Université d'Orléans

Licence Informatique – Parcours U-GPEx MINERVE

Immersion en laboratoire

de février à mai 2025

Rapport et compte-rendu à propos d'OCamlFRP

Nom : Nicolas Paul
Encadrant : Frédéric Dabrowski

Résumé

Ce rapport présente le travail réalisé dans le cadre du programme MINERVE, centré sur l'exploration d'une bibliothèque de programmation réactive fonctionnelle. L'objectif principal consistait à utiliser et composer des programmes reposant sur le paradigme afin d'en analyser les mécanismes, d'en documenter l'usage, et d'identifier des potentielles améliorations. Le document revient ainsi les connaissances acquises, les développements expérimentaux effectuées, ainsi que des critiques sur les forces et les limites de la bibliothèque.

Remerciements

Je tiens à exprimer ma profonde gratitude à l'équipe du programme MINERVE pour l'opportunité d'une immersion en laboratoire, ainsi qu'à l'équipe du LMV, en particulier à Frédéric Dabrowski, pour son encadrement attentif tout au long de ce projet. Je remercie également le personnel éducatif de l'université, ainsi que Thomas Théault pour ses conseils éclairés.

Table des matières

1 Introduction

1.1 Le parcours Excellence MINERVE

Le programme universitaire d'excellence MINERVE, porté par l'Université d'Orléans et ses partenaires dans le cadre du projet FRANCE 2030, vise à former des chercheurs à la pointe des disciplines scientifiques contemporaines. Il se distingue par une approche rigoureuse et pluridisciplinaire des problématiques de recherche.

Le premier semestre de licence propose une découverte appronfondie de la science et de la recherche à travers des cours riches, soulignant les enjeux contemporains et les solutions systématiques à développer, tout en enseignant des outils mathématiques et informatiques indispensables à une science plus libre et ouverte. Le second semestre est consacré à une immersion en laboratoire, favorisant la transition entre théorie et pratique.

1.2 Une bibliothèque de FRP

C'est dans ce contexte que s'inscrit mon travail sur OCamlFRP, une bibliothèque développée en OCaml et implantant un paradigme de programmation réactive fonctionnelle sur les flèches. Ce projet, réalisé sous la supervision de Frédéric Dabrowski, membre de l'équipe « Langages, Modèles et Vérification » (LMV) du Laboratoire d'Informatique Fondamentale d'Orléans, porte sur le développement de programmes réactifs exploitant la bibliothèque à des fins d'illustration ainsi que sur l'identification des modifications, plus ou moins majeures, à apporter à son architecture et à ses primitives.

L'équipe dans laquelle j'ai effectué mon immersion, LMV, existe depuis 2015 au sein du laboratoire et s'intéresse à l'avancement des problématiques de sécurité, de stabilité, et d'intégrité des systèmes d'information. Ses travaux portent sur le développement de nouvelles bibliothèques, de langages de programmation, ainsi que sur les méthodes formelles de vérification.

2 OCamlFRP

2.1 La programmation réactive fonctionnelle par flèches

2.1.1 Les bases de la programmation réactive fonctionnelle

La programmation réactive fonctionelle (FRP) est un paradigme de programmation déclaratif conçu pour exprimer et maintenir des systèmes d'information dits réactifs—c'est-à-dire des systèmes dont le comportement évolue au fil des intéractions avec leur environnement—. Le paradigme repose sur la nécessité d'encapsuler la notion de temps et de gestion des ressources réactives.

Initialement formulé en 1997 par Conal Elliot et Paul Hudak dans le cadre d'une recherche sur une méthode expressive de créer des médias intéractifs par ordinateur [5], le sujet s'est rapidement imposé dans la littérature académique. Il a été rapproché aux langages synchrones comme Lustre [3] et Esterel [4] dans l'objectif de concevoir de nouveaux modèles plus efficaces et rigoureux.

Aujourd'hui encore, la FRP constitue un domaine de recherche actif. Différentes implantations en font le témoignage : Elm [13], proposant une approche *message-driven* populaire en développement Web ; et Yampa [12], fondé sur une implantation rigoureuse du modèle *Arrowized* et utilisé pour modéliser des des robots et des jeux vidéo.

Le paradigme cherche ainsi une modélisation déclarative du temps et des événements en s'appuyant les notions de *behaviours* et d'*events*. Les *behaviours* sont des fonctions continues du temps :

$$Behaviour_\alpha : Time \longrightarrow \alpha$$

Et les *events* des suites discrètes de valeurs dans le temps :

$$Event_\alpha : [(Time, \alpha)]$$

Bien qu'élégante et intuitive, l'expérience à montré que le modèle imaginé originellement était trop simpliste, faisant apparaître deux problèmes contre la sécurité des systèmes réactifs :

i. des fuites mémoires (*space-leaks*), car un système réactif doit en principe conserver l'historique des valeurs ; et
ii. des fuites temporelles (*time-leaks*), dans la mesure où chaque mise à jour requiert la réévaluation complète du système.

Ces limitations, acceptables dans le contexte de Fran introduit par Conal Elliot et Paul Hudak, sont critiques pour des systèmes plus larges. Il faut donc déterminer une modélisation alternative.

2.1.2 L'abstraction de la flèche

Au cours des années 2000, John Hughes cherche à généraliser le concept de monade, populaire en programmation fonctionnelle. Il introduit alors une abstraction nommée *Arrow* [7], offrant une structure plus flexible, adaptée à des contextes variés. Il est parfois dit que « les flèches sont aux fonctions ce que les monades sont objets » [14]. Ross Paterson présente en 2001 une notation terse pour manipuler les flèches [8]. Une approche à la FRP basée sur les flèches est introduite par Henrik Nilsson, Antony Courtney, et John Peterson, [10] et qui constitue aujourd'hui le cœur Yampa.

L'intérêt particulier pour les flèches repose sur deux aspects :

- elles permettent d'éviter les fuites au niveau de l'implantation, sans complexifier le travail du programmeur ; et
- la notation proposée par Ross Paterson est élégante en particulier grâce à la composition des flèches entre elles.

La sémantique du paradigme demeure simple et libérale dans son implantation. Evan Czaplicki, createur du langage Elm, propose par exemple de représenter les flèches par des fonctions à signaux :

$$SF_{\alpha,\beta} : Signal_\alpha \longrightarrow Signal_\beta$$

Avec :

$$Signal_\alpha : Time \longrightarrow \alpha$$

Cette simplicité conceptuelle explique le succès des flèches comme paradigme robuste en FRP.

2.1.3 La construction par coitération

La coitération est une méthode mathématique permettant de représenter des données potentiellement infinies en les décomposant en étapes observables successives. Il devient alors possible de définir un flux infini de données comme ci :

$$Stream = (S \longrightarrow T \times S) \times S$$

Avec $S \longrightarrow T \times S$ une fonction de transition associant à un état un état suivant et une valeur de type T, et S un état initial.

Sur le plan catégorique, la coitération rejoint les coalgèbres, des structures duales aux algèbres classiques. Définie comme une paire (C, γ), avec $\gamma : C \to F(C)$ (F un foncteur, C une catégorie), une coalgèbre décrie l'évolution d'un état au sein d'un système dynamique. La coitération correspond alors à la construction d'un morphisme vers une coalgèbre finale, solution d'une équation coinductive, et garantissant la cohérence des comportements récursifs provenant de la solution.

De ce fait, la coitération constitue une base solide pour implanter une FRP à flèches. Quelques travaux ont déjà exploré l'idée d'exprimer ses systèmes réactifs avec des coalgèbres [6].

2.2 Implantation dans la bibliothèque

2.2.1 Documentation et explications techniques

La première tâche que j'ai entreprise a été la documentation des interfaces de la bibliothèque comme `Stream`, `Arrows`, et `FrpEngine`. Cela m'a permis de nommer les abstractions proposées, ce qui facilite le développement et l'utilisation de la bibliothèque. La documentation s'est avérée essentielle pour les tâches suivantes.

En complément, j'ai écrit un tutoriel détaillé à destination des nouveaux utilisateurs, inspiré de l'approche *Literate Programming* [2] proposée par Donald Knuth. Le résultat est plutôt positif : des collègues ont pu comprendre sans connaissance préalable les enjeux de la bibliothèque. Le tutoriel complet est disponible dans le fichier `examples/basics.ml` du dépôt.

2.2.2 Conception de programmes

La deuxième étape de mon travail a consisté au développement de programmes vitrines. Deux exemples peuvent être mis en avant : un jeu où l'ordinateur sélectionne aléatoirement un entier entre 0 et 10 que l'utilisateur

doit deviner ; et une réplique du jeu *Snake*. Les deux programmes se trouvent dans les fichiers `examples/guess.ml` et `examples/snake.ml` du dépôt.

Ce qui est particulièrement remarquable, c'est la possibilité de produire un code concis, comme observable dans la figure 2.1. Les flèches utilisées sont courtes, souvent exprimées en une seule ligne, et le reste du jeu n'est qu'une composition de celles ci.

```
let game =
  loop
    (calculate_snake_head
      >>> move_snake_head
      >>> check_apple_collision
      >>> choice
        (generate_apple >>> increment_score)
        remove_snake_tail
      >>> fanin id id
      >>> fanout make_scene id)
    Env.{ snake     = [24]
        ; direction = Right
        ; apple     = 30
        ; score     = 0
        ; }
```

FIGURE 2.1 – Boucle du jeu *Snake*.

Il convient de souligner que la conception de programmes devient intuitive et élegante : le programmeur n'est exposé qu'à la logique de son application et non à une architecture intrusive et inadaptée imposée par le langage hôte.

2.2.3 Simplification des expressions existantes

La dernière tâche de mon immersion a été de déterminer des changements possibles sur les primitives existantes. Un travail d'optimisation venant simplifier certaines définitions internes dans une logique de réduction de redondance.

À titre d'exemple, j'ai introduit explicitement la primitive `swap`, me permettant d'exprimer `second` comme deux `swap` et un `first` (figure 2.2). Dans le même esprit, j'ai refactorisé les flèches `fanin` et `fanout`.

Ces transformations ont eu un impact direct sur le module `Arrows`, dont le volume de code a été réduit d'environ moitié, sans régression. Cette simplification diminue d'autant le risque d'erreurs logiques et facilite l'évolution de la bibliothèque. Il est probable que ce travail puisse être poursuivi d'avantage, en isolant codant plus de primitives permettant d'en dériver d'autres.

```
let second f = swap >>> first f >>> swap
```

FIGURE 2.2 – La flèche `second`, triviale.

2.3 Discussions et réflexions

2.3.1 Caractères du paradigme

La FRP à flèches s'est révélé d'une efficacité surprenante. Sa composition claire et sa notation concise permettent de concevoir des programmes interactifs avec très peu de code. Le jeu *Snake* le démontre : là où une boucle de jeu classique prendrait une centaine de lignes de code, elle est ici réduite à une poignée de flèches auto-descriptives. Cette simplicité dévoile une puissance qui allège la logique applicative et recentre le développeur sur le comportement du système et non sur sa structure.

Mais certaines limites demeurent. La question de la structure de données passant entre flèches reste ouverte : faut-il privilégier une importante structure unique, plus intuitive, ou des flux fins manipulés par des primitives comme `swap` et `dup`, plus idiomatiques ? Ce choix impacte directement la lisibilité et la maintenabilité du code. Enfin, pour étendre l'usage du paradigme à des systèmes plus vastes (« programming in the large »), une réflexion s'impose sur la communication entre deux moteurs réactifs distincts.

2.3.2 Embarqué au système Caml

OCaml offre un système de typage puissant et ergonomique adapté à un style de programmation applicatif. Néanmoins, l'intégration avec le paradigme des flèches dévoile rapidement des limites. La syntaxe du langage ne se prête pas au style tacite, imposant parfois l'introduction de variables intermédiaires qui brouillent la lisibilité de la composition offerte par les flèches. Cela affecte la maintenabilité du code.

À cela s'ajoutent des tensions entre la coitération et le système d'inférence de types d'OCaml. Dans certains cas, l'inférence échoue à déduire des types construits par coitération, ce qui force à annoter explicitement des types censés être évidents. Cela nuit à practicalité attendue de la composition. Enfin, la séparation des environnements d'exécutions par des modules et foncteurs est plus lourd que bénéfique : je doute que la promesse de modularité soit exploitable dans la majorité des cas concrets, la séparation étant superflue aux besoins fonctionnelles d'une application.

2.3.3 Rapprochements et ouvertures

Par sa promotion d'un style purement compositionnel, OCamlFRP s'inscrit dans la continuité des idées de John Backus sur les langages fonctionnels [1], ou Joy [9]. Cette orientation invite à explorer ce paradigme avec un mélange de typage fort, comme les langages Cat et Kitten. Ces systèmes pourraient inspirer des évolutions dans la syntaxe et le typage d'OCamlFRP en réponse aux limitations soulevées précedemment.

Par ailleurs, pour résoudre la question encore ouverte de la communication entre moteurs et notamment dans un cadre concurrentiel, l'approche par acteurs [11] semble prometteuse. Elle permettrait d'isoler les boucles réactives tout en les connectant proprement. Mon avis final est qu'OCamlFRP incarne une altenative fiable pour concevoir des systèmes réactifs maintenables et performants.

3 Conclusion

Cette immersion dans le monde de la recherche m'a permis de dépasser les stéréotypes que l'on peut avoir depuis l'extérieur. J'ai découvert une réalité exigeante, complexe et stimulante : un processus itératif où le doute, l'essai, l'erreur et l'ajustement méthodique sont la norme.

Travailler au sein d'un laboratoire, c'est aussi faire l'apprentissage d'un écosystème de collaboration intellectuelle, où chaque idée est nourrie par les lectures, les échanges et les retours critiques. Cette dynamique collective, parfois exigeante, a été d'une grande richesse afin d'appronfondir ma compréhension des enjeux abordés. J'ai pris conscience de l'importance de maintenir un équilibre entre précision méthodologique et adaptation pratique.

Au-delà de l'expérience académique, cette immersion a été une révélation en me confortant dans mon désir de poursuivre sur la voie de la recherche et d'y contribuer. J'en retiens la nécessité de développer des compétences techniques solides, mais surtout une posture critique, patiente et ouverte. Je ressors de cette expérience avec une motivation renouvelée et des outils concrets.

4 Bibliographie

[1] J. Backus, « Can programming be liberated from the von Neumann style ? a functional style and its algebra of programs, » *Commun. ACM*, t. 21, n° 8, p. 613-641, août 1978, ISSN : 0001-0782. DOI : 10.1145/359576.359579.

[2] D. E. Knuth, « Literate programming, » *The computer journal*, t. 27, n° 2, p. 97-111, 1984.

[3] N. Halbwachs, P. Caspi, P. Raymond et D. Pilaud, « The synchronous data flow programming language LUSTRE, » *Proceedings of the IEEE*, t. 79, n° 9, p. 1305-1320, 1991. DOI : 10.1109/5.97300.

[4] G. Berry et G. Gonthier, « The Esterel synchronous programming language : design, semantics, implementation, » *Science of Computer Programming*, t. 19, n° 2, p. 87-152, 1992, ISSN : 0167-6423. DOI : https://doi.org/10.1016/0167-6423(92)90005-V.

[5] C. Elliott et P. Hudak, « Functional reactive animation, » *SIGPLAN Not.*, t. 32, n° 8, p. 263-273, août 1997, ISSN : 0362-1340. DOI : 10.1145/258949.258973.

[6] P. Caspi et M. Pouzet, « A Co-iterative Characterization of Synchronous Stream Functions, » *Electronic Notes in Theoretical Computer Science*, t. 11, p. 1-21, 1998, CMCS '98, First Workshop on Coalgebraic Methods in Computer Science, ISSN : 1571-0661. DOI : https://doi.org/10.1016/S1571-0661(04)00050-7.

[7] J. Hughes, « Generalising monads to arrows, » *Science of Computer Programming*, t. 37, n° 1, p. 67-111, 2000, ISSN : 0167-6423. DOI : https://doi.org/10.1016/S0167-6423(99)00023-4.

[8] R. Paterson, « A New Notation for Arrows, » in *International Conference on Functional Programming*, Firenze, Italy : ACM Press, sept. 2001, p. 229-240. adresse : http://www.soi.city.ac.uk/~ross/papers/notation.html.

[9] M. Von Thun et R. Thomas, « Joy : Forth's functional cousin, » in *Proceedings of the 17th EuroForth Conference*, 2001.

[10] H. Nilsson, A. Courtney et J. Peterson, « Functional reactive programming, continued, » in *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, sér. Haskell '02, Pittsburgh, Pennsylvania : Association for Computing Machinery, 2002, p. 51-64, isbn : 1581136056. doi : 10.1145/581690.581695. adresse : https://doi.org/10.1145/581690.581695.

[11] J. Armstrong, « Making reliable distributed systems in the presence of software errors, » thèse de doct., 2003.

[12] P. Hudak, A. Courtney, H. Nilsson et J. Peterson, « Arrows, Robots, and Functional Reactive Programming, » in *Advanced Functional Programming : 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002. Revised Lectures*, J. Jeuring et S. L. P. Jones, éd. Berlin, Heidelberg : Springer Berlin Heidelberg, 2003, p. 159-187, isbn : 978-3-540-44833-4. doi : 10.1007/978-3-540-44833-4_6.

[13] E. Czaplicki, « Elm : Concurrent frp for functional guis, » *Senior thesis, Harvard University*, t. 30, 2012.

[14] J. Ischard, F. Dabrowski, J. Chouquet et F. Loulergue, « A Mechanized Formalization of an FRP Language with Effects, » in *ACM Symposium on Applied Computing (SAC)*, ACM, éd., Sicily, Italy, mars 2025. adresse : https://hal.science/hal-04795983.