UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Llenguatges de Programació

Treball Dirigit

OCaml

Índex

1 Introducció **3**

2 Paradigmes de Programació **3**
2.1 Paradigma Imperatiu 4
2.2 Paradigma Declaratiu 5
2.3 Paradigma Funcional 5

3 Descripció i Propòsit del Llenguatge **6**
3.1 Inicis de OCaml 6
3.2 Usos 6
3.3 Característiques Generals de OCaml 6
3.3.1 Sistema de Tipus i Inferència 6
3.3.2 Sistema d'Execució 7
3.3.3 Orientació a Objectes 7
3.4 Característiques Funcionals de OCaml 7
3.4.1 Funcions d'Ordre Superior 7
3.4.2 Recursivitat 7
3.4.3 Currificació 8
3.4.4 Avaluació Mandrosa 8
3.4.5 Immutabilitat de Dades 8
3.5 Limitacions Funcionals 9
3.6 Conclusions 9

4 Estudi Bibliogràfic **10**

1 Introducció

El propòsit d'aquest treball dirigit és cobrir els aspectes més essencials de *OCaml*, un llenguatge de programació funcional d'alt nivell. En aquesta memòria es recercarà informació sobre els paradigmes del llenguatge i s'explicarà amb detall el seu funcionament. A més a més, es mostraran les característiques més transcendentals de *OCaml* així com un breu resum de la seva història, el seu àmbit d'ús i alguns exemples de codi per mostrar-ne la interfície.

2 Paradigmes de Programació

Els paradigmes de programació són estils fonamentals d'organització i abstracció del codi que determinen com s'estructura i s'executa un programa. Un paradigma no només expressa la intenció del programador, sinó que estableix un marc conceptual complet que defineix:

- Com es representen i s'organitzen les dades
- Com es manipula l'estat del programa
- Quins són els mecanismes d'abstracció disponibles
- Quins són els patrons de control de flux permesos

Dit d'una altra manera, un paradigma actua com un conjunt coherent de principis i regles que proporcionen un model mental específic per abordar problemes computacionals. Aquests principis fonamentals, no només influeixen en com es resolen els problemes, sinó també en com es conceptualitzen i es modelen les solucions en el procés de desenvolupament de software. Els paradigmes principals són l'Imperatiu i el Declaratiu. És del paradigma declaratiu on es considera que hi està inclòs el Funcional, paradigma en el qual es basa el llenguatge objecte d'estudi *OCaml*.

(Fonts: [KF19], [Han07] i [Hud89]).

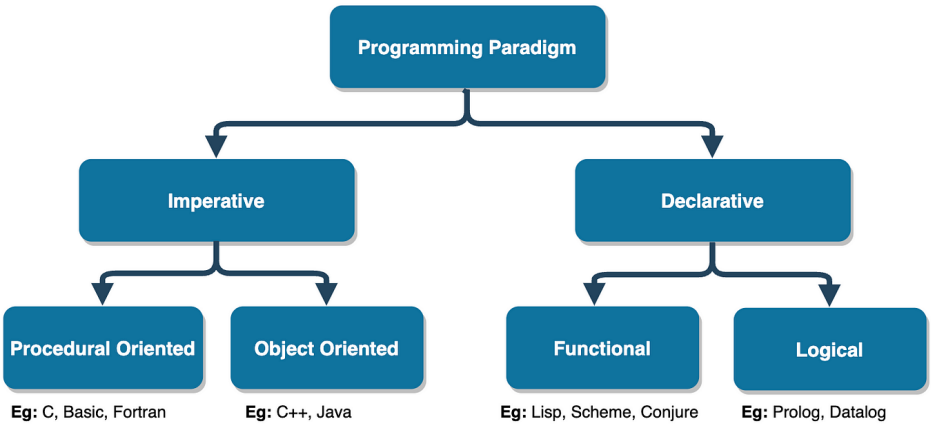

Figure 1: Paradigmes de Programació

Com es pot observar en la Figura 1, el paradigma Funcional ve derivat del Declaratiu.

2.1 Paradigma Imperatiu

El paradigma imperatiu es basa a descriure la computació en termes que modifiquen l'estat d'un programa. Es caracteritza per tres aspectes fonamentals:

- Primer, es basa en l'ús d'un estat del programa que utilitza variables modificables, on les dades es guarden en posicions de memòria que poden ser actualitzades durant l'execució.
- Segon, s'estructura mitjançant una seqüència d'instruccions que s'executen en ordre, on cada instrucció pot modificar l'estat del programa i aquest ordre és determinant per obtenir el resultat esperat.
- Tercer, implementa mecanismes de control de flux com bucles *for, while*, condicionals *if-then-else* i salts i subrutines *goto, funcions* per dirigir l'execució del programa.

Aquest paradigma és similar al format que tenen les instruccions pas per pas d'una recepta de cuina i és el paradigma que s'utilitzen en llenguatges com el C++. L'avantatge principal dels llenguatges amb paradigmes imperatius és la facilitat de comprensió per a programadors novicis.

Amb tot i això, el paradigma pot esdevenir més difícil conforme la complexitat del problema augmenta, particularment en el tractament d'efectes secundaris inesperats i en la gestió de la concurrència.

(Fonts: [HM89] i [KF19])

A continuació es mostra un problema simple amb C++ (veure Listing 1).

```cpp
// Suma de tots els elements d'un vector d'enters
int main() {
    int total = 0;                          //total de la suma
    vector<int> vec {1, 2, 3, 4};           //vector d'enters
    for(int i = 0; i < vec.size(); ++i){    //per a tot element de vec
        total += vec[i];                    //incrementem el valor
    }
    cout << "Total = " << total << endl;    //imprimim el resultat
}                                           //Total = 10
```

Listing 1: Exemple d'un programa que calcula la suma d'un vector d'enters en C++.

Aquest programa exemplifica com faria la suma d'elements d'un vector una persona sense coneixements de programació. Hom començaria agafant el primer element i el sumaria a un total que inicialment és 0, agafaria el següent element i el sumaria al total, així successivament fins que no quedessin elements al vector.

Aquest exemple és molt útil per veure com resoldre programes de forma imperativa, on la consecució d'una sèrie de passos porten al resultat final esperat.

2.2 Paradigma Declaratiu

El paradigma declaratiu, per contrast en respecte a l'imperatiu, expressa la lògica d'una computació sense descriure'n el flux de control. Dit en altres paraules, quan es programa amb llenguatges declaratius, només importa el resultat, el procediment no interessa. Les característiques principals són:

1. No té estat mutable.
2. L'ordre d'execució és irrellevant.
3. Es centra en expressar el problema en comptes de resoldre'l.

Dels diversos subparadigmes que deriven del declaratiu, el funcional és un dels més importants.

(Fonts: [Han07] i [Hud89])

2.3 Paradigma Funcional

Els llenguatges que utilitzen aquest paradigma, basen la programació en l'avaluació de funcions matemàtiques. Totes les operacions es realitzen mitjançant l'aplicació de funcions.

El paradigma funcional es defineix per tres característiques fonamentals que el diferencien d'altres paradigmes.

- Primer, igual que el paradigma declaratiu, es basa en la immutabilitat de les dades, el que significa que les dades no poden ser modificades un cop creades i cada operació genera noves dades en comptes de modificar les existents.
- Segon, utilitza funcions pures que sempre retornen el mateix resultat per als mateixos arguments, no tenen efectes secundaris i no depenen de cap estat global del programa.
- Tercer, empra la recursió com a mecanisme principal per treballar amb estructures de dades, substituint així els bucles iteratius típics d'altres paradigmes.

(Fonts: [Hud89] i [Har16])

Vegin un exemple de programació simple amb el LP funcional Haskell (Listing 2):

```haskell
--Suma de tots els elements d'una llista d'enters
sumList :: [Int] -> Int  --La funcio rep una llista i retorna un int
sumList [] = 0                        --Cas base: La llista esta buida
sumList (x:xs) = x + sumList xs       --Cas recursiu

main :: IO ()
main = print (sumList [1, 2, 3, 4]) --Imprimeix la suma dels
                                    --elements (10)
```

Listing 2: Exemple d'un programa que suma una llista d'enters en Haskell.

Aquest programa exemplifica la naturalesa recursiva del paradigma funcional. La funció *sumList* es defineix mitjançant dos casos: el cas base (quan la llista és buida, retorna 0) i el cas recursiu (suma el primer element amb la suma recursiva de la resta d'elements).

3 Descripció i Propòsit del Llenguatge

3.1 Inicis de OCaml

OCaml (Objective Categorical Abstract Machine Language) és un LP creat per l'entitat estatal francesa de recerca *INRIA* [1] al 1996. Aquesta versió disposa de construccions d'Orientació a Objectes, successora de l'extensió anterior del mateix llenguatge anomenada Caml Light.

(Fonts: [Mil+07])

3.2 Usos

OCaml és un llenguatge de programació d'alt nivell que destaca per ser multiparadigma, combinant principalment la programació funcional amb característiques imperatives i orientades a objectes. Aquesta versatilitat permet als desenvolupadors aprofitar els beneficis de la programació funcional pura, característica heretada de la família ML, mentre manté la flexibilitat d'utilitzar construccions imperatives quan són necessàries i incorpora un sistema d'objectes únic en el seu disseny. El llenguatge es distingeix especialment per la seva capacitat de detectar errors en temps de compilació gràcies al seu robust sistema de tipus estàtics, fet que el fa particularment valuós en entorns on la fiabilitat del codi és crítica.

S'utilitza principalment en el desenvolupament d'eines de compilació, sistemes de verificació formal i aplicacions financeres d'alt rendiment. Un exemple destacat és el seu ús per part de Jane Street en sistemes de trading d'alta freqüència, així com per Facebook en el desenvolupament de Flow, una eina d'anàlisi estàtic per a JavaScript.

(Fonts: [Rém00], [FPL07] i [MMH13])

3.3 Característiques Generals de OCaml

3.3.1 Sistema de Tipus i Inferència

El sistema de tipus d'OCaml és estàtic, fortament tipat i amb inferència de tipus, i ofereix diverses característiques que el fan potent i flexible. Tipus estàtic significa que els tipus són verificats en temps de compilació, no en temps d'execució, permetent detectar errors de tipus abans que el programa s'executi. El tipus fort no permet conversions implícites entre tipus diferents. Per exemple, una suma *Int* + *Float* dona error ja que OCaml requereix convertir explícitament un tipus a l'altre.

La inferència de tipus en OCaml és una de les seves característiques més potents i distintives, basada en l'algorisme Hindley-Milner, que permet al compilador deduir automàticament els tipus sense necessitat de declaracions explícites. El sistema inclou tipus polimòrfics (genèrics) que permeten escriure codi que funciona amb múltiples tipus de dades, proporcionant flexibilitat i reusabilitat al codi, ja que una mateixa funció o estructura pot treballar amb diferents tipus. També inclou un sistema de mòduls i functors per organitzar i combinar diferents tipus i comportaments, oferint una manera d'estructurar programes grans i gestionar la visibilitat del codi. Per últim, OCaml permet la definició de tipus per crear nous tipus de dades personalitzades, com ara registres, variants i tipus recursius, que permeten modelar estructures de dades complexes mantenint la seguretat del sistema de tipus.

(Fonts: [Rém00] i [FPL07])

[1] Institut National de Recherche en Informatique et en Automatique

3.3.2 Sistema d'Execució

El sistema d'execució d'OCaml combina compilació a codi natiu i bytecode, interpretació interactiva, gestió eficient de memòria i suport per a concurrència i paral·lelisme. Amb el compilador natiu (ocamlopt), genera executables independents altament optimitzats, mentre que el compilador a bytecode (ocamlc) produeix codi portàtil que s'executa amb la màquina virtual (ocamlrun). Per al desenvolupament ràpid, el toplevel (utop) permet executar codi de forma interactiva. La gestió automàtica de memòria amb un *garbage collector* generacional optimitza el rendiment. OCaml suporta concurrència amb biblioteques com *Lwt* i *Async* per a operacions asíncrones, i paral·lelisme real amb Multi-Core OCaml. A més, facilita la integració amb codi extern (com C) i ofereix eines com Dune per a la gestió de projectes i OPAM per a dependències. Aquest sistema equilibra desenvolupament ràpid, portabilitat i alt rendiment.

(Fonts: [Rém00], [MMH13] i [OCa24])

3.3.3 Orientació a Objectes

Tot i que aquesta no és la seva característica principal, ja que és un llenguatge funcional per disseny. OCaml integra suport per a OO com una extensió opcional, proporcionant una combinació interessant de paradigmes *(Fonts: [Rém00])*.

3.4 Característiques Funcionals de OCaml

3.4.1 Funcions d'Ordre Superior

OCaml permet l'ús de funcions d'ordre superior. Aquestes són funcions que accepten a altres funcions com a paràmetres. Al igual que en Haskell, OCaml incorpora varies funcions d'ordre superior per treballar amb múltiples estructures de dades (*iter, map, filter, ...*) *(Fonts: [Hud89])*. També se'n poden implementar de noves amb facilitat (veure Listing 3).

```
let apply_twice f x = f (f x)
let add_one x = x + 1
let result = apply_twice add_one 1   (* Retorna 3 *)
```

Listing 3: Exemple d'una funció d'ordre superior en OCaml.

3.4.2 Recursivitat

Com es típic en la majoria de llenguatges funcionals, OCaml no té bucles com *for* o *while* i depèn de la recursivitat per iterar sobre estructures de dades *(Fonts: [Rém00])*. Vegeu el Listing 4 per entendre com funciona la suma d'un vector d'enters.

```
let rec suma_array arr pos =
    if pos >= Array.length arr then 0
    else arr.(pos) + suma_array arr (pos + 1)
let arr = [|1; 2; 3; 4|]
let resultat1 = suma_array arr 0     (* Retorna 10 *)
```

Listing 4: Exemple d'una funció recursiva en OCaml.

A més a més, OCaml implementa l'optimització de la recursivitat terminal (*tail recursion*), que és crucial per l'eficiència. Quan una funció és recursiva terminal, el compilador d'OCaml la converteix automàticament en un bucle iteratiu, evitant així el desbordament de la pila *(Fonts: [MMH13])*.

3.4.3 Currificació

OCaml utilitza la currificació de manera nativa i aquesta és una de les seves característiques funcionals més importants. La currificació en OCaml significa que totes les funcions realment accepten un sol argument i retornen una funció *(Fonts: [Har16])*.

```
let suma x y = x + y
let resultat1 = suma 1 2          (* Podem utilitzar-la de manera "
    normal" 1 + 2 = 3 *)

(* Pero tambe podem currificar-la *)
let suma_dos = suma 2             (* Funcio que suma 2 a un numero *)
let resultat2 = suma_dos 1        (* Tambe retorna 3 *)
```

Listing 5: Exemple de Currificació en OCaml.

En aquest exemple (veure Listing 5), quan escrivim *suma 3*, OCaml no genera un error per falta d'arguments, sinó que retorna una nova funció que espera un argument.

3.4.4 Avaluació Mandrosa

L'avaluació mandrosa (lazy evaluation) és una tècnica en què les expressions no es calculen fins que el seu valor és necessari, permetent una execució més eficient i el suport a estructures de dades infinites *(Fonts: [Har16] i [FPL07])*.

Tot i no ser el sistema d'avaluació per defecte, ja que, a no ser que se li especifiqui lo contrari, OCaml utilitza la avaluació *eager*. Això es pot canviat usant el tipus *lazy* (veure Listing 6).

```
(* Crear una expressio mandrosa *)
let calcul_costos = lazy (
    Printf.printf "Calculant...\n";
    let resultat = List.init 1000000 (fun x -> x * x) in
    List.hd resultat
)
(* L'expressio no s'avalua fins que utilitzem Lazy.force *)
let resultat = Lazy.force calcul_costos
(* Imprimira "Calculant..." i retornara el primer element *)
```

Listing 6: Exemple d'Avaluació Mandrosa en OCaml.

3.4.5 Immutabilitat de Dades

La immutabilitat de dades és una propietat en què les estructures de dades no es poden modificar després de ser creades, garantint consistència, seguretat i facilitant el raonament del codi *(Fonts: [MMH13])*.

OCaml hi té una relació interessant, ja que adopta un enfocament híbrid. Per defecte, les dades en OCaml són immutables, però el llenguatge també permet la mutabilitat quan es necessita explícitament (veure Listing 7).

```
let x = 5                (* Les vinculacions (bindings) son
    immutables per defecte *)
(* x <- 6 *)             (* Aixo donaria un error de compilacio *)

let comptador = ref 0    (* Crear una referencia mutable *)
(* Modificar el valor *)
comptador := !comptador + 1      (* El valor ara es 1 *)
let valor = !comptador           (* Llegir el valor *)
```

Listing 7: Exemple d'Immutabilitat en OCaml.

3.5 Limitacions Funcionals

Tot i els seus nombrosos avantatges, OCaml presenta certes limitacions des del punt de vista de la programació funcional pura que cal considerar. A diferència de llenguatges purament funcionals com Haskell, OCaml adopta un enfocament híbrid que, si bé ofereix flexibilitat, pot comprometre alguns principis funcionals fonamentals.

Una d'elles és l'avaluació eager per defecte, que contrasta amb l'avaluació mandrosa nativa d'altres llenguatges funcionals. Tot i que OCaml permet l'ús del tipus lazy, aquesta solució requereix una gestió explícita que pot resultar menys elegant i eficient en certes operacions.

La integració del paradigma orientat a objectes introdueix complexitats addicionals en el sistema de tipus i pot generar tensions amb els principis funcionals purs, especialment en aspectes com l'herència i el polimorfisme, que poden interferir amb la composició funcional. A més, la possibilitat d'utilitzar construccions imperatives i efectes secundaris, encara que útil en la pràctica, pot comprometre la transparència referencial i dificultar el raonament sobre el comportament del programa.

Una altra restricció significativa és l'absència d'un sistema de mònades integrat com el de Haskell, fet que complica la gestió d'efectes secundaris i pot resultar en solucions menys elegants per a certes operacions.

(Fonts: [Hud89], [Har16], [FPL07], [Rém00])

3.6 Conclusions

OCaml és un llenguatge de programació funcional que, gràcies al seu estatus híbrid entre diversos paradigmes, permet agafar característiques importants de cada un d'ells sense corrompre la seva essència funcional. El seu sistema de tipus sofisticat i la detecció d'errors en temps de compilació el fan particularment adequat per a aplicacions on la fiabilitat del codi és crítica, com en sectors financers i en el desenvolupament d'eines de verificació formal. OCaml representa així un equilibri excel·lent entre els principis funcionals i les necessitats pràctiques del desenvolupament modern.

4 Estudi Bibliogràfic

Per abordar la realització d'aquest treball, s'ha dut a terme una recerca bibliogràfica exhaustiva i sistemàtica. La bibliografia consultada es pot estructurar en tres àmbits principals, tot i que algunes fonts són transversals i contribueixen a múltiples aspectes del treball:

- **Documentació sobre paradigmes:** Fonts centrades en els fonaments dels paradigmes de programació, essencials per contextualitzar OCaml dins l'espectre de llenguatges de programació.
- **Documentació sobre informació d'OCaml:** Referències que cobreixen l'evolució històrica, característiques distintives i propòsit del llenguatge.
- **Documentació sobre programació en OCaml:** Materials específics sobre aspectes pràctics, incloent programació funcional, sistema de tipus i característiques avançades.

A continuació es detallen les fonts consultades, totes elles extretes d'articles científics i llibres d'autors distingits en la seva respectiva matèria, fet que la converteixen en una bibliografia sòlida.

- **Krishnamurthi, S. (2019).** *Programming Paradigms and Beyond*. Font moderna que analitza els paradigmes de programació des d'una perspectiva educativa, proporcionant un marc conceptual sòlid per entendre les diferències i relacions entre paradigmes. Especialment rellevant per la classificació i comparació dels paradigmes imperatiu, declaratiu i funcional [KF19].
- **Harper, R. (2016).** *Practical Foundations for Programming Languages*. Obra que examina els fonaments teòrics dels llenguatges de programació, oferint una base matemàtica i conceptual per entendre els diferents paradigmes. La seva anàlisi dels sistemes de tipus i avaluació ha estat especialment valuosa per comprendre les característiques d'OCaml [Har16].
- **Hudak, P. (1989).** *Conception, Evolution, and Application of Functional Programming Languages*. Article fonamental que explora l'evolució i els principis de la programació funcional. Tot i ser una font més antiga, la seva anàlisi dels conceptes funcionals purs ha estat essencial per contextualitzar les característiques funcionals d'OCaml [Hud89].
- **Hanus, M. (2007).** *Multi-paradigm declarative languages*. Exploració detallada del paradigma declaratiu i la seva integració amb altres paradigmes. La seva anàlisi ha estat clau per entendre com OCaml combina elements declaratius amb altres paradigmes [Han07].
- **Hughes, J. (1989).** *Structured Programming: Theory and Practice*. Obra que estableix els fonaments de la programació estructurada i el paradigma imperatiu. La seva explicació dels conceptes imperatius ha permès entendre millor com OCaml incorpora característiques imperatives en un llenguatge principalment funcional [HM89].
- **Milner, R., et al. (2007).** *The History of Standard ML*. Document crucial per entendre els orígens i l'evolució d'OCaml, traçant la seva història des de ML fins a la seva forma actual. Proporciona context històric i tècnic sobre les decisions de disseny que han influït en el desenvolupament d'OCaml [Mil+07].
- **Rémy, D. (2000).** *Using, understanding, and unraveling the OCaml language from practice to theory and vice versa*. Font comprehensiva que connecta la teoria amb la pràctica d'OCaml. La seva anàlisi del sistema de tipus, inferència de tipus i model d'execució ha estat fonamental per entendre tant els aspectes teòrics com pràctics del llenguatge [Rém00].

- **Fisher, K., et al. (2007).** *Compiling for Advanced Object-Oriented Features in OCaml*. Article que examina en profunditat la implementació d'objectes en OCaml, proporcionant una visió detallada de com el llenguatge integra la programació orientada a objectes amb el paradigma funcional. Ha estat especialment útil per entendre les limitacions i avantatges d'aquesta integració [FPL07].
- **Minsky, Y., et al. (2013).** *Real World OCaml: Functional programming for the masses*. Guia pràctica que mostra com OCaml s'utilitza en entorns de producció reals. La seva cobertura d'exemples pràctics i patrons de disseny ha estat essencial per entendre els usos i aplicacions modernes del llenguatge [MMH13].
- **OCaml Team. (2024).** *OCaml Manual, version 5.2*. Documentació oficial que serveix com a referència autoritativa per les característiques i comportament d'OCaml. Ha estat una font crucial per verificar detalls tècnics i entendre les últimes capacitats del llenguatge, especialment en relació amb les característiques més recents [OCa24].

References

[KF19] Shriram Krishnamurthi and Kathi Fisler. "Programming Paradigms and Beyond". In: *The Cambridge Handbook of Computing Education Research*. Ed. by Sally A. Fincher and Anthony V. Robins. Cambridge Handbooks in Psychology. Cambridge: Cambridge University Press, 2019, pp. 377–413.

[Han07] Michael Hanus. "Multi-paradigm declarative languages". In: (2007), pp. 45–75.

[Hud89] Paul Hudak. "Conception, Evolution, and Application of Functional Programming Languages". In: *ACM Computing Surveys (CSUR)* 21.3 (1989), pp. 359–411.

[HM89] Joan K Hughes and John I Michtom. *Structured Programming: Theory and Practice*. Sams, 1989.

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.

[Mil+07] Robin Milner et al. "The History of Standard ML". In: *Conference Record of the Second ACM SIGPLAN Conference on History of Programming Languages*. ACM, 2007, pp. 1–53.

[Rém00] Didier Rémy. "Using, understanding, and unraveling the OCaml language from practice to theory and vice versa". In: *International Summer School on Applied Semantics*. Springer, 2000, pp. 413–536.

[FPL07] Kathleen Fisher, François Pottier, and Xavier Leroy. "Compiling for Advanced Object-Oriented Features in OCaml". In: *Journal of Functional Programming* 17.3 (2007), pp. 225–254.

[MMH13] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional programming for the masses*. O'Reilly Media, Inc., 2013.

[OCa24] OCaml Development Team. *OCaml Manual, version 5.2*. Accessed: 2024-11-25. OCaml Development Team. 2024. URL: `https://ocaml.org/manual/5.2/index.html`.