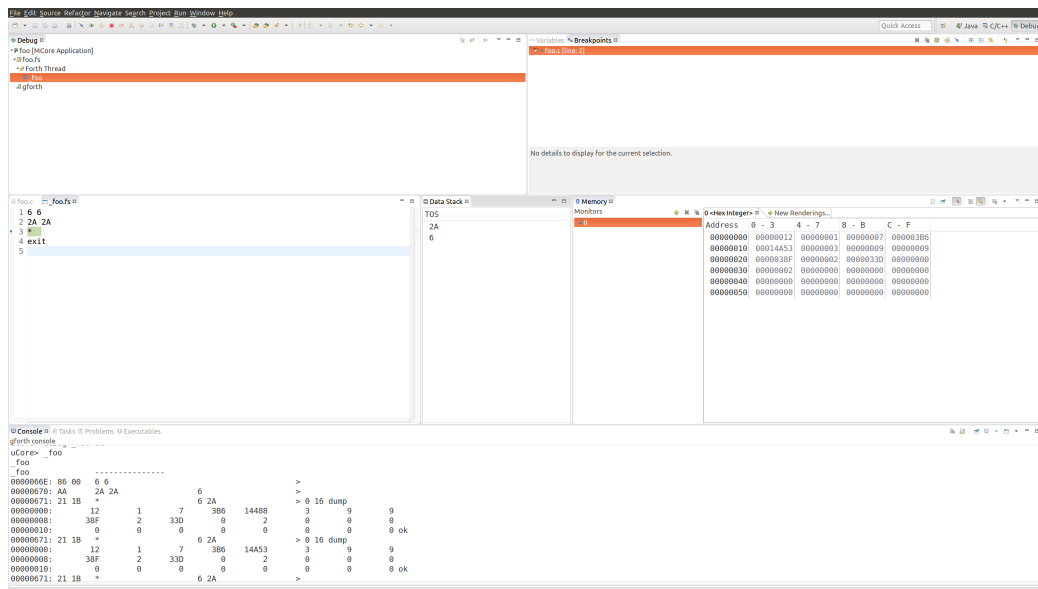


Bachelor-Thesis

Eclipse Entwicklungsumgebung für MicroCore

Benjamin Neukom

August 2015



Betreuer: Carlo Nicola

Inhaltsverzeichnis

| | |
|------------------------------------------------------------|-----------|
| 1. Einleitung | 5 |
| 2. Die Eclipse Platform | 6 |
| 2.1. Eclipse als Platform | 6 |
| 2.2. Plugins | 6 |
| 2.3. Extension Points | 6 |
| 2.4. Eclipse basierte MCore Entwicklungsumgebung | 7 |
| 2.4.1. JDT | 7 |
| 2.4.2. XText | 7 |
| 2.4.3. Dynamic Language Toolkit Framework | 8 |
| 2.4.4. Eclipse C/C++ Development Tools | 8 |
| 2.4.5. Verwendung für MCore Eclipse | 8 |
| 3. Compiler Integration | 9 |
| 3.1. Integration im Eclipse CDT | 9 |
| 3.1.1. Configuration | 9 |
| 3.1.2. Tool | 9 |
| 3.1.3. Toolchain | 9 |
| 3.1.4. CDT Builder | 10 |
| 3.2. Project Nature | 10 |
| 4. Programm Launch | 11 |
| 4.1. Launch | 11 |
| 4.1.1. Run | 12 |
| 4.1.2. Debug | 12 |
| 5. Forth Kommunikation | 13 |
| 5.1. Prozess Kommunikation | 13 |
| 5.1.1. GDB/MI-Commands | 13 |
| 5.1.2. Direkte Kommunikation mit dem Prozess | 13 |
| 5.2. Kommunikation | 14 |
| 5.3. API Design | 14 |
| 5.4. Implementierung | 14 |
| 5.4.1. Klassenbeschreibung | 16 |
| 6. Debugger | 18 |
| 6.1. Breakpoints | 18 |
| 6.1.1. Per Konsole | 18 |

| | |
|----------------------------------------------------------------------|-----------|
| 6.1.2. Im Source Code | 18 |
| 6.2. Konsolen basierter Debugger | 19 |
| 6.3. Forth Debugger | 19 |
| 6.3.1. CDT oder JDT Debugging Mechanismen | 19 |
| 6.3.2. Debugger Aktionen | 19 |
| 6.3.3. Stack View | 22 |
| 6.3.4. Memory View | 22 |
| 6.4. C-Debugger | 23 |
| 7. Entwicklungsumgebungs Einstellungen | 24 |
| 7.1. Umbilical | 24 |
| 7.2. Loader | 24 |
| 8. Optimierungen | 25 |
| 8.1. Peephole Optimierung | 25 |
| 8.1.1. Beispiele | 26 |
| 8.1.2. Optimierungen | 27 |
| 8.1.3. Automatische Generierung von Peephole Optimierungen | 28 |
| 8.1.4. Constant Propagation | 28 |
| 8.1.5. Resultate und Tests | 28 |
| 8.1.6. Mögliche Erweiterungen | 29 |
| A. Literaturverzeichnis | 30 |
| B. Abbildungsverzeichnis | 31 |
| C. Tabellenverzeichnis | 32 |
| D. Ehrlichkeitserklärung | 33 |

Im ersten Teil dieser Arbeit wird beschrieben, wie eine auf Eclipse basierende Entwicklungsumgebung implementiert werden kann. Es wird gezeigt, was Eclipse für Features für Entwicklungsumgebungen bereitstellt und wie diese genutzt werden können um eine Entwicklungsumgebung darauf aufzubauen. Im zweiten Teil werden verschiedene Peephole Optimierungen für den Compiler untersucht und mit der aktuellen Peephole Implementation des Forth-Cross Compilers verglichen.

1. Einleitung

MicroCore ist eine CPU mit Harvard-Architektur, die eine Teilmenge der Forth-Sprache als Maschinensprache benutzt. MicroCore wurde in Zusammenarbeit mit der Firma Send GmbH in Hamburg entwickelt. Für die Version 1.71 wurde ein C-Compiler mit scc (Stack-C-Compiler) implementiert. Ziel dieser Arbeit ist, eine auf Eclipse basierte Entwicklungsumgebung zu entwickeln, welche den scc als Compiler verwendet. Es sollte möglich sein, den ganzen Prozess, vom Quellcode bis zur Kompilierung und dem Herunterladen auf das HW-Target in der Entwicklungsumgebung durchführen kann. Im ersten Kapitel wird beschrieben, was Eclipse als Plattform für Möglichkeiten bietet. Danach wird erklärt wie der Compiler integriert wurde. Im dritten Kapitel wird beschrieben, wie das kompilierte uForth Programm aus der Entwicklungsumgebung gestartet werden kann und was dafür implementiert wurde. Im Kapitel "Forth Kommunikation" wird auf die Kommunikation mit dem Forth Prozess eingegangen, wie diese designt und implementiert wurde. Im nächsten Kapitel wird gezeigt, auf welche Arten der Debugger in die Entwicklungsumgebung eingebunden wurde und wie diese noch erweitert werden könnten. Im nächsten Kapitel "Entwicklungsumgebungs Einstellungen" werden die verschiedenen Einstellungen, welche für die Entwicklungsumgebung notwendig sind, beschrieben. Im Kapitel "Optimierungen" werden dann nach einige Peephole Optimierungen für den Compiler vorgestellt und mit der aktuellen Peephole Optimierung des Forth-Cross Compilers verglichen.

2. Die Eclipse Plattform

In diesem Kapitel wird gezeigt, was Eclipse für Möglichkeiten bietet, um eine moderne Entwicklungsumgebung zu implementieren. Diese verschiedenen Möglichkeiten werden verglichen und die Vor- und Nachteile aufgezeigt. Es werden auch die wichtigsten Eclipse Features, welche für das Entwickeln von Eclipse Rich Client Platform (RCP) Applikationen benötigt werden, beschrieben.

2.1. Eclipse als Plattform

Eclipse RCP bietet eine Basis um beliebige (nicht zwingendermassen Entwicklungsumgebungen), Betriebssystem unabhängige, Applikationen zu entwickeln. Es bietet Mechanismen, wie Plugins und Extension Points, um modulares programmieren zu unterstützen und vereinfachen. Auch bietet das Framework Features, wie das Konzept von Views, Editoren und vielen mehr, welche häufig in Applikationen gebraucht werden.

2.2. Plugins

Eclipse Applikationen nutzen eine auf der OSGi Spezifikation basierten Runtime. Eine Komponente in dieser Runtime ist ein Plugin. Eine Eclipse RCP Applikation besteht aus einer Ansammlung dieser Plugins. [1] Plugins haben einen Lebenszyklus und können zur Laufzeit geladen oder entladen werden. Ein Eclipse Plugin kann Extension Points anderer Plugins ansteuern und eigene Extension Points oder APIs anbieten. Diese werden im Plugin.xml File, welches jedes Plugin besitzt definiert.

2.3. Extension Points

Um Plugins erweitern zu können, bietet Eclipse das Konzept von Extension Points an. Über Extension Points können Plugins eine bestimmte Funktionalität anbieten, welches von anderen Plugins deklarativ angesteuert werden kann.

Es existiert zum Beispiel ein Plugin, welches einen Extension Point für Views definiert. Andere Plugins können über diesen Extension Point, deklarativ in einem XML File, eine neue View erstellen und diese konfigurieren. [2]

In folgendem Screenshot ist zu sehen, wie über einen Extension Point eine neue Eclipse View erstellt werden kann.

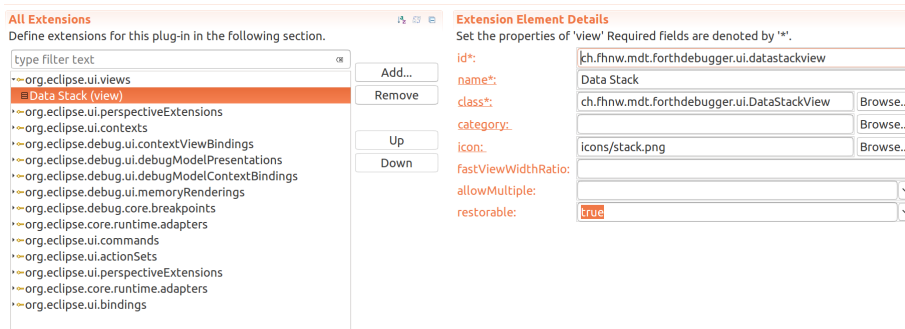


Abbildung 2.1.: Ein Plugin, welches einen Extension Point anbietet. Andere Plugins können diese deklarativ in einem XML File ansteuern.

Es ist auch möglich eigene Extension Points zu definieren, falls ein Plugin für andere Entwickler offen stehen soll für Erweiterungen.

2.4. Eclipse basierte MCore Entwicklungsumgebung

Eclipse als Grundlage für eine Entwicklungsumgebung zu verwenden eignet sich besonders gut, da Eclipse schon einiges an Funktionalität für eine Entwicklungsumgebung zur Verfügung stellt und schon viele Entwicklungsumgebungen (PHP, C/C++, Python und D unter anderem) basierend auf Eclipse entwickelt wurden. Auch existieren einige Tools, auf welche ich noch genauer eingehen werde, wie Xtext und DLTk, welche das entwickeln einer Entwicklungsumgebung weiter vereinfachen. In folgenden Kapiteln wird kurz auf die Möglichkeiten, welche von Eclipse bereitgestellten Technologien verwendet werden könnten um die Entwicklungsumgebung zu implementieren.

2.4.1. JDT

Eine Möglichkeit die MCore Entwicklungsumgebung zu implementieren wäre, die Standard Features von dem Eclipse Java Development Tools (JDT) zu verwenden. Dies wäre sehr Aufwendig, da alle Features für C, wie Syntax Highlighting, Code Completion oder eine Outline neu implementiert werden müssten.

2.4.2. XText

XText ist ein Framework, welches es erleichtert, eine auf Eclipse basierte Entwicklungsumgebungen zu programmieren. Es ermöglicht auf schnelle Weise ein Grundgerüst einer Entwicklungsumgebung mit Features wie:

- Ein Editor mit Syntax Coloring

- Code Completion
- Compiler Integration
- Ein Java-basierter Debugger
- Eine Outline
- Indexing

zu generieren. [3] Es muss lediglich eine ANTLR [4] Grammatik für die Sprache definiert werden. Der grosse Nachteil ist, dass C, inklusive Preprozessor, zu parsen sehr kompliziert ist und eine Entwicklungsumgebung somit auch mit Xtext nicht einfach zu implementieren ist.

2.4.3. Dynamic Language Toolkit Framework

Das Dynamic Language Toolkit (DLTK) ist ein weiteres Framework, welches ein Grundgerüst für eine Entwicklungsumgebungen generieren kann. Ursprünglicherweise war das Framework nur für dynamische Sprachen geeignet, es kann aber auch für statische Sprachen verwendet werden. Die D Entwicklungsumgebung wurde mit dem DLTK realisiert [5]. Es bestehen aber wieder dieselben Nachteile wie bei Xtext. Da C schwierig zu parsen ist, müsste trotz dem Framework noch viel selbst implementiert werden. D verwendet keinen Preprozessor, deshalb konnte für diese Entwicklungsumgebung das DLTK Framework verwendet werden.

2.4.4. Eclipse C/C++ Development Tools

Das Eclipse C/C++ Development Tools (CDT), ist eine Eclipse Distribution mit Unterstützung für C und C++. Das CDT bietet viele Features, welche schon vom JDT bekannt sind für C an und stellt Extension Points zur Verfügung um diese für eine eigene Entwicklungsumgebung zu gebrauchen. So kann mit relativ wenig Aufwand einen neuen Compiler in die Entwicklungsumgebung eingebunden werden. Dies wurde auch schon mehrfach gebraucht, um verschiedene C/C++ Compiler im Eclipse CDT zu integrieren. Auf die Einbindung des Compilers wird im Kapitel 3 genauer eingegangen.

2.4.5. Verwendung für MCore Eclipse

Ich habe mich dazu entschieden, das Eclipse CDT als Target Platform zu wählen. Somit können alle Features, welche das Eclipse CDT zur Verfügung stellt, gebraucht werden. Frameworks, welche ein Grundgerüst einer Entwicklungsumgebung generieren, funktionieren nur Teilweise für C und sind somit keine guten Alternativen.

3. Compiler Integration

In diesem Kapitel wird beschrieben, wie der Compiler in die Entwicklungsumgebung eingebunden wurde um ein C-File nach Forth zu übersetzen. Und welche Möglichkeiten mit dem CDT für das Editieren von C-Files zur Verfügung stehen.

3.1. Integration im Eclipse CDT

Das CDT stellt Extension Points zur Verfügung, welche gebraucht werden können um einen Compiler zu integrieren. Diese Extension Points werden in den nächsten Kapitel erläutert und es wird erklärt wie der LCC dadurch in die Entwicklungsumgebung eingebunden wird.

3.1.1. Configuration

Eine Konfiguration wird gebraucht um verschiedene Standard Tools und Optionen bereit zu stellen, um ein Projekt auf eine gewisse Weise zu kompilieren. Normalerweise existieren für ein Projekt zwei Konfigurationen. Eine Debug- und eine Releasekonfiguration.

Verwendung

Es gibt nur eine Konfiguration für den Release Build. Debug spezifische Files werden von dem Launch generiert, da der Compiler damit nichts zu tun hat.

3.1.2. Tool

Definiert ein Tool, wie zum Beispiel ein Compiler oder Linker, welches verwendet wird im Buildprozess.

Verwendung

Es wird nur ein Tool verwendet, welches den LCC Compiler aufruft und somit das Forth File generiert. Für dieses Tool wurde noch eine Option (`-S-Q`) definiert, welche den Peephole Optimizer deaktiviert.

3.1.3. Toolchain

Eine Liste von Tools welche gebraucht werden um den Output des Projekts zu generieren.

Verwendung

Die Toolchain beinhaltet nur das vorhin beschriebene Tool um den LCC Compiler aufzurufen.

3.1.4. CDT Builder

TODO benennt die Forth Files um (WHY?)

3.2. Project Nature

TODO findet Errors im Path (falls Eclipse nicht richtig aufgesetzt wurde)

4. Programm Launch

In diesem Kapitel wird beschrieben, wie das kompilierte uForth Programm aus der IDE gestartet werden kann und was dafür implementiert wurde.

4.1. Launch

Im Eclipse werden zwischen zwei Launch Möglichkeiten unterschieden. Der unterschied zwischen beiden wird in den nächsten Kapiteln erklärt.

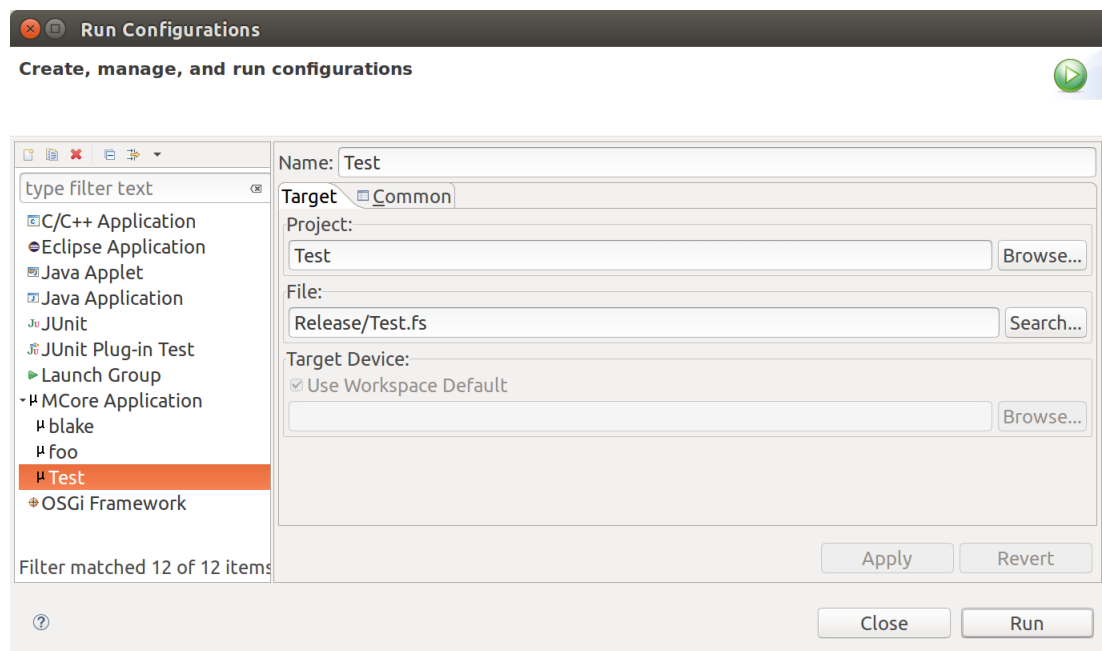


Abbildung 4.1.: Der Launch Configuration Dialog. Im Dialog kann eine neue MCore Launch Konfiguration erstellt werden. Um eine gültige Konfiguration zu erstellen muss das Projekt und das Forth File, welches gestartet werden soll angegeben werden.

4.1.1. Run

Mit der Run Konfiguration wird zuerst der Loader in den Forth Workspace kopiert (dieser muss in einer Umgebungsvariable definiert sein). Danach wird der Forth Prozess gestartet und der Umbilical Port mittels `Umbilical: /dev/ttyUSBX` gesetzt. Der Umbilical Port und der Loader müssen in der MCore Settings Page gesetzt werden (Siehe Kapitel 7). Falls der Umbilical Port ungültig ist, wird beim starten des Programmes eine Fehlermeldung gezeigt.

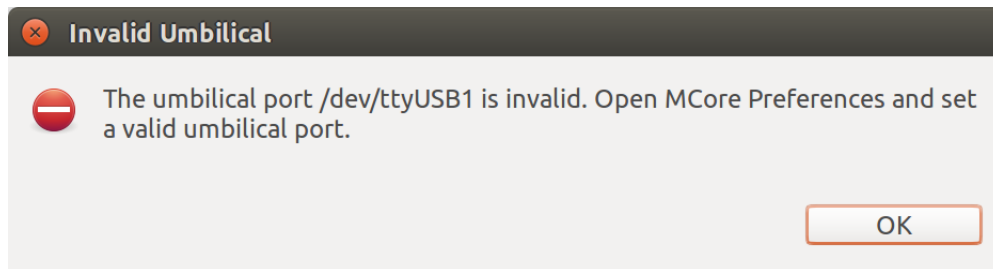


Abbildung 4.2.: Dialog, welcher angezeigt wird, wenn ein Program gestartet wird und der Umbilical Port nicht gültig ist. Der Port kann in der MCore Preference Page geändert werden.

4.1.2. Debug

In der Debug Konfiguration werden zusätzlich noch alle Funktionen des gestarteten Forth Files disassembled. Dies wird gemacht, dass der Debugger die Funktionen in einem File anzeigen kann. Es kann nicht der vom Compiler generierte Code genommen werden, da der Forth Cross-Compiler noch Optimierungen am Code vornimmt, deshalb müssen die Funktionen zur Laufzeit noch disassembld werden.

5. Forth Kommunikation

In diesem Kapitel wird beschrieben, wie die Entwicklungsumgebung mit dem Forth Prozess kommuniziert. Die Kommunikation mit dem Forth Prozess ist von zentraler Bedeutung, da viel der Funktionalität der Entwicklungsumgebung davon abhängt, dass die Kommunikation stabil läuft. Es wird gezeigt, wie die Kommunikation designt, implementiert und getestet wurde.

5.1. Prozess Kommunikation

Um mit dem Prozess zu kommunizieren, gibt es grundsätzlich zwei Möglichkeiten. Die erste ist ein Machine Interface zu implementieren, oder direkt mit dem Prozess kommunizieren. In folgenden Kapiteln werden die beiden Möglichkeiten kurz beschrieben.

5.1.1. GDB/MI-Commands

Eine Möglichkeit mit dem Prozess zu kommunizieren wäre, ein MachineInterface (MI) wie es für den GDB implementiert wurde, zu gebrauchen. GDB/MI ist ein Linien basiertes Maschinen orientiertes Text Interface zu dem GDB. Es wurde dazu entwickelt, damit der GDB als Debugger in ein grössers System einfacher einbindbar ist. [6] Eine MI ähnliches Interface wäre mit grossem Aufwand verbunden, da das Interface zuerst definiert werden müsste. Dafür könnte aber viel des CDT Debugging Mechanismus verwendet werden, da der CDT Debugger auch auf dem GDB/MI basiert. Auch ist der CDT Debugger sehr kompliziert [7] und es wäre ein grosser Einarbeitungsaufwand notwendig um den Forth Debugger darauf basieren zu können. [7]

5.1.2. Direkte Kommunikation mit dem Prozess

Eine weitere Möglichkeit wäre, direkt die Befehle an den Prozess senden und auf Antworten warten. Dafür müsste einige Klassen implementiert werden, um das Kommunizieren zu vereinheitlichen und vereinfachen.

Probleme bei der Kommunikation

Bei der Kommunikation mit dem Prozess können einige Probleme Auftreten. Es kann sein, dass der Prozess plötzlich keine Antworten mehr gibt. Auch weiss man nicht, wie lange es dauern wird, bis der Prozess Antwort gibt. Diese Probleme müssen mit den im vorigen genannten Klassen gelöst werden.

5.2. Kommunikation

Ich habe mich dafür entschieden, die Kommunikation nicht über ein MI ähnliches Interface zu implementieren. Der Aufwand ein solches Interface zu designen und die Einarbeitungszeit für den CDT Debugger wären zu gross. Die Kommunikation wird also direkt mit dem Prozess erfolgen. Das Design und die Implementation wird in folgenden Kapiteln besprochen.

5.3. API Design

In einem ersten Schritt wurde ein API designt, welches verwendet werden soll um die Kommunikation mit dem Prozess möglichst einfach zu halten.

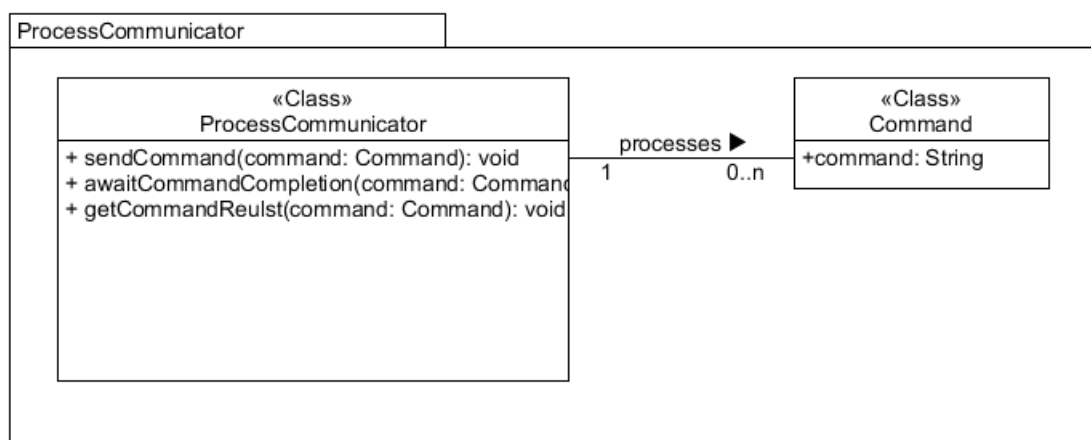


Abbildung 5.1.: Ein erstes Design des Prozess API. Die zentrale Kommunikation geschieht über die Communicator Klasse. Es können Commands abgesetzt werden und es kann auf Resultate gewartet werden.

5.4. Implementierung

Bei der Implementierung kamen dann einige Änderungen hinzu. In folgendem Klassendiagramm sind die Änderungen ersichtlich und die Klassenstruktur wird noch genauer erläutert.

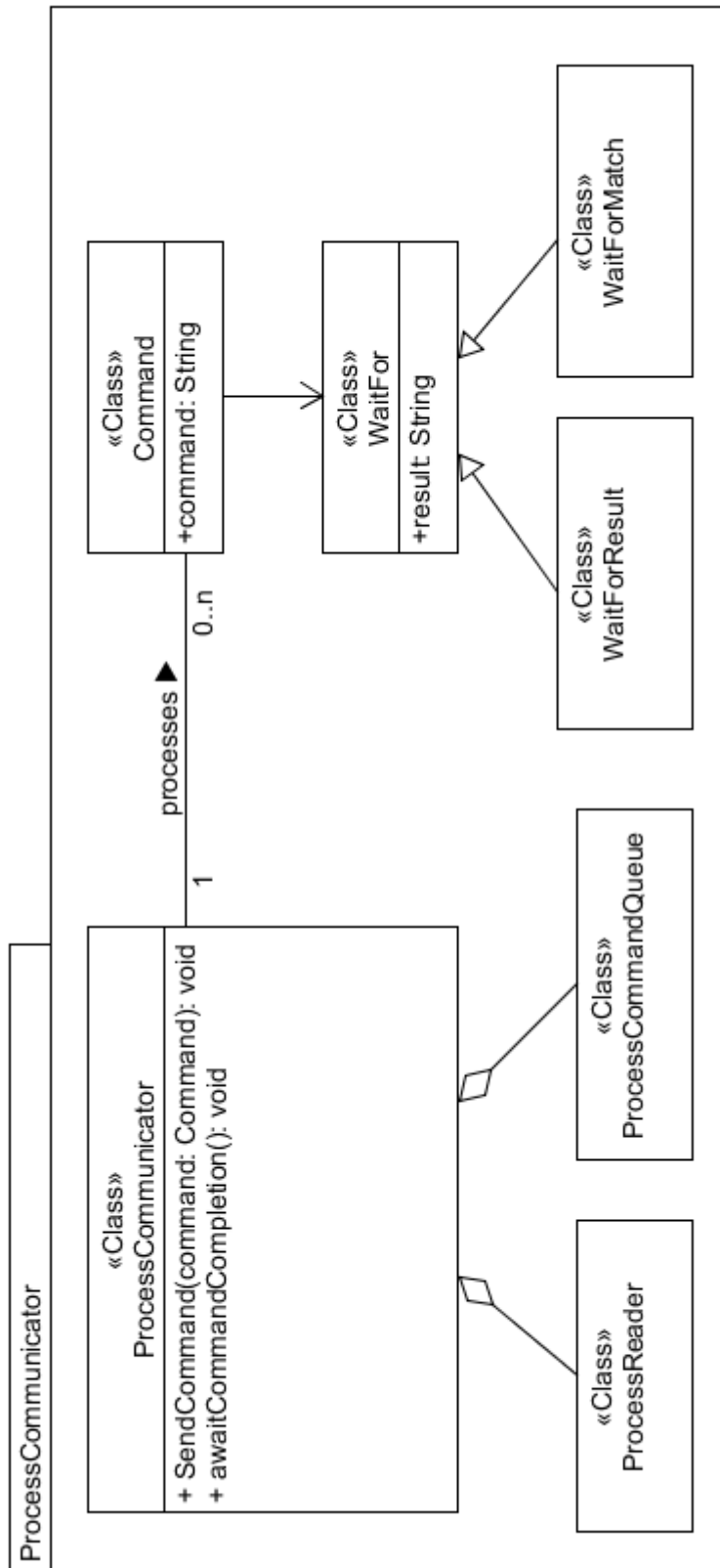


Abbildung 5.2.: Klassendiagramm der.

5.4.1. Klassenbeschreibung

Über das `ProcessCommunicator` API können Commands an den Prozess gesendet werden. Das API stellt blockierende und nicht blockierende Methoden für das senden von Commands an. So können Commands über die `sendCommand` Funktion einen Command abschicken, ohne zu blockieren. Falls auf ein bestimmtes Resultat gewartet werden muss, kann die Funktion `sendCommandAwaitResult` verwendet werden, welche den aufrufenden Thread blockiert, bis das Resultat eingetroffen ist.

ProcessCommunicator

Der `ProcessCommunicator` ist die zentrale Kommunikationsstelle. Über den `ProcessCommunicator` können die Commands gesendet werden und gleichzeitig auch auf die angeforderten Resultate warten.

ProcessReader

Der `ProcessReader` ist eine verschachtelte Klasse des `ProcessCommunicator`, welche als Thread im Hintergrund den Stream des Prozesses liest und verarbeitet. Der `ProcessReader` notifiziert alle Threads, welche auf ein Resultat warten, falls dies eingetroffen ist.

ProcessCommandQueue

Die `ProcessCommandQueue` ist eine verschachtelte Klasse des `ProcessCommunicator`, welche als Thread im Hintergrund Commands verarbeitet und an den Prozess sendet. Die `ProcessCommandQueue` blockiert, falls dies von einem Command spezifiziert wurde. Wenn die `ProcessCommandQueue` blockiert werden keine weiteren Commands mehr verarbeitet, bis das der `WaitFor` Klasse spezifizierten Resultat von dem Prozess geschrieben wurde.

Command

Die `Command` Klasse repräsentiert ein Command, welcher über den `ProcessCommunicator` an den Prozess gesendet werden kann. In einem Command kann eine `WaitFor` spezifiziert werden, falls der Command blockieren soll, bis ein Resultat von dem Prozess geliefert wird.

WaitFor

Mit der abstrakten `WaitFor` Klasse können auf Resultate des Prozesses gewartet werden. Dafür wurden verschiedene Implementationen bereitgestellt. Mit der `WaitForResult` können auf String Resultate gewartet werden. Mit der `WaitForMatch` Klasse kann per Regex auf ein Resultat gewartet werden. In folgendem Pseudocode wird die Funktionalität der `WaitForMatch` Klasse im Zusammenhang mit dem senden eines Commands gezeigt.

```
sendCommandAwaitMatch("foo", "[1-9]")
```

Der aktuelle Thread wird blockiert, bis der Prozess eine Ziffer zwischen 0 und 9 schreibt. Für das warten auf ein Resultat wurde ein Timeout von 10 Sekunden implementiert. Falls der Prozess in dieser Zeit keine Antwort gibt, wird eine `CommandTimeoutException` geworfen und der `ProcessCommunicator` wird heruntergefahren, das keine weiteren Commands mehr gesendet werden können.

6. Debugger

In diesem Kapitel wird beschrieben, wie der Debugger in Eclipse integriert wurde. Es wird aufgezeigt, was für Möglichkeiten existieren einen Debugger in Eclipse zu integrieren und welche implementiert wurden.

6.1. Breakpoints

Als erstes müssen für den Debugger Breakpoints gesetzt werden können. Dafür stehen zwei Möglichkeiten zur Verfügung, welche in den nächsten Kapiteln erläutert und verglichen werden.

6.1.1. Per Konsole

Die erste Möglichkeit ist, die Breakpoints per Konsole zu setzen. Das Senden der Commands funktioniert mit den im Kapitel 5 beschriebenen Klassen. In der Konsole kann der Command

```
debug _function
```

abgesetzt werden, um einen Breakpoint zu setzen.

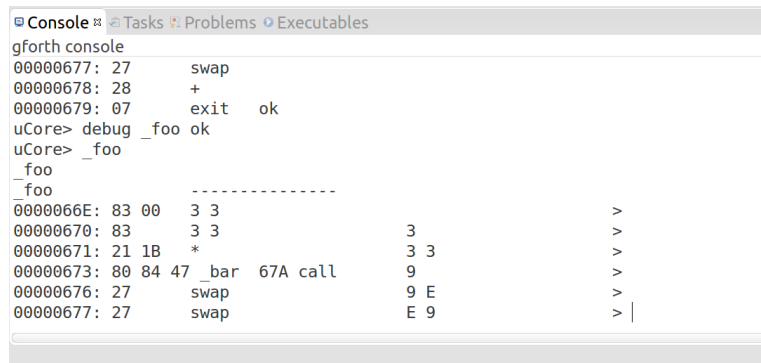
6.1.2. Im Source Code

Eine weitere Möglichkeit ist, Breakpoints im C-File zu setzen. Dies wurde so umgesetzt, dass der Breakpoint nur auf eine Funktionsdefinition gesetzt werden kann. Alle anderen Zeilen des C-Source Codes können nicht direkt auf den übersetzten Forth Code abgebildet werden und sind deshalb nicht erlaubt für die Breakpoints.

Eclipse CDT stellt den Abstract Syntax Tree (AST) des C-Files zur Verfügung. Mit Hilfe des AST kann überprüft werden, ob sich der Breakpoint wirklich auf einer Funktionsdefinition befindet.

6.2. Konsolen basierter Debugger

Eine erste Implementation des Debuggers ist, den schon existierenden Forth Konsolen Debugger im Eclipse zu integrieren. Dieser kann mit dem im Kapitel 5 beschriebenen Prozess Kommunikationsmitteln angesteuert und in einer Eclipse Console View angezeigt werden.



```
gforth console
00000677: 27      swap
00000678: 28      +
00000679: 07      exit    ok
uCore> debug _foo ok
uCore> _foo
 _foo
 _foo
-----
0000066E: 83 00    3 3                                     >
00000670: 83      3 3                                     >
00000671: 21 1B    *                                     >
00000673: 80 84 47 _bar 67A call                          9 >
00000676: 27      swap                                  9 E >
00000677: 27      swap                                  E 9 > |
```

Abbildung 6.1.: Konsolen basierter Debugger. Der Debugger kann über die Eclipse Console View gesteuert werden.

6.3. Forth Debugger

Eine weitere Möglichkeit ist, das Debug User Interface von Eclipse zu verwenden um den Debugger zu steuern. Dies ist für den Endanwender angenehmer, da alle Informationen des Debuggers in einem User Interface ersichtlich sind.

6.3.1. CDT oder JDT Debugging Mechanismen

Das Eclipse JDT stellt mehrere Möglichkeiten zur Verfügung, wie ein Debugger integriert werden kann. Es können die vom Eclipse JDT verwendeten Mechanismen (vor allem das Plugin `org.eclipse.debug.core`), oder die vom CDT erweiterten Mechanismen (vor allem das Plugin `org.eclipse.cdt.debug.core`), welche verwendet werden um einen C oder C++ Debugger zu integrieren. Das vom CDT zur Verfügung gestellte Plugin wird vor allem dazu verwendet, um einen neuen C oder C++ Debugger zu integrieren, da es sich aber um einen Forth Debugger handelt, werden diese Erweiterungen nicht gebraucht. Ich habe mich deshalb dazu entschieden, das JDT Debugging zu verwenden.

6.3.2. Debugger Aktionen

Für den Debugger wurden einige Aktionen, welche schon im Eclipse verwendet werden, implementiert und einige neue Forth spezifische Aktionen hinzugefügt.

Resume

Mit der Resume Aktion kann der während dem Debuggen fortgeführt werden. Dafür wird ein `end-trace` Command an den Forth Prozess gesendet. Der `end-trace` Command entfernt auch alle Breakpoints, diese werden automatisch nach der Ausführung neu gesetzt.

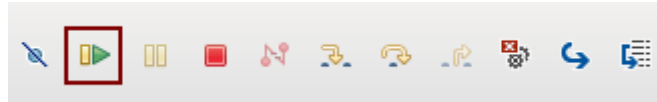


Abbildung 6.2.: Resume Aktion.

Terminate

Mit der Kill Aktion kann der Prozess heruntergefahren werden, in dem der `bye` Command gesendet wird. Falls der Prozess nicht mehr reagiert, sollte die Kill Aktion verwendet werden.

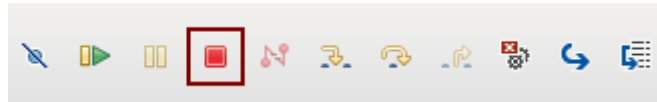


Abbildung 6.3.: Terminate Aktion.

Step Into

Mit der Step Into Aktion kann in eine Funktion gesprungen werden, falls die nächste Zeile ein Funktionsaufruf ist. Dafür wird ein `nest` Command an den Forth Prozess gesendet

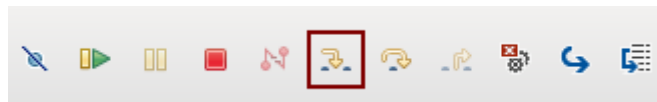


Abbildung 6.4.: Step Into Aktion.

Step

Mit der Step Aktion kann ein normaler single step ausgeführt werden. Dafür wird ein `CR` Command an den Forth Prozess gesendet.



Abbildung 6.5.: Step Aktion.

Kill

Mit der Kill Aktion kann der Prozess terminiert werden, in dem das Kill Signal gesendet wird. Im Normalfall sollte die Terminate Aktion verwendet werden, da sie den Forth Prozess über den `bye` Command beendet.



Abbildung 6.6.: Kill Aktion.

After

Die After Aktion setzt den Breakpoint nach der nächsten Instruktion, falls es sich um einen Rückwärts Sprung handelt, welcher möglicherweise von einer `UNTIL`, `REPEAT`, `LOOP` oder `NEXT` Instruktion kompiliert wurde. Der Rest der Schleife wird deshalb ohne Unterbruch ausgeführt.

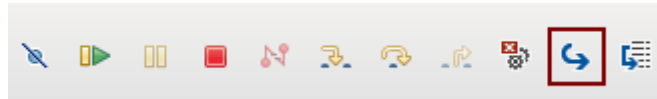


Abbildung 6.7.: After Aktion.

Jump

Mit der Jump Aktion kann über die nächste auszuführende Instruktion gesprungen werden. Dafür wird ein `jump` Command an den Forth Prozess gesendet.

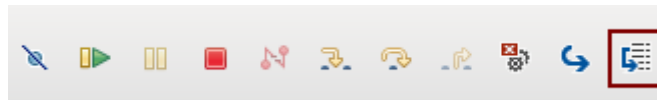


Abbildung 6.8.: Jump Aktion.

6.3.3. Stack View

In der Stack View wird der aktuelle Dstack angezeigt. Die Stack View wird automatisch nach jedem Steppen des Debuggers aktualisiert.

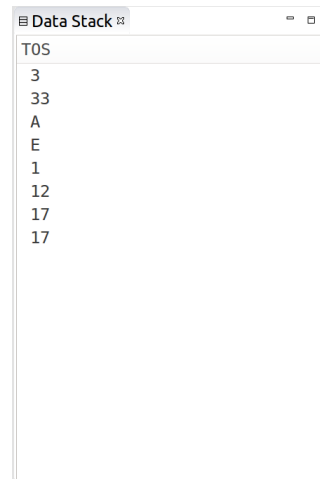


Abbildung 6.9.: Stack View mit aktuellem Dstack Inhalt. Der Top Of Stack (TOS) ist zuoberst in der Liste.

6.3.4. Memory View

In der Memory View kann ein Memory Dump, welcher mit dem `dump` Befehl von uForth abgefragt werden kann, angezeigt werden. Der Memory Dump wird automatisch nach jedem Steppen des Debuggers aktualisiert.

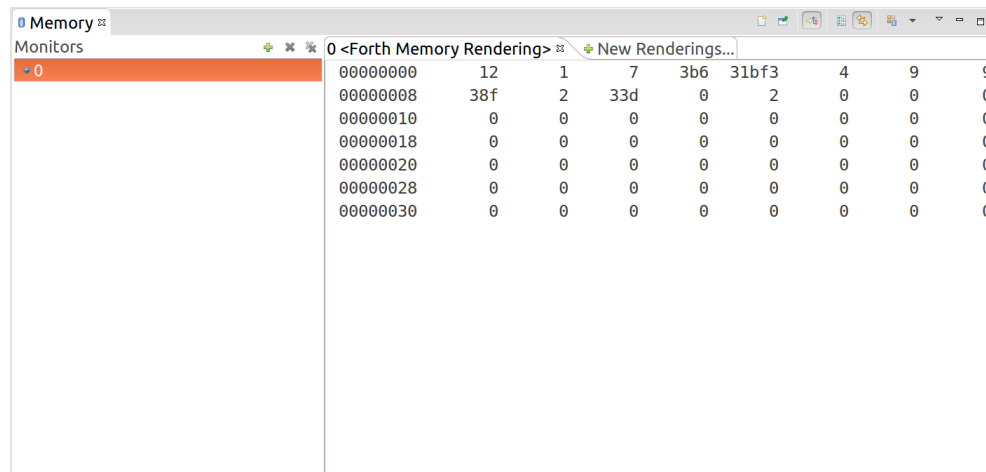


Abbildung 6.10.: Eine Memory Dump, welcher

6.4. C-Debugger

Eine mögliche Erweiterung, wäre den Debugger so zu integrieren, das er direkt auf dem C-Source Code arbeitet (nicht wie bis jetzt, auf dem generierten Forth Code). Dies konnte nicht umgesetzt werden, da Debug Informationen des Compilers fehlen. Der C-Source Code kann nicht auf den entsprechenden generierten Forth Source Code abgebildet werden.

7. Entwicklungsumgebungs Einstellungen

Für die Entwicklungsumgebung wurde eine Eclipse Preference Page erstellt. In diesem Kapitel wird erklärt, was in dieser Preference Page für Einstellungen vorgenommen werden können und was diese für Auswirkungen haben. Die Einstellungen sind im Eclipse Preference Dialog unter MCore zu finden.

7.1. Umbilical

Der Umbilical Port kann über einen Dialog, welcher alle möglichen Ports anzeigt, ausgewählt werden.

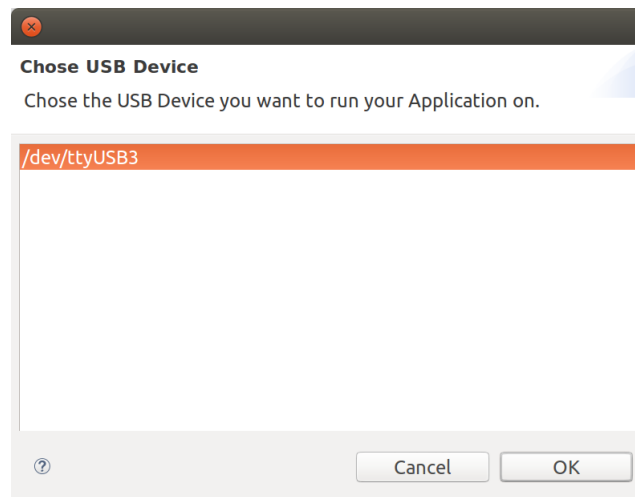


Abbildung 7.1.: Dialog über welchen der Umbilical Port gesetzt werden kann.

Der Port wird, wenn das Programm ausgeführt wird, automatisch gesetzt.

7.2. Loader

Der Loader ist das File, welches von gforth mittels dem Befehl:

```
gforth ./loader.fs
```

gestartet wird. Im Loader befindet sich ein Platzhalter `$INPUT_FILE`, welcher bei der Ausführung des Programms mit dem Namen des Files ersetzt wird.

8. Optimierungen

In diesem Kapitel wird beschrieben, was für Optimierungen für den Compiler implementiert wurden. Im Forth Cross-Compiler sind schon einige Peephole Optimierungen implementiert. In diesem Kapitel werden die neu implementierten Optimierungen mit denen des Cross-Compilers verglichen und gezeigt wo noch bessere Optimierungsstrategien verwendet werden können.

8.1. Peephole Optimierung

Peephole Optimierungen ist eine Art von Optimierung, welche auf einer kleinen Sequenz von Instruktionen durchgeführt wird. Dieses Sequenz wird Peephole oder auch Window genannt. Die Peephole Optimierung versucht Sets von Instruktionen durch kürzere oder schnellere Instruktionen zu ersetzen. [8] Peephole Optimierungen können die grösse des Codes um 15–40 Prozent verkleinern und sind heute in allen gängigen Compilern implementiert. [9] Zu den Peephole Optimierungen gehören unter anderen folgende Arten von Optimierungen:

- Constant Folding - Konstante Expressions auswerten
- Constant Propagation - Konstante Werte in Expressions substituieren
- Strength Reduction - Langsame Instruktionen mit äquivalenten schnellen Instruktionen ersetzen.
- Combine Operations - Mehrere Operationen mit einer äquivalenten ersetzen
- Null Sequences - Unötige Operationen entfernen [8]

8.1.1. Beispiele

Folgend einige Peephole Optimierungs Beispiele anhand von Forth Code.

Constant Propagation

Folgende Instruktionen

```
1  
2  
swap  
+  
dup
```

können durch:

```
2  
2
```

ersetzt werden. Die Instruktionen swap, + und dup können schon zur Kompilierzeit durchgeführt werden.

Combine Operations

Folgende zwei Instruktionen

```
rot  
rot
```

können durch

```
-rot
```

ersetzt werden. Die zwei rot Instruktionen sind äquivalent zu einer -rot Instruktion. Oder die folgenden zwei Instruktionen

```
dup  
drop
```

können durch

```
nop
```

ersetzt werden. Die zwei Instruktionen heben sich auf und können somit entfernt werden.

8.1.2. Optimierungen

Für den Compiler wurden Prototypen mässig zwei Optimierungen in Java implementiert. Die erste Optimierung versucht benachbarte Instruktionen zu vereinfachen. Die zweite Optimierung ist eine einfache Constant Propagation. Die beiden Optimierungen werden in den nächsten Kapiteln genauer beschrieben. Der neue Optimizer wird in zwei Phasen durchgeführt:

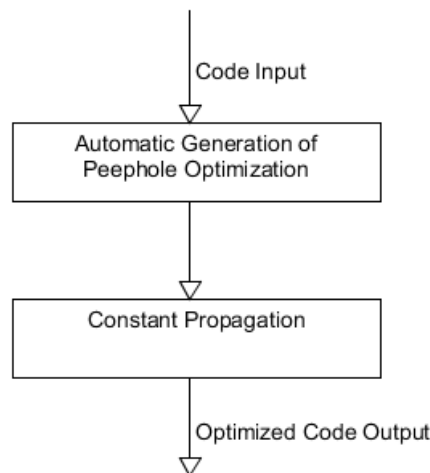


Abbildung 8.1.: Die zwei Phasen des Optimizers.

Im Cross-Compiler wurden unter anderem folgende Peephole Optimierungen schon implementiert.

1. `<lit> + ld`, `<lit> + st`, `<lit> + @`, and `<lit> + \!` werden mit automatisch inkrementierenden Speicherzugriff Instruktionen ersetzt, wenn `<lit>` sich zwischen 4 und 3 befindet
2. Folgende Stack Operationen:
 $swap, swap \rightarrow nop$
 $-rot, rot \rightarrow nop$
 $swap, + \rightarrow +$

Für eine komplette Liste siehe "real time, object oriented with debugger" [10].

8.1.3. Automatische Generierung von Peephole Optimierungen

Klassische Peephole Optimizer versuchen häufig einige Maschinenspezifische Patterns zu korrigieren. Der von Davidson und Fraser [9] beschriebene Algorithmus (PO) verwendet eine Machine Description, simuliert benachbarte Instruktionen und versucht diese mit äquivalenten, schnelleren Instruktionen zu ersetzen. Für den Forth Optimizer wurde ein Teil des PO objektorientiert implementiert.

8.1.4. Constant Propagation

Unter Constant Propagation versteht man, dass vorwärts substituieren von Konstanten im Code. Dies kann zur Folge haben, dass mehrere Instruktionen schon zur Kompilierzeit ausgewertet werden können, wie bei den Beispielen 8.1.1 zu sehen ist.

8.1.5. Resultate und Tests

Die Resultate des Optimierers wurden mit verschiedenen Forth Funktionen getestet und die Resultate mit dem Peephole Optimizer des uForth Cross-Compilers verglichen. Die Funktionen sind im Anhang zu finden.

| Funktion | Orig | Ref | Neu | Kommentar |
|----------|------|-----|-----|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| _Init | 124 | 110 | 116 | Beide Optimizer konnten Code weg optimieren. Die Referenz Implementation produziert vor allem wegen der Speicherzugriffsoptimierung kürzeren Code. Constant Propagation konnte keine durchgeführt werden. |

Tabelle 8.1.: Resultate des neuen Optimizers verglichen mit dem Cross-Compiler Optimizer.

Es hat sich herausgestellt, dass bei allen Realen Beispielen, die Constant Propagation nur wenig Code optimieren konnte. Dies ist vermutlich der Fall, weil Konstante Expressions schon vom Compiler optimiert werden und die implementierte Constant Propagation zu primitiv ist. Im nächsten Kapitel werden einige Erweiterungen vorgeschlagen um diese effizienter zu gestalten.

PO konnte Regeln finden, welche vom Forth-Cross Compiler noch nicht erkannt wurden. Unter anderem folgende:

swap, swap → *nop*
—rot, rot → *nop*
swap, + → *+*

Diese Regeln könnten im Forth Cross-Compiler integriert werden. Im Moment wurden nur Operationen auf dem Stack von PO simuliert, das heisst, es könnten noch mehr Pat-

terns gefunden werden. Im nächsten Kapiteln werden auch noch mögliche Änderungen an PO aufgezeigt.

8.1.6. Mögliche Erweiterungen

SSA Conditionals und Memory Access Superoptimizer in Forth integrieren in Compiler

A. Literaturverzeichnis

- [1] John Arthorne. Faq what is a plug-in? https://wiki.eclipse.org/FAQ_What_is_a_plug-in%3F, 2011.
- [2] Vogella. Eclipse extension points and extensions - tutorial. <http://www.vogella.com/tutorials/EclipseExtensionPoint/article.html>, 2013.
- [3] Xtext. <https://eclipse.org/Xtext/>, 2013.
- [4] Antlr. <http://wwwantlr.org/>.
- [5] Bruno Medeiros. D development tools. <https://github.com/bruno-medeiros/DDT>.
- [6] The gdb/mi interface. https://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html.
- [7] Understanding the gnu debugger machine interface (gdb/mi). <http://www.ibm.com/developerworks/library/os-eclipse-cdt-debug2/>.
- [8] Peephole optimization. https://en.wikipedia.org/wiki/Peephole_optimization.
- [9] J. W. Davidson and C. W. Fraser. The design and application of a retargetable peephole optimizer. In *ACM Trans. Program. Lang. Syst.*, pages 191–202, 1980.
- [10] Klaus Schleisiek. uforth real time, object oriented with debugger, 2013.

B. Abbildungsverzeichnis

| | |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 2.1. Ein Plugin, welches einen Extension Point anbietet. Andere Plugins können diese deklarativ in einem XML File ansteuern. | 7 |
| 4.1. Der Launch Configuration Dialog. Im Dialog kann eine neue MCore Launch Konfiguration erstellt werden. Um eine gültige Konfiguration zu erstellen muss das Projekt und das Forth File, welches gestartet werden soll angegeben werden. | 11 |
| 4.2. Dialog, welcher angezeigt wird, wenn ein Program gestartet wird und der Umbilical Port nicht gültig ist. Der Port kann in der MCore Preference Page geändert werden. | 12 |
| 5.1. Ein erstes Design des Prozess API. Die zentrale Kommunikation geschieht über die Communicator Klasse. Es können Commands abgesetzt werden und es kann auf Resultate gewartet werden. | 14 |
| 5.2. Klassendiagramm der. | 15 |
| 6.1. Konsolen basierter Debugger. Der Debugger kann über die Eclipse Console View gesteuert werden. | 19 |
| 6.2. Resume Aktion. | 20 |
| 6.3. Terminate Aktion. | 20 |
| 6.4. Step Into Aktion. | 20 |
| 6.5. Step Aktion. | 20 |
| 6.6. Kill Aktion. | 21 |
| 6.7. After Aktion. | 21 |
| 6.8. Jump Aktion. | 21 |
| 6.9. Stack View mit aktuellem Dstack Inhalt. Der Top Of Stack (TOS) ist zuoberst in der Liste. | 22 |
| 6.10. Eine Memory Dump, welcher | 22 |
| 7.1. Dialog über welchen der Umbilical Port gesetzt werden kann. | 24 |
| 8.1. Die zwei Phasen des Optimizers. | 27 |

C. Tabellenverzeichnis

| | |
|------------------------------------------------------------------------------------------|----|
| 8.1. Resultate des neuen Optimizers verglichen mit dem Cross-Compiler Optimizer. | 28 |
|------------------------------------------------------------------------------------------|----|

D. Ehrlichkeitserklärung

Hiermit bestätigen die Autoren, diese Arbeit ohne fremde Hilfe und unter Einhaltung der gebotenen Regeln erstellt zu haben.

Benjamin Neukom

Ort, Datum

Unterschrift