = MAX(SIGNED_32)	7FFFFFFFFFFFFFFFS = MAX(SIGNED_64)

6.4. Set-Typen

6.4.1. Weshalb ein neuer Set-Typ?

Da die Alpha Prozessoren 64-bit Prozessoren sind, wäre es wünschenswert, auch SET auf 64 Bit zu erweitern. Für den Garbage Collector des AOS (siehe Abschnitt 10.4.2) ist es sogar nötig, einen 64-bit Set-Typ zu haben, um Adressen manipulieren zu können: Eine Simulation durch 32-bit Sets würde den Garbage Collector ungemein verkomplizieren und verlangsamen.

Der Oberon-Report stünde dem nicht im Wege, da er SET definiert als „die Menge der ganzen Zahlen zwischen 0 und MAX(SET)" [Wir92, Abschnitt 6.1]. Die Größe von SET generell auf 64 Elemente zu erhöhen ist nicht möglich, da low-level Module weiterhin 32-bit Sets benötigen.

Man könnte die Set-Größe aber, wie bei Adressen und Zeigern, von einer Option abhängig machen! Je nach Option wäre SET 32 oder 64 Elemente groß, Bezeichner des jeweils anderen Typs könnten nicht deklariert werden. Allerdings wäre dann eine Möglichkeit zu suchen, um 32-bit Sets mit 64-bit Sets zu kombinieren: Für low-level Module (und anderen, je nach gewählter Option) wären weiterhin 32-bit Set verfügbar, und an diese müßten 64-bit Sets zugewiesen und übergeben werden können.

Eine zuerst vielversprechend scheinende Lösung (siehe nächster Absatz), scheitert dann aber doch (siehe weiter unten).

Es seien Set das 32-bit Set und Longset das 64-bit Set. Damit definieren wir folgende Zuweisungs-Regeln:

$$\begin{aligned} \textit{longset} := \textit{set} \Rightarrow & \forall x \in \{\mathsf{0} \ldots \mathsf{MAX(Set)}\} : x \in \textit{longset} \Leftrightarrow x \in \textit{set} \\ & \wedge \forall x \in \{\mathsf{MAX(Set)+1} \ldots \mathsf{MAX(Longset)}\} : x \notin \textit{longset} \end{aligned}$$

$$\begin{aligned} \textit{set} := \textit{longset} \Rightarrow & \forall x \in \{\mathsf{0} \ldots \mathsf{MAX(Set)}\} : x \in \textit{longset} \Leftrightarrow x \in \textit{set} \\ & \wedge \forall x \in \{\mathsf{MAX(Set)+1} \ldots \mathsf{MAX(Longset)}\} : x \in \textit{longset} \rightarrow \text{Abbruch} \end{aligned}$$

Sprich: Ein kurzes Set kann immer an ein langes Set zugewiesen werden, wobei die Elemente von MAX(Set)+1 bis MAX(Longset) dann nicht im langen Set enthalten sind.

Ein langes Set kann nur dann an ein kurzes Set zugewiesen werden, wenn das lange Set keines der Elemente zwischen MAX(Set)+1 und MAX(Longset) enthält; andernfalls wird das Programm mit einem Laufzeitfehler beendet.

Der mögliche Programm-Abbruch im zweiten Fall entspräche dem Verhalten von SHORT bei Überlauf. Diese Analogie ist zwingend, will man die Typsicherheit von Oberon erhalten. Ob die Konvertierung implizit oder explizit geschehen soll, spielt hierbei keine Rolle.

Diese Überlegungen scheitern aber am Komplement von Set-Literalen[17]: Wenn der *Typ* von Set-Literaten abhängig ist von einer Compiler-Option, dann ändert sich auch der *Wert* von -{}: das eine Mal gilt $-\{\} = \{0 \ldots \mathsf{MAX(LONGSET)}\}$, das andere Mal gilt $-\{\} = \{0 \ldots \mathsf{MAX(SET)}\}$!

Dies unterscheidet die Änderung des Set-Typs von der Änderung anderer Typen: der Typ legt den Wert des Set-Komplement fest! Betrachten wir das unäre Minus:

$$\begin{aligned} -\textit{set} & \rightarrow & \{0 \ldots \mathsf{MAX(set\text{-}typ)}\} - \textit{set} \\ -\textit{int} & \rightarrow & 0 - \textit{int}; \end{aligned}$$

Damit kann – obige Zuweisungs-Regeln vorausgesetzt – in einem 64-bit Modul kein Literal -{} (oder ein anderes Komplement eines Set-Literals) an ein 32-bit Modul übergeben werden: das Set würde unzulässige Elemente enthalten und das Programm beendet. Die einzige Lösung – die Zuweisungs-Regeln hier zu lockern – widerspricht der Philosophie von Oberon.

Damit ist eine Compiler-Option zum Setzen der Set-Größe zu verwerfen. Da wir aber *zusätzlich* einen 64-bit Set-Typ benötigen, müssen wir einen zweiten Set-Typ einführen.

Diese Entscheidung ist im Rahmen der oben festgelegten Ziele zu betrachten (siehe Abschnitt 5.1); wäre die Kompatibilität zu bestehenden Programmen kein Ziel, käme man zu einem anderen Ergebnis.

[17]Bei Set-Variablen tritt dieses Problem nicht auf, da hier der Typ bereits durch den Typ der Variablen feststeht.

6.4.2. Ein neuer Set-Typ ...

Wie oben beschrieben, ist ein zweiter Set-Typ nötig; dieser wurde LONGSET genannt und enthält die ganzen Zahlen von 0 bis MAX(LONGSET) = 63. SET ist weiterhin 32 Bit groß, unabhänging von der Option /PointerSize. Parallel zu diesen Typen wurden die Typen SYSTEM.SET_32 und SYSTEM.SET_64 als Aliase definiert. Der pervasive Typ LONGSET steht auch im 32-bit Modus (/PointerSize=32) als pervasiver Typ mit 64 Elementen zur Verfügung, der Typ SYSTEM.SET_64 sowieso.

Literale und Variablen der beiden Typen sind nicht miteinander kompatibel.

Dem langen Set wurde nicht der Name SET gegeben, um Probleme von vornherein zu vermeiden, und: an der Sprache sollten möglichst wenig Änderungen gemacht werden.

Konvertierungs-Funktionen wie in Abschnitt 6.4.1 diskutiert, wurden nicht implementiert: Die Semantik von Mengen rechtfertigt keine Konvertierung zwischen Mengen als vorgegebenes Konstrukt. Sollte in Spezialfällen doch eine Konvertierung nötig sein, kann sie mit folgenden kleinen Funktionen selbst vorgenommen werden:

```
PROCEDURE Set * (ls: LONGSET): SET;
  BEGIN RETURN SYSTEM.VAL(SET, ls);   (* nimmt nur die unteren 32 Bit *)
END Set;

PROCEDURE Longset * (s: SET): LONGSET;
  (* nur sicher, weil ‘s’ auf dem Stack liegt,
   * sonst Gefahr von Access-Violation
   *)
  BEGIN RETURN SYSTEM.VAL(LONGSET, s) (* nimmt 32 Bit zuviel, *)
             * LONGSET{MAX(SET)+1 .. MAX(LONGSET)}; (* die werden gelöscht *)
END Longset;
```

6.4.3. ... macht eine Syntaxänderung nötig

Bleibt noch ein Problem zu lösen: Welchen Typ haben Set-Literale? Die Idee, wie bei Ganzzahlen einen generischen Typ einzuführen, scheitert wieder am Set-Komplement – siehe oben.

Um unterscheiden zu können, muß also der Typ bei Set-Literalen angegeben werden, wofür eine Syntaxänderung nötig war: Optional kann einem Set-Konstruktor ein Bezeichner für einen Set-Typ vorangestellt werden. Wird kein Bezeichner angegeben, so ist der Typ SET, was genau dem bisherigen Verhalten entspricht. Die Änderung ist also verträglich mit bestehenden Anwendungen.

Die EBNF-Syntax für SET wurde von

```
set ::=             "{" [Element {"," Element}] "}".
```

geändert in

```
set ::= [Qualident] "{" [Element {"," Element}] "}".
```

Hier einige Beispiele für die Deklaration von Set-Komplementen:

```
CONST set =    -{}; longset = -LONGSET{};
CONST set = -SET{}; longset = -LONGSET{};

TYPE SET = LONGSET; CONST set = -SYSTEM.SET_32{}; CONST longset = -SET{};
```

Einen Punkt gibt es hier zu beachten: durch die Deklaration TYPE SET = LONGSET ändert sich *nicht* der Typ der Set-Konstruktoren ohne vorangestellten Bezeichner; {7..9} ist weiterhin vom Typ SYSTEM.SET_32. Dies ist korrekt! Zwar legt die Redeklaration von SET nahe, daß Set-Konstruktoren anschließend auch den redeklarierten Set-Typ haben, aber:

1. Sollte ein Set-Konstruktor nach der Deklaration TYPE SET = LONGINT etwa den Typ LONGINT haben? Oder welchen anderen Typ?
2. LONGSET wird nur verwendet, wenn SET nicht groß genug ist; es wird wohl niemand generell auf LONGSET umstellen: es brächte keinerlei Vorteile.
3. Beim ersten Compilieren würden die so entstandenen Fehler moniert.

Eine andere Variante der Syntaxänderung wurde verworfen: dem Set-Konstruktor wäre ein "L" angehängt worden. Diese Änderung war mit der existierenden Compiler-Struktur aber nur mit großen Eingriffen realisierbar. Zudem ist die jetzt gewählte Lösung flexibel erweiterbar für weitere Set-Typen.

Die gewählte Syntax findet sich bereits in Modula-2 und anderen Oberon/Oberon-2 Compilern, die unterschiedlich große Set-Typen kennen, bspw. AmigaOberon [Sie]. Bei der Portierung von AOS hat sich zudem gezeigt, daß diese Syntax es ermöglicht, ohne Änderungen am Programmtext von einen Set-Typ auf den anderen zu wechseln, siehe Abschnitt 10.4.2.

6.4.4. Weshalb den neuen Set-Typ nicht aus SYSTEM?

Die Entscheidung, den neuen Set-Typ pervasiv in die Sprache aufzunehmen, ist nicht klar begründbar. Es gibt gute Gründe dafür und gute Gründe dagegen, die sich die Waage halten.

Gegen die Aufnahme des neuen Typen als Standardbezeichner spricht:

- Der Typ ist compiler- bzw. systemabhängig, gehört also in das Modul SYSTEM.
- Es ist nicht *nötig*, den Typ pervasiv in die Sprache aufzunehmen.

Für die Aufnahme des neuen Typen in die Sprache spricht:

- LONGSET ist ein abstrakter Typ wie LONGINT auch.
- Die Sprache kennt verschiedene Ganzzahl-Typen, weshalb sollte sie nicht auch verschiedene Set-Typen kennen?
- Auch andere Compiler kennen einen pervasiven Typ LONGSET, etwa AmigaOberon [Sie].

```
integer ::= digit {digit} | digit {hexDigit} "H" .
set     ::= "{" [Element {"," Element}] "}" .

integer ::= digit {digit} | digit {hexDigit} ("H" | "S") .
set     ::= [Qualident] "{" [Element {"," Element}] "}" .
```

Abbildung 11: Die geänderten Syntax-Regeln (oben alt, unten neu)

6.5. Sonstige Änderungen oder Erweiterungen

Da Prozedur-Variablen meist als Zeiger realisiert werden, ist auch bei Prozedur-Variablen und Prozedur-Typen darauf zu achten, daß ihre Größe vom Compiler-Modus abhängt – so erforderlich.

Weitere Änderungen oder Erweiterungen an der Sprache waren nicht nötig.

6.6. Änderungen der Syntax

Zur Unterstützung von 64-bit Sprachelementen als Ergänzung zu 32-bit Elementen waren lediglich zwei Syntaxänderungen nötig: für lange Hexadezimalzahlen (siehe Abschnitt 6.3) und für lange Sets (siehe Abschnitt 6.4.3). Beide Änderungen (Abbildung 11) sind verträglich mit bestehenden Anwendungen.

Die erweiterte Syntax steht im 32- und 64-bit Modus zur Verfügung. Dies ist stringent, da im 32-bit Modus auch 64-bit Dezimal-Literale und lange Sets zulässig sind.

7. Gemischte Verwendung von 32- und 64-bit Modulen

> To use 64-bit address space, some simple applications need only be recompiled for a uniform 64-bit pointer size. For example, self-contained DEC C applications that rely on only the C run-time library, without using system services or other libraries, can take this approach. Real-world applications are seldom this clean-cut, however. [BNP96]

7.1. Motivation

Die gemischte Verwendung von 32- und 64-bit Modulen (kurz: gemischte Verwendung) ist für einige Problempunkte bei der Erweiterung verantwortlich und erhöht die Komplexität der Problemstellung. Weshalb ist sie überhaupt nötig und weshalb wird sie nicht einfach weggelassen?

- Noch nicht alle Systemdienste von *OpenVMS* sind 64-bit-fähig: Allen voran kann das X Window System nicht mit 64-bit Daten umgehen.[18] Entsprechend darf die Schnittstellen-Definition[19] auch nur 32-bit Zeiger enthalten.

  Um diese Systemdienste in einer 64-bit Anwendung (wenn auch mit Einschränkungen) nutzen zu können, muß der Compiler zulassen, kurze und lange Zeiger gemischt zu verwenden.

- Die bisherige Betriebssystem-Schnittstelle von *OpenVMS* ist weiterhin gültig. Um auf sie zugreifen zu können, müssen 32-bit Zeiger auch in 64-bit Anwendungen möglich sein.

- Manche Module gehen fest von einer Zeigergröße von 32 Bit aus, etwa der Garbage Collector im Oberon-System. Die Umstellung solcher Module kann einen großen Aufwand bedeuten, der vielleicht nicht gerechtfertigt ist. Vielleicht ist die Umstellung aus anderen Gründen gar nicht möglich: Lizenzrecht, Quelltext nicht verfügbar, etc.

  Möchte (oder muß) man solche Module in einer Anwendung nutzen, die auf 64 Bit erweitert werden soll, muß es einen Weg geben, kurze und lange Zeiger zu kombinieren.

Daraus resultiert das eingangs formulierte Ziel:

> Es soll möglich sein, Module, die im 32-bit Modus kompiliert wurden, mit Modulen zu mischen, die im 64-bit Modus kompiliert wurden. (Seite 9)

Damit aus der gemischten Verwendung keine Komplikationen entstehen, müssen alle Änderungen und Erweiterungen der Sprache untersucht werden, ob und wenn ja, welche

[18] Laut Aussage des Software Support von Digital ist eine 64-bit Version von X 11 inzwischen geplant oder in Vorbereitung.

[19] Die Implementierung wird mit *OpenVMS* geliefert.

Auswirkungen sie auf die gemischte Verwendung haben. Die Ergebnisse aus den hier vorgestellten Überlegungen haben direkt Einfluß auf die Erweiterungen am Compiler (siehe Kapitel 9).

7.2. Anforderungen seitens der Sprachdefinition

Untersuchen wir zunächst die Anforderungen, die sich aus der Sprachdefinition Oberons ergeben.

7.2.1. Adreß-Typen und Zeiger

Die Größe von Zeiger-Typen in einem Modul wird mit der Option /POINTERSIZE festgelegt. Allerdings können Zeiger anderer Größe (direkt oder indirekt) importiert werden aus einem Modul, das mit einer anderen /POINTERSIZE compiliert wurde; siehe auch Abschnitt 7.4.

Daraus ergeben sich drei Anforderungen:

1. Die Symboldatei – sie beschreibt die Schnittstelle – muß die Größe der Zeiger-Typen enthalten; siehe Abschnitt 9.10.1.

   Es genügt nicht, anzugeben, mit welcher /POINTERSIZE dieses Modul compiliert wurde: das Modul kann einen Zeiger-Typen re-exportieren[20], aus einem Modul, das mit einer anderen /POINTERSIZE compiliert wurde. Damit wäre die Symboldatei mit „32 Bit“ gekennzeichnet, enthielte aber den re-exportierten 64-bit Bezeichner.

2. Der Compiler muß in der Lage sein, innerhalb eines Moduls mit unterschiedlich großen Zeigern umzugehen. Hierzu sind entsprechende Kompatibilitäts-Regeln zu entwerfen.

   Die Regeln wurden bereits in Abschnitt 6.1.2 beschrieben, sind aber eine Folge dieser Überlegungen; Abschnitt 9.6 beschreibt ihre Umsetzung.

3. Beim Allozieren von dynamischen Speicher mit `NEW(p)` oder `SYSTEM.NEW(p)` muß Speicher in einem Bereich alloziert werden, der durch den Zeiger `p` adressiert werden kann.

   Dies betrifft nur 32-bit Zeiger, da sie nicht den gesamten 64-bit Adreßraum adressieren können.

7.2.2. Ganzzahl-Typen

Die gemischte Verwendung der Ganzzahl-Typen stellt kein Problem dar: die unterschiedlichen Typen haben auch unterschiedliche Semantik (ausgedrückt durch die unterschiedlichen Bezeichner) – anders als bei Zeigern, die der gleichen Sematik unterliegen.

[20] Bspw. importiert das Modul X den Bezeichner Y.a und bietet ihn in seiner eigenen Schnittstelle (ggf. mit einem andernen Namen) wieder an.

Da die Größe der pervasiven Ganzzahl-Typen aber von /PointerSize abhängt, muß sichergestellt werden, daß für die Symboldatei eine Notation verwendet wird, die den genauen Typ (mit seiner Größe) festlegt; siehe hierzu Abschnitt 9.10.1.

Der Compiler A2O war bereits in der Lage, mit 64-bit Ganzzahlen zu arbeiten [Dot94]; diese waren auch bereits in eine erweiterte Typ-Hierarchie eingebaut. Die daraus folgenden Implikationen waren bereits beachtet worden und hielten einer Überprüfung stand.

7.2.3. Hexadezimal-Zahlen

Die Änderung der Syntax, um 64-bit Hexadezimal-Zahlen notieren zu können, hat keinerlei Auswirkung auf die gemischte Verwendung: es handelt sich lediglich um eine andere Darstellungsform von ganzen Zahlen.

Der gegenseitige Import ist ebenfalls problemlos, da 64-bit Ganzzahlen in beiden Modi zur Verfügung stehen (SYSTEM.SIGNED_64).

7.2.4. Set-Typen

Die Einführung eines zweiten, größeren Set-Typs hat ebenfalls keine Auswirkung auf die gemischte Verwendung: die Typen sind nicht miteinander kompatibel, eine Vermischung ist hier von vornherein ausgeschloßen.

Da LONGSET in beiden Modi zur Verfügung steht, ist auch der gegenseitige Import problemlos.

7.2.5. Symboldatei

Das Symboldatei-Format muß unabhängig von /PointerSize sein, aber gleichzeitig die Schnittstelle genau beschreiben. Hierzu muß es unterschiedliche Symbole für die unterschiedlich großen Zeiger geben. Auch müssen die Ganzzahl-Typen so notiert werden, daß bei gemischter Verwendung keine Verwechslung möglich ist. Die Lösung wird in Abschnitt 9.10.1 vorgestellt.

7.3. Anforderungen seitens *OpenVMS*

Die Alpha Prozessoren benutzen zur Adressierung *immer* alle 64 Bit eines Registers. Eine 32-bit Adresse wird hierbei wie eine vorzeichenbehaftete Ganzzahl betrachtet: Das Vorzeichen wird von Bit 31 nach Bit 63 propagiert, d. h. die Bits 31 bis 63 sind entweder alle 0 oder alle 1.[21] Dies ist die „kanonische“ Darstellung von Ganzzahlen und Adressen. Der Compiler ist dafür verantwortlich, 32-bit Adressen richtig zu laden. Da es sich um Konventionen des Prozessors und des Betriebssystems handelt, ist dies unabhängig vom Compiler-Modus einzuhalten. [Dig96c, Dig96a]

Das bedeutet, daß alle Adreßrechnungen in 64-bit Arithmetik gemacht werden müssen: Kopieren der Parameter, Dereferenzieren, Indizieren, et cetera. Dies hat A2O von Anfang an unterstützt.

[21] Diese Bedingung nennt sich „must be sign extended“, kurz MBSE.

In einer gemischten 32-/64-bit Umgebung können aber 64-bit Adreßwerte in ein Modul wandern, das mit /POINTERSIZE=32 compiliert wurde und mit 64-bit Adressen nicht umgehen kann. Dies kann in zwei Fällen passieren, die im Folgenden beschrieben werden.

7.4. Variablenparameter und SYSTEM.ADR

Die erste kritische Stelle ist die Verwendung von SYSTEM.ADR in einem 32-bit Modul, siehe Prozedur ptr32.nil in Abbildung 12: Zwar werden alle Adreßrechnungen in 64 Bit durchgeführt, auch wenn mit /POINTERSIZE=32 compiliert wurde, aber SYSTEM.ADR liefert LONGINT, im 32-bit Modus also SYSTEM.SIGNED_32 (nur 32 Bit).

Damit muß im 32-bit Modus die 64 Bit lange Adresse des referenzierten Objekts in 32 Bit passen, also eine gültige 32-bit Adresse sein. Dies ist sie genau dann, wenn das Vorzeichenbit von Bit 31 nach Bit 63 propagiert wurde. Um diese Bedingung sicherzustellen, muß A2O bei SYSTEM.ADR im 32-bit Modus Code erzeugen, in zwei Fällen:

1. bei der Referenzierung von formalen Variablenparametern, und
2. bei Dereferenzierung eines 64-bit Zeigers.

Ist die ermittelte Adresse keine gültige 32-bit Adresse, wird das Programm mit der Fehlernummer SS$_ARG_GTR_32_BITS[22] abgebrochen. Der Laufzeittest wird in Abschnitt 9.15.2 beschrieben.

7.5. Zeiger als Variablenparameter

Variablenparameter verwendet man üblicherweise als (Ein- und) Ausgabeparameter. Es wird nur die Adresse das aktuellen Parameters übergeben ('call by reference') und die aufgerufene Prozedur kann den Parameter dann mit Daten füllen. Hierbei müssen aktueller und formaler Typ *genau* übereinstimmen, in Oberon-2 sind nicht einmal erweiterte Records zugelassen.

Bei der gemischten Verwendung hat dies zur Folge, daß bei Zeigern als Variablenparametern die Größe relevant ist – auch wenn dies aus der Sprachdefinition heraus nicht ersichtlich ist.[23] Darum ist nur der eine Aufruf von ptr32.nil in Abbildung 12 erlaubt.[24]

Wären beide Aufrufe zulässig, würde ptr32.nil zuwenig Daten schreiben (nur 32 statt 64 Bit) und der Zeiger wäre nach der Rückkehr von dort eben *nicht* NIL. Analog würde bei der Übergabe eines 32-bit Zeigers an einen formalen 64-bit Zeigertyp zuviele Daten geschrieben und damit andere überschrieben.[25]

[22] Diese Statusnummer wurde von *OpenVMS* 7.0 eingeführt.

[23] Für LONGINTist es anders, denn da wird auf einen SYSTEM-Typ abgebildet. Aber es gibt eben keinen SYSTEM.PTR TO X oder ähnliches.

[24] Für die Übergabe eines 32-bit Zeigers an einen formalen 64-bit Zeigertyp gilt Analoges. Aber das Beispiel sollte übersichtlich bleiben.

[25] Diese Problem ließe sich nur umgehen, indem der Compiler für alle Zeiger 64 Bit allozieren würde – was in Records nicht möglich ist!

Genau genommen sind von dieser Problematik nur formale Variablenparameter des Typs SYSTEM.ADR betroffen: das sind die einzigen, an die man überhaupt andere Zeigertypen als aktuelle Parameter übergeben kann. Und da Oberon keine strukturelle Gleichheit von Typen kennt, sind in unterschiedlichen Modulen definierte Typen *nie* gleich – auch wenn beide Zeiger den gleichen Basistyp haben und beide Module mit der gleichen Option /POINTERSIZE übersetzt werden.[26]

Ansonsten sind 32-bit Module in der Lage, aktuelle Parameter aus dem gesamten 64-bit Adreßbereich als Variablenparameter zu verarbeiten: Gemäß den ProgrammierKonventionen von *OpenVMS* [Dig96b] werden alle Parameteradressen 64-bit breit übergeben und auch 64-bit breit weiterverarbeitet.

Bei Werteparametern ('call by value') ist obige Regel gelockert: hier können 32-bit Zeiger an 64-bit Zeiger übergeben werden. Der Compiler übergibt einfach den kanonischen Wert des Zeigers (in 64-bit). Da die Programmier-Konventionen von *OpenVMS* vorsehen, daß alle Parameter per Referenz übergeben werden,[27] wird in diesem Fall der Wert erst auf dem Stack abgelegt und dann die Adresse des Stacks statt der Adresse der Zeigervariablen übergeben.

7.6. Auswirkungen auf die Modulbibliothek

Bei den meisten Modulen einer Modul-Bibliothek kann man mit einer einzigen Implementierung auskommen, egal, ob der Aufrufer im 32- oder 64-bit Modus compiliert wird. Dies klappt mit allen Modulen, bei denen keine Zeiger als Variablen oder Variablenparameter in der Schnittstelle vorkommen.

Beispielsweise können Prozedur-Deklarationen wie

```
TYPE anyPointerType* = POINTER TO ...

PROCEDURE Get* (p: anyPointerType);

PROCEDURE ReadString*(VAR a: ARRAY OF CHAR);

PROCEDURE WriteString*(a: ARRAY OF CHAR);
```

aktuelle Parameter aus dem gesamten 64-bit Adreßbereich akzeptieren, wenn die Module mit /POINTERSIZE=64 compiliert wurden. Wenn sicher ist, daß das Modul intern nicht SYSTEM.ADR verwendet, genügt es sogar, es mit /POINTERSIZE=32 zu compilieren.

Dagegen muß bei Deklarationen wie

```
PROCEDURE Get* (VAR p: anyPointerType);
```

die Größe des Zeiger-Typs des aktuellen Parameters der des formalen Parameters entsprechen. Für ein Modul, dessen Schnittstelle eine solche Prozedur enthält, wären also zwei Implementierungen nötig. Allerdings sollte es hier in den meisten Fällen genügen, die betroffenen Module neu zu compilieren.

[26] Synonyme wie TYPE P = ptr32.P32 sind der selbe Typ, und haben damit auch immer die gleiche Größe.
[27] Bei Werteparametern ist dann die aufgerufene Prozedur dafür verantwortlich, eine Kopie des Wertes zu machen.

```
MODULE ptr32; (* compiliert mit /PointerSize=32 *)

  IMPORT SYSTEM;

  TYPE
    Rec * = RECORD END;
    P32 * = POINTER TO T;  (* 32 Bit groß *)

  PROCEDURE nil * (VAR p: SYSTEM.PTR); (* erwartet 32-bit Zeiger, *)
                       (* da Modul mit /PointerSize=32 compiliert *)
  BEGIN
    p := NIL;
  END nil;

  PROCEDURE addr * (VAR t: T): LONGINT;
  BEGIN
    RETURN SYSTEM.ADR(t); (* kritisch bei gemischter Verwendung *)
  END addr;

END ptr32.


MODULE ptr64; (* compiliert mit /PointerSize=64 *)

  IMPORT ptr32;

  TYPE
    P64 = POINTER TO ptr32.Rec;  (* 64 Bit groß *)

  VAR
    p32: ptr32.P32;
    p64: P64;
    i: LONGINT;

BEGIN
  NEW(p32); (* alloziert 32-bit Zeiger im 32-bit Speicherbereich *)
  NEW(p64); (* alloziert 64-bit Zeiger im 64-bit Speicherbereich *)

  ptr32.nil(p32);  (* zulässig *)
  ptr32.nil(p64);  (* nicht zulässig, da VAR-Parameter *)

  i := ptr32.addr(p64^); (* zulässig, aber Problem dort *)

END ptr64;
```

Abbildung 12: Kritische Stellen bei gemischter Verwendung (Beispiel)

```
#pragma required_pointer_size save     /* bisherige Pointersize speichern */
#pragma required_pointer_size 64       /* Pointersize auf 64 Bit setzen   */
typedef char * char_ptr64;             /* 64-bit Char-Zeiger definieren   */
#pragma required_pointer_size restore  /* Pointersize wiederherstellen    */
```

Abbildung 13: Beispiel für eine Header-Datei, die Pragmas auf einzelne Typdefintionen beschränkt [BNP96]

8. Lösung in DEC C

Abschnitt 7 beschreibt die Realisierung der gemischten Verwendung von 32- und 64-bit Zeigern bei ALPHA OBERON. Im folgenden wird die Lösung diskutiert, die Digital bei ihrem Compiler DEC C verwendet.

DEC C kennt ebenfalls eine Option /POINTERSIZE, mit der die Größe von Zeigern für ein Compilat auf 32 oder 64 Bit gesetzt werden kann. Zusätzlich unterstützt DEC C Pragma-Anweisungen (siehe Abbildung 13), um die Zeigergröße für einzelne Typdefinitionen einzustellen.

Der Vorteil liegt klar auf der Hand: innerhalb eines Modules (in C genauer: Quelltextes) können unterschiedliche Zeigergrößen leicht realisiert werden – A2O benötigt hierfür verschiedene Module, die mit unterschiedlichem /POINTERSIZE compiliert werden.

Trotzdem ist die Vorgehensweise von A2O der von DEC C überlegen:

Header-Dateien werden bei einer Include-Anweisung textuell in den C-Quelltext eingefügt[28] (siehe Abbildung 14), die Optionen zu diesem Zeitpunkt schlagen sich auf die Schnittstellendefinition nieder. Dies führt zu der abstrusen Situation, daß die Schnittstelle nicht von deren Anbieter bestimmt wird, sondern vom Klienten.

Um diesen Effekt bei den Zeigergrößen zu vermeiden, müssen alle Header-Dateien mit den in Abbildung 13 gezeigten Pragma-Anweisungen umgeben werden. Ein Beispiel hierfür findet sich in [Smi96]. Wird ein Modul später auf eine andere Zeigergröße umgestellt, genügt es nicht, das Modul neu zu kompilieren: auch die Header-Datei muß angepaßt werden.

Dadurch schleichen sich leicht Inkonsistenzen (sprich Fehler) ein, die frühestens beim Linken bemerkt werden. Würden C-Compiler auch Symboldateien verwenden, würden diese Fehler bereits beim Compilieren moniert.

[28] In Oberon wird die Schnittstellenbeschreibung in Form von Symboldateien gelesen, siehe Abschnitt 9.10.

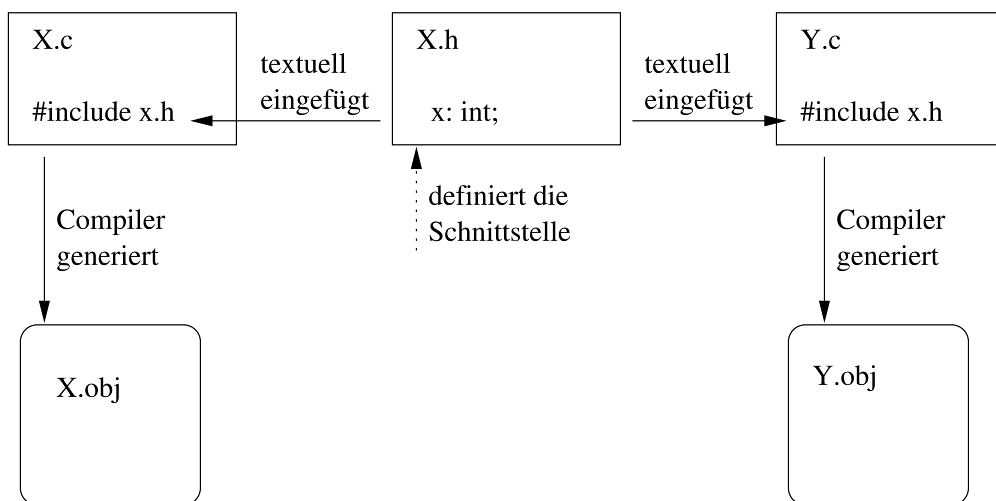

Abbildung 14: Datenfluß der Schnittstellendefinition in C

9. Erweiterungen am Compiler

Um die in Abschnitt 6 beschriebenen Erweiterungen zu implementieren, waren einige Änderungen am ALPHA OBERON-2 COMPILER A2O nötig. Diese werden im Folgenden diskutiert.

9.1. Änderungen am Modula-2–Compiler

Der ALPHA OBERON-2 COMPILER A2O ist in Modula-2 geschrieben. Um in A2O intern mit 64 Bit rechnen zu können, mußte der Modula-2-Compiler MAX entsprechend erweitert werden. Dieser unterstützt nun 64-bit Arithmetik durch den Typ SYSTEM.SIGNED_64.

Da Modula-2 nur einen vorzeichen-behafteten Ganzzahl-Typ (INTEGER) kennt, wurde kein pervasiver 64-bit Ganzzahl-Typ eingebaut, die beiden Ganzzahltypen sind auch nicht kompatibel. Es wurde allerdings parallel zum vorzeichenlosen Ganzzahl-Typ CARDINAL ein vorzeichenloser 64-bit Typ SYSTEM.UNSIGNED_64 eingeführt.

Probleme gab es an einigen Stellen, an denen ein generischer Ganzzahltyp geholfen hätte, wie er in Oberon verwendet wird:

- bei Ausdrücken der Form $-3 + a64bitIntVar$ oder $-(-5)$ (auf Grund des internen Aufbaus des Compilers);
- bei Zuweisungen von 32-bit Ganzzahl-Literalen oder -Konstanten an 64-bit Ganzzahlen;[29]
- bei Set-Konstruktoren von Sets über Ganzzahltypen[30], hier muß {7 .. a64bitIntVar} erlaubt sein; ebenso muß beim zweiten Parameter von INCL und EXCL eine 64-bit Variable zulässig sein.

Erschwert wurde die Problematik dadurch, daß Modula-2 auch einen vorzeichenlosen Ganzzahltyp kennt und der Modula-2-Compiler intern einen weiteren Typen für konstante Ganzzahlen verwendet, die im Intervall [0 . . . MAX(INTEGER)] liegen: diese sind kompatibel zu CARDINAL und INTEGER.

Es wurde *kein* 64-bit Modus eingebaut, wie ihn A2O nun unterstützt: Alle Zeiger sind weiterhin 32 Bit, ebenso INTEGER und CARDINAL. Insgesamt wurden nur die nötigsten Änderungen gemacht, um die Stabilität des Produkts nicht zu gefährden.

Offen bleibt, ob es einfacher gewesen wäre, MAX intern komplett auf 64-bit Ganzzahlen umzustellen und INTEGER und CARDINAL dann als Unterbereichstypen von SYSTEM.SIGNED_64 bzw. SYSTEM.UNSIGNED_64 zu definieren. Dies hätte aber zu anderen, evtl. komplexeren Problemen führen können, z. B. beim Erzeugen der Symboldateien.

[29] MAX unterstützt noch keinen 64-bit Konstanten.

[30] Modula-2 kennt – im Gegensatz zu Oberon – Sets über Typen; dies können Ganzzahl-, Unterbereichs- oder Aufzählungstypen sein.

9.2. Neue Compiler-Option /PointerSize

Um den Compiler in den 32- oder 64-bit Modus schalten zu können, hat A2O eine neue Option /POINTERSIZE=*N*, deren Wert die Größe der Zeigertypen bestimmt. Zulässige Werte sind 32 und 64, vorgegeben ist 32.

Der Modus entscheidet über die Größe der Zeiger und damit über die Größe von LONGINT – und in Folge über die Semantik der in Abschnitt 6.2.2 beschriebenen Prozeduren.

9.3. Konstante Ganzzahlen

Intern wurde der Compiler so umgestellt, daß alle Ganzzahl-Berechnungen mit 64 Bit durchgeführt werden, wofür einige Änderungen am Modula-2 Complier MAX nötig waren (siehe Abschnitt 9.1).

Erste Versuche, nur einige wenige Datenfelder zu ändern, ließen viele Folge-Änderungen erwarten. Darum schien der einfachste Weg zu sein, den Compiler gleich komplett umzustellen.

Damit gibt es – bis auf wenige Ausnahmen, siehe Abschnitt 9.14.2 – keine Restriktionen auf 32 Bit mehr.

9.3.1. Ganzzahl-Literale

Der Zahlenbereich für Ganzzahl-Literale wurde – unabhängig von der Option /POINTERSIZE – auf [MIN(SIGNED_64)...MAX(SIGNED_64)] erweitert.

Das Symboldatei-Format wurde erweitert, damit auch 64-bit Konstanten exportiert werden können.

9.3.2. Hexadezimal-Zahlen

Der Scanner des Compilers wurde erweitert, um 64-bit Hexadezimal-Zahlen gemäß der neuen Syntax verarbeiten zu können. Die Änderung betraf nur den kleinen Teil in der Prozedur Number, der Hexadezimal-Zahlen interpretiert.

Da die hexadezimale Darstellung nur eine Darstellungsform von Ganzzahlen ist, wurden keine anderen Teile des Compilers beeinflußt.

9.4. Handhabung der neuen Typen

Hier muß etwas ausgeholt werden:

Ein Compiler bildet die abstrakten Datentypen der Sprache auf konkrete Datentypen des Prozessors[31] ab. Diese Abbildung hängt unter Umständen von einigen Parametern ab. So kann man bspw. bei A2O wählen, ob als Fließkommatypen IEEE- oder VAX-Format benutzt werde soll, oder eben wie groß LONGINT ist. Intern muß der Compiler alle diese Typen kennen und unterscheiden können. Hier betrachten wir nur die Ganzzahltypen, für die Fließkommatypen gilt entsprechendes.

[31] oder der virtuellen Maschine

A2O kennt vier Ganzzahltypen, die alle aus SYSTEM importiert werden können: SIGNED_8, SIGNED_16, SIGNED_32 und SIGNED_64, wobei die Nummer die Anzahl der Bits angibt. SHORTINT und INTEGER werden immer auf SIGNED_8 bzw. SIGNED_16 abgebildet, die Zuordnung von LONGINT auf SIGNED_32 bzw. SIGNED_64 geschieht in Abhängigkeit von /POINTERSIZE.[32]

Die Abbildungsrichtung hat noch einen (gewünschten) Nebeneffekt: Der Compiler kann verschiedene Listen erzeugen (etwa zur Analyse des Syntax-Baums) in denen damit immer die konkreten SYSTEM-Typen genannt werden.

Würden dagegen die SYSTEM-Typen auf die pervasiven Typen abgebildet, gäbe es Probleme:

- drei der Ganzzahltypen würden mit ihrem pervasiven Bezeichner benannt, der vierte (je nach /POINTERSIZE SIGNED_32 oder SIGNED_64) als SYSTEM-Typ.
- Da die Abbildung von /POINTERSIZE abhängt, wäre Verwirrung vorprogrammiert, sobald man 32- und 64-bit Module mischt.
- Es wäre nicht mehr klar, für welchen Typ der Bezeichner „LONGINT“ steht: auch die Ausgabe der Typbezeichner hinge von /POINTERSIZE ab.[33]

9.5. Ganzzahl-Typen

Als Konsequenz der Umstellung haben einige Routinen eine geänderte Semantik (siehe auch Abschnitt 6.2.2), die der Compiler beherrschen muß. Hierzu wurden gezielt alle Regeln untersucht, die die Ganzzahl- oder (generischen) Zeiger-Typen betreffen. Zum Auffinden dieser Regeln wurde der gesamte Quelltext des Compilers auf entsprechende Bezeichner hin durchsucht.

Im Frontend mußten hauptsächlich die Kompatibilitätsregeln angepaßt werden, teilweise in Abhängigkeit von /POINTERSIZE. Da das Backend bereits von Anfang an auf 64 Bit ausgelegt bzw. dafür vorbereitet war, hielt sich die Anzahl der Änderungen auch hier in Grenzen.

Um ständige Unterscheidungen nach /POINTERSIZE zu umgehen, wurde eine neue Variable adrinttyp eingeführt: Sie enthält je nach /POINTERSIZE einen Zeiger auf SYSTEM.SIGNED_32 oder SYSTEM.SIGNED_64. Dies macht viele Stellen nicht nur effizienter, sondern auch wesentlich lesbarer.

9.6. Adreß-Typen und Zeiger

Im 32-bit Modus (/POINTERSIZE=32) haben alle neu deklarierten Zeiger-Typen eine Größe von 32 Bit, können also nur 32-bit Adressen aufnehmen. Im 64-bit Modus

[32] Analog gilt dies – und die folgenden Ausführungen – für SYSTEM.PTR und die Abbildung auf SYSTEM.ADDRESS_32 bzw. SysAdr64.

[33] Beim Modula-2-Compiler MAX führte genau dies vor einigen Jahren zu Problemen: MAX schreibt die Typbezeichner in die Symboldatei; beim Vervollständigen der Fließkommatypen war der Bezeichner REAL aber doppelt belegt. A2O hat dieses Problem nicht, da die Symboldateien grundsätzlich anders organisiert sind (siehe Abschnitt 9.10.1).

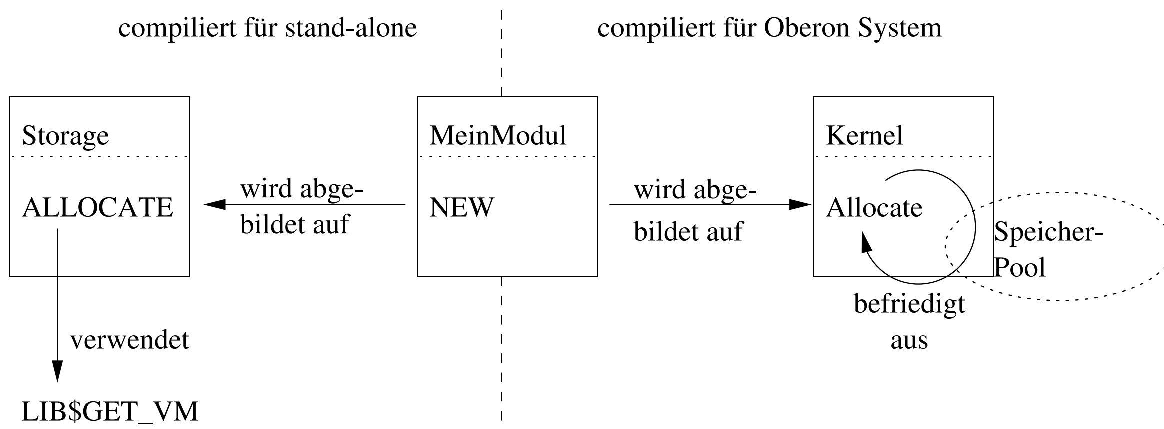

Abbildung 15: Abbildung der Speicherverwaltungs-Routinen

(/PointerSize=64) sind alle neu deklarierten Zeiger-Typen 64 Bit groß, können also Adressen des gesamten 64-bit Adreßbereichs aufnehmen.

Unter „neu deklariert“ ist hier eine Typ-Deklaration der Form TYPE X = POINTER TO XDesc; zu verstehen. Bei einer Deklaration der Form TYPE Y = X; wird in *diesem* Sinne kein neuer Typ deklariert[34], da Y nur ein Synonym für X ist.

Die Größe des Typen SYSTEM.PTR ist ebenfalls von /PointerSize abhängig. Als generische Zeiger-Typen mit fester Größe wurden SYSTEM.ADDRESS_32 (32 Bit) und SYSTEM.ADDRESS_64 (64 Bit) definiert, die in beiden Compiler-Modi zur Verfügung stehen – also unabhängig von /PointerSizesind.

Analog zu adrinttyp (siehe oben) wurde auch für SYSTEM.PTR eine Variable eingeführt, die je nach /PointerSize belegt wird: sysptrtyp verweist entweder auf SYSTEM.ADDRESS_32 oder SYSTEM.ADDRESS_64.

Der Import von 32-bit Zeigern in Modulen, die im 64-bit Modus compiliert werden, ist erlaubt; ebenso umgekehrt. Dies ist das Herzstück der gemischten Verwendung von 32- und 64-bit Modulen; diese und die dabei zu beachtenden Punkte werden in Abschnitt 7 ausführlich diskutiert.

9.7. Dynamischer Speicher

Dynamische Objekte werden in Oberon mittels NEW vom Heap angefordert. Im Oberon-System wird dieser im Modul Kernel verwaltet; NEW und SYSTEM.NEW werden auf dortige Prozeduren abgebildet (siehe Abschnitt 10.4.2 und [Goe96, Abschnitt 4.2]).

Für stand-alone Anwendungen wird ein komplizierteres, aber auch flexibleres Verfahren verwendet (Abbildung 15):

[34] Selbstverständlich ist dies aber eine Typ-Deklaration im Sinne der Sprachdefinition.

Zeigergröße	Oberon-Prozedur	Systemdienst	Speicherbereich
32-bit	Storage64.ALLOCATE_32	LIB$GET_VM	32-bit
64-bit	Storage64.ALLOCATE	LIB$GET_VM_64	64-bit

Abbildung 16: Die Allokations-Routinen

Damit Anwendungen ggf. ihre eigene Speicherverwaltung implementieren können, werden bei A2O NEW und auf Prozeduren des Modules Storage64 abgebildet.[35] Dieses kann bei Bedarf durch eine andere Implementierung ersetzt werden, etwa um den Speicher in einem Pool zu verwalten. Die zur Allokation im einzelnen verwendeten Prozeduren und Systemdienste sind in Abbildung 16 zusammengestellt.

Es gibt noch eine Variante des Moduls Storage64, das für beide Allokations-Prozeduren LIB$GET_VM verwendet: Diese wird benötigt für *OpenVMS* vor Version 7.0, bei dem LIB$GET_VM_64 noch nicht vorhanden war. Damit ist es möglich, 64-bit Anwendungen unter *OpenVMS* vor Version 7.0 zu nutzen, nachdem sie lediglich neu gelinkt wurden[36] – natürlich vorausgesetzt, die Anwendung ist nicht auf den 64-bit Adreßbereich angewiesen.

Hier noch eine kurze Beschreibung der beiden verwendeten Systemdienste:

LIB$GET_VM liefert im Variablenparameter baseadr die Anfangsadresse eines zusammenhängenden Speicherblocks im 32-bit Speicherbereich:

```
PROCEDURE LIB$GET_VM * (
      numbytes: SYSTEM.SIGNED_64;
   VAR baseadr : SYSTEM.ADDRESS_32): SYSTEM.SIGNED_32;
```

Der Rückgabewert der Prozedur ist der Status, z. B. 'done' oder 'not enough memory available'.

LIB$GET_VM_64 ist ein neuer Betriebssystemdienst seit *OpenVMS* Version 7.0, der einen Speicherblocks im 64-bit Speicherbereich liefert:

```
PROCEDURE LIB$GET_VM_64* (
      numbytes: SYSTEM.SIGNED_64;
   VAR baseadr : SYSTEM.ADDRESS_64): SYSTEM.SIGNED_32;
```

Der Parameter numbytes hat den Typ SYSTEM.SIGNED_64, es ist also theoretisch möglich, bis zu 2^{64} Bytes auf einen Schlag zu allozieren.

[35] Dies ist ein „Erbe“ von Modula-2, das aber leider von anderen stand-alone Oberon-Compilern nicht unterstützt wird.

[36] Alternativ kann man auch einfach zweierlei 'shared images' ('shared library', 'DLL') verwenden und spart damit sogar das Linken.

9.8. Set-Typen

Für die Unterstützung von großen Sets (64 Elemente/Bits) waren nur wenige Eingriffe nötig:

Der Parser wurde auf die neue Syntax umgestellt: Hierzu wurde die Prozedur Faktor erweitert, damit sie das Auftreten eines Bezeichners für einen Set-Typ gefolgt von einer öffnenden geschweiften Klammer als Beginn eines Set-Konstruktors betrachtet. Die anderen beteiligten Prozeduren können nun die maximal zulässigen Elementnummern je nach Set-Typ unterscheiden. Als Nebeneffekt dieser Erweiterung könnten weitere Set-Typen leicht hinzugefügt werden – sollte es je welche geben.

Im Backend des Compilers waren Änderungen nötig, um MAX(Set-Typ) in Abhängigkeit vom Set-Typ zu setzen.[37] Die Code-Erzeugung für Set-Operationen mußte nicht geändert werden, da die Alpha Prozessoren Bit-Operationen immer auf einem 64-bit Register ausführt.

9.9. Behandlung von Variablenparameter

Bei formalen Variablenparametern wird die Adresse des aktuellen Parameters übergeben (call by reference). A2O hat hierfür schon immer 64 Bit vorgesehen, für die Unterstützung des 64-bit Adreßbereichs war hier keine Änderung mehr nötig, siehe auch Abschnitt 7.3.

Dadurch kann aber eine 64-bit Adresse in ein 32-bit Modul wandern, ohne daß dies an der Schnittstelle sichtbar ist: Die Adresse des aktuellen Parameters kann 64 Bit groß sein und damit auch eine 64-bit Adresse enthalten. In einem 32-bit Modul kann dies zu Problemen führen, etwa bei SYSTEM.ADR (siehe Abschnitt 6.2.2) und bei der Erstellung von String-Deskriptoren (siehe Abschnitt 9.12.2).

9.10. Symboldatei

Die Symboldatei beschreibt die Schnittstelle, die ein Modul exportiert. Sie hat gewisse Ähnlichkeit mit der Header-Datei bei C-Compilern, aber es gibt eine Reihe wichtiger Unterschiede:

- Symboldateien werden vom Compiler erzeugt und sind *per Definition* fehlerfrei.[38]
- Header-Dateien werden textuell in den C-Quelltext eingefügt, die Semantik des darin Beschriebenen kann daher beim „Import" beeinflußt werden: sie werden neu interpretiert. Siehe auch Abbildung 14.
- Symboldateien beschreiben die Schnittstelle in der Interpretation des zugehörigen Moduls, also des Schnittstellen-Anbieters.
- Header-Dateien beschreiben die Schnittstelle in der Interpretation des „importierenden" Moduls, also des Schnittstellen-Nutzers.

[37] MIN(Set-Typ) ist in Oberon immer 0 – anders als etwa in Modula-2.

[38] Soweit ein Compiler die Fehlerfreiheit prüfen kann. Hat der Programmierer z. B. eine Konstante mit einem „falschen" Wert definiert, findet sich dieser Wert auch in der Symboldatei. Der Compiler kann nicht prüfen, ob der angegebene Wert „richtig" oder „falsch" ist.

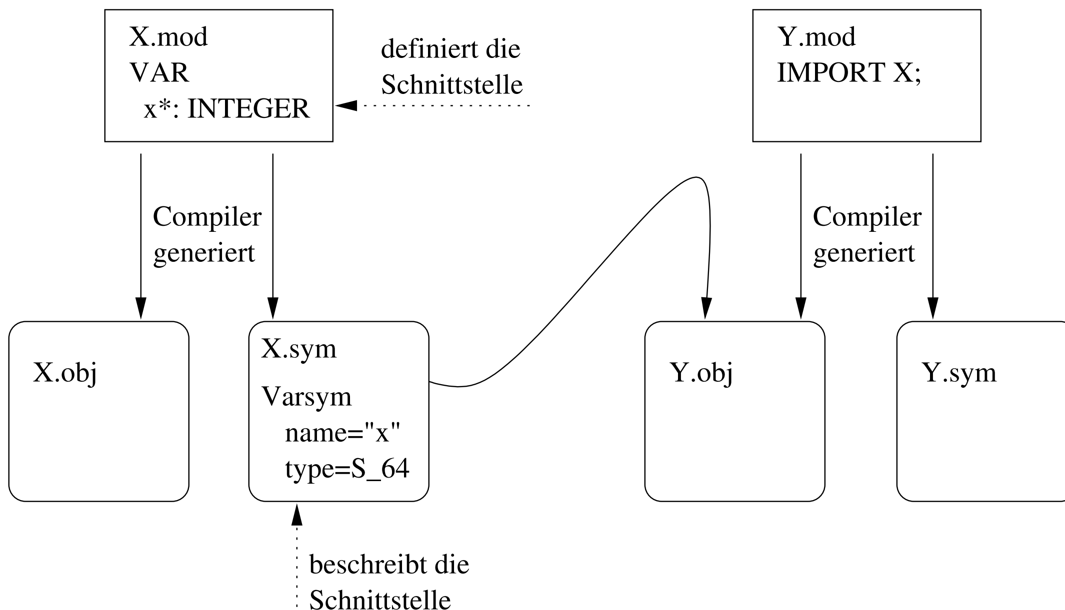

Abbildung 17: Datenfluß der Schnittstellendefinition in Oberon

Zum Unterschied beim Zeitpunkt der Interpretation vergleiche auch Abbildung 17 (Symoldatei) mit Abschnitt 8 (Header-Datei).

Neben der reinen Beschreibung der Schnittstelle enthält jede Symboldatei auch den sog. Modul-Schlüssel. Dieser stellt beim Linken sicher, daß alle Module zusammenpassen. [Goe96, Abschnitt 4.1]

Die folgenden Abschnitte beschreiben, welche Änderungen nötig waren, um Symboldateien 64-bit-fähig zu machen.

9.10.1. Symboldatei-Format

Wie in Abschnitt 7.2.2 beschrieben, muß die Symboldatei eindeutig die Größe der Ganzzahl-Typen enthalten. Hierzu wird das Symbol des – in Abhängigkeit von /PointerSize – äquivalenten SYSTEM-Typs in die Symboldatei geschrieben: von den abstrakten Ganzzahl-Typen SHORTINT, INTEGER und LONGINT wird auf die SYSTEM-Typen konkretisiert (siehe Abschnitt 9.4).

Angenommen, ein Modul exportiere nur eine Ganzzahl-Variable; dann wird die gleiche Symboldatei ergeugt, [39] egal, ob die Variable als LONGINT definiert ist und mit /PointerSize=64 compiliert wird, oder ob die Variable als SYSTEM.SIGNED_64 definiert ist. In beiden Fällen hat die Variable in der Symboldatei den Typ SYSTEM.SIGNED_64.

Darüberhinaus war die Symboldatei zu erweitern für

[39] Natürlich abgesehen vom Modul-Schlüssel, in den normalerweise Datum und Uhrzeit miteinfließen.

- 64-bit Zeiger,[40]
- 64-bit Ganzzahl-Konstanten,
- den Typ LONGSET, und
- Konstanten vom Typ LONGSET.

Durch diese Erweiterung mußte die Versionsnummer der Symboldateien erhöht werden. Damit kann die neue Version von A2O die bisherigen Symboldateien nicht mehr lesen, um umgekehrt. Eine inkompatible Änderung am Format der Symboldateien soll möglichst vermieden werden, um dem Anwender die daraus entstehenden Komplikationen zu ersparen.[41] In diesem Fall ließ sich die Änderung der Versionsnummer nicht vermeiden: Das Format der Symboldateien ist zwar weiterhin das gleiche, aber andere wichtige Eckwerte wurden geändert. Dies wird im Folgenden näher erläutert:

Beim Schreiben der Symboldateien werden den zu exportierenden Typen aufsteigende Nummern zugewiesen, wobei die ersten Positionen mit den Basis- und SYSTEM-Typen besetzt sind. Diese Nummer wird aber nur bei der Referenzierung angegeben: bei der „Deklaration“ in der Symboldatei ist sie ja bekannt, nämlich die nächste freie Position.

Beim Lesen der Symboldatei werden die dort deklarierten Typen in der Reihenfolge ihres Auftretens in der Symboldatei in ein Feld eingetragen. Die ersten Positionen des Feldes sind wiederum mit den Basis- und SYSTEM-Typen besetzt. Bei der Referenzierung eines Typs wird seine Ordinale (Ordnungsnummer) gelesen und das entsprechende Feld-Element genommen.

Ändert sich zwischen dem Schreiben und dem Lesen einer Symboldatei die Anzahl (oder auch die Reihenfolge) des Basis- und SYSTEM-Typen, so wird mit der gelesenen Ordinale der falsche Typ referenziert. Dies führt offensichtlich zu Problemen.

Eine andere, zukunftsgerichtete, Möglichkeit wäre gewesen, zusätzlich die Anzahl der Basis- und SYSTEM-Typen in die Symboldatei zu schreiben. Die Differenz aus dem Wert in der Symboldatei und der wirklichen Anzahl wäre dann der Korrekturwert für die Ordinale. Dieses Verfahren würde aber ein Vertauschen von Basis- und/oder SYSTEM-Typen nicht auffangen.

Den Nachteil dieses Verfahrens – das Lesen der Symboldateien benötigt mehr Zeit – wiegen seine Vorteile nicht auf; zumal Änderungen dieser Art sehr selten vorkommen und ein geändertes Symboldatei-Format dann in Kauf genommen werden kann.

[40] Der Typ SYSTEM.ADDRESS_64 wird intern als 64-bit Zeiger auf „nichts“ gehandhabt, ähnlich dem void * in C; entsprechendes gilt für SYSTEM.ADDRESS_32. Darum ist für die generischen Zeiger-Typen kein eigenes Symbol in der Symboldatei erforderlich.

[41] Um auf der sicheren Seite zu sein, braucht der Anwender zwar „nur“ alle Module neu zu compilieren; aber schon daß kann in eine größere Aktion ausarten.

9.10.2. Symboldatei-Behandlung

Beim Kompilieren mit /PointerSize=64 haben manche pervasive Typen eine andere Größe, als beim Kompilieren mit /PointerSize=32; dies kann zu geänderten Symboldateien führen. Um die Symboldateien des 32-bit Modus nicht zu überschreiben – dies könnte zu Versionskonflikten führen[42] –, haben Symboldateien im 64-bit Modus die Endung .syn64, statt .syn.[43]

Beim Lesen von Symboldateien im 64-bit Modus wird zuerst versucht, die Datei mit der Endung .syn64 zu öffnen; schlägt dies fehl, wird die Endung .syn probiert. Im 32-bit Modus wird nur versucht, die Datei mit der Endung .syn zu öffnen.

Beim Schreiben der Symboldatei wird mit der gleichen Methode die alte Version der Symboldatei zum Vergleich auf Unterschiede geöffnet. Unterscheidet sich die Symboldatei im 64-bit Modus nicht von dem im 32-bit Modus, wird keine neue Symboldatei erzeugt: die Symboldatei des 32-bit Modus ist ja im 64-bit Modus lesbar.

Dieses Verfahren sorgt dafür, daß unterschiedliche Symboldateien für den 32- und 64-bit Modus möglich, aber nicht zwingend sind.

9.11. Objektdateien und Oberon Load Files

Im 64-bit Modus haben die Objektdateien (für stand-alone) immer die Endung .obj64, Oberon Load Files (für das Oberon-System) immer die Endung .olf64. Dadurch lassen sich die verschiedenen Versionen eines Moduls leicht unterscheiden und auf Wunsch entsprechend binden.

Objektdateien des 32-bit Modus lassen sich mit denen des 64-bit Modus mischen, solange die sog. Modul-Schlüssel zusammenpassen (siehe auch Abschnitt 7). Bei Oberon Load Files ist dies nicht möglich: aos kann nur in einem der beiden Modi arbeiten (siehe Abschnitt 10) und lädt daher nur Oberon Load Files eines Modus.

Im Folgenden werden die nötigen Änderungen an den beiden Dateiformaten beschrieben:

9.11.1. Änderungen am Objektdatei-Format

Am Objektdatei-Format waren keine Änderungen nötig – dies wäre auch nicht möglich, da daß Format von *OpenVMS* vorgegeben ist. Allerdings mußten in den Routinen, die die Objektdateien erzeugen, alle Stellen untersucht werden, an denen Ganzzahl- oder Zeigertypen behandelt werden. Hier waren etliche kleine Änderungen nötig, etwa beim Erzeugen der Typdeskriptor-Sektion [Goe96, Abschnitt 5.2].

In den Debugger-Informationen werden nun die Datentypen für 64-bit Zeiger unterstützt.

[42] Die 32-bit Objektdatei hat einen anderen Schlüssel als die neue Symboldatei und die Klienten „sehen“ einen andere Schittstelle als zuvor.

[43] a2o verwendet seit jeher die Endung .syn, um Überscheidungen mit Modula-2-Symboldateien zu verhindern: diese haben die Endung .sym.

9.11.2. Änderungen am Oberon Load File Format

Das Oberon Load File Format [Goe96, Anhang A] mußte für die Unterstützung von 64-bit Zeigern etc. nicht erweitert werden: fast alle Zahlen, die darin auftreten, werden als komprimierte Ganzzahl geschrieben.[44] Die einzigen Felder, deren Länge im Oberon Load File Format in Bytes festgelegt ist, sind:

- der Modul-Schlüssel, dessen Länge *immer* 8 Byte geträgt,
- zwei Datenfelder im Prozedurdeskriptor, deren Länge von *OpenVMS* festgelegt ist, und
- ein Bitfeld, daß ein 32-bit Set darstellt (dessen Länge sich nur ändern wird, falls andere, tiefgreifende Änderungen am Dateiformat vorgenommen werden).

Im Compiler mußte damit nur die Prozedur erweitert werden, die die komprimierten Ganzzahlen schreibt: sie muß 64-bit Ganzzahlen akzeptieren. Da im Compiler intern sowieso alle Ganzzahlen auf 64 Bit umgestellt wurden (siehe Abschnitt 9.3), war hierfür kein gesonderter Eingriff nötig.

Die Code- und Konstanten-Sektionen des Moduls werden im Oberon Load File immer nur als Byte-Folge gespeichert und beim Laden uninterpretiert in den Speicher kopiert. Änderungen in der Code-Erzeugung oder der Struktur der Konstanten-Sektion beeinflussen daher das Oberon Load File Format nicht.

9.12. *OpenVMS Alpha*-abhängige Erweiterungen

9.12.1. Prozedur-Typen und Prozedur-Variablen

In Oberon gibt es Prozedur-Variablen und Prozedur-Typen [Wir92, Abschnitt 6.5], deren Größe bei A2O so bestimmt wird:

- Wenn das Modul, das die Deklaration der Prozedur P enthält, mit /POINTERSIZE=X compiliert wurde, dann ist die Größe von P X DIV 8.
- Wenn das Modul, das die Deklaration des Prozedur-Typen T enthält, mit /POINTERSIZE=X compiliert wurde, dann ist die Größe von T X DIV 8.

  Unter „Deklaration" ist hier eine Typ-Deklaration der Form TYPE Y = PROCEDURE(...); zu verstehen – analog der Deklaration von Zeigern, siehe Abschnitt 9.6.
- Die Größe der Prozedur-Variablen v: T ist SIZE(T).

Die Unterscheidung der Größe entsprechend /POINTERSIZE ist nötig, da der Inhalt von Prozedur-Variablen intern durch einen Zeiger repräsentiert wird.[45] Die Prozedur-Variablen müssen also groß genug sein, Zeiger aufzunehmen.

[44] Bei komprimierten Ganzzahlen hängt die Anzahl der Bytes in der Datei vom Wert der Zahl ab, nicht vom Typ.

[45] Dies ist das übliche Vorgehen und keine Eigenheit von A2O.

In stand-alone Anwendungen würden auch Prozedur-Variablen als 32-bit Zeiger weiterhin funktionieren – auch bei der Verwendung des 64-bit Adreßbereichs: Denn bei *OpenVMS Alpha* ist eine Prozedur-Variable ein Zeiger auf ein 'Linkage-Pair' (siehe [Goe96, Abschnitt 5.1]). Diese müssen wegen einer Beschränkung im Linker von *OpenVMS* Version 7.0 im 32-bit Speicherbereich liegen. Aber das Verlagern der 'Linkage-Pairs' in den dynamischen Speicher, wie es für AOS benötigt wird (siehe Abschnitt 10.3), würde verhindert.

9.12.2. Stringdeskriptoren

String-Deskriptoren werden in *OpenVMS* für String-Parameter benutzt. A2O verwendet sie jedoch nur für die Schnittstellen zum Betriebssystem oder Modulen, die in anderen Sprachen geschrieben wurden. Innerhalb von Oberon-Programmen ist die Länge entweder zur Compile-Zeit bekannt (ARRAY nn OF Typ), oder wird bei dynamischen Arrays immer als 64-bit Zahl übergeben; die Adresse ist in beiden Fällen immer 64 Bit groß.

Die Datenstruktur der String-Deskriptoren wurde mit *OpenVMS* Version 7.0 von 8 Bytes

```
RECORD
  strLen:  SYSTEM.SIGNED_16;
  ident:   SYSTEM.SIGNED_16 (* := 0; *)
  address: SYSTEM.SIGNED_32
END
```

erweitert auf 24 Bytes:

```
RECORD
  selfId1: SYSTEM.SIGNED_16 (* :=  1; must be one (MBO) *)
  ident:   SYSTEM.SIGNED_16 (* :=  0; *)
  selfId2: SYSTEM.SIGNED_32 (* := -1; must be minus one (MBMO) *)
  strLen:  SYSTEM.SIGNED_64;
  address: SYSTEM.SIGNED_64
END
```

Der 24 Bytes lange Deskriptor kann Adressen oberhalb von 2^{32} und Längen größer 2^{16} beschreiben. Die Elemente selfId1 und selfId2 entsprechen den alten Elementen strLen und address. Die Werte dieser Felder entscheiden, ob es sich um einen kurzen (8 Bytes) oder langen (24 Bytes) Deskriptor handelt: Bei einem langen Deskriptor muß selfId1 $= 1$ (strLen) und selfId2 $= -1$ (address) sein.[46] [47] Detaillierte Informationen hierzu finden sich in [Dig96b, Abschnitt 5.1].

Steht zur Compile-Zeit fest, daß die Länge kleiner als MAX(SIGNED_16) ist, und die Adresse im 32-bit Bereich liegt[48], so wird ein kurzer Deskriptor erzeugt. Ist zur Compile-Zeit sicher, daß einer der Werte nicht in die 8 Byte Struktur passen würde, wird direkt ein großer Deskriptor erzeugt.

[46] Diese Art von Strukturen nennt man selbst-identifzierend (engl. self-identifying).

[47] Durch diese Konstruktion verursacht ein Systemdienst, der noch keine 64-bit Deskriptoren unterstützt, beim Zugriff eine Schutzverletzung: -1 ist nie eine gültige Adresse.

[48] Das ist bei stand-alone Programmen immer dann der Fall, wenn konstante Strings übergeben werden.

Wenn zur Compile-Zeit nicht sicher ist, daß sowohl die Stringlänge als auch die Adresse in einen 8 Byte String-Deskriptor passen, wird zur Laufzeit geprüft: entsprechend wird dann ein kurzer oder langer String-Deskriptor erzeugt. Dies kann auch in einem Modul passieren, das mit /PointerSize=32 compiliert wurde: Beispielsweise könnte ein String via Variablenparameter übergeben worden sein, von dessen Adresse zur Compile-Zeit nicht sicher ist, ob es eine 32-bit Adresse ist.

Der Deskriptor wird bei /PointerSize=64 nicht automatisch als 24-byte Struktur erzeugt, um nicht unnötigerweise die Nutzung von Systemdiensten auszuschließen, die die langen Deskriptoren noch nicht unterstützen.[49] Bei /PointerSize=32 wäre die Entscheidung zur Laufzeit immer noch nötig, siehe oben.

9.13. Sonstige Erweiterungen

Andere Erweiterungen, als die oben beschriebenen, waren nicht nötig und wurden nicht vorgenommen.

9.14. Beschränkungen durch die Implementierung

9.14.1. Aufgehobene Beschränkungen

Da der Compiler intern jetzt alle Ganzzahl-Berechungen mit 64 Bit durchführt, konnten noch einige Beschränkungen aufgehoben werden:

- Der Wertebereich für konstante Literale und konstante Ausdrücke wurde erweitert auf [MIN(SIGNED_64) ... MAX(SIGNED_64)] (siehe Abschnitt 9.3.1)
- Die Anzahl der Dimensionen bei Arrays wurde erweitert auf MAX(SIGNED_64) – wobei bereits die bisherige Beschränkung auf MAX(SIGNED_32)$= 2 \times 10^9$ eher theoretischer Natur war, als in der Praxis relevant.
- Die Anzahl der Elemente pro Dimension bei dynamischen Arrays (alloziert mit NEW oder übergeben als Parameter) wurde erweitert auf MAX(SIGNED_64).

  Bei statischen Array-Typen ist die Anzahl der Elemente pro Dimension weiterhin beschränkt auf MAX(SIGNED_32) (siehe Abschnitt 9.14.2).[50]

9.14.2. Verbliebene bzw. neue Beschränkungen

Auch nach den Erweiterungen enthält der Compiler noch einige Einschränkungen. Diese sollten sich aber in der Praxis nicht bemerkbar machen.

[49] Bspw. unterstützt der Systemdienst zum Erzeugen von Fehlermeldungen aus Fehlernummern nur die kleine Struktur. Ohne ihn könnte die Ausnahmebehandlung aber keinen sinnvollen Hinweis liefern.

[50] Sollte dies in der Praxis ein Problem werden, läßt sich ein Programm leicht auf dynamische Arrays umstellen: Oberon dereferenziert bei z. B. arrayPtr[i] den Zeiger automatisch. Damit sind nur an den (wenigen) Stellen Eingriffe nötig, an denen das Array als Ganzes angesprochen wird.

1. Die Größe von Record-Typen und die Gesamtgröße der globalen oder lokalen Variablen ist weiterhin beschränkt auf MAX(SIGNED_32).

2. Die Anzahl der Elemente pro Dimension bei statischen Arrays ist weiterhin beschränkt auf MAX(SIGNED_32).

3. Module, die Adreßrechnung betreiben, müssen die Grenze bei MAX(SIGNED_64) = 2^{63} beachten: alle höheren Adressen haben Bit 63 gesetzt und werden damit als negative Ganzzahlen betrachtet. Da Oberon keinen vorzeichenlosen Ganzzahltypen kennt, muß dieser Fall gesondert behandelt werden.[51]

   Module, die keine Adreßrechnung betreiben – und das sind die allermeisten –, sind hiervon nicht betroffen.

9.15. Implementierung der geänderten Prozeduren

Wie in Abschnitt 6.2.2 beschrieben, ändert sich die Semantik einiger Prozeduren bei Verwendung von /POINTERSIZE=64: sie erwarten bzw. liefern weiterhin LONGINT, mit /POINTERSIZE=64 ist dies aber SYSTEM.SIGNED_64. Da der Compiler intern natürlich zwischen SYSTEM.SIGNED_32 und SYSTEM.SIGNED_64 unterscheidet, wurde er erweitert, um (evtl. in Abhängigkeit von /POINTERSIZE) beide Typen zu akzeptieren.

Im Folgenden werden diese Änderungen vorgestellt.

9.15.1. SHORT, LONG, SYSTEM.SHORT und SYSTEM.LONG

Die neue Funktion SYSTEM.SHORT und die geänderte Funktion SHORT unterscheiden sich nur für einen einzigen Fall: SHORT konvertiert bei einem Eingabeparameter vom Typ SYSTEM.SIGNED_64 und /POINTERSIZE=64 auf SYSTEM.SIGNED_16.

Damit war zur Realisierung der beiden Funktionen im wesentlichen eine Änderungen nötig; hier der entsprechende Programmausschnitt:

```
...
ELSIF f = QInt THEN (* input is of type SIGNED_64 *)
  IF (fctno = OPE.shortfn) & (OPM.ptr64 IN OPM.CompOptions) THEN
   Convert(x, OPT.inttyp)
  ELSE Convert(x, OPT.linttyp) END
ELSIF f = LReal THEN Convert(x, OPT.realtyp)
...
```

Wobei QInt die Ordinale für SYSTEM.SIGNED_64 ist (siehe Abschnitt 9.10.1). OPT.inttyp und OPT.linttyp stellen die Typen SYSTEM.SIGNED_16 bzw. SYSTEM.SIGNED_32 dar.

Für LONG und SYSTEM.LONG gilt ähnliches; sie wurden analog implementiert.

[51] Z. B. indem man an dieser Speicherstelle einige Bytes immer alloziert hält.

9.15.2. SYSTEM.ADR

SYSTEM.ADR liefert die Adresse einer Variablen; wobei auch im 32-bit Modus Adressen auftreten können, die nicht im 32-bit Speicherbereich liegen, siehe Abschnitt 7.4. Darum muß in gewissen Fällen geprüft werden, ob die ermittelte Adresse eine 32-bit Adresse ist; nur dann kann sie unverfälscht an eine 32-bit Ganzzahl zugewiesen werden.

Bei A2O wird hierbei die Adresse *immer* in ein Register geladen.[52] Im Register ist eine Adresse genau dann eine 32-bit Adresse, wenn die Bits 31 bis 63 alle 0 oder alle 1 sind (vorzeichen-erweitert, siehe Abschnitt 7.3). Der folgende kurze Alpha Assembler-Code prüft die Bedingung und bricht das Programm ggf. mit dem Laufzeitfehler „PointerGreater32Bit" ab – die zu prüfende Adresse steht hierbei in Register Reg_X:

```
      ADDL   R31,   Reg_X, Reg_Y
      CMPEQ  Reg_X, Reg_Y, Reg_Y
      BEQ    Reg_Y, lab2
      BRA    lab1
lab2: CallRunTimeError(PointerGreater32Bit)
lab1: ....
```

Der Trick an diesem Code ist: Mit nur einer Instruktion wird Bit 31 von Reg_X nach Bit 63 propagiert, das Ergebnis muß dem Eingangswert entsprechen. Im Einzelnen:

Die erste Anweisung ist eine Addition von zwei 32-bit Zahlen, deren Ergebnis immer eine 64-bit Zahl ist: Register 31 ist immer Null, Reg_Y enthält also die Zahl, die in den unteren 32 Bit von Reg_X steht – allerdings vorzeichen-erweitert! Enthält Reg_Y nicht den gleichen 64-bit Wert wie Reg_X, so ist der Inhalt von Reg_X keine 32-bit Zahl (bzw. Adresse). Dies wird mit CMPEQ geprüft, anschließend wird entsprechend verzweigt.

Dieser Code ist sehr effizient: Die Alpha Prozessoren nehmen beim Pipelining an, daß bedingte Vorwärtssprünge nicht genommen werden. Falls der Test gelingt, hat der Prozessor den Sprung nach lab1 bereits vorausberechnet – der Test hat kaum Zeit gekostet. Falls der Test fehlschlägt und der bedingte Vorwärtssprung zu lab2 doch genommen wird, kommt es auf etwas Zeitverlust nicht an: das Programm wird sowieso abgebrochen.

9.15.3. ENTIER

Die Standardprozedur[53] ENTIER(r: REAL): LONGINT muß je nach Compiler-Modus SYSTEM.SIGNED_32 oder SYSTEM.SIGNED_64 liefern. Bei /POINTERSIZE=64 entspricht ENTIER damit der Standardprozedur SYSTEM.LENTIER; letztere liefert immer SYSTEM.SIGNED_64.

Die Umstellung ließ sich sehr einfach realisieren:

Damit der Compiler effizient mit Standardprozeduren umgehen kann, hat jede eine Ordinale. Beim Auflösen eines Bezeichners für eine Standardprozedur bekommt der Parser die Information „Standardprozedur *n*".

[52] Ist ist bei RISC-Prozessoren, zu denen auch die Alpha Prozessoren zählen, generell üblich.

[53] Standardprozeduren sind Prozeduren, die der Compiler von Haus aus kennt, z. B. ASH, SYSTEM.MOVE und ENTIER.

Um ENTIER auf SYSTEM.LENTIER abzubilden, wird also lediglich der Bezeichner ENTIER mir der Ordinale für SYSTEM.LENTIER assoziiert. Die Funktion SYSTEM.LENTIER war bereits implementiert, weshalb keine weiteren Änderungen nötig waren.

9.15.4. ASH

Die Funktion ASH ist definiert als ASH(x: LONGINT): LONGINT; hat der Eingabeparameter einen „kleineren" Typ als LONGINT, so wird er von Oberon automatisch nach LONGINT konvertiert. A2O akzeptiert zusätzlich Eingabeparameter vom Typ SYSTEM.SIGNED_64, das Ergebnis ist dann ebenfalls vom Typ SYSTEM.SIGNED_64; siehe auch Abschnitt 6.2.2.

Ob LONGINT 32 oder 64 Bit groß ist, hängt von /POINTERSIZE ab, es wird daher anhand von adrintyp (siehe Abschnitt 9.5) geprüft, ob konvertiert werden muß. Die entsprechende Zeile im Quelltext ist diese:

```
IF Typesize(input.typ) < Typesize(addrinttyp) THEN
  Convert(input, adrinttyp) END;
```

Bei /POINTERSIZE=32 und Eingabetyp SYSTEM.SIGNED_64 gilt $Typesize(input.typ) > Typesize(adrinttyp)$, es wird also nicht konvertiert.

An der Code-Erzeugung war keine Änderung nötig: Das Verschieben des Bitmusters wird in Registern vorgenommen. Da eine 32-bit Ganzzahl im 64-bit Register immer vorzeichen-erweitert – also als 64-bit Ganzzahl – dargestellt wird, muß hier gar nicht unterschieden werden. Der bisherige Code war bereits in der Lage, 64-bit Ganzzahlen zu schieben. Der Unterschied liegt lediglich darin, ob der Compiler das Ergebnis als 32- oder 64-bit Zahl betrachtet.

9.15.5. Die restlichen geänderten Prozeduren

Bei den Parametern, die eine Speicheradresse darstellen (formaler Typ LONGINT), akzeptieren SYSTEM.MOVE, SYSTEM.GET, SYSTEM.PUT, und SYSTEM.BIT – unabhängig von /POINTERSIZE! – sowohl SYSTEM.SIGNED_32 als auch SYSTEM.SIGNED_64.

Es ist weder nötig noch sinnvoll, die Parameter auf 32 Bit (bei /POINTERSIZE=32) bzw. 64 Bit (bei /POINTERSIZE=64) festzulegen: Die Alpha Prozessoren benutzt zur Adressierung *immer* alle 64 Bit eines Registers. Siehe hierzu auch Abschnitt 7.3.

Diese Flexibilität weicht die Typsicherheit von Oberon nicht auf: Als einziges Problem könnte entstehen, daß von einer 64-bit Adresse nur die unteren 32 Bit benutzt weren. Dies kann aber nur geschehen, wenn die 64-bit Adresse mit SYSTEM.VAL mutwillig abgeschnitten wird. Damit wurde die Typsicherheit allerdings schon bei SYSTEM.VAL außer Kraft gesetzt,[54] nicht bei der Verwendung der Adresse.

Bei SYSTEM.ESTABLISH und SYSTEM.REVERT ist der Ergebnistyp immer LONGINT. Damit keine Fallunterscheidung benötigt wird, wird hier mit der Variable adrinttyp gearbeitet (siehe Abschnitt 9.5).

[54] Gerade dazu ist SYSTEM.VAL ja da.

9.16. Änderung der Code-Qualität in 32-bit Modulen

Natürlich stellt sich die Frage, ob die Code-Qualität bei 32-bit Modulen durch die Erweiterung auf 64 Bit gelitten hat. Kriterien seien hierfür die Länge des erzeugten Codes und, damit verbunden, die Ausführungsgeschwindigkeit.

Die einzigen Stellen, an denen im 32-bit Modus gegegenenfalls mehr Code erzeugt wird als vor der Erweiterung, sind String-Deskriptoren und SYSTEM.ADR:

- String-Deskriptoren kommen in Oberon Programmen selten vor.
- Bei SYSTEM.ADR werden zwar einen Reihe von Instruktionen zusätzlich erzeugt, davon werden aber im Normalfall nur drei mehr als zuvor ausgeführt.

Die Qualität des erzeugten Codes bei 32-bit Modulen hat sich also nicht nennenswert verschlechtert.

10. Implementierung von AOS

Dieses Kapitel beschreibt, welche Änderungen und Erweiterungen an AOS vorgenommen wurden. In Abschnitt 5.3 sind die Ziele genannt, die hierbei verfolgt wurden. Hier nocheinmal in Kurzform:

- AOS soll den 64-bit Adreßbereich nutzen können.
- Die Tauglichkeit der Spracherweiterung soll getestet werden.
- AOS soll keinerlei 32-bit Beschränkungen mehr enthalten.

Um in einer Anwendung die Vorteile von 64-bit Ganzzahlen und 64-bit Adressen sinnvoll nutzen zu können, muß das gesamte ALPHA OBERON SYSTEM mit /POINTERSIZE=64 compiliert werden. Andernfalls steht für die Anwendung der größte Teil der Modulbibliothek des Oberon-Systems nicht zur Verfügung: Viele Prozeduren der Bibliothek haben Zeiger als Parameter.[55]

Die gemischte Verwendung von 32- und 64-bit Zeigern in AOS würde zudem die Speicherverwaltung verlangsamen und verkomplizieren: Der Garbage Collector müßte den Stack in Schritten von 4 *und* 8 Bytes durchsuchen [Goe96, Abschnitt 7], um alle Zeiger zu finden.

Unglücklicherweise geht das Oberon-System an vielen Stellen inhärent davon aus, daß Zeiger und LONGINT beide 32 Bit groß sind. Dies wird z. B. deutlich, wenn Laufzeitstrukturen aufgebaut werden, oder im Garbage Collector(siehe auch nächster Abschnitt).

10.1. Vorbereitungen bei der 32-bit Version

Bei der Portierung des Oberon-Systems auf *OpenVMS Alpha* wurden Teile des Systems bereits auf 64 Bit vorbereitet – etwa der Garbage Collector, die Ausnahmebehandlung, die Arraydeskriptoren und die Typdeskriptoren –, siehe [Goe96, Abschnitt 4.6].

Die meisten Module des Oberon-System waren aber noch auf 32 Bit ausgelegt. Dies manifestierte sich an vielen low-level Stellen durch Literale (wie -8, -4, 4, 8, 12) zur Speicher-Allokation oder Adreßrechnung. Typischerweise kommen sie in der Nähe von SYSTEM.GET und SYSTEM.PUT vor, weitere Kandidaten sind SYSTEM.ADR und SYSTEM.VAL. Auch die Vorkommen von SHORT und LONG unterstellten zum Teil bestimmte Typgrößen.

Leider sind viele Programmierer von Anwendungen diesem schlechten Beispiel gefolgt.

10.2. Vorgehensweise

Die Portierung von AOS auf 64 Bit wurde in zwei Stufen vollzogen:

[55] Zur Problematik bei der gemischten Verwendung unterschiedlich großer Zeiger siehe Abschnitt 9.6 und Abschnitt 7.

1. Umstellen aller Module auf 64-bit Zeiger und 64-bit LONGINT mit anschließendem Bootstrap und Test im 32-bit Adreßraum.

   Dadurch wurden alle Probleme und Abhängigkeiten behoben, die aus der Größe der Typen resultiert.

2. Testen von AOS unter Verwendung von 64-bit Speicher.

   Dies diente der Suche von Fehlern, die durch nicht-64-bit-fähige Module oder Betriebssystemdienste verursacht wurden.

Im folgenden werden nur die Komponenten beschrieben, an denen größere Änderungen nötig waren; zwischen den beiden Stufen wird nicht explizit unterschieden.

Bei allen Änderungen im Quelltext des Oberon-Systems wurde darauf geachtet, daß das Oberon-System auch noch mit /POINTERSIZE=32 compiliert werden kann. Hierzu wurden zwei Tricks verwendet, die kurz erklärt werden sollen:

1. Um Füllbytes in Datenstrukturen – es werden immer nur 32-bit Ganzzahlen auf 64 Bit aufgefüllt – optional zu entfernen, exportiert das Modul A2OLayout[56] (siehe [Goe96, Abschnitt 5.1.2]) einen Typ PAD. In der 32-bit Version von A2OLayout ist dieser als SYSTEM.SIGNED_32 definiert, in der 64-bit Version als leerer Record (Größe null Bytes).

2. Um eine 64-bit Ganzzahl, die eine Adresse darstellt, auf 32 Bit zu kürzen, falls das Modul mit /POINTERSIZE=32 kompiliert wurde, sie andernfalls aber nicht zu kürzen, werden Anweisungen dieser Art verwendet:

   ```
   fp := SYSTEM.VAL(LONGINT, invoContext.iReg[FP]);
   ```

   Da ein 32-bit Modul sowieso nur im 32-bit Adreßbereich lauffähig ist, können die Adressen unbesorgt abgeschnitten werden.

10.3. Bootlader und Modullader

AOS enthält einen dynamischen Modullader, der *OpenVMS*-Linker mit seinen Beschränkungen wird nicht benutzt. Damit können alle Daten, inklusive dem Programmcode, im 64-bit Adreßbereich liegen. Wenn *OpenVMS* (und X 11) es erlauben würde, könnten auch die Stacks von Coroutinen (Threads) dort liegen: dies ist die einzige verbliebene 32-bit Beschränkung in AOS_64 (siehe Abschnitt 10.4.3).

[56]Dieses Modul gibt es in zwei Versionen: Eine für 32 Bit und eine für 64 Bit. Da AOS hierfür unterschiedliche Datei-Endungen hat (siehe Abschnitt 9.10.2 und Abschnitt 9.11), können keine Konflikte entstehen.

Der Bootlader alloziert einen Speicherblock (siehe Abschnitt 10.4) und lädt dort hinein alle Module, die für den Bootstrap benötigt werden. Er enthält damit zum Teil den gleichen Programmtext, den auch der Modullader enthält.

Um zu wählen, ob der Speicherblock im 32- oder im 64-bit Adreßbereich alloziert wird, unterstützt der Bootlader nun eine Option mit dem (etwas unglücklich gewählten) Namen /POINTERSIZE.

Die Quelltexte beider Lader waren bereits auf 64 Bit vorbereitet, daher waren in diesen Modulen nur wenige Stellen zu korrigieren:

- verbliebene Annahmen über SIZE(LONGINT) = 4 = SIZE(ADDRESS)
- Umstellen der Datenstrukturen auf 64 Bit durch Entfernen von Füllbytes.
- Korrektur von 64-bit Ganzzahlen, wenn in den Strukturen nur 32 Bit vorgesehen sind, denn die Leseprozedur für komprimierte Ganzzahlen (Files.ReadNum) liefert mit /POINTERSIZE=64 eine 64-bit Ganzzahl.

  Dies betrifft z. B. den Eintrag identLen in den Typdeskriptoren [Goe96, Abschnitt 5.2], bei dem 32 Bit auch in Zukunft ausreichen werden.

Beim Bootlader mußten zudem einige Aufrufe von *OpenVMS* Systemdiensten geändert werden, die noch nicht 64-bit-fähig sind: auch hier werden ggf. lokale Parameter verwendet (ähnlich wie in Abschnitt 10.7 beschrieben). Bei der Initialisierung des Speicherblocks für das Oberon-System wird nun der von der Zeiger-Größe abhängige Set-Typ verwendet (siehe Abschnitt 10.4.2).

10.4. Speicherverwaltung

Das Oberon-System übernimmt die Speicherverwaltung des Heaps selbst: Es fordert einen Speicherblock vom Betriebssystem an und verwaltet ihn mit Hilfe des Garbage Collectors (siehe Abschnitt 10.4.2). Die Größe des Heaps kann beim Starten des Oberon-Systems angegeben werden, die Vorgabe ist 4 MB im 32-bit Speicher.

Mit einer neuen Option kann bei Starten des Oberon-Systems bestimmt werden, ob der Speicherblock im 32- oder im 64-bit Adreßbereich alloziert wird: AOS kann dadurch veranlaßt werden, nur Speicher im 32-bit Adreßbereich zu verwenden, auch wenn es mit 64-bit Zeigern arbeitet. Damit läßt sich ggf. schnell herausfinden, ob ein Fehler durch abgeschnittene Zeiger verursacht wird.

10.4.1. Allokation

Wie in Abschnitt 9.7 beschrieben, bildet A2O die Prozeduren NEW und SYSTEM.NEW auf Storage64.ALLOCATE (bzw. Storage64.ALLOCATE_32) ab. Für AOS werden sie statt dessen direkt auf Kernel.ALLOCATE_64 und Kernel.ALLOCATE_32 abgebildet, siehe hierzu auch [Goe96, Abschnitt 7.2].

Da AOS nur einen Speicherblock verwaltet, benutzen beide Allokationsroutinen intern die gleiche Unterprozedur; bei Kernel.ALLOCATE_32 wird aber geprüft, ob der Speicher im

32-bit Adreßbereich alloziert wurde. Falls nicht, wird das Programm – nicht das Oberon-System! – abgebrochen: mögliche Fehler sind so schnell entdeckt. Allerdings sollte die Prozedur Kernel.ALLOCATE_32 in AOS_64 *nie* gerufen werden, da in AOS_64 nur 64-bit Zeiger vorkommen.

Zusätzlich stellt das Modul Unix eine Prozedur Malloc bereit, die Speicher mit dem Systemdienst LIB$GET_VM im 32-bit Adreßbereich direkt von *OpenVMS* angefordert. Dies wird z. B. für Stacks der Coroutinen (siehe Abschnitt 10.4.3) und die 'access blocks' des Ein-/Ausgabesystems (siehe Abschnitt 10.6) genutzt.

Der mit Unix.Malloc allozierte Speicher wurde nicht vom Oberon-System vergeben, darf also auch nicht vom Garbage Collector verfolgt werden: der Garbage Collector würde in die Irre laufen. Daher mußten alle Zeigervariablen, die solchen Speicher referenzierten, in LONGINT geändert werden – begünstigt dadurch, daß Malloc ebenfalls LONGINT liefert. Bei Dereferenzierungen der ehemaligen Zeigervariablen muß nun mit temporären Zeigern gearbeitet werden, an die der Wert zugewiesen wird.

Ein weitergehender Lösungsansatz wird in Abschnitt 12.1 beschrieben.

10.4.2. Garbage Collector

Auch der Garbage Collector war bereits auf 64 Bit vorbereitet. Diese Vorbereitung erwieß sich als sehr brauchbar, da lediglich *zwei* Änderungen systematisch durchgeführt werden mußten; damit waren weniger Änderungen nötig, als erwartet wurde.

- Umstellen der Datenstrukturen auf 64 Bit durch Entfernen von Füllbytes und ggf. korrigieren von LONGINT in SYSTEM.SIGNED_32 (wenn das Datum 32 Bit groß bleiben soll).

- Umstellen der Sets von 32 auf 64 Bit, da der Garbage Collector Adressen auf Bit-Ebene manipuliert.

  Damit der Garbage Collector weiterhin im 32-bit Modus funktioniert, wird ein Typ Set aus A2OLayout[57] importiert, der der Größe der Zeiger entspricht: Durch die Syntaxerweiterung für Set-Konstruktoren (siehe Abschnitt 6.4.3) kann für die 32- und 64-bit Version der gleiche Quelltext benutzt werden.

10.4.3. Der Stack

Das Ausnahmebehandlungs-System von *OpenVMS* erwartet, daß der Stack im 32-bit Adreßbereich liegt, sonst funktionieren viele der Systemdienste nicht (siehe z. B. Abschnitt 10.5). Das Problem trat erst zum Vorschein, als der Runtime-Debugger im 64-bit Modus laufen sollte: der Debugger verwendet Coroutinen, für die jeweils ein eigener Stack angelegt wird – im Heap das Oberon-Systems.[58] Um dieses Problem zu umgehen, wird

[57]Dieses Modul gibt es in zwei Versionen, siehe Fußnote 56.

[58]Der Stack der Hauptroutine des Systems ist der Stack, der beim Starten des Bootladers angelegt wird; er wird von *OpenVMS* immer im 32-bit Bereich angelegt.

jetzt eine geänderte Version des Moduls Coroutines verwendet: sie alloziert den Stack mit Unix.Malloc im 32-bit Adreßbereich.

Fraglich ist allerdings, ob man die Stacks überhaupt in den 64-bit Speicher verlegen sollte, solange die 'access blocks' des Ein-/Ausgabesystems im 32-bit Speicher liegen müssen (siehe Abschnitt 10.6) und X 11 R 6 nur 32-bit Adressen verarbeiten kann (siehe Abschnitt 10.7): Wenn die Stacks nicht mehr im 32-bit Adreßbereich liegen, müssen hier andere Lösungen gefunden werden.

10.5. Ausnahmebehandlung

Die Ausnahmebehandlung in *OpenVMS* ist von der Schnittstelle her auf 64 Bit ausgelegt, enthält aber noch einige Fehler:

- Der Stack muß noch im 32-bit Adreßbereich liegen, damit alle Betriebssystemdienste der Ausnahmenbehandlung funktionieren.
- Der Prozedurdeskriptor (siehe [Goe96, Abschnitt 5.1]) der 'call-back' Prozedur bei SYS$PUT_MSG muß im 32-bit Adreßbereich liegen.

  SYS$PUT_MSG erzeugt aus Fehlernummern Fehlermeldungen und überläßt der 'call-back' Prozedur deren Ausgabe.

Die Zwischenlösung für das Stack-Problem ist in Abschnitt 10.4.3 beschrieben. Sollte der Stack jemals im 64-bit Adreßbereich liegen, müssen alle Routinen der Ausnahmebehandlung im Oberon-System überprüft werden, ob sie 64-bit-konform sind.

Um das zweite Problem zu umgehen, wird ggf. ein Teil der 'Linkage-Sektion' (enthält die 'Linkage-Pairs') des Moduls System (implementiert die Ausnahmebehandlung des Oberon-Systems) in den 32-bit Adreßbereich kopiert. Dies ist ein „übler Hack", der viel Wissen über die interne Code-Struktur voraussetzt, aber die einzige Möglichkeit, den Fehler in *OpenVMS* zu umgehen.

10.6. Ein-/Ausgabesystem (Record Management Service)

Die Prozeduren des *OpenVMS*-Ein-/Ausgabesystems benötigen die sog. 'file access blocks' und 'record access blocks' weiterhin im 32-bit Speicher. Auch können viele Adreß-Felder in diesen Datenstrukturen nur 32-bit aufnehmen. Die Daten selbst können im 64-bit Adreßbereich liegen, siehe unten.

Das Module RMSAccess, das eine Abstraktionsschicht über dem Ein-/Ausgabesystems bildet, prüft nun, ob diese Bedingungen erfüllt sind – falls nicht, wird das Programm abgebrochen. Innerhalb des Oberon-Systems wird RMSAccess nur vom Modul Unix verwendet, das diese Bedingungen einhält: die 'access blocks' werden mit Malloc im 32-bit Speicher alloziert, Parameter ggf. erst auf den Stack kopiert (siehe Abschnitt 10.4.3).

Die Prüfungen innerhalb von RMSAccess scheinen damit unnötig, sind aber eine zusätzliche Sicherheitsstufe – auch falls das Modul anderweitig verwendet wird.[59]

[59]Nach Fertigstellung von AOS-64 wurde noch ein Modul Directories auf AOS portiert, das ebenfalls direkt RMSAccess verwendet.

Die Pufferdaten selbst müssen nicht kopiert werden, da der erweiterte Record Management Service (RMS) 64-bit Adressen und Pufferlängen erlaubt. Die erweiterten RMS Datenstrukturen sind selbst-identifizierend; es wird ähnlich vorgegangen bei bei den Stringdeskriptoren (siehe Abschnitt 9.12.2): Wenn die 32-bit große Pufferadresse -1 ist, dann findet sich die 64-bit Adresse in einem Feld, das an die alten Datenstruktur angehängt wurde.

Die erweiterte Datenstruktur ist seit *OpenVMS* Version 7.0 verfügbar, RMSAccess entscheidet daher nur anhand der Betriebssytem-Version, ob es diese benutzt – nicht anhand der Adressen! Probleme entstehen hierdurch keine, da bei Verwendung einer Version vor 7.0 auch keine 64-bit Adressen auftreten können.

10.7. Schnittstelle zu X 11

Ein großes Problem stellte die Schnittstelle zum X Window System (X 11 R 6) dar: sie akzeptiert keine 64-bit Zeiger oder Ganzzahlen; laut Digital ist dies auch nicht geplant.[60] Darum müssen nun alle Parameter auf 32 Bit konvertiert werden; Daten müssen ggf. sogar in den 32-bit Speicher umkopiert werden.

Alle Parameter, die per Referenz übergeben werden, müssen hierzu auf den Stack kopiert werden, auch wenn es sich um große Datenblöcke handelt; bei Rückgabeparametern müssen die Daten anschließend wieder zurückkopiert werden. Für Datenblöcke, deren Größe zur Compilezeit nicht bekannt ist, muß Platz auf dem Stack dynamisch geschaffen werden – ein ziemlicher „Hack“.

Vorgenommen werden müssen diese Änderungen in allen Modulen, die X11 verwenden. Dazu gehören nicht nur Module des System-Kerns (u.a. Display, Fonts, Input), sondern auch „Anwendungs“-Module[61]; momentan betroffen ist das Modul Pictures von der Universität Linz [Pic]. Anhang A enthält eine Anleitung, um Module schrittweise anzupassen.

Das Kopieren der großen Datenblöcke ist in keinster Weise portabel und setzt genaue Informationen über die internen Datenstrukturen voraus. Aber es spart wertvolle Zeit: Die Allokation auf dem Stack ist wesentlich schneller als im Heap. Und manche dieser Prozeduren werden sehr häufig gerufen, z. B. DrawString.

Für alle Prozeduren aus X 11 einen 'Wrapper' zu schreiben, ist nicht möglich: Die Größe der Datenblöcke ist nicht immer verfügbar oder der zusätzliche Prozeduraufruf wäre zu ineffizient.

Das Zwischenspeichern der Parameter auf dem Stack ist eine Lösung, solange die Stacks im 32-bit Speicher liegen. Diese Einschränkung kann also erst aufgehoben werden, wenn X 11 mit 64-bit Daten zurechtkommt.

Diese Erweiterung des X 11-Clients könnte aber nur dessen Beschränkung auf den 32-bit Speicherbereich aufheben.[62] Weiterhin bestehen blieben die Beschränkungen, die das Client-Server-Protokoll mit sich bringt, etwa die maximale Größe von Bitmaps.

[60] Nach Abschluß der Programmierarbeiten gab Digital bekannt, daß sie X 11 nun doch auf 64 Bit erweitern wollen.
[61] Soweit man in Oberon überhaupt von solchen sprechen kann.
[62] Womit hier allerdings viel gedient wäre.

10.8. Hausgemachte Probleme?

Wie auf Seite 10 beschrieben, beschränkt der *OpenVMS*-Linker statische Daten und den Programmcode auf den 32-bit Speicher, AOS kennt diese Einschränkung nicht. Eine Reihe der bisher genannten Probleme rühren daher, daß Betriebssystemdienste (incl. X 11) keine 64-bit Adressen unterstützen; darum wird kurz untersucht, ob die Freizügigkeit von AOS ein Grund hierfür sein kann:

Stacks und Ausnahmebehandlung: Dieses Problem tritt auch in stand-alone Anwendungen auf, die Coroutinen (oder ähnliches) benutzen, falls diese die Stacks im 64-bit Adreßbereich allozieren möchten.

Prozedurdeskriptoren und Ausnahmebehandlung: Dieses Problem kann bei stand-alone Anwendungen nicht auftreten, da 'Linkage-Pairs' als statische Daten immer im 32-bit Speicher liegen.

X 11: Dieses Problem wird nur zum Teil gelöst, wenn statische Daten – und damit globale Variablen – im 32-bit Speicher liegen: große Datenblöcke werden aber auch im 64-bit Speicher alloziert.

Die Entwickler bei Digital gehen offensichtlich davon aus, daß nur einzelne Programmteile, die den 64-bit Adreßraum brauchen, auf 64 Bit umgestellt werden [BNP96] – nicht ganze Anwendungen von der Komplexität des Oberon-Systems. Je nach Betrachtungsweise wird ja auch nur die Modulbibliothek des Oberon-Systems erweitert, um nach oben hin eine 64-bit Schnittstelle zu gewährleisten – wie es die DEC C Laufzeitbibliothek auch tut: Die Anwendungen im Oberon-System fallen dann unter „simple“ gemäß dem Eingangszitat vom Abschnitt 7.

10.9. Untersuchung des Speichermehrbedarfs

Zum Abschluß wird noch untersucht, wie stark der Speicherbedarf durch die Erweiterung gestiegen ist:

Der Speichermehrbedarf des „neuen“[63] 32-bit ALPHA OBERON SYSTEMS im Vergleich zum „alten“ ist marginal: lediglich bei den in Abschnitt 9.16 beschriebenen Konstrukten werden einige Bytes mehr Code erzeugt. Sie kommen aber nur in low-level Modulen des Oberon-Systems vor und auch dort nur relativ selten.

Der Speichermehrbedarf des 64-bit ALPHA OBERON SYSTEMS im Vergleich zur „alten“ 32-bit Version beträgt zwischen ca. 9,2 Prozent und ca. 30 Prozent – abhängig von den geladenen/laufenden Anwendungen: Der höhere Wert wurde direkt nach dem Start des Oberon Systems ermittelt, der niedrigere nach dem Öffnen des 'Kepler'-Bilds 'Palette'.

'Kepler' [Tem] ist eine Vektorgraphik-Anwendung, 'Palette' dient als 'Toolbox' zum Zeichnen von 'Kepler'-Objekten – enthält also nur wenige Objekte.

Woher rührt dieser doch eher geringe Mehrbedarf? Es wäre zu erwarten gewesen, daß er viel stärker steigt!? Woher kommt der große Unterschied innerhalb eines Modus?

[63] D.h. erzeugt mit dem neuen Compiler, nach dem Einbau der Änderungen.

Struktur	/PointerSize	Größe (incl. Tag)		Speicherblock	
ParcDesc	32	168		192	
	64	320	+ 90%	320	+ 66%
PieceDesc	32	36		64	
	64	56	+ 55%	64	+ 0%
DisplayMsg	32	40		64	
	64	72	+ 80%	96	+ 50%

Abbildung 18: Speichermehrbedarf bei einigen Datenstrukturen

- Über 50 % des Speichers wird vom Programmcode belegt. Hinzukommen noch andere Moduldaten, deren Größe sich nur durch starke Eingriffe[64] ins System ermitteln ließe, etwa die 'Linkage-Sektion'.
- Dateipuffer werden mit Unix.Malloc im 32-bit Speicher alloziert und entziehen sich der Statistik. Sie fallen aber nicht sehr ins Gewicht: Zum Zeitpunkt der Messungen waren 4 Dateien geöffnet mit je einem 4 KB Puffer.
- Gerade im low-level Bereich des Oberon-Systems wird oft INTEGER oder SHORTINT verwendet, wenn der Wertebereich ausreicht. Diese beiden Typen wurden aber nicht verändert, insbesondere nicht vergrößert.
- Benachrichtigungs-Strukturen ('Notification Messages'), die im Oberon-System häufig verwendet werden, werden üblicherweise auf dem Stack angelegt, nicht im Heap.
- Bei der Allokation von dynamischem Speicher entsteht Verschnitt: Es werden immer Blöcke von 32 Bytes vergeben; für das Record-Kennzeichen ('tag') werden immer 8 Bytes benötigt, für dynamische Arrays 24 Bytes – unabhängig vom Modus. Bei kleinen Strukturen ist der Verschnitt also überproportional groß.

  Abbildung 18 zeigt einige Beispiele von häufig benutzten Strukturen:

  Textframes.ParcDesc beschreibt einen Paragraphen in einem Text und enthält 38 LONGINT-Elemente,

  Texts.PieceDesc dient der Verwaltung von Textstücken

  Textframes.DisplayMsg ist eine typische Benachrichtigungs-Struktur, die nur auf dem Stack angelegt wird.

  Gut zu erkennen ist, daß der Verschnitt die Vergrößerung relativiert oder gar auffangen kann.
- 'Kepler' besteht aus 10 großen Modulen, 'Palette' enthält aber nur wenige Objekte. Damit erhöht sich in der Meßkonstellation der Anteil der Daten sehr, die von /PointerSize unabhängig sind – z. B. Programm-Code und 'Linkage-Sektion'.

[64] Um an diese Daten zu gelangen, mußte die Schnittstelle von Modules geändert werden. Als Folge müßte das gesamte Oberon-System neu kompiliert werden.

11. Zusammenfassung

Die moderate Spracherweiterung ist eine geeignete 64-bit Erweiterung für Oberon. Sie fügt sich gut in die Sprache ein und bedurfte nur zweier Erweiterungen der Syntax. Die Umsetzung von AOS auf 64 Bit zeigt, daß die gewählten Spracherweiterungen sinnvoll sind: Lediglich die low-level Module mußten – wegen Problemen mit dem Betriebssystem teils aber aufwenig – geändert werden, die Klienten des Oberon-Systems liefen ohne Änderungen nach Neukompilieren.

Die Umsetzung zeigt aber auch, daß verbleibende 32-bit Restriktionen in *OpenVMS* eine volle Portierung auf 64 Bit erschweren: allem voran sei hier das X 11 Window System genannt. Diese lassen sich auch nur zum Teil umgehen, wenn statische Daten und der Programmcode weiterhin im 32-bit Speicher liegen – wie es der *OpenVMS*-Linker bewirkt. Hier wird aber deutlich, daß Digital davon ausgeht, daß nur Anwendungen (und die Laufzeitbibliotheken) erweitert werden, die eine reellen Nutzen von 64 Bit haben [BNP96].

Ingesamt kann das Projekt als Erfolg gewertet werden: die zahlreichen Erweiterungen sowie viele umfangreiche V4 Zusatzanwendungen laufen problemlos – fast 400 Module!

Der Mehrbedarf an Speicher durch die komplette Umstellung des ALPHA OBERON SYSTEMS auf 64-bit Zeiger und 64-bit Ganzzahlen ist moderat.

Die anfangs gesetzten Ziele wurden ohne Einschränkung erreicht.

12. Ausblick

12.1. Verbesserte Speicherverwaltung

Noch immer muß beim Start des Oberon-Systems angegeben werden, wie groß der Speicherblock sein soll, der vom Betriebssystem angefordert werden soll. Damit läßt sich der 64-bit Adreßraum momentan noch nicht dynamisch nutzen.

Abhilfe könnte einen kleine Erweiterung des Garbage Collectors schaffen: dieser sollte, sobald der angeforderte Speicherblock voll ist, einen weiteren vom Betriebssystem anfordern. Ein ähnlicher Garbage Collector existiert bereits für Oberon4Amiga [O4A]. Dieser gibt die Speicherblöcke sogar wieder an das Betriebssystem zurück, wenn sie komplett ungenutzt sind.[65]

Eine verbesserte Speicherverwaltung würde fast automatisch auch eine Lösung beinhalten für die in Abschnitt 9.14.2 Punkt 3 genannten Probleme bei der Adreßrechnung.

12.2. Erweiterung des Modula-2-Compilers

Der Modula-2-Compiler MAX wurde nur minimal erweitert, um A2O auf 64-bit Ganzzahlen umzustellen. Ein nächster Schritt wäre nun, auch MAX auf 64 Bit umzustellen.

Modula-2 kennt zwar weniger Ganzzahltypen (einen vorzeichenlosen und einen vorzeichenbehaften), aber zusätzlich Unterbereichstypen (Subranges). Bei der Erweiterung würden die beiden Ganzahltypen komplett auf 64 Bit umgestellt und die bisherigen 32-bit Typen als Subranges implementiert. Damit sollten die meisten Probleme der Typ-Kompatibilität gelöst sein.

[65] AOS könnte hierauf verzichten, da jeder Prozeß seinen eigenen virtuellen Adreßraum hat. Beim Amiga teilen sich alle Prozeße den gleichen Adreßraum.

A. Anleitung zum Ändern von X-11-Klienten

Da das X Window System (X 11 R 6) von *OpenVMS* keine 64-bit Zeiger oder Ganzzahlen akzeptiert, müssen die Module, die X 11 benutzen, geändert werden. Dabei wird schrittweise vorgegangen:

1. Importiere zusätzlich Unix.
2. Deklariere eine Typ X11_LONGINT = X11.LONGINT.
3. Ersetzte SYSTEM.ADR in Aufrufen von X-11-Prozeduren durch Unix.Adr32.

   Damit werden eventuell verbliebene Fehler zur Laufzeit aufgefangen.
4. Prüfe, ob die X-11-Prozedur Variablenparameter erwartet. Wenn ja, ändere entsprechend dem Beispiel in Abschnitt A.1 (unten).
5. Prüfe, ob große Datenblöcke an X 11 übergeben werden. Ein typisches Anzeichen hierfür ist die Verwendung von SYSTEM.ADR (bzw. Unix.Adr32), worauf man sich aber nicht verlassen kann. Ändere entsprechend dem Beispiel in Abschnitt A.2.
6. Compiliere das Modul
7. Bei allen Fehlern, die von unterschiedlich großen Ganzzahltypen herrühren: Ändere die Typdefinition auf X11_LONGINT. Falls dies die Schnittstelle ändern würde, beachte das Beispiel in Abschnitt A.3.
8. Fahre fort bei 4), bis keine Fehler mehr moniert werden.
9. Falls nicht benötigt, entferne die Deklaration von X11_LONGINT und den Import von Unix.
10. Teste das Modul ausführlich, um übersehene Probleme aufzuspühren.

A.1. Die X 11-Prozedur erwartet Variablenparameter

In diesem Fall muß sichergestellt werden, daß die übergebenen Variablen im 32-bit Speicher liegen; bis auf weiteres wird der Stack immer dort liegen.

Ersetze

```
PROCEDURE DoOne(VAR w: X11.Window): LONGINT;
BEGIN
  ...
  RETURN X11.Kill(w);
```

durch

```
PROCEDURE DoOne(VAR w: X11.Window): LONGINT;
VAR x11_window: X11.Window; result: LONGINT;
BEGIN
  ...
  x11_window := w;
  result := X11.Kill(x11_window);
  w := x11_window;
  RETURN result;
```

Wird ein Modul mit /POINTERSIZE=32 compiliert, so können die Daten nur im 32-bit Speicher liegen. Darum kann optimiert werden, falls die Übersichtlichkeit nicht zu sehr leidet:

```
PROCEDURE DoSomeThing(VAR window: X11.Window): LONGINT;
  VAR x11_window: X11.Window; result: LONGINT;
BEGIN
  ...
  IF SIZE(SYS.PTR) = SIZE(SYS.SIGNED_32) THEN
    RETURN X11.Kill(window);
  ELSE
    x11_window := window;
    result := X11.Kill(x11_window);
    window := x11_window;
    RETURN result;
  END;
```

Dies führt zu effizienterem Code, wenn mit /POINTERSIZE=32 compiliert wird: Das Ergebins der IF-Anweisung ist zur Compilezeit bekannt und wird optimiert, so daß nur einer der beiden Fälle im erzeugten Code erscheint.

A.2. Adressen großer Datenblöcke werden übergeben

Hier sind zwei Fälle zu unterscheiden:

Falls die Größe des Datenblocks fest ist, wird vorgegangen, wie bei Variablenparametern: es wird eine lokale Variable des entsprechenen Typs (notfalls ein ARRAY OF SYSTEM.BYTE) angelegt und die Daten dort hineinkopiert.

Falls die Größe des Datenblocks variabel ist, muß dynamisch Platz am Stack gemacht und die Daten in den so geschaffenen Raum kopiert werden. Die Berechnung des Stacks kann nicht in eine Prozedur ausgelagert werden, da nach deren Rückkehr der Stack wieder bereinigt wäre. Die Größe des Datenblocks ist vom Einzelfall abhängig.

Ersetze

```
PROCEDURE ConvertOne (base, width, height: LONGINT): X11.One;
BEGIN
  RETURN X11.CreateOne(base, width, height)
```

durch

```
PROCEDURE ConvertOne (base: LONGINT; width, height: LONGINT): X11.One;
  VAR one: X11.One; size, sp: SYS.SIGEND_64;
BEGIN
  IF SIZE(LONGINT) >= SIZE(SYS.SIGNED_64) & Unix.IsInP2Space(base) THEN
    size := (width DIV 8) * height; (* anhaengig vom Einzelfall *)
    size := SYS.VAL(SYS.SIGNED_64, SYS.VAL(LONGSET, size+0FH) * LONGSET{4..63});
    S.GETREG(30, sp); SYS.PUTREG(30, sp - size);
    SYS.MOVE(base, sp, size);
    one := CreateOne(display, primary, SYS.VAL(X11_LONGINT, sp), width, height);
    S.PUTREG(30, sp);
  ELSE
    one := CreateOne(display, primary, SYS.VAL(X11_LONGINT, base), width, height)
  END;
  RETURN one
```

A.3. Die Schnittstelle ist betroffen

Die Schnittstelle der Prozedur soll nicht geändert werden, darum wird der Eingabewert beim Aufruf der X-11-Prozedur ggf. auf 32 Bit gekürzt. Dieses Muster ist dann einsetzbar, wenn die X-11-Prozedur keinen Variablenparameter erwartet. Das Ergebnis der IF-Anweisung ist zur Compilezeit bekannt und wird optimiert, so daß nur einer der beiden Fälle im erzeugten Code erscheint.

Ersetze

```
PROCEDURE DoTwo(pixels: LONGINT);
BEGIN
  ...
  X11.Call(pixels);
  ...
```

durch

```
PROCEDURE DoTwo(pixels: LONGINT);
BEGIN
  ...
  IF SIZE(LONGINT) = SIZE(X11_INTEGER) THEN
    X11.Call(pixels);
  ELSE
    X11.Call(SYS.SHORT(pixels));
  END;
  ...
```

B. Glossar

Ausnahme (engl.: exception) Tritt auf, wenn etwas Unerwartetes passiert, z. B. ein Programm versucht, nicht vorhandenen Speicher zu benutzen.

Bootlader Programm, das die für den ↑Bootstrap benötigten Module in den Speicher lädt und die Programmkontrolle dorthin übergibt. ↑Bootstrap

Bootstrap Wie Münchhausen sich an den Haaren aus dem Sumpf gezogen hat, lädt ein System sich selbst in den Rechner. Hierzu muß ein möglichst kleiner, aber ausreichender Teil des Systems in den Rechner gelangen. Beim Oberon System ist dies Aufgabe des ↑Bootladers.

call-back Beim Aufruf einer (System-)Routine wird dieser eine Prozedur übergeben, die später von der Routine aufgerufen wird. Da die Prozedur dann im Kontext des Programms abläuft, spricht man von einem call-back.

Code-Sektion ↑Programm-Sektion, die den Programmcode enthält.

DEC Digital Equipment Corporation, USA.

Exception-Handler Routine, die aktiviert wird, wenn eine ↑Ausnahme auftritt. Sie soll die Ausnahme bearbeiten und z. B. eine Fehlermeldung ausgeben.

Garbage Collector Speicherbereinigung. Sorgt dafür, daß Speicherbereiche als unbelegt markiert werden, wenn sie durch keine Zeiger mehr erreichbar sind.

Heap Zur Verfügung stehender Hauptspeicher, hier: Speicher, den das Oberon System zur Verfügung hat.

kanonische Darstellung einheitliche Darstellung entsprechend den Regeln. Bei Alpha Prozessoren werden vorzeichenbehaftete 32-bit Ganzzahlen in Registern immer als 64-bit Ganzzahlen dargestellt. Unter *OpenVMS Alpha* werden 32-bit Adressen als vorzeichenbehaftete Ganzzahlen in die Register geladen.

Linkage-Pair Wird unter OpenVMS benötigt, um eine Prozedur aufzurufen. Enthält die Adressen des zugehörigen ↑Prozedurdeskriptors und des Prozeduranfangs (entry address).

Linkage-Sektion ↑Programm-Sektion, die die Informationen enthält, die zum Aufrufen von Prozeduren nötig sind.

Modul-Schlüssel Ein eindeutiges Kennzeichen, um übereinstimmende Schnittstellen sicherzustellen. Beim Kompilieren eines Moduls bekommt dessen Schnittstelle einen „Stempel“ (den Schlüssel). Beim Linken müssen alle Module, die diese Schnittstelle benutzen, ebenfalls diesen Schlüssel haben.

Modullader Der Teil des Oberon Systems, der für das Nachladen der Module zuständig ist.

Oberon Load File, OLF Gegenstück zur ↑Objektdatei, jedoch in einem eigenen Format, das auf die Bedürfnisse des ↑Modulladers und des Oberon Systems ausgelegt ist.

Objektdatei Datei in einem festgelegten Format, aus der der Linker ein ausführbares Programm zusammenstellen kann. Enthält alle hierfür benötigten Daten wie Programmcode und von außen benötigte Symbole.

pervasiver Typ (durchdringender Typ) Ein ↑vordefinierter Typ, der standardmäßig sichtbar ist. In manchen Programmiersprachen, z. B. Modula-2, können die Bezeichner dieser Typen nicht überdefiniert werden. Die pervasiven Typen in Oberon sind: SHORTINT, INTEGER, LONGINT, REAL, LONGREAL, CHAR, BOOLEAN und SET. ↑SYSTEM-Typ

Programm-Sektion Abschnitt in einer OpenVMS Objektdatei, der gewisse Attribute (z. B. read/write, read-only) hat. Der Linker setzt zusammengehörige Programm-Sektionen aneinander.

propagieren verbreiten, hier: Der Wert von Bit 31 wird nach Bit 63 propagiert: Die Bits 32 bis 64 bekommen den gleichen Wert wie Bit 31.

Prozedurdeskriptor Enthält Daten, die der OpenVMS Aufrufstandard [Dig96b] für jede Prozedur verlangt. Gibt z. B. an, wo der Programmcode der Prozedur beginnt und welche Register die Prozedur benötigt.

Stack-Pointer Prozessor-Register, das den Zeiger auf den Stack enthält.

stand-alone hier: Ohne Oberon System.
'stand-alone' Anwendungen benötigen das Oberon-System nicht und werden direkt aus der *OpenVMS*-Shell gestartet.

System-Service allgemeine Bezeichnung für eine Systemdienst-Routine, die OpenVMS zur Verfügung stellt, z. B. SYS$OPEN zum Öffnen einer Datei.

SYSTEM-Typ Ein ↑vordefinierter Typ, der aus dem compiler-internen Modul SYSTEM importiert werden muß, um ihn verwenden zu können. AOS kennt mehrere Fließkomma-Typen, die als SYSTEM-Typen zur Verfügungn stehen.

vordefinierter Typ Ein Typ, der im Compiler vordefiniert ist. Üblicherweise sind dies Ganzzahl- und Fließkommazahl-Typen, ein Typ für „Zeichen“ (charakter) und ähnliches.

Literatur

[BNP96] Thomas R. Benson, Karen L. Noel, und Richard E. Peterson. The OpenVMS Mixed Pointer Size Environment. *Digital Technical Journal*, 8(2), 1996.

[Dig94] Digital Equipment Corporation, Maynard, Massachusetts. *OpenVMS Programming Concepts Manual*, 1994.

[Dig96a] Digital Equipment Corporation, Maynard, Massachusetts. *OpenVMS Alpha Guide to 64-Bit Addressing and VLM Features*, Nov 1996. for V7.1.

[Dig96b] Digital Equipment Corporation, Maynard, Massachusetts. *OpenVMS Calling Standard*, Nov 1996. for V7.1.

[Dig96c] Digital Equipment Corporation, Maynard, Massachusetts. *OpenVMS Programming Concepts Manual*, 1996.

[Dot94] Günter Dotzel. Alpha AXP/OpenVMS Modula-2 and Oberon-2 Compiler Project. In Peter Schulthess (Hsg.) *Proceedings of the Joint Modular Languages Conference, University of Ulm, Germany, 28-30 September 1994*. Universitätsverlag Ulm, 1994.

[Goe96] Hartmut Goebel. Portierung von Oberon auf Alpha AXP. Studienarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany, 1996.

[Hof95] Markus Hof. A Run Time Debugger for Oberon – User's Guide and Programming Interfaces. Technical Report 5, Institut für Informatik, Johannes Kepler Universität Linz, 1995. `ftp://ftp.ssw.uni-linz.ac.at/pub/Reports/Report5.ps.Z`.

[Mös91] Hanspeter Mössenböck. The Programming Language Oberon-2. Technical Report 160, Department Informatik, ETH Zürich, 1991.

[O4A] Oberon4Amiga. `ftp://ftp.inf.ethz.ch/pub/software/Oberon/OberonV4/Amiga/1.4/`.

[OOC] Mailinglist 'OOC'. `http://www.uni-kl.de/OOC/files/mailing-list/`.

[Pic] Oberon Ascii-Coded file. `ftp://oberon.ssw.uni-linz.ac.at/pub/Oberon/LinzTools/MacPicElems.Cod`.

[Sie] Fridtjof Siebert. AmigaOberon Demo. `ftp://ftp.inf.ethz.ch/pub/software/Oberon/OberonV4/Amiga/AmigaOberonDemo/`.

[Smi96] Duane A. Smith. Adding 64-bit Pointer Support to 32-bit Run-time Library. *Digital Technical Journal*, 8(2), 1996.

[Tem] Josef Templ. Oberon Ascii-coded file. `ftp://oberon.ssw.uni-linz.ac.at/pub/Oberon/LinzTools/Kepler.Cod`.

[WG92] Niklaus Wirth und Jürg Gutknecht. *Project Oberon: The Design of an Operating System and Compiler.* Addison-Wesley, 1992.

[Wir92] Niklaus Wirth. *The programming language Oberon*, pages 281–305. Addison-Wesley, 1992.

Index

Kursive Seitenzahlen markieren die Haupt-Erklärungen, Sterne markieren Einträge im Glossar.

A
A2O 1, 8–10, 16, 24, 25, 29, 31–33, 35, 36, 38–41, 44, 45, 49, 56
A2OLayout (Modul) 48, 50
Alpha Oberon 29, *siehe* A2O, AOS, AOS_64
Alpha Prozessor 17, 24, 36, 44, 45
AOS IV, V, 1, 2, 7, 9–11, 17, 20, 31, 39, 41, 47–49, 51, 53, 55, 56, 61
AOS_64 48, 50
Arraydeskriptor 47
Ausnahme 60*
-behandlung 47, 50, *51*, 53
-behandlungsroutine *siehe* Exception-Handler

B
Bootlader *48–49*, 50, 60*

C
Code-Sektion 60*
Compiler
-Optionen *siehe* /POINTERSIZE
DEC C *29*, 53
Modula-2 *siehe* MAX
Oberon *siehe* A2O
Coroutines (Modul) 51

D
Darstellung
kanonische 24
Deskriptor *siehe* Prozedurdeskriptor, Typdeskriptor, Arraydeskriptor, Stringdeskriptor
Directories (Modul) 51
Display (Modul) 52
dynamischer Speicher *siehe* Heap

E
Exception *siehe* Ausnahme
Exception-Handler 60*

F
Fonts (Modul) 52

G
Ganzzahltypen *siehe* Typhierarchie
Garbage Collector *50*, 60*
GC *siehe* Garbage Collector
gemischte Verwendung 34, 47, *siehe* Abschnitt 7

H
Heap 10, 34, 50, 60*

I
Input (Modul) 52

K
kanonische Darstellung 24, 60*
Kernel (Modul) 34
ALLOCATE_32 *49–50*
ALLOCATE_64 *49–50*
Kompatibilität
32/64 Bit 11–12, 23, *48*

L
Laden *siehe* Modul
LIB$GET_VM 35, 50
LIB$GET_VM_64 35
Linkage-Pair 41, 60*
Linkage-Sektion 51, 60*
Linker *siehe* Modullader
Beschränkung *10*, 41, 48, 53
Listaeral
Sets 19
Literal 10, 21, 31, 32, *32*, 42, 47
Hexadazimal *32*
Hexadezimal *16–17*
Set *18*, 19
Loadfile *siehe* OLF

M
Malloc *siehe* Unix.Malloc
MAX IV, *31*, 32, 33, 56
Modul-Schlüssel 37, 39, 60*
Modula-2 1, *31*, 39
Compiler *siehe* MAX
Modulbibliothek 47, 53
Modules (Modul) 54
Modullader *48–49*, 60*

N

NEW *34–35*, 49, *siehe* SYSTEM.NEW

O

Oberon
  Report 7, 17
  System I, III, V, 1, 2, 7, 9, 10, 22, 34, 39, 47–51, 53–56, 61
   Alpha Oberon *siehe* A2O, AOS, AOS_64
Oberon Load File II, 2, 39, 40, 61, 61*
  Format IV, 40, *40*
Objektdatei *39*, 61*
  Format *39*
OLF *siehe* Oberon Load File
OpenVMS I, IV, V, 2, 4, 5, 7, 10, 22, 24–26, 35, 39–41, 48–53, 55, 57, 61
  Alpha V, 1, 2, 4–6, 8, 10, 16, 40, 41, 47, 60
  VAX 4
Optionen
  Compiler- *siehe* /POINTERSIZE

P

pervasiver Typ 61*
Pictures (Modul) 52
/POINTERSIZE II, 11, 12, 19, 23, 24, 26, 29, 32–34, 37, 40, 43, 45, 49, 54
/POINTERSIZE=32 13, 15, 16, 19, 25, 26, 33, 39, 42, 45, 48, 58
/POINTERSIZE=64 12, 13, 15, 16, 26, 34, 37, 39, 42–45, 47, 49
Portierung
  in zwei Stufen 47
  Vorgehen 47
  Ziele 47
Probleme
  hausgemacht *53*
Programm-Sektion 61*, *siehe* Code- und Linkage-Sektion
Prozedurdeskriptor 40, 51, 53, 61*

R

Record Management Service (RMS) *51–52*
RMSAccess (Modul) 51, 52

S

Schlüssel *siehe* Modul-Schlüssel, Typdeskriptor
Set *36*, 50
Speicher
  -verwaltung *siehe* Garbage Collector
  dynamischer *siehe* Heap
SS$_ARG_GTR_32_BITS 25
Stack *50–51*, 51–53
stand-alone 2, 10, 34, 39, 41, 53, 61*
Storage64 (Modul) 35
  ALLOCATE 35, *49–50*
  ALLOCATE_32 35, *49–50*
Stringdeskriptor *41–42*, 52
Symboldatei 32, *36–37*
  Behandlung *39*
  Format *37–38*
Syntaxänderung *21*
  Hexadezimal-Zahlen *16–17*, 24, 32
  Sets *19–20*, 24, 36, 50
SYS$PUT_MSG 51
System (Modul) 51
SYSTEM.NEW 35, 49, *siehe* NEW

T

Typdeskriptor 47, 49
Typhierarchie 1, 16

U

Unix (Modul) 50, 51, 57
  Malloc *50*, 51, 54

V

Vorbereitung
  in 32-bit Version *47*
Vorgehen
  Portierung *47–48*

X

X Window System *siehe* X 11
X11 (Modul) 52
X 11 II, V, 22, 48, 51, *52*, 52, 53, 55, 57