Dynamische Deadlock-Suche in nebenläufigen funktionalen Programmen

Frank Huch
Institut für Informatik
Christian-Albrechts-Universität zu Kiel
fhu@informatik.uni-kiel.de

Zusammenfassung

Wir präsentieren eine Debugger für Concurrent Haskell. Seine wesentlichen Features sind: Visualisierung von Threads und Kommunikation, interaktives Scheduling, automatische Deadlock-Suche im Hintergrund, Leiten des Benutzers in gefundenen Deadlock, Optimierung der Suche durch Partial Order Reduktion. Die Implementierung konnte ohne Modifikation der Concurrent Haskell Laufzeitumgebung vorgenommen werden.

1 Concurrent Haskell Debugger

Nebenläufige Systemen können neue, bei sequentiellen Systemen nicht auftretende Fehler enthalten, wie z.B. Deadlocks, Lifelocks oder die Nichteinhaltung eines wechselseitigen Ausschluss. Im Gegensatz zu klassischen Fehlern sequentieller Programme hängen diese Fehler vom Prozess-Scheduling ab und treten möglicherweise nur in bestimmten Systemläufen auf.

Zum Finden von Fehlern werden in der Regel Debugger verwendet, welche den Programmablauf visualisieren und so ein besseres Verständnis der Fehlersituation ermöglichen sollen. Wir haben einen Debugger für Concurrent Haskell (eine nebenläufige Erweiterung der Programmiersprache Haskell) entwickelt. Dieser ermöglicht es nebenläufige Haskell-Programme auszuführen, die Kommunikation zu visualisieren und manuell unterschiedliche Prozess-Schedules auf Fehler zu überprüfen. Außerdem können, zum besseren Verständnis des aktuellen Systemzustands, die zuletzt ausgeführten Kommunikationsaktionen der Threads im Sourcecodes angezeigt werden.

Die Implementierung des gesamten Debuggers war ohne Modifikation des Haskell Laufzeitsystems, rein in Form einer Haskell Bibliothek möglich. Hierbei haben wir ein Modul `ConcurrentDebug` (mit identischem Interface wie das Concurrent Haskell Modul `Concurrent`) entwickelt, welches Debug-Varianten sämtlicher Concurrent Haskell Funktionen zur Verfügung stellt. Importiert ein Benutzer anstelle des Moduls `Concurrent` das Modul `ConcurrentDebug`, so wird, ohne weitere Änderungen an dem Programm, während der Ausführung zusätzlich eine Gui (siehe Abbildung 1) gestartet, die den aktuellen Zustand des Concurrent Haskell Programms (Threads und Kommunikationsabstraktionen) darstellt. Über zusätzliche Bestätigungsnachrichten von der Gui können die Systemthreads auch blockiert werden und interaktiv durch den Benutzer wieder frei gegeben werden. Somit kann der Benutzer das Scheduling beliebig steuern.

2 Deadlock-Suche

Trotz der Möglichkeit das Scheduling zu beeinflussen, ist es häufig problematisch Fehler, wie Deadlocks, zu finden, da der entsprechende Pfad durch den Benutzer nicht gefunden wird. Als Lösung haben wir den Debugger so erweitert, dass er im Hintergrund selbständig nach Deadlocks sucht und den Benutzer ggf. in einen gefundenen Deadlock führt. Während der Suche werden alle möglichen Thread-Schedulings ausgeführt und die hierbei entstehenden Systemzustände analysiert. Da der Suchraum im allgemeinen sehr groß (meist sogar unendlich) ist, wird die Suche durch eine maximale Zahl von zu besuchenden Zuständen begrenzt. Als Suchstrategie verwenden wir Iterative-Deepening.

Für das Backtracking bei der beim Iterative-Deepening durchgeführten Tiefensuche wird Haskells Sequenz-Operator `(>>=)` (und die `IO`-Monade) als Datenkonstruktor umdefiniert. Die entstehende Datenstruktur wird dann interpretiert, wodurch wir einen eigenen Scheduler in unserer Bibliothek implementieren können. Alte Systemzustände können gespeichert und nach der Suche wieder verwendet werden. Kommunikationsaktionen zum Senden und Empfangen von Nachrichten können durch entsprechende Umkehraktionen rückgängig gemacht werden. Ausgaben des Programms werden während der Suche ignoriert, da sie sonst mehrfach ausgeführt würden. Erwartet das Programm Eingaben, so wird die Deadlock-Suche erfolglos abgebrochen. Das Programm wird in der Regel in Abhängigkeit einer Eingabe verzweigt und Eingaben des Benutzers während der Suche sind nicht sinnvoll.

Als Optimierung der Deadlock-Suche setzen wir zusätzlich Techniken aus der Verifikation nebenläufi-

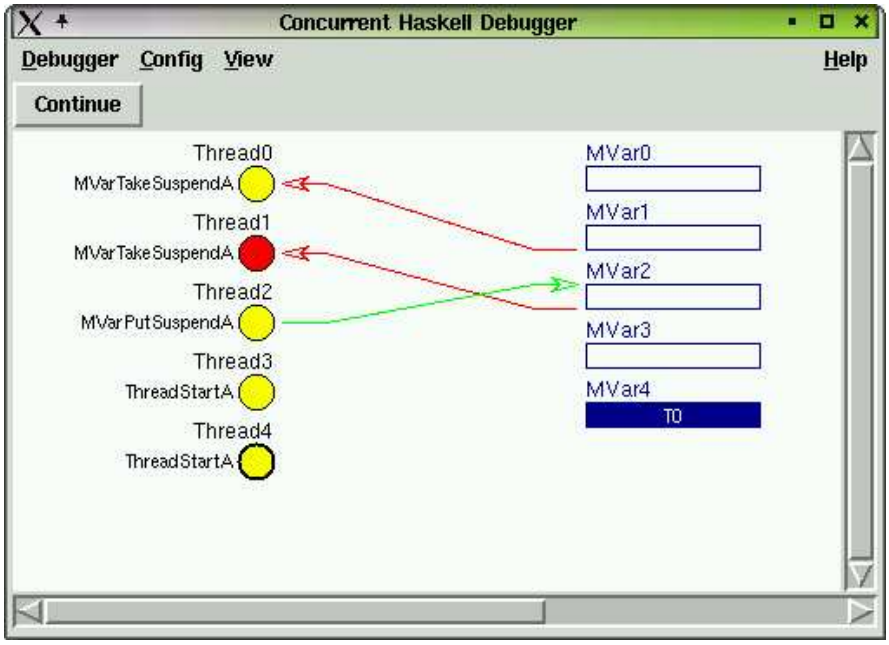

Abbildung 1: Der Concurrent Haskell Debugger für 5 dinierende Philosophen

ger Systeme ein: führen zwei Threads unabhängige Aktionen aus, so ist es nicht nötig diese Aktionen in allen möglichen Schedulings zu untersuchen. Nur für abhängige Aktionen (z.B. Zugriff auf die gleiche Kommunikationsvariable) müssen alle möglichen Schedulings untersucht werden. Partial Order Reduction ermöglicht es hier den Suchraum stark zu verkleinern und eine Suche sehr viel effizienter und bis zu einer größeren Tiefe durchzuführen.

Abbildung 1 zeigt den Einsatz unseres Debuggers für fünf dinierende Philosophen (Threads 0 bis 4). Die Stäbe sind als veränderbare Variablen (`MVar0` bis `MVar4`) implementiert, wobei im abgebildeten Zweitpunkt nur die `MVar4` gefüllt ist, also nur Stab 4 auf dem Tisch liegt. Die Pfeile verdeutlichen, welche Aktionen die Threads als nächstes ausführen würden (so will z.B. Thread2 den Stab 2 zurück auf den Tisch legen). Das System hat bereits einen Pfad in den Deadlock gefunden. Der Pfad beginnt mit der nächsten Aktion des Threads 4, was durch den dickeren Kreis um den Thread angezeigt wird. Nach Auswahl dieses Threads wird der Benutzer schrittweise in den vorhandenen Deadlock geführt und kann hierbei Schwachstellen/Fehler seines System nachvollziehen.

In der Praxis können mit unserem System vorhandene Deadlocks sehr häufig gefunden werden. Als Beispiel findet unser System in unterschiedlichen Implementierungen der dinierenden Philosophen mit bis zu 15 Philosophen bereits beim Systemstart einen Pfad in den vorhandenen Deadlock. Bei größerer Philosophenzahl wird der Deadlock zwar nicht beim Systemstart, aber während der Ausführung (nach dem Start aller Threads) gefunden.

3 Ausblick

In weiteren Arbeiten soll der Debugger weiterentwickelt werden. Hierbei sind insbesondere weiter Sichten auf das in Concurrent Haskell implementierte System angedacht, welche insbesondere ein besseres Verständnis des Nachrichten-/Datenflusses zwischen den Threads ermöglichen sollen.

Als weitere Erweiterung soll das System nicht nur Deadlocks automatisch suchen, sondern auch benutzerspezifizierbare Eigenschaften, wie Livelocks oder Race-Konditions. Da diese Eigeschaften nicht automatisch erkannt werden können, soll der Benutzer sie in einer Logik (z.B. Linearzeit Temporal Logik,LTL) spezifizieren. Während der Suche werden die spezifizierten LTL-Formeln dann analysiert und der Benutzer ggf. in den Systemzustand, welcher die Eigenschaft verletzt, geführt.

Weitere Informationen zum Concurrent Haskell Debugger finden sich in [1].

Literatur

[1] J. Christiansen and F. Huch. Searching for deadlocks while debugging Concurrent Haskell programs. *ACM SIGPLAN Notices*, 39(9):28–39, September 2004. Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming (ICFP '04).