Ein Debugger für Alice/SEAM

Fortgeschrittenen Praktikum

Jens Regenberg, *jens@ps.uni-sb.de*

Betreuer:
Thorsten Brunklaus, *bruni@ps.uni-sb.de*
Leif Kornstädt, *kornstae@ps.uni-sb.de*

26. Mai 2003

Zusammenfassung

Dieser Bericht beschreibt Entwurf und Implementierung des Debuggers für Alice/SEAM. Der Debugger für Alice/SEAM ermöglicht es dem Benutzer nebenläufige Alice Programme ausdrucksweise auszuführen. Er bietet Funktionalität zum Betrachten der Laufzeitumgebungen einzelner Threads. Der Debugger setzt sich dabei aus zwei Teilen zusammen. Einem sprachabhängigen Teil, der Alice spezifische Daten generiert beziehungsweise verwaltet und einem generischen Teil, der auch von anderen Sprachen auf SEAM verwendet werden kann. Insgesammt ist die Architektur des Debuggers für Alice/SEAM modular, so dass er sehr leicht um einige Features erweitert werden kann.

Inhaltsverzeichnis

1 Einleitung

Bei der Entwicklung von Software treten unumgänglich Fehler auf. Die Suche nach diesen Fehlern und deren Beseitigung wird Debugging genannt. Der Compiler meldet syntaktische Fehler. Semantische Fehler dagegen werden häufig nicht vom Compiler erkannt. Sie resultieren aus Denkfehlern oder Missverständnissen zwischen Compiler und Programmierer. Ein Debugger ist ein Werkzeug, das den Programmierer bei der Suche nach solchen Fehlern unterstützt. Es erlaubt dem Programmierer, sein Programm an wohldefinierten Stellen anzuhalten und es in kleinen Schritten weiter zu verfolgen. Ein Debugger bietet weiterhin die Möglichkeit, die Laufzeitumgebung des Programmes zu betrachten, um so eventuell falsche Berechnungen zu entdecken. Dieser Bericht beschreibt das Design und die Implementierung eines Debuggers für Alice [Pro]. Alice basiert auf der Programmiersprache SML, ist aber um nebenläufige, offene und Constraint Programmierung erweitert.

1.1 Aufgabenstellung

Ziel des Fortgeschrittenen Praktikums war es einen funktionalen Debugger für Alice/SEAM zu schreiben. Ein Debugger unterstützt den Programmierer bei der Suche nach semantischen Fehlern mit zwei Arten von Informationen. Die Laufzeitumgebung gibt Aufschluss über die Werte von Variablen, während der Kontrollfluss den Verlauf eines Programmes anzeigt. In einer funktionalen Sprache wie Alice verfolgt man den Kontrollfluss an dem zentralen Konstrukt, den Ausdrücken. Dazu bietet der Debugger die Möglichkeit Ausdrücke schrittweise einzeln auszuwerten und Argumente und Ergebnis des Ausdrucks zu betrachten.

1.2 Verwandte Arbeiten

Debugger für funktionale Sprachen sind kein neues Konzept. Es gibt bereits mehrere Debugger für andere funktionale Sprachen. Der Debugger für Alice/SEAM übernimmt viele Ideen der Debugger für SML [TA95] und des OZ-Debuggers [Lor99]. Es gibt ebenfalls noch Debugger für CAML, Poly ML und ANU ML. Die letzgenannten wurden nicht veröffentlicht.

Der OZ-Debugger [Lor99] war Vorbild für den Alice Debugger in Bezug auf Konzept der Maschinen - Debugger Kommunikation. Auch die Funktionalität zum Steuern einzelner Threads wurde vom OZ-Debugger übernommen.

Der SML-Debugger [TA95] ist ein ausdrucksbasierter Debugger. Er hat die Möglichkeit deterministisch Schritte rückgängig zu machen und umgekehrt auch zu wiederholen.

Im Kontext von Nebenläufigkeit ist das Widerrufen beziehnugsweise das Wiederherstellen von Berechnungsschritten nicht möglich. Um eine deterministische Wiederholung einzelner Berechnungsschritte zu garantieren, müsste man die Zustände aller Threads speichern, sowie sämtliche Interaktionen zwischen verschiedenen Threads. Dies würde bedeuten, dass dies auch für Threads gelten muss, die nicht vom Debugger kontrolliert werden.

Der Debugger für Alice/SEAM ist ebenso wie der SML-Debugger ein ausdrucksbasierter Debugger. Er braucht wie der SML-Debugger Typen zur eindeutigen Darstellung von Werten. Dieses “Problem” tritt im Kontext von OZ nicht auf. Der Algorithmus zum automatischen Rekonstruktion der Laufzeittypen von Argumenten polymorpher Funktionen von Tolmach und Appel [TA95] ist leider nicht ohne weiteres auf den Debugger für Alice/SEAM übertragbar. Dies wird jedoch genauer in Abschnitt 2.4 besprochen.

2 Fehlersuche in Alice

Die Fehlersuche mit Hilfe eines Debuggers besteht aus zwei Aspekten, der Steuerung des Kontrollflusses und der Untersuchung der Laufzeitumgebungen. Der folgende Abschnitt beschreibt die Anforderungen an den Alice Debugger.

```
val x = .foo(2,3).
```

Abbildung 1: Entry- und Exitpunkt einer Applikation durch "`.`" dargestellt

```
fun foo 0 = 0
  | foo n = n + foo2 (n-1)

and foo2 0 = 0
  | foo2 n = n + foo (n-1)
```

Aufrufstack von `foo 4`:

```
 1 <- foo2 1
   -> foo 2
   -> foo2 3
   -> foo 4
```

Abbildung 2: Aufrufstack nach vier rekursiven Aufrufen.

Ziel der Steuerung des Kontrollflusses ist es fehlerhafte Ausdrücke innerhalb eines Programms zu entdecken. Deklarationen sind bei der Fehlersuche nur dann interessant, wenn sie Ausdrücke enthalten. Der Alice Debugger unterscheidet zwischen den vier folgenden Klassen von Ausdrücken.

- *Applikation*: Anwendung eines Konstruktors oder einer Funktion
- *Konditional*: If-then-else, case, Pattern Matching
- *Raise*: Erzeugen einer Ausnahme
- *Handle*: Abfangen einer Ausnahme

Diese Ausdrücke heißen Step Punkte. Ein Ausdruck kann aus verschiedenen Gründen fehlerhafte Werte berechnen. Beispielsweise kann eine Funktion auf falsche Argumente angewandt werden oder aber die Funktion berechnet einen falschen Wert (vgl. Abb. 2). Der Debugger kann den Kontrollfluss direkt vor und direkt nach dem Steppunkt unterbrechen. Die Stelle direkt vor dem Einstieg in den Steppunkt heißt Entrypunkt, die Stelle direkt nach dem Steppunkt heißt Exitpunkt. Ein Bespiel dafür ist in Abbildung 2 gegeben.

2.1 Aufrufstack

Der Aufrufstack eines Threads besteht aus den Aufrufen, die zu dem momentanen Zustand des Threads geführt haben. Anhand des Aufrufstacks kann der Programmierer verfolgen, welches Konstrukt der Thread gerade bearbeitet. Erreicht ein Thread einen Entrypunkt, so wird dem Aufrufstack des Threads ein neuer Aufruf hinzugefügt. Verlässt er hingegen einen Exitpunkt, so wird der oberste Aufruf vom Aufrufstack entfernt. Abbildung 2.1 zeigt den Aufrufstack von `foo 4`. Entrypunkte werden durch "`->`" dargestellt. "`<-`" bezeichnet einen Exitpunkt. Links neben dem Exitpunkt steht das Ergebnis der Applikation.

2.2 Thread Zustände

Während seiner Ausführung kann ein Thread verschiedene Zustände durchlaufen. Prinzipiell gibt es drei verschiedene Zustände.

- `Runnable` beschreibt, dass ein Thread bereit ist mit seinen Berechnungen fortzufahren.
- Der Thread ist im Zustand `Terminated`, wenn er seine Berechnungen beendet hat.

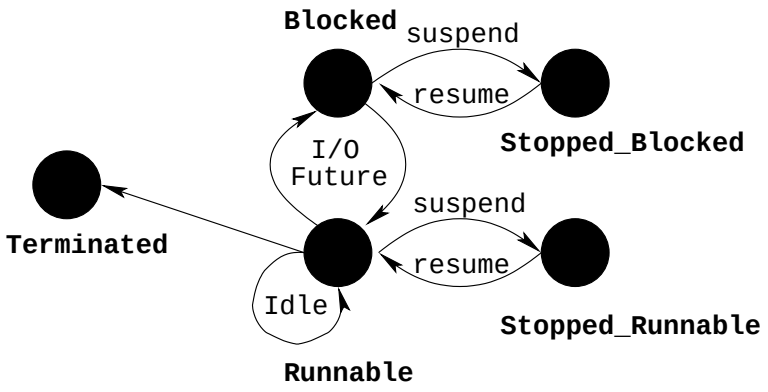

Abbildung 3: Thread Zustände

- Ein Thread ist `Blocked`, wenn er versucht auf eine ungebundene Future zuzugreifen will oder auf I/O wartet.

Der Programmierer kann die Berechnung eines Threads unterbrechen. Damit er nachvollziehen kann, in welchem Zustand die Berechnung des Threads unterbrochen wurde, fügt der Alice Debugger zu diesen Zuständen noch zwei weitere hinzu. Die Zustände `Stopped_Runnable` und `Stopped_Blocked` werden dann angenommen, wenn der Programmierer einen Thread mit *suspend* (siehe auch Abschnitt 2.3) anhält. Sie dienen zur Unterscheidung, welcher Zustand dem Anhalten vorrausging. Das erweiterte Zustandsmodel für Threads ist in Abbildung 3 dargestellt.

2.3 Ausführungskontrolle

Alice ist eine nebenläufige Sprache. Würde der Debugger alle Threads gleichzeitig kontrollieren, könnte der Benutzer bei stark nebenläufigen Programmen sehr schnell die Übersicht verlieren. Aus diesem Grund kontrolliert der Debugger nur bestimmte Threads. Es gibt drei Möglichkeiten, die Kontrolle des Debuggers über einen Thread zu beschreiben.

- Der Thread wird nicht vom Debugger kontrolliert. (*none*)
- Der Thread wird vom Debugger kontrolliert. (*debug*)
- Der Thread war unter Kontrolle des Debuggers, wird jetzt aber nicht mehr von ihm kontrolliert (*detach*)

Ein Thread ist nicht automatisch unter Debuggerkontrolle. Er wird erst vom Debugger kontrolliert, wenn er einen statischen Breakpoint erreicht. Ein statischer Breakpoint kann durch das Built-in `breakpoint` gesetzt werden. Ein Thread gerät außerdem unter Kontrolle des Debuggers, wenn der erzeugenden Thread zum Zeitpunkt der Erzeugung vom Debugger kontrolliert wird. Erreicht ein Thread im Laufe seiner Berechnungen einen Breakpoint, so hält der Debugger die Berechnungen des Threads auf dem Entry Punkt an, der direkt auf den Breakpoint folgt.

Entschließt sich der Benutzer einen Thread nicht weiter zu beobachten, kann er ihn aus der Kontrolle des Debuggers entlassen (*detach*). Ist ein Thread einmal in diesem Zustand, kann er nicht mehr vom Debugger kontrolliert werden.

Ist ein Thread unter Kontrolle des Debuggers, so kann der Programmierer die Ausführung des Threads steuern. Er kann dabei auf die verschiedene Funktionen zugreifen, die die schrittweise Ausführung und Zugriff auf die Laufzeitumgebung des Threads ermöglichen:

- `step` führt die Berechnungen bis zum Erreichen des nächsten Entry oder Exit Punkts fort.
- `next` überspringt alle folgenden Entry und Exit Punkte bis der Thread den Exit Punktes erreicht, der zu dem aktuellen Entry Punkt korrespondierendiert.

```
let
  val pair = fn x => fn y => if true then (x,y) else (x,y)
  val pair1 = pair 1
  val pairt = pair true
  val f = fn (h::t) => pairt h
in
  f [1,2,3]
end
```

Abbildung 4: Nicht triviale Typrekonstruktion

- `continue` überspringt alle Entry und Exit Punkte, bis der Thread seinen Aufrufstack vollständig abgearbeitet hat. Der Debugger unterbricht den Kontrollfluss an dem Exit Punkt des untersten Frames des Aufrufstacks.
- `unleash n` überspringt alle Entry und Exit Punkte bis der Thread die obersten `n` Frames des Aufrufstacks abgearbeitet hat. Mit Hilfe von `unleash` kann man `next` und `continue` darstellen.
- `suspend` stoppt die Berechnungen eines Threads. War der Thread im Zustand `Runnable`, ist er danach im Zustand `Stopped_Runnable`. Analog gilt: War der Thread im Zustand `Blocked` so ist er nach dem Aufruf von `suspend thread` im Zustand `Stopped_Blocked`.
- `resume` nimmt die Berechnungen eines gestoppten Threads wieder auf. Der Zustand des Threads wechselt von `Stopped_Runnable` (`Stopped_Blocked`) zu `Runnable` (`Blocked`)
- `setBreakpoint` setzt einen dynamischen Breakpoint für einen Thread. Ein dynamischer Breakpoint muss auf einen Entry Punkt fallen.
- `deleteBreakpoint` entfernt einen dynamischen Breakpoint.
- `detach` entlässt einen Thread aus der Kontrolle des Debuggers. Für den Programmierer gibt es danach keine Möglichkeit mehr den Thread zu steuern oder die Laufzeitumgebung des Threads zu untersuchen.
- `getRuntimeEnvironment` liefert die Laufzeitumgebung eines Threads.
- `lookup` berechnet den Wert einer Variablen aus der Laufzeitumgebung eines Threads.

2.4 Werte und Typen

Die Laufzeitumgebung beinhaltet die Namen und Werte von lokalen und globalen Bezeichnern. Der Benutzer des Debuggers kann an jedem Entry oder Exit Punkt auf die Laufzeitumgebung zugreifen. Alice verwendet eine typbasierte Darstellung von Werten. Die Annotation der statischen Typen liefert leider keine umfassende Lösung. Ein Problem, dass dadurch nicht gelöst wird, ist die Bestimmung des dynamischen Typs von Argumenten in polymorphen Funktionen. Ein möglicher Ansatz für eine automatische Typrekonstruktion ist in [TA95] vorgestellt. Das folgende Beispiel demonstriert, warum Typen rekonstruiert werden müssen und warum der in [TA95] vorgestellte Algorithmus nicht vollständig auf den Debugger für Alice/SEAM übertragbar ist.

Steht der Debugger auf dem Entry Punkt des Konditionals in `pair` (Abbildung 2.4) und will den dynamischen Typ von `y` berechnen, ist dies nicht ohne weiteren Aufwand möglich. Das Typschema der Funktion $\forall\alpha,\beta.\alpha \to \beta \to \alpha * \beta$ trifft keine Aussage über den dynamischen Typ von `y`. Unter der Annahme, dass sich der dynamische Typ durch einen Prozeduraufruf nicht verändert, kann man im Aufrufstack nach unten steigen, bis man den Aufruf von pair findet. Dort muss man dann den dynamischen Typ des formalen Argumentes bestimmen. Dies muss eventuell rekursiv mit dem gleichen Algorithmus geschehen. Den Aufrufstack zu betrachten ist aber im Allgemeinen nicht ausreichend ([GG92]). Sei die Situation aus Abbildung 2.4 gegeben und der Benutzer

steht immer noch auf dem Entry Punkt des Konditionals in `pair`. Dieses Mal möchte er den dynamischen Typ von `x` wissen. Zu diesem Zeitpunkt ist jedoch die Applikation von `pair` auf `true` schon abgearbeitet und somit nicht mehr auf dem Aufrufstack. Eine Verbindung von `x` zu dem passenden Aufruf ist in dem bisher vorgestellten Konzept des Alice Debuggers nicht möglich. Würde man diese Verbindung herstellen, so würde man verhindern, das Stackframes vom Garbage Collector gelöscht werden.

Aus diesem Grund wurde die automatische Typrekonstruktion vorläufig aus dem Alice Debugger ausgeklammert und durch eine manuelle Typannotation ersetzt. Der Benutzer des Debuggers kann die dynamischen Typen von "polymorphen" Werten manuell festlegen. Bei den meisten Applikationen polymorpher Funktionen, wie zum Beispiel `List.map` oder `List.foldl`, kennt der Benutzer den dynamischen Typ der Argumentliste. Zum Festlegen der Typen gibt es die beiden Funktionen `setType` und `resetType`. Mit diesen Funktionen kann man einen Typ für einen Bezeichner festlegen bzw. auf den polymorphen Typ wiederherstellen. Die Festlegung ist für alle Bezeichner gleichen Namens innerhalb eines Threads gültig.

3 Die Architektur des Alice Debuggers

Der Debugger für Alice/SEAM besteht aus zwei Teilen. Einem Teil, der in die SEAM integriert ist und einem Teil, der den Zustand des Systems speichert und darstellt. Dieser zweite Teil ist der für den Benutzer sichtbare Debugger und wird im folgenden auch als Debugger bezeichnet. Der in SEAM integrierte Teil des Debuggers generiert

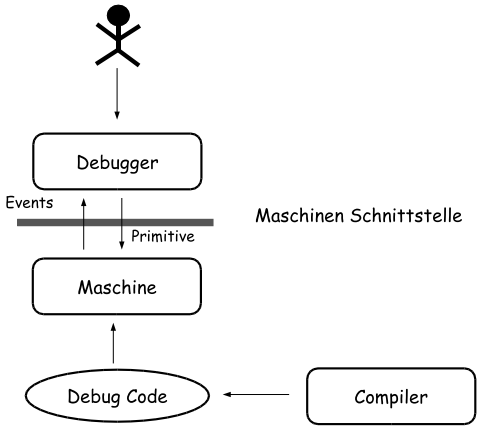

Abbildung 5: Aufbau des Alice Debuggers

Events. Über diese Events kommuniziert er Zustandsänderungen der virtuellen Maschine an den Debugger. Der Debugger kann durch Primitive, die SEAM bereitstellt, die virtuelle Maschine steuern.

3.1 Komponenten des Alice Debuggers

Die Architektur des Alice Debuggers lehnt sich an das *Model-View-Controller* Architektur Muster an (vgl. Abbildung 6). In diesem Architektur Muster repräsentiert das *Model* den Zustand des Systems. Die *View* stellt diesen Zustand für den Benutzer dar. Mit Hilfe der *Controller* kann man den Zustand des *Models* verändern.

Im Kontext des Alice Debuggers gibt es verschiedene *Views*. Diese stellen einen Teil des Debuggerzustands dar. Beispielsweise gibt es eine *View* die den Taskstack eines Threads darstellt, während eine andere *View* die Laufzeitumgebung eines Threads darstellt. Ändert das Model seinen Zustand, aktualisieren die *Views* automatisch ihre Darstellung. *Controller* sind der Benutzer des Debuggers und die Virtuelle Maschine.

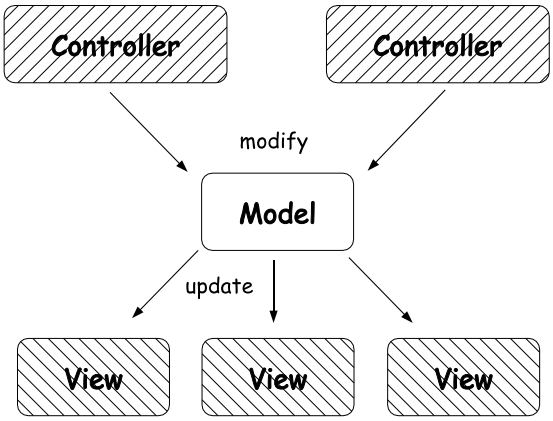

Abbildung 6: *Model -View -Controller* Architektur Muster

Einen Überblick über das angepasste *Model -View -Controller* Muster gibt Abbildung 7

3.1.1 Der Compiler

Der Compiler von SEAM wurde erweitert. Er annotiert nun Entry und Exit Punkte. Dazu wurden zwei neue Instruktionen in den Instruktionssatz von SEAM eingefügt. Ebenso werden zusätzlich Typen und Bezeichner von Funktionen und lokalen Variablen annotiert. Wenn der Code mit Debug Informationen generiert wurde, beinhaltet der Datentyp `annotation` in Abbildung 3.1.1 die Bezeichner und Typen der lokalen Variablen, sowie das Typschema der Funktion. Ist der Code ohne Debug Informationen übersetzt worden, beinhaltet die `annotation` lediglich die Anzahl der lokalen Variablen.

3.1.2 Das Eventmodel

SEAM kommuniziert über Events mit dem Debugger. Diese Events werden auf einen Event Strom geschrieben. Der Debugger hat über die SEAM Schnittstelle Zugriff auf diesen Event Strom. Wie oben bereits erwähnt ist SEAM ein *Controller* des Debuggers. Die Events, die SEAM generiert, werden auch System Events genannt. Der Benutzer des Debuggers ist ein weiterer *controller*. Er steuert den Debugger ebenfalls durch Events. Diese Events heißen Benutzer Events.

Die System Events bezeichnen immer eine bestimmte Änderung des Zustands der Virtuellen Maschine. Sie sind in Abbildung 3.1.2 aufgelistet. Sie lassen sich in zwei Kategorien einteilen. *Alice spezifischen* Events sind `Entry`, `Exit` und `Breakpoint`. Sie werden aus dem vom Compiler erzeugten Code generiert und sind somit abhängig von Alice. Die zweite Kategorie sind *generischen Events*. Diese stellen Zustandsänderungen einzelner Threads dar. Sie sind für alle Sprachen auf SEAM gleich. Ein Debugger für Java/SEAM könnte also ebenfalls diese Events benutzen.

Die Benutzer Events aus Abbildung 3.1.2 beschreiben im Wesentlichen die in Abschnit 2.3 definierten Funktionen zur Steuerung des Debuggers.

3.1.3 Die SEAM Schnittstelle

Die Schnittstelle zwischen Debugger und der Virtuellen Maschine soll möglichst einfach sein, muss aber ermöglichen die in Abschnitt 2 vorgestellte Funktionalität zu implementieren. Die in Abbildung 11 vorgestellte Schnittstelle erfüllt diese Vorraussetzungen. Dabei lässt

- `step` einen Thread bis zum nächsten Entry oder Exit Punkt weiterrechnen.

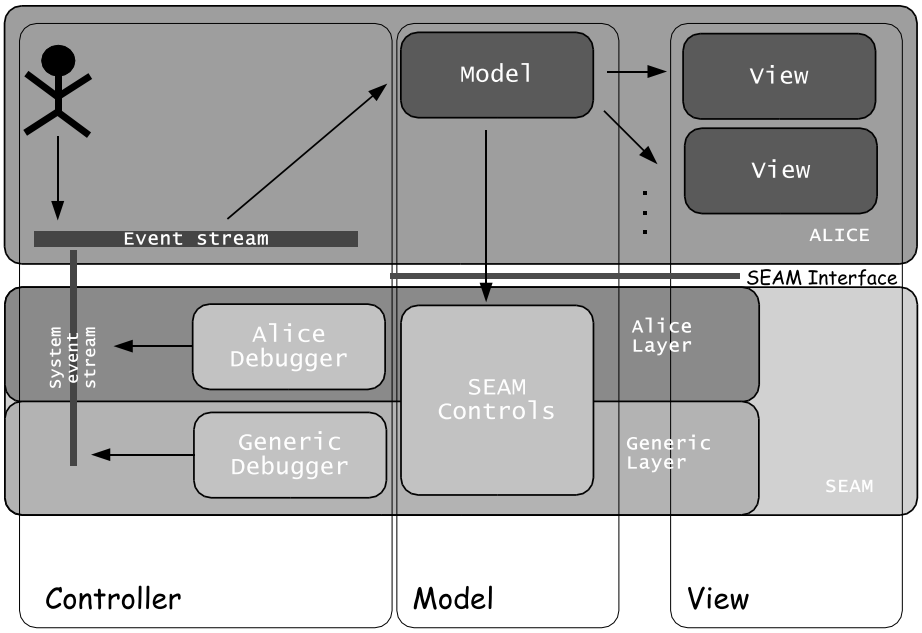

Abbildung 7: Architektur des Alice Debuggers

- `detach` veranlasst, dass für einen Thread keine weiteren Events mehr generiert werden, er also vom Debugger ignoriert wird.
- `breakpoint` denotiert einen statischen Breakpoint. Gleichzeitig wird der aufrufende Thread unter Beobachtung des Debuggers gestellt, sofern das nicht bereits geschehen ist.
- `getRuntimeEnvironment` liefert die Laufzeitumgebung eines Threads zurück.
- `getEventStream` bietet Zugriff auf den Strom der System Events.

3.1.4 Die *View*

Die *View* stellt dem Benutzer den Zustand des Debuggers dar. Da dieser Zustand im allgemeinen sehr komplex sein kann, ist es sinnvoll diesen in unterschiedliche Aspekte zu zerlegen. Es kann dann eine *View* für jeden dieser Aspekte geben. Die Signatur des *View* Moduls ist in Abbildung 12 dargestellt. Die Signatur einer *View* ist in Abbildung 12 angegeben.

- `new` erzeugt eine neue *View* .
- `initialize` initialisiert eine *View* und meldet diese als Beobachter bei einem *Model* an.
- `update` aktualisiert die Anzeige der *View* .

Da das *Model* seine sämtlichen Beobachter kennt, wird die `update` Funktion automatisch aufgerufen, wenn sich der Zustand des *Model* ändert.

3.1.5 Das *Model*

Das *Model* speichert den Zustand des Systems. Es konsumiert den Event Strom des Systems und den des Benutzers. Entsprechend den Events auf den Event Strömen verändert es seinen Zustand und aktualisiert die *Views* oder steuert SEAM über die

```
datatype instr =
    Entry of coord * entry_point * instr
  | Exit  of coord * exit_point * idRef * instr
    ....

and abstractCode =
    ....
  | Function of coord * value option vector *
                annotation *
                idDef args * instr * liveness
    ....

and annotation =
    Debug  of (string * typ) option vector * typ
  | Simple of int
```

Abbildung 8: Erweiterung des Abstrakten Codes

```
datatype event =
   Entry      of thread * pos * step_point
 | Exit       of thread * pos * value * typ
 | Breakpoint of thread * pos * step_point

 | Uncaught   of thread * exn
 | Terminated of thread
 | Blocked    of thread
 | Runnable   of thread
  ...
```

Abbildung 9: System Events des Alice Debuggers.

```
datatype event =
  ...
 | Step             of thread
 | Next             of thread
 | Continue         of thread
 | Suspend          of thread
 | Resume           of thread
 | SetBreakpoint    of thread * pos
 | RemoveBreakpoint of thread * pos
 | Detach           of thread
 | Unleash          of thread * int
 | SetType          of thread * string * typ
 | ResetType        of thread * string
```

Abbildung 10: Benutzer Events

```
sig
   step                  : thread -> unit
   detach                : thread -> unit
   breakpoint            : unit -> unit
   getRuntimeEnvironment : thread -> env

   getEventStream        : unit -> event list
end
```

Abbildung 11: SEAM Schnittstelle

```
signature VIEW =
    sig
        type view
        val new        : unit  -> view
        val initialize : view * model -> unit
        val update     : view -> thread -> unit
    end
```

Abbildung 12: Die Signatur des View Moduls

```
signature MODEL =
    sig
        type model

        val new : unit -> model
        val addView : model *
                      (thread -> unit) -> unit

        val handleEvent : model -> event -> unit

        val getCurrentposition :
             model * thread -> position
        val getCurrentStepPoint :
             model * thread -> stackstp
        val getEnvironment : model * thread -> env
        val getThreadState :
             model * thread -> thread_state
        val getThreadIDs :
             model -> thread list
        val getEventStack :
             model * thread -> stackentry list
        val getType :
             model *
             thread -> string -> typ option
end
```

Abbildung 13: Die Signatur des Model Moduls

SEAM Schnittstelle. In Abbildung 13 ist die Signatur des *Models* beschrieben. Die in der Signatur deklarierten Funkionen haben folgende informale Semantik:

- `new` erzeugt ein neues Model.
- `addView` fügt der Liste von Beobachtern eine neue View zu. Alle Beobachter werden über Änderungen im Model benachrichtigt.
- `handleEvent` dient zum Abarbeiten von Events, die von verschiedenen *Controllern* auf den Event Strom geschrieben wurden.
- `getCurrentPosition` liefert die aktuelle Position eines Threads.
- `getCurrentStepPoint` liefert den aktuellen StepPoint eines Threads.
- `getEnvironment` liefert die Laufzeitumgebung eines Threads.
- `getThreadState` liefert den Zustand eines Threads. Dieser ist entweder `Runnable`, `Blocked` oder `Terminated`.
- `getThreadIDs` liefert eine Liste aller Threads, die zum Zeitpunkt des Aufrufes unter Debuggerkontrolle stehen.
- `getEventStack` liefert den Aufrufstack eines Threads
- `getType` berechnet eine Funktion, die Bezeichnern aus der Laufzeitumgebung eines Threads einen Typ zuordnet.

3.2 Modifikationen am System

SEAM besteht grob gesehen aus zwei verschiedenen Lagen. Die erste ist die generische Lage, in der Daten verwaltet werden, die nicht spezifisch für eine Sprache sind. Darunter fallen beispielsweise Threads und Worker, die nicht behandelte Ausnahmen verabeiten. Die zweite Lage verwaltet sprachspezifische Daten. Hierzu zählen zum Beispiel der Abstrakte Code und Interpreter für diesen Code.

SEAM muss in beiden Lagen modifiziert werden. Sowohl die Laufzeitumgebung als auch die System Events müssen generiert werden. Die generischen Events `Runnable`, `Blocked` und `Terminated` werden in der Klasse Thread generiert. Sie werden in den jeweiligen Methoden erzeugt, in denen der Zustand des Threads geändert wird. Das ebenfalls generische `Uncaught` Event wird in dem BindFutureWorker erzeugt. Dieser Worker ist für das bearbeiten nicht behandelter Ausnahmen verantwortlich.

Die Alice spezifischen Events `Entry`, `Exit` und `Breakpoint` werden im AbstractCodeInterpreter generiert. Dieser generiert ein Entry beziehungsweise Breakpoint Event wenn er eine Entry Instruktion abarbeitet. Das Exit Event wird entsprechend generiert, wenn der AbstractCodeInterpreter eine Exit Instruktion behandelt.

Die Laufzeitumgebung der Threads wird ebenfalls im AbstractCodeInterpreter generiert. Dazu wird jedesmal, wenn eine Closure erzeugt wird, eine Liste mit Bezeichner - Wert - Typ Tupeln der lokalen Variablen gebildet und ein Link auf die erzeugende Closure gelegt.

Threads können durch den Debugger drei neue Zustände erreichen. Im ersten Zustand sind sie noch nicht unter Kontrolle des Debuggers. Der zweite Zustand besagt, dass sie unter Kontrolle des Debuggers sind, während der letzte Zustand besagt, dass sie nicht mehr unter der Beobachtung des Debuggers stehen. Diese drei Zustände werden durch ein neues Flag in der Klasse Thread dargestellt.

4 Implementierung

Die Implementierung des Debuggers für die Alice VM benötigte einige Modifikationen am bestehenden System. Der Compiler wurde von Leif Kornsädt erweitert. Zum Generieren der System Events und Steuerung der Alice VM werden ungefähr 1300 Zeilen Code benötigt. Weiterhin benötigt die Implementierung von *Model*, *Views* und *Controllern* noch einmal 1200 Zeilen Alice Code.

Bootstrap Zeiten	
1.4 GHz, Linux	Zeit [min]
Alice ohne Debugger	37:57
Alice mit Debugger	38:30

Tabelle 1: Bootstrap Zeiten der Alice VM

4.1 Der System Event Strom

Der System Event Strom wird aus zwei verschiedenen Quellen gespeist (siehe auch Abschnitt 3.1.2): Zum einen aus der generischen Lage von SEAM zum anderen aus der Alice Lage. Der Strom wird schließlich durch die SEAM Schnittstelle für den Debugger zugänglich gemacht, ist also eine Alice Datenstruktur. Diese kann aber nicht schon in der generischen Lage generiert werden. In ihr sind keine Informationen darüber vorhanden, wie eine Alice Datenstruktur aussieht.

Beide Lagen generieren aus diesem Grund nicht direkt einen Strom, sondern zuerst einen Container, der sämtliche Informationen des zu generierenden Events enthält. Zu einem solchen Container gehört ein passender Accessor. Mit dessen Hilfe kann man generisch auf die Daten in dem Container zugreifen. Der Container und der Accessor werden zu einem generischen *DebuggerEvent* zusammengefasst. Der für den Debugger sichtbare Eventstrom wird dann durch das SEAM Interface aus den generischen Eventstrom erzeugt.

4.2 Laufzeitumgebung

Die Laufzeitumgebung eines Threads besteht aus Bezeichnernamen und den dazugehörigen Werten und Typen. Zugriff auf die lokalen Bezeichner bietet die Closure. Semi-lokale Variablen sind nicht in der aktuellen Closure erreichbar. Sie sind aber in einer der umgebenden Closures als lokale Variablen vorhanden. Die Idee bei der Erzeugung der Laufzeitumgebung eines Threads ist es, bei der Erzeugung einer Closure einen Link auf die erzeugende Closure hinzuzufügen (Chaining). So hat man von jeder Closure aus auch Zugriff auf die Semi-lokalen Variablen. Der Link auf die erzeugende Closure wird nur dann erzeugt, wenn die dazugehörige Funktion mit Debug Informationen übersetzt wurde. Dies kann man am Abstrakten Code der Funktion anhand der `annotation` aus Abbildung 3.1.1 entscheiden. Ist eine Funktion ohne Debug Informationen übersetzt, so ist sie aus Sicht des Debuggers primitiv. Das heißt insbesondere, dass der Debugger keinen Step Punkte innerhalb dieser Funktion erreichen kann.

4.3 Die SEAM Schnittstelle

Die Schnittstelle zwischen dem Debugger und der Virtuellen Maschine zieht sich durch die generische und die Alice Lage von SEAM. Im generischen Teil der Schnittstelle sind die Funktionen zum Steuern der einzelnen Threads untergebracht. Die Funktionen die Zugriff auf den Eventstrom und die Laufzeitumgebung liefern, liegen dagegen in der Alice Lage.

5 Zahlen

5.1 Bootstrapping

Die Modifikationen an Compiler und Alice VM haben keinen gravierenden Einfluss auf die Leistung. So braucht die Alice VM ohne die Modifikationen des Debuggers im Schnitt 37:57 Minuten für einen vollständigen Bootstrap Vorgang. Die Alice VM mit Debugger Modifikationen benötigt für den gleichen Vorgang (ohne Debug Code) im Schnitt 38:30 Minuten (siehe auch Tabelle 5.1).

Codegröße			
Datei	Code	Debug Code	Wachstum [%]
Client	3372	9857	293
DEBUGGER_CONTROLS-sig	4753	12528	264
DebuggerControls	7108	16171	228
EVENTLISTENER-sig	4528	7052	156
EnvView	7840	33656	430
Eventlistener	6246	17080	273
MODEL-sig	9968	21818	219
Model	14311	57403	401
STREAM-sig	3753	5095	136
SourceView	5988	24670	412
Stream	4122	6334	154
TextView	8337	26319	316
VIEW-sig	3484	7913	227
		Wachstum im Schnitt	**270**

Tabelle 2: Codegrößen der Alice Debugger Komponenten

5.2 Debug Code

Durch die zusätzlichen Annotationen wird der vom Compiler erzeugte Code größer. Der in Alice geschriebene Teil des Debuggers wurde auf einem 1.4 GHz Linux-Rechner im Schnitt um 270% größer wenn er mit Debug Code übersetzt wurde. Die absoluten Zahlen sind in Tabelle 2 aufgeführt. Die für die Erzeugung von Debug Code benötigte Zeit steigt im Schnitt lediglich um 5%.

6 Zusammenfassung

6.1 Diskussion

In diesem Bericht wurde das Design und die Implementierung eines Debuggers für die Alice VM vorgestellt. Die Debug-Kapazitäten des Debuggers umfassen die Grundsprache SML, erweitert um Nebenläufigkeit. Constraint Programmierung wird nicht explizit unterstützt.

Ein wichtiges Entwurfsziel war es, die Architektur des Alice Debuggers so modular wie möglich zu halten. Eine Graphische Benutzeroberfläche soll zu einem späteren Zeitpunkt hinzugefügt werden. Das verwendete *Model -View -Controller* Architektur Pattern eignet sich dafür bestens. Die bisher implementierte Benutzerschnittstelle ist textuell orientiert. Für einfache Anwendungen ist dies noch ausreichend. Aber bei nebenläufigen Programmen erreicht diese Schnittstelle sehr schnell einen Punkt, an dem der Benutzer die Übersicht verliert.

Ein weiteres Ziel war es, die Schnittstelle des Debuggers zu Alice VM möglichst einfach zu halten. So beschränken sich die Funktionen zum Steuern der Alice VM auf ein Minimum. Komplexere Steuer-Funktionen werden durch die Einfachen emuliert (z.B.: next: Das Model ignoriert so lange Entry Events, bis es das passende Exit Event erhält).

Die Bestimmung der Laufzeittypen von Argumenten polymorpher Funktionen hat sich während des Praktikums als sehr komplex herausgestellt. Es ist nicht klar, ob es ausreicht das Typschema einer Funktion zu erkennen, um daraus die Typen der Argument einer Applikation zu berechnen. Ein möglicher Lösungsansatz ist hier auch die Typapplikation explizit durchzuführen, wie es in System F gemacht wird.

6.2 Zukünftige Arbeiten

Es gibt noch mehrere Möglichkeiten den Alice Debugger zu erweitern. Einige davon werden in den folgenden Abschnitten erläutert.

6.2.1 Automatische Typrekonstruktion

Die manuelle Typannotation ist aufwendig und umständlich. Zudem ist sie extrem unsicher. Ein falscher Typ kann zu einem Absturz des Systems fhren. Ein Algorithmus zur automatischen Typrekonstruktion, der partiell für den Alice Debugger funktioniert, wird in [TA95] vorgestellt. Es wäre interessant zu erfahren, ob der Algorithmus zu einem vollständigen Algorithmus für den Alice Debugger erweitert werden kann.

6.2.2 Graphische Benutzeroberfläche

Zu Beginn dieses Fortgeschrittenen Praktikums gab es in Alice/SEAM noch keine Anbindung an eine Grafikbibliothek. In der Zwischenzeit ist jedoch eine GTK - Schnittstelle hinzugekommen. Eine graphische Benutzeroberfläche ist unbedingt nötig, wenn man stark nebenläufige Programme debuggen möchte.

Literatur

[GG92] Benjamin Goldberg and Michael Gloger. Polymorphic type reconstruction for garbage collection without tags. In *LISP and Functional Programming*, pages 53–65, 1992.

[Lor99] Benjamin Lorenz. Ein Debugger für Oz. Master's thesis, Universität des Saarlandes, April 1999.

[Pro] Programming System Labs, Fachbereich Informatik, Universität des Saarlandes, http://www.ps.uni-sb.de/alice/. *Alice Homepage*.

[TA95] Andrew P. Tolmach and Andrew W. Appel. A debugger for standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.