Résumé de “Sur le pouvoir expressif des structures applicatives et monadiques indexées”

Jan Malakhovski*

IRIT, University of Toulouse-3 and ITMO University

Version: 1.05; October 25, 2019

Abstract

Il est bien connu que des constructions théoriques très simples telles que les structures `Either` (équivalent type théorique de l'opérateur logique “ou”), `State` (représentant des transformateurs d'état composables), `Applicative` (application des fonctions généralisée) et `Monad` (composition de programmes séquentielles généralisée), nommés structures en Haskell, couvrent une grande partie de ce qui est habituellement nécessaire pour exprimer avec élégance la plupart des idiomes informatiques utilisés dans les programmes classiques. Cependant, il est usuellement admis qu'il existe plusieurs classes d'idiomes couramment utilisés qui ne s'intègrent pas bien à ces structures, les exemples les plus remarquables étant les transformations entre arbres (types de données, dont l'utilisation doit s'appuyer soit sur les motifs généralisés soit sur une infrastructure de méta programmation lourde) et traitement des exceptions (qui sont d'habitude supposés nécessiter un langage spécial et une prise en charge de l'exécution).

Ce travail a pour but de montrer que beaucoup de ces idiomes peuvent, en fait, être exprimés en réutilisant ces structures bien connues avec des modifications mineures (le cas échéant). En d'autres termes, le but de ce travail est d'appliquer les principes du rasoir KISS (Keep It Stupid Simple) et/ou d'Occam aux structures algébriques utilisées pour résoudre des problèmes de programmation courants.

Techniquement parlant, ce travail a pour but de montrer que les généralisations naturelles de classes de types `Applicative` et `Monad` de Haskell, associées à la possibilité d'en faire des produits cartésiens, en produisent un cadre commun très simple pour exprimer de nombreuses choses pratiques, dont certaines sont des nouvelles méthodes très commodes pour exprimer des idées de programmation communes, tandis que les autres peuvent être vues comme systèmes d'effets. Sur ce dernier point, si l'on veut généraliser des exemples présentés dans une approche de la conception de systèmes d'effets en général, on peut alors considérer la structure globale de cette approche comme un cadre quasi syntaxique qui permet d'ériger une structure générale du cadre “mariage” [28] au dessus de différents systèmes d'effets adhérant aux principes de base. (Bien que ce travail ne soit pas trop approfondi dans la dernière, car il est principalement motivé par des exemples qui peuvent être immédiatement appliqués à la pratique de Haskell.)

Il convient toutefois de noter qu'en fait, ces observations techniques n'ont rien d'étonnant: `Applicative` et `Monad` sont respectivement des généralisations de composition fonctionnelle et linéaire des programmes; ainsi, naturellement, les produits cartésiens de ces deux structures doivent couvrir en grande partie ce que les programmes font habituellement.

1 Introduction

Les premiers ordinateurs programmables tels que Colossus (1943-1945) et même les premières révisions d'ENIAC (1945-1955) n'étaient pas des ordinateurs à programme enregistré et ne pouvaient être programmés qu'à l'aide de plugboards et de commutateurs mécaniques.

IBM 650 (1953-1969), le premier ordinateur fabriqué en série, utilisait un tambour magnétique comme mémoire (généralement chargé à partir de cartes perforées) et le code d'opération de chaque

*papers@oxij.org; preferably in English, with ”thesis” as a substring of the subject line

instruction (opcode) devait spécifier explicitement l'adresse de l'instruction suivante (les instructions de type `jump` des langages d'Assembly modernes sont executées de la même manière).

Le PDP-11 (1970-1990) semble être le premier ordinateur à gérer correctement les appels de sous-programmes imbriqués de profondeur non fixe (c'est-à-dire, prenant en charge la récursivité et la modularité arbitraire), même si des sous-routines simples étaient admises déjà sur l'ENIAC.

Ce que ces premiers exemples montrent, c'est que le concept même d'un programme essentiellement linéaire organisé en sous-routines modulaires et éventuellement récursives n'était pas pris en charge par le matériel jusqu'aux années 1970. Il est toutefois intéressant de noter que dès que le matériel a atteint le niveau approprié, UNIX et le langage de programmation C [6, 7, 22] sont nés. Le matériel équivalent et ces systèmes logiciels sont encore omniprésents.

(On pourrait argumenter que le seul grand changement dans l'architecture informatique couramment utilisée depuis les années 1970 est la vulgarisation de SIMD pour les calculs numériques. C'est le fait que presque tous les ordinateurs destinés aux consommateurs sont maintenant livrés avec des GPU prêts à l'emploi. On peut évoquer aussi la virtualisation du matériel, introduite pour la première fois sur IBM System/370 en 1972, puis oubliée jusqu'au milieu des années 2000, mais le support matériel pour la virtualisation arbitrairement imbriquée et l'utilisation logicielle de ces fonctionnalités, dont QubesOS [21] serait un bon exemple moderne, est en pratique plutôt absent au moment de la rédaction de cet ouvrage.)

L'histoire des langages de programmation de haut niveau commence avec FORTRAN initialement développé par John Backus pour IBM 704 vers 1956 et par LISP développé par John McCarthy chez MIT au peu près au même temps.

La famille FORTRAN de langages impérativement compilés munis de typage stricte, y compris ALGOL, C et leurs descendants peut être considérée comme, au début, une simple tentative pour créer un langage d'Assembly universel, avec ensuite un transfert-horizontal-de-gènes/incorporation de constructions de programmation structurées telles que les expressions conditionnelles comme `if-then-else`, boucles (FORTRAN 77), tableaux, modules (les deux à Fortran 90, deuxième à C++20), parfois mélangés à des constructions orientées objet de Simula (dont C++ est le premier exemple) et après 50 années, à des idées de programmation fonctionnelle (C++11 et ultérieur).

La famille LISP de langages fonctionnels interprétés et typés dynamiquement se développait au contraire en partant du λ-calcul proposé par Alonzo Church et ses étudiants dans les années 1930 et 1940, dans le but explicite de créer un formalisme informatique minimaliste universel [2, 3] et de construire au dessus. Aux fins de cette discussion, les deux caractéristiques les plus importantes de LISP étaient la capacité de *déclarer* de nouvelles constructions de langage à l'aide de "formes spéciales" (qui étaient, en réalité, des fonctions partiellement paresseuses dans un langage à évaluation glouton) et la capacité de décrire ses propres programmes (réflexion). Cette dernière propriété signifiait que la génération de code d'exécution et la méta-programmation étaient faciles et que, plus important encore, le langage pouvait s'interpréter de manière triviale, permettant ainsi des extensions arbitraires. Le résultat final est que la plupart des variantes du LISP à ce jour peuvent évaluer les termes des autres variantes.

Divers mélanges des deux approches sont apparus au fil des ans. Deux familles remarquables sont:

- des langages impératifs (généralement) interprétés à typage dynamique, commençant par Smalltalk et représentés par Python, Ruby, JavaScript, entre autres; et
- des langages fonctionnels (généralement) compilés statiquement typés, commençant par ML et représentés par les modernes OCaml, SML et Haskell, entre autres.

Parmi ceux-ci, la séquence de langages LISP $\rightarrow$ ML $\rightarrow$ Miranda $\rightarrow$ Haskell est plutôt intéressante, car le passage de LISP à ML a remplacé le typage dynamique par un système de types polymorphes et par le syntaxe infixe au prix de la perte de formes spéciales et de la réflexion, le passage à Miranda a basculé vers l'évaluation paresseuse par défaut (donnant ainsi la plupart de ce que font les formes spéciales), et le passage à Haskell a ajouté des classes de types (donnant ainsi beaucoup de ce que font les types dynamiques) et a réintroduit la réflexion, parmi beaucoup d'autres choses.

En d'autres termes, Haskell a été conçu pour exprimer aisément les éléments couramment abordés dans la théorie des langages de programmation (PLT), car ses termes ressemblent à ceux utilisés dans

les mathématiques au niveau de l'école, le caractère strict permet (mais ne garantit pas) l'efficacité, et en plus il possède suffisamment de morceaux de LISP et de systèmes de types plus puissants (tels que des types dépendants) pour exprimer (ou au moins indiquer comment ils pourraient être exprimés) des concepts applicables à des pans entiers de langages de programmation. Et en effet, la plupart de la littérature citée dans cet ouvrage utilise Haskell ou une variante de ML.

Haskell est également étonnamment populaire pour une langue "académique" qui reste constamment dans le Top-50 de TIOBE Index [26] (mesures de recherche), avec son référentiel de code public le plus populaire, Hackage [5], répertoriant plus de 5000 packages.

Comme une remarque à part, la manière habituelle d'expliquer pourquoi les langages impératifs (tels que FORTRAN, ALGOL, C) ont "gagnés" contre LISP est de noter que ce dernier nécessite trop de transistors (dans le processeur etc.) pour pouvoir être évalué à une vitesse acceptable. Lorsque FORTRAN utilisait un seul opération `add` des langages d'Assembly, LISP-machine nécessitait énormément de vérifications de types en cours d'exécution.

Ensuite, le regain de popularité des descendants de Smalltalk tels que Python, Ruby, JavaScript à la fin des années 1990 et au début des années 2000 peut s'expliquer par leur similarité sémantique générale avec les descendants de FORTRAN mais avec des niveaux de satisfaction plus élevés des programmeurs (syntaxe plus simple sans signatures de type explicites, gestion automatique de la mémoire, etc) et, d'autre part, l'augmentation du nombre de transistors disponibles sur un processeur grand public, suivie de l'avènement de la compilation juste-à-temps (JIT). Notez cependant que le code le plus performant pour les systèmes écrits dans ces langages est toujours implémenté en C et en FORTRAN pour être appelé par lesdits interprètes via une interface de fonction étrangère (FFI). Par exemple, NumPy [19], une bibliothèque Python pour les calculs numériques hautes performances (et probablement la bibliothèque Python la plus connue dans les cercles universitaires), est une Pythonic "wrapper" sur un tas de code C (et un peu de code FORTRAN, traduit en C).

Le regain d'intérêt pour la programmation fonctionnelle dans la seconde moitié des années 2000 s'accompagne toutefois de l'apparition de techniques de compilation qui les rendent utilisables dans des systèmes de logiciel hautes performances. Cela permet notamment à certaines de ces langues de produire des systèmes complets ou presque complets style "mono-langage full-stack". Comme exemple, on peut prendre le projet MirageOS [13], un système d'exploitation modulaire entièrement écrit en ML. De même, les bibliothèques standard Go [24], Haskell [4] et Rust [25] tentent également de limiter leur utilisation des FFI. Ce qui, bien sûr, peut être perçu comme un avantage ("Yay! Code lisible dans un langage sain et sûr!") par contre certains langages utilisent beaucoup de C FFI dans leur bibliothèques standard (par exemple, Python), ce que peut être vu comme défaut ("Uhg! Maintenant chaque infrastructure remet tout en œuvre à partir de zéro!").

Notez cependant que les CPU classiques sont essentiellement des interpréteurs de code machine (séquences d'opcodes) compilés en matériel (les traces métalliques et les portes semi-conductrices sont ensuite "interprétées" par les lois physiques de l'électromagnétisme). C'est pourquoi les langages les plus proches d'Assembly sont plus faciles à compiler de manière à ce que les programmes de langage source sémantiquement efficaces soient compilés en programmes opcode efficaces à évaluer sur ces machines. Les GPU évoqués ci-dessus, d'abord commercialisés sous le nom d' "accélérateurs graphiques", sont désormais considérés comme une pièce essentielle de la machinerie moderne, rendant notamment viables les techniques modernes de rendu et de traitement des images. Par conséquent, il serait intéressant de voir des systèmes logiciels développés spécifiquement pour les ordinateurs avec "accélérateurs FPGA", car les réductions de graphes effectuées par les interprètes de langages de programmation fonctionnels peuvent être beaucoup plus efficaces sur de telles machines (par exemple, voir le projet Reduceron [17, 18]).

Autrement dit, il n'est pas tout à fait évident que les descendants de FORTRAN "gagneraient" encore aux systèmes informatiques fonctionnant dans un avenir rapproché, car les programmes pour ordinateurs dotés de calculs réversibles (comme ceux basés sur l'électromagnétisme brut et les ordinateurs quantiques) sont tout à fait fonctionnels [1, 23], il serait donc peut-être plus efficace et plus simple sur le plan cognitif de mettre en œuvre ces systèmes en langues fonctionnelles de haut en bas.

Dans tous les cas, ce travail traite de calculs un peu plus conventionnels. Les principales structures algébriques discutées dans ce travail sont les `Monad`es introduits dans la programmation fonctionnelle

à partir de la théorie de catégories par Moggi [15, 16] et popularisés par Wadler [27], ainsi que les `Applicative` `Functor`s introduits par McBride et Paterson [12]. Ces deux structures peuvent être considérées comme des généralisations simples des compositions de programmes linéaires et fonctionnelles, à savoir des généralisations des opérateurs “point-virgule” et “appel de fonction”.

2 Le résumé étendu

Si vous demandez à un programmeur Haskell pratiquant de décrire succinctement les classes de types `Applicative` et `Monad` à un autre programmeur pratiquant (lui dans un langage impératif), quelque chose comme “un opérateur d’application/d’appel à fonction surchargée” et un “opérateur de point-virgule surchargé” serait probablement entendu. Ces structures sont utiles pour quelques raisons.

- Premièrement, l’utilisation d’opérateurs génériques réduit quelque peu le passe-partout habituel en autorisant des combinateurs génériques (par exemple, `mapM`).
- Deuxièmement, et plus important encore, ces structures fournissent un niveau d’abstraction commode qui cache des détails non pertinents (dont le `Either` `Monad` qui cache la `Left` moitié du calcul jusqu’à ce qu’il devienne pertinent en est un excellent exemple).

En ce que concerne les opérateurs `call` et `ret` de la plupart des langages d’Assembly classiques, un programmeur en microcode CPU (ou un langage d’Assembly RISC suffisant) pourrait vous demander pourquoi vous avez même besoin de ces instructions alors que vous pouvez simplement `push`/`pop` les pointeurs et utiliser `jump`. De même, un programmeur pour IBM 650 pourrait affirmer que même le séquençage linéaire des instructions et le pointeur d’instruction sont superflus, chaque instruction pouvant simplement spécifier explicitement l’adresse de la prochaine instruction. De même, pour `Applicative` and `Monad`, bien que l’on puisse simplement utiliser explicitement des implémentations particulières `(<*>)` et `(>>=)`, le fait d’avoir ces opérateurs pour représenter un niveau d’abstraction encore plus élevé peut être encore plus pratique. (Cependant, il peut être problématique de montrer cette commodité à un programmeur habitué à une langue ne disposant pas des moyens suffisants d’expression, comme avec `Either` `Monad`.)

Il est toutefois intéressant de noter que, après avoir expliqué pourquoi `Applicative` et `Monad` sont utiles et qu’ils sont effectivement très populaires dans les programmes Haskell, on se heurtera au fait qu’apparemment, il n’y a pas beaucoup d’instances de ces structures qui soient applicables. En fait, `Either` et `State` reunis semblent couvrir presque tout:

- les calculs qui risquent d’échouer s’enroulent généralement dans `Either`,
- une fonction `main` dans un programme Haskell, plus ou moins, interprète simplement une transformateur de `State` de `RealWorld` qui calcule les sorties du programme à partir des entrées de programme (c’est-à-dire, `IO` `Monad`, bien qu’il puisse avoir d’autres interprétations),
- la plupart des autres choses sont soit des cas particuliers de ces deux (par exemple, `Maybe`), ses compositions (l’analyseur syntaxique, par exemple, est simplement une composition de `State` et `Either` avec `Stream` au lieu de `RealWorld`), ou les résultats de ses transformations “mécaniques” (par exemple, codés selon Scott).

Le fait que `Either` et `State` `Applicative`s et `Monad`s puissent exprimer tant rend encore plus intéressant de regarder de près les choses fréquemment utilisées qu’apparemment, au contraire, ils *ne peuvent pas* exprimer.

Premièrement, notons qu’hormis le pur `Either` et ses cas particuliers, Haskell fournit de nombreux autres mécanismes de traitement des erreurs, notamment des exceptions imprécises et plusieurs classes de types différents prétendant implémenter le générique `throw` et `catch` avec une sémantique légèrement différente.

Deuxièmement, le type `State s a = s -> (a, s)` utilise un seul type `s` des deux côtés de la flèche. Si l’on doit adopter un point de vue fondamentaliste selon lequel tous les calculs ne sont que des compositions de transformateurs d’états et doivent être exprimés en tant que tels, il apparaît

alors immédiatement que `State` est trop restrictif pour le cas d'utilisation générale, car il ne peut pas exprimer les transitions d'états entre types de données arbitraires.

En d'autres termes, même si un programmeur fondamentaliste de Haskell pouvait se sentir satisfait en transformant les `Stream`s (en particulier, `String`s) en types de données à l'aide d'une bibliothèque des combinateurs d'analyseurs syntaxiques telle que Parsec [11], il/elle serait obligé de succomber à plusieurs approches différentes de la gestion des erreurs en établissant les correspondances (pattern-matching) de types de données manuellement ou avec des bibliothèques telles que SYB [10], Uniplate [14], Multiplate [20] et Lenses [8, 9].

Ce qui ne veut pas dire que faire toutes ces choses est intrinsèquement mauvais, mais il est intéressant de voir à quel point on peut faire avec juste `Either`, `State`, `Applicative` et `Monad` et leurs extensions naturelles, c'est-à-dire qu'il est intéressant de voir combien peut être fait avec des constructions théoriques très basiques et leurs combinaisons. Le but de ce travail est de montrer que l'ensemble des choses exprimables à l'aide de ces structures est étonnamment vaste. Ou, plus spécifiquement, pour montrer que *tous* les problèmes communément pensés nécessitant les précautions particulières mentionnées ci-dessus peuvent en fait être résolus en réutilisant ces structures bien connues avec des modifications mineures (le cas échéant).

3 Contributions

Chaque élément de la liste suivante, à notre connaissance, constitue une contribution clé.

- Nous avons établi que les operateurs

  ```
  throw :: e -> c a
  catch :: c a -> (e -> c a) -> c a
  ```

  sont les cas speciaux des operateurs `Monad`iques `pure` (`return`) et `(>>=)` (`bind`)

  ```
  pure :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  ```

  (on peut substituer $[a \mapsto e, m \mapsto \lambda\_.c\ a]$ dans ses types).

- Par conséquent, un type `c e a` de calcul avec deux indexes, dont `e` indique un type d'erreurs et `a` indique un type de valeurs, peut être vu comme `Monad` deux fois: une fois par rapport à `e` et une fois par rapport à `a`.

  ```
  class ConjoinedMonads c where
    pure  :: a -> c e a
    (>>=) :: c e a -> (a -> c e b) -> c e b

    throw :: e -> c e a
    catch :: c e a -> (e -> c f a) -> c f a
  ```

  De plus, pour une telle structure, le `throw` est un zéro à gauche pour `(>>=)` et le `pure` est un zéro à gauche pour `catch`.

- Nous montrons que le type de `catch` ci-dessus est le type le plus général pour toute structure `Monad`ique `\a -> c e a` avec des opérateurs de `throw` et de `catch` supplémentaires satisfaisant la sémantique opérationnelle classique (via une simple unification des types pour plusieurs équations qui découlent de la sémantique de ces opérateurs). Ou bien, de manière duale, nous prouvons que `(>>=)` a le type le plus général pour exprimer des calculs séquentiels pour la structure `Monad`ique `\e -> c e a` (avec des opérateurs nommés `throw` et `catch`) avec des opérateurs additionnels `pure` et `(>>=)` satisfaisant la sémantique opérationnelle conventionnelle.

- La substitution de `Constant Functor` pour `c` dans `ConjoinedMonads` ci-dessus (c'est-à-dire, en fixant le type d'erreur) génère la définition de `MonadError` et, avec quelques redéfinitions équivalentes, `MonadCatch`. De la même façon, `IO` avec des redéfinitions similaires est aussi une instance de ConjoinedMonads (avec les mises en garde habituelles).

- `ExcepT` et quelques autres structures concrètes moins connues et potentiellement nouvelles ont des opérateurs de ce type et leurs correspondances sémantiques (ou peuvent être définis de manière équivalente de sorte que la partie centrale de la structure résultante correspond alors) exactement à la sémantique de `Monad`.

- Le type-class Monad a une représentation "fish" bien connue, où l'opérateur "`bind`" `(>>=)` est remplacé par l'opérateur "`fish`"

  ```
  (>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
  ```

  et les lois de `Monad` sont juste les lois de monoïde ordinaire.

  Par conséquent, toutes ces structures peuvent être vues comme des paires de monoïdes sur des types bi-indexés avec des éléments d'identité pour les `bind`s respectifs, et comme des zéros de gauche pour les `bind`s conjoints.

  Nous trouvons cette symétrie hypnotique et la généralisons plus loin dans une observation qu'il s'agit d'un exemple de produits cartésiens de classes de types.

- La réponse à "Pourquoi personne ne l'a-t-il pas remarqué déjà?" semble être que cette structure ne peut pas être bien exprimée en Haskell.

- En même temps, il a au moins plusieurs instances utiles:

  - Des combinateurs d'analyseurs syntaxiques (parser combinators) qui décrivent précisément les erreurs qu'ils produisent et qui réutilisent les combinateurs `Monad`iques courants pour l'analyse et le traitement des erreurs. Par exemple, le type `many` pour un tel combinateur d'analyseur syntaxique garantit qu'il ne peut pas renvoyer d'erreur

    ```
    many :: c e a -> c f [a]
    ```

    (parce que `f` ici est arbitraire, il ne peut pas être réduit à une valeur particulière) et

    ```
    choice :: [c e a] -> c e a
    ```

    est une instance de `sequence` combinateur `Monad`ique.
  - Les exceptions conventionnelles exprimées à l'aide de `Reader Monad` et de `callCC` de second rang (dont l'idée semble être nouvelle).
  - Erreur-explicite `IO`, ces dernières et des structures similaires ayant une motivation similaire ont déjà été proposées, mais ils n'ont pas utilisé le fait que leur "autre moitié" est aussi une `Monad`e.

- Nous remarquons que de nombreuses structures pratiquement intéressantes peuvent être décrites comme un produit cartésien d'erreurs de traitement de structure et de calculs de traitement de structure, ce qui suggère comme une direction intéressante la conception du langage de programmation.

- Nous remarquons que de nombreux calculs avec `Applicative` peuvent être interprétés comme fournissant un *mécanisme* permettant de construire un type de données avec des "ports" "connectables" par des sous-calculs. Nous observons que c'est cette propriété qui les rend tellement plus appropriées en pratique que la manière habituelle de construire les mêmes calculs en utilisant une composition conventionnelle.

- Nous distillons cette observation en une structure algébrique plus générale de (et/ou technique d'expression) des calculs de type "quasi-`Applicative`" et pouvons présenter plusieurs autres instances (c'est-à-dire non-`Applicative`) de cette structure, qui comprend des curieuses structures de "familles" fonctionnant avec des types de données codés selon Scott comme s'il s'agissait de listes hétérogènes de valeurs typées.
- Ensuite, nous montrons qu'il existe en fait une famille infinie de telles structures de type "quasi-`Applicative`". Cette famille peut être décrite succinctement comme une famille de calculs pour des machines au multi-pile généralisés avec des types de données arbitraires et/ou fonctions comme des "piles".
- Ensuite, nous observons que notre "quasi-`Applicative`" est en réalité une généralisation naturelle de l' `Applicative` conventionnel au niveau de types dépendants.
- Nous remarquons que combinateurs d'analyseurs monadiques peuvent être généralisés en `Monad`es indexées, permettant ainsi d'analyser (transformer entre) des types de données/arbres arbitraires.

References

[1] Thorsten Altenkirch and Jonathan Grattage. "A functional quantum programming language". In: (2005). arXiv: quant-ph/0409065 (cit. on p. 3).

[2] Henk Barendregt. "The Impact of Lambda Calculus in Logic and Computer Science". In: (1997). URL: http://www-users.mat.umk.pl/~adwid/materialy/doc/church.pdf (cit. on p. 2).

[3] Felice Cardone and J. Roger Hindley. *History of Lambda-calculus and Combinatory Logic*. 2006. URL: http://www.users.waitrose.com/~hindley/SomePapers_PDFs/2006CarHin,HistlamRp.pdf (cit. on p. 2).

[4] GHC Project Authors. *Hackage: The base package, version 4.9.0.0*. 2016. URL: https://hackage.haskell.org/package/base-4.9.0.0 (cit. on p. 3).

[5] *Hackage: Haskell Central Package Archive*. 2018. URL: https://hackage.haskell.org/ (cit. on p. 3).

[6] S. C. Johnson and B. W. Kernighan. *The Programming Language B*. Technical report 8. Murray Hill, NJ, USA: Bell Laboratories, 1973 (cit. on p. 2).

[7] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Upper Saddle River, NJ 07458, USA: Prentice-Hall, 1978, pp. x + 228. ISBN: 0-13-110163-3 (cit. on p. 2).

[8] Edward Kmett. *Lenses, Folds and Traversals*. 2013. URL: http://lens.github.io/ (cit. on p. 5).

[9] Edward Kmett et al. *Hackage: The lens package, version 4.17*. 2018. URL: https://hackage.haskell.org/package/lens-4.17 (cit. on p. 5).

[10] Ralf Lämmel and Simon Peyton Jones. "Scrap your boilerplate: a practical approach to generic programming". In: ACM Press, Jan. 2003, pp. 26–37. URL: https://www.microsoft.com/en-us/research/publication/scrap-your-boilerplate-a-practical-approach-to-generic-programming/ (cit. on p. 5).

[11] Daan Leijen, Paolo Martini, and Antoine Latter. *Hackage: The Parsec package, version 3.1.11*. 2016. URL: https://hackage.haskell.org/package/parsec-3.1.11 (cit. on p. 5).

[12] Conor McBride and Ross Paterson. "Applicative Programming with Effects". In: *Journal of Functional Programming* 18.1 (2008), pp. 1–13. URL: http://www.soi.city.ac.uk/~ross/papers/Applicative.pdf (cit. on p. 4).

[13] MirageOS Project Authors. *MirageOS: A programming framework for building type-safe, modular systems*. 2019. URL: https://mirage.io/ (cit. on p. 3).

[14] Neil Mitchell and Colin Runciman. *Uniplate*. 2007. URL: http://community.haskell.org/~ndm/uniplate/ (cit. on p. 5).

[15] Eugenio Moggi. "Computational λ-Calculus and Monads". In: *Logic in Computer Science (LICS)*. June 1989, pp. 14–23. URL: http://www.disi.unige.it/person/MoggiE/ftp/lics89.ps.gz (cit. on p. 4).

[16] Eugenio Moggi. "Notions of computation and monads". In: *Information and Computation* 93.1 (1991). URL: http://www.disi.unige.it/person/MoggiE/ftp/ic91.pdf (cit. on p. 4).

[17] Matthew Naylor and Colin Runciman. "The Reduceron reconfigured and re-evaluated". In: *Journal of Functional Programming* 22.4–5 (Sept. 2012), pp. 574–613. ISSN: 0956-7968 (print), 1469-7653 (electronic). DOI: 10.1017/S0956796812000214. URL: https://www.cambridge.org/core/product/9818E081664ADAFE9F61F1AEDAD0B043 (cit. on p. 3).

[18] Matthew Naylor, Colin Runciman, and Jason Reich. *The Reduceron*. 2010. URL: https://www.cs.york.ac.uk/fp/reduceron/ (cit. on p. 3).

[19] NumPy Project Authors. *NumPy: Scientific Computing with Python*. 2019. URL: https://numpy.org/ (cit. on p. 3).

[20] Russell O'Connor. *Hackage: The multiplate package, version 0.0.3*. 2015. URL: https://hackage.haskell.org/package/multiplate-0.0.3 (cit. on p. 5).

[21] Qubes OS Project Authors. *Qubes OS: A reasonably secure operating system*. 2019. URL: https://www.qubes-os.org/ (cit. on p. 2).

[22] Dennis M. Ritchie and Ken Thompson. "The UNIX time-sharing system". In: *Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973*. Ed. by ACM. New York, NY, USA: ACM Press, 1973, ??–?? URL: https://www.bell-labs.com/usr/dmr/www/cacm.html (cit. on p. 2).

[23] Amr Sabry. *Topics in Programming Languages: Reversible and Quantum Computing*. 2011. URL: https://www.cs.indiana.edu/~sabry/teaching/b629/s11/ (cit. on p. 3).

[24] The Go Programming Language Project Authors. *The Go Programming Language: Standard Library Packages*. 2019. URL: https://golang.org/pkg/ (cit. on p. 3).

[25] The Rust Programming Language Project Authors. *The Rust Standard Library*. 2019. URL: https://doc.rust-lang.org/std/ (cit. on p. 3).

[26] *TIOBE Index*. 2019. URL: https://www.tiobe.com/tiobe-index/ (cit. on p. 3).

[27] Philip Wadler. "The essence of functional programming". In: *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Albuquerque, New Mexico, January 19–22, 1992*. Ed. by ACM. ACM order number 54990. New York, NY, USA: ACM Press, 1992, pp. 1–14. ISBN: 0-89791-453-8 (cit. on p. 4).

[28] Philip Wadler and Peter Thiemann. "The marriage of effects and monads". In: *ACM Transactions on Computational Logic* 4.1 (Jan. 2003), pp. 1–32. URL: http://homepages.inf.ed.ac.uk/wadler/papers/effectstocl/effectstocl.ps.gz (cit. on p. 1).