Diplomarbeit

Ein Debugger für Oz

Benjamin Lorenz

Entworfen und implementiert unter Betreuung von
Prof. Dr. Gert Smolka, Dipl.-Inform. Ralf Scheidhauer
und Dipl.-Inform. Leif Kornstaedt am
Fachbereich Informatik der Universität des Saarlandes

8. April 1999

Hiermit versichere ich, daß ich die vorliegende Arbeit selbständig und lediglich unter Verwendung der im Literaturverzeichnis angegebenen Referenzen erstellt habe.

(Benjamin Lorenz)

Ein Debugger für Oz

Diplomarbeit

Benjamin Lorenz
lorenz@ps.uni-sb.de
Fachbereich Informatik
Universität des Saarlandes

Zusammenfassung

Im Rahmen des Software-Entwicklungsprozesses nimmt die Fehlersuche eine wichtige Stellung ein. *Syntaktische* Fehler können bereits vom Compiler entdeckt werden, *semantische* Fehler dagegen sind häufig schwer zu finden; der Zeitbedarf hierfür kann enorm hoch sein. In vielen professionellen Programmierumgebungen existiert daher ein leistungsfähiges Werkzeug, ein *Debugger*, der die Fehlersuche erleichtert, indem das fehlerhafte Programm an beliebigen Stellen angehalten und zusammen mit seinen Daten untersucht werden kann.

In dieser Arbeit beschreibe ich den Entwurf eines Debuggers für die Programmiersprache Oz sowie seine Implementierung im Programmiersystem Mozart.

Danksagung

Nach einer langen Zeit harter und nicht selten recht nervenaufreibender Arbeit kann ich heute mit Stolz auf ein Stück Programm blicken, das bereits mehreren Leuten gute Dienste geleistet hat und dessen Benutzung mir selbst immer wieder großen Spaß bereitet. Ohne die breite Unterstützung, die ich von so vielen Personen erhalten habe, wäre die Arbeit allerdings sicher nicht das geworden, was sie heute ist. Ihr Lieben, habt Dank für all Eure Hilfe!

Explizit nennen möchte ich Patrick Gebhard, der mich regelmäßig spätestens um halb eins mittags aus dem Bett klingelte, damit ich noch rechtzeitig in der Mensa zum Frühstück erscheinen konnte, Leif Kornstaedt, der seinerseits Ozcar mit Erfolg einsetzt und daher stets konstruktive Kritik einbringen konnte, Michael Mehl und Ralf Scheidhauer, die aufgrund ihrer tiefen Kenntnisse der Innereien des Mozart-Systems mehrmals absolut unerklärliche Phänomene, über deren Ursache ich mehrere Nächte grübeln mußte, innerhalb von Minuten aufklären konnten, Joachim Niehren, der zu dieser Arbeit den ersten Satz der Einleitung beigesteuert hat, alle übrigen Mitarbeiter des Lehrstuhls, die allesamt zur freundlichen, ja auch freundschaftlichen Atmosphäre beitragen, die ich häufig empfinde, und natürlich Prof. Dr. Gert Smolka, der mir dieses reizvolle Thema angeboten hat und mich durch Lob, Diskussion und manchmal auch leichten Druck immer wieder aufs neue motiviert hat, „am Ball“ zu bleiben.

Sprachkonventionen

Viele Begriffe aus der Computer- und insbesondere der Debuggerwelt kommen aus dem Englischen. Ich habe versucht, möglichst viele Worte zu übersetzen, sofern eine gute deutsche Übersetzung existiert. „Haltepunkt“ etwa ist ein perfektes Äquivalent zu „breakpoint“. Allerdings habe ich mich an verschiedene Worte derart gewöhnt, daß auf mich ihr deutsches Pendant zu fremd wirkt, als daß ich es guten Gewissens benutzen könnte. „Stack“ ist ein Beispiel hierfür.

Weiterhin verwende ich in dieser Arbeit – der besseren Lesbarkeit zuliebe – durchweg männliche Personenbezeichnungen. Meine Leserinnen mögen Nachsicht zeigen und sich ebenso angesprochen wissen!

Inhaltsverzeichnis

1

Einleitung

Irren ist menschlich. Selbst wenn bei der Übersetzung eines neuen Programms kein Fehler gemeldet wird, ist es deshalb recht unwahrscheinlich, daß das Programm zu hundert Prozent im Sinne des Programmierers funktioniert. Neu geschriebene Programme enthalten, vor allem dann, wenn sie etwas komplexer sind, neben formalen Fehlern (die meist schon vom Compiler entdeckt werden) auch noch eine zweite Fehlerart, nämlich Laufzeitfehler, die sich erst bei der Ausführung des Programms bemerkbar machen und ihren Ursprung entweder in Denkfehlern des Programmierers oder in Mißverständnissen zwischen Compiler und Programmierer haben. Die Lokalisierung von Fehlern und ihre Beseitigung wird im allgemeinen als Debugging bezeichnet. Dieser Terminus und seine Derivate wurden eigentlich für eine dritte Fehlerart erfunden, die sich heutzutage nur noch selten findet: In der Urzeit der Computerei hatte sich einmal ein toter Käfer (engl. „bug"), der zwischen zwei Relaiskontakten eingeklemmt war, als die Ursache eines äußerst rätselhaften Fehlers entpuppt.[1] Weil sich die Suche nach Fehlern durch bloßes Betrachten eines Programms und entsprechende Denkarbeit in vielen Fällen recht schwierig gestaltet, verwenden Programmierer einen Debugger, mit dem sich das zu untersuchende Programm schrittweise ausführen, an vorgegebenen Stellen unterbrechen und zusammen mit seinen Daten untersuchen läßt.

1.1 Begriffsklärungen

Im Gegensatz etwa zu Compilern sind Debugger relativ wenig untersuchte Werkzeuge in der Informatik. Die Literaturrecherche im Vorfeld dieser Arbeit brachte jedenfalls vor allem Papiere über Algorithmen und Implementierungen von Debuggern für spezielle Sprachen und Anwendungen zu Tage, weniger grundsätzliche Überlegungen zum Thema Fehlersuche. Dies mag daran liegen, daß das Aufgabengebiet von Debuggern nicht exakt definiert und abgegrenzt ist. Welche Typen von Fehlern sollen sie finden helfen? In welchem Abschnitt des Softwareentwicklungsprozesses sollen sie eingesetzt werden? Wo endet Debugging und wo beginnen Tätigkeiten wie Optimieren, Verifizieren oder Testen? Dies sind nur einige der Fragen, auf die unterschiedliche Entwickler unterschiedliche Antworten geben. Unstrittig dagegen ist die Tatsache, daß das Suchen nach Fehlern zu den zeit- und kostenintensivsten Arbeiten bei der Programmerstellung zählt [Ros96].

[1] Das besagte Tierchen soll sich heute, einer Geschichte Grace Hoppers zufolge, in einem Museum in Virginia befinden, wo es als erster Bug der Computergeschichte bewundert werden kann [Ros96].

Was ist ein Bug? Im engeren Sinne versteht man unter einem Bug einen Programmierfehler, der sich zur Ausführungszeit des Programms bemerkbar macht. Dies impliziert, daß es sich nicht um einen formalen Fehler handelt, denn das Programm konnte ja bereits ordnungsgemäß übersetzt werden. Man kann die Definition jedoch abschwächen und im Rahmen des Softwareentwicklungsprozesses ebenso Fehler in der Anforderungsanalyse und im Programmdesign als Bugs bezeichnen [Mül83].

Was bedeutet Debugging? Der Begriff „Debugging“ bezeichnet die Tätigkeit der *Lokalisierung* und *Beseitigung* von Fehlern in Softwaresystemen. Nicht verwechselt werden darf er mit dem verwandten Begriff des *Testens*. Beim Testen wird das Programm mit verschiedenen Eingaben (insbesondere kritischen Randwerten) gestartet und seine berechneten Ausgaben (Ist-Werte) mit den gemäß der Spezifikation erwarteten Ausgaben (Soll-Werte) verglichen. Abweichungen deuten auf Fehler hin, die durch Debugging gefunden und beseitigt werden können. Ein neuer Testlauf, *Regressionstest* genannt, muß anschließend stattfinden, um die erfolgreiche Korrektur der Fehler nachzuweisen und sicherzustellen, daß sich keine neuen Bugs eingeschlichen haben.

Wozu dient ein Debugger? Zunächst ist ein Debugger ein Werkzeug, das den Softwareentwickler bei der Suche nach Fehlern unterstützt. Darüber hinaus kann er helfen, ein Programm besser zu verstehen, was insbesondere bei Wartungsaufgaben für Personen, die das Programm nicht selbst geschrieben haben, eine große Hilfe bedeutet. Jedoch ist ein Debugger kein Allheilmittel. Er kann die eigene Denkarbeit nicht ersetzen, vielmehr gilt der Grundsatz: „Erst denken, dann debuggen“.

1.2 Klassifikation

Es können zwei Debugging-Methoden unterschieden werden: *Statisches Debugging* beinhaltet jede Art von Fehlersuche in Dokumenten, also in der Anforderungsspezifikation, dem Design oder dem Quellcode.[2] *Dynamisches Debugging* hingegen bezeichnet die Fehlersuche in Programmen während oder nach ihrer Ausführung [Mül83, Fla97].

1.2.1 Statisches Debugging

Statisches Debugging basiert auf der Beobachtung, daß Compiler bei der statischen Analyse eine Reihe von Informationen ermitteln, die zur Fehlersuche nützlich sein können. So werden beispielsweise bei der (interprozeduralen) Datenflußanalyse Referenzen von Variablen verfolgt, um Zugriffe auf nicht initialisierte Variablen zu erkennen oder Schleifen zu optimieren. Weiterhin kennt der Compiler die Aufrufhierarchie der Prozeduren. Wird nun diese Information nach Abschluß der statischen Analyse nicht gelöscht, sondern in eine Datenbank aufgenommen, so kann diese dazu verwandt werden, einen Debugger zu konstruieren, der beispielsweise folgende Anfragen zu beantworten imstande ist: „An welchen Stellen im Programm wird die Variable X referenziert, ehe bei Ausführung des Programms Zeile 42 erreicht wird?“, „Wie sieht, beginnend bei der Prozedur P, der Aufrufgraph aus?“. Systeme, welche statisches Debugging implementieren, sind MAP [TSP83] und MrSpidey [Fla97]. Im Programmiersystem

[2] Die folgenden Betrachtungen beschränken sich auf Debugging im Quellcode.

OMEGA [PL83] wird der Schwerpunkt auf eine relationale Datenbank gelegt, die neben statischen Informationen über das Programm (Parsebaum, Symboltabellen, Unterschiede zwischen verschiedenen Versionen, Konfigurationsbeschreibungen, u. a.) auch Laufzeitinformation speichert.

In dieser Arbeit werde ich nicht weiter auf statisches Debugging eingehen und „Debugging“ synonym für „dynamisches Debugging“ gebrauchen.

1.2.2 Dynamisches Debugging

Dynamisches Debugging setzt zur Ausführungszeit des Programms an und ist die am häufigsten anzutreffende Methode der Fehlersuche. Während beim Testen Fehler durch ihre *Wirkung* auf das beobachtbare Programmverhalten gefunden werden, ist das Ergründen der *Ursache* für dieses (fehlerhafte) Verhalten das wichtigste Ziel beim dynamischen Debuggen [CR83].

In [Gol94] werden verschiedene, für das Verständnis des dynamischen Debuggings wertvolle Begriffe eingeführt. Die Existenz eines Fehlers macht sich in der Regel bemerkbar durch *Fehlverhalten* des Programms: Eine Berechnung liefert ein falsches Ergebnis, divergiert oder bricht mit einem Laufzeitfehler ab. Beobachtet wird dieses Verhalten an der *Fehlerstelle*. Die Kunst besteht nun darin, entlang des *Fehlerpfades* die Programmausführung zurückzuverfolgen, um den *Fehlerursprung* ausfindig zu machen. Schließlich muß eine *Fehlerkorrektur* stattfinden. Betrachten wir z. B. folgendes Oz-Programm (die Notation sollte selbsterklärend sein – im nächsten Kapitel wird die Sprache näher vorgestellt):

```
1  local
2     X = 0
3     fun {F N}
4        case N < 2 then N else
5           N * {F (N-1)}
6        end
7     end
8  in
9     {Show (100 div {F X})}
10 end
```

Das Programm verhält sich fehlerhaft, indem es mit dem Laufzeitfehler „division by zero“ an der Fehlerstelle (Zeile 9) abbricht. Der Fehlerpfad

```
100 div 0,  {F 0},  X = 0
```

führt zum Fehlerursprung (Zeile 2) und liefert die nötige Erkenntnis zur Fehlerkorrektur: Im Basisfall der Rekursion (Zeile 4) muß 1 zurückgegeben werden, nicht `N`.

1.2.3 Abstraktionsebenen

Die Möglichkeiten von Debuggern auf *Maschinenebene* beschränken sich im wesentlichen auf die Ausgabe der Maschinenregister, des Programmzählers sowie Datenbereichen des Speichers in tabellarischer Form. Sie existieren heutzutage nur noch vereinzelt, etwa als Kern-Debugger von Betriebssystemen [NBD97].

Debugger auf *Quellsprachebene* werden, weil sie auf Symbolen wie Variablen- und Funktionsnamen der Quellsprache arbeiten, auch *symbolische Debugger* genannt. Während der Compiler eine Transformation von Quellcode in Maschinencode durchführt, muß der symbolische Debugger eine (partielle) Umkehrabbildung von Maschinencode in Quellcode realisieren – eine nicht triviale Aufgabe. Doch der Gewinn für den Entwickler ist enorm, denn das Debugging-Modell kann an das Programmier-Modell angepaßt werden. Dadurch reduziert sich die Einarbeitungszeit in die Debuggerbedienung, Programmcode braucht nicht pro Maschinenbefehl, sondern kann pro Quelltextzeile oder Funktionsapplikation ausgeführt werden, und die Auswertung beliebiger Ausdrücke der Quellsprache in gegebenem Kontext ist möglich. Durch diesen Zugewinn an Abstraktion lassen sich viele Fehler wesentlich schneller finden.

Manche symbolischen Debugger stellen zusätzlich eine Maschinenansicht zur Verfügung, um auch solche Fehler finden zu helfen, die lediglich auf Assemblerebene sichtbar sind und unter Umständen auf Bugs in der Entwicklungsumgebung selbst (Compiler, Assembler, u. a.) hindeuten.

1.3 Schlüsselkonzepte

Im folgenden beschreibe ich die Schlüsselkonzepte, die in fast jeder Implementierung eines Debuggers auf die ein oder andere Weise umgesetzt werden.

1.3.1 Haltepunkte

Haltepunkte (engl. „breakpoints“) erlauben dem Benutzer, ein Programm an wohldefinierten Programmpunkten anzuhalten, um seinen Zustand zu untersuchen. *Quelltext-Haltepunkte* (engl. „code breakpoints“) stoppen die Programmausführung, wenn eine bestimmte Zeile im Quelltext erreicht wird, *Daten-Haltepunkte* (engl. „data breakpoints“ oder „watchpoints“) hingegen, wenn eine bestimmte Variable referenziert wird [Kep93]. Bei *bedingten Haltepunkten* wird die Programmausführung nur dann unterbrochen, wenn zusätzlich zu den genannten Bedingungen ein boolescher Ausdruck, der bei jedem Erreichen des Haltepunktes in der Laufzeitumgebung neu evaluiert wird, den Wert „wahr“ liefert.

Um Haltepunkte in hardwarenahen Sprachen wie C effizient implementieren zu können, bedarf es der Unterstützung auf Hardwareebene. Viele moderne Prozessoren bieten daher eine spezielle *Haltepunktinstruktion*. Kommt sie zur Ausführung, so wird der Debugger aktiviert und mit den nötigen Informationen über den aktuellen Programmzustand versorgt [Ros96]. Die Implementierung von Haltepunkten bei interpretierten Sprachen erweist sich als wesentlich einfacher, denn der Interpreter bietet die Möglichkeit, den Microcode der Instruktionen beliebig zu verändern.

1.3.2 Einzelschritt-Ausführung

Um den Programmablauf genau verfolgen zu können, gibt es in vielen Debuggern die Möglichkeit, das Programm in Einzelschritten ablaufen zu lassen. Festzulegen ist die Granularität der Ausführungsschritte. Es bieten sich hier verschiedene sinnvolle Möglichkeiten. Der GNU-Debugger [SP95] beispielsweise

faßt alle Anweisungen einer Quelltextzeile zu einem atomaren Schritt zusammen; der Emacs-Lisp-Debugger [LaL94] interpretiert – nicht unüblich bei funktionalen Sprachen – jeden Ein- und Austritt einer Funktion als Schritt.

Bei Prozedur- oder Funktionsapplikationen existieren grundsätzlich zwei Möglichkeiten, mit welcher Schrittweite fortzufahren ist: Die oftmals *step into* genannte Operation betritt die Prozedur, während *step over*[3] die Prozedurapplikation als atomaren Schritt behandelt und den aktuellen Skopus nicht sichtbar verläßt.

Eine einfache Möglichkeit, Einzelschritt-Ausführung zu implementieren, besteht im sukzessiven Setzen von (unsichtbaren) Haltepunkten jeweils an der Stelle, wo der Programmlauf nach Abarbeitung des aktuellen Schrittes wieder unterbrochen werden soll [Ros96]. Probleme ergeben sich jedoch bei Programmen, in denen Prozeduren mit und ohne Debuginformation wechselseitig aufgerufen werden. Derlei Probleme werde ich ausführlich in Kapitel 4.4 behandeln und ein besseres Prinzip zur Realisierung der gewünschten Funktionalität vorstellen. Es wird sich zeigen, daß die Operation *step over* lediglich einen Sonderfall der allgemeineren Operation *unleash* darstellt, welche die Programmausführung bis zum Erreichen beliebig anzugebender Segmente im Aufrufstack fortsetzt.

1.3.3 Kontextinformation

Die drei wichtigsten Fragen, die sich ein Programmierer nach Anhalten des Programms (sei es wegen einer Ausnahmesituation oder eines Haltepunktes) stellt, sind „Wo bin ich?“, „Wie kam ich hierhin?“ und „Was ist hier los?“.

Wo bin ich? Eine Antwort auf die erste Frage sind die zum aktuellen Programmzähler korrespondierenden Quelltextkoordinaten (Datei, Zeilen- und Spaltennummer). Viele Debugger besitzen daher eine *Quelltextanzeige*, die einen Ausschnitt des momentan „interessanten“ Quelltextstücks zeigt und die aktuelle Position besonders markiert. Bei integrierten Entwicklungsumgebungen wird zur Anzeige i. d. R. der zum System gehörende Editor benutzt, mit dem Vorteil, daß der Quelltext unmittelbar korrigiert werden kann.

Wie kam ich hierhin? Die zweite Frage zu beantworten hilft i. a. der Aufrufkeller. Die meisten Debugger bieten eine *Stackanzeige* mit der Möglichkeit, im Stack zu „wandern“. In Verbindung mit der Quelltextanzeige, die entsprechend dem momentan ausgewählten Kellereintrag angepaßt wird, läßt dies eine schnelle Orientierung zu.

Was ist hier los? Die dritte Frage schließlich kann durch Angabe der aktuellen Variablenbelegungen in der *Variablenanzeige* beantwortet werden. Zusätzlich gestatten manche Debugger die Auswertung beliebiger Ausdrücke in gegebenem Kontext.

1.4 Debugtechniken

Es werden viele unterschiedliche Techniken der Fehlersuche vorgeschlagen, von denen ich im folgenden einige interessante oder verbreitete vorstelle.

[3] Synonym existieren die Bezeichnungen *next* und *skip*.

1.4.1 Instrumentation

Das Protokollieren des Programmablaufs und die Analyse seines Zustands kann „von Hand“ durch den Programmierer oder automatisiert durch einen Debugger geschehen. In beiden Fällen muß auf Quellsprach- oder Maschinenebene der Code *instrumentiert*, d. h. um zusätzlichen *Debugcode* ergänzt werden [TA95].

Im ersten Fall werden dazu vorübergehend Ausgabeanweisungen in den Quellcode eingestreut, die die Werte verschiedener Programmvariablen ausgeben. Durch die sinnvolle Wahl der Variablen und der Ausgabepositionen im Quellcode, kombiniert mit einer Portion Scharfsinn, lassen sich bereits verschiedene Fehler finden; gerade in Kreisen von Programmierern mit nur mäßigen Debugkenntnissen erfreut sich diese Technik einiger Beliebtheit. Komfortabler wird es allerdings, wenn ein Debugger die Arbeit des Protokollierens übernimmt. Hierzu ist i. d. R. eine enge Zusammenarbeit mit dem Compiler notwendig, um die Instrumentation hinreichend effizient und effektiv gestalten zu können. Viele Systeme, etwa der GNU-Debugger gdb [SP95], verwenden diese Technik, um Variableninformation bereitzustellen, manche nutzen den Debugcode weiterhin zum Setzen von Haltepunkten, so z. B. der in [TA95] vorgestellte ML-Debugger.

Noch komfortabler wird es, wenn Teile der Ausführung eines Programms im Sinne einer „Undo“-Funktion rückgängig gemacht werden können. Im Idealfall sollte in der Programmausführung genauso „geblättert“ werden können wie in einer Textdatei [PL89]. Das System *Igor* kommt diesem Wunsch recht nahe [Fel89]. Regelmäßig beim Erreichen sogenannter *Überwachungspunkte* (engl. „checkpoints“) schreibt Igor ein Abbild des momentanen Programmzustandes in eine Datei. Auf diesen Daten bietet das System eine Reihe komfortabler Funktionen an, unter anderem ist es in vielen Fällen sogar möglich, ein abgestürztes Programm wieder „zum Leben zu erwecken“, indem es mit dem Zustand, der an einem beliebigen Überwachungspunkt vorlag, neu gestartet wird.

1.4.2 Algorithmisches Debugging

Diese Debugtechnik, alternativ *deklaratives Debugging* genannt, wurde ursprünglich im Kontext von Prolog entwickelt [Sha83, KSF90], später jedoch generalisiert, um sie auch auf andere Programmiersprachen anwendbar zu machen, vgl. z. B. [Gre94]. Es folgt eine Beschreibung ihres Basisprinzips.

Idee

Nehmen wir an, ein Programm hat einen Bug, der sich durch falsches Ein-/Ausgabeverhalten manifestiert. Nehmen wir weiter an, das Programm bestehe aus einer Vielzahl von ineinander geschachtelten Prozeduraufrufen. Dann gibt es zunächst zwei Möglichkeiten, den Bug zu lokalisieren: Entweder ist die initial aufgerufene Prozedur P fehlerhaft, oder aber eine der von dieser aufgerufenen Prozeduren $P_1 \ldots P_n$ produzierte ein falsches Ergebnis. Um herauszufinden, welcher der beiden Fälle vorliegt, werden die Ergebnisse der Prozeduraufrufe von $P_1 \ldots P_n$ auf ihre Richtigkeit geprüft. Sind sie alle korrekt, so ist die fehlerhafte Prozedur gefunden, nämlich P. Ansonsten wird das Spiel bei der ersten eine falsche Ausgabe erzeugenden Prozedur $P_i, 1 \leq i \leq n$, fortgesetzt, das Fehlverhalten also entlang der Aufrufhierarchie weiter verfolgt.

Algorithmus

Ein algorithmischer Debugger läßt das Programm zunächst komplett durchlaufen und baut einen Aufrufbaum auf. Anschließend findet ein Frage-Antwort-Spiel mit dem Benutzer statt: Der Debugger präsentiert eine Prozedurapplikation,

die einem Knoten im Aufrufbaum entspricht, und der Benutzer muß angeben, ob das Verhalten dieses Knotens korrekt ist. Wenn ja, so wird mit dem benachbarten Knoten weitergemacht, ansonsten am fehlerhaften Knoten im Baum abgestiegen. Die Suche endet bei einem fehlerhaften Knoten, der als Ursprung des Bugs vom Debugger ausgegeben wird, wenn er entweder ein Blatt des Aufrufbaumes ist, oder alle seine Unterknoten korrektes Verhalten an den Tag legen.

In der Praxis erweist sich deklaratives Debugging als nervenzehrende Angelegenheit, da in „Real-World"-Anwendungen eine sehr große Zahl von Fragen gestellt werden muß, selbst wenn durch Verwendung einer Datenbank Wiederholungen von Fragen ausgeschlossen werden [Nil94].

1.4.3 Zusicherungen

In verschiedenen Programmiersprachen gibt es das Konzept der *Zusicherung* (engl. „assertion"): An strategisch wichtigen Stellen im Programmcode werden Anweisungen eingefügt, die testen, ob eine bestimmte Invariante erfüllt ist. Ist dies nicht der Fall, so bricht das Programm unter Angabe der aktuellen Position und im Idealfall auch des Kontextes ab. Der Entwickler hat nun die Möglichkeit, in einem stark eingeschränkten Suchraum nach der Ursache des Fehlers zu fahnden. Eine Stärke dieses Verfahrens ist, daß der Quellcode zur Erzeugung des auszuliefernden Systems nicht verändert werden muß. Vielmehr bestimmt eine Compileroption, ob die Zusicherungen in den Code eingefügt werden oder nicht. Beispiele für Sprachen mit Zusicherungen sind Eiffel [ISE97], welches dieses Konzept in der Sprachdefinition verankert hat, und ANSI C, wo ein entsprechendes Makro in der Standard-Bibliothek existiert [ANS90].

1.5 Was Debugging schwierig macht

Analyse vs. Intuition

Nicht zu unrecht bezeichnen manche Programmierer Debugging eher als Kunst denn als Wissenschaft: Je versteckter ein Fehler im System sitzt, desto weniger sind systematische Suchstrategien geeignet, ihn aufzuspüren. Oft führt eine gesunde Mischung aus *Intuition* („mein Gespür sagt mir, der Fehler liegt eher hier als dort, ...") und *Analyse* („diesen Wert dürfte die Variable X an dieser Stelle gar nicht haben") zum Ziel. Der häufige Wechsel zwischen diesen beiden Denkweisen erweist sich als schwierig [Gra83].

Verfügbarkeit

Viele Programmiersysteme werden zunächst ohne oder mit nur unzureichender Debugging-Unterstützung ausgeliefert. Die Notwendigkeit, rechtzeitig im Entwicklungsprozeß über eine geeignete Debugger-Infrastruktur nachzudenken, wird leicht unterschätzt. Die nachträgliche Realisierung der verlangten Eigenschaften ist teuer, führt zu einer schlechten Integration ins System und kostet vor allem Zeit. Außerdem wird vom Markt in dieser Hinsicht kaum Druck auf die Hersteller ausgeübt, weshalb der Debugger schnell Zeiteinsparungen zum Opfer fällt [Ros96].

Informationsgehalt

Die hilfreiche Information zum richtigen Zeitpunkt zu erhalten, dies ist der Wunsch jedes Programmierers, der einen Debugger zu Rate zieht. Ist das Werkzeug nicht in der Lage, durch geeignete Filter die jeweils relevante Information zu extrahieren, so ist das ebenso schlecht und wenig hilfreich, wie eine Präsentation

der Daten in unbrauchbarem Format (etwa die Ausgabe eines Strings als Folge von Hexadezimalzahlen).

Vertrauen

Die Information, die der Programmierer über den Zustand eines Programms während der Fehlersuche bekommt, sollte möglichst korrekt sein. Gerade Programme zur Fehlersuche müssen daher besonders robust sein, mit partieller oder fehlender Information umgehen können und ihr Wissen dem Benutzer verständlich präsentieren. Wurde der Programmierer einmal durch falsche Information in die Irre geführt, so wird er das Werkzeug in Zukunft meiden [Zel84].

Heisenberg-Prinzip

Wann immer man ein System untersucht, muß man mehr oder weniger stark in dessen interne Abläufe eingreifen. Hierdurch können Messungen im System verfälscht werden. So verhält es sich auch in der Softwaretechnik: Ein Debugger sollte das Laufzeitverhalten eines Programms nach Möglichkeit nicht verändern, was jedoch in der Praxis stets in vielfacher Hinsicht passiert. So muß beispielsweise vom Compiler ein anderer, um Debuginformation angereicherter Maschinencode erzeugt werden – Grund genug für einen genügend boshaften Bug, nicht mehr aufzutreten. Die Eigenschaft von Meßsystemen, die Meßgröße zu beeinflussen, ist in der Physik das Kriterium zur Abgrenzung quantenmechanischer Effekte von gewöhnlichen. Nach einem der Begründer der Quantenmechanik, dem Physiker Werner Heisenberg, wird das Auftreten dieser Eigenschaft *Heisenberg-Prinzip* genannt [Gra83].

1.6 Zum Schmunzeln

Im Jargon existiert der Begriff des *Heisenbugs*, einem Fehler, der verschwindet oder sich in seinen Symptomen verändert, sobald man nach ihm sucht. Ein *Bohr-Bug* stellt das Gegenteil dar: ein unter möglicherweise unbekannten, aber wohldefinierten Bedingungen reproduzierbarer Bug. Das schönste Phänomen aber ist zweifelsfrei der *Schrödinbug*; es folgt seine Definition aus [R+96], wo alle eben erwähnten Begriffe nachgeschlagen werden können:

> *schroedinbug* /shroh'din-buhg/ /n./ [MIT: from the Schroedinger's Cat thought-experiment in quantum physics] A design or implementation bug in a program that doesn't manifest until someone reading source or using the program in an unusual way notices that it never should have worked, at which point the program promptly stops working for everybody until fixed. Though [...] this sounds impossible, it happens; some programs have harbored latent schroedinbugs for years. [...]

Anzumerken ist, daß das Phänomen des Schrödinbugs offensichtlich seinerseits einen Schrödinbug darstellt: Seit ich von ihm gelesen habe, begegne ich ihm in erstaunlicher Regelmäßigkeit.

1.7 Aufbau der Arbeit

Im folgenden zweiten Kapitel werden die Grundlagen für das Verständnis dieser Arbeit gelegt. Es beginnt mit einer Kurzeinführung in die Programmiersprache

Oz, wobei eine Akzentsetzung auf diejenigen Aspekte der Sprache erfolgt, die für Design und Implementierung des Debuggers eine Rolle gespielt haben. Anschließend wird die Programmierumgebung Mozart, eine Implementierung von Oz, vorgestellt. Das dritte Kapitel beschäftigt sich mit dem Entwurf des Debuggers. Dabei stehen die drei Themenbereiche Ausführungskontrolle, Programmkontext und Nebenläufigkeit im Zentrum der Betrachtung. Im vierten Kapitel wird die Umsetzung der im vorangegangenen Kapitel diskutierten Konzepte und Ideen in einen lauffähigen Debugger beschrieben. Verschiedene Probleme, die es sowohl beim Entwurf als auch bei der Implementierung zu überwinden galt, werden im fünften Kapitel erörtert, ebenso Designentscheidungen grundsätzlicher Art und Überlegungen, wie der Debugger – auch im Hinblick auf die Fortentwicklung der Sprache Oz selbst – weiter verbessert werden kann.

2

Oz

Im diesem Kapitel stelle ich die Programmiersprache Oz vor, lege ihre wesentlichen Konzepte dar und vermittle eine erste Idee, welche Anforderungen an einen Debugger für diese Sprache gestellt werden müssen. Eine Implementierung von Oz, das Programmiersystem Mozart, steht anschließend im Mittelpunkt der Betrachtung.

2.1 Die Sprache Oz

Oz ist eine nebenläufige, constraintbasierte Programmiersprache, die funktionale und objektorientierte Programmierparadigmen unterstützt.[1] Mit ihrer Entwicklung wurde im Jahre 1991 am Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI) unter Leitung von Professor Dr. Smolka begonnen; mittlerweile beteiligen sich die Universität des Saarlandes, die katholische Universität Louvain in Belgien und das Swedish Institute of Computer Science (SICS) an der Entwicklung [S+98]. Oz besteht aus einer schlanken *Kernsprache*, ergänzt um „syntaktischen Zucker" zur Bereitstellung gängiger Programmierabstraktionen.

Nebenläufigkeit

Die Ausführung eines Oz-Programms braucht nicht streng sequentiell zu erfolgen. Vielmehr können verschiedene *Threads* nebenläufig ausgeführt werden. Bis auf hier nicht weiter diskutierte Ausnahmen erfolgt die Abspaltung eines Threads grundsätzlich an vom Programmierer *explizit* angegebenen Stellen im Programmtext. Hierdurch ist eine Kontrolle über die Anzahl der laufenden Threads und deren auszuführenden Code möglich. Eine frühere Version von Oz benutzte *implizite* Nebenläufigkeit: Sobald eine Anweisung aufgrund fehlender Information nicht sofort ausgeführt werden konnte, wurde automatisch ein neuer Thread abgespalten, der mit der Ausführung eben dieser Anweisung zum nächstmöglichen Zeitpunkt betraut wurde. Die Erfahrung lehrte jedoch, daß implizite Nebenläufigkeit verschiedene fundamentale Nachteile mit sich bringt, vor allem die fehlende Kontrolle über Threaderzeugung und die Unmöglichkeit, den Programmfluß statisch zu bestimmen. Durch die Umstellung des Nebenläufigkeitsmodells wurde die Implementierung des Debuggers in seiner jetzigen Form erst möglich und sinnvoll.

Logische Variablen

Variablen von Oz ähneln denen von Prolog: Eine neu deklarierte Variable ist zu-

[1] Auf Constraintprogrammierung werde ich in dieser Arbeit nicht weiter eingehen. Die Implementierung des Debuggers sieht vor, daß Berechnungen mit Constraints nicht behindert werden, andererseits aber auch nicht beobachtbar sind.

nächst undeterminiert, ihr Wert also noch nicht bekannt. Im Gegensatz zu den meisten imperativen Sprachen, in denen eine neu deklarierte Variable sofort einer neuen Lokation zugeordnet wird, die im Falle einer fehlenden Initialisierung einen eventuell zufälligen Wert hat, kann in Oz mit undeterminierten Variablen genauso gearbeitet werden wie mit determinierten. Werden keine Prozeduren betrachtet, so können zunächst zwei Sorten von Variablen unterschieden werden. *Toplevel-Variablen* werden durch das Schlüsselwort **`declare`** eingeführt:

```
declare X Y
```

Sie bleiben solange sichtbar, wie sie nicht durch Deklaration neuer Toplevel-Variablen gleichen Namens überdeckt werden. *Lokale Variablen* werden folgendermaßen erklärt:

```
local X Y in ... end
```

Ihr Skopus beschränkt sich auf den Bereich zwischen **`local`** und **`end`**.

Constraint-speicher

Über den Wert einer Variablen gibt der *Constraintspeicher* Auskunft. Benötigt eine Operation mehr Informationen als momentan im Constraintspeicher vorhanden (etwa bei der Addition zweier Variablenwerte, von denen der eine noch nicht bekannt ist), so *blockiert* der ausführende Thread solange, bis genügend Information hinzugekommen ist. Einmal determiniert, kann der Wert einer Variable nicht mehr verändert werden. Sie repräsentiert somit weniger eine Speicherzelle in einem Maschinenmodell, als vielmehr eine Variable im mathematischen Sinne. Üblicherweise werden Variablen mit den beschriebenen Eigenschaften *logische Variablen* genannt [Smo95].

Typisierung

Oz ist *dynamisch getypt*: Zur Übersetzungszeit sind die Typen der meisten Variablen noch nicht bekannt. Erst zur Laufzeit erfolgen Tests, die garantieren, daß beispielsweise Prozeduren mit Argumenten korrekten Typs aufgerufen werden. In den folgenden Abschnitten stelle ich die wichtigsten Datentypen von Oz vor, sofern sie relevant für das Verständnis dieser Arbeit sind. Eine vollständige Beschreibung findet sich in [H$^+$97].

2.1.1 Das Oz-Universum

Es können einfache und zusammengesetzte Typen unterschieden werden. Unter anderem gehören zur ersten Klasse *Zahlen*, *Literale*, *Prozeduren*, *Zellen*, *Threads* und *Locks*, zur zweiten *Records* mit den Teiltypen *Listen* und *virtuelle Strings*.

Zahlen

Zwei disjunkte Mengen von Zahlen sind in Oz definiert, *Integers* und *Floats*. Während letztere in ihrer Größe und Genauigkeit begrenzt sind, können Integers beliebig groß sein. Typkonvertierungen zwischen beiden Zahlentypen erfolgen nie implizit, die Evaluierung des Ausdrucks `1+1.0` führt daher zu einem Laufzeitfehler. Die Notation entspricht den üblichen Standards; Vorsicht ist beim unären Minus geboten, welches durch die Tilde dargestellt wird.[2]

```
declare
N = 0    I = ~42    K = 20000000001  %% Integers
P = 0.0  Q = ~42.0  R = 2.0e10       %% und Floats
```

Literale

Die Menge der *Literale* besteht aus *Atomen* und *Namen*. Atome werden denotiert durch Zeichenketten, die in einfache Anführungszeichen gesetzt sind: `'Hugo'`, `'42'`, `'\''`. Beginnt die Zeichenkette mit einem Kleinbuchstaben, so können die Anführungsstriche weggelassen werden. Drei besondere Atome sind `'|'` und `nil` zur Listen- sowie `'#'` zur Tupelkonstruktion (s. u.). Namen besitzen keine Notation. Sie werden durch die Funktion `NewName` erzeugt und existieren hauptsächlich zu Sicherheitszwecken: An eine Variable gebunden, erstreckt sich der Sichtbarkeitsbereich eines Namens genau über denjenigen der Variable – eine Eigenschaft, die z. B. im Objektsystem zur Realisierung von privaten Methoden genutzt wird. Zwei spezielle Namen sind an die Variablen `'true'` und `'false'` gebunden und stellen die booleschen Wahrheitswerte dar. Die Schlüsselworte **`true`** und **`false`** werden auf die entsprechenden Variablen abgebildet.

Records und Tupel

Das *Record* ist ein zusammengesetzter Datentyp. Es besitzt ein Literal als Markierung (*Label*) und beschreibt eine endliche Abbildung von Feldern (*Features*) auf Oz-Werte. Dabei kann ein Feature entweder ein Integer oder ein Literal sein. Der Punkt-Operator dient zur Selektion einzelner Felder. Ein Teiltyp des Records ist das *Tupel*. Seine Feldnamen sind ausschließlich die Integers 1 bis n, $n \geq 0$, deren Angabe nicht erforderlich ist. Ein besonderes Label ist das Zeichen `'#'`, welches als Mixfix-Operator benutzt werden kann.

```
declare
R = a(b:17 c:4)     %% Ein Record
{Show R.b+R.c}      %% Zugriffe auf seine Felder

T       = a(b c)      %% Ein Tupel
U       = a(1:b 2:c)  %% mit zwei äquivalenten
L = a  V = L(2:c b)   %% Kollegen

W = a # b # c       %% entspricht '#'(a b c)
```

Listen

Mit Hilfe der beiden Atome `'|'` und `nil` werden in Oz *Listen* konstruiert, die induktiv definiert sind: Das Atom `nil` ist eine Liste, ebenso wie das Tupel `'|'(X Xs)`, unter der Voraussetzung, daß `Xs` seinerseits eine Liste ist. Um die Notation zu vereinfachen, gelten folgende Äquivalenzen:

```
'|'(X nil) == X|nil == [X]
```

Es können geschlossene und offene Listen unterschieden werden. Erstere enden mit `nil` und haben damit eine wohldefinierte Länge, letztere mit einer undeterminierten Variable:

```
declare L S X Y in
L = 1|2|3|nil %% Eine geschlossene Liste
S = a|b|X     %% und eine offene, die
X = c|Y       %% beliebig verlängert werden kann
```

Mit einer offenen Liste kann ein *Strom* (engl. „stream“) realisiert werden, der – wie im Beispiel oben demonstriert – eine (konzeptuell) unendliche Folge von Werten kodiert.

Strings

Für Zeichenketten (*Strings*) existiert in Oz kein eigener Datentyp. Listen mit Ele-

[2] Dies ist eine Konvention aus der Sprache SML.

menten aus einem Teilbereich der Integers dienen als Ersatz, indem ihre Elemente als ISO-Latin-1-Codes interpretiert werden: Der String `"Hello"` wird demnach durch die Liste `[72 101 108 108 111]` kodiert. *Virtuelle Strings* sind Zahlen, Atome und Strings, die unter Tupelkonstruktion mit `'#'` als Label abgeschlossen sind: `("It's"#2)#'late!'`.

Prozeduren

Oz ist eine vollständig kompositionale Sprache. Prozeduren sind *first class*, können also zur Laufzeit erzeugt und an Variablen gebunden werden:

```
local
   proc {F G X P}
      P = proc {$ Y} {G X+Y} end
   end
in
   {{F Show 41} 1}
end
```

Funktionen stellen in Oz keinen eigenen Typ dar. Vielmehr werden sie auf Prozeduren mit zusätzlichem Ausgabeargument zurückgeführt.[3]

```
fun {F X}            proc {F X Out}
   X+1       ==>       Out = X+1
end                  end
```

Zellen

Werden für eine Berechnung zustandsbehaftete Größen benötigt, so können *Zellen* benutzt werden. Sie bieten Speicherplatz für genau einen Oz-Wert und erlauben destruktive Programmierung, wie sie aus imperativen Sprachen bekannt ist.

```
local
   C = {NewCell 'startwert'}
in
   {Show {Access C}}           % Ausgabe von 'startwert'
   {Assign C 42}
   {Show {Exchange C $ 'neu'}} % Access und Assign in einem
                               % atomaren (!) Schritt;
                               % Ausgabe von 42
   {Show {Access C}}           % Ausgabe von 'neu'
end
```

Threads

Die verschiedenen Berechnungseinheiten eines Programms sind in Oz ebenfalls First-Class-Datenstrukturen. Verschiedene Bibliotheksfunktionen operieren auf *Threads*. So können sie unter anderem angehalten oder terminiert werden, genauso wie eine Thread-Umschaltung (engl. „preemption“) erzwungen werden kann.

Locks

Mit Hilfe von Locks werden Quelltextbereiche definiert, die exklusiv von einem Thread zu gegebener Zeit ausgeführt werden können.

```
local
   L = {NewLock}
in
```

[3] Man nennt dies *relationale Syntax*. Sie zeugt noch von der Prolog-orientierten Vergangenheit von Oz. Zur Zeit sind Überlegungen im Gange, wie die Basissprache funktional definiert werden kann [Smo98].

```
   lock L then
      {Show 'Hier ist das Lock gesetzt'}
   end
end
```

Versucht ein weiterer Thread, den durch **lock...end** geschützten Bereich zu betreten, so blockiert er genau solange, bis das Lock wieder freigegeben ist. Warten mehrere Threads, so erfolgt eine *faire* Auswahl desjenigen Threads, der als nächster das Lock zu betreten berechtigt ist.[4]

2.1.2 Kommentare

Kommentare bis zum Zeilenende werden durch das Zeichen % eingeleitet, solche über mehrere Zeilen in `/*` und `*/` eingeschlossen.

2.1.3 Anweisungen und Ausdrücke

Ein Programm in Oz besteht aus einer Folge von *Anweisungen*. Jede Anweisung kann wiederum *Ausdrücke* als Bestandteile enthalten. Im folgenden stelle ich diejenigen Anweisungen vor, die zum Verständnis der Beispielprogramme relevant sind.

skip Die leere Anweisung **skip** tut schlicht überhaupt nichts. Sie wird z. B. in Verbindung mit Konditionalen (s. u.) eingesetzt.

Definition Prozedur-Definitionen wie

```
declare
proc {P X Y Z}
   Z = X+Y
end
```

sind ebenso Anweisungen wie eine Reihe anderer **xxx...end**-Konstrukte:

```
local ... end  %% Variablendeklaration
lock ... end   %% durch ein Lock kontrollierter Bereich
thread ... end %% Threaderzeugung
try ... end    %% Installation eines Ausnahmenfängers
```

Konditional Konditionale gibt es in mehreren Varianten. Einige davon testen auf boolesche Werte (der Testausdruck heißt *Arbiter*), andere erlauben *Musterabgleich* (engl. „pattern matching“); wahlweise können dabei die angegebenen Muster parallel oder sequentiell überprüft werden.

```
%% ein Konditional ohne Musterabgleich
case X == 1 then {P1}
elsecase X > 1 then {P2}
else {P3}
end
```

[4] „Fair“ bedeutet in diesem Zusammenhang, daß gemäß dem Prinzip „first come, first serve“ vorgegangen wird.

```
%% links ein paralleles Konditional mit Musterabgleich,
%% rechts sein sequentielles Pendant
case X                      case X
of Pattern1 then {P1}       of Pattern1 then {P1}
[] Pattern2 then {P2}       elseof Pattern2 then {P2}
[] Pattern3 then {P3}       elseof Pattern3 then {P3}
else {P4}                   else {P4}
end                         end
```

Alle in einem Muster vorkommenden Variablen sind implizit deklariert (mit auf die jeweilige Klausel beschränktem Skopus) und werden, sofern das Muster paßt, an die korrespondierenden Teilwerte gebunden.

2.1.4 Objekte

Oz ermöglicht objektorientierte Programmierung mit Mehrfachvererbung. Klassen gliedern sich in einen Datenbereich, zu dem einerseits *Attribute* zählen, die auf Zellen abbilden und somit veränderliche Daten speichern können, und andererseits *Features*, die Werte aus dem Oz-Universum bezeichnen und somit Konstanten darstellen, und Operationen, *Methoden* genannt, die auf diesen Daten rechnen. Methoden können auf zwei verschiedene Arten aufgerufen werden: Als Objekt- oder als Methodenapplikation. Die erste Variante verwendet dynamische, die zweite, die nur innerhalb von Methoden verwendbar ist, statische Bindung.

Als einfaches Beispiel sei die folgende Definition eines Zähler-Objekts gegeben, das bei jedem Aufruf der Methode `inc` eine jeweils um eins erhöhte Integerzahl zurückliefert:

```
declare
class Counter
  attr i : 0
  meth inc($)
    i <- @i+1
  end
end

C = {New Counter inc(_)}
{Show {C inc($)}}
```

Eine vollständige Beschreibung des Objektsystems findet sich in [Hen97].

2.1.5 Ausnahmebehandlung

Einen flexiblen Mechanismus zum Umgang mit Fehlersituationen stellt *Ausnahmebehandlung* (engl. „exception handling“) dar. Ausnahmen können *geworfen* und von *Fängern* (engl. „handlers“), die sie verarbeiten, *gefangen* werden. Ist beispielsweise während der Applikation einer Prozedur P mit dem Wurf der Ausnahme `fehler(X)` zu rechnen, so wird die Prozedur nicht direkt aufgerufen, sondern zunächst ein Fänger installiert, der die Ausnahme zu verarbeiten imstande ist:

```
try {P 42}
catch fehler(Z) then {Handler Z}
end
```

Dabei könnte P wie folgt definiert sein:

```
declare
proc {P X}
  case X == 42 then skip else
     raise fehler(X) end
  end
end
```

Beim Wurf einer Ausnahme wird die Ausführung der Applikation abgebrochen und der Programmlauf im passenden Fänger fortgesetzt. Existiert dieser nicht, so spricht man von einer *nicht gefangenen Ausnahme* (engl. „unhandled exception"), die die sofortige Terminierung des ausführenden Threads zur Folge hat und als Laufzeitfehler angezeigt wird.

2.1.6 Nebenläufigkeit

Oz gestattet nebenläufige Programmierung, also die Untergliederung einer Berechnung in verschiedene Threads, die nebeneinander arbeiten und über logische Variablen miteinander kommunizieren können. Hierzu existiert das Konstrukt **thread...end**, welches sowohl in Anweisungs- als auch in Ausdrucksposition auftreten darf:

```
local
   X = thread 17+4 end   %% Ausdrucksposition
in
   thread {Show X+1} end %% Anweisungsposition
end
```

Zusammen mit der Bibliotheksfunktion `Wait`, die bei Applikation den ausführenden Thread genau solange blockiert, wie die übergebene Variable nicht determiniert ist, können bereits einfache Synchronisationen zwischen Threads programmiert werden:

```
local Sync in
  thread {Show 'erster!'} Sync = unit end
  thread {Wait Sync} {Show 'zweiter!'} end
end
```

Unter Verwendung von Zellen kann eine k-Semaphore implementiert werden. Später (in Kapitel 3.7) wird eine modifizierte Version dieses Codes dazu dienen, Konkurrenzsituationen (engl. „race conditions") zu provozieren und zu erklären:

```
declare
fun {NewSemaphore K}
   X
   fun {Allocate N}
      case N > 0 then unit|{Allocate N - 1}
```

```
      else X
      end
   end
   Hd = {NewCell {Allocate K}}
   Tl = {NewCell X}
in
   proc {$ P} Old New in
      {Exchange Hd Old New}
      case Old of _|Rest then New = Rest end
      try
         {P}
      finally X in
         {Exchange Tl unit|X X}
      end
   end
end

declare S = {NewSemaphore 3} % definiert eine 3-Semaphore,
{S proc {$} {Show a} end}    % die hier zum Einsatz kommt
```

2.2 Die Implementierung Mozart

In diesem Abschnitt stelle ich die Entwicklungsumgebung des Programmiersystems *Mozart* vor. Der Beschreibung ihrer Architektur folgt eine Betrachtung verschiedener Aspekte der virtuellen Maschine, dem „Herzen“ des Systems.

2.2.1 Architektur

Die Entwicklungsumgebung von Mozart besteht aus mehreren Komponenten, wie in Abbildung 2.1 dargestellt. Es kann untergliedert werden in ein Basis-System, dessen Komponenten das ***O**z **P**rogramming **I**nterface* (*OPI*) bilden, und darauf aufbauende Werkzeuge, die sich unterschiedlichsten Aufgaben widmen.

Basis-System

Zur Quelltext-Eingabe dient *Emacs*, ein sehr leistungsfähiger Editor,[5] der unter Zuhilfenahme der Sprache *Emacs-Lisp*, einem Lisp-Dialekt, unter anderem syntaxgesteuertes farbliches Hervorheben und automatisches Einrücken beherrscht. Emacs besitzt außerdem die Aufgabe des Ladens von Quelltextbereichen in den Compiler. Man spricht von *Anfragen* (engl. „queries“), die abgesetzt werden.

Emacs startet den *Emulator*. Dabei handelt es sich um eine in C++ geschriebene *virtuelle Maschine* [Meh98], die platformunabhängigen Bytecode interpretiert, der vom komplett in Oz geschriebenen *Compiler* erzeugt wird.[6] Zur Realisierung von graphischen Anwendungen existiert eine Anbindung an die Skriptsprache

[5] Es existieren mehrere Implementierungen, die alle in ihrer Basisfunktionalität vergleichbar sind. Die bekanntesten sind *GNU-Emacs* und *XEmacs*, welche beide von Mozart unterstützt werden.

[6] Der Emulator ist die wesentliche Komponente von Mozart, die plattformabhängig ist. Es existieren Portierungen auf alle gängigen Betriebssysteme, so etwa Linux, NetBSD, FreeBSD, Solaris, AIX, Irix und Windows95/NT, um nur einige zu nennen.

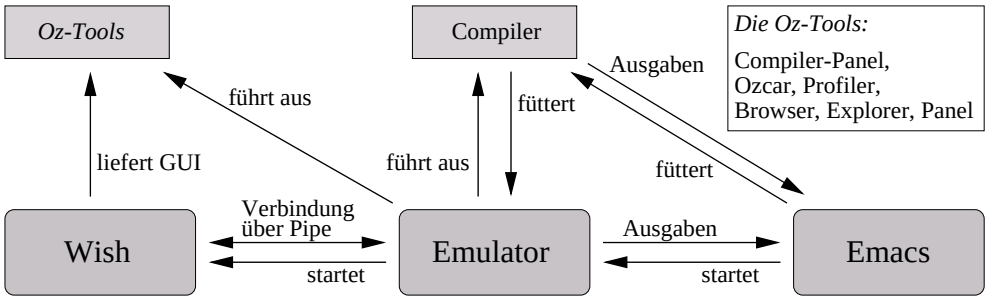


Abbildung 2.1: Die Architektur der Entwicklungsumgebung Mozart

Tcl/Tk [Ous94]. Dazu startet der Emulator einen weiteren Prozeß, den Interpreter *wish*, mit dem er über eine Unix-Pipe in Kontakt bleibt. Schließlich existieren eine Reihe von Werkzeugen, die allesamt in Oz geschrieben sind.

Werkzeuge

Mit dem *Browser* [Pop95] können Oz-Ausdrücke graphisch ansprechend dargestellt werden, wobei Teilausdrücke aus- und einblendbar sind. Fehlt Information über angezeigte Variablenwerte im Constraintspeicher, so wartet der Browser auf das Eintreffen neuer Information, um dann sofort die Anzeige anzupassen. Der *Explorer* unterstützt die Entwicklung von constraintbasierten Programmen zur Lösung von Suchproblemen [Sch97]; mit ihm können Suchbäume angezeigt und gezielt Informationen über einzelne Knoten abgerufen werden. Der *Profiler* dient zur Optimierung von Oz-Programmen. Er mißt (unter anderem) den Speicherverbrauch und die Aufrufhäufigkeit verschiedener Prozeduren.

2.2.2 Datenstrukturen des Emulators

Der Emulator stellt die „CPU“ des Mozart-Systems dar. Er kann mehrere als C++-Klassen modellierte Threads verwalten und nebenläufig rechnen lassen. Bevor ein neuer Thread erzeugt wird, muß der von diesem Thread auszuführende (Maschinen-)Code verfügbar sein. Dazu existiert ein ausgewiesener Speicherbereich im Emulator, *Codesegment* genannt, in den sämtlicher Code geladen wird. Ein neuer Thread bekommt eine initiale Aufgabe, einen *Task*, z. B. in Form einer auszuführenden Prozedur zugeteilt. Im Zuge der Ausführung können weitere Aufgaben hinzukommen, nämlich genau dann, wenn innerhalb der initial applizierten Prozedur weitere Prozeduren aufgerufen werden; ein *Taskstack* entsteht.

Taskstack

Dieser Stack und insbesondere seine interne Darstellung und Verwaltung spielt eine zentrale Rolle bei der Implementierung des Debuggers. Er besteht aus verschiedenen *Segmenten* (engl. „frames“), die ihrerseits als Dreitupel $(PC\ Y\ G)$ realisiert sind (vgl. Abbildung 2.2) und Aufgaben des Threads kodieren:

- PC Dies ist die Adresse, auf die der Programmzähler zur Wiederaufnahme der Abarbeitung dieser Aufgabe gesetzt werden muß. Alternativ kann hier ein *Spezialtask* kodiert sein, etwa zum Freigeben eines Locks oder zur Markierung eines Ausnahmenfängers.
- Y, G Diese Felder dienen zur Referenzierung der zum Stacksegment gehörenden lokalen und globalen Umgebung. Bei Spezialtasks werden sie als Universalspeicher für unterschiedliche Daten verwendet.

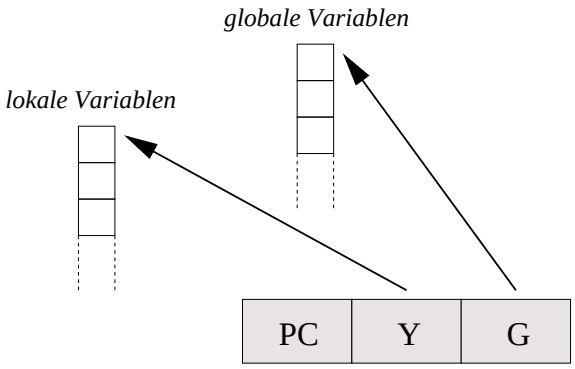


Abbildung 2.2: Die interne Repräsentation eines Stacksegments

Builtins

Als Schnittstelle zwischen C++-Code des Emulators und Oz-Code der Bibliotheken und Anwenderprogramme dienen primitive Prozeduren, sogenannte *Builtins*, die mit den von Betriebssystemen her bekannten Systemaufrufen verglichen werden können. Einmal auf Oz-Seite bekannt gemacht, kann ein Builtin wie eine gewöhnliche Prozedur verwendet werden, mit dem einzigen Unterschied, daß ihre Implementierung nicht sichtbar ist, weil sie als C++-Code im Emulator „versteckt“ ist.

3

Spezifikation

In diesem Kapitel beschreibe ich zunächst die Ziele, die beim Entwurf des Debuggers im Vordergrund standen. Anschließend erläutere ich seine Architektur und Funktionsweise mit Schwerpunktlegung auf die Schlüsselthemen *Ausführungskontrolle*, *Programmkontext* und *Nebenläufigkeit*.

3.1 Entwurfsziele

Beim Entwurf des Debuggers standen verschiedene Kriterien im Vordergrund.

Vollständigkeit Der Debugger soll eine möglichst große Teilsprache von Oz unterstützen. Dabei war es für eine Sprache wie Oz, die verschiedene Programmierparadigmen in sich vereint, nicht einfach, eine zum Debuggen geeignete Abstraktionsebene zu finden.

Integration Er soll sich harmonisch in die Entwicklungsumgebung integrieren. Programme sollen wie gewohnt aus Emacs heraus geladen und Haltepunkte von dort gesetzt werden können. Die aktuelle Position im Programm bei Einzelschritt-Abarbeitung soll angezeigt werden.

Effizienz Die Implementierung der Debugger-Kernroutinen soll möglichst effizient sein. Hierbei gilt es, einen günstigen Kompromiß zwischen Debugging-Komfort einerseits und geringen Laufzeiteinbußen andererseits zu finden.

3.2 Architektur

Der Oz-Debugger ist konzeptuell in mehrere Schichten und Komponenten gegliedert, wie in Abbildung 3.1 dargestellt. Das Programm wird zunächst vom Compiler in Maschinencode übersetzt. Dabei wird dieser mit *Debuginformation* versehen. Der Maschinencode gelangt im Emulator zur Ausführung. Gesteuert wird der Programmlauf durch Ozcar, eine graphische Oberfläche, die unter Verwendung von Tcl/Tk [Ous94] in Oz programmiert ist [MS97]. Weiterhin liefert Ozcar Informationen über den aktuellen Zustand des untersuchten Programms und erlaubt, in beliebigen Kontexten Anweisungen auszuführen oder Ausdrücke zu evaluieren. Unterstützt wird Ozcar vom Emacs-Editor, indem dieser die Anzeige der momentanen Quellcodeposition und die Haltepunktverwaltung übernimmt.

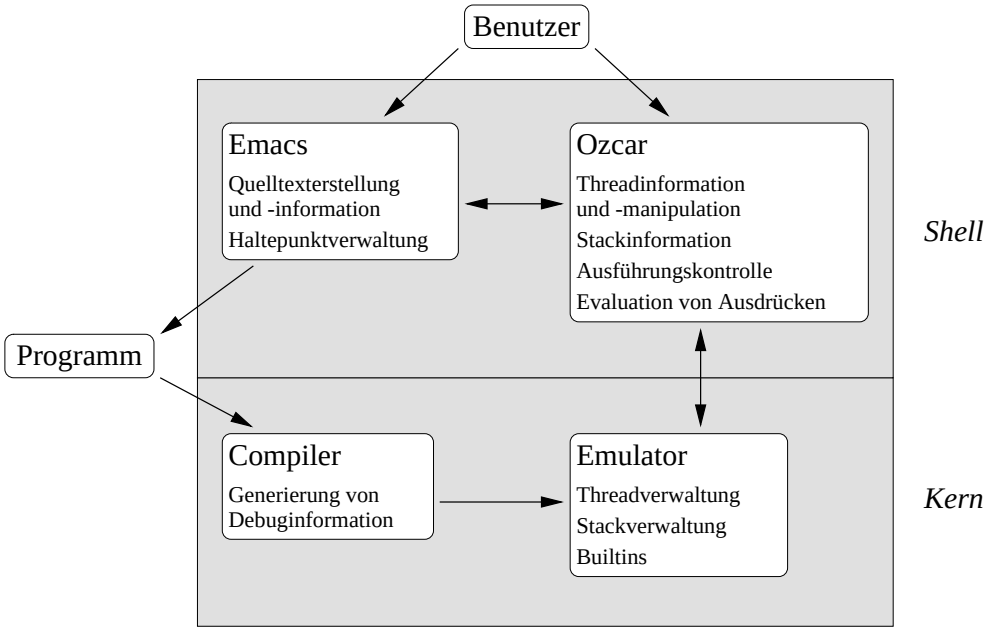


Abbildung 3.1: Die Architektur des Oz-Debuggers

3.3 Ausführungskontrolle

In diesem Abschnitt geht es um die Frage, wie der Programmablauf durch den Debugger beeinflußt werden kann. Wie in Kapitel 2.1.6 erläutert, besteht ein Programm zur Laufzeit aus einem oder mehreren Threads, die vom Oz-Emulator nebenläufig abgearbeitet werden.

Damit ergeben sich im Hinblick auf einen Debugger mehrere Fragen:

1. Muß der Benutzer Zugriff auf *alle* Programme im System haben, oder reicht es, Kontrolle über solche Programme zu besitzen, die nach dem Start des Debuggers geladen wurden?
2. Sollen für den Programmierer einzelne Threads eines Programms als solche erkenn- und selektierbar sein? Wenn ja, sollen dann stets *alle* Threads eines Programms angezeigt werden, oder macht eventuell eine Unterscheidung in „wichtige“ und „unwichtige“ Threads Sinn?
3. Welche primitiven Operationen sind nötig, um die Ausführung eines einzelnen Threads in der nötigen Weise steuern zu können?
4. Welche Operationen sind nötig, um das Zusammenspiel mehrerer nebenläufig agierender Threads kontrollieren zu können?

Programm-Selektion

Zunächst zum ersten Punkt: Grundsätzlich können *alle* Programme, die zu einem bestimmten Zeitpunkt vom Emulator ausgeführt werden, vom Debugger untersucht werden, sofern sie vom Compiler mit Debuginformation versehen wurden.

Thread-Selektion

Zu Punkt 2: Da die Zuordnung von Threads zu einem bestimmten Programm so ohne weiteres nicht möglich ist, wurde beim Design des Oz-Debuggers ein anderer Weg beschritten: Es können nur einzelne Threads untersucht werden, die (logische) Zuordnung zu einem Programm wird nicht angezeigt. Um bei mehreren Threads trotzdem nicht den Überblick zu verlieren, wurden sie um Felder

zur Speicherung einer Thread-Nummer (engl. *thread id*) und einer Vater-Thread-Nummer (engl. *parent thread id*) erweitert. Es ergibt sich somit eine Baumstruktur, die als solche dem Benutzer graphisch präsentiert wird. Das Anmelden eines Threads beim Debugger wird im Englischen üblicherweise *attachment* genannt, das Gegenstück *detachment* [SP95]. Daher heißen im Oz-Debugger die entsprechenden Operationen *attach* und *detach*. Wie und wann die Anmeldung eines Threads genau geschieht, wird in Abschnitt 3.5 näher beleuchtet.

Ausführungs-Primitiven

Zu Punkt 3: Verschiedene andere Debugger sowohl für imperative als auch funktionale Sprachen lieferten genügend Ideen für mögliche Interaktionen mit Threads [SP95, LaL94, Lim96]. So werden in der Regel mindestens die Operationen *step into* und *step over* bereitgestellt: Die erste betritt bei einer Prozedurapplikation den Rumpf der Prozedur, während die zweite die gesamte Applikation zu einer atomaren Operation zusammenfaßt.

Zur Vereinfachung sowohl auf konzeptueller als auch implementierungstechnischer Ebene wurde nach einer Möglichkeit gesucht, die beschriebene Funktionalität mit so wenigen Primitiven wie möglich zu realisieren. Das Ergebnis hiervon ist die Operation *unleash*, die die Ausführung eines Threads bis zu einer anzugebenden Tiefe im Aufrufstack fortsetzt. Es zeigt sich, daß alle weiteren beschriebenen Funktionen, bis auf *step into*, auf *unleash* zurückgeführt werden können.

Nebenläufigkeit

Zu Punkt 4: Nebenläufig interagierende Threads synchronisieren sich in der Regel über logische Variablen. Daher muß deren Manipulation im Debugger möglich sein, sprich, sie müssen interaktiv gebunden werden können. Weiterhin muß der Programmierer genaue Kontrolle darüber haben, welche Threads gerade laufen, welche blockiert und welche vom Debugger angehalten sind. Im Abschnitt 3.7 werde ich auf die Aspekte der Nebenläufigkeit näher eingehen.

3.3.1 Threads als zentrale Datenstruktur

In Oz bilden Threads eine First-Class-Datenstruktur. Viele Operationen des Emulators arbeiten auf ihnen (etwa `Thread.terminate` zum Terminieren eines Threads). Jedoch bedarf es im Debugger einer einfachen Identifizierung jedes Threads, daher wurde das von Unix für Prozesse bekannte Konzept einer Identifikationsnummer übernommen.

Thread-Hierarchie

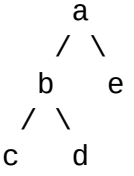
Bei Erzeugung bekommt jeder Thread eine Nummer zugeteilt, die über eine entsprechende primitive Prozedur jederzeit erfragt werden kann. Weiterhin wird die Nummer des erzeugenden Threads gespeichert, so daß eine hierarchische Struktur entsteht. Ein Beispiel möge dies verdeutlichen:

```
local
   proc {ID Name}
      local T = {Thread.this} in
         {Show Name#{Thread.id T}#{Thread.parentId T}}
      end
   end
in
   {ID a}
   thread {ID b} thread {ID c} end
                 thread {ID d} end end
```

```
   thread {ID e} end
end
```

Dieses Programm könnte z. B. folgende Zeilen drucken, aus denen sich der danebenstehende Baum gewinnen läßt:

```
a#708#1
b#709#708
e#710#708
c#711#709
d#712#709
```



Thread-Zustände

Zu jedem Zeitpunkt befindet sich ein Thread in einem der nachfolgend beschriebenen Zustände:

- **bereit** Der Thread ist lauffähig, muß aber auf eine Aktivierung durch den Emulator warten (schließlich kann dieser zu jeder Zeit nur genau einen Thread behandeln, sofern die Implementierung nicht parallele Ausführung zuläßt).
- **laufend** Der Thread ist lauffähig und bekommt momentan vom Emulator Rechenzeit zugeteilt.
- **blockiert** Der Thread ist nicht lauffähig, da er auf die Erfüllung einer Synchronisationsbedingung wartet.
- **terminiert** Der Thread hat seinen Stack abgearbeitet oder es wurde eine nicht gefangene Ausnahme geworfen.

Stoppen eines Threads

Der Debugger muß Kontrolle darüber besitzen, wann ein Thread vom Emulator bearbeitet wird und wann nicht. Eine Möglichkeit, dies zu realisieren, wäre, den Thread zu blockieren, ihn also geeignet auf einer besonderen, unter der Obhut des Debuggers stehenden logischen Variable blockieren zu lassen. Die Lösung ist jedoch nicht schön, da eine wichtige Information für den Benutzer verloren geht: Es ist nicht mehr erkennbar, ob ein Thread nur deshalb blockiert, weil er vom Debugger angehalten wurde, oder weil andere Gründe vorliegen. Daher existiert eine weitere Zustandsinformation, die orthogonal zu den letztgenannten ist: Ein Thread kann *gestoppt* sein. Ist dies der Fall, so bekommt er auch dann keine Rechenzeit zugeteilt, wenn er nicht blockiert ist. Abbildung 3.2 zeigt das Zustandsübergangsdiagramm.

Trace-Modus

In vielen Situationen muß sich ein Thread unterschiedlich verhalten, je nachdem, ob er gerade unter Debuggerkontrolle steht oder nicht. Daher besitzt jeder Thread einen *Trace-Modus*, der entsprechend aktiviert oder deaktiviert sein kann. Von ihm hängt es ab, ob Nachrichten über Blockieren, Terminieren oder Fangen von Ausnahmen generiert werden (s. Abschnitt 3.3.3), weiterhin vererbt ein Thread den Trace-Modus an von ihm erzeugte Subthreads (vgl. Abschnitt 3.7).

Step-Modus

Um Einzelschritt-Ausführung zu ermöglichen, besitzen Threads schließlich noch einen *Step-Modus*, der den Thread anweist, mit Erreichen des nächsten Steppunktes (dessen Definition im nächsten Abschnitt erfolgt) automatisch anzuhalten. Abschnitt 3.3.5 vertieft diese Thematik.

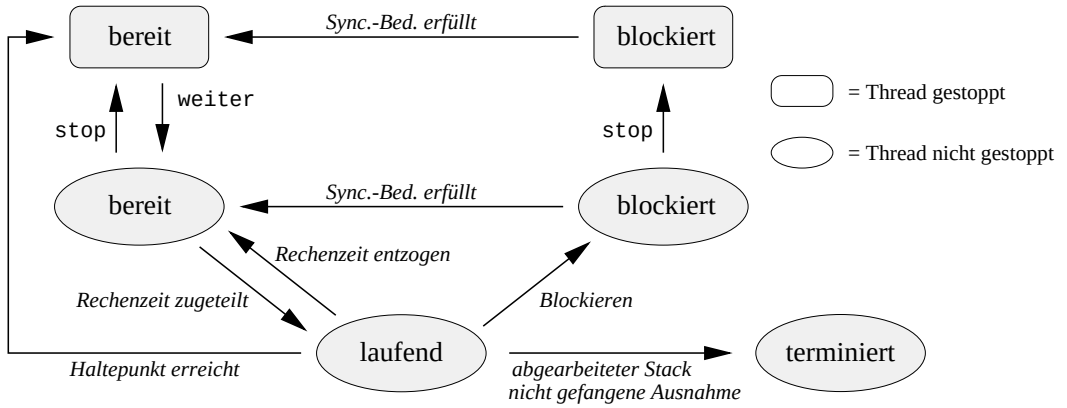


Abbildung 3.2: Die Thread-Zustände und ihre Übergänge

3.3.2 Steppunkte

Durch Stoppen eines Threads ist es möglich, dessen Ausführung zu einem beliebigen, nicht weiter spezifizierten Zeitpunkt zu unterbrechen. Es bedarf nun einer genaueren Festlegung derjenigen Stellen im Programm, an denen das Programm bei Unterbrechung tatsächlich anhält. Dies führt zu folgender Definition: Ein *Steppunkt* ist ein ausgezeichneter Programmpunkt, an dem der Debugger einen Thread anhalten und zu einem späteren Zeitpunkt weiterlaufen lassen kann. Nach Stoppen des Threads wird dieser noch genau bis zum nächsten Steppunkt weiterlaufen und dann anhalten.

In den folgenden Abschnitten wird ein Mechanismus vorgestellt, der die *schrittweise Abarbeitung* eines Threads – von Steppunkt zu Steppunkt – ermöglicht. Dabei bestimmt deren Dichte die Feinkörnigkeit, mit der ein Programm im Ablauf beobachtet werden kann. Ist sie zu gering, so treten bei der schrittweisen Abarbeitung irritierende Sprünge im Quelltext auf; dem Programmierer fällt es dann schwer, die Abarbeitung zu verfolgen. Ist sie zu hoch, so wird Einzelschritt-Ausführung schnell zu einer nervenzehrenden Angelegenheit, da man nicht „voran“ kommt. Daher ist die sorgfältige Auswahl der Steppunkte von entscheidender Bedeutung für eine leistungsfähige Ausführungskontrolle.

Es werden nun die für den Oz-Debugger gewählten Steppunkte in Reihenfolge ihrer Wichtigkeit vorgestellt. Hervorzuheben ist, daß alle Steppunkte dual existieren: Sowohl Ein- als auch Austritt der nachfolgend beschriebenen Konstrukte bilden einen solchen Punkt.

Applikation Oz als funktionale Sprache „lebt“ von der Prozedurapplikation, sie steht im Zentrum jeder Programmausführung. Daher wurde sie als Steppunkt gewählt. Der Programmierer kann die aktuellen Argumente überprüfen und entscheiden, ob er die Prozedur betreten möchte oder nicht. Weiterhin können die auf den Argumenten hinzugekommenen Constraints untersucht werden.

Konditional Das Konditional ist ein weiterer Kandidat für einen Steppunkt. Ein Anhalten vor Ausführung des Konditionals ermöglicht die Untersuchung des Arbiters.

Ausnahmenfänger Um über die Installation von Ausnahmenfängern informiert zu werden, bildet deren Installationsstelle (`try`) ebenfalls einen Steppunkt. Hierdurch reduziert

sich das Risiko, daß der Programmierer durch das Fangen von Ausnahmen und die damit verbundene Verkleinerung des Stacks irritiert wird.

Locking Beim Setzen von Locks kann das Lock untersucht und z. B. mit anderen Locks verglichen werden („Setze ich das richtige Lock?“).

Sollte sich herausstellen, daß noch weitere Punkte strategisch interessant sind und daher als Steppunkte markiert werden müssen, so kann dies durch eine Modifikation ausschließlich des Compilers jederzeit realisiert werden.

3.3.3 Das Ereignismodell

Im letzten Abschnitt wurde erklärt, wie ein Thread angehalten werden kann: Man schickt ihm die Nachricht *stop*, woraufhin er am nächsten Steppunkt anhält. Doch wo liegt dieser Punkt? Der Debugger-Kern muß der Shell das Ereignis „Thread T hat am Steppunkt P angehalten“ mitteilen. Dazu wurde ein Modell entwickelt, welches nun vorgestellt wird.

Charakteristischerweise arbeiten viele Debugger *ereignisgesteuert*: Nachdem sie das zu untersuchende Programm geladen und gestartet haben, warten sie auf Ereignisse, die ihnen von außen mitgeteilt werden. Darauf reagieren sie mit – entweder selbst generierten oder vom Programmierer angegebenen – Aktionen oder ignorieren das Ereignis.

Für den Oz-Debugger werden die folgenden Ereignisse in Form von Records definiert:

`entry(thr:T kind:K data:D args:A file:F line:L)`

Der Thread T hat in Zeile L in der Datei F an Steppunkt K angehalten, wobei $K \in \{call, cond, handler, lock\}$ (s. Abschnitt 3.3.2). Die Werte von D und A sind abhängig von K: Im Falle eines Prozeduraufrufs findet sich eine Liste mit den Aufrufargumenten in A, die Prozedur selbst in D. Bei allen übrigen Steppunkten ist A nicht definiert, und D referenziert bei einem Konditional den Arbiter, bei einem Lock das Lock, bei einem Ausnahmenfänger schließlich ist D undefiniert. Ist T bei der Shell noch nicht angemeldet, so dient diese Nachricht gleichzeitig als Anmeldung.

`exit(thr:T kind:K data:D args:A file:F line:L)`

Dieses Ereignis ist dem vorangegangenen sehr ähnlich, allerdings zeigt es den Austritt aus K an, die Positionsinformation ist entsprechend angepaßt: Bei einem Funktionsaustritt wird die Aufrufstelle referenziert, bei den übrigen Konstrukten die Zeile, die das entsprechende **end**-Schlüsselwort enthält.

`breakpoint(thr:T)`

Thread T hat einen Haltepunkt erreicht. Die Nachricht wird ignoriert, wenn T noch nicht unter Debuggerkontrolle steht. Ansonsten informiert die Shell den Benutzer über das Ereignis und stellt sicher, daß bei Eintreffen der nächsten `entry`-Nachricht alle gespeicherten Daten über den Thread aktualisiert werden.

`blocked(thr:T)`

Thread T ist blockiert.

`ready(thr:T)`

Thread T ist nicht mehr blockiert.

`exception(thr:T exc:E)`

Die Ausnahme E wurde vom Thread T geworfen, aber nicht gefangen.

`update(thr:T)`

Der Stack von Thread T hat sich derart verändert, daß die Shell ihre gespeicherten Daten über T aktualisieren muß (durch Rückfragen beim Debugger-Kern), da sie inkonsistent geworden sind. Dies kann z. B. nach Werfen und Fangen einer Ausnahme der Fall sein, da hierbei der Stack teilweise abgeräumt wird – unsichtbar für die Shell.

`term(thr:T)`

Der Thread T hat terminiert.

Man beachte, daß es kein Ereignis `created(thr:T ...)` gibt, um anzuzeigen, daß ein neuer Thread T erzeugt wurde. Statt dessen nimmt die Debugger-Shell genau dann einen Thread auf, wenn er seine erste `entry`-Nachricht schickt. Dies hat einen entscheidenden Vorteil: Die Nachricht über einen neuen Thread ist eng gekoppelt mit der Existenz von Debuginformation im vom Thread auszuführenden Code. Fehlt diese, so wird niemals eine `entry`-Nachricht generiert, und der Thread bleibt für den Benutzer unsichtbar.

3.3.4 Der Aufrufstack

Sowohl bei der Ausführungskontrolle als auch bei der Untersuchung des Programmkontextes spielt der Aufrufkeller eines Threads eine zentrale Rolle. Wozu dient er? Dazu ein kleines Beispiel:

```
fun {Fac N}
   case N < 2 then 1 else
      N * {Fac N-1}
   end
end

{Fac 4 _}
```

Diese wohlvertraute Funktion berechnet die Fakultät ihres Eingabeargumentes. Und zwar tut sie dies *rekursiv*, ruft sich also selbst auf, solange, bis die Evaluierung von `N < 2` den Wert „true“ liefert. Folgende Beobachtung ist nun wichtig: Die Berechnung der Fakultät von 4 wird auf die Fakultät von 3 zurückgeführt, und so weiter. Alle geschachtelten Funktionsapplikationen müssen *zuerst* abgearbeitet werden, bevor schließlich das Produkt 4*6 ausgerechnet werden kann. Das heißt, es muß irgendwo im Thread ein Speicher existieren, der sich die Multiplikationen 4*?, 3*? und 2*? „merkt“. Dieser Speicher ist genau der Stack, jede verzögerte Multiplikation steht in einem *Segment* (engl. „frame“) dieses Stacks.

Endrekursion

Um den Stack möglichst klein zu halten, gibt es eine Optimierung, die der Compiler durchführen kann, genannt *Endrekursionsoptimierung*. Auch dazu wieder ein kleines Beispiel:

```
declare
fun {Fac2 N A}
   case N < 2 then A else
      {Fac2 N-1 N*A}
   end
end

{Fac2 4 1 _}
```

Diese Version der Fakultätsfunktion benötigt zwar ein weiteres Argument, einen *Akkumulator*, der initial den Wert 1 haben muß, ist dafür aber *endrekursiv*: Die notwendigen Multiplikationen werden direkt beim Funktionsaufruf durchgeführt. Dies bedeutet, daß das Ergebnis jeder geschachtelten Applikation das gleiche ist, nämlich der Akkumulator aus der innersten Applikation. Dies hat als erfreuliche Konsequenz, daß zur Berechnung der Fakultät gar kein Stack zur Zwischenspeicherung mehr benötigt wird. Die Rekursion ist zu einer Iteration geworden, wie sie aus imperativen Sprachen bekannt ist. Weitere Funktionen, die der Stack übernimmt, werden im Kapitel 3.6 beleuchtet.

3.3.5 Schrittweise Programmausführung

Mit den nötigen Grundlagen ausgestattet, können wir uns nun dem interessantesten Teil der Ausführungskontrolle widmen: Bis jetzt hatte der Programmierer lediglich die Möglichkeit, einen Thread zu starten und (an unbestimmter Stelle) anzuhalten. Eine wesentliche Erweiterung des Aktionsraums ergibt sich durch die Unterstützung von Einzelschritt-Ausführung. Die Schrittweite soll dabei möglichst variabel sein. Es stellt sich heraus, daß eine einzige Operation *unleash* ausreicht, um fast alle wichtigen Schrittweiten abdecken zu können.

Betrachten wir noch einmal die erste Fassung der Fakultätsfunktion aus dem vorangegangenen Abschnitt. Nehmen wir an, die Programmausführung sei vor dem zweiten rekursiven Aufruf unterbrochen worden. Der Stack sieht also momentan folgendermaßen aus (in der Literatur wächst ein Stack meist nach oben, denn es wird ja etwas *auf* ihn gelegt):

```
=> 3 {Fac 2 _}
   2 {Fac 3 _}
   1 {Fac 4 _}
```

step into Die erste Möglichkeit, einen Schritt zu tun, ist, die Applikation zu betreten (*step into*). Was ist hierfür zu tun? Zunächst muß der Emulator den Thread weiterlaufen lassen; es bleibt zu klären, wie er rechtzeitig wieder anhält. Dazu existiert der in Abschnitt 3.3.1 bereits kurz vorgestellte Step-Modus. Befindet sich ein Thread in diesem, so wird er automatisch mit Erreichen des nächsten Steppunktes vom Emulator wieder gestoppt. Die Operation *step into* ist die einzige, bei der nach Wiederanhalten des Threads der Stack eventuell gewachsen ist. Die folgenden Einzelschritt-Operationen dienen alle dazu, den Stack in geregelter Weise gleich groß zu lassen oder zu verkleinern.

step over Auch diese Schrittweite ist recht geläufig: Die aktuell auszuführende Applikation soll nicht betreten, sondern als atomarer Schritt behandelt werden.

finish

Ähnlich sieht es bei der Operation *finish* aus, allerdings gilt es hier, das aktuelle Stacksegment fertig abzuarbeiten. Wie die einzelnen Operationen den Stack verändern, ist im folgenden dargestellt:

```
   step into       unleash to frame 3  unleash to frame 2
                       (step over)         (finish)

=> 4 {< 2 2 _}      <= 3 {Fac 2 2}      <= 2 {Fac 3 6}
   3 {Fac 2 _}         2 {Fac 3 _}         1 {Fac 4 _}
   2 {Fac 3 _}         1 {Fac 4 _}
   1 {Fac 4 _}
```

Es liegt nahe, die beiden Operationen *step over* und *finish* als Sonderfälle einer allgemeineren Operation *unleash to frame n* zu betrachten. Diese bekommt als Argument ein beliebiges (existierendes) Stacksegment und sorgt dafür, daß der Thread bis zum Verlassen dieses Segments weiterläuft.

Wird neben Prozedurapplikationen auch bei den übrigen Steppunkten deren Dualität dahingehend ausgenutzt, daß der Eintritt des jeweiligen Konstruktes auf dem Aufrufstack vermerkt wird, so können die gerade diskutierten Operationen entsprechend verallgemeinert werden: *step over*, abgesetzt auf einem Konditional, wird die Ausführung dann z. B. so lange fortsetzen, bis die passende Klausel fertig abgearbeitet wurde.

3.4 Haltepunkte

Selbst mit den leistungsfähigen Einzelschrittmöglichkeiten ist es mitunter recht mühsam, die Programmausführung gezielt an eine bestimmte Stelle im Quelltext zu lenken. Bewährt hat sich bei solchen Problemen das Konzept des *Haltepunktes*[1] (engl. „breakpoint“). Diese gibt es in vielen verschiedenen Varianten, wie bereits in Kapitel 1.3.1 diskutiert.

Zur Betrachtung der für den Oz-Debugger in Frage kommenden Varianten ist eine weitere Differenzierung in *logische* und *physikalische* Haltepunkte angebracht [Ros96]. Erstere beziehen sich auf Punkte im Quellcode des Programms; es ist dies die Ebene, die für den Benutzer interessant ist, auf der er Haltepunkte setzen und löschen können soll. Letztere befinden sich im ausführbaren Maschinencode und sorgen schließlich für das Anhalten eines Threads. Es muß nun eine Abbildung existieren, die die Position des Haltepunktes im Quelltext einer entsprechenden Position im Maschinencode zuordnet. Hauptaufgabe beim Design des Haltepunktmechanismus' für den Oz-Debugger war einerseits, eine solche Abbildung zu finden, andererseits, die physikalischen Haltepunkte geeignet zu kodieren.

3.4.1 Statische Haltepunkte

Eine einfache erste Lösung wurde in Form sogenannter *statischer Haltepunkte* gefunden. Sie werden einmalig vor Kompilation des Programms gesetzt und können zur Laufzeit nicht mehr gelöscht werden. Das Prinzip ist sehr einfach: Der ge-

[1] Alternativ spricht man im Deutschen auch von *Unterbrechungspunkten*.

wünschte logische Haltepunkt wird durch Einfügen einer Applikation der nullstelligen primitiven Prozedur `Debug.breakpoint` im Quelltext markiert. Ein die Applikation ausführender Thread wird dann beim Erreichen des nächsten Steppunktes angehalten und in den Trace-Modus versetzt (sofern er sich noch nicht in diesem befand). Die im letzten Absatz erwähnte Abbildung ist hierbei trivial, ergibt sich die Position des physikalischen Haltepunktes doch unmittelbar aus der Übersetzung des Quelltextes. Die Kodierung in der Oz-Maschine ist ebenfalls sehr einfach; die Arbeitsweise der Prozedur `Debug.breakpoint` ist Inhalt des Kapitels 4.5.1.

Statische Haltepunkte können auch ohne Debugger vom Programmierer zur Codierungszeit oder vom Compiler zur Übersetzungszeit angebracht werden, weiterhin „überleben" sie verschiedene Invokationen der Oz-Maschine – und damit auch des Debuggers –, weshalb sie sich gut dazu eignen, persistente Haltepunkte an neuralgischen Stellen im Programm (oder in Bibliotheken!) während der Testphase anzubringen. Um eine Emacs-Anfrage zu debuggen, kann der Compiler schließlich automatisch einen statischen Haltepunkt am Anfang des neu erzeugten Codes plazieren.

3.4.2 Dynamische Haltepunkte

Das Fehlen der Möglichkeit, Unterbrechungspunkte zur Laufzeit setzen und auch wieder löschen zu können, stellt eine starke Einschränkung dar. *Dynamische Haltepunkte* schaffen hier Abhilfe. Sie werden grundsätzlich nach Laden des Programms durch Angabe der Position im Quelltext, an welcher der Programmlauf unterbrochen werden soll, gesetzt. Allerdings ist die Abbildung auf eine passende Stelle im bereits geladenen Maschinencode nun nicht mehr trivial; sie wird in Kapitel 4.5.2 erläutert.

Eine etwas elaboriertere Variante eines dynamischen Haltepunktes, die in der vorliegenden Arbeit jedoch nicht zum Einsatz kommt, ergibt sich, wenn zugelassen wird, daß die zu einem logischen Haltepunkt korrespondierende Stelle im Maschinencode noch gar nicht geladen ist.[2]

Um sowohl das Debugmodell als auch die Implementierung einfach zu halten, liegt es nahe, als Positionen für mögliche dynamische Unterbrechungspunkte genau die Steppunkte aus Abschnitt 3.3.2 zu wählen. Somit existieren zwei Situationen, in denen ein Thread bei Erreichen eines Steppunktes anhalten muß:

1. Der Thread befindet sich im Step-Modus.
2. Der Steppunkt ist als Haltepunkt ausgezeichnet.

Dynamische Unterbrechungspunkte konsistent zu verwalten und dem Benutzer zu präsentieren stellt sich in einer interaktiven Entwicklungsumgebung als recht schwierig heraus. Wie soll z. B. verfahren werden, wenn in einem Emacs-Puffer Haltepunkte in verschiedenen Zeilen gesetzt sind und anschließend der Puffer komplett mit neuem Inhalt gefüllt wird – sollen die physikalischen Haltepunkte dann invalidiert und neue gesetzt werden? Wenn nicht, wie soll mit der Tatsache umgegangen werden, daß die Quelltextstelle zu einem physikalischen Haltepunkt gar nicht mehr existiert oder einen anderen Inhalt besitzt? Derlei Fragen werde ich mich in Kapitel 5.2 widmen.

[2] Der Unix-Debugger gdb gestattet dies.

3.5 Anmelden von Threads

Es wurden nun alle Grundlagen der Programmausführung unter Debuggerkontrolle diskutiert, so daß wir uns der Frage zuwenden können, wann und wie denn eigentlich Threads beim Debugger angemeldet werden sollen.[3] Welcher Zeitpunkt dies ist, hängt vom Fehlertyp und den Absichten des Programmierers ab. Grundsätzlich existieren zunächst einmal zwei Möglichkeiten für den Fall, daß der Debugger *vor* Auftreten des Fehlers aktiv werden soll.

3.5.1 Aufnahme vor Auftreten des Fehlers

Der Thread kann zunächst direkt bei Programmstart unter Debuggerkontrolle gestellt werden. Dazu bietet es sich an, den Compiler einen statischen Haltepunkt am Anfang des ausführbaren Codes generieren zu lassen. Der initial gestartete Thread wird daraufhin direkt am ersten Steppunkt anhalten und der Debugger-Shell durch eine `entry`-Nachricht von seiner Existenz berichten. Dem Programmierer bietet sich nun die Gelegenheit, dynamische Unterbrechungspunkte zu setzen und den Thread anschließend, eventuell im Einzelschritt-Modus, weiterlaufen zu lassen.

Soll der Thread nach seiner Erzeugung erst einmal ohne Debuggerkontrolle loslaufen, bis er eine für den Programmierer interessante Stelle im Quellcode erreicht hat, so generiert der Compiler keinen statischen Haltepunkt; statt dessen sorgt der Programmierer selbst dafür, an passender Stelle einen Unterbrechungspunkt anzubringen.

3.5.2 Aufnahme nach Auftreten des Fehlers

Soll der Thread erst *nach* Auftreten des Fehlers unter Debuggerkontrolle gelangen, so bieten sich wieder zwei Alternativen:

Das Programm „hängt“. Ursache hierfür ist entweder Divergenz oder Verklemmung. In beiden Fällen gibt es ein Problem: Da sich der ausführende Thread noch nicht im Trace-Modus befindet, wird über sein Blockieren keine Nachricht an die Debugger-Shell versandt, ebenso kann ohne Kenntnis der Stelle, wo z. B. die Endlosrekursion auftritt, keine „Bananenschale“ in Form eines dynamischen Unterbrechungspunktes ausgelegt werden. Nötig ist für diesen Fall eine Liste aller momentan im System existenten Threads (ähnlich dem *ps*-Kommando unter Unix), aus der durch Untersuchung der einzelnen Stacks der richtige herausgefunden und anschließend beim Debugger angemeldet werden kann. Dieses Vorgehen ist allerdings recht unhandlich; mit Hilfe eines Tricks ist es möglich, einem blockierenden Thread auf die Schliche zu kommen, ohne daß sich dieser im Trace-Modus befindet: Dazu wird er als „niemals blockierend“ markiert.

Tut er es doch, so wirft er eine entsprechende Ausnahme, die der Debugger-Shell mitgeteilt wird. Und genau dies ist die letzte Möglichkeit, wie ein Thread angemeldet werden kann. In diesem Fall wird die Debugger-Shell über die `exception`-Nachricht auf den Thread aufmerksam gemacht. Das Programm kann zwar nicht mehr weiter ausgeführt werden (der Stack des entsprechenden

[3] Während die Betrachtung hier auf einzelne Threads beschränkt ist, widmet sich Abschnitt 3.7 der Frage, wie mit Gruppen von verwandten Threads umgegangen werden kann.

Threads ist ja komplett geleert worden), jedoch bietet sich die Möglichkeit, aus den gesicherten Daten des Stacks Kontextinformation zusammenzustellen, die Rückschlüsse über den Grund des Absturzes zuläßt (s. Abschnitt 3.6).

3.6 Programmkontext

Wurde ein Programm angehalten, so muß über den aktuellen Stand seiner Abarbeitung möglichst viel relevante Information abgerufen werden können. Bei der Beantwortung der drei Fragen „Wo bin ich?“, „Wie kam ich hierhin?“ und „Was ist hier los?“ ist *Kontextinformation* hilfreich.

3.6.1 Stack

In Abschnitt 3.3.4 wurde bereits im Hinblick auf Funktionen zur Ausführungskontrolle erklärt, wie der Aufrufstack aufgebaut ist und wozu er dient. Hier werden nun die Möglichkeiten aufgezeigt, wie er zur näheren Bestimmung des Programmkontextes herangezogen werden kann.

Aktuelle Parameter

Zunächst müssen jedem Stacksegment die relevanten Daten des entsprechenden Steppunktes entnehmbar sein, also im Falle einer Prozedur die Prozedur selbst, zusammen mit ihren aktuellen Parametern, bei Konditionalen der Arbiter und bei Locks das Lock. Die Information darüber ist im Emulator nicht explizit abgelegt, daher ist es notwendig, sie über einen Trick verfügbar zu machen (vgl. Kapitel 4.4.1). Wird eine Quelltextanzeige in die Untersuchung mit einbezogen, so ist keine weitere Variableninformation (s. u.) notwendig, um bei Prozeduren eine Zuordnung von formalen zu aktuellen Parametern vornehmen zu können.

Kontextwechsel

Weiterhin müssen verschiedene Segmente des Stacks selektierbar sein, mit dem Effekt, daß eine Quelltextanzeige die jeweilige Aufrufstelle markiert. Damit dies gelingt, ist wiederum entsprechende Emulator-Unterstützung erforderlich (vgl. Kapitel 4.1.2). Außerdem besitzt jedes Stacksegment eine eigene Umgebung, die aus lokalen und globalen Variablen besteht, und deren Bereitstellung Aufschluß auch über die Werte derjenigen Variablen geben kann, die keine aktuellen Parameter darstellen.

3.6.2 Variablen

Innerhalb jeder Prozedur können sowohl lokale als auch globale Variablen existieren. *Lokale* Variablen sind einerseits die formalen Argumente der Prozedur, andererseits alle innerhalb des Prozedur-Rumpfs durch ein (eventuell verkürztes) **`local...in...end`**-Konstrukt eingeführten Variablen.

Für die Definition der *globalen* Umgebung einer Prozedur gibt es (mindestens) zwei Möglichkeiten: Entweder werden *alle* im aktuellen Kontext sichtbaren nicht lokalen Variablen als global betrachtet, oder nur eine Teilmenge dieser, nämlich genau diejenigen, die in der Prozedur referenziert werden. Im ersten Fall sollte es elaborierte Funktionen zum Filtern bestimmter Teilmengen von Variablen geben, um den Benutzer nicht mit Information zu „erschlagen“ (die globale Umgebung kann erfahrungsgemäß sehr groß werden), im zweiten Fall ist dies nicht unbedingt erforderlich, allerdings ist die bereitgestellte Information dann nicht

komplett – lediglich die „interessanteren“ Variablen sind abrufbar.[4] Die aktuelle Implementierung geht einen Zwischenweg, der Inhalt des Kapitels 4.6 ist.

Da in Oz logische Variablen zum Einsatz kommen, die ohne Umweg über Lokationen direkt – sofern sie bereits determiniert sind – einen Oz-Wert bezeichnen, kann die von anderen Debuggern her bekannte Funktionalität, Werte von Variablen destruktiv zu verändern, nicht bereitgestellt werden. Allerdings können noch undeterminierte Variablen an beliebige Oz-Werte gebunden werden, und es ist möglich, bei Variablen, die Zellen bezeichnen, deren Inhalt zu verändern. Die Benutzerschnittstelle für solche Aktionen wird in Kapitel 4.8.1 beschrieben.

3.7 Nebenläufigkeit

Den Schluß dieses Kapitels soll die Betrachtung der Fähigkeiten des Debuggers zur Untersuchung nebenläufiger Programme darstellen. Gerade bei diesen ist das Fehlerpotential enorm; meist ist fehlende Synchronisation oder Indeterminismus die Quelle des Übels [Dol96]. Nun ist der Mensch nur bedingt fähig, mehrere gleichzeitig ablaufende Aktivitäten komplett zu erfassen. Deren passende Visualisierung und Steuerung mit Hilfe eines Debuggers ist somit von großer Bedeutung.

In Oz ist es sehr leicht, ein Programm zu schreiben, das mehrere tausend Threads innerhalb kürzester Zeit erzeugt. Die folgende Funktion zur Berechnung der Fakultät benutzt z. B. einen „Stack aus Threads“:

```
fun {DebuggerFeind N}
   case N < 2 then 1 else
      N * thread {DebuggerFeind N-1} end
   end
end
```

Es erweist sich für die folgenden Betrachtungen als sinnvoll, zwischen *gering nebenläufigen* und *massiv nebenläufigen* Programmen zu unterscheiden. Bei ersteren erledigen relativ wenige Threads ein relativ großes Arbeitspensum, bei letzteren ist es gerade umgekehrt. Die Eignung der Funktionen des Oz-Debuggers soll besonders für die erste Kategorie zugesichert werden; massiv nebenläufige Programme stellen erhöhte Ansprüche an die Benutzerschnittstelle (man denke etwa an ein- und ausklappbare Unterbäume von Threads) und werden zur Eingrenzung der Arbeit (noch) nicht speziell unterstützt.

3.7.1 Anmelden verwandter Threads

In Abschnitt 3.5 wurde bereits der Frage nachgegangen, wie einzelne Threads eines Programms unter Debuggerkontrolle gestellt werden können. Haltepunkte erwiesen sich hier als das Mittel der Wahl. Bei nebenläufigen Programmen bietet sich jedoch noch eine weitere, implizite Methode der Threadaufnahme an, die von der in Abschnitt 3.3.1 diskutierten Threadverkettung Gebrauch macht: Sofern der Benutzer dies wünscht, werden alle Subthreads eines bereits angemeldeten

[4] „Murphy's Gesetz“ lehrt uns allerdings, daß sich die einzig wirklich wichtige Variable dann stets *nicht* in der Liste der globalen Variablen befinden wird.

Threads automatisch in den Trace- und Step-Modus gesetzt. Dies hat zur Folge, daß sie mit Erreichen des ersten Steppunktes anhalten und der Debugger-Shell über die `entry`-Nachricht von ihrer Existenz berichten. Eine Untersuchung aller hierarchisch zusammengehörigen Threads ab einem bestimmten Startknoten im Threadbaum ist somit möglich.

3.7.2 Zusammenspiel mehrerer Threads

Wie soll nun die Abarbeitung mehrerer angemeldeter Threads erfolgen? Soll immer nur ein Thread zu einem bestimmten Zeitpunkt laufen dürfen? Wenn nein, wie soll der Programmierer den Überblick behalten über den Ablauf der einzelnen Programmaktivitäten? Verschiedene Festlegungen wurden hierzu beim Entwurf des Debuggers getroffen.

Ein ausgewiesener Thread ist der *aktuelle Thread*. Von ihm werden sämtliche verfügbaren Informationen dargestellt, also Stack, Variablen und Quelltextposition. Weiterhin können auf ihm Aktionen ausgeführt werden (*step*, *unleash*, etc.), und einige von ihm gemeldete Ereignisse werden besonders behandelt (etwa dem Benutzer über die Statuszeile der Shell direkt mitgeteilt). Zum Erhalt der Übersicht über das Geschehen wird vom Debugger die Anzahl der gleichzeitig laufenden Threads klein gehalten. So werden grundsätzlich alle Threads, unmittelbar bevor sie unter Debuggerkontrolle gelangen, vom Kern erst einmal gestoppt. Der Benutzer kann jedoch mehrere Threads, etwa durch die Operation *unleash*, gleichzeitig weiterlaufen lassen. Entsprechend programmiert, kann dies auch automatisch durch die Shell erfolgen – das Protokoll zwischen Kern und Shell ist hier auf maximale Flexibilität ausgelegt.

Betrachten wir nun einige typische Fehlersituationen, wie sie bei nebenläufiger Programmierung häufig auftreten. Es wird sich zeigen, daß bereits mit relativ wenig Unterstützung an richtiger Stelle ein Debugger umfassende Hilfe anbieten kann.

Verklemmung

Wenn eine Synchronisations-Bedingung in gegebenem Kontext nicht erfüllt werden kann, so spricht man von *Verklemmung*. Ein einfacher Fall dieser Fehlersituation liegt vor, wenn ein einzelner Thread aufgrund einer vergessenen Variableninitialisierung blockiert:

```
local
   class Foo
      feat X
      attr Y
      meth doIt(Z)
         Y <- self.X + Z
      end
   end
in
   {New Foo doIt(42) _}
end
```

Diese Sorte Fehler tritt in der täglichen Programmierpraxis mit großer Regelmäßigkeit auf, ist jedoch leicht in den Griff zu bekommen. Es genügt, die `blocked(thr:T)`-Nachricht geeignet auszuwerten: Der oberste Eintrag des

Stacks von Thread T liefert genau die Position im Quelltext, an der T blockiert. Aus Effizienzgründen wird der Stack jedoch nicht zusammen mit der `blocked`-Nachricht verschickt, sondern muß explizit angefordert werden, denn in verschiedenen Situationen blockieren Threads hin und wieder kurzzeitig an für den Programmierer uninteressanten Stellen, etwa bei asynchroner Kommunikation über Unix-Pipes oder beim Builtin `Delay`. Eine frühe Designentscheidung, den Thread bei Blockieren stets automatisch anzuhalten, wurde aus diesem Grunde wieder verworfen, denn sie führte zu der Notwendigkeit, je nach Programm regelmäßig den Thread neu starten zu müssen, wie in folgendem Beispiel:

```
local
   X = {Tk.returnInt winfo(depth '.')}
in
   {Show X+1}
end
```

Die Anfrage beim Wish-Prozeß (s. Kapitel 2.2) dauert so lange, daß die primitive Prozedur `'+'` kurzzeitig blockiert, da der Wert der Variable `X` noch nicht determiniert ist.

Pattsituation

Wenden wir uns nun einem weiteren Fall von Verklemmung zu, an dem zwei Threads beteiligt sind, die „ewig" aufeinander warten. Man spricht hier von einer *Pattsituation* (engl. „deadlock").

```
local
   L1 = {NewLock}
   L2 = {NewLock}
in
   thread
      lock L1 then lock L2 then
         {Show 'thr 1 ready'}
      end end
   end
   thread
      lock L2 then lock L1 then
         {Show 'thr 2 ready'}
      end end
   end
end
```

Wenn der erste Thread das Lock `L1` und der zweite Thread das Lock `L2` besitzt, so kann keiner der beiden mit der Berechnung fortfahren. Das Beispiel sieht „künstlich" aus, da es bewußt einfach gehalten und symmetrisch aufgebaut ist. In der Praxis können die Codebereiche der an der Berechnung beteiligten Threads weit auseinander liegen, am prinzipiellen Problem ändert sich jedoch nichts; der Debugger kann es genauso wie im ersten Beispiel unmittelbar aufspüren. Weiterhin können Verklemmungen durch geschickte Verzahnung von Einzelschritt-Ausführung *provoziert* werden; insofern stellt der Debugger hier auch ein geeignetes Mittel dar, um nebenläufige Algorithmen auf die Probe zu stellen.

Konkurrenz-Situation

Eine weitere Fehlerklasse stellen *Konkurrenzsituationen* (engl. „race conditions“) dar, die dann entstehen, wenn verschiedene Threads zum gleichen Zeitpunkt auf die gleichen Daten zugreifen. Werden diese Zugriffe nicht synchronisiert, so kann es passieren, daß Daten inkonsistent werden und Sicherheitslücken oder Fehler mannigfaltiger Art entstehen, weil ein Thread von Bedingungen ausgeht, die ein anderer unbemerkt in der Zwischenzeit geändert haben kann. Problematisch ist der Indeterminismus, mit dem solche Probleme auftreten: Bei mehreren Testläufen eines Programms mit identischen Eingaben kann einmal alles gut gehen und ein anderes Mal das „große Chaos“ ausbrechen. Außerdem liegen Fehlerstelle und Fehlerursprung (s. Kapitel 1.2.2) mitunter weit entfernt voneinander.

Greifen wir als Beispiel den Code der k-Semaphore aus Kapitel 2.1.6 auf. An zwei Stellen (Zeilen 13–16 und 21–24) wurden (vermeintlich) minimale Änderungen vorgenommen:

```
1  declare
2  fun {NewSemaphore K}
3     X
4     fun {Allocate N}
5        case N > 0 then unit|{Allocate N - 1}
6        else X
7        end
8     end
9     Hd = {NewCell {Allocate K}}
10    Tl = {NewCell X}
11 in
12    proc {$ P} Old New in
13       % !!!
14       % {Exchange Hd Old New}
15       {Access Hd Old}
16       {Assign Hd New}
17       case Old of _|Rest then New = Rest end
18       try
19          {P}
20       finally X in
21          % !!!
22          % {Exchange Tl unit|X X}
23          {Access Tl unit|X}
24          {Assign Tl X}
25       end
26    end
27 end
```

Durch die Auflösung der Atomizität des Zugriffs auf die Zellen `Hd` und `Tl` entsteht genau dann das Problem einer Konkurrenzsituation, wenn zwischen lesendem und schreibendem Zugriff eine Threadumschaltung stattfindet und ein weiterer Thread die Semaphore betritt: War ein Thread vor dem Aufruf von `Assign` in Zeile 16 angehalten, so vergrößert sich die Semaphore um eins, wird also zu einer $(k+1)$-Semaphore, war er dagegen in Zeile 24 angehalten, so verkleinert sie sich um denselben Betrag.

Dieser Fehler ist wesentlich schwerer zu finden als die zuvor beschriebenen. Zunächst muß der Programmierer den Fehlerursprung finden – dazu kann eine elaborierte Umgebungsanzeige und insbesondere ein Dialog zur Berechnung von Oz-Ausdrücken hilfreich sein, denn der Zugriff auf den aktuellen Wert der Zellen ist dann – in Verbindung mit gleichzeitiger Einzelschritt-Abarbeitung *verschiedener* Threads – jederzeit möglich. Richtig komfortabel würde es durch Daten-Haltepunkte (vgl. Kapitel 1.3.1), die einen Thread automatisch etwa bei jedem Zugriff auf eine bestimmte Zelle anhalten können. Sie stehen auf der Agenda zur Weiterentwicklung des Debuggers an hoher Stelle.

4

Implementierung

In diesem Kapitel beschreibe ich die Realisierung des Debuggers, indem ich zunächst einen Überblick über die Implementierung der einzelnen Komponenten gebe, und anschließend die drei Schwerpunkte *Codegenerierung*, *Ausführungskontrolle* und *Kontextbestimmung* im Detail behandle. Die momentane Implementierung besitzt beinahe vollständig die in Kapitel 3.1 geforderte Funktionalität. Nicht nur am Lehrstuhl wurde mit dem Debugger bereits erfolgreich Fehlersuche betrieben. Gründe für das Fehlen bestimmter Komponenten, etwa einer komfortablen Haltepunkt-Verwaltung, werde ich in Kapitel 5.2 nennen.

Die Implementierung erstreckt sich über praktisch alle Teile des Oz-Systems. Zunächst ist es nötig, den Compiler dahingehend zu erweitern, daß er auf Wunsch um Debuginformation erweiterten Code generiert. Der Emulator muß diese Information auswerten und sowohl zur Ausführungskontrolle als auch zur Kontextbestimmung einsetzen. Dabei sollte sichergestellt werden, daß auch Code mit keiner oder nur partieller Debuginformation korrekt behandelt wird. Zur Steuerung und Visualisierung des Debugging-Vorgangs bedarf es einer komfortablen Benutzeroberfläche. Diese Aufgabe teilen sich in der aktuellen Implementierung das unter Zuhilfenahme des Tcl/Tk-Moduls in Oz programmierte Werkzeug *Ozcar* und *Emacs*, der Standard-Editor des Mozart-Systems. Um Implementierungsdetails auszublenden, gebe ich im folgenden nicht immer den exakten Code an, sondern lediglich sein auf die wesentlichen Punkte reduziertes Skelett. Der interessierte Leser findet den kompletten Quelltext in [S+98].

4.1 Codegenerierung

Die meisten Compiler verändern die Codegenerierung, wenn debugbarer Code entstehen soll. Bisweilen wurden hierzu recht komplexe Systeme aus Zusatzinformation entwickelt, vgl. z. B. [M+93]. Debuginformation kann in Form von speziellen Instruktionen in den Code eingefügt werden. Zu deren Plazierung existieren grundsätzlich zwei Möglichkeiten:

- hinter dem zur Ausführung kommenden Code, also an einem „toten Punkt“ im Programm;
- eingestreut in den übrigen Code.

Beide haben ihre Vor- und Nachteile. Der Vorteil der ersten Variante ist, daß die Instruktionen niemals zur Ausführung kommen, das System also nicht langsa-

mer wird; der Vorteil der zweiten Variante ist, daß vielfältigere Einsatzmöglichkeiten existieren. So kann der Debuginformation z. B. eine operationale Semantik zugeordnet werden, um Informationspropagierung zu ermöglichen. Bei der vorgestellten Implementierung werden beide Möglichkeiten kombiniert.

4.1.1 Veränderte Instruktionsauswahl

Der Compiler verwendet bei Generierung von Debugcode an einigen Stellen andere, teilweise weniger effiziente Instruktionen. Dies hat mehrere Gründe:

- **Einfachheit** Der verwendete Befehlssatz sollte möglichst regulär sein, um die Implementierung des Debugger-Kerns einfach halten zu können.
- **Funktionalität** Verschiedene Instruktionen, allen voran das Konditional in all seinen Spielarten, lassen in der optimierten Variante keine konsistente Einzelschritt-Ausführung zu und werden daher durch besser geeignete Äquivalente ersetzt.

4.1.2 Markierung der Steppunkte

In Kapitel 3.3.2 wurden die Steppunkte in einem Oz-Programm festgelegt, also die Stellen, an denen es möglich sein soll, den Programmlauf zu unterbrechen. Realisiert wird dies durch zwei neue Instruktionen, `debugEntry` und `debugExit`, die wie folgt aufgebaut sind:

```
debugEntry(<file> <line> <column> <kind>)
debugExit(<file> <line> <column> <kind>)
```

Sie werden vom Compiler unmittelbar vor und nach Steppunkten in den Code eingefügt. Hierbei ist `<file>` entweder eine Quelltextdatei oder ein Emacs-Puffer, `<line>` und `<column>` geben Zeile und Spalte an. `<kind>` definiert den Typ des Steppunktes (`'call'`, `'cond'`, `'handler'` oder `'lock'`).

Hierdurch ist es möglich, an zentraler Stelle im Emulator, nämlich genau bei Ausführung beider oben genannter Instruktionen, die gesamte Steuerzentrale des Debugger-Kerns unterzubringen. Sollen weitere Steppunkte definiert werden, so reicht es im allgemeinen, den Compiler so zu ändern, daß an entsprechender Stelle zusätzliche `debugEntry`- und `debugExit`-Befehle generiert werden. Wie damit die Programmausführung im einzelnen gesteuert wird, ist Inhalt des Abschnitts 4.4.3.

`\line`-*Direktive*

Der Compiler kann den zu übersetzenden Quelltext entweder direkt aus einer Datei oder in Form eines virtuellen Strings laden. Um auch im letzteren Fall Debuginformation erzeugen zu können, kennt der Compiler eine `line`-Direktive:

```
\line <line> <file>
```

Sie dient dem Compiler ausschließlich zur Orientierung und wird z. B. von Emacs bei der Generierung von Anfragen eingesetzt. Der Dateiname ist in diesem Fall der aktuelle Emacs-Puffer, den Zeilenoffset berechnet eine kleine Emacs-Lisp-Funktion. Problematisch wird es bei der verteilten Ausführung von Code, wenn

ein Puffername nicht mehr notwendigerweise zum lokalen Emacs gehört, sondern eventuell zu einem im fernen Australien. Momentan entsteht in solch einem Fall reichlich Konfusion. Eine Lösung wäre, den Puffer- und Dateinamen als Präfix einen Rechnernamen mitzugeben. Eine weitergehende Betrachtung dieser Problematik werde ich in Kapitel 5.2 anstellen.

Eine wichtige Einschränkung, die sich aus der Positionierung der Debuginformation ergibt, sollte an dieser Stelle erwähnt werden: Der Compiler ist bei Generierung von Debugcode nicht in der Lage, Endrekursions-Optimierungen durchzuführen, da jeder `call`-Instruktion eine `debugExit`-Instruktion folgt, die nach Rücksprung aus der Prozedur noch aufgerufen werden muß. Dies kann bei größeren Programmen zu hohem temporären Speicherbedarf, Laufzeiteinbußen oder gar einem Stacküberlauf führen. Eine Idee zur Rückgewinnung von Endrekursion wird in Kapitel 5.4 angegeben.

4.1.3 Sicherung von Variablennamen

Beim Übersetzen des Quelltextes gehen die ursprünglich gewählten Variablennamen verloren. Zum Zeitpunkt des Debuggens muß es jedoch möglich sein, sie zurückzugewinnen. Hierzu dienen zwei weitere Instruktionen:

```
localVarname(<name>)
globalVarname(<name>)
```

Sie werden am Ende jeder Prozedurdefinition generiert, und zwar je eine für jede in der Prozedur sichtbare lokale bzw. globale Variable. Der Trick besteht nun darin, die Sortierung der Instruktionen gerade den Offsets in den internen Datenstrukturen anzupassen. Somit wird es einfach, aus diesen eine Liste von Paaren `Variablenname#Wert` zu generieren.

4.1.4 Compilerschalter

Wieviel Debuginformation der zu generierende Code enthalten soll, kann mit Hilfe dreier Compilerschalter festgelegt werden:

```
\switch +debuginfocontrol  % markiere Steppunkte
\switch +debuginfovarnames % erzeuge Variableninformation
\switch +debuginfo         % beides zusammen
```

Weiterhin kann der Compiler durch Laden von

```
\switch +runwithdebugger
```

angewiesen werden, einen Haltepunkt unmittelbar an den Anfang allen bei einer Anfrage neu erzeugten Codes zu setzen (s. Abschnitt 4.5).

4.2 Die Schnittstelle zwischen Kern und Shell

Wie in Kapitel 3.3.3 ausgeführt, arbeitet die Debugger-Shell ereignisorientiert: Benutzerbefehle müssen verarbeitet, vom Debugger-Kern generierte Nachrichten interpretiert und entsprechend aufbereitet dem Benutzer präsentiert werden.

Zum Transport dieser Nachrichten wird ein *Strom* verwandt (s. Kapitel 2.1.1), dessen Ende im Emulator gespeichert ist.

Schreiben Beschrieben wird der Strom durch Aufruf der folgenden Funktion:

```
void debugStreamMessage(TaggedRef message) {
  Assert(onToplevel());
  TaggedRef newTail = OZ_newVariable();
  int ret = OZ_unify(Tail,cons(message,newTail));
  Tail = newTail;
  Assert(ret == PROCEED);
}
```

Diese wird ihrerseits bei allen in 3.3.3 definierten Ereignissen aufgerufen, beispielsweise beim Blockieren eines Threads. In diesem Fall enthält die Variable `message` den Ausdruck `blocked(thr:T)`.

Lesen In der Shell werden die Nachrichten vom Strom gelesen und verarbeitet:

```
proc {ReadEvalLoop S}
   case S
   of M|T then
      {Ozcar processMessage(M)}
      {ReadEvalLoop T}
   end
end
{ReadEvalLoop {Debug.getStream}}
```

Triggerung Die Nachrichtengenerierung im Emulator erfolgt nur, wenn dieser sich im Debugmodus befindet, was durch Aufruf eines entsprechenden Builtins beim Start der Debugger-Shell geschieht.

4.3 Erweiterung von Threads

Drei neue, ausschließlich vom Debugger verwendete Bits steuern das Verhalten einzelner Threads während einer Debug-Sitzung.

Stop-Bit Das *Stop-Bit* signalisiert dem Scheduler des Emulators, daß der entsprechende Thread angehalten ist und somit keine Rechenzeit zugeteilt bekommen darf.

Trace-Bit Das *Trace-Bit* bedeutet, daß sich der Thread unter Kontrolle des Debuggers befindet. Nur dann erzeugt der Emulator Nachrichten über den Zustand dieses Threads, etwa, wenn er blockiert oder stirbt.

Step-Bit Ist das *Step-Bit* gesetzt, so bedeutet dies, daß sich der Thread im Einzelschritt-Modus befindet. Er läuft dann genau bis zum Erreichen des nächsten Steppunktes und hält dort an.

4.4 Modifikationen des Stacks

Im Kapitel 2.2 wurde beschrieben, wie die interne Repräsentation des *Taskstacks* der Oz-Maschine aussieht. Andererseits wurde in Kapitel 3.3.4 die Bedeutung des

Aufrufstacks erklärt. Beide Begriffe müssen sorgfältig auseinander gehalten werden, da sie unterschiedliche Dinge beschreiben: Im Taskstack stehen sogenannte *Fortführungen* (engl. „continuations“), also im wesentlichen Programmzähler, die angeben, wo im jeweiligen Stacksegment nach Rückkehr aus einer Prozedurapplikation fortzufahren ist, während im (nur konzeptuell existierenden) Aufrufstack die Stellen der Prozedurapplikationen zusammen mit den Prozeduren selbst vermerkt sind. Ein kleines Beispiel soll den Unterschied verdeutlichen:

```
 1 local                 % Aufrufstack            Taskstack
 2   proc {F X}          %================================
 3     proc {G Y}        % 4: -> {Show 21}, 4
 4       {Show X+Y}      % 3: -> {G 4}, 8          3: G, 5
 5       {Show g#done}   % 2: -> {F 17}, 12       2: F, 9
 6     end               % 1: -> {Top}, intern    1: Top, 13
 7   in
 8     {G 4}
 9     {Show f#done}
10   end
11 in
12   {F 17}
13   {Show top#done}
14 end
```

Neben dem Quelltext finden sich sowohl der Aufruf- als auch der Taskstack unmittelbar vor Applikation des Builtins `Show` in Zeile 4. Die Zahlen hinter dem Komma geben Zeilennummern an, beim Taskstack sind sie das Ergebnis einer Abbildung von Fortführungsadressen auf Zeilennummern. Beispielsweise wird die Ausführung des Rumpfes der Prozedur `F` nach Beendigung der Applikation von `G` in Zeile 9 fortgesetzt. `Top` ist eine vom Compiler bei jeder Anfrage generierte Prozedur, die als Rumpf den Code der Anfrage selbst enthält, und die bei Ausführung des Codes intern appliziert wird.

Folgendes fällt bei Betrachtung beider Stacks auf:

- Der Taskstack ist einen Eintrag kleiner als der Aufrufstack, da die Oz-Maschine das Builtin direkt appliziert, ohne es vorher auf den Taskstack zu legen. Auf dem Aufrufstack dagegen erscheint es.
- Die Zeilennummer in Höhe n des Aufrufstacks ist immer um eins kleiner als die in Höhe $n-1$ des Taskstacks.

Für den Anwender ist der Taskstack eigentlich uninteressant. Er enthält Informationen, die die Oz-Maschine zum korrekten Arbeiten benötigt, die ansonsten aber eher irritieren als nützen. Eine frühe Idee bei der Implementierung des Debuggers war, die oben erwähnten Ähnlichkeiten zwischen beiden Stacks auszunutzen und eine Abbildung zwischen ihnen zu realisieren. Sie sah vor, aus je *zwei* Taskstack-Segmenten *ein* Aufrufstack-Segment zu konstruieren:

```
%  3: G, 5
%                ==>  2: {G ...}, 8
%  2: F, 9
%                ==>  1: {F ...}, 12
%  1: Top, 13
```

Dies stellte sich jedoch bald als unbefriedigend heraus, aus mehreren Gründen: So versagt das Mapping, wenn nur ein einzelnes Segment auf dem Taskstack liegt. Außerdem fehlt Information über die Argumente der Applikation. Gesucht wurde nach einem Weg, wie der Taskstack so ergänzt werden kann, daß er folgende Bedingungen erfüllt:

- **Vollständigkeit** Sämtliche Daten, die zum Aufbau des Aufrufstacks benötigt werden, sollen in ihm zu finden sein.
- **Robustheit** Auch bei Ausführung von Code, der nur teilweise mit Debuginformation kompiliert wurde, soll ein korrekter und maximal aussagekräftiger Aufrufstack generiert werden können.
- **Konsistenz** Der Stack, der durch Verfolgen und Verarbeiten der einzelnen `entry`- und `exit`-Nachrichten bei Einzelschrittausführung entsteht, soll weitestgehend mit demjenigen Stack übereinstimmen, den man auf Anfrage per Builtin vom Emulator mitgeteilt bekommt.

Insbesondere die letztgenannte Eigenschaft ist wichtig zur Einsparung von Ressourcen, da der Stack dann über weite Strecken von der Debugger-Shell selbst berechnet werden kann, und nicht teilweise redundante Information wieder und wieder vom Emulator angefordert werden muß.

Der nächste Abschnitt zeigt des Problems Lösung: Es werden – ähnlich zur Debuginformation im Programmcode – *Debugsegmente* auf den Taskstack gelegt, die sämtliche oben geforderten Eigenschaften zusichern und außerdem eine elegante Implementierung der Operation *unleash* zulassen. Insofern dienen sie sowohl zur Ausführungskontrolle als auch zur Kontextbestimmung.

4.4.1 Ein Spezialtask

Der neue Spezialtask hat einen ähnlichen Aufbau wie die übrigen Spezialtasks des Emulators: Der Programmzähler hat einen besonderen Wert, der den Typ des Tasks anzeigt (`C_DEBUG_CONT_Ptr`), `Y` zeigt auf ein Objekt der Klasse `OzDebug`, welches verschiedene stackrelevante Informationen speichert, `G` schließlich enthält eine der Konstanten `STEP`, `NOSTEP` oder `EXIT`. Deren Bedeutung wird in Kürze bei der Beschreibung der Operation *unleash* erklärt.

Wie und wann gelangen die Debugsegmente nun auf den Stack? Wie in Abschnitt 4.1.2 erläutert, werden bei Erzeugung von Debugcode um jeden Steppunkt `debugEntry`- und `debugExit`-Instruktionen gesetzt. Die Idee ist nun, in den Taskstack des Emulators mit Hilfe dieser Instruktionen und der Debugsegmente einen „Stack aus Steppunkten“ so einzubetten, daß die zusätzlichen Tasks ausschließlich zu Debuggingzwecken verwandt werden, jedoch niemals selbst zur Ausführung kommen, weil sie stets rechtzeitig wieder vom Stack entfernt werden.

Konkret funktioniert das folgendermaßen: Bei Ausführung einer `debugEntry`-Instruktion wird zunächst ein neues `OzDebug`-Objekt erzeugt und je nach Steppunkt-Typ mit relevanten Daten initialisiert. Das Feld `PC` zeigt auf die `debugEntry`-Instruktion (um von dort bei Bedarf Informationen über Dateinamen, Zeile und Spalte zu extrahieren), `Y` und `G` zeigen auf die Felder der lokalen und globalen Variablen. Bei Prozedurapplikationen finden sich im Feld `Data` die Prozedur

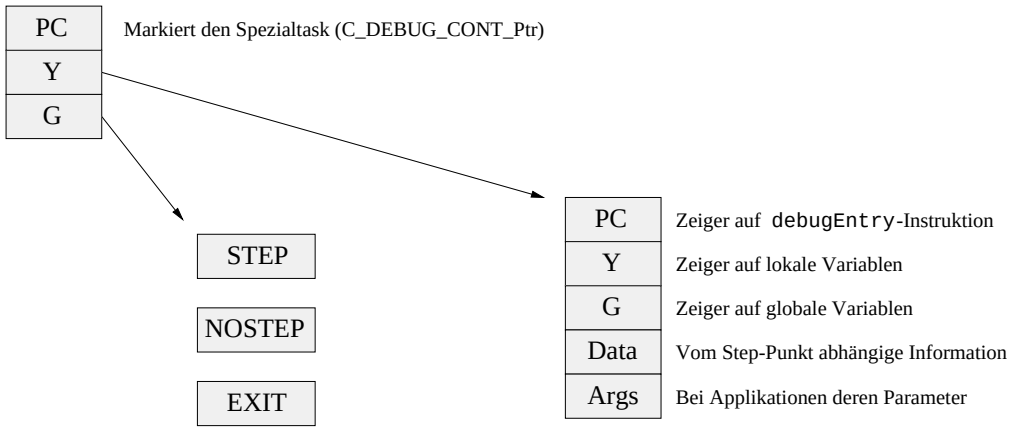


Abbildung 4.1: Ein Debugsegment mit dazugehörigem `OzDebug`-Objekt

selbst und im Feld `Args` ihre Parameter. Bei Konditionalen liefert `Data` den Arbiter, bei Locks das Lock, `Args` ist jeweils undefiniert. Bei Ausnahmenfängern sind `Data` und `Args` undefiniert.

In der Regel wird die nächste Instruktion nach `debugEntry` ihrerseits eine Fortführungsadresse auf den Stack legen, so daß der Taskstack aus dem letzten Beispiel nun folgende Gestalt annimmt (Die Debugsegmente wurden durch ein initiales D gekennzeichnet):

```
%  erweiterter Taskstack
% ======================
%  6: G, 5
%  5: D [ -> {Show 21}, 4 ]
%  4: F, 9
%  3: D [ -> {G 4}, 8 ]
%  2: Top, 13
%  1: D [ -> {F 17}, 12 ]
```

Bei Ausführung der Instruktion `debugExit` wird das (dann wieder ganz oben auf dem Stack liegende) Spezialsegment vom Stack entfernt.

4.4.2 Erfragen des Aufrufstacks

Über das Builtin `Thread.taskstack` kann der Aufrufstack eines Threads angefordert werden. Hierzu durchsucht es den Taskstack nach allen Debugsegmenten und generiert daraus einen Aufrufstack. Dabei wird sichergestellt, daß auch bei der Vermengung von Code mit und ohne Debuginformation keine Segmente verloren gehen, indem beim Auffinden von zwei „herkömmlichen Segmenten“ ohne dazwischenliegendes Debugsegment zumindest der Prozedurname im Aufrufstack erscheint, denn dieser ist auch in Nicht-Debugcode vermerkt, zusammen mit der Position im Quellcode, wo die Definition der Prozedur beginnt.

Je nach Aufruf des Builtins enthält ein Stacksegment entweder direkt – soweit bekannt – alle lokalen und globalen Variablen, oder lediglich eine Segmentnummer, über die zu einem späteren Zeitpunkt die Variablen angefordert werden können. Abschnitt 4.6 geht vertiefend auf die Untersuchung der Umgebung ein.

4.4.3 Die Operation *unleash*

Idee

Mit Hilfe der neuen Datenstrukturen aus dem letzten Abschnitt kann leicht ein Mechanismus implementiert werden, der einen Thread genau so lange weiterlaufen läßt, bis ein bestimmtes Debugsegment freigeräumt wurde. Genau hier kommen die Konstanten `STEP`, `NOSTEP` und `EXIT` ins Spiel. Die Idee ist wiederum sehr einfach: Beim Entfernen eines Debugsegments (also bei Ausführung von `debugExit`) wird der Wert von `G` ausgelesen. Ist er `NOSTEP`, so kann der Thread weiterlaufen. Ist er dagegen `STEP`, so wird der Thread angehalten, dafür gesorgt, daß bei seiner Fortsetzung `debugExit` erneut ausgeführt wird, und das Debugsegment wieder auf den Stack zurückgelegt, allerdings mit der Konstante `EXIT`. Dies ist ein Trick mit folgendem Hintergrund: Die Information, daß der Thread gerade einen Steppunkt verlassen hat, soll noch auf dem Stack zu finden sein, obwohl der nicht erweiterte Taskstack darüber keine Auskunft mehr gibt. Andererseits muß aber sichergestellt werden, daß bei Weiterlaufen des Threads (also bei Wiederausführung von `debugExit`) nicht unmittelbar wieder angehalten wird und das Debugsegment ordnungsgemäß vom Stack entfernt wird. Insofern erfüllt `EXIT` denselben Zweck wie `NOSTEP`, liefert allerdings zusätzlich die Information, daß der Steppunkt verlassen wurde.

Realisierung

Wie funktioniert nun die Operation *unleash*? Sie ist als zweistelliges Builtin definiert:

```
void unleash(Thread *t, int frame);
```

Bei Eingabe eines Threads und einer Segmentnummer durchläuft es einmal komplett den Stack des Threads und manipuliert ihn dahingehend, daß alle Debugsegmente, die oberhalb der übergebenen Nummer liegen, mit `NOSTEP`, alle übrigen mit `STEP` markiert werden. Anschließend darf der Thread weiterlaufen.

Beispiel

Unter der Annahme, daß der Befehl *unleash to frame 5* abgesetzt wurde, sieht der Stack aus dem letzten Beispiel nach Wiederanhalten des Threads so aus:

```
%  erweiterter Taskstack
% ======================
%  5: D [ <- {Show 21}, 4 ]
%  4: F, 9
%  3: D [ -> {G 4}, 8 ]
%  2: Top, 13
%  1: D [ -> {F 17}, 12 ]
```

Man beachte, daß ohne Debugsegmente nun die Information über die momentane Position in Prozedur *G* verloren wäre.

4.5 Haltepunkte

Der Oz-Debugger unterstützt zwei Arten von Haltepunkten, *statische* und *dynamische*, die in ihrer Wirkung identisch sind, aber in unterschiedlichen Situationen eingesetzt werden.

4.5.1 Statische Haltepunkte

Die Idee ist, daß ein statischer Haltepunkt verschiedene Invokationen der Oz-Maschine „überleben“ soll, daher keine Markierung im Codesegment des Emulators darstellt, sondern im Quelltext. Konkret handelt es sich um die Applikation des nullstelligen Builtins `Debug.breakpoint`, die genau vor der Stelle im Quellcode gemacht wird, an der die Programmausführung unterbrochen werden soll. Hierdurch werden Trace- und Step-Bit des aktuellen Threads gesetzt, sofern eines der beiden noch nicht gesetzt war. Beim nächsten Steppunkt wird der Thread daher anhalten und eine `entry`-Nachricht für die Debugger-Shell generieren.

Sofern der Schalter `runwithdebugger` gesetzt ist, erzeugt der Compiler während einer Debugsitzung statische Haltepunkte am Anfang jedes generierten Codeblocks, so daß durch Emacs-Anfragen erzeugte Threads automatisch erst einmal anhalten.

4.5.2 Dynamische Haltepunkte

Wie ihr Name vermuten läßt, können dynamische Haltepunkte während der Laufzeit eines Threads gesetzt bzw. gelöscht werden. Dazu werden sie im Codesegment des Emulators vermerkt.

Prinzip

Viele Debugger *ersetzen* dazu diejenige Instruktion, an der das Programm anhalten soll, durch eine spezielle *Haltepunktinstruktion* und sichern den ursprünglichen Code an anderer Stelle, um ihn bei Bedarf wieder zurückschreiben zu können [Ros96]. Der Oz-Debugger geht einen anderen, trickreichen Weg, indem die `debugEntry`-Instruktion, die sich vor jedem Steppunkt befindet, ausgenutzt wird: Das Setzen eines Haltepunktes wird einfach durch Negation der Zeilennummer kodiert. Arbeitet der Emulator im Debugmodus eine `debugEntry`-Instruktion ab, so testet er jedesmal, ob die Zeilennummer negativ ist. Ist dies der Fall, so werden wiederum, genau wie beim statischen Haltepunkt, die entsprechenden Bits des Threads gesetzt, um ihn anhalten zu lassen. Ohne die Existenz von Debuginformation können dynamische Haltepunkte nicht gesetzt werden – eine Einschränkung, die in den meisten Fällen unkritisch sein sollte.

Umsetzung

Das Builtin `breakpointAt` erledigt die Aufgabe des Negierens. Es ist wie folgt definiert:

```
bool breakpointAt(char *file, int line, bool action);
```

Alle im Codesegment des Emulators liegenden `debugEntry`-Instruktionen werden daraufhin untersucht, ob sie die passende Datei- und Zeileninformation besitzen. Existieren solche Instruktionen, so werden, je nach Wert des Argumentes `action`, Haltepunkte dort gesetzt oder gelöscht. In diesem Fall ist der Rückgabewert der Funktion „true“, andernfalls „false“. Um die Suche nach passender Debuginformation effizient zu machen, werden bei jedem Einladen von neuem Code in einer internen Liste des Emulators die wesentlichen Inhalte aller Debuginformationen kompakt gespeichert, zusammen mit den Programmzählern, wo sie im Codesegment zu finden sind.

Probleme

Problematisch wird es, wenn dasselbe Programm mehrmals geladen wird. Es existieren dann mehrere Kopien davon im Codesegment, was die Frage aufwirft, wie mit dynamischen Haltepunkten in solch einem Fall verfahren werden soll. Kapitel 5.2.2 widmet sich dieser Problematik.

4.6 Umgebungen

In Kapitel 3.6.2 wurden verschiedene Möglichkeiten diskutiert, welche Umgebungsinformation in einer Variablenanzeige präsentiert werden soll. In einer frühen Implementierung generierte der Compiler nur für diejenigen globalen Variablen entsprechende `globalVarname`-Instruktionen, die in der Prozedur auch referenziert wurden. Dies sparte zwar Speicher zur Laufzeit, jedoch stellte sich heraus, daß es oftmals wünschenswert ist, auch die Werte von nicht referenzierten Variablen zu untersuchen, was nur umständlich durch Kontextwechsel (Wahl eines anderen Stacksegments) möglich war. Daher wird in der aktuellen Implementierung ein Kompromiß zwischen Speicherverbrauch und Benutzerfreundlichkeit eingegangen, indem die globale Umgebung komplett aufgebaut wird, jedoch mit Filterung aller Toplevel-Variablen.[1]

4.6.1 Zugriff auf Variablen eines Stacksegments

Eine Referenz auf die jeweilige Umgebung wird in jedem Stacksegment gespeichert. Wurde der Code mit dem Compilerschalter `debuginfovarnames` übersetzt, so sind zusätzlich die Variablennamen ermittelbar und eine Untersuchung der Umgebung mit Ozcar ist möglich. Um von dort auf die Variablen eines beliebigen (existenten) Stacksegments zugreifen zu können, wird eine eindeutige *Segmentnummer* benutzt, die sich ohne weiteren Rechenaufwand aus der Höhe des aktuellen Stacks berechnen läßt und in verschiedenen Situationen Ozcar mitgeteilt wird: Jede `entry`- und `exit`-Nachricht auf dem Nachrichtenstrom (s. Abschnitt 4.2) enthält eine Segmentnummer, außerdem können in den Datenstrukturen, die das Builtin `Thread.taskstack` liefert, Segmentnummern enthalten sein.

Zur Anforderung der Umgebung in bestimmter Stackhöhe eines Threads wurde folgendes Builtin implementiert:

```
OZValue getFrameVariables(Thread *t, int frameId);
```

Seine Arbeitsweise ist wie folgt: Im C++-Objekt des Threads t existiert eine Referenz auf dessen Taskstack – ebenfalls ein C++-Objekt. Dessen Methode `getFrameVariables`, aufgerufen mit der Segmentnummer `frameId` als Argument, berechnet aus der Nummer zunächst das passende Stacksegment. Nun existieren zwei Möglichkeiten, je nachdem, wie der Compilerschalter `debuginfocontrol` beim Übersetzen des Codes gesetzt war: Entweder handelt es sich um ein Debugsegment oder um eine „gewöhnliche“ Fortführung. In beiden Fällen ist es in ähnlicher Weise mit Hilfe des gesicherten Programmzählers möglich, das Ende der jeweiligen Prozedur zu erreichen und die dort (eventuell) liegenden Spezialinstruktionen `localVarname` und `globalVarname` auszuwerten. Eine Liste der gefundenen Variablen mit ihren Werten, die über Referenzen im Stacksegment zugreifbar sind, ist schließlich das Ergebnis des Builtins.

[1] Letztere können jederzeit vom Compiler erfragt werden, es geht also keine Information verloren.

4.6.2 Evaluierung von Ausdrücken

In gegebenem Kontext Ausdrücke zu evaluieren oder Anweisungen auszuführen, gehört zur Standardfunktionalität jedes leistungsfähigen Debuggers. Mit Hilfe des in Oz selbst geschriebenen Compilers ist es sehr einfach, auch den Oz-Debugger mit dieser Fähigkeit auszustatten. Der Compiler ist als Klasse realisiert – es können also beliebig viele Instanzen gebildet werden, von denen eine ausgewiesene der *OPI-Compiler* darstellt, mit dem der Benutzer im Entwicklungssystem arbeitet. Jedem Übersetzungsvorgang legt der Compiler eine Umgebung zugrunde, welche z. B. im Entwicklungssystem aus der Toplevel-Umgebung, wie sie durch die Oz-Bibliothek definiert ist, und den vom Benutzer durch **`declare`** eingeführten Variablen besteht.

Um nun einen Ausdruck oder eine Anweisung im Kontext eines bestimmten Stacksegments zu übersetzen, wird zunächst ein neues Compilerobjekt erzeugt und mit der aktuellen Umgebung des OPI-Compilers initialisiert. Auf diese Weise können in Emacs deklarierte Variablen benutzt werden. Anschließend wird es um die globalen und lokalen Variablen des Stacksegments ergänzt. Durch die gewählte Reihenfolge können also im Falle von Namensgleichheit Variablen des OPI-Compilers überschrieben werden. Der virtuelle String mit dem zu übersetzenden Code kann nun dem Compiler übergeben werden. Befindet sich die Variable **`self`** in der Umgebung, so wird dafür gesorgt, daß auch Attributzugriffe mit Hilfe des `@`-Operators möglich sind.

4.7 Ausnahmebehandlung

Wie in Kapitel 2.1.5 erläutert, können gefangene und nicht gefangene Ausnahmen unterschieden werden. Die Debugger-Shell muß in beiden Fällen informiert werden, da sich der Stack des entsprechenden Threads verändert.

4.7.1 Gefangene Ausnahmen

Nach Wurf einer Ausnahme durchsucht der Emulator den Stack nach einem passenden Fänger – mit Erfolg genau dann, wenn das **`try...catch...end`**-Konstrukt verwendet wurde und eine der **`catch`**-Klauseln auf die Ausnahme paßt. Alle Stacksegmente bis zum Fänger sind dann vom Stack entfernt worden, eine eventuell gerade existierende Stackanzeige in der Shell ist also inkonsistent zum realen Stack. Aus diesem Grunde wird die Nachricht `update(thr:T)` auf den Nachrichtenstrom geschrieben, die die Shell anweist, den kompletten Stack von Thread T neu zu berechnen.

4.7.2 Nicht gefangene Ausnahmen

Verläuft die Suche nach einem passenden Fänger negativ, so ist der Stack irgendwann völlig abgeräumt, der entsprechende Thread also terminiert. Die Ursache dieses Laufzeitfehlers soll untersucht werden können, weshalb die Nachricht `exception(thr:T exc:E)` existiert: Sie teilt der Shell mit, daß der Thread T mit der Ausnahme E terminiert wurde. E ist dabei ein Record, in dem verschiedene Informationen zum Typ der Ausnahme kodiert sind. Unter anderem findet sich in

ihm ein *gesicherter Stack*, der genau aus den Segmenten besteht, die auf der Suche nach dem Fänger vom Stack entfernt wurden, ansonsten in seinem Aufbau dem des Builtins `Thread.taskstack` entspricht. Dies ermöglicht die Anzeige eines *Post-Mortem-Stacks*.

Da der eigentliche Stack des Threads gar nicht mehr existiert, kann über Segmentnummern nicht mehr auf die Umgebung zugegriffen werden. Aus diesem Grunde befindet sich bei einem gesicherten Stack alle Variableninformation direkt in den erzeugten Datenstrukturen.

4.8 Benutzeroberfläche

In diesem Abschnitt werden die beiden Komponenten der Benutzeroberfläche, Ozcar und Emacs, vorgestellt.

4.8.1 Ozcar

Das in Oz mit Hilfe des Tcl/Tk-Moduls programmierte Werkzeug Ozcar übernimmt eine Vielzahl von Aufgaben, von denen ich im folgenden die wichtigsten vorstelle. Abbildung 4.2 zeigt das Hauptfenster während einer Debugsitzung.

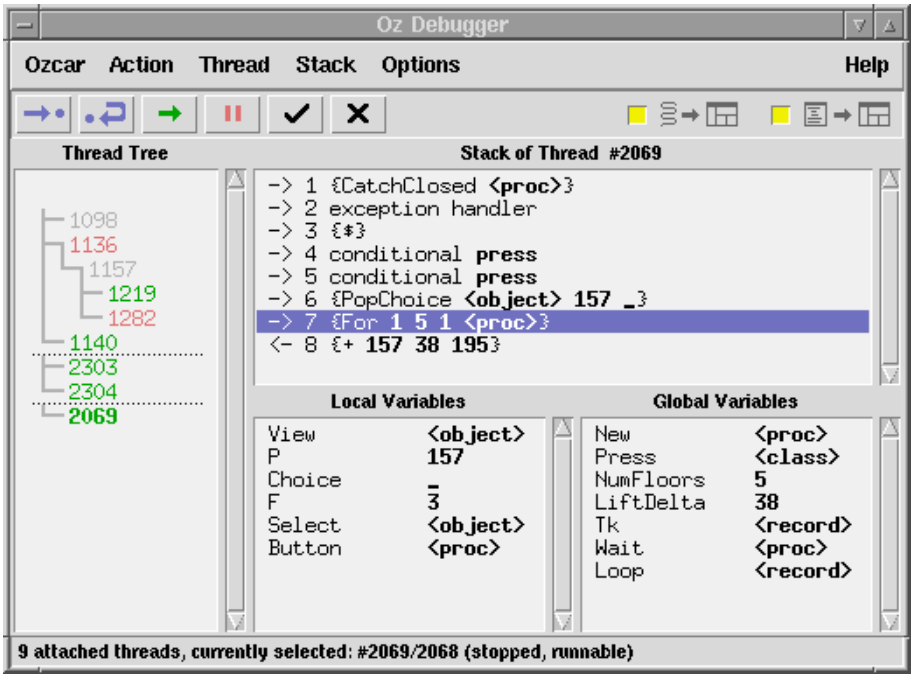


Abbildung 4.2: Ozcar bei der Arbeit

Visualisierung Zunächst ist Ozcar in der Lage, unterschiedliche Zustands- und Kontextinformationen über Threads anzuzeigen. Im linken Fenster werden alle momentan unter Debuggerkontrolle stehenden Threads, durch ihre Threadnummer repräsentiert, aufgelistet. Eine Baumgraphik spiegelt ihre hierarchische Beziehung zueinander wider. Unterschiedliche Farben markieren verschiedene Threadzustände: Grün bedeutet „bereit“, rot „blockiert“ und grau „terminiert“.

Im Stackfenster rechts daneben ist der Stack des aktuell selektierten Threads zu sehen. Er besteht aus acht Segmenten, zuletzt wurde die Summe der Zahlen 157 und 38 berechnet. Selektiert ist Segment Nr. 7, dessen lokale und globale Variablen in den beiden darunterliegenden Fenstern angezeigt werden. Abgesehen von Integer- und Fließkommazahlen, booleschen Konstanten und Atomen werden als Variablenwerte standardmäßig nur Typen angegeben, um Platz zu sparen und die Übersichtlichkeit zu erhöhen. Ein Mausklick auf die Typinformation aktiviert den Browser, der detailliertere Angaben machen kann. Undeterminierte Variablenwerte werden durch einen Unterstrich dargestellt.

Konfiguration Das Verhalten des Debuggers in verschiedenen Situationen kann durch eine Reihe von Schaltern beeinflußt werden.

- **Attach Queries** Diese Option ist gekoppelt mit dem Schalter `runwith-debugger` des OPI-Compilers (vgl. Abschnitt 4.1.4). Ist dieser Schalter gesetzt, so wird in den Maschinencode jeder Anfrage am Anfang ein statischer Haltepunkt eingefügt, so daß der die Anfrage ausführende Thread unmittelbar nach Erzeugung unter Debuggerkontrolle gerät.
- **Attach Subthreads** Ist dieser Schalter aktiviert, so werden alle Threads, die ein bereits angemeldeter Thread erzeugt, automatisch ebenfalls unter Debuggerkontrolle gestellt. Dabei sorgt der Debugger-Kern dafür, daß sie bei Erreichen des ersten Steppunktes anhalten, um dem Programmierer die Wahl offenzuhalten, wie er mit dem Thread verfahren möchte. Experimentiert wurde aber auch mit verschiedenen Optionen, was Ozcar seinerseits mit solchen Threads anstellen könnte. So mag es z. B. interessant sein, jedem Subthread den Befehl *unleash 0* zu geben: Zwar kann dann keinerlei Kontextinformation gewonnen werden, da die Threads ja bis zu ihrem Tode weiterlaufen, jedoch bekommt der Programmierer durch die Anzeige des Threadbaumes eine Vorstellung von Anzahl und hierarchischer Struktur der an einer Berechnung beteiligten Threads.
- **Use Emacs** Ist eine Quelltextanzeige nicht erwünscht, so kann mit diesem Schalter die Erzeugung der Balkengraphik in Emacs-Puffern ausgeschaltet werden.

Steuerung Mit Hilfe verschiedener Knöpfe können die in 3.3.5 definierten Aktionen auf Threads ausgeführt werden. Weiterhin existieren Menüpunkte zum komfortablen Entfernen *aller* angemeldeten Threads, wahlweise auch mit deren vorheriger Terminierung.

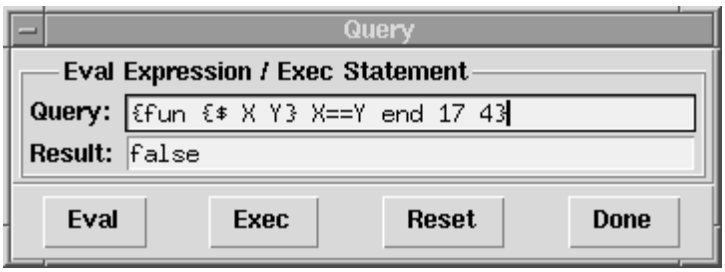


Abbildung 4.3: Der Anfrage-Dialog

In gegebenem Kontext Ausdrücke zu evaluieren oder Anweisungen auszuführen ist Sinn und Zweck des Anfrage-Dialogs aus Abbildung 4.3. Bei jedem Druck auf

einen der Knöpfe *Eval* oder *Exec* wird intern ein neues Compiler-Objekt erzeugt (s. Abschnitt 4.6.2) und mit der Umgebung des aktuell selektierten Stacksegments initialisiert. Anschließend wird die eingegebene Zeichenkette je nach gewähltem Knopf als Ausdruck oder Anweisung übersetzt und das Ergebnis[2] angezeigt. Im Falle von Divergenz oder Verklemmung kann mit dem Reset-Knopf eine laufende Berechnung abgebrochen werden.

4.8.2 Emacs

Von großer Bedeutung für eine schnelle Orientierung während des Debuggens ist die Existenz einer Quelltextanzeige. In einer frühen Version der Implementierung wurde Ozcar daher um ein weiteres Tk-Fenster für diese Aufgabe ergänzt. In diesem konnten durch Mausklicks Haltepunkte gesetzt oder gelöscht werden; weiterhin wurde die momentane Position eines angehaltenen Threads durch verschiedenfarbige Balken in den Quelltextzeilen markiert. Allerdings hatte diese Lösung einen entscheidenden Nachteil: Ein Editieren des Quelltextes innerhalb des Tk-Fensters war nicht möglich und wäre auch nur mit unverhältnismäßig großem Aufwand zu realisieren gewesen. Es lag daher nahe, den Emacs-Editor, der bei interaktiver Benutzung des Mozart-Systems ohnehin in dessen Zentrum steht, stärker in den Debugvorgang zu integrieren.

In der aktuellen Implementierung übernimmt Emacs folgende Aufgaben:

Starten und Stoppen des Debuggers

Die Emacs-Lisp-Funktion (`oz-debug-start <bool>`) sendet über den OPI-Compiler je nach übergebenem Argument die Nachrichten `{Ozcar on}` bzw. `{Ozcar off}`. Alles weitere regelt Ozcar: Im ersten Fall öffnet er das Hauptfenster, versetzt den Emulator in den Debugmodus und aktiviert verschiedene Compilerschalter. Im zweiten Fall passiert genau das Gegenteil, dabei werden eventuell noch unter Debuggerkontrolle stehende Threads nicht terminiert, so daß durch erneuten Start von Ozcar genau dort weitergearbeitet werden kann, wo zuvor unterbrochen wurde.[3]

Setzen und Löschen von Haltepunkten

Die Funktion (`oz-breakpoint <bool>`) sendet die Nachricht

```
{Ozcar bpAt(<file> <line> <yesno>)}
```

wobei `<file>` und `<line>` der aktuellen Position des Cursors entsprechen, und `<yesno>` ein boolescher Wert ist, der anzeigt, ob ein dynamischer Haltepunkt gesetzt oder gelöscht werden soll. Ozcar sorgt für den Rest (s. Abschnitt 4.5.2).

Positionsinformation

Zur Anzeige der aktuellen Zeile und Spalte eines angehaltenen Threads verwaltet Emacs sogenannte *Overlays*. Dies sind Textbereiche mit definierbaren Textattributen, die beliebig in allen Emacs-Puffern verschoben werden können. Gesteuert wird die Positionierung der Overlays von Ozcar, der eine „magische Zeichenkette“ auf die Standardausgabe des Emulators druckt, die bei einer interaktiven Programmiersitzung mit dem Emacs-Puffer `*Oz Emulator*` verbunden ist:

```
proc {MagicEmacsBar File Line Column State}
   Output =
   proc {$ VS}
```

[2] Anweisungen liefern das Ergebnis **`unit`** zurück.

[3] Voraussetzung ist allerdings, daß bei Deaktivierung von Ozcar alle angemeldeten Threads gestoppt waren.

```
      {Print {String.toAtom {VirtualString.toString VS}}}
   end
in
   {Output 'oz-bar '#File#' '#Line#' '#Column#' '#State}
end
```

Aus diesem Puffer filtert Emacs alle magischen Zeichenketten und wertet sie aus. Liest er z. B. `'oz-bar Oz 4 5 blocked'`, so zeichnet er zwei Overlays in Zeile 4 des Buffers `Oz` wie in Abbildung 4.4.

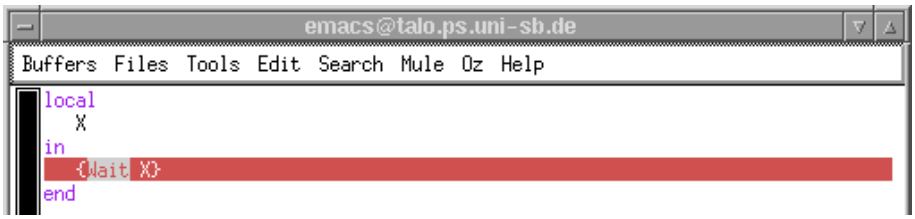


Abbildung 4.4: Emacs zeigt das Verweilen eines Threads auf dem Builtin `Wait` an

Doch Emacs liefert nicht nur dem Programmierer Positionsinformation in Form farbiger Balken, sondern auch dem Compiler beim Einladen von Anfragen in Form von `\line`-Direktiven, sofern der Compiler nicht selbst aus einer Datei liest, sondern den Quelltext in Form eines virtuellen Strings zur Verfügung gestellt bekommt. Dies ermöglicht die Erzeugung von Debuginstruktionen mit korrekter Positionsinformation, unabhängig von der jeweils verwendeten Anfrage-Variante.

4.9 Umfang des Quelltextes

Zur Realisierung des Debuggers waren an vielen Stellen in allen wesentlichen Komponenten des Mozart-Systems Änderungen und Erweiterungen nötig. Außerdem mußte die Benutzeroberfläche Ozcar geschrieben werden. Letztere besteht aus gut 4000 Zeilen Quelltext. Im Emulator kamen etwa 1000 Zeilen C++-, im Compiler 500 Zeilen Oz-Code hinzu. Für die Steuerung des Debuggers durch Emacs sowie die Implementierung der Quelltextanzeige waren noch einmal 200 Zeilen Emacs-Lisp-Code nötig.

5

Diskussion

In diesem letzten Kapitel gehe ich zunächst auf verschiedene Entwurfsentscheidungen ein, die bei der Entwicklung des Debuggers eine Rolle gespielt haben. Anschließend widme ich mich den Problemen, die sich durch inkrementelle Codegenerierung in einer interaktiven Programmierumgebung ergeben, bevor eine Effizienzanalyse die vorliegende Implementierung auf ihre Tauglichkeit zur Unterstützung größerer Programmierprojekte prüft. Ein Ausblick auf mögliche Weiterentwicklungen des Debuggers rundet das Kapitel ab.

5.1 Entwurfsentscheidungen

Zu Beginn dieser Diplomarbeit war noch völlig unklar, mit welchem Aufwand die Entwicklung eines Debuggers für das Mozart-System zu realisieren sein würde, welche Ansprüche an einen Debugger für die Sprache Oz gestellt werden müssen, und ob das ganze Projekt überhaupt gelingt. Auf dem Weg bis zur nun vorliegenden Implementierung wurden viele Ideen geboren, im Gespräch mit Kommilitonen diskutiert, mit Hilfe von Prototypen evaluiert, und manchmal teilweise oder vollständig wieder verworfen.

Debugging-Ebene

In Kapitel 1.2 wurde bereits unterschieden zwischen Debuggern auf Maschinen- und solchen auf Quellsprachebene. Ein (recht einfacher) Debugger der ersten Kategorie existierte bereits zu Beginn dieser Arbeit. Er war jedoch primär für die Entwickler von Mozart selbst gedacht und nicht für Anwender, die Programme in Oz schreiben. Die Möglichkeiten kennend, die integrierte Entwicklungsumgebungen wie beispielsweise die Borland IDE oder der GNU-Debugger in Verbindung mit Emacs für Quellebenen-Debugging bieten, war recht schnell der Entschluß gefaßt, etwas ähnliches auch für Oz zu realisieren.

Nun ist das Mozart-System in seiner Gesamtheit jedoch sehr komplex: Es unterstützt objektorientierte, constraintbasierte, verteilte und nebenläufige Programmierung. Es stellte sich die Frage, ob es überhaupt ein Debugmodell gibt, das alle Fähigkeiten von Oz gebührend berücksichtigen kann; schließlich war eines der Hauptziele, keinen spezialisierten Debugger zu entwerfen, der beispielsweise nur Objekte zu untersuchen imstande ist, sondern vielmehr ein Werkzeug, das sich, genügend weit „unten“ im System verankert, zur Fehlersuche im gesamten Oz-Universum eignet. Um verschiedene wohlverstandene Konzepte von existierenden Debuggern sowohl für imperative als auch funktionale Programmiersprachen übernehmen zu können, wurde der Thread als zentrale Datenstruktur

gewählt, ein *Thread-Debugger* entworfen. Um die Arbeit einzugrenzen, wurden dabei verschiedene Festlegungen getroffen:

Kernsprache als Ausgangsbasis

Der Schritt des Compilers, jedes Programm zunächst in die Kernsprache von Oz zu übersetzen, soll dahingehend ausgenutzt werden, daß dieser (des syntaktischen Zuckers beraubte) Zwischencode als Ausgangsbasis für verschiedene Debuggerfunktionen verwendet wird. Dies vereinfacht insbesondere die Benutzerschnittstelle, führt jedoch – aufgrund der feinkörnigen Verteilung von Steppunkten – in einigen Fällen zu (für den weniger geübten Oz-Programmierer) irritierenden Effekten. So wird z. B. in der Zeile

```
case 2 == 1 then {Show wunder} else {Show ok} end
```

zunächst das Builtin == ausgeführt, und dann das Konditional. Bei Einzelschritt-Ausführung springen die Positionsbalken in Emacs also teilweise gegen Leserichtung des Quelltextes. Ursache ist die Kernsprachen-Expansion:

```
local BoolGuard1 UnnestApply1 UnnestApply2 in
  UnnestApply1 = 2
  UnnestApply2 = 1
  BoolGuard1 = UnnestApply1 == UnnestApply2
  case BoolGuard1 then ... end
end
```

Umgang mit Constraints

Der Debugger soll nur für die *flache Teilsprache* von Oz sinnvoll einsetzbar sein; die Unterstützung der Fehlersuche in Constraintprogrammen liegt (zumindest vorerst) außerhalb seines Zuständigkeitsbereiches.[1] Insbesondere die Suche [Sch97] kann nicht abgedeckt werden, schon allein aufgrund der Tatsache, daß es sich dabei um massiv nebenläufige Vorgänge handelt, für die eine völlig andere Benutzeroberfläche entworfen werden müßte. Allerdings sollen tiefe Berechnungen, die z. B. bei Verwendung des relational gebrauchten Konditionals **`if...then...end`** auftreten, durch einen aktiven Debugger nicht unterbunden werden; die dazugehörigen Threads werden in der aktuellen Implementierung einfach ignoriert:

```
local
  fun {Fac N} ... end  %% berechnet z.B. die Fakultät
in
  if {Fac 3} = 1 then {Show wunder} else {Show ok} end
end
```

Selbst nach energischem Setzen von Haltepunkten gelingt es nicht, den (in einem tiefen Berechnungsraum arbeitenden) Thread zur Berechnung von `{Fac 3 _}` bei Ozcar anzumelden.

5.2 Problematik der inkrementellen Codegenerierung

In der interaktiven Entwicklungsumgebung Mozart werden alle Programme von einer ausgewiesenen Compilerinstanz, dem OPI-Compiler, übersetzt (s. Kapi-

[1] Die meisten Berechnungen mit Constraints spielen sich in sogenannten *tiefen Berechnungsräumen* ab, deren korrekte Behandlung durch den Debugger mit einigen Schwierigkeiten verbunden ist.

tel 2.2). Dieser arbeitet inkrementell, die jedem Übersetzungsvorgang zugrunde gelegte Umgebung vergrößert sich also monoton. Aber auch im Emulator wächst ein Speicherbereich kontinuierlich, nämlich das Codesegment, in welchem aller Maschinencode abgelegt wird.[2] Verschiedene für den Debugger essentielle Abbildungen gestalten sich dadurch schwierig; konkret handelt es sich zum einen um die Zuordnung einer Maschineninstruktion zu einer entsprechenden Quelltextzeile, zum anderen um die Zuordnung eines logischen Haltepunktes zu einem passenden physikalischen Pendant.

5.2.1 Aktualität des Quelltextes

Nehmen wir an, das folgende Programm wurde aus einem Emacs-Puffer geladen, und die Programmausführung ist vor Applikation des Builtins `Show` gestoppt (die Zeilenposition des Pfeils entspricht der des Balkens in Emacs, die Richtung der des Pfeils aus der Stackanzeige von Ozcar):

```
1    declare
2    proc {P X}
3 =>    {Show X}
4    end
5    {P 42}
```

Im Codesegment ist also vermerkt, daß sich die Applikation in der dritten Zeile befindet. Nehmen wir weiter an, daß nun oberhalb des Programms weitere Zeilen in den Puffer eingefügt werden, bevor schließlich die Applikation zur Ausführung kommt:

```
1    local X = 17 Y = 4 in
2       {Show X+Y}
3 <= end
4
5    declare
6    proc {P X}
7       {Show X}
8    end
9    {P 42}
```

Die Positionsinformation, die der Emulator liefert, ist jetzt offensichtlich inkonsistent zum tatsächlichen Inhalt des Emacs-Puffers. Wie kann hier vorgegangen werden? Mehrere Ideen wurden durchdacht:

- Die Inkonsistenz wird belassen, der Programmierer jedoch darüber informiert, daß die Positionsinformation möglicherweise nicht korrekt ist, je nachdem, was der Vergleich liefert zwischen dem Zeitpunkt, an dem der Maschinencode erzeugt wurde, und dem Zeitpunkt der letzten Modifikation des Puffers.
- Die Inkonsistenz wird behoben, indem Emacs alle Veränderungen eines Puffers mitprotokolliert und auf die vom Emulator gelieferten Koordinaten entsprechende Offsets hinzuaddiert. Diese Vorgehensweise dürfte allerdings sehr fehlerträchtig sein.

[2] Das Codesegment bleibt bei der automatischen Speicherbereinigung unangetastet.

5.2.2 Haltepunktverwaltung

Die inkrementelle Codegenerierung hat noch weitere Konsequenzen, denn sie verhindert eine einfache, intuitive Haltepunktverwaltung. Um das Dilemma vor Augen zu führen, sei folgender Quelltext gegeben:

```
1    declare
2    proc {Foo}
3       {Show 'foo!'}
4    end
5
6    declare
7    proc {Foo}
8       {Show 'bar!'}
9    end
10
11   {Foo}
```

Nehmen wir an, es werden zunächst die Zeilen 1-4 geladen, gefolgt von Zeile 11, anschließend die Zeilen 6-9, nochmals gefolgt von Zeile 11. Werden nun beide Threads, auf der selben initialen Zeile 11 startend, in einzelnen Schritten abgearbeitet, so ist die Verwunderung groß, wenn der eine Thread in Zeile 3, der andere jedoch in Zeile 8 springt – `Foo` ist eben nicht gleich `Foo` ...

Sei nun vor Abarbeitung der beiden Threads ein dynamischer Haltepunkt in Zeile 3 gesetzt worden. Der eine Thread wird anhalten, der andere nicht. Doch gehen wir noch weiter und laden nun erneut Zeile 1-4. Jetzt existieren also zwei Instanzen des Maschinencodes für die erste `Foo`-Prozedur im Codesegment des Emulators. Alle weiteren Applikationen von `Foo` werden nun nicht mehr in Zeile 3 anhalten, denn sie führen eine Kopie des Maschinencodes aus, in welcher der zur Quelltextzeile korrespondierende physikalische Haltepunkt gar nicht gesetzt ist.

Welche Lösungsideen existieren für diese Problematik?

- Emacs übernimmt eine Verwaltung aller dynamischen Haltepunkte und sorgt dafür, daß in sämtlichen Instanzen des Maschinencodes die physikalischen Haltepunkte automatisch gesetzt werden (durch passende Aufrufe von `bpAt(...)` nach jeder Anfrage, s. Kapitel 4.8.2). Ein passendes Einfärben von Zeilen, in denen Haltepunkte gesetzt sind, ist dann leicht realisierbar. Sobald jedoch der Compiler Teile des Quelltextes selbst aus einer Datei liest (etwa aufgrund einer `\insert`-Direktive), ist Emacs hilflos.
- Der Emulator verwaltet dynamische Haltepunkte. Dies behebt das eben genannte Problem, schafft aber ein neues, denn die Anzeige von Haltepunkten durch Emacs ist erschwert – dieser müßte bei jeder Erzeugung eines neuen Puffers beim Emulator nachfragen, ob eventuell Haltepunkte mit passendem Puffernamen vorliegen.

5.3 Effizienzbetrachtungen

Jeder Debugger trägt mit sich die Last der Debuginformation herum. Insbesondere auf folgende drei Parameter wirkt sich dies negativ aus:

- Codegröße
- Ausführungsgeschwindigkeit
- Speicherbedarf

Während ein Anwachsen der Codegröße im allgemeinen unkritisch ist,[3] kann eine wesentliche Verlangsamung des Codes bereits den Nutzen eines Debuggers in Frage stellen, zu hoher Speicherverbrauch seinen Einsatz gar unmöglich machen.

Mit dem Oz-Debugger wurden verschiedene Tests durchgeführt, die vor allem Aufschluß über seine Tauglichkeit zur Fehlersuche in größeren Programmen geben sollten. Dies lieferte zum Teil überraschende Ergebnisse, vor allem hinsichtlich des Speicherbedarfs. Um es vorweg zu nehmen: Trotz einiger Ineffizienzen läßt sich mit der vorliegenden Implementierung auch im zur Zeit größten existierenden Oz-Programm, dem Compiler für Oz, bereits hinreichend komfortabel Fehlersuche betreiben.

5.3.1 Codegröße

Mehrere Faktoren sorgen dafür, daß Debugcode deutlich größer als Nicht-Debugcode ist. So führt der Compiler verschiedene Optimierungen nicht durch und fügt sowohl an Steppunkten als auch hinter Prozeduren besondere Instruktionen ein. Deren Aufbau und Funktion lassen folgende allgemeine Aussagen zu:

Variablen-Anzahl

Je mehr lokale und globale Variablen in einer Prozedur definiert werden, desto größer wird der erzeugte Code. Dies hat seine Ursache in der Tatsache, daß pro Variable genau eine Spezialinstruktion hinter die Prozedur gesetzt wird, die Auskunft über den Namen der Variable gibt, und *jede* lokale Variable tatsächlich erzeugt wird, auch wenn sie vom Compiler wegoptimiert werden könnte, wie in folgendem Beispiel:

```
local X Y in
   {Show 'X und Y werden gar nicht benutzt'}
end
```

Steppunkt-Anzahl

Der Code wächst mit der Anzahl der Steppunkte im Programm, bedingt durch die Spezialinstruktionen, die sich vor und hinter jedem solchen Punkt befinden.

Zur Gewinnung konkreter Zahlen wurden mehrere Oz-Werkzeuge mit verschiedenen Typen von Debuginformation kompiliert (vgl. Kapitel 4.1.4) und die sich ergebenden Dateigrößen mit dem „Original“ verglichen. Das Ergebnis zeigt Tabelle 5.1.

Auffällig ist der deutlich größere Code des Browsers, wenn Variableninformation einkompiliert wird. Eine Erforschung der Ursache liefert die Erkenntnis, daß der Programmierstil wesentlichen Einfluß auf den Vergrößerungsfaktor hat. Betrachten wir dazu zwei semantisch äquivalente Formulierungen eines Programms:

Der erste Programmierstil kann mit „Halte Schachtelungstiefe gering, deklariere Variablen am Stück“ umschrieben werden, während auf den zweiten „Halte

[3] Festplattenplatz ist heutzutage billig, und obgleich das Codesegment im Emulator zur Laufzeit *einmalig* mehr Speicher belegt, ist dies relativ zum vergrößerten Gesamtspeicherverbrauch *pro Codeausführung* vernachlässigbar.

Annotationen	Compiler		Browser		Explorer		Ozcar		Profiler	
keine	412k	1,0	180k	1,0	113k	1,0	92k	1,0	44k	1,0
Steppunkte	753k	1,8	319k	1,8	201k	1,8	174k	1,9	79k	1,8
Variablen	2.062k	5,0	1.958k	10,9	507k	4,5	568k	6,2	130k	3,0
maximal	2.379k	5,8	2.067k	11,5	587k	5,2	627k	6,8	156k	3,5

Tabelle 5.1: Der Einfluß verschiedener Compilerschalter auf die Größe einiger Oz-Werkzeuge

Sichtbarkeitsbereich von Variablen minimal, nimm dafür größere Schachtelungstiefen in Kauf“ paßt. Problematisch beim ersten Stil ist, daß alle Prozeduren `P1` bis `P5` die 26 ungebundenen Variablen in ihrer globalen Umgebung stehen haben, keine einzige von ihnen aber tatsächlich referenziert wird. Andererseits wirkt der Quelltext kompakter und damit übersichtlicher.

5.3.2 Ausführungsgeschwindigkeit und Speicherbedarf

Bedingt durch das Ausführungsmodell entstehen höhere Laufzeiten der Programme immer dann, wenn Debuginformation einkompiliert wurde. Zusätzlich spielt es eine Rolle, ob sich der Oz-Emulator im Debugmodus befindet, denn nur dann wird der in Kapitel 4.4.1 vorgestellte Spezialtask in den Stack integriert.[4]

Zum Testen der Geschwindigkeit und des Speicherverbrauchs verschieden übersetzten Codes wurden mehrere Programme mit unterschiedlicher Debuginformation versehen und je einmal mit eingeschaltetem bzw. ausgeschaltetem Debugmodus ausgeführt. Dabei kam ein pentiumbasierter PC mit einer Taktrate von 133 MHz und dem Betriebssystem Linux zum Einsatz. Als ein interessantes Beispiel sei folgendes Programm angeführt. Es berechnet pythagoreische Tripel, also Zahlen, die der Gleichung $a^2 + b^2 = c^2$ genügen, wobei $a < b$ und $b \leq 200$.

Speicherverbrauch

Während die Geschwindigkeit bei unterschiedlichen Testkonfigurationen stark variiert (s. u.), hält sich der Speicherbedarf erfreulich konstant bei niedrigen 100 kB. Dies ist einer Optimierung zu verdanken, die erst aufgrund schlechter Meßergebnisse in einem früheren Testlauf vorgenommen wurde. Hier stieg der Speicherverbrauch steil an (auf 40 MB und mehr), sobald der Compilerschalter `debuginfovarnames` gesetzt war. Eine Ergründung der Ursache ergab, daß hierfür die relationale Syntax der Basissprache von Oz (mit-)verantwortlich ist, in der jedes Funktionsergebnis an eine Variable gebunden wird (s. Kapitel 2.1.1). Diese wurde bei Nicht-Debugcode wegoptimiert, sonst jedoch stets erzeugt, mit dem Effekt, daß die automatische Speicherbereinigung jede Menge zu tun hatte.

Die Optimierung besteht nun darin, die Ergebnisvariable eines funktional benutzten Builtins bei Ausführung von Debugcode nur dann zu erzeugen, wenn eine Betrachtung des Ergebnisses in Ozcar möglich ist, konkret also, wenn sich der ausführende Thread im Step-Modus befindet. Die Speicherersparnis hierdurch kann enorm sein – im Testbeispiel beträgt sie über 99 Prozent! Allerdings handelt es sich in dieser Hinsicht um ein „ideales“ Beispiel, denn es wird nicht eine einzige lokale Variable deklariert. Im Normalfall fällt der Gewinn geringer aus, wie im zweiten Test (s. u.) ermittelt.

[4] Diese Aussage ist nicht ganz korrekt: Um den Stack eines bestimmten Threads unabhängig vom Debugmodus des Emulators konsistent zu halten, erfolgt eine Verwaltung seiner Spezialtasks auf jeden Fall dann, wenn sein Trace-Bit gesetzt ist.

Geschwindigkeit

Das Ergebnis der Geschwindigkeitsmessung ist in Tabelle 5.2 dargestellt. Zu bemerken sind folgende Zusammenhänge: Kommt Code ohne Debuginformation zur Ausführung, so ist es für die Geschwindigkeit irrelevant, ob der Debugger eingeschaltet ist oder nicht. Dies ist wichtig vor allem für den Compiler, der sonst während einer Debug-Sitzung langsamer wäre als sonst.

Annotationen	Debugmodus aus		Debugmodus an	
keine	4,3s	1,0	4,3s	1,0
Steppunkte	10,9s	2,5	39,5s	9,2
Variablen	4,9s	1,1	5,0s	1,2
maximal	11,3s	2,6	40,2s	9,3

Tabelle 5.2: Ausführungsdauer der Applikation `{Pyth 200 _}`

Die Existenz von Kontrollfluß-Information wirkt sich stark auf die Geschwindigkeit aus, denn es werden die zusätzlichen Instruktionen `debugEntry` und `debugExit` bei jedem Steppunkt ausgeführt; Aufbau und Verwaltung des in den Taskstack eingebetteten Aufrufstacks verursachen weitere Kosten, wenn sich der Emulator im Debugmodus befindet. Umgebungs-Information dagegen hemmt den Programmfluß nur unwesentlich – verständlich, kommt doch der meiste zusätzliche Code niemals zur Ausführung.

Das vorige Beispielprogramm ist insofern interessant, als es den *best-case* hinsichtlich des Speicherbedarfs und den *worst-case*, was Ausführungsgeschwindigkeit anbelangt, demonstriert. Allerdings ist es sehr klein und benutzt nur in recht bescheidenem Umfang die Möglichkeiten von Oz, so daß es sich nicht zur Messung realistischer *Durchschnittswerte* für die Performanz des Debuggers eignet. Eine zweite Meßreihe wurde daher angefertigt; diesmal mußte der Compiler, mit jeweils unterschiedlicher Debuginformation versehen, die Debugger-Shell Ozcar übersetzen. Tabelle 5.3 zeigt das Ergebnis. Um den hohen Zahlen zum Speicherverbrauch den Schrecken zu nehmen, sei angemerkt, daß es sich hierbei um die Summe allen *temporär* benötigten Speichers handelt, der regelmäßig von der automatischen Speicherbereinigung wieder freigegeben werden kann.

Annotationen	Debugmodus aus				Debugmodus an			
keine	16,8s	1,0	13,1M	1,0	16,9s	1,0	13,1M	1,0
Steppunkte	78,2s	4,7	33,8M	2,6	107,2s	6,4	35,1M	2,7
Variablen	30,9s	1,8	133,6M	10,2	31,2s	1,9	133,6M	10,2
maximal	95,2s	5,7	160,3M	12,2	126,4s	7,5	161,8M	12,4

Tabelle 5.3: Zeit- und Speicherbedarf zur Ausführung unterschiedlich übersetzten Codes

5.3.3 Übersetzungsgeschwindigkeit und Speicherbedarf

Schließlich ist es nicht nur interessant, wie effizient Debugcode ausgeführt werden kann, sondern auch, wie teuer dessen Erzeugung ist, welchen zusätzlichen Aufwand es also für den Compiler bedeutet, Debugcode zu generieren. Um diese Frage zu klären, wurden Zeit und Speicherverbrauch des Compilers zur Übersetzung von Oz-Werkzeugen mit unterschiedlich viel Debuginformation gemessen.

Diesmal kam ein Linux-PC mit Pentium-Pro-Prozessor (200MHz) zum Einsatz. Das Ergebnis zeigt Tabelle 5.4. Die höheren Kosten zur Übersetzung des Browsers können auf die in Abschnitt 5.3.1 diskutierten Programmierstil-Unterschiede zurückgeführt werden.

Annotationen	Browser				Profiler				Explorer			
keine	21,5s	1,0	28,4M	1,0	2,1s	1,0	3,6M	1,0	10,3s	1,0	16,8M	1,0
Steppunkte	25,0s	1,2	33,1M	1,2	2,8s	1,3	4,2M	1,2	12,3s	1,2	19,3M	1,1
Variablen	45,3s	2,1	114,9M	4,0	2,9s	1,4	6,6M	1,8	14,7s	1,4	33,4M	2,0
maximal	64,3s	3,0	159,2M	5,6	3,6s	1,7	8,5M	2,4	19,1s	1,9	43,8M	2,6

Tabelle 5.4: Zeit- und Speicherbedarf zur Generierung von optimiertem Code sowie Debugcode

Die Zahlen belegen, daß der Oz-Debugger einen Vergleich mit gängigen anderen Debuggern nicht scheuen muß. So benötigt beispielsweise auch gcc zur Generierung von Debugcode (Compilerschalter `-g3`) etwa 1,5 bis 2 mal mehr Zeit.[5]

5.4 Ausblick

Der Oz-Debugger in seiner jetzigen Form ist ein voll funktionstüchtiges Werkzeug, das bereits zur Fehlersuche in „Real-World"-Programmen verwendet wird. Allerdings ist die Entwicklung keineswegs abgeschlossen – einige aus anderen Debuggern wohlbekannte Fähigkeiten fehlen noch, andere sich aus dem Kontext von Oz und dessen Weiterentwicklung ergebende Funktionalitätswünsche harren ebenfalls noch ihrer Erfüllung. Im einzelnen geht es u. a. um folgende Punkte:

Endrekursion

Die Praxis zeigt, daß der Verlust von Endrekursion in vielen Fällen inakzeptabel ist. Als Beispiel sei folgende Prozedur angegeben:

```
proc {Loop L}
   case L of H|T then
      {DoSomething H}
      {Loop T}
   end
end
```

Programmfragmente dieser Bauart treten bei ereignisorientierten Programmen auf. Wird keine Endrekursionsoptimierung vorgenommen, so ist die Anzahl der verarbeitbaren Ereignisse von der maximal möglichen Stackhöhe abhängig – indiskutabel, da eigentlich gar kein Stack benötigt wird. Die Rückgewinnung von Endrekursion scheint jedoch nur um den Preis einer vereinfachten, weniger komfortablen Einzelschrittausführung möglich zu sein, indem die Dualität der Steppunkte aufgegeben wird.

Bedingte Haltepunkte

Neben den bereits implementierten *unbedingten* Haltepunkten ist auch die Möglichkeit interessant, *bedingte* Haltepunkte setzen zu können. Als Bedingungen könnten dabei in Frage kommen:

- Ein Ausdruck evaluiert zu „true"

[5] Um dies herauszufinden, wurden ausgewählte (größere) Quelltextdateien des Oz-Emulators einmal mit und einmal ohne den Schalter `-g3` übersetzt. Optimierung war jeweils ausgeschaltet.

- Eine Variable wird gebunden
- Ein anderer Thread wird geweckt

Objekte Zustandsinformation über Objekte könnte besser zugänglich gemacht werden. Denkbar ist etwa die Auflistung aller Features und Attribute in der Variablenanzeige von Ozcar.

Verteilter Code Die Möglichkeiten zum Debuggen von verteiltem Code sind verbesserbar. So sollte über eine netzwerk-transparente Quelltextanzeige nachgedacht werden, ebenso wie Ozcar in seinen Ausgaben differenzieren könnte zwischen Variablen des lokalen Systems einerseits und Proxy-Variablen andererseits [HVS97].

Literaturverzeichnis

[ANS90] *Rationale for American National Standard for Information Systems – Programming Language – C*. `http://www.hensa.ac.uk/ftp/archive/doc/standards/ansi/X3.159-1989/`, 1990.

[CR83] LORI A. CLARKE und DEBRA J. RICHARDSON. *The Application of Error-Sensitive Testing Strategies to Debugging*. In Mark Scott Johnson (Hrsg.), *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Bd. 18(8) von *SIGPLAN Notices*. Aug. 1983.

[Dol96] JULIAN TIMOTHY DOLBY. *Parasight: a debugger for concurrent object-oriented programs*. Diplomarbeit, Universität Illinois, 1996.

[Fel89] STUART I. FELDMAN. *A System for Program Debugging via Reversible Execution (IGOR)*. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, Bd. 24(1) von *SIGPLAN Notices*. Jan. 1989.

[Fla97] CORMAC FLANAGAN. *Effective Static Debugging via Componential Set-Based Analysis*. Dissertation, Rice Universität, Mai 1997.

[Gol94] MICHAEL GOLAN. *A Very High Level Debugging Language (Duel)*. Dissertation, Princeton Universität, Jan. 1994.

[Gra83] WAYNE C. GRAMLICH. *Debugging Methodology (Session Summary)*. In Mark Scott Johnson (Hrsg.), *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Bd. 18(8) von *SIGPLAN Notices*. Aug. 1983.

[Gre94] STEVE GREGORY. *A channel-oriented debugger for concurrent logic programs (COD)*. Fachbereich Informatik, Universität Bristol, 1994.

[H+97] MARTIN HENZ [u. a.]. *The Oz Standard Modules*. Oz Dokumentation, DFKI, Saarbrücken, 1997.

[Hen97] MARTIN HENZ. *Objects in Oz*. Dissertation, Universität des Saarlandes, Mai 1997.

[HVS97] SEIF HARIDI, PETER VAN ROY und GERT SMOLKA. *An Overview of the Design of Distributed Oz*. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation (PASCO '97)*. ACM Press, Maui, Hawai, USA, Juli 1997.

[ISE97] ISE INC. `http://www.eiffel.com`, 1997.

[Kep93] DAVID KEPPEL. *Fast Data Breakpoints*. Techn. Ber. UWCSE 93-04-06, Universität Washington, Fachbereiche Informatik und Engineering, Apr. 1993.

[KSF90] M. KAMKAR, N. SHAHMEHRI und P. FRITZSON. *Bug Localization by Algorithmic Debugging and Program Slicing*. In P. Deransart und J. Małuszyński (Hrsg.), *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming*. 1990.

[LaL94] DANIEL LALIBERTE. *Edebug User Manual – A Source Level Debugger for GNU Emacs Lisp*. `http://www.hypernews.org/~liberte/Edebug/edebug_toc.html`, 1994.

[Lim96] THE HARLEQUIN GROUP LIMITED. *MLWorks User Guide*, Nov. 1996.

[M+93] JULIA MENAPACE [u. a.]. *The "stabs" debug format*. Free Software Foundation, Inc., mit Beisteuerungen von Cygnus Support, 1993.

[Meh98] MICHAEL MEHL. *An Abstract Machine for Oz - Records, Threads, Deep Guards*. Dissertation, Technische Fakultät der Universität des Saarlandes, 1998. In Vorbereitung.

[Mül83] MONIKA A. F. MÜLLERBURG. *The Role of Debugging within Software Engineering Environments*. In Mark Scott Johnson (Hrsg.), *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Bd. 18(8) von *SIGPLAN Notices*. Aug. 1983.

[MS97] MICHAEL MEHL und CHRISTIAN SCHULTE. *Window Programming in DFKI Oz*. Oz Dokumentation, DFKI, Saarbrücken, 1997.

[NBD97] *ddb(4), in-kernel debugger, NetBSD Programmer's Manual*, 1997.

[Nil94] HENRIK NILSSON. *A Declarative Approach to Debugging for Lazy Functional Languages*. Diplomarbeit, Linköping Universität, Schweden, 1994.

[Ous94] JOHN K. OUSTERHOUT. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[PL83] MICHAEL L. POWELL und MARK A. LINTON. *A Database Model of Debugging*. In Mark Scott Johnson (Hrsg.), *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Bd. 18(8) von *SIGPLAN Notices*. Aug. 1983.

[PL89] DOUGLAS Z. PAN und MARK A. LINTON. *Supporting Reverse Execution of Parallel Programs*. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, Bd. 24(1) von *SIGPLAN Notices*. Jan. 1989.

[Pop95] KONSTANTIN POPOV. *An Exercise in Concurrent Object-Oriented Programming: Oz Browser*. In *WOz '95, International Workshop on Oz Programming*. Institut Dalle Molle d'Intelligence Artificielle Perceptive, Martigny, Schweiz, 1995.

[R+96] ERIC RAYMOND [u. a.]. *The new hacker's dictionary*. `http://www.ccil.org/jargon`, 1996.

[Ros96] JONATHAN B. ROSENBERG. *How Debuggers Work. Algorithms, Data Structures, and Architecture*. Wiley Computer Publishing, 1996.

[S+98] GERT SMOLKA [u. a.]. *The Mozart Programming System*. `http://www.ps.uni-sb.de/mozart/`, 1998.

[Sch97] CHRISTIAN SCHULTE. *Oz Explorer: A Visual Constraint Programming Tool*. In Lee Naish (Hrsg.), *Proceedings of the Fourteenth International Conference on Logic Programming*. MIT Press, Leuven, Belgien, Juli 1997.

[Sha83] EHUD Y. SHAPIRO. *Algorithmic Program Debugging*. MIT Press, 1983.

[Smo95] GERT SMOLKA. *The Oz Programming Model*. In Jan van Leeuwen (Hrsg.), *Computer Science Today*, Bd. 1000 von *Lecture Notes in Computer Science*. Springer, Berlin, 1995, S. 324–343.

[Smo98] GERT SMOLKA. *Concurrent Constraint Programming Based on Functional Programming*. In *Proceedings of the European Joint Conferences on Theory and Practice of Software (ETAPS)*, Lecture Notes in Computer Science. Springer, Lissabon, Portugal, 1998.

[SP95] RICHARD M. STALLMAN und ROLAND H. PESCH. *Debugging with GDB*. Free Software Foundation, Inc., 1995.

[TA95] ANDREW TOLMACH und ANDREW W. APPEL. *A Debugger for Standard ML*. *Journal of Functional Programming*, Bd. 5(2), Apr. 1995, S. 155–200.

[TSP83] RON TISCHLER, ROBIN SCHAUFLER und CHARLOTTE PAYNE. *Static Analysis of Programs as an Aid to Debugging*. In Mark Scott Johnson (Hrsg.), *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Bd. 18(8) von *SIGPLAN Notices*. Aug. 1983.

[Zel84] POLLE T. ZELLWEGER. *Interactive source-level debugging*. Techn. Ber. CSL-84-5, Xerox Parc Palo Alto Research Center, 1984.

Index