Programming Languages from Hell
Proseminar im Sommersemester 2010

Objektorientiertes Programmieren in OCaml

Fabian Streitel
Technische Universität München

07.06.2010

Zusammenfassung

Diese Arbeit gibt einen Überblick über die objektorientierten Features der Sprache OCaml, indem sie deren Hauptmerkmale herausgearbeitet und beschreibt. Anschließend wird ein Vergleich mit den Merkmalen einer klassischen objektorientierten Programmiersprache gezogen, um OCamls Stärken und Schwächen in diesem Bereich zu beurteilen. Hierfür wird das weit verbreitete Java als Beispiel herangezogen.

1 OCaml - Eine Übersicht

Objective CAML (kurz: OCaml) ist ein Abkömmling der ML-Familie, und wie diese zuallererst eine funktionale Sprache. Es besitzt die für diese Sprachfamilie typische starke Typisierung und einen weit ausgereifte Typinferenz-Algorithmus, der auf dem Hindley-Milner Algorithmus basiert. [Waz06, 3–4]

Allerdings ist OCaml keine rein funktionale Sprache, denn es wurde seit seiner ersten Version um einige weitere Programmier-Paradigmen erweitert, unter anderem auch um einige objektorientierte Konzepte.

Letzteres ist von besonderem Interesse, da sich die Objektorientierung in OCaml trotz vieler oberflächlicher Gemeinsamkeiten doch stark von der klassischer Programmiersprachen unterscheidet.

Deshalb sollen im Folgenden zuerst die grundsätzlichen Konzepte der Objektorientierung OCamls herausgearbeitet werden. Im Anschluss daran werden dann die Unterschiede zu den Konzepten Javas aufgezeigt.

2 Objektorientierung in OCaml

Im Folgenden wird die Objektiorientierung OCamls näher erläutert. Hierbei wird auch an einigen Stellen auf Module verwiesen, ein orthogonales Konzept

aus der funktionalen Programmierung. Da dieses die Objektorientierung nur peripher tangiert, wird darauf allerdings nicht näher eingegangen.

2.1 Objektkonstruktion

Ein Objekt wird erstellt, indem seine Instanzvariablen sowie die Definitionen seiner Methoden angegeben werden:

```
1 let lilly =
2     object (self)
3         val name = "Lilly"
4         val mutable age = 23
5         method private birthyear = 2010 - age
6         method grow_older years =
7             age <- age + years; self#birthyear
8     end
```

Methoden werden hierbei durch das Schlüsselwort `method` gekennzeichnet, das optional von den Schlüsselwörtern `private` für private Methoden gefolgt werden kann. Felder werden durch `val` (für unveränderliche Felder) oder `val mutable` (für veränderliche Felder) gekennzeichnet.

Spezielle Bedeutung hat das optionale Konstrukt `(self)` nach dem `object`-Schlüsselwort in Zeile 2. Dies erlaubt es, das konstruierte Objekt an eine Variable zu binden, so dass der Zugriff auf die eigenen Methoden innerhalb eines Objekts möglich ist, wie Zeile 7 zeigt. [Smi06, 228]

Der Bezeichner `self` ist hierbei frei wählbar.

Der Aufruf einer Methode eines Objekts erfolgt über den `#` Operator, wie in Zeile 7 zu erkennen.

2.2 Objekttypen

Da OCaml strikt typisiert ist, haben auch Objekte einen festen Typ. Dieser richtet sich allerdings nicht wie in nominativen Sprachen nach dem Namen einer Klasse oder eines Konstruktors, sondern ausschließlich nach den sichtbaren Methoden, die das Objekt anbietet. [Gar06, 45]

Zum Beispiel lautet der Typ des Objekts aus Kapitel 2.1 wie folgt:

```
1 < grow_older: int -> int >
```

Die private Methode `birthyear` tritt in der Typbeschreibung nicht auf, da sie nach außen hin nicht sichtbar ist.

2.2.1 Row Types

Die Typen aus dem vorangegangenen Kapitel sind sogenannte *Row Types*. Das Konzept stammt ursprünglich von record Datentypen, wurde aber für OCamls Objektsystem übernommen und erweitert.

Jeder Objekttyp besteht aus einer „Reihe“ von Methodendeklarationen, die den Namen und den Typ der Methode angeben. [Ré00, 82]

Das Beispiel aus dem vorangegangenen Kapitel beschreibt also ein Objekt, das genau eine Methode namens `grow_older` mit dem Typ `int -> int` besitzt. Objekte, die zwar die geforderte Methode enthalten, aber auch noch weitere Methoden anbieten, sind inkompatibel mit diesem Typ.

Deshalb wird diese Art von Typen auch geschlossene Objekttypen genannt. [RV98, 3]

2.2.2 Offene Objekttypen

Es gibt aber auch die Möglichkeit, offene Objekttypen zu deklarieren. Diese beschreiben ein Objekt, das die angegebene Reihe von Methoden besitzt, aber noch beliebige weitere Methoden anbieten kann.

Repräsentiert wird dies durch eine sogenannte anonyme *Row Variable*, dargestellt durch eine Ellipse (`..`).

So kann ein Objekt vom Typ `< grow_older : int -> int; get_name : string; get_age : string >` an eine Funktion übergeben werden, die Objekte vom Typ `< grow_older : int -> int; .. >` akzeptiert, da die überschüssigen Methoden `get_name` und `get_age` von der Row Variable akzeptiert werden.

Die Row Variable steht also stellvertretend für alle überschüssigen Methoden, die ein Objekt zusätzlich anbietet. Dadurch wird der offene Objekttyp polymorph, denn er steht jetzt für eine Vielzahl an verschiedenen möglichen Typen.

Wichtig ist zu beachten, dass es sich um *echten* Polymorphismus handelt, das heißt, die Row Variable wird, analog zu ein Typparameter `'a` eines polymorphen Datentyps, an die überschüssigen Methoden gebunden. Zwei Row Variables stehen also niemals für den selben Typ. [Ré00, 74] [Gar06, 45–46]

Zeile 3 des folgenden Beispiels produziert also einen Typfehler, da die erste Row Variable an die Methode `pet : string` gebunden ist und die zweite an `age : int`, weshalb die zwei Typen trotz syntaktischer Gleichheit trotzdem semantisch unterschiedlich sind.

```
1 let lilly = object method name = "Lilly" method pet = "Marvin" end
2 let dilbert = object method name = "Dilbert" method age = 12 end
3 let person_list = [ (lilly : <name : string; ..>); (dilbert : <name
      : string; ..>) ]
```

2.2.3 Subtyping

OCaml definiert eine Beziehung zwischen diesen Objekttypen: Das sogenannte Subtyping. Diese Relation gibt an, ob ein Typ ein Untertyp eines anderen Typs ist.

Dabei wird zwischen *Width Subtyping* und *Depth Subtyping* unterschieden.

Width Subtyping besagt, dass ein Typ `u` ein Untertyp des Typs `o` ist, wenn `u` *mindestens* alle Methoden von `o` implementiert.

Depth Subtyping besagt, dass ein Typ `u` ein Untertyp des Typs `o` ist, wenn `u` die gleichen Methoden anbietet wie `o` und für jeden Methodentyp in `u` gilt, dass er ein Subtyp des Typs der entsprechenden Methode in `o` ist. Hierbei müssen die Subtyping-Regeln der restlichen Typen in OCaml angewandt werden, die sich allerdings von Typ zu Typ unterscheiden. [Hic07, 168]

2.3 Klassen

Allerdings kann es sehr umständlich sein, diese Objekttypen stets auszuschreiben. Dieses Problem kann mit Klassen gelöst werden.

Eine Klasse beschreibt einen bestimmten Objekttyp und gibt eine Implementierung seiner Methoden vor. Außerdem erstellt sie ein Alias für den Typ der durch sie beschriebenen Objekte.

```
1 class person =
2     object (self)
3         val name = "Lilly"
4         val mutable age = 23
5         method private birthyear = 2010 - age
6         method grow_older years =
7             age <- age + years; self#birthyear
8     end
```

Nach dieser Definition ist der Bezeichner `person` ein Alias für den im vorherigen Kapitel dargestellten geschlossenen Objekttyp `< grow_older : int -> int >`. Ebenso ist der Bezeichner `#person` ein Alias für die offene Version dieses Typs: `< grow_older : int -> int; .. >`.

Des Weiteren agiert die Klasse zusammen mit dem `new`-Operator als Konstruktorfunktion, die neue Objekte dieses Typs bauen kann. Diese Objekte besitzen dann die in der Klasse angegebenen Methoden und ihre Implementierungen.

```
1 let lilly = new person
```

Die Konstruktorfunktion kann auch parametrisiert sein, wie im folgenden Beispiel:

```
1 class person init_name init_age =
2     object (self)
3         val name = init_name
4         val mutable age = init_age
5         method private birthyear = 2010 - age
6         method grow_older years =
7             age <- age + years; self#birthyear
8     end
9 let lilly = new person "Lilly" 23
```

2.3.1 Initializers

Es kann vorkommen, dass bei der Konstruktion eines Objekts stets bestimmte Aktionen ausgeführt werden müssen, beispielsweise das Füllen von Feldern mit Werten.

Hierzu bietet OCaml sogenannte Initializer an, die stets bei jeder Objektkonstruktion ausgeführt werden. So gibt die folgende Klasse für jedes Objekt, das sie konstruiert eine Meldung auf der Standardausgabe aus.

```
1 class person init_name init_age =
2     object (self)
3         val name = init_name
4         val mutable age = init_age
5         initializer Printf.printf "%s was born %i years ago.\n"
              init_name init_age
6     end
```

Dieses Feature ist besonders wichtig, weil es OCaml nicht erlaubt, auf den Inhalt eines anderen Feldes innerhalb einer Feldinitialisierung zuzugreifen.

```
1 class person init_name init_age =
2     object (self)
3         val name = init_name
4         val mutable age = init_age
5         val initial_age = age
6     end
```

Dieses Beispiel schlägt in Zeile 5 fehl, da versucht wird, auf das Feld `age` zuzugreifen, während das Feld `initial_age` initialisiert wird.

2.3.2 Polymorphismus

Wie die meisten Datentypen in OCaml unterstützen auch Klassen Polymorphismus. Hierbei muss zwischen polymorphen Methoden und polymorphen Konstruktoren unterschieden werden.

Eine polymorphe Klasse wird durch Angabe von beliebig vielen Typvariablen vor dem Klassennamen deklariert:

```
1 class ['a] person init_name init_age (init_pet : 'a) =
2     object (self)
3         val name = init_name
4         val mutable age = init_age
5         val mutable pet = init_pet
6         method set_pet new_pet = pet <- new_pet
7     end
```

Wichtig hierbei ist zu beachten, dass nicht das Objekt selbst polymorph ist, sondern nur sein Konstruktor. Das bedeutet, dass, sobald der Typparameter eines fertig konstruierten Objekt einmal festgelegt wurde, dieser nicht mehr gewechselt werden kann. [DGRV08, 47] So erzeugt Zeile 3 im folgenden Beispiel einen Typfehler:

```
1 let lilly = new person "Lilly" 23 "Marvin"
2 let cat = object method name = "Felix" end
3 lilly#set_pet cat
```

In Zeile 1 wird die Typvariable `'a` auf `string` festgelegt und kann in Zeile 3 nicht mehr auf einen Objekttyp geändert werden.

Eine polymorphe Methode wird ähnlich wie eine polymorphe Funktion definiert, allerdings muss der Typ der Methode explizit angegeben werden. Hierbei müssen alle Typparameter vor der Typsignatur genannt werden, gefolgt von einem Punkt.

```
1  class person init_name init_age (init_pet : 'a) =
2      object (self)
3          val name = init_name
4          val mutable age = init_age
5          val mutable pet = init_pet
6          method pick_favourite : 'a. 'a list -> 'a option =
7              fun items -> match items with
8                  | [] -> None
9                  | a::b -> Some a
10     end
```

2.3.3 Vererbung

Um Codeduplizierung zu vermeiden, kann eine neue Klasse die Definitionen einer bestehenden Klasse wiederverwenden, erweitern und überschreiben, indem sie von dieser erbt. Dies geschieht mittels des `inherit`-Schlüsselworts:

```
1  class person (init_name : string) =
2      object
3          method name = init_name
4      end
5
6  class qualified_person init_name init_surname =
7      object (self)
8          inherit person init_name as super
9          method firstname = super#name
10         method surname = init_surname
11         method name = self#firstname ^ " " ^ self#surname
12     end
```

Optional kann, wie im obigen Listing, das Objekt der Basisklasse mit dem `as`-Schlüsselwort an einen Bezeichner gebunden werden (Zeile 9), was das Aufrufen der Methoden der Basisklasse ermöglicht, wie Zeile 10 veranschaulicht.

OCaml unterstützt Mehrfachvererbung, es können also beliebig viele Basisklassen spezifiziert werden. Dabei werden frühere Definitionen einer Methode oder eines Felds von späteren überschrieben. [SB00, 1]

Das hierbei entstehende Problem von Mehrfachdefinitionen wird durch eine einfache Regel gelöst: Spätere Definitionen überschreiben frühere Definitonen. Hierbei müssen allerdings alle Definitionen den gleichen Typ besitzen.

```
1  class home_owner (init_place : string) =
2      object
3          method place = init_place
4      end
5
6  class worker (init_place : string) =
7      object
8          method place = init_place
```

```
 9      end
10
11  class home_owning_worker home_place work_place =
12      object
13          inherit home_owner home_place
14          inherit worker work_place
15      end
16
17  let lilly = new home_owning_worker "Garching" "London"
```

Der Aufruf `lilly#place` im obigen Beispiel würde dementsprechend `"London"` zurückliefern, da die Definition der `place` Methode in der `worker` Klasse die Definition in der `home_owner` Klasse überschreibt.

2.3.4 Vererbung und Subtyping

Ein sehr häufiges Missverständnis ist, dass Vererbung und Subtyping in OCaml zusammenhängen. In Wirklichkeit handelt es sich aber um zwei orthogonale Konzepte.

Subtyping bezieht sich nur auf den Typ eines Objekts, also die Reihe seiner Methoden und deren Typen. Vererbung hingegen arbeitet auf der Klassenebene. Sie erlaubt nur ein Wiederverwenden des Codes einer anderen Klasse, sagt jedoch nichts über die Objekttypen der beiden Klassen aus. [Ré00, 92]

Relativ einleuchtend ist, dass es Klassen gibt, die in einer Subtypebeziehung zueinander stehen, aber nicht in einer Vererbungsbeziehung, zum Beispiel:

```
1   class person (init : string) =
2       object
3           method name = init
4       end
5
6   class young_person (init : string) =
7       object
8           method name = init
9           method age = 18
10      end
```

`young_person` ist hier ein Subtyp von `person` nach dem Prinzip des Width-Subtyping.

Aber auch das Gegenteil ist möglich, allerdings seltener. So hat zum Beispiel eine Klasse, die eine binäre Methode enthält, nur ihren eigenen Objekttyp als Subtyp. Folglich sind alle Subklassen einer solchen Klasse nicht Subtypen von dieser (vergleiche hierzu [Ré00, 79]).

2.3.5 Abstrakte Klassen

In OCaml sind abstrakte Klassen und Methoden durch das `virtual`-Schlüsselwort gekennzeichnet. Eine solche Methode muss mit einem Typ angegeben werden.

```
1   class virtual person (init_name : string) =
2       object
3           method name = init_name
```

```
4         method virtual pick_favourite : 'a. 'a list -> 'a option
5     end
```

Eine Klasse, die mindestens eine virtuelle Methode besitzt, muss als `virtual` markiert werden. Diese Klassen können nicht mit einem Konstruktor-Aufruf instantiiert werden. [Smi06, 234]

2.4 Class Types und Coercion

Oftmals kann es wichtig sein, verschiedene Klassen von Objekten, die teilweise die gleichen Methoden anbieten, zusammen zu verarbeiten, zum Beispiel in einer Liste:

```
1  let lilly =
2      object
3          method name = "Lilly"
4          method pet = "Marvin"
5      end
6
7  let dilbert =
8      object
9          method name = "Dilbert"
10         method pet = "Dogbert"
11         method pet2 = "Catbert"
12     end
13 let persons = [lilly; dilbert]
```

Die Objekte haben allerdings unterschiedliche Typen, da sie unterschiedliche Methoden anbieten. Deshalb führt der Versuch, beide in eine Liste einzufügen zu einem Fehler (Zeile 13).

Um dies dennoch zu ermöglichen, muss der Typ der beiden Variablen `lilly` und `dilbert` gleich sein. Da dies, wie in Kapitel 2.2.2 bereits gezeigt, nicht mit offenen Objekttypen gelöst werden kann, muss ein anderes Konzept verwendet werden. OCaml bietet hierzu Class Types und Coercion an.

```
1  class type person =
2      object
3          method name : string
4          method pet : string
5      end
```

Wie Abstrakte Klassen erzeugen Class Types jeweils einen Typalias für ihren geschlossenen und offenen Objekttyp und können nicht instantiiert werden, allerdings stehen sie nicht für Vererbung zur Verfügung.

Im Anschluss an obige Definition können nun beide Objekte mit Hilfe einer Coercion – gekennzeichnet durch den Operator `:>` – auf den Typ `person` „reduziert“ werden. Dabei „vergisst“ der Compiler die zusätzlichen Methoden der beiden Objekte sozusagen.

```
1  let lilly_person = (lilly :> person)
2  let dilbert_person = (dilbert :> person)
```

Da sie nun beide den gleichen Typ haben, können sie in eine Liste eingefügt werden:

```
1 let persons = [lilly_person; dilbert_person]
```

Coercion muss allerdings nicht zwangsweise mit Class Types durchgeführt werden. Jeder geschlossene Objekttyp kann hierfür hergenommen werden. Also sind auch durch Klassen erzeugte Aliase oder eine direkte Angabe eines solchen Typs, z.B. `< name : string; pet : string >` möglich.

3 OCaml im Vergleich

Es ist bereits aus den obigen Beispielen und Erklärungen ersichtlich, dass sich die Objektorientierung in OCaml in einigen Punkten stark von der klassischer objektorientierter Sprachen unterscheidet. Im Folgenden werden die Kernunterschiede zwischen OCaml und Java herausgearbeitet, das hier exemplarisch für diese Klasse von Sprachen steht.

3.1 Konstruktion von Objekten

Einer der wichtigsten Unterschiede besteht in der Objektkonstruktion. In Java ist es nur erlaubt, Objekte über die Konstruktoren einer Klasse zu erstellen. In OCaml hingegen können Objekte direkt und jederzeit durch Angabe ihrer Methoden und Felder erzeugt werden (vergleiche Kapitel 2.1).

Aber auch die Konstruktion über Klassen unterscheidet sich wesentlich von Java. So ist in Java ein expliziter Aufruf des Konstruktors der Oberklasse nötig, um diesen auszuführen. In OCaml hingegen werden die Initializer aller Oberklassen bei jeder Konstruktion in der Reihenfolge ihrer Definition ausgeführt. Der Programmierer hat hier keine Möglichkeit, Initializer auszulassen oder deren Reihenfolge zu bestimmen.

3.2 Typensystem

Offensichtlich verwenden Java und OCaml ein deutlich unterschiedliches Typsystem. Besonders bei Objekten ist dies klar erkennbar.

In OCaml ist der Typ eines Objekts durch seine Methoden und Felder bestimmt; Typnamen, die durch Klassendefinitionen eingeführt werden sind nur Aliase für diese Objekttypen (vgl. Kapitel 2.3). In Java hingegen wird ein Typ nur über seinen Namen indentifiziert.

In OCaml sind also zwei Klassen, die die selben Methoden angeben, vom Typ her identisch, auch wenn ihre Klassennamen unterschiedlich sind; in Java nicht.

Dies bedingt auch, dass in OCaml der Zusammenhang zwischen einem Objekttyp und seiner semantischen Bedeutung nicht so stark ist wie in Java.

Ein Java-Objekt des Typs `Cowboy` ist jederzeit klar als solcher identifizierbar. Hingegen kann ein OCaml-Objekt mit dem Typ `< draw : unit; position :`

`int * int >` sowohl einen Cowboy, als auch ein Rechteck beschreiben. Dies bedeutet auch, dass Funktionen, die einen bestimmten Objekttyp erwarten, dennoch Objekte erhalten können, die nicht ihren semantischen Voraussetzungen entsprechen und dementsprechend keine sinnvolle Verarbeitung ermöglichen. Dies ist bekannt als das „Accidental Conformance“ Problem. [PZ06, 6–7]

3.3 Vererbung und Subtyping

In Java sind Subtyping und Vererbung gekoppelt. Erbt eine Klasse von einer anderen, so ist sie automatisch ihr Subtyp. Dies führt dazu, dass Objekte, die durch Casting auf einen allgemeineren Typ reduziert wurden, sehr einfach wieder auf einen spezifischeren zurückgeführt werden können. Diese Rückführung des Typs wird auch als *Narrowing* bezeichnet.

Da in OCaml diese Beziehungen getrennt sind, erlaubt der Compiler kein Narrowing, da zur Laufzeit nicht mehr entschieden werden kann, was der richtige Typ eines Objekts ist. [Hic07, 170–171]

So schlägt etwa die letzte Zeile des folgenden Codelistings fehl:

```
1 class animal = object method eat = () end
2 class dog =
3     object
4         inherit animal
5         method bark = print_string "bark!\n"
6     end
7 let some_animal = ((new dog) :> animal)
8 let some_dog = (some_animal :> dog)
```

Dies kann nur mit Tricks umgangen werden, etwa indem mit Exceptions und Pattern Matching Runtime Type Information simuliert wird. Siehe hierzu etwa [Hic07, 172]

3.4 Sichtbarkeiten

Die klassischen objektorientierten Sprachen bieten ein striktes, mehrstufiges Sichtbarkeitssystem an. Java zum Beispiel verfügt über die Sichtbarkeitsbereiche `private`, `protected`, `default` und `public`. Diese Bereiche können auf Methoden, Felder und Klassen angewandt werden und bestimmen, welche anderen Teile eines Programms darauf zugreifen dürfen. OCaml dagegen bietet nur `private` als Sichtbarkeitsbegrenzung für Methoden an. So annotierte Methoden können nur vom jeweiligen Objekt selbst gesehen werden. Felder sind grundsätzlich nur innerhalb eines Objekts sichtbar. [Smi06, 233]

Auch was Vererbung angeht sind die Sichtbarkeitsgrenzen in OCaml grundverschieden von denen Javas. So geben die Sichtbarkeitsgrenzen in Java auch vor, wie Subklassen die Methoden und Felder der Oberklassen sehen und überschreiben können. OCaml wählt hier eine andere Strategie. Hier sind alle Methoden und Felder der Oberklasse für die Subklasse zugänglich und können auch überschrieben werden. Es ist sogar möglich, die Sichtbarkeit von Methoden der Oberklasse zu ändern, beispielsweise mit `private` annotierte Methoden sichtbar

zu machen. Es gibt allerdings Mechanismen, um strengere Sichtbarkeitsgrenzen zu schaffen, etwa mittels Constraints. [Hic07, 188]

Auch kann das Signatursystem von Modulen genutzt werden, um die interne Repräsentation von Klassen zu verstecken und so eine Kapselung der Daten gegenüber anderen Modulen zu erreichen.

4 Zusammenfassung

Insgesamt kann man sagen, dass OCaml einen anderen Ansatz verfolgt als die meisten Programmiersprachen mit Objektorientierung.

Da OCaml seine Stärken aus seinem strengen Typensystem zieht, mussten sich auch die Objekte diesem unterwerfen. Dies führt dazu, dass manche Konzepte aus den klassischen OO-Sprachen nicht direkt umgesetzt werden konnten, zum Beispiel die Vererbung oder die Datenkapselung.

Dieses Manko ist aber gleichzeitig auch ein positiver Einfluss, denn die Strenge des Typsystems bietet dem Programmierer zugleich auch eine erhöhte Sicherheit und kann dabei helfen, Programmierfehler zu vermeiden.

Außerdem ermöglicht OCaml mit neuen Features – wie dem direkten Erstellen von Objekten ohne explizite Klassen – auch neue Möglichkeiten und ermöglicht einen anderen, funktionaleren Programmierstil als die klassischen objektorientierten Sprachen.

Insgesamt bietet OCaml eine solide Objektorientierung, die durchaus mit der anderer objektorientierter Sprachen mithalten kann.

Literatur

[DGRV08] Doligez, Damien ; Garrigue, Jacques ; Rémy, Didier ; Vouillon, Jérôme: *The Objective Caml system release 3.11*. http://caml.inria.fr/pub/docs/manual-ocaml/index.html. Version: November 2008

[Gar06] Garrigue, Jacques: Private row types: Abstracting the unnamed. In: *of Lecture Notes in Computer Science*, Springer, 2006, 44–60

[Hic07] Hickey, Jason: *Introduction to Objective Caml*. Cambridge University Press, 2007

[PZ06] *Kapitel* A Feature Composition Problem and a Solution Based on C++ Template Metaprogramming. In: Porkoláb, Zoltán ; Zólyomi, István: *Generative and Transformational Techniques in Software Engineering*. Springer Berlin / Heidelberg, 2006, S. 459–470

[RV98] Rémy, Didier ; Vouillon, Jérôme: Objective ML: An effective object-oriented extension to ML. In: *Theory and Practice of Object-Oriented Systems* 4 (1998), S. 27–50

[Ré00] Rémy, Didier: Using, Understanding, and Unraveling the OCaml Language. From Practice to Theory and Vice Versa. In: *Applied Semantics*, Springer-Verlag, 2000. – ISBN 3–540–44044–5, 413-536

[SB00] S Boulmé, R R. T Hardin H. T Hardin: Polymorphic data types, objects, modules and functors: is it too much. In: *RR 014, LIP6, Université Paris 6*, 2000

[Smi06] Smith, Joshua B.: *Practical OCaml*. Apress, 2006

[Waz06] Wazny, Jeremy R.: *Type inference and type error diagnosis for Hindley/Milner with extensions*, University of Melbourne, Australia, Diss., 2006