Die Programmiersprache F#*

Daniel Brunner
daniel@dbrunner.de

30. September 2020

*Diese Ausarbeitung habe ich im Rahmen einer Studienleistung an der FernUniversität in Hagen, Fakultät für Mathematik und Informatik, angefertigt.

Inhaltsverzeichnis

1 Einleitung

Obschon die objektorientierte Programmierung das vorherrschende Programmierparadigma darstellt, erfreut sich auch die funktionale Programmierung einer gewissen Beliebtheit. Sie orientiert sich an dem Funktionsbegriff der Mathematik dahingehend, dass das Ergebnis einer Funktion nur von ihrem Eingabewert abhängen soll. Das führt in dieser Strenge natürlich dazu, dass man entweder auf Seiteneffekte komplett verzichten muss oder diese an bestimmte Stellen im Programmcode bündelt. F# stellt eine solche funktionale Programmiersprache für das .NET Framework dar. Sie zeichnet sich wie ihre Vorgängerinnen durch ein starkes, polymorphes Typsystem und erweiterbare Datentypen aus.

Ein wichtiges Konzept stellt die sogenannte referentielle Transparenz dar. Dieses besagt, dass ein und dieselbe Variable an verschiedenen Stellen ihres Gültigkeitsbereichs immer den gleichen Wert bezeichnet. Somit kann man eine Variable durch ihre Definition ersetzen ohne die Bedetung zu verändern. Dies erleichtert die Programmtransformation und ermöglicht den Nachweis von bestimmten Eigenschaften (vgl. Erwig 1999, S. 8 f.). Programme funktionaler Sprachen bestehen in der Regel aus einer Reihe von Definitionen und Ausdrücken. Anweisungen, insbesondere Zuweisungen an Variablen gibt es zumeist nicht (vgl. Erwig 1999, S. 8 f.). Verbunden mit diesen Ansätzen ist das Versprechen, leichter typsicheren, prägnanten und korrekten Code zu erzeugen, der zudem noch leichter wartbar sein soll (vgl. Syme 2020, S. 3, Carter 2020d).

Ziel dieser Arbeit ist es, die Programmiersprache F# vorzustellen. Sie ist keine "reine" (vgl. Syme, Granicz und Cisternino 2015, S. 55). funktionale Programmiersprache, sondern erlaubt auch Seiteneffekte wie Ein- und Ausgabe und beinhaltet weitere imperative Konstrukte. Darüberhinaus kann mit ihr auch objektorientiert programmiert werden. Neben der reinen Vorstellung von F# soll auch ein Blick auf die Interaktion mit anderen .NET-Sprachen geworfen werden und auf Einschränkungen dieser Interaktion hingewiesen werden.

Im Folgenden werden in Abschnitt 2 wichtige Grundzüge von F# an Beispielen dargestellt. Im darauf folgenden Abschnitt 3 werden nicht-funktionale Eigenschaften der Sprache dargestellt sowie an einem Beispiel das Zusammenspiel incl. einiger Einschränkungen mit anderen .NET-Sprachen, in diesem Fall C#, illustriert. Ein Ausblick auf die zukünftige Sprachentwicklung (Abschnitt 5) schließt die Arbeit ab.

2 Grundzüge der Programmiersprache F#

2.1 Geschichte und Verhältnis zu anderen .NET-Sprachen

Entstanden ist F# bei Microsoft Research in Cambridge (England) während der Entwicklung der Generics für .NET. Ihre Hauptideen bezieht sie von den Sprachen der

ML-Sprachfamilie,[1] besonders beeinflusst ist sie von OCaml (vgl. Syme 2020, S. 7f.). Erste Implementierungen entstanden als Versuch, die Programmiersprache OCaml auf das .NET Framework zu portieren. Seinerzeit wurde das Projekt noch mit der Bezeichnung Caml.NET geführt (vgl. Syme 2020, S. 17–18, Syme, Granicz und Cisternino 2015, S. 2). Der Hauptentwickler Don Syme[2] lies sich – so seine Schilderung in Syme 2020 – von Xavier Leroy, einem der Entwickler von OCaml (vgl. Leroy 2020), überzeugen, nicht nur eine Portierung zu versuchen, sondern auch Sprachelemente zu variieren, so dass sie ggf. besser zum .NET Framework passten. Im Zuge dieser Diskussionen und der ersten Arbeiten fiel auch die Entscheidung einen anderen Namen, nämlich F#, zu wählen (vgl. Syme 2020, Kapitel 6).

Im Jahr 2007 entschied Microsoft, die Sprache offiziell in den Kanon der .NET-Sprachen aufzunehmen (vgl. Syme 2020, S. 38) und mit "Visual F# 2.0" wurde die Version 2.0 im Jahr 2010 veröffentlicht (vgl. Syme 2020, S. 49). F# ist seit dem eine von drei von Microsoft bereit gestellten Programmiersprachen für das .NET Framework (vgl. Microsoft 2020b). Im Folgenden wird F# in der Version 4.7 für das .NET Core Framework, Version 3.1 betrachtet. Da ab Version 5 des .NET Frameworks alle Varianten des Frameworks zusammengeführt werden sollen und zukünftig auf der Code-Basis von .NET Core basieren sollen (vgl. Lander 2019), wird im Folgenden nur vom .NET Framework gesprochen, auch wenn es sich genau genommen um das .NET Core Framework handelt.

Der F#-Compiler erzeugt aus dem Quellcode von F# den intermediären Code des .NET Frameworks, die sogenannte Common Intermediate Language (CIL) (vgl. Microsoft 2020d), diese stellt eine maschinenunabhängige Repräsentation des Programms dar. Bei der CIL handelt es sich um eine objektorientierte Assembler-Sprache, die als Stack-Maschine realisiert ist. Die Ausführungsumgebung von .NET, die sogenannte Common Language Runtime (CLR), übersetzt diese Repräsentation zur Laufzeit in den endgültigen Maschinencode (JIT: Just in Time) und führt diesen auf dem jeweiligen Zielsystem aus (vgl. Microsoft 2020c). War das frühere .NET Framework auf die Windows-Plattform von Microsoft beschränkt, kann der entstehende Code mittlerweile auch auf Linux und macOS ausgeführt werden.[3]

Funktionale Sprachen, die Sprachen der ML-Familie im Besonderen, sind ausdrucksorientierte Sprachen. Dies bedeutet, dass die Ausführungsumgebung nach und nach Definitionen und Ausdrücke zu Werten auswertet. Definitionen für Typen, Werte und Funktionen verändern die Ausführungsumgebung und bei der Auswertung werden Ausdrücke durch ihren resultierenden Wert ersetzt. Werte sind dabei Ausdrücke, die nicht weiter ausgewertet werden können. Ein solcher Wert hat immer einen eindeutig bestimmbaren Typ, auch Funktionen haben einen Typ (vgl. Erwig 1999, S. 15).[4]

[1] Zur Geschichte von Standard ML siehe MacQueen, Harper und Reppy 2020, auch verfügbar als LaTeX-Quellcode unter *The Standard ML Language Family* 2020.
[2] Er zeichnet auch für die Generics in C# mitverantwortlich, siehe Syme 2020, S. 10–12.
[3] Bspw. durch .NET Core (vgl. Microsoft 2020a).
[4] Im Folgenden wird gelegentlich auf Literatur zu Standard ML zurückgegriffen, soweit dort die Konzepte

Der Kern von F# baut auf diesen Prinzipien auf (vgl. Syme 2020, S. 3):

- Sie folgt dem funktionalen Programmierparadigma und hat ein strenges Typsystem.
- Programme bestehen aus Definitionen von Typen, Variablen und Funktionen und die Typen dieser Definitionen werden automatisch ermittelt (Typinferenz).
- Die Sprache unterstützt die Komposition von Funktionen, Pattern Matching, Rekursion, Listen und weitere Kollektionen, Funktionen höherer Ordnung.

Dabei versucht die Sprache die Programmiererin mit einem möglichst der reinen funktionalen Programmierung nahe kommenden Standardverhalten zu unterstützen. Zu nennen sind hierzu (vgl. Wlashin 2020, Abschnitt "Out-of-the-box behavior for types", Syme, Granicz und Cisternino 2015, S. 10):

- Datentypen in F# sind unveränderbar, also immutable.
- Es wird auf die Verwendung eines Null-Typs für die Typen der F#-Standardbibliothek verzichtet.
- Die meisten Typen bringen einen Test auf strukturelle Gleichheit mit.

F# zielt auf eine Vielzahl von Anwendungsfällen und beinhaltet daher viele Sprachelemente und eine große Standardbibliothek. Folglich können sich die Ausführungen nur auf wesentliche Elemente des Sprachkerns beziehen. Hingewiesen sei jedoch auf Carter 2020c und Syme, Granicz und Cisternino 2015, in denen diese Konzepte ausführlich dargestellt werden. Zu nennen sind beispielhaft Rechnen mit Einheiten (SI, Währungen etc.), Type Providers, weitere Standardtypen (Option, Set, Result, Enumeration, Map...), eine besondere Syntax und Erweiterung (sogenannte computational expressions) für asynchrone Programmierung, seiteneffektbehaftete Programmteile, monadische Strukturen.

Zusammengefasst ist F# eine quelloffene, plattformübergreifende, streng typisierte, prägnante Programmiersprache, einsetzbar für eine Vielzahl von Anwendungen und Domänen (vgl. Syme 2020, S. 4).

2.2 Werte und Ausdrücke

F# ist eine ausdrucksorientierte Sprache. Ein F#-Programm wird im Kern als eine Reihe von Ausdrücken angesehen, die jeweils ausgewertet werden und Werte als Ergebnisse zurückliefern. In der Folge führt diese Art der Auswertung dazu, dass im Gegensatz zu bspw. C# nur auf solche Definitionen zurückgegriffen werden kann, die auch textlich vorher definiert wurden. Etwaige zirkuläre Abhängigkeiten müssen vorher durch

identisch sind und besonders geschickt dargestellt wurden.

geeignete Sprachkonstrukte aufgelöst werden.[5] Mit einem speziellen mitgeliefertem Programm, dem F# Interactive, kann man dies auch als Lesen-Auswerten-Ergebnis-Schleife (Read Eval Print Loop, oder REPL) umsetzen und Ausdrücke interaktiv auswerten lassen:

```
> 1 + 2;;
val it : int = 3
```

Im diesem Beispiel werden zwei Zahlen addiert und das Ergebnis wird zurückgeliefert. In der REPL wird mit zwei Semikolons angezeigt, dass der Ausdruck beendet ist und die Auswertung starten soll (vgl. Syme, Granicz und Cisternino 2015, S. 8). Für jedes Ergebnis gibt die REPL auch den entsprechenden Typ an, hier der Typ `int` für eine Ganzzahl. An den Bezeichner `it` wird jeweils das Ergebnis der letzten Auswertung gebunden, so dass als Folgeausdruck damit weiter gearbeitet werden kann:

```
> it + 10;;
val it : int = 13
```

Folgender Ausdruck gibt eine Zeichenkette auf der Konsole aus:

```
> printf "Hello, world!\n";;
Hello, world!
val it : unit = ()
```

Obschon es hier nicht auf einen Rückgabewert, sondern nur auf den Effekt ankommt, liefert `printf` auch ein Ergebnis, hier vom Typ `unit`, der nur einen Wert, nämlich `()`, annehmen kann. Dieser Typ steht für die Abwesenheit eines Werts (vgl. Carter 2020c, Eintrag "Unit Type"). An diesem Beispiel sieht man auch, wie Funktionen auf ihre Argumente angewendet (appliziert) werden können, es folgt der Name der Funktion sowie mit Leerzeichen getrennt das eine oder mehrere Argumente.[6] So könnte eine Funktion zur Berechnung der Fläche eines Rechtecks wie folgt definiert und aufgerufen werden:

```
> let flaeche x y = x * y;;
val flaeche : x:int -> y:int -> int

> flaeche 10 20;;
val it : int = 200
```

[5] Vgl. für eine Diskussion des Problems und Lösungsmöglichkeiten Syme, Granicz und Cisternino 2015, S. 175 f. und Wlashin 2020, Abschnitt "The 'Dependency cycles' series'".

[6] An dieser Stelle sei schon darauf hin gewiesen, dass Funktionen nur ein Argument entgegennehmen und nur einen Rückgabewert liefern; wie hier dennoch mit mehreren Argumenten umgegangen wird – das sogenannte Currying – wird in Abschnitt 2.5 beschrieben.

Nach dem `let` sieht man anhand der Ausgabe der REPL den vom Compiler ermittelten Typ der Funktion, die nun an den Bezeichner `flaeche` gebunden wurde, hier noch zusätzlich mit der Bezeichnung der Variablen. Lässt man den Bezeichner danach nochmals auswerten, so erhält man:

```
> flaeche;;
val it : (int -> int -> int) = <fun:it@8-1>
```

Der Typ der Funktion bezeichnet somit die Typen der Argumente, getrennt durch Pfeile und dem Typ des Rückgabewertes, das ist der zuletzt angegebene Typ. Hier akzeptiert die Funktion also zwei Argumente vom Typ `int` und liefert auch ein `int` zurück. Alternativ kann man die Typen der Parameter und des Rückgabewertes auch annotieren, die Annotation wird mit einem Doppelpunkt gebildet und sieht wie folgt aus:

```
> let flaeche (x: int) (y: int) : int = x * y;;
val flaeche : x:int -> y:int -> int
```

Die Funktionsapplikation ist linksassoziativ ausgestaltet, d. h. gegeben zwei Funktionen `f` und `g` und einem Parameter `x`, wird

```
> f g x
```

derart ausgewertet, dass zuerst `f` mit `g` appliziert wird und anschließend das Ergebnis mit `x`.[7] Die Auswertungsreihenfolge kann durch Klammerung entsprechend beeinflusst werden, möchte man also zuerst `g` mit `x` aufrufen, so muss der Aufruf

```
> f (g x)
```

lauten. Ohne Klammerung würde er wie bei `(f g) x` ausgewertet (vgl. Wlashin 2020, Abschnitt "Function associativity and composition").

Oben wurde mit Absicht der Begriff "Bindung" verwendet, denn anders als in anderen Sprachen gibt es in dem Sinne keine "Variablen", es gibt Bezeichner, an die man Werte – die wie oben erwähnt ja auch Funktionen sein können – binden kann (vgl. Wlashin 2020, Abschnitt "Function Values and Simple Values" und Erwig 1999, S. 25 f.). Beispielsweise kann man die oben erstelle Funktion auch noch an einen weiteren Bezeichner binden:

```
> let area = flaeche;;
val area : (int -> int -> int)

> area 20 20;;
val it : int = 400
```

[7] Es ist ein beliebter Fehler, eine etwaig notwendige Klammerung zu vergessen. Das Ergebnis ist eine manchmal schwer zuzuordnende Fehlermeldung des Typsystems ("This value is not a function and cannot be applied"), siehe Wlashin 2020, Abschnitt "Troubleshooting F#".

Dies wirft unmittelbar die Frage auf, ob man auch Funktionen gänzlich ohne `let` erzeugen kann und in der Tat geht dies mit dem Schlüsselwort `fun`, wie das folgende Beispiel illustriert:

```
> fun x y -> x * y;;
val it : x:int -> y:int -> int

> (fun x y -> x * y) 9 9;;
val it : int = 81

> let flaeche = fun x y -> x * y;;
val flaeche : x:int -> y:int -> int

> flaeche 10 20;;
val it : int = 200
```

Hierbei passiert bei den vier Aufrufen Folgendes: Beim ersten Aufruf wird eine sogenannte anonyme Funktion (vgl. Erwig 1999, S. 32).[8] erzeugt, allerdings an keinen – genau genommen nur an `it` – Bezeichner gebunden und auch nicht aufgerufen. Danach wird die gleiche Funktion – man beachte die notwendige Klammerung – auf zwei Zahlen angewendet und das Ergebnis der Multiplikation wird auf der REPL zurückgeliefert. Im dritten Aufruf wird die erzeugte Funktion an den Bezeichner `flaeche` gebunden und im vierten Aufruf wie in den obigen Beispielen gezeigt aufgerufen.

Die Bindungen von Werten an Bezeichner erfolgen jeweils in einer Umgebung, entweder der globalen Umgebung oder in Umgebungen, die einen begrenzten Gültigkeitsbereich haben, wie beispielsweise als lokale Bindungen in einer Funktion.[9]

Auf den dargestellten Datentypen sind eine Reihe von Operationen wie Arithmetik, String-Operationen, Auswertungen mit Wahrheitswerten (`not` für die Negation `&&` für ein logisches Und und `||` für ein logisches Oder) etc. möglich. Es steht als Bibliothek der Umfang des gesamten .NET Frameworks zur Verfügung. Für weitere Details zu Operatoren und Funktionen der Standard-Bibliothek von F# sei auf Carter 2020c verwiesen.

Zur Fallunterscheidung existiert ein `if ... then ... else` Ausdruck. Blöcke werden in F# ähnlich wie in Python durch Einrückung umgesetzt. Als Alternative zu dieser Syntax gibt es auch eine ausführlichere Syntax, in der Blöcke mit `begin`, `in`, `end` eingefasst

[8] Ein anderer Ausdruck hierfür ist Lambda-Ausdruck (vgl. Carter 2020c, Abschnitt "Lambda Expressions: The fun keyword").

[9] F# setzt dabei statisches Binden (oder auch lexical scope) der Bezeichner um, d. h. umgangssprachlich gesprochen, kann man durch Lesen des Programmcodes erkennen, welchen Wert ein Bezeichner haben wird. Anders gewendet, die Definitionen beziehen sich immer auf die zum Zeitpunkt der Definition geltende Umgebung. Das Gegenteil wäre dynamisches Binden (oder dynamic scope), wo dies nur zur Laufzeit entschieden werden kann. Statisches Binden ist heute bei Programmiersprachen vorherrschend, dynamisches Binden kommt nur in wenigen Sprachen wie einigen Lisp-Dialekten vor und wird als wenig vorteilhaft angesehen (vgl. Erwig 1999, S. 28, Krishnamurthi 2012, Kapitel 6.4).

werden (vgl. Carter 2020c, Abschnitt "Verbose Syntax"). Rekursive Funktionen, also solche, die sich selbst aufrufen, müssen mit dem Schlüsselwort `rec` gekennzeichnet sein. Hier ein Beispiel einer Funktion, die die n-te Fakultät berechnet:

```
35  let rec fak x  =
36      if x = 0 then 1
37      else x * fak (x - 1)
```

Der Typ dieser Funktion ist `int -> int`. An dieser Stelle soll auf ein weiteres, sehr mächtiges Konstrukt zur Fallunterscheidung hingewiesen werden, das Pattern Matching (vgl. Carter 2020c, Abschnitt "Pattern Matching"). Die oben dargestellte Funktion zur Berechnung der Fakultät sähe beispielsweise mit Pattern Matching wie folgt aus:

```
38  let rec fak n =
39      match n with
40      | 0 -> 1
41      | n -> n * (fak (n - 1))
```

Mittels des `match ... with` Ausdrucks (vgl. Carter 2020c, Abschnitt "Match expressions") wird das Pattern Matching eingeleitet und die einzelnen Muster wie dargestellt mit `| ... ->  ...` unterschieden. Diese Muster können wie im Beispiel einfache Konstanten sein, es können aber auch komplexere Datenstrukturen wie Tupel, Records, Listen etc. als Muster angegeben werden. Nach dem `->` wird der für diesen Fall auszuwertende Ausdruck erwartet. Bei der Auswahl zutreffender Muster kommt es auf die Reihenfolge der Muster an, denn es wird das erste zutreffende Muster ausgewählt. Da `match ... with ...` ein Ausdruck ist, muss er auch zu einem Wert auswerten. Aus diesem Grund muss es immer mindestens ein zutreffendes Muster geben, sogenanntes "exhaustive matching". Im anderen Fall gibt der Compiler eine Warnung ("unvollständige Musterübereinstimmung") aus. Diese kann, wenn man den Ausdruck nicht verbessert, zu einem Laufzeitfehler (Match Failure Exception) führen (vgl. Wlashin 2020, Abschnitt "Match expressions"). Es gibt mit dem Wildcard-Muster, das mit einem Underscore dargestellt wird, die Möglichkeit eines immer zutreffenden Musters:

```
42  > match 42 with
43  -     | 1 -> "Eins"
44  -     | 2 -> "Zwei"
45  -     | _ -> "Eine andere Zahl";;
46  val it : string = "Eine andere Zahl"
```

Pattern Matching kann darüber hinaus unter anderem auch in `let`-Ausdrücken zum Zerlegen von Datenstrukturen verwendet werden, wie im Abschnitt 2.3, S. 10, anhand von Tupeln gezeigt wird.

2.3 Einfache und zusammengesetzte Datentypen

Basistypen

F# bietet eine Reihe von Basis- oder Standardtypen, deren Notation der Syntax von Standard ML (vgl. Milner, Tofte und Harper 1990) entlehnt ist:[10]

Typbezeichner	Beschreibung	Beispielwerte	.NET-Bezeichner
`bool`	Wahrheitswerte	`true`, `false`	`System.Boolean`
`int`	Ganzzahl (32 Bit)	`10`, `0x16`, `0b01010`	`System.Int32`
`float`/`double`	Fließkommazahlen (64-Bit)	`3.1415`, `3.4e10`	`System.Double`
`string`	Zeichenkette (Unicode)	`"abc"`	`System.String`
`unit`	Typ, der nur einen Wert hat	`()`	`Core.Unit`

Obschon für Zeichenketten der Standardtyp `System.String` verwendet wird, kann er in F# nur als unveränderlicher (immutable) Typ verwendet werden. Jedes Auswählen eines Substrings oder Aufteilen des Strings führt zu neuen Zeichenketten, die ursprüngliche wird nicht verändert. Sämtliche Basistypen in F# sind unveränderlich (vgl. Syme, Granicz und Cisternino 2015, S. 10).

Neben diesen Basistypen gibt es eine Reihe von zusammengesetzten Datentypen bzw. selbst erstellbaren Datentypen. Hiervon werden einige im Hinblick auf die Objektorientierung und das Zusammenspiel mit anderen .NET-Sprachen vorgestellt.

Tupel

Mit Tupeln werden Gruppierungen von Werten beschrieben. Die Typen dürfen dabei unterschiedlich sein (vgl. Carter 2020c, Abschnitt "Tuples" und Erwig 1999, S. 21f.). Einige Beispiele für Tupel sind:

```
47  > // Tupel aus Zahlen
48  - (1, 2, 3);;
49  val it : int * int * int = (1, 2, 3)
50
51  > // Tupel gemischt aus Zahlen und Strings
52  - ("eins", 2, "drei");;
53  val it : string * int * string = ("eins", 2, "drei")
54
55  > // Tupel mit Ausdrücken
56  - (1+2, 3*4);;
57  val it : int * int = (3, 12)
```

[10] Die Tabelle wurde sinngemäß übernommen von Syme, Granicz und Cisternino 2015, S. 29, für weitere Basistypen wie unterschiedliche Ganzzahlen, Fließkommazahlen unterschiedlicher Genauigkeit etc., siehe Carter 2020c, Abschnitt "Basic Types".

Zur Notation des Typs werden die jeweiligen Typen mit einem `*` verbunden. Dieser Notation liegt folgender Gedanke zugrunde: Fasst man einen Typ als die Menge seiner Werte auf, so kann man sich den Typ des Tupels als kartesisches Produkt der jeweiligen Typen seiner Komponenten vorstellen (vgl. Erwig 1999, S. 22). Ergänzend sei noch darauf hingewiesen, dass wenn man mehr als ein Resultat zurückliefern möchte, diese in der Regel in einem Tupel zusammenfasst (vgl. Syme, Granicz und Cisternino 2015, S. 556). Hierauf können an der Aufrufstelle mittels Pattern Matching die einzelnen Werte für die Weiterverarbeitung extrahiert werden. Dies kann mittels `match` oder aber auch mit `let` erfolgen (vgl. u. a. Erwig 1999, S. 26):

```
> let (Vorname, Nachname) = ("Hans", "Mustermann");;
val Vorname : string = "Hans"
val Nachname : string = "Mustermann"
```

Tupel erhalten keinen Bezeichner, sondern sie werden ausschließlich über die enthaltenen Typen mittels der `*`-Notation beschrieben.

Records

Mit einem Record-Typ (vgl. Carter 2020c, Abschnitt "Record") können Gruppierungen von Werten mit Bezeichnern umgesetzt werden. Für einen Record-Typ wird ein neuer Typ eingeführt und dieser erhält ebenfalls einen Bezeichner. Neue Typen oder Typerweiterungen werden in F# mit dem Schlüsselwort `type` eingeleitet. Ein Beispiel für eine Person und ein Unternehmen könnte so aussehen:[11]

```
type PersonTyp = {
  Vorname : string
  Nachname : string
  Geburtsdatum : System.DateTime
}

type UnternehmenTyp = {
  Firma : string
  Anschrift: string
}
```

Genutzt werden kann dies wie folgt:[12]

```
> let ada = { Vorname = "Ada"; Nachname = "Lovelace";
-     Geburtsdatum = new System.DateTime(1815,12,10)};;
val ada : PersonTyp = { Vorname = "Ada"
                        Nachname = "Lovelace"
```

[11] Mit `System.DateTime` wird die Standardfunktionalität des .NET Frameworks für Zeit und Datum verwendet.

[12] Das `new` ist an dieser Stelle optional.

```
75                          Geburtsdatum = 10.12.1815 00:00:00 }
76
77    > let siemens = { Firma = "Siemens Aktiengesellschaft";
78      -     Anschrift = "Werner-von-Siemens-Str. 1, 80333 München"};;
79    val siemens : UnternehmenTyp =
80    { Firma = "Siemens Aktiengesellschaft"
81      Anschrift = "Werner-von-Siemens-Str. 1, 80333 München" }
```

Record-Typen bilden eine Situation ab, in der jedes Element vorhanden sein muss, also eine Und-Beziehung, es braucht also Vorname, Nachname und Geburtsdatum, um eine Person dem Beispiel nach korrekt abzubilden. Die Reihenfolge, in der die Elemente angegeben werden, ist jedoch irrelevant.

Vereinigungstypen (Discriminated Unions)

Ein weitere Möglichkeit zur Bildung von Typen sind "Discriminated Unions", diese bilden eine Oder-Beziehung ab. Beispielsweise könnte eine Definition für einen Geschäftspartner wie folgt ausgestaltet sein:

```
82    type Geschaeftspartner =
83        | Person of PersonTyp
84        | Unternehmen of UnternehmenTyp
```

Ein Geschäftspartner ist also entweder eine Person oder ein Unternehmen. Die Bezeichner `Person` und `Unternehmen` im Beispiel werden Discriminators genannt und werden genutzt, um einen entsprechenden Vereinigungstyp zu bilden:

```
85    let hans = Person { Vorname = "Hans"; Nachname = "Mustermann";
86    -     Geburtsdatum = new System.DateTime(1970,05,01)};;
87    val hans : Geschaeftspartner = Person { Vorname = "Hans"
88                                            Nachname = "Mustermann"
89                                            Geburtsdatum = 01.05.1970 00:00:00 }
90
91    > let kiosk = Unternehmen { Firma = "Kleiner Kiosk";
92    -     Anschrift = "Musterstr. 10, Musterhausen"};;
93    val kiosk : Geschaeftspartner =
94      Unternehmen { Firma = "Kleiner Kiosk"
95                    Anschrift = "Musterstr. 10, Musterhausen" }
```

Entsprechend können mittels Pattern Matching die Fälle unterschieden werden:

```
96    let nameGP gp =
97        match gp with
98        | Person p -> p.Vorname + " " + p.Nachname
99        | Unternehmen u -> u.Firma
```

Auch dieses Pattern Matching erfordert, dass sämtliche Fälle behandelt werden, ansonsten wird vom Compiler eine Warnung ausgegeben. Die Funktion angewendet auf die oben gebildeten Beispiele ergibt:

```
> nameGP hans;;
val it : string = "Hans Mustermann"

> nameGP kiosk;;
val it : string = "Kleiner Kiosk"
```

Derartige Discriminated Unions bilden somit Daten ab, bei denen es eine feste, nicht erweiterbare Menge an Alternativen gibt. Der Compiler stellt sicher, dass diese Einschränkung auch jederzeit eingehalten wird (vgl. Syme, Granicz und Cisternino 2015, S. 94).

Die vorgestellten Möglichkeiten mittels Records und Discriminated Unions Daten der realen Welt geschickt in eine typsichere Modellierung zu überführen, wird von Wlaschin 2018 als besondere Vorteil des Typsystems angesehen. Für die Korrektheit eines Programms bietet das Pattern Matching einen Mechanismus, der sicherstellt, dass auch sämtliche auftretende Fälle von der Programmiererin behandelt werden.

Listen

Ein in der funktionalen Programmierung häufig genutzter Datentyp ist die Liste. Listen erlauben auf einfache Weise das Programmieren mit Kollektionen (vgl. Erwig 1999, S. 23). In F# sind diese rekursiv derart definiert, dass eine Liste entweder die leere Liste oder ein Tupel mit dem Listenkopf und der Restliste darstellt. Eine Liste von Ganzzahlen[13] könnte man also wie folgt implementieren:[14]

```
type int list =
  | ([])
  | (::) of int * int list
```

Der `::` Operator[15] bildet aus einem Kopf und einer Restliste eine neue Liste. Listen sind in F# ebenfalls als unveränderliche Datenstrukturen ausgebildet. Einige Beispiele zur Nutzung:

```
> [1;2;3];;
val it : int list = [1; 2; 3]

```

[13] Auf die generische Ausgestaltung von Listen wird im Folgeabschnitt eingegangen. An dieser Stelle kann schon so viel gesagt werden, dass Listen in F# polymorph ausgestaltet sind, also Funktionen auf Listen für beliebige zugrunde liegende Typen gebildet werden können, siehe Erwig 1999, S. 25.

[14] Siehe auch für den generisch ausgebildeten Fall weiter unten Abschnitt 2.4, S. 14 (vgl. Syme, Granicz und Cisternino 2015, S. 93).

[15] In Lisp würde man diesen `cons` nennen.

```
> 1 :: [2;3];;
val it : int list = [1; 2; 3]

> 1 :: [];;
val it : int list = [1]
```

Listen in F# sind homogen, also von einem Datentyp. Möchte man mehrere Datentypen mischen, so bieten sich bspw. wie oben skizzierten Discriminated Unions an, das Beispiel verwendet die oben eingeführten Typen zu Geschäftspartnern und führt zu einer Liste, in der alle Elemente vom Typ Geschäftspartner sein müssen:

```
> [hans; kiosk];;
val it : Geschaeftspartner list =
  [Person { Vorname = "Hans"
            Nachname = "Mustermann"
            Geburtsdatum = 01.05.1970 00:00:00 };
   Unternehmen { Firma = "Kleiner Kiosk"
                 Anschrift = "Musterstr. 10, Musterhausen" }]
```

2.4 Typsystem und Typinferenz

F# besitzt ein statisches Typsystem und die Ersteller der Sprache versprechen, dass die Typsicherheit zur Kompilierzeit sichergestellt werden kann. Soweit es um die Korrektheit von Programmen geht, bieten gerade solche statischen Typsysteme die Chance, logische Fehler automatisch aufzufinden (vgl. Steimann 2019, S. 148). Typfehler zur Laufzeit sollen somit nicht mehr auftreten können. Darüberhinaus arbeitet das Typsystem mit sogenannter Typinferenz (vgl. Carter 2020c, Abschnitt "Type Inference"). Das bedeutet, dass der Compiler die Typen durch Analyse der Aufrufe und Rückgabewerte zur Kompilierzeit zu bestimmen versucht. Dabei analysiert er den Programmcode von oben nach unten, von links nach rechts und von innen nach außen (vgl. Syme, Granicz und Cisternino 2015, S. 11). Somit sollen Typannotationen nur in wenigen Fällen nötig sein. Der resultierende Code, so das Versprechen, wird knapper.

Das Typsystem von F# ist polymorph ausgestaltet. Das bedeutet, dass Funktionen für eine ganze Klasse von Typen definiert werden können. Somit müssen bestimmte Funktionen nur einmal definiert werden und können für Werte beliebigen Typs eingesetzt werden (vgl. Syme, Granicz und Cisternino 2015, S. 95 f. Erwig 1999, S. 38).

So ist beispielsweise der Datentyp der Liste polymorph ausgestaltet, also ist eine Liste immer eine Liste eines bestimmten zugrunde liegenden Typs. Folglich kann die Implementierung für eine Liste aus Abschnitt 2.3, Seite 13, polymorph wie folgt geschrieben werden, dabei bezeichnet `'T` einen Typparameter (vgl. Syme, Granicz und Cisternino 2015, S. 93):

```
type 'T list =
  | ([])
  | (::) of 'T * 'T list
```

Typparameter werden erst zu einem späteren Zeitpunkt mit einem konkreten Wert, also einem konkreten Typ, belegt (vgl. Steimann 2019, S. 182). Bei der Typinferenz versucht der Compiler einen möglichst allgemeinen Typ zu bestimmen. So kann die folgende Funktion, die ein Tupel aus dem übergebenen Parameter erzeugt, mit jedem Typ (erkennbar an der Nutzung der Typvariable `'a`) aufgerufen werden:

```
> let f x = (x, x);;
val f : x:'a -> 'a * 'a
```

Der Vorteil der so genannten Automatic Generalization (vgl. Carter 2020c, Abschnitt "Automatic Generalization") ist, dass die Typparameter nicht händisch eingefügt werden müssen, sondern der Compiler diese automatisch erkennt. Somit erspart sich die Programmiererin auch hier eine Typannotation (vgl. Syme, Granicz und Cisternino 2015, S. 11).

Dabei versucht F# sich am Typsystem von .NET so weit es geht zu orientieren. Ein bestimmter Typ Ganzzahlen in .NET ist somit der gleiche Typ in F#. Dies vereinfacht die Interoperationalität (vgl. Syme 2020, S. 18). Die Bezeichnung der meisten Typen passt sich allerdings stärker den ML-orientierten Sprachen an als der Bezeichnung des .NET-Typsystems.[16] Während in funktionalen Sprachen normalerweise Typen bestimmte Werte haben und durch Funktionen Werte eines Typs zu Werten eines anderen Typs ausgewertet werden, umfasst das Typsystem von .NET noch einige weitergehende Aspekte (vgl. Syme, Granicz und Cisternino 2015, S. 108 f.): So unterscheidet .NET zwischen Typen mit Wertsemantik und solchen mit Referenzsemantik. F# übernimmt darüberhinaus von .NET das Subtyping. Des Weiteren gibt es noch Typen für die sogenannten Delegates, Attribute, Exceptions und Enums (vgl. Syme, Granicz und Cisternino 2015, S. 109).

Beim Subtyping erlaubt F# auch explizite Typumwandlungen. Dabei werden Typumwandlungen zu Supertypen (Up casts mit dem Operator `:>`) vom Compiler zur Kompilierzeit geprüft. Typumwandlungen zu Subtypen (Down casts mit dem Operator `:?>`) können demgegenüber nur zur Laufzeit geprüft werden. Der Grund hierfür liegt darin, dass an jeder Stelle, wo ein bestimmter Typ gefordert ist, auch ein Subtyp davon auftauchen darf. Der Compiler kann zur Kompilierzeit noch gar nicht entscheiden, welcher Typ an der Umwandlungsstelle tatsächlich auftritt. Erfolgt beim Down cast eine unzulässige Umwandlung, wird eine `System.InvalidCastException` geworfen und zwar zum Zeitpunkt der versuchten Typumwandlung (vgl. Syme, Granicz und Cisternino 2015, S. 110–111). Siehe hierzu auch das folgende Codebeispiel, hier wird `x` zuerst zu `System.Object`[17]

[16] Vgl. die Tabelle auf Seite 10 sowie für eine Übersicht Syme, Granicz und Cisternino 2015, S. 559 und Milner, Tofte und Harper 1990.

[17] In F# wird dieser Typ auch mit `obj` bezeichnet (vgl. Syme, Granicz und Cisternino 2015, S. 108).

umgewandelt und das Ergebnis `y` zugewiesen, anschließend wieder nach `int` und anschließend wird probiert, `y` zu `System.String` umzuwandeln, was zum Werfen der genannten Ausnahme führt.

```
> let x = 1;;
val x : int = 1

> let y = x :> obj;;
val y : obj = 1

> let z = y :?> int;;
val z : int = 1

let i = y :?> string;;

> let i = y :?> string;;
System.InvalidCastException: Unable to cast object of type 'System.Int32'
to type 'System.String'.
    at <StartupCode$FSI_0016>.$FSI_0016.main@()
```

Einige Operatoren insbesondere aus dem mathematischen Bereich erlauben auch unterschiedliche Typen.[18] Dennoch müssen diese einheitlich verwendet werden. So kann man mit `+` zwar Ganzzahlen und Fließkommazahlen addieren, jedoch darf man diese nicht in einem Aufruf mischen, wie das folgende Beispiel zeigt:

```
> 1 + 2.0;;

  1 + 2.0;;
  ----^^^

  error FS0001: Der Typ "float" stimmt nicht mit dem Typ "int" überein.
```

Man kann sich dies auch daran klar machen, dass jeder Infix-Operator wie `+` auch als Funktionsaufruf in der Präfix-Notation dargestellt werden kann: `(+) 1 2`. Hierbei sind die Klammern eine Syntaxvorgabe, um die Funktion des Operators direkt zu referenzieren.[19]

[18] Dies ist unterschiedlich zu bspw. OCaml, bei denen es eigene Operatoren für die verschiedenen Typen gibt oder bei Haskell, wo das Überladen der Operatoren mit sogenannten Typklassen – zugegebenermaßen ist dies ein etwas unglücklich gewählter Begriff, da die Typklassen von Haskell nichts mit Klassen der objektorientierten Programmierung gemein haben – realisiert wird. In F# wird dieses Überladen durch Typparameter, die zur Kompilierzeit aufgelöst werden, realisiert (vgl. Syme 2020, S. 27 f.).

[19] Analog kann man auch eigene Infix-Operatoren bilden, wie in `let (++) x y = List.append x [y]`, siehe Syme, Granicz und Cisternino 2015, S. 129 f.

2.5 Funktionen und Funktionen höherer Ordnung

Wie oben erwähnt, sind auch Funktionen Werte. Somit ist es möglich, dass Funktionen als Argumente akzeptiert und auch als Rückgabewerte zurückgegeben werden. Solche Funktionen bezeichnet man als Funktionen höherer Ordnung (vgl. Erwig 1999, S. 47 und Bird und Wadler 1992, S. 35).

Wie in der Fußnote 6 auf Seite 6 angedeutet, haben Funktionen in F# eigentlich genau ein Argument und liefern auch genau einen Rückgabewert zurück (vgl. Wlashin 2020, Abschnitt "Currying" und Erwig 1999, S. 32). Nun gibt es Funktionen wie die oben eingeführte Funktion `flaeche`, die mehrere Argumente akzeptieren. Tatsächlich befinden sich dahinter mehrere mit einander kombinierte Funktionen, die jeweils nur einen Parameter erlauben. Dies kann man auch umgekehrt nutzen: So bilden diese Funktionen, wenn man ihnen nur ein Argument übergibt, eine neue Funktion, die ein Argument weniger verlangt und auf die restlichen Argumente anwendbar ist. Diese Technik bezeichnet man nach dem Logiker Haskell Curry als Currying (vgl. Erwig 1999, S. 50 mit dem Hinweis, dass diese Notation eigentlich zuerst von dem deutschen Mathematiker Schönfinkel erfunden wurde, daher auch als Schönfinkeln bezeichnet werden kann. Diese Bezeichnung hat sich allerdings nicht durchgesetzt). Damit wird das zuerst angegebene Argument sozusagen in die resultierende Funktion "eingebacken"[20]. Technisch gesprochen erlaubt F# die sogenannte partielle Auswertung (vgl. Syme, Granicz und Cisternino 2015, S. 47). Die Funktion `flaeche` von Seite 6 lässt sich daher auch so nutzen:

```
> let k10 = flaeche 10;;
val k10 : (int -> int)

> k10 1;;
val it : int = 10

> k10 99;;
val it : int = 990
```

Dies liest sich so, dass im ersten Schritt an `k10` eine neue Funktion gebunden wird, nämlich die Funktion `flaeche`, in der der erste Parameter mit `10` eingebacken wurde. Die Ursprungsfunktion wurde also partiell ausgewertet. Mit `k10` kann anschließend wie in der Folge ersichtlich weiter gearbeitet werden.

Entsprechend liest sich auch der Funktionstyp von `flaeche`, also `(int -> int -> int)` alternativ so: Die Funktion akzeptiert eine Ganzzahl und liefert eine Funktion zurück, die eine Ganzzahl als Argument akzeptiert und eine Ganzzahl als Ergebnis zurückliefert.

[20] Ein Ausdruck, den ich zuerst bei Scott Wlaschin gelesen habe und den ich für eine gelungene Bezeichnung halte (vgl. Wlashin 2020, Abschnitt "Partial application").

Die Funktion `List.map` ist ein Beispiel für eine Funktion, die eine Funktion als Argument akzeptiert. Sie nimmt eine Funktion und wendet diese auf alle Elemente einer Liste an. Rückgabewert ist die Liste mit den Rückgabewerten der jeweiligen Funktionsaufrufe. Folgendes Beispiel illustriert die Nutzung, hier werden sämtliche Ganzzahlen einer Liste quadriert:

```
157   > List.map (fun x -> x * x) [1; 2; 3; 4];;
158   val it : int list = [1; 4; 9; 16]
```

Der Typ der Funktion `List.map` lautet `(('a -> 'b) -> 'a list -> 'b list)`. Hieran sieht man, das `List.map` von der polymorphen Definition der Listen profitiert, denn die Funktion muss nur einmal für alle möglichen Arten von Listen geschrieben werden. Eine mögliche Implementierung – wieder unter Nutzung des Pattern Matching – könnte wie folgt aussehen:

```
159   let rec mymap f lst  =
160     match lst with
161     | [] -> []
162     | x :: xs -> f x :: mymap f xs
```

3 Nicht-funktionale Eigenschaften

F# ist wie erwähnt keine "reine" funktionale Sprache, sondern bietet auch nicht-funktionale Sprachkonstrukte. Hierzu gehören Seiteneffekte, veränderbare Datenstrukturen, Objektorientierung, Ausnahmebehandlung, imperative Schleifenkonstrukte. Im Folgenden werden veränderbare Datenstrukturen sowie die Objektorientierung kurz vorgestellt, um anschließend das Zusammenspiel mit einem C#-Programm zu illustrieren.

3.1 Veränderbare Datenstrukturen

Im Standard ist jede Bindung eines Symbols an einen Wert unveränderbar (immutable). Gelegentlich mag es allerdings erforderlich sein, den einmal zugewiesenen Wert nachträglich zu ändern. Hierzu muss bei der Definition das Schlüsselwort `mutable` hinzugefügt werden, für die Zuweisung wird der Operator `<-` verwendet. Bei Zuweisungen prüft F#, ob der zugewiesen Wert auch vom korrekten Typ ist, also zuweisungskompatibel ist. Im Folgenden ein Beispiel hierzu:

```
163     > let mutable f = 10;;
164     val mutable f : int = 10
165
166     > f <- 20;;
167     val it : unit = ()
168
```

```
> f <- "Hello!";;

f <- "Hello!";;
-----^^^^^^^^

stdin(6,6): error FS0001: Dieser Ausdruck sollte den folgenden Typ
aufweisen:
    "int"
Er ist jedoch vom folgenden Typ:
    "string"
```

Der Variable `f` wird zu Beginn der Wert `10` zugewiesen, anschließend mit dem Zuweisungsoperator die Zahl `20`. Das Ergebnis der Zuweisung ist der Wert `()` vom Typ `unit`.

3.2 Objektorientierung

Im Folgenden wird das obige Beispiel der Personen und Unternehmen als Objekte umgesetzt. Hierbei wird der Empfehlung gefolgt, möglichst auf Implementierungsvererbung zu verzichten und statt dessen wird ein Interface für die Ausgabe des Namens genutzt. Dieses kann wie folgt angegeben werden (vgl. Syme, Granicz und Cisternino 2015, S. 136-139 und Wlashin 2020, Abschnitt "Interfaces"):

```
type IGeschaeftspartner =
    abstract member Name: unit -> string
```

Auch für die Objektorientierung werden alle neuen (abstrakten) Klassen, Interfaces mit `type` definiert. Bei Interfaces werden die Typen der Eigenschaften (entweder Felder oder Methoden) wie dargestellt angegeben. Die Klassen für die Personen und Unternehmen sehen wie folgt aus:

```
type Person(vorname: string, nachname: string,
            geburtsdatum: System.DateTime) =
    member this.Vorname = vorname
    member this.Nachname = nachname
    member this.Geburtsdatum = geburtsdatum

    interface IGeschaeftspartner with
        member this.Name() =
            this.Vorname + " " + this.Nachname

type Unternehmen(firma: string, anschrift: string) =
    member this.Firma = firma
    member this.Anschrift = anschrift
```

```
194
195         interface IGeschaeftspartner with
196             member this.Name() =
197                 this.Firma
```

Im Tupel nach der Bezeichnung der Klasse können die Argumente für den Konstruktor angegeben werden, in diesem Fall wurde auch eine Typannotation eingefügt, um sicherzustellen, dass wirklich Zeichenketten und ein Datum übergeben werden. Die Eigenschaften werden dem Standardverhalten von F# folgend als unveränderliche Felder umgesetzt (vgl. Syme, Granicz und Cisternino 2015, S. 125-128, Wlashin 2020, Abschnitt "Classes"). Dabei hat das `this` die Bedeutung wie "self" in anderen Sprachen und bezeichnet das Empfangsobjekt, also die aktuelle Instanz. In F# kann die Programmiererin diesen Bezeichner frei wählen, er kann also auch `self` oder `x` lauten. Der Grund für diese Entscheidung war, dass man in geschachtelten Konstrukten dieser Art leichter die Übersicht behalten kann, wenn man den Parameter frei wählen kann (vgl. Syme 2020, S. 25).

Ähnlich wie in C# können die Methoden mit entsprechenden Schlüsselworten als statisch gebunden, abstrakt, überschrieben, überschreibbar deklariert werden. Ebenso erfolgt die Vererbung von Basisklassen in identischer Weise. Bei der Programmierung mit Interfaces gibt es jedoch einen Unterschied: F# erfordert wie im obigen Beispiel illustriert zwingend die explizite Interface-Implementierung (vgl. Steimann 2019, S. 262 f. und Wagner 2020, Abschnitt "Explicit Interface Implementation"). Eigenschaften von Objekten können ebenfalls als `mutable` definiert werden, um eine Veränderung des Zustands von Objekten umzusetzen (vgl. Syme, Granicz und Cisternino 2015, S. 133 f.). Nun können dem Beispiel folgend Objekte instanziiert werden:

```
198   > let ada = Person("Ada", "Lovelace", System.DateTime(1815,12,10))
199   val it : Person = FSI_0005+Person {Geburtsdatum = 10.12.1815 00:00:00;
200                                      Nachname = "Lovelace";
201                                      Vorname = "Ada";}
202
203   > let siemens = Unternehmen("Siemens Aktiengesellschaft",
204                         "Werner-von-Siemens-Str. 1, 80333 München")
205   val it : Unternehmen =
206     FSI_0005+Unternehmen
207       {Anschrift = "Werner-von-Siemens-Str. 1, 80333 München";
208        Firma = "Siemens Aktiengesellschaft";}
```

Etwas überraschend ist, dass die Nutzung von `Name` fehlzuschlagen scheint:[21]

```
209   > ada.Name();;
210
```

[21] Vgl. für diese Darstellung und Vorgehensweise Wlashin 2020, Abschnitt "Interfaces".

```
oop.fsx(111,5): error FS0039: Der Typ "Person" definiert nicht das
Feld, den Konstruktor oder den Member "Name"
```

Die Lösung ist, dass F# an dieser Stelle wirklich den Interface-Typ erwartet, es ist also ein Up cast zum Interface nötig. Dies dürfte in der Praxis an den meisten Stellen jedoch kein großes Problem darstellen, wenn als Typ für die Parameter von Funktionen direkt der Interface-Typ gefordert wird, wie die folgende Interaktion zeigt:

```
> // Up cast zum Interface-Typen
(ada :> IGeschaeftspartner).Name();;
val it : string = "Ada Lovelace"

> // Funktion zur Ausgabe des Namens, die Typannotation ist hier
// notwendig, da mehrere Klassen/Interfaces die Methode Name
// definieren/implementieren können.
let printName (gp: IGeschaeftspartner) =
    printfn "Name des Geschäftspartners: %s" (gp.Name())
val printName : gp:IGeschaeftspartner -> unit

> printName ada;;
Name des Geschäftspartners: Ada Lovelace
val it : unit = ()

> printName siemens;;
Name des Geschäftspartners: Siemens Aktiengesellschaft
val it : unit = ()
```

Einige Eigenschaften der Objektorientierung hat F# jedoch nicht umgesetzt: So erlaubt sie keinen Zugriffsmodifizierer `protected`. Hintergrund ist, dass F# den Programmiererinnen einen Stil hin zu Schnittstellenvererbung und Delegation anstelle von Implementierungsvererbung empfiehlt.[22] Unter dieser Prämisse würden `protected` deklarierte Methoden oder Felder eher zu mehr Implementierungsvererbung führen (vgl. Syme 2020, S. 26).

4 Zusammenwirken mit anderen .NET-Sprachen

Eine Funktion in F# wird vom Compiler als statische Methode einer Klasse umgesetzt. Diese Klasse erhält als Namen entweder den Namen des Moduls, in dem sie definiert wurde oder einen von F# automatisch erzeugten.[23] Somit kann eine F#-Funktion von

[22] Vgl. für dieses Muster Steimann und Keller 2020, S. 112 sowie die Empfehlung bei Syme, Granicz und Cisternino 2015, S. 553f. oder Carter 2020a.

[23] Um eine Funktion auch "wiederzufinden" sollten diese bei der Entwicklung immer in Module strukturiert werden.

jeder anderen .NET-Sprache aus aufgerufen werden. Ebenso ist sie via Reflection auch introspektierbar (vgl. Syme 2020, S. 18, Syme, Granicz und Cisternino 2015, S. 481–486). An einem einfachen Beispiel soll nun einerseits das Zusammenspiel mit anderen .NET-Sprachen gezeigt werden. Gegeben das folgende F#-Programm:

```
231  namespace B
232
233    type B() =
234
235        let mutable i = 0
236        member this.getI() = i
237        member this.add(x) = i <- i + x
238
239    module Fak =
240
241        let rec fak n =
242            match n with
243            | 0 -> 1
244            | n -> n * (fak (n - 1))
245
246        [<EntryPoint>]
247        let Main args =
248            printfn "Die Fakultät von 6 ist: %d" (fak 6)
249            0
```

Der vorstehende Code ist wie folgt zu verstehen: Die Namespaces in F# entsprechen denen von .NET (vgl. Syme, Granicz und Cisternino 2015, S. 167f.). Darunter kann der Programmcode in Module aufgeteilt werden. Diese kompilieren jeweils zu einer statischen Klasse (vgl. Wlashin 2020, Abschnitt "Organizing functions"). Die mit dem Attribut `[<EntryPoint>]` annotierte Funktion stellt den Einsprungspunkt in das Programm dar. Die Klassendefinition für `B` findet sich auf der Ebene des Namespaces, würde sie auch in einem Modul definiert, so landete sie als innere Klasse in der entstehenden Klasse für das Modul. Dies ist nicht immer wünschenswert, daher wird geraten, Klassen nicht in Modulen zu definieren (vgl. Wlashin 2020, Abschnitt "Classes").

In der im Folgenden in C# programmierten Klasse `A` wird eine ähnliche Klasse gebildet und in der `Main`-Methode die oben beschriebene Fakultätsfunktion aufgerufen:[24]

```
250  using System;
251  using B;
252
253  namespace A {
254
```

[24] Das `open` Schlüsselwort verhält sich ähnlich wie das `using` von C#, damit kann man direkt auf Bezeichner in Namespaces zugreifen ohne diese immer mit angeben zu müssen.

```
255     public class A {
256       private int i = 0;
257
258       public void add(int x) {
259         i = i + x;
260       }
261
262       public int getI() {
263         return i;
264       }
265
266       static void Main(string[] args) {
267         Console.WriteLine("Die Fakultät von 5 ist: " + Fak.fak(5));
268       }
269     }
270   }
```

Die Nutzung kann man sich im F# Interactive wie folgt vorstellen:

```
271   #r "A/bin/Debug/netcoreapp3.1/A.dll";;
272   #r "B/bin/Debug/netcoreapp3.1/B.dll";;
273
274   open A;;
275   open B;;
276
277   > let a = A();;
278   val a : A.A
279
280   > let b = B()
281   val b : B.B
282
283   > a.add 10;;
284   val it : unit = ()
285
286   > b.add 20;;
287   val it : unit = ()
288
289   > a.getI ();;
290   val it : int = 10
291
292   > b.getI ();;
293   val it : int = 20
294
295   > Fak.fak 5;;
```

```
296   val it : int = 120
```

Man sieht, es spielt für die Programmiererin keine Rolle, ob sie den C#- oder den F#-Code aufruft. Ebenso, wenn man das Programm `A` von der Kommandozeile aus startet, so erhält man auch dort ohne weitere Dinge zu beachten das Ergebnis der Fakultätsfunktion:

```
297   $ dotnet run
298   Die Fakultät von 5 ist: 120
```

Einige Eigenheiten an der Schnittstelle zu anderen .NET-Bibliotheken und -Sprachen sollen abschließend noch vorgestellt werden:

Code aus den Standard-F#-Modulen erzeugen niemals Null-Werte und die Programmiererin sollte dies auch tunlichst vermeiden, um möglichst viel Nutzen aus dem Typsystem zu ziehen.[25] In der Interaktion mit anderen .NET-Sprachen ist die Verwendung zum Teil allerdings unvermeidlich (vgl. Syme, Granicz und Cisternino 2015, S. 159).[26] Ein simples Beispiel hierzu ist:

```
299     let gross (s: string) =
300     -     s.ToUpper();;
301     val gross : s:string -> string
302
303     > gross "abc";;
304     val it : string = "ABC"
305
306     > gross null;;
307     System.NullReferenceException: Object reference not set to an instance
308        of an object.
309        at <StartupCode$FSI_0042>.$FSI_0042.main@()
310     Aufgrund eines Fehlers beendet
```

Hier erhält die Funktion `gross` durch die Typannotation einen Parameter vom Typ `string`. Dieser entspricht dem Zeichenkettentyp des .NET Frameworks, also `System.String`. Dieser wird in der Klassenhierarchie abgebildet, d. h. dass der formale Parameter `s` jeweils den Typ `System.String` oder jeden Subtyp und auch `null` annehmen darf. Die Funktion selbst nutzt die Methode `ToUpper`, um sämtliche Zeichen in Großbuchstaben zu wandeln. Da nun nicht der Basistyp verwendet wird, ist ein Aufruf der Funktion auch mit `null` erlaubt, was folgerichtig zu einer Null Reference Exception führt, da das Empfängerobjekt `null` mit dem Methodenaufruf nichts anzufangen weiß.

[25] Durch den Verzicht auf Null-Werte können Laufzeitfehler wie im folgenden Beispiel vermieden werden. Vgl. allgemein zu den Null-Werten Carter 2020c, Abschnitt “Null Values“, Wlashin 2020, Abschnitt “The Option type“.

[26] Das war seinerzeit eine Designentscheidung, Null-Werte zuzulassen (vgl. Syme 2020, S. 20). Der Hintergrund von Null-Werten verbunden mit den Problemen, die damit auftreten können, beschreibt Tony Hoare, der sich hierfür verantwortlich sieht, in Hoare 2011.

Als weiteren Typ für eine Kollektion kann die Programmiererin Arrays verwenden. Diese werden durch die Arrays aus `System.Array` des .NET Frameworks umgesetzt. Eine etwas unerwartete Einschränkung dabei ist, dass sie in F# veränderbar (mutable) sind. Man hätte erwartet, dass auch diese durch eine rein funktionale Implementierung umgesetzt werden. Da sämtliche anderen reinen F#-Kollektionen als unveränderbare Datentypen umgesetzt sind, muss bei der Programmierung mit Arrays auf diese Ausnahme geachtet werden. Im Fall der Fälle kann es sein, dass die Programmiererin dachte, sie erhält bei einer Operation eine neue Kopie eines Arrays, aber statt dessen wird das ursprüngliche Array an der Aufrufstelle modifiziert. Seinen Grund findet dies darin, dass Arrays speziell für rechenintensive Anwendung genutzt werden können (vgl. Syme, Granicz und Cisternino 2015, S. 62).

Für alle Bibliotheken, die auch von nicht F#-Programmiererinnen genutzt werden, wird empfohlen, dem .NET Library Design Guidelines zu folgen und auf F#-spezifische Idiome zu verzichten bzw. durch andere Sprachkonstrukte des .NET Frameworks zu ersetzen (vgl. Carter 2020b). Dies umfasst beispielsweise den Listen-Typ von F#, aber auch den Funktionstyp. Insbesondere sollte der F#-Datentyp für Listen in den öffentlichen Schnittstellen nicht verwendet werden, statt dessen eher Arrays oder Implementierungen für `IEnumerable<T>`. Auf den Funktionstyp von F# sollte man zugunsten von Delegates ebenfalls verzichten (vgl. Syme, Granicz und Cisternino 2015, S. 542 f.). Da Methodenaufrufe in .NET kein Currying erlauben, müssen von F# aus immer sämtliche Argumente in einem Tupel übergeben werden.

5 Ergebnisse und Ausblick

F# zeigt sich als funktionale Sprache mit nicht-funktionalen Sprachelementen. Diese erlauben Seiteneffekte einfach umzusetzen und darüber hinaus wird durch die Objektorientierung das Zusammenspiel mit den anderen .NET Programmiersprachen erst sinnvoll ermöglicht.[27]

Allerdings kann das Typsystem und der Compiler von F# typische Programmierprobleme wie Null-Objekt-Referenzen, Probleme veränderbare Arrays nicht erkennen und damit fällt die Stärke des Systems hinter dem von stärker funktional ausgerichteten Sprachen wie Haskell zurück. Für die praktische Programmierarbeit bedeutet dies, dass man, um die oben dargestellten Eigenschaften und Techniken nutzen zu können, diese Umstände insbesondere bei der Nutzung der .NET-Standardbibliothek oder Bibliotheken anderer .NET-Sprachen beachtet und entsprechend programmiert werden muss.

Diese Einschränkung trifft auch auf die Programmierung mit Seiteneffekten zu. Wird deren Code nicht an eine eigene Stelle lokalisiert, kann es sein, dass Seiteneffekte auftreten,

[27] Außerdem bleiben der Einsteigerin schwer verständliche Konstrukte wie Monaden für Ein-/Ausgabe bei Haskell erspart.

die nicht intendiert waren. Um diesen Umstand muss man sich immer bei Programmiersprachen mit Seiteneffekten, imperativen Konstrukten bemühen. Er findet hier jedoch besondere Erwähnung, da ein Versprechen der funktionalen Programmierung darin besteht, dass die sogenannte referentielle Transparenz eingehalten wird (vgl. Erwig 1999, S. 8).

Vorgenannte Sprachkonstrukte erfordern aktives Beachten seitens der Programmiererin. Will man sich hierum nicht kümmern, so muss man den Blick zu "reinen" funktionalen Sprachen wie Haskell wenden. Dort erkauft man sich die Reinheit mit zum Teil für die Anfängerin nicht leicht zu durchschauenden Sprachkonstrukten.[28]

Herausfordernd war in der Vergangenheit die parallele Sprachentwicklung von C# und F#. So wurden beispielsweise zu ähnlicher Zeit leicht unterschiedliche Konzepte für Nebenläufigkeit in beiden Sprachen umgesetzt. In der Folge tauchten manche neu eingeführte Elemente der einen mit etwas Verzug in der anderen Sprache auf (vgl. Syme 2020, S. 51f.). Aus dieser Erfahrung heraus wird die weitere Entwicklung sich wohl an der von C# und der .NET-Ausführungsumgebung anlehnen, um miteinander nicht gut harmonierende Sprachelemente zu vermeiden (vgl. Syme 2020, S. 52).

Dennoch, im Ergebnis ist es gelungen, eine quelloffene, plattformübergreifende, funktionale Programmiersprache, mit einem großem Umfang an Sprachelementen, für das .NET Framework zu etablieren. Mit der Entscheidung, auch imperative und objektorientierte Sprachkonstrukte zuzulassen, steht der Programmiererin die ganze Bandbreite an Bibliotheken des .NET Frameworks und was dafür entwickelt wurde[29] zur Verfügung. Somit bietet F# eine weitere Möglichkeit, Programme und Bibliotheken für das .NET Framework zu entwickeln und dabei pragmatisch Eigenschaften funktionaler Programmierung zu nutzen.

[28] Ausdruck hierfür ist schon alleine die unzählige Schar an Einführungen zum Programmieren mit Monaden. Ein einordnender Beitrag hierzu ist Petricek 2018.
[29] So erwähnt Syme 2020, S. 45f. mehr als 120 000 Pakete auf dem Paketrepository NuGet.

A Literaturverzeichnis

Bird, Richard und Philip Wadler (1992). *Einführung in die funktionale Programmierung*. Hanser-Studienbücher der Informatik. München: Hanser [u.a.] 284 S. ISBN: 978-3-446-17001-8.

Carter, Phillip (2020a). *F# Coding Conventions*. URL: https://docs.microsoft.com/en-us/dotnet/fsharp/style-guide/conventions (besucht am 12. 07. 2020).

– (2020b). *F# Style Guide*. URL: https://docs.microsoft.com/en-us/dotnet/fsharp/style-guide/ (besucht am 12. 07. 2020).

– (2020c). *Language Reference - F#*. URL: https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/ (besucht am 01. 07. 2020).

– (2020d). *What Is F#*. URL: https://docs.microsoft.com/en-us/dotnet/fsharp/what-is-fsharp (besucht am 07. 07. 2020).

Erwig, Martin (1999). *Grundlagen funktionaler Programmierung*. München: Oldenbourg. 192 S. ISBN: 978-3-486-25100-5.

Hoare, Tony (2011). *Null References: The Billion Dollar Mistake*. URL: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/ (besucht am 10. 07. 2020).

Krishnamurthi, Shriram (2012). *Programming Languages: Application and Interpretation*. Second Edition. URL: http://cs.brown.edu/courses/cs173/2012/book/ (besucht am 11. 07. 2020).

Lander, Richard (6. Mai 2019). *Introducing .NET 5*. URL: https://devblogs.microsoft.com/dotnet/introducing-net-5/ (besucht am 15. 07. 2020).

Leroy, Xavier (2020). *A History of OCaml – OCaml*. URL: https://ocaml.org/learn/history.html (besucht am 16. 07. 2020).

MacQueen, David, Robert Harper und John Reppy (2020). „The History of Standard ML“. In: *Proceedings of the ACM on Programming Languages* 4 (HOPL), S. 1–100. ISSN: 2475-1421, 2475-1421. DOI: 10.1145/3386336. URL: https://dl.acm.org/doi/10.1145/3386336 (besucht am 07. 07. 2020).

Microsoft (2020a). *.NET Core Overview*. URL: https://docs.microsoft.com/en-us/dotnet/core/about (besucht am 16. 07. 2020).

– (2020b). *.NET Programming Languages | C#, F#, and Visual Basic*. URL: https://dotnet.microsoft.com/languages (besucht am 15. 07. 2020).

– (2020c). *Common Language Runtime (CLR) Overview - .NET Framework*. URL: https://docs.microsoft.com/en-us/dotnet/standard/clr (besucht am 16. 07. 2020).

– (2020d). *What Is Managed Code?* URL: https://docs.microsoft.com/en-us/dotnet/standard/managed-code (besucht am 16. 07. 2020).

Milner, R., Mads Tofte und Robert Harper (1990). *The Definition of Standard ML*. Cambridge, Mass: MIT Press. 101 S. ISBN: 978-0-262-13255-8 978-0-262-63132-7.

Petricek, Tomas (2018). „What We Talk about When We Talk about Monads“. In: *The Art, Science, and Engineering of Programming* 2.3, S. 12. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2018/2/12. URL: http://programming-journal.org/2018/2/12 (besucht am 08. 07. 2020).

Steimann, Friedrich (2019). *Objektorientierte Programmierung*. Kurs 01814, Version 1w. Hagen: FernUniversität in Hagen. 339 S.

Steimann, Friedrich und Daniela Keller (2020). *Moderne Programmiertechniken und -methoden*. Kurs 01853, Version w. Hagen: FernUniversität in Hagen. 263 S.

Syme, Don (2020). „The Early History of F#“. In: *Proceedings of the ACM on Programming Languages* 4 (HOPL), S. 1–58. ISSN: 2475-1421, 2475-1421. DOI: 10.1145/3386325. URL: https://dl.acm.org/doi/10.1145/3386325 (besucht am 06. 07. 2020).

Syme, Don, Adam Granicz und Antonio Cisternino (2015). *Expert F# 4.0*. Fourth edition. Books for Professionals by Professionals. New York, NY: Apress. 582 S. ISBN: 978-1-4842-0741-3.

The Standard ML Language Family (2020). The Standard ML Language Family. URL: https://github.com/SMLFamily/The-Definition-of-Standard-ML-Revised (besucht am 07. 07. 2020).

Wagner, Bill (2020). *C# Programming Guide*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/ (besucht am 16. 07. 2020).

Wlaschin, Scott (2018). *Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#*. Version: P1.0. The Pragmatic Programmers. Raleigh, North Carolina: The Pragmatic Bookshelf. 290 S. ISBN: 978-1-68050-254-1.

Wlashin, Scott (2020). *F# for Fun and Profit*. URL: https://fsharpforfunandprofit.com/ (besucht am 10. 07. 2020).