Implementieren einer funktionalen Programmiersprache für Mikrokontroller

Facharbeit von Thomas Herzog

Motivation

Vor einiger Zeit bin ich auf das Thema der funktionalen Programmierung aufmerksam geworden. Es war etwas komplett anderes als man gewohnt war.

Schnell fand ich großes Interesse an der mathematischen Nähe, den Closures[1] und den vielen Abstraktionsmöglichkeiten – es hat mein Denken beim Programmieren verändert.

In der Mikrokontrollerprogrammierung dominiert das imperative Programmierparadigma. Da Mikrokontroller oft sehr maschinennah programmiert werden, spiegelt imperative Programmierung gut die sequentielle Abarbeitung von Befehlen dar, wie sie eine CPU ausführen würde.

Ich sah das Projekt LambdaCan[2] – ein Lambdakalkül-Interpreter für Mikrokontroller. Der komplizierteste Ausdruck den LambdaCan errechnen kann ist 11 + 12 = 23.

Mein Ziel ist es nun, eine funktionale Programmiersprache zu entwickeln, die mindestens diesen Ausdruck errechnen kann, sowie Closures und Lambdafunktionen unterstützt.

Die von mir designte Programmiersprache hat den Codenamen **beagle**. Der beagle-Kompiler ist in Haskell programmiert, einer rein funktionalen Sprache. Haskellkenntnisse sind von Vorteil aber nicht notwendig. Wenn keine Haskellkenntnisse vorhanden sind ist es vorteilhaft, der Syntax eine nicht zu große Bedeutung zu geben, da sie für ungeübte Augen ablenkend wirken kann.

Da sich imperatives Programmieren sehr bewehrt gemacht hat in diesem Teilgebiet der Informatik, ist diese Facharbeit eher als Forschungsarbeit zu sehen, wie gut sich funktionale Programmierung und Mikrokontrollertechnik vereinen lassen.

Allgemeines

Wenn man eine neue Programmiersprache entwickeln will, muss man diese gut durchplanen und designen.

Das schönste Design ist nutzlos wenn es keine Möglichkeit gibt diese Sprache überhaupt anzuwenden. Es muss also ein Programm her, welches mit der neu erdachten Sprache arbeiten kann.

Um dies zu erreichen muss ein Interpreter oder Kompiler programmiert werden.

Ein Interpreter ließt die Sprache ein, verarbeitet sie und führt das Programm aus. Die Ausführung geschiet allerdings im Interpreter selbst. Meist gibt es einen großen Array, der den Speicher des auszuführenden Programmes darstellt und ein großes Switch-Case-Statement, welches die einzelnen Befehle auf diesen Array anwendet.
Dies macht Interpretersprachen oft sehr flexibel, da sie sehr nah mit dem Interpreter arbeiten.

Ein Kompiler arbeitet zunächst genau wie ein Interpreter. Der Quelltext muss eingelesen und vorbereitet werden um damit arbeiten zu können.
Der entscheidene Unterschied ist die Ausführung des neuen Programmes. Der Kompiler erzeugt eine neue, meist Ausführbare Datei.

Oft hört man in dem Teilgebiet Kompilerbau den Begriff **Transpiler**. Ein Transpiler generiert nicht

1 Closures sind nicht ausgeführte Funktionen die ihre bisherigen Parameter gespeichert haben. Siehe http://gafter.blogspot.de/2007/01/definition-of-closures.html
2 http://alum.wpi.edu/~tfraser/Software/Arduino/lambdacan.html

wie die meisten Kompiler Maschinencode, sondern ein Programm in einer anderen Programmiersprache.

Da ein nicht assembliertes Assemblerprogramm auch als Quelltext in einer anderen Programmiersprache angesehen werden kann (welche in diesem Fall die Assemblersprache ist), ist die Unterscheidung zwischen Transpiler und Kompiler oft sehr schwammig.

In dieser Facharbeit wird ein Transpiler programmiert, der ein C Programm generiert.

Im Folgenden werden aufgrund der schwammigen Unterschiede zwischen Kompilern und Transpilern die beiden Begriffe synonym benutzt – im Endeffekt wird aus einer Quellsprache ein Zielprogramm generiert, egal ob man den Vorgang kompilieren oder transpilieren nennt.

Ein Kompiler läuft beim Bearbeiten der Inputdaten üblicherweise durch 3 Phasen:

1. Frontend
2. Middleend
3. Backend

Das Frontend beschäftigt sich mit dem Transformieren der Eingabe in Textform zu einem Format, das im Speicher des Rechners besser verarbeitet werden kann.

Das Middleend arbeitet auf diesen vom Frontend generierten Strukturen und führt Transformationen und Optimierungen durch.

Das Backend ist zuständig für die Generierung von Bytecode, der dann in die gewünschte Zielsprache (Assembler, C, …) übersetzt werden kann.

In den folgenden Kaptieln wird jede Phase einzeln erklärt und die Funktionsweise anhand von Codebeispielen gezeigt.

Terminologie

Da bei einem Kompiler / Transpiler viele Programmiersprachen benutzt werden ist eine klare Unterscheidung dieser vorteilhaft.

Um zwischen Transformationen und Vergleichen unterscheiden zu können, wird das Symbol `==` für Vergleiche und das Symbol `-->` für Transformationen verwendet.

Im restlichen Dokument werden folgende Begriffe verwendet:

- Quellsprache
  - Sprache, für die ein Kompiler programmiert wird. In diesem Fall *beagle.*
- Zielsprache
  - Sprache, die für die Generierung des endgültigen Programmes genutzt wird. In diesem Fall *C.*
- Implementierungssprache
  - Sprache, in der der Kompiler geschrieben wird. In diesem Fall *Haskell.*

1. Frontend

Das Frontend liest den Text der kompiliert werden soll ein und ist selbst meißt in 2 Phasen gegliedert:

1. Lexerphase
2. Parserphase

1.1 Lexer

Beim Lexen wird der Text in einzelne Tokens unterteilt. Gleichzeitig werden Zahlen in Werte umgewandelt (von dem String "42" zu dem Wert 42, mit dem gerechnet werden kann). Wenn folgende Tokenstruktur vorliegt

```
data Token = Str String  -- String Konstante
           | Id  String  -- Namen (Identifier)
           | ParanO      -- Klammer auf
           | ParanC      -- Klammer zu
```

ist zu erwarten, dass folgendes gilt:

```
tokenize "print(\"hello world.\")" --> [Id "print",
                                        ParanO,
                                        Str "hello world.",
                                        ParanC]
```

In meiner konkreten Implementierung wird das Lexing-Tool Alex[3] benutzt. Dieses Tool ermöglicht es, den lexikalischen Aufbau in Form von regulären Ausdrücken anzugeben. Aus diesen Ausdrücken wird dann eine Haskell-Quelldatei erzeugt.

Die Alex-Quelldatei ist `src/Lexer.x`.

Aus der Alex-Datei wird eine Funktion `lexer :: String -> [Token]` generiert. Sie bekommt den Quelltext, der tokenized werden soll, als Parameter übergeben und gibt eine Liste der Tokens zurück.

1.2 Parser

Ein Parser ließt eine Liste aus Tokens ein und gibt einen Syntax-Baum (auch Abstrakter Syntax Baum genannt. engl. Abstract Syntax Tree → AST) zurück.

Er erarbeitet also die syntaxtische Analyse des Quellcodes.

Die Struktur des zu kompilierenden Quelltextes wird über den AST dargestellt. Der Syntaxbaum wird durch einen algebraischen Datentyp definiert. Der wichtigste Datentyp ist der eines Ausdrucks.

```
data Expr = If Expr Expr Expr
          | Op String Expr Expr -- Binäre Operatoren
          | Lit Word64          -- Zahlen
          | Str String
```

3 Link zur Alex-Projektseite: https://www.haskell.org/alex/

```
          | BoolConst Bool
          | Var Name
          | Lam Name Type Expr  -- Lambdafunktion
          | Funcall Expr [Expr] -- Funktionsaufruf
          | Let Name Expr Expr
          | Block [Expr]        -- Sequenz von Ausdrücken
```

Um diesen herum wird der entgültige `Toplevel` Datentyp definiert, der eine Globale Definition darstellt (Funktionen / Variablen).

Es gibt viele Möglichkeiten einen Parser zu implementieren. Wenn man einen Parser per Hand programmiert gibt es verschiedene Ansätze, zum Beispiel Top-Down-Parsing oder Bottom-Up-Parsing.

Bei vielen und komplexeren Syntaxregeln ist es empfehlenswert einen Parsergenerator zu verwenden. Ein solcher nimmt, ähnlich wie das Lexing-Tool Alex, eine Liste von Regeln entgegen und erzeugt daraus den entsprechenden Quellcode in der Implementierungssprache.

Ich benutze das Tool Happy[4], welches mit BNF-ähnlicher Syntax arbeitet und zur Kompilierungszeit des Kompilers den passenden Parsercode generiert. Die Happy-Quelldatei ist `src/Parser.y`.

Die Syntax von beagle in Wirth Syntax Notation

```
toplevel    = { (functiondef | vardef) } .
functiondef = name "(" param { "," param} ")" ":" type "=" expr ";" .
vardef      = name                           ":" type "=" expr ";" .
param       = name ":" type

expr = "if" "(" expr ")" expr "else" expr
     | "let" name "=" expr "in" expr
     | "{" { expr ";" } "}"
     | expr operator expr
     | factor

factor = expr "(" expr { "," expr } ")"
       | number
       | string
       | "\" "(" name ":" type ")" "->" expr
       | "true"
       | "false"
       | name
       | "(" expr ")"

type = name
     | type "->" type
     | "(" type ")"
```

Die generierte Parsefunktion hat folgenden Typ: `[Token] -> Either String [Toplevel]`. Das ein `Either String [Toplevel]` zurückgegeben wird spiegelt die Möglichkeit eines Syntaxfehlers wieder.

Mit dem Parsen ist die Phase des Frontends beendet und die Liste aus Definitionen wird im Middleend weiter verarbeitet.

---

4 Link zur Happy-Projektseite: https://www.haskell.org/happy/

2. Middleend

Das Middleend hat für diesen Kompiler folgende Aufgaben:

1. Syntaktischen Zucker entfernen
2. Typsicherheit gewährleisten
3. Den AST in ein Superkombinatorprogramm transformieren

2.1 Syntaktischen Zucker entfernen

Syntaktischer Zucker beschreibt Syntaxregeln, die durch andere ersetzt werden können. Sie dienen meißt nur zu einer Verbesserung der Leserlichkeit.

Bei dem Definieren von Funktionen gibt es in der Quellsprache syntaktischen Zucker.

Zum Beispiel sind folgende Funktionen equivalent:

```
f(x : num) : num = x + 2;

g : num -> num = \(x : num) -> x + 2;
```

Unter anderem sind binäre Operatoren nur syntaktischer Zucker für Funktionsaufrufe:

```
x + 2 --> ($__add__$(x))(2)
```

Der name `$__add__$` ist laut lexikalischen Regeln kein gültiger Bezeichner und kann deshalb nur vom Kompiler selbst generiert werden.

Zudem werden Funktionen mit mehreren Parametern gecurried. Dies bedeutet, dass eine Funktion die mehrere Parameter bekommt in eine Reihe von Funktionen umgewandelt wird, die jeweils nur einen Parameter bekommen.[5]

```
add(a : num, b : num) : num = a + b;
```

ist equivalent zu

```
add : num -> num -> num = \(a : num) -> \(b : num) -> a + b;
```

Wie man sehen kann wird jede Funktion in eine oder mehrere Lambdafunktionen umtransformiert. Dies ist sehr praktisch für den Typchecker, da nicht zwischen Funktionen und Variablen unterschieden werden muss. (Siehe 2.2)

2.2 Typsicherheit gewährleisten

beagle ist eine statisch typisierte Sprache, was bedeutet, dass zur Kompilierungszeit für jeden Ausdruck ein passender Typ gefunden werden kann.

Das Gegenstück zur statischen Typisierung ist die dynamische Typisierung. Bei dieser Technik sind Typen nur zur Laufzeit existent. Das heißt auch, dass Typfehler erst zur Laufzeit entdeckt werden

5 Erläuterung mithilfe des Lambdakalküls und Kategorientheorie: https://ncatlab.org/nlab/show/currying

können.

Keine der beiden Typisierungsformen ist besser als die andere, die Unterschiede liegen in der Philosophie: Bei dynamischen Sprachen will man oft einfach etwas programmieren („quick and dirty") wobei man bei statisch typisierten Sprachen Wert auf weniger Fehler zur Laufzeit legt.[6]

Im beagle-Kompiler sind die Typen für die binären Operatoren und vordefinierten Funktionen und Variablen enthalten (`defaultTypes` in `src/Codecheck.hs`).

Das Typsystem in *beagle* ist ein sehr einfaches. Es gibt keine Polymorphie und keine nutzerdefinierten Datentypen, wobei die Implementierung dieser Elemente nur den Parser, Typchecker und Bytecodekompiler betreffen würden.

Ein Thema über welches es sich zu reden lohnt ist **Typinferenz**. In *beagle* müssen die Typen von Parametern und bei globalen Funktionen auch die Typen der Rückgabewerte explizit erwähnt werden.

```
test(a : num) : num -> num = \(b : num) -> a;
```

Wenn die Typen etwas komplizierter werden wird der Code schnell unübersichtlich. In einigen Sprachen kann deshalb auf das explizite schreiben der Typen verzichtet werden – der Kompiler errechnet die Typen selbst. Dies ist aufwändig zu implementieren, erleichtert aber das Programmieren in der Quellsprache erheblich. Die oben dargestellte Funktion würde in einer Sprache mit Typinferenz so aussehen:

```
test(a) = \b -> a;
```

Wenn das Typsystem polymorphe Typen unterstützt wird der Typ dieser Funktion sogar noch genereller: `a -> b -> a` – die Funktion die den zweiten Parameter ignoriert und den ersten zurückgibt.

Haskell arbeitet mit Typinferenz und einer Erweiterung eines sogenannten Hindley-Milner Typsystems. Hindley und Milner waren logiker und haben ein Typsystem für ein erweitertes Lambdakalkül entwickelt[7]. Da die Implementierung eines Hindley-Milner Typystems sehr aufwändig ist, wird bei dem beagle-Kompiler auf polymorphe Typen und Inferenz verzichtet.

Das Kernstück des Typcheckers ist die Funktion `getType` in `src/Codecheck.hs.`

`getType` gibt den Typ eines Ausdruckes oder einen Typfehler zurück. Für Zahlen, Bool'sche Werte und Stringkonstanten ist dies sehr trivial – der Typ ist bereits bekannt.

```
getType :: TypeEnv -> CoreExpr -> Either String Type
getType te e = case e of

    CLit _ -> pure $ TyName "num"

    CStr _ -> pure $ TyName "str"

    CBool _ -> pure $ TyName "bool"
```

Bei Funktionsaufrufen wird verglichen, ob der aufzurufende Wert überhaupt eine Funktion ist und anschließend ob der Typ des Parameters mit dem gegebenen Argument übereinstimmt.

6 Artikel über Unterschiede in Typsystemen: http://www.lispcast.com/static-vs-dynamic-typing

7 Hindley-Milner, MIT Notizen: http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-827-multithreaded-parallelism-languages-and-compilers-fall-2002/lecture-notes/L06HindleyMilnerPrint.pdf

```
CApp l r -> do
        l' <- getType te l
        r' <- getType te r
        case l' of
            TyFun tx te | tx == r' -> pure te
            _                      -> fail $ "Cannot call\n     " ++ show l'
                                          ++ "\nwith " ++ show r'
```

Gibt die Funktion einen Fehler zurück, wird die Kompilierung abgebrochen.

2.3 Superkombinatorprogramm erzeugen

Die Form des Codes mit all seinen Lambdafunktionen ist für eine Übersetzung in C oder Maschinencode nicht gut geeignet.

Optimal wäre es, wenn man eine Funktion in beagle zu einer Funktion in C kompilieren könnte.

Glücklicherweise gibt es ein Verfahren mit dem man genau dies erreicht – Lambdafunktionen in globale Funktionen umwandeln. Ein solches Programm nennt sich dann **Superkombinatorprogramm**, da es aus Superkombinatoren besteht. Dieses Verfahren nennt sich **Lambdaliften**.

Ein Superkombinator ist eine Funktion, die n Parameter hat und keine freien Variablen beinhaltet.[8]

Ein Kombinator ist ein Ausdruck, der keine freien Variablen beinhaltet, somit ist ein Superkombinator nur eine Spezialisierung eines Kombinators.[9]

Somit stellt sich die Frage, was freie Variablen sind.

Eine Variable ist frei, wenn sie nicht durch eine Lambdafunktion oder ein `let`-konstrukt eingefürt wurde.

Beispiele:

Gebundene Variablen

```
\(x : num) -> let y = 3 in x + y
```

Freie Variablen

```
\(x : num) -> y;
let test = 1337 in b;
```

Da beagle mit Currying arbeitet, werden viele Lambdafunktionen vom Kompiler generiert (welche auch freie Variablen beinhalten können).

Beispiel für Lambdafunktionen mit freien Variablen:

```
let add = \(a : num) -> \(b : num) -> a + b in add(2, 3);
```

In diesem Fall ist die Variable `a` in der inneren Lambdafunktion *frei*.

Wie transformiert man nun so einen Ausdruck so, dass er keine freien Variablen mehr enthält?

---

8 https://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/PAGES/223.HTM

9 https://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/PAGES/224.HTM

Da Superkombinatoren per Definition n Parameter besitzen, können diese beiden Lambdafunktionen zu einem Superombinator umgewandelt werden – einer, der `a` **und** `b` als Parameter bekommt. So ist `a` nicht mehr frei.

Wenn die Ausdrücke etwas komplizierter sind, werden die freien Variablen einfach als zusätzliche Parameter an die Funktion übergeben.

```
const(a : num) : num -> num =
        let f = \(b : num) -> a // a ist hier frei.
        in f;
```

Durch die Transformation entstehen folgende Superkombinatoren:

```
const(a : num) : num -> num =
        let f = $1(a) // $1 ist die Lambdafunktion. a wird übergeben, damit a in
                      // dem Superkombinator bekannt ist.
        in f;

$1(a : num, b : num) : num = b; // a ist die freie Variable, die zum Parameter
                                // wurde
```

Diese neu erstellten Superkombinatoren sind nun unsere „globalen” Funktionen, die wir zur Codegenerierung benutzen.

Das Buch „The implementation of functional programming languages” von Simon Peyton Jones erklärt diesen Prozess genauer in Kapitel 13.[10]

10 https://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/PAGES/220.HTM

3. Backend

Die dritte und letzte Phase des Kompilers ist einzig und allein für die Erstellung des Codes in der Zielsprache zuständig.

Das Backend wird ebenfalls in mehrere Phasen unterteilt:

1. Kompilieren zu Bytecode
2. Bytecode optimieren
3. Generieren von C Code

3.1 Kompilieren zu Bytecode

Im Kompilerbau ist es oft üblich vom AST erst in einen Zwischencode, den Bytecode, zu kompilieren. Dieser Bytecode ist oft abstrakter als konkreter Maschinencode und bietet oft noch Möglichkeiten zum Optimieren.

Bei der Planung des Bytecodes sollte man schon entschieden haben, mit welcher Architektur man arbeiten will (Registerbasiert, Stackbasiert).

Ich habe mich für eine stackbasierte Kompilierung entschieden, da diese viel simpler ist und da durch die nähe zum Lambdakalkül ein sehr großer Teil des Codes nur aus Funktionsaufrufen besteht, wo ohnehin schon mit Stacks gearbeitet wird.

Die Idee der Kompilierung in Bytecode ist es, aus der rekursiven Struktur der Ausdrücke eine flache Liste aus Befehlen zu erzeugen.

Der Befehlssatz hat nur 9 Befehle, die hier in Haskell-Data-Syntax dargestellt sind (siehe `src/Bytecode.hs`):

```
data Instruction = PushNum    Word64
                 | PushStr    String
                 | PushVar    Int    -- Offset im Stack
                 | PushGlobal String
                 | Call       Int    -- N Aufrufe
                 | Unwind     Int    -- N Variablen entfernen
                 | Pop
                 | Return
                 | Cond [Instruction] [Instruction]
```

Die `PushX`-Befehle pushen neue Werte auf den Stack. Lokale Variablen und Parameter befinden sich ebensfalls auf dem Stack. Referenziert werden diese über ein Offset von der „Spitze” des Stacks.

Der `Unwind`-Befehl entfernt n Stackelemente unter dem obersten.
Aus dem Stack `[1, 2, 3, 4]` wird nach einem `Unwind 2` Befehl folgender Stack: `[1, 4]`, wobei 1 das untere und 4 das obere Ende des Stacks darstellt.

Der `Pop`-Befehl entfernt das oberste Stackelement.
Aus dem Stack `[1, 2, 3, 4]` wird nach einem `Pop` Befehl folgender Stack: `[1, 2, 3]`, wobei 1 das untere und 4 das obere Ende des Stacks darstellt.

Der `Call`-Befehl ruft eine Funktion auf. Die Funktion und der Parameter müssen auf dem Stack liegen. Da, wie bereits erwähnt, Funktionsaufrufe einen großen Teil des Codes ausmachen, ist es von Vorteil, den Call-Befehl um ein Attribut zu erweitern, welches die Anzahl der Aufrufe darstellt. Dies ermöglicht auch eine effizientere Ausführung des Programmes.

Das eigentliche Kompilieren findet in der `generateCode` Funktion statt. Alle konstanten Werte (Zahlen, Strings und Bool'sche Werte) werden in Push-Operationen kompiliert.

Alle anderen Strukturen des ASTs werden rekursiv kompiliert.

Beispiel von Inputprogramm und dem resultierendem Bytecode:

Inputprogramm:

```
answer : num = 2 * 21;

let_test : num = let x = 2
                 in  x * answer;
```

Bytecode:

```
answer
    PushNum 21
    PushNum 2
    PushGlobal "$__mult__$"
    Call 1
    Call 1
    Unwind 0

let_test
    PushNum 2
    PushGlobal "answer"
    PushVar 1
    PushGlobal "$__mult__$"
    Call 1
    Call 1
    Unwind 1
    Unwind 0
```

3.2 Bytecode optimieren

Da Funktionsaufrufe so häufig sind, wurde der `Call` Befehl um ein Attribut erweitert: die Anzahl der Aufrufe.

Die einfache kompilier-Funktion `generateCode` generiert aber noch unoptimierten Code.

Es ist also sinnvoll, den Code zu optimieren. Deshalb gibt es eine Funktion `simplify`, welche aufeinanderfolgende `Call` und `Unwind` Befehle zusammenfasst.

Der Bytecode aus dem obrigen Beispiel wird vereinfacht zu:

```
answer
    PushNum 21
    PushNum 2
    PushGlobal "$__mult__$"
    Call 2
```

```
let_test
    PushNum 2
    PushGlobal "answer"
    PushVar 1
    PushGlobal "$__mult__$"
    Call 2
    Unwind 1
```

Wer mehr Aufwand betreibt, kann diese simplify Funktion noch beliebig erweitern, sodass sie zum Beispiel konstante Berechnungen wie 2 + 3 erkennt und durch den Wert 5 ersetzt.
Zusätzlich könnten Rekursionen durch Schleifen ersetzt werden, was zu besserer Speichernutzung und Ausführungszeit führen würde.

Da solche Optimierungen oft sehr komplex und aufwendig sind, wird bei dem beagle Kompiler auf diese erweiterten Optimierungen verzichtet.

3.3 Generieren von C Code

Computer, die mit der von Neumann Architektur arbeiten, benutzen zum Speichern der Rücksprungadressen den Stack des Computers[11].

Da der beagle Kompiler Code generiert, der alle Werte auf dem Stack speichert, muss zwischen den Werten und den Rücksprungadressen unterscheiden werden können. Dies geht am einfachsten, wenn man **zwei seperate Stacks** benutzt.

Da C Code in der Ausführung ebenfalls nur auf die call und ret Befehle des Prozessors zugreift, benutzt der Funktionsmechanismus von C den **Hardwarestack** des Computers.

Es muss also ein eigener Stack implementiert werden, der die Werte speichert.

Alle Funktionen, die zur Laufzeit zur verfügung stehen müssen, werden die **Laufzeitumgebung**, oder oft einfach nur Runtime, genannt.

Dazu mehr im Abschnitt *Laufzeitumgebung*.

Da beagle currying unterstützt, und damit auch Closures, muss auf irgendeine Weise die Anzahl der benötigten Parameter gespeichert werden. Dafür werden Funktionsobjekte angelegt.

Das hat die Folge, dass jede Funktion nicht nur aus dem nötigen Code besteht, sondern auch aus einem Objekt, das die "Metadaten" der Funktion enthält. Dieses Objekt wird bei jedem Aufruf kopiert, da es mehrere Closures gleichzeitig geben kann.

Für jede Funktion / jeden Wert werden jeweils die Prototypen generiert, damit es nicht zu Problemen in der Reihenfolge der Funktionsdefinitionen im C Code kommt.

Werte, die keine Funktionen sind (also Superkombinatoren ohne Paramater), werden nur in das Objekt mit den Metadaten kompiliert. Das Objekt selbst wird über eine Funktion generiert und zurückgegeben.

Da bereits der Bytecode aus mehreren Befehlen besteht, kann jeder Befehl in ein C-Statement übersetzt werden.

Jeder Befehl im Bytecode wird durch eine Funktion realisiert (siehe *Laufzeitumgebung*).

Diese direkte Übersetzung geschiet in der Funktion `genInstr` in der Datei `src/Codegen.hs`. Die

11 Abschnitt call, ret: http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Funktion übersetzt einen Befehl des Bytecodes in einen String, der den C Code enthält.

```
genInstr :: Instruction -> String
genInstr i = case i of

    PushNum i -> "__push_num(" ++ show i ++ ");"

    PushStr s -> "__push_str(" ++ show s ++ ");"

    PushVar i -> "__push_val(__read_val(" ++ show i ++ "));"

    PushGlobal g -> "__push_val(" ++ mangle g ++ "());"

    Call n -> "__call(" ++ show n ++ ");"

    Unwind n -> "__unwind(" ++ show n ++ ");"

    Pop -> "__pop();"

    Return -> "return;"

    Cond t e -> "if (__pop_num(0)) {" ++ tc ++ "} else {" ++ ec ++ "}"
        where tc = unwords (map genInstr t)
              ec = unwords (map genInstr e)
```

Die Funktion `mangle` ändert einen Bezeichner so ab, dass er ein eindeutiger **gültiger** C-Bezeichner ist.

Aus dem Bytecode

```
PushNum 21
PushNum 2
PushGlobal "$__mult__$"
Call 2
```

wird folgende Sequenz aus C Statements generiert:

```
__push_num(21);
__push_num(2);
__push_val(builtin__mult__builtin());
__call(2);
```

Laufzeitumgebung

Eine Laufzeitumgebung nennt man Code, der nötig ist, um ein Programm ausführen zu können.

Im Fall eines beagle-Programmes muss für die Existenz eines seperaten Stacks gesorgt werden, sowie für die Realisierung von Closures.

Ein Wert (sowohl Zahlen, Strings und Closures) werden in einer Struktur `__val` gespeichert.

```
struct __val {
    void *data;
    struct __node *args;
};
```

In dem `data` Feld werden Zahlen, Zeiger auf Strings und Zeiger auf Funktionen gespeichert.

Das Feld `args` ist nur für Closures relevant undspeichert die bisherigen gegebenen Parameter in einer einfach verketteten Liste.

Der Code dieser Laufzeitumgebung befindet sich in der Datei `lib/runtime.c`.

Notwendig sind Funktionen zum Bearbeiten des Stacks:

```
void __push_val(struct __val v);
void __push_num(uint16_t val);
void __push_str(char *s);
struct __val __read_val(uint16_t o);
void __pop();
void __pop_n(uint16_t n);
uint16_t __pop_num(uint16_t o);
void __unwind(uint16_t o);
```

Notwendig ist noch eine Funktion zum Aufrufen von Funktionen: `__call`

Bei jedem Aufruf der `__call` Funktion wird überprüft ob bereits alle Parameter gegeben sind. Wenn dies der Fall ist, werden die Inhalte der Liste auf den Stack gelegt und die Funktion aufgerufen.

Nach einem erfolgreichen Aufruf einer Funktion können die Parameter gegebenenfalls gelöscht werden. Da es aber durchaus vorkommen kann, dass Closures über mehrere Funktionen verteilt aufgerufen werden können, ist es nicht sicher die Parameter zu löschen.

Um die Sicherheit zu gewährleisten bedarf es eines Garbage-Collectors oder statischer Analysen. Da die einfachste Form des Garbage-Collectings, das Zählen von Referenzen auf ein Objekt, viel zusätzlichen Speicherplatz bedarf habe ich mich gegen diese Methode entschieden.

Auch ein Garbage-Collector, der einen bestimmten Speicherbereich auf zugriffe Untersucht, würde die Performance stark beeinträchtigen.

Man hat die Möglichkeit über ein Kompilerflag (`--free-args`) das automatische (und damit unsichere) Freigeben des Speichers zu aktivieren. Dies sollte allerdings nur getan werden, wenn sichergestellt ist, dass keine Closures mit bereits zugewiesenem Speicher für Parameter kopiert werden.

Zusätzlich gibt es 2 weitere Kompilerflags, die die Codegenerierung beeinflussen

1. `--comp`

   Kompiliert Code für die Ausführung auf einem Rechner. (Dummy lcd Funktionen)

2. `--stack-size`

   Setzt die größe des beagle-Stacks in Bytes. (Ein Wert besteht aus 4 Byte)

Zu Testzwecken wird eine LCD-Library, die von mir programmiert wurde, in die Laufzeitumgebung hinzugefügt, damit das Ergebnis einer Berechnung auch auf einem Mikrokontroller dargestellt werden kann.

Die `main` Funktion des C Codes befindet sich in der Datei `lib/runtime_main.c` und wird an den fertig kompilierten C Code angefügt. In dieser Funktion kann die LCD Ansteuerung verhindert werden oder mehr als nur eine beagle Funktion aufgerufen werden.

Fazit

Mit dem Programm

```
main : num = 11 + 13;
```

kann gezeigt werden, dass die beagle Implementierung mächtiger ist als LambdaCan.

Natürlich können auch komplexere Funktionen ausgeführt werden.

Beispiel:

```
main : num = fib(24);

fib(n : num) : num =
      if (n < 2)
            n
      else
            fib(n - 1) + fib(n - 2);
```

Dieser Code errechnet die größte Fibonaccizahl, die mit 16 Bit darstellbar ist: 46368
Aufgrund der vielen rekursiven Funktionsaufrufe ist es erforderlich das Flag `--free-args` zu setzen.

Auch übergeordnete Funktionen werden wie erwartet korrekt ausgeführt:

```
main : num = apply_n_times(5, \(n : num) -> n + 1, 1);

apply_n_times(n : num, f : num -> num, x : num) : num =
      if (n == 0)
            x
      else
            apply_n_times(n - 1, f, f(x));
```

Nach der Ausführung des Programmes wurde der Wert 6 errechnet.

Mit dem Ergebnis der Facharbeit bin ich durchaus zufrieden, da der generierte Code stabil läuft und der Kompilercode so modular ist, dass er sich leicht erweitern lässt.

Mögliche Erweiterungen

Um das Programmieren in beagle angenehmer und leichter zu gestalten, würden sich folgende Erweiterungen anbieten

- nutzerdefinierte Datentypen
- einfache Anbindung an C Funktionen
- Typinferenz
- polymorphe Typen
- Garbage Collector
- Module um Code in Dateien zu trennen