Programmation fonctionnelle : langages, paradigme et écosystème

Axel Carayon, Ana-Maria Cerbulean, Corentin Gobbo, Mohamed Masbah Abou Laich, and Erik Martin-Dorel

Université Toulouse III - Paul Sabatier

Résumé Né à la fin des années 1950 dans le berceau du milieu académique, le paradigme de programmation fonctionnelle est devenu au fil du temps un paradigme très influent dans l'industrie du génie logiciel. De nombreux langages très populaires sont en effet multi-paradigmes, et permettent de tirer profit des caractéristiques intrinsèques du fonctionnel, tel que la composition de fonctions pures qui permet de mieux gérer la complexité croissante des projets logiciels. Ce document s'attache, en s'appuyant sur des articles emblématiques ou historiques liés à ce paradigme, à donner une vue d'ensemble des principaux concepts qu'on retrouve usuellement dans les langages fonctionnels, de leur intérêt, et de leur évolution.

Mots-clés : λ-calcul. Effets de bord. Fonctions d'ordre supérieur. Polymorphisme. Évaluation paresseuse. Comparaison de paradigmes. LISP. Haskell. OCaml. JavaScript.

1 Introduction

1.1 Qu'est-ce que la programmation fonctionnelle ?

La programmation fonctionnelle se trouve parmi les paradigmes de programmation les plus anciens et ayant un taux de popularité élevé depuis quelques dizaines d'années. C'est un paradigme de programmation qui se construit sur la notion de « fonction » au sens mathématique du terme. Construite de la même façon que les fonctions mathématiques, la programmation fonctionnelle cherche à éviter les effets de bord, les mutations de variables et privilégie la récursivité aux structures itératives comme les boucles. Dans un programme fonctionnel, chaque élément peut être vu et interprété en tant que fonction. Grâce à sa modularité et son allure de preuve mathématique, l'approche fonctionnelle est polyvalente, elle permet de développer des programmes complexes, mais néanmoins fiables et maintenables. Comme explicité dans l'article *How functional programming mattered ?* [5], la programmation fonctionnelle est aujourd'hui à la pointe de nombreux écosystèmes de programmation en génie logiciel.

1.2 Pourquoi cet état de l'art ?

Dans un monde où les logiciels deviennent de plus en plus complexes et gourmands en ressources, l'approche fonctionnelle apporte une réponse aux besoins modernes de la programmation. Beaucoup de langages impératifs contemporains ont intégré, au fil des années, l'approche fonctionnelle (Java 8, C++ 11, JavaScript). Les concepts de λ-expressions, de fermetures, d'évaluation paresseuse, ou bien de fonctions d'ordre supérieur sont reconsidérés. Le paradigme fonctionnel est utilisé dans l'industrie de pointe et enseigné dans de nombreux établissements d'enseignement supérieur. C'est pourquoi nous avons décidé de donner une vue d'ensemble sur le développement historique de ce paradigme, en priorisant ses concepts les plus importants.

1.3 Plan du document

Nous verrons, dans la section 2, les principaux aspects de la programmation fonctionnelle (λ-calcul, les fonctions d'ordre supérieur, l'évaluation paresseuse, les effets de bord). Les limites de ces aspects de la programmation fonctionnelle, nous amènerons à analyser, dans la section 3, quels sont les impacts d'une telle approche pour la nouvelle génération de logiciels.

2 Les principaux concepts du paradigme fonctionnel

2.1 Les origines et le lien avec le λ-calcul

Le λ-calcul est un système formel vu comme le socle des langages fonctionnels. Contrairement à ce que l'on pourrait penser, le λ-calcul n'est pas le premier

des langages de programmation fonctionnels [12]. Il a été reconnu pour sa compatibilité avec le paradigme fonctionnel et est venu s'intégrer dans le coeur de l'implémentation des langages fonctionnels.

Néanmoins, la notion de définissabilité dans le λ-calcul a formé une base pour le progrès de paradigme fonctionnel. C'est ce qui a permis de formaliser les idées de calcul effectif, la calculabilité, la récursivité, faire évoluer les stratégies d'évaluation parallèle [17], et introduire les λ-expressions dans les langages de programmation. Cette formalisation a donné naissance à des compilateurs plus efficaces et des environnements de développement plus complets et sophistiqués. De plus, les notions de fonctions anonymes, variables, et applications dans le λ-calcul ont formé la fondation des langages de programmation fonctionnels purs.

Certes, le λ-calcul tel qu'inventé par Church avait plus d'impact sur la logique mathématique et la représentation des calculs, que sur le paradigme fonctionnel, mais plusieurs langages de programmation fonctionnelle se sont inspirés de ce modèle, notamment dans l'intégration des stratégies de réduction gloutonne ou paresseuse qui viennent de l'ordre normal et applicatif du lambda calcul, ou bien le typage et le non-typage des variables qui s'inspirent des deux catégories de λ-calcul ; typés et non typés [12].

2.2 Fonctions pures et transparence référentielle

La définition d'un langage fonctionnel n'est pas quelque chose de strictement précise et bien que le sujet soit toujours d'actualité, les critères fondamentaux et universellement reconnus des langages fonctionnels sont que les fonctions sont des fonctions pures et l'absence d'effets de bord dans le langage [17].

En programmation, on dit qu'une fonction produit un effet de bord quand elle modifie un élément en dehors de l'environnement local de celle-ci. Une fonction qui ne comporte aucun effet de bord est appelé une fonction pure, c'est-à-dire que si on lui donne toujours les mêmes paramètres d'entrée, elle retournera toujours le même résultat et l'état de l'environnement sera toujours le même.

Un exemple en JavaScript : la première fonction est pure, alors que la seconde est impure, l'exécuter deux fois à l'identique produira un résultat différent.

```
//fonction pure
function square(x) => {
    return (x*x)
}

let y = 10;
//fonction impure
function addToY(x) => {
    y += x;
    return y;
}
```

Les fonctions pures permettent d'obtenir la transparence référentielle, c'est-à-dire que dans l'application d'une expression, on peut remplacer n'importe quel

terme par un autre terme tant qu'il renvoie le même résultat, l'expression restera inchangée.

C'est une explication du succès à retardement de la programmation fonctionnelle. Les programmes devenant de plus en plus complexes, la programmation fonctionnelle, dénuée d'effets de bord permet d'obtenir plus facilement un code maintenable, on peut facilement réécrire une fonction ou refactoriser une partie du code en s'assurant de l'intégrité du reste.

De plus, avec la popularité des tests unitaires, les fonctions pures s'y prêtent parfaitement puisque leur comportement est très prévisible et permet de faire abstraction du contexte extérieur.

2.3 Polymorphisme et fonctions d'ordre supérieur

A l'instar de l'absence d'effet de bord, les fonctions d'ordre supérieur sont incontournables en programmation fonctionnelle.
Ce sont les fonctions remplissant au moins l'une de ces deux caractéristiques :

— elles prennent une ou plusieurs fonctions en arguments ;
— elles renvoient une fonction.

Toutes les autres fonctions sont des fonctions du premier ordre.
Un exemple en JavaScript d'une fonction d'ordre supérieur qui prend une fonction et un nombre en paramètre et applique deux fois la fonction sur le nombre.

```
const twice = f => x => f(f(x));
```

Un des intérêts des fonctions d'ordre supérieur est la possibilité de curryfier les fonctions, en transformant des fonctions à plusieurs arguments à une fonction retournant une fonction sur les arguments restants.

Les fonctions d'ordre supérieur et les fonctions curryfiées offrent donc plus de modularité au code puisqu'on peut facilement modifier une fonction sans devoir tout réécrire puisqu'il suffit de modifier une des fonctions en paramètre pour changer son comportement.

Couplé avec la transparence référentielle, cela transforme la factorisation du code en un jeu d'assemblage. Il suffit simplement de changer les appels et les interactions des fonctions entres elles sans avoir à les réécrire.

Encore une fois, la programmation fonctionnelle est en avance sur son temps puisque maintenant, plus de 30 ans après la création d'un langage comme Haskell, les fonctions d'ordre supérieures sont maintenant intégrées dans tous les langages, même les langages qui ne sont pas fonctionnels de base comme Java, Python ou C++.

2.4 Les promesses

Les promesses, aussi appelées « futurs » sont des techniques de synchronisation utilisées dans les langages concurrents. Les promesses trouvent leurs origines dans la programmation fonctionnelle car elles utilisent le système de clôture de ce paradigme.

Une promesse fonctionne en créant une clôture, dont l'évaluation sera retardée, elle utilise également le système de callbacks.

Plus simplement, une promesse peut être vue comme une fonction qui prend une ou plusieurs autres fonctions en paramètre, ces fonctions sont appelées callbacks. Lorsque le traitement de la fonction principale est terminé, un des callbacks est alors appelé. Par exemple, une fonction qui compte jusqu'à 1000000, cette fonction bouclera jusqu'à atteindre la millionième itération, une fois celle-ci atteinte, le callback de succès sera renvoyé.

En outre les promesses servent à faire de l'asynchrone [1], elles sont donc utilisées pour les traitements lourds et/ou longs. Un exemple courant est un site qui doit télécharger une longue liste d'utilisateurs au chargement de la page via une api externe. Ce chargement prend du temps, et nous voulons donc un écran de chargement qui soit présent le temps du traitement, il nous suffit de faire une promesse, et lorsque la liste est entièrement chargée, nous pouvons enlever la page de chargement.

Voici des exemples d'illustrations des promesses en JavaScript :

Tout d'abord la création d'une promesse : Ici, on peut voir qu'un traitement asynchrone est effectué, une promesse reçoit deux callbacks, un callback resolve() et un callback reject(), la méthode resolve() est appelée une fois que le traitement est terminé et valide. Si une erreur se produit, la méthode reject() sera appelée.

```
const unePromesse = new Promise( (resolve, reject) => {
    let traitementAsynchoneOk = ....
    if ( traitementAsynchoneOk ) resolve("Ok")
    else reject("Une erreur s'est produite")
})
```

Il existe plusieurs façons d'utiliser les promesses en JavaScript. Tout d'abord, la première disponible était la version ci-dessous, cette méthode permet simplement d'appeler une promesse, lorsque le traitement asynchrone de la promesse est terminé, le code qui se trouve dans le then() est exécuté, then() possède un callback qui a en paramètre la valeur souhaitée de la promesse. Si une erreur survient lors du traitement de la promesse, ce sera le catch() qui sera executé à la place du then().

```
const traitementPromesse = () => {
    unePromesse
        .then( response => console.log(response) )
        .catch( err => console.log(err) )
}
```

Bien que cette méthode soit claire, elle possède un souci, le résultat d'une promesse peut également être une promesse, dans les faits cela arrive régulièrement et nous avons alors affaire à des promesses chaînées qui nous donne donc

une cascade de .then() imbriqués, lorsque ce nombre de promesses chaînées est élevé, le code devient alors moins lisible, la programmation plus laborieuse et la maintenance plus difficile. Une solution à ce problème a été ammené à partir de ECMAScript 2017 avec l'utilisation des mots-clés async et await. Cette nouvelle approche permet de pouvoir chaîner les promesses avec plus de lisibilité mais n'apporte pas de nouvelles fonctionnalités.

```
const traitementPromesse = async () => {
    let reponseAsynchrone = await unePromesse()
    // La réponse du then(response) dans reponseAsynchrone
}
```

2.5 L'évaluation paresseuse et l'évaluation gloutonne

Dans le λ-calcul, l'ordre de l'évaluation n'est pas important pour le résultat final d'une expression, car selon la théorie de Church-Rosser tel que mentionné dans [12], la forme normale d'un lambda terme est unique si elle existe. Par contre, cet ordre impacte le temps que prend l'évaluation et la mémoire utilisée, et donc la performance d'un programme.
L'évaluation gloutonne ou comme appelé dans l'article [17] « Ordre applicatif », est la méthode la plus répandue et utilisée dans les langages de programmation. Cela est dû au fait qu'elle est Turing-complet [12], ainsi qu'à la difficulté qu'a posé l'implémentation de la méthode paresseuse, la difficulté étant l'incompatibilité de la réduction de graphe utilisée dans cette stratégie pour éviter la ré-évaluation des termes avec l'architecture de la machine.
La méthode gloutonne consiste tout simplement à évaluer l'argument de la fonction avant d'évaluer l'appel de la fonction, chose qui peut réduire considérablement le temps de calcul dans certains cas, prenons l'exemple d'une fonction qui illustre un de ces cas, la fonction prend un argument et calcul le carré de ce dernier :

```
    (lambda x. (* x x)) (+ 2 3)
=> (lambda x. (* x x)) 5
=> (* 5 5)
=> 25
```

On voit clairement que l'argument (+ 2 3) est évalué en premier avant l'appel à la fonction carré.
La méthode d'évaluation paresseuse aussi appelé « Ordre normal », consiste à évaluer l'appel à la fonction d'abord, et n'évalue l'argument que quand c'est nécessaire, si on applique la fonction de l'exemple précédent en changeant la stratégie de l'évaluation, ça donnera :

```
    (lambda x.(* x x)) (+ 2 3)
=> (* (+ 2 3) (+ 2 3))
=> (* 5 (+ 2 3))
=> (* 5 5 )
=> 25
```

On voit que l'évaluation gloutonne est plus performante que paresseuse dans ce cas, mais il en existe d'autre où en appliquant la méthode gloutonne l'expression ne converge pas ou fait des calculs non nécessaires par exemple :

```
// Eval.gloutonne :
   ( lambda x. 1) (+ 2 3)
=> ( lambda x. 1) 5
=> 1

// Eval.paresseuse :
   ( lambda x. 1) (+ 2 3)
=> 1
```

On constate que l'argument (+ 2 3) est évalué alors qu'on l'a pas utilisé dans le corps de l'appel à la fonction principale.

2.6 Compilation

Outre les problèmes et difficultés standards que propose l'exercice d'écrire un compilateur comme la non-ambiguïté ou l'optimisation, la programmation fonctionnelle est très différente des autres paradigmes et propose un défi supplémentaire afin de pouvoir implémenter les fonctionnalités du langage. La référence « Compiling a functionnal programming language »[20] illustre parfaitement ces problématiques, et montre en détails les fonctionnalités propres à la programmation fonctionnelle (comme la récursivité terminale) et comment ils ont pu être implantées.

A tout cela s'ajoutent les assistants de preuve comme Coq[8], qui en plus d'être des langages purement fonctionnels, permettent d'écrire des programmes mathématiquement prouvés, ce qui ajoute une nouvelle dimension à la compilation d'un programme, puisque celui-ci se retrouve à devoir composer avec de nouvelles règles afin de pouvoir prouver les programmes. Ceci réduit ainsi les champs d'action du langage puisqu'on doit respecter des contraintes qui ne sont pas imposées dans un autre langage sans vérification de preuves.

Ces fonctionnalités sont des avantages de la programmation fonctionnelle, permettant d'écrire des programmes sûrs et stables. De plus, l'écriture d'un compilateur étant un exercice de plus en plus complexe, les langages de plus haut niveau sont maintenant utilisés pour rendre la tâche moins difficile. Les langages fonctionnels offrent une modularité et une stabilité qui les rendent de plus en plus utilisés pour la rédaction des compilateurs avec la multiplication de compilateurs comme ocamllex et ocamlyacc par exemple.

2.7 Fonctionnel versus impératif

Les différences entre les deux paradigmes vont au-delà de l'abstraction et le contrôle de l'état, c'est une façon différente de penser et d'architecturer les programmes. On choisit d'aborder seulement dans cet article deux aspects qui nous semblent les plus distinctifs entre les deux paradigmes.

Abstraction Quand il s'agit d'implémenter un programme de plus haut niveau, le paradigme fonctionnel se positionne comme la meilleure option, grâce à la simplicité d'écriture de ces programmes et la concentration sur le « quoi », et non pas sur le « comment ».

L'abstraction dans le paradigme fonctionnel offre une facilité d'implémentation, de lecture, de modularité, et de réutilisation (en concevant un programme comme une composition de nombreuses petites fonctions), qu'on retrouve difficilement chez le paradigme impératif.

Contrôle de l'état Les langages impératifs sont caractérisés par le contrôle de changement d'état implicite d'une ou plusieurs variables. Ces langages en général adoptent la notion de la séquence d'instructions à l'aide des commandes comme l'affectation, et les boucles d'itération. Ce contrôle oblige les programmeurs à être plus prudents avec la gestion de la mémoire, la gestion des erreurs, et contrer les effets de bord.

De l'autre côté, les langages fonctionnels sont caractérisés par l'absence d'état implicite. Les modèles calculatoires adoptés sont la « fonction », et la programmation par expressions, où l'état est porté explicitement dans les fonctions et la séquence d'instructions est remplacée par la composition de fonctions et la récursion [17]. Le concept qui illustre ceci est la transparence référentielle expliquée dans la section 2.2.

Et l'orienté objet ? Une étude[14] a été faite pour comparer le langage Ocaml et le langage C++. La conclusion de cette étude était qu'il n'y avait pas de différences fondamentales en termes de maintenabilité, de sûreté, de qualité de code, et d'efficience entre ces deux langages. Un choix entre ces deux langages ne peut donc pas être rationnel, et doit être basé sur la subjectivité et les préférences de chaque équipes de développement.

3 Impact et perspectives

3.1 Le fonctionnel et la programmation parallèle

Avec l'atteinte du pic de la fréquence avec laquelle un processeur peut effectuer des calculs, une nouvelle solution est apparue, c'est la distribution de charge et le calcul parallèle. Mais, pour être capable de faire du parallélisme, il est nécessaire d'avoir des programmes décomposé et les plus simples possible, car sinon, il y a de grandes chances d'avoir des parties inter-dépendante [19].
Un des avantages des langages fonctionnels, c'est que le parallélisme dans un programme fonctionnel est implicite grâce à la transparence référentielle et l'indépendance de l'ordre d'exécution sur le résultat final. La détection par le système et l'allocation aux processeurs est donc automatique [17], mais le problème est que rien ne garantit l'optimalité. Sachant que dans le cas général la stratégie optimale est indécidable, plusieurs heuristiques sont utilisées telle que le « load balancing » (équilibrage de charge ?), mais la question qui s'est posée et ce que les

chercheurs ont essayé de répondre c'est : « et si les programmeurs connaissent une stratégie de mapping optimale pour un programme s'exécutant sur une machine en particulier ? » [17], d'où le design des langages de programmations parafonctionnels. Ce sont des extensions de langages telle que des annotations pour assigner des expressions à des processeurs de la forme :

```
exp on proc
```

3.2 Le fonctionnel et le génie logiciel

Longtemps critiquée pour ses performances, la programmation fonctionnelle a aujourd'hui mis tout le monde d'accord. De nombreux langages sont maintenant multi-paradigmes et essaient donc de profiter des avantages associés à ses différents paradigmes[14].

L'approche consistant à privilégier la composition de fonctions dans le paradigme fonctionnel comporte comme atout la facilité de découper un problème en sous problèmes plus simples et bien spécifiés, qui eux-mêmes seront découpés en sous-sous problèmes, jusqu'à avoir des problèmes « atomiques » (dans le sens premier du terme, c'est-à-dire insécables). Cette approche permet donc de construire et d'appréhender plus facilement des programmes complexes.

Le JavaScript, qui est aujourd'hui l'un des langages de programmation les plus utilisés au monde, en est un parfait exemple. Il adopte de plus en plus le fonctionnel. La bibliothèque React [1] qui est bien connue pour aider les équipes de développement à créer de gros projets, est en grande partie basée sur la programmation fonctionnelle. En effet, le principe de React est de créer des composants, qui peuvent incorporer d'autres composants afin d'avoir une architecture très claire et explicite[2]. Ces composants React sont en fait des fonctions, et cela nous renvoie donc à l'un des principaux avantages de la programmation fonctionnelle mentionné précédemment.

4 Conclusion

L'intégration des principaux concepts du paradigme fonctionnel peut apporter beaucoup d'avantages dans le développement, la maintenance et dans la réalisation des tests d'intégration d'un logiciel. L'approche théorique de ce paradigme est, des fois, plus difficile à comprendre et à utiliser dans le développement. Certains de ses concepts sont très complexes, ayant une approche mathématique. Néanmoins, grâce à sa modularité [17] et flexibilité, ce paradigme a trouvé une place très importante dans l'industrie de génie logiciel, la développement d'applications de bureau, web et mobiles, des compilateurs, des jeux vidéos, dans l'intelligence artificielle etc. Les langages de programmation fonctionnels ou multi-paradigmes [6], ont beaucoup évolué au fil des années et

1. son « framework » le plus utilisé d'après `https://2020.stateofjs.com/fr-FR/technologies/front-end-frameworks`

se sont adaptés aux besoins des développeurs.

Le paradigme fonctionnel continue à être étudié et développé. De même, il continue à prendre de l'ampleur dans l'industrie. Maintenant, nous pouvons seulement nous demander, comment ce paradigme sera vu et utilisé dans les futurs écosystèmes de programmation ?

5 Références

[1] Simon Archipoff, David Janin et Bernard P. Serpette. « Des promesses, des actions, par flots, en OCaml ». In : *JFLA 2020 - 31ème Journées Francophones des Langages Applicatifs*. Sous la direction de Zaynah Dargaye et Yann Regis-Gianas. Gruissan, France, jan. 2020. URL : https://hal.archives-ouvertes.fr/hal-02389651 (visité le 18/02/2021).

[2] Alexander Sobolev et Sergey Zykov. « Functional Programming Patterns in JavaScript ». en. In : *Intelligent Decision Technologies 2019*. Sous la direction d'Ireneusz Czarnowski, Robert J. Howlett et Lakhmi C. Jain. Smart Innovation, Systems and Technologies. Singapore : Springer, 2020, pages 299-312. ISBN : 9789811383113. DOI : 10.1007/978-981-13-8311-3_26.

[3] Guillaume Claret. « Program in Coq ». Theses. Université Sorbonne Paris Cité, sept. 2018. URL : https://hal.inria.fr/tel-01890983 (visité le 18/02/2021).

[4] Mark Priestley. « AI and the Origins of the Functional Programming Language Style ». en. In : *Minds & Machines* 27.3 (sept. 2017), pages 449-472. ISSN : 0924-6495, 1572-8641. DOI : 10.1007/s11023-017-9432-7. URL : http://link.springer.com/10.1007/s11023-017-9432-7 (visité le 18/02/2021).

[5] Zhenjiang Hu, John Hughes et Meng Wang. « How functional programming mattered ». In : *National Science Review* 2.3 (sept. 2015), pages 349-370. ISSN : 2095-5138. DOI : 10.1093/nsr/nwv042. URL : https://doi.org/10.1093/nsr/nwv042 (visité le 18/02/2021).

[6] D. A. Turner. « Some History of Functional Programming Languages ». en. In : *Trends in Functional Programming*. Sous la direction de David Hutchison et al. Tome 7829. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013, pages 1-20. ISBN : 978-3-642-40446-7 978-3-642-40447-4. DOI : 10.1007/978-3-642-40447-4_1. URL : http://link.springer.com/10.1007/978-3-642-40447-4_1 (visité le 03/03/2021).

[7] Eric Thivierge et Marc Feeley. « Efficient compilation of tail calls and continuations to JavaScript ». en. In : *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming - Scheme '12*. Copenhagen, Denmark : ACM Press, 2012, pages 47-57. ISBN : 978-1-4503-1895-2. DOI : 10.1145/2661103.2661108. URL : http://dl.acm.org/citation.cfm?doid=2661103.2661108 (visité le 03/03/2021).

[8] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack et Laurent Théry. « Extending Coq with Imperative Features and Its Application to SAT Verification ». en. In : *Interactive Theorem Proving*. Sous la direction de Matt Kaufmann et Lawrence C. Paulson. Lecture Notes in Computer Science. Berlin, Heidelberg : Springer, 2010, pages 83-98. isbn : 978-3-642-14052-5. doi : 10.1007/978-3-642-14052-5_8.

[9] Konrad Hinsen. « The Promises of Functional Programming ». en. In : *Comput. Sci. Eng.* 11.4 (juil. 2009), pages 86-90. issn : 1521-9615. doi : 10.1109/MCSE.2009.129. url : http://ieeexplore.ieee.org/document/5076325/ (visité le 18/02/2021).

[10] Paul Hudak, John Hughes, Simon Peyton Jones et Philip Wadler. « A history of Haskell : being lazy with class ». en. In : *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. San Diego California : ACM, juin 2007. isbn : 978-1-59593-766-7. doi : 10.1145/1238844.1238856. url : https://dl.acm.org/doi/10.1145/1238844.1238856 (visité le 18/02/2021).

[11] Jacques Garrigue. « Relaxing the Value Restriction ». In : *The Third Asian Workshop on Programming Languages and Systems, APLAS'02, Shanghai Jiao Tong University, Shanghai, China, November 29 - December 1, 2002, Proceedings*. 2002, pages 31-45.

[12] Henk Barendregt. « The impact of the lambda calculus in logic and computer science ». en. In : (1997), page 36.

[13] John Greiner. « Weak polymorphism can be sound ». en. In : *Journal of Functional Programming* 6.1 (jan. 1996), pages 111-141. issn : 1469-7653, 0956-7968. doi : 10.1017/S0956796800001593. (Visité le 03/03/2021).

[14] R. Harrison, L.G. Samaraweera, M.R. Dobie et P.H. Lewis. « Comparing programming paradigms : an evaluation of functional and object-oriented programs ». en. In : *Softw. Eng. J. UK* 11.4 (1996), page 247. issn : 02686961. doi : 10.1049/sej.1996.0030. url : https://digital-library.theiet.org/content/journals/10.1049/sej.1996.0030 (visité le 18/02/2021).

[15] K.W. Ng et C.K. Luk. « A survey of languages integrating functional, object-oriented and logic programming ». en. In : *Microprocessing and Microprogramming* 41.1 (avr. 1995), pages 5-36. issn : 01656074. doi : 10.1016/0165-6074(94)00017-5. url : https://linkinghub.elsevier.com/retrieve/pii/0165607494000175 (visité le 03/03/2021).

[16] Philip Wadler. « The essence of functional programming ». In : *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '92. New York, NY, USA : Association for Computing Machinery, fév. 1992, pages 1-14. isbn : 978-0-89791-453-6. doi : 10.1145/143165.143169. url : https://doi.org/10.1145/143165.143169 (visité le 17/02/2021).

[17] Paul Hudak. « Conception, evolution, and application of functional programming languages ». en. In : *ACM Comput. Surv.* 21.3 (sept. 1989), pages 359-411. issn : 0360-0300, 1557-7341. doi : 10.1145/72551.72554.

URL : https://dl.acm.org/doi/10.1145/72551.72554 (visité le 18/02/2021).

[18] J. Hughes. « Why Functional Programming Matters ». In : *The Computer Journal* 32.2 (jan. 1989), pages 98-107. ISSN : 0010-4620. DOI : 10.1093/comjnl/32.2.98. URL : https://doi.org/10.1093/comjnl/32.2.98 (visité le 17/02/2021).

[19] S. L. Peyton Jones. « Parallel Implementations of Functional Programming Languages ». In : *The Computer Journal* 32.2 (jan. 1989), pages 175-186. ISSN : 0010-4620. DOI : 10.1093/comjnl/32.2.175. URL : https://doi.org/10.1093/comjnl/32.2.175 (visité le 17/02/2021).

[20] Luca Cardelli. « Compiling a functional language ». en. In : *Proceedings of the 1984 ACM Symposium on LISP and functional programming - LFP '84*. Austin, Texas, United States : ACM Press, 1984, pages 208-217. ISBN : 978-0-89791-142-9. DOI : 10.1145/800055.802037. URL : http://portal.acm.org/citation.cfm?doid=800055.802037 (visité le 28/02/2021).