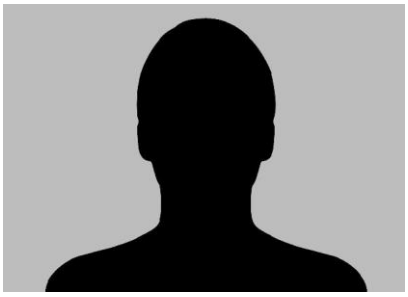
Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

InIT Institut für angewandte Informationstechnologie

Functional Go



Diplomand
Ramon Benno Rüttimann

Dozierende
Karl Rege
Gerrit Burkert

Innerhalb der letzten zehn Jahre haben Konzepte und Ideen aus dem funktionalen Programmieren im Alltag von vielen Entwicklern Fuss gefasst. Häufig wird empfohlen, eine rein funktionale Programmiersprache wie zum Beispiel Haskell zu lernen, um sich mit diesen Konzepten vertraut zu machen. Viele haben jedoch Mühe, eine neue Syntax und ein neues Paradigma gleichzeitig zu lernen. Das Ziel dieser Arbeit ist deswegen, mit Hilfe einer multiparadigmatischen Programmiersprache mit bekannter Syntax einen einfacheren Einstieg in funktionales Programmieren zu ermöglichen.

Um dieses Ziel zu erreichen, wurde die Programmiersprache Go aufgrund ihrer syntaktischen Simplizität und Vertrautheit gewählt. Da Listen jedoch oft eine zentrale Rolle im funktionalen Programmieren einnehmen, ist ein Nachteil dieser Wahl, dass Go keinen eingebauten Datentyp für Listen besitzt. Zwar wird dieser Nachteil durch Gos `Slices' gemildert, jedoch fehlen viele Funktionen höherer Ordnung, um mit Listen zu arbeiten - `map', `filter' und `reduce', um einige zu nennen. Da Gos Typensystem keinen Polymorphismus bietet, müssen diese Funktionen im Compiler implementiert werden, um eine möglichst benutzerfreundliche Verwendung zu ermöglichen.

Zusätzlich dazu wird die Bedeutung von rein funktionalem Programmieren im Kontext dieser Arbeit festgelegt und auf Basis dieser Definition das Code-Analyse-Tool `funcheck' entwickelt, welches nicht-funktionale Konstrukte im Programmcode meldet.

Mit den neuen eingebauten Funktionen `fmap', `filter', `foldr', `foldl' und `prepend' sowie dem Linter `funcheck' erweist sich Go als geeignete Programmiersprache, um einen einfachen Einstieg in funktionales Programmieren zu ermöglichen. Der primäre Grund spiegelt sich auch im Go-Idiom `clear is better than clever' wider. Obwohl funktionaler Go Code länger ist als in funktionalen Sprachen, ist dieser auch einfacher nachzuvollziehen. Des Weiteren zeigt die Arbeit aber auch, dass es keine Alternative zu einer rein funktionalen Sprache wie Haskell gibt, um sich funktionales Programmieren vollständig anzueignen. Haskells zwar ungewöhnliche, aber prägnante Syntax sowie das Design der Sprache - das Typensystem, Pattern Matching, die Reinheitsgarantien und vieles mehr - bilden hierfür eine solide und oft verwendete Grundlage.

```go
// Counterpart to Haskell's `derive Show`
//go:generate stringer -type parity

type parity int

const even parity = 0
const odd parity = 1

// shouldBe returns a function that returns true if
// an int is of the given parity
func shouldBe(p parity) func(i int) bool {
  return func(i int) bool {
    return i%2 == int(p)
  }
}

func main() {
  l := []int{1, 2, 3, 4, 5}
  lst := fmap(
    func(i int) int { return i * 5 },
    prepend(0, l),
  )

  addToString := func(s string, i int) string {
    return s + strconv.Itoa(i) + " "
  }

  // fold over even / odd numbers and add them to
  // a string
  evens := foldl(addToString, even.String()+": ",
    filter(shouldBe(even), lst))
  odds := foldl(addToString, odd.String()+": ",
    filter(shouldBe(odd), lst))

  fmt.Println(evens, odds)
  // even: 0 10 20  odd: 5 15 25
}
```

Demonstration von funktionalem Go Code