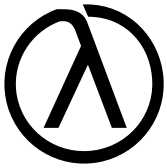

Présentation de Haskell

Les langages de programmation diffèrent les uns des autres par diverses caractéristiques adoptées par leurs syntaxes respectives et leurs méthodes d'exécution.

Au premier rang de ces caractéristiques, on retrouve évidemment les paradigmes de programmation dans lesquels ils s'inscrivent. Mais il y a d'autres caractéristiques à considérer également : système de **typage** (plus ou moins fort), le fait que les programmes écrits avec le langage en question soient plutôt destinés à être **compilés** ou à être **interprétés**, le modèle d'**évaluation** (**strict** ou **paresseux**) des données manipulées par le programme…

Le présent article s'intéresse donc au langage Haskell, pour la raison qu'il s'agit du langage de programmation préféré de l'auteur de ce site. Mais il ne s'agit pas d'un tutoriel devant permettre au lecteur de se mettre à ce langage (pour peu qu'il ait décidé de le faire). Il s'agit plutôt d'un article présentant les caractéristiques générales de Haskell, notamment à travers la grille de lecture constituée par les critères exprimés peu avant, pour ensuite exposer en quoi cette combinaison particulière de caractéristiques en fait un langage intéressant.

1. Les caractéristiques de Haskell

Paradigme de programmation

Haskell est un **langage fonctionnel pur**.

Il adopte donc le paradigme de la programmation fonctionnelle, dans une forme plutôt stricte qui interdit[1] notamment, de façon déroutante pour les adeptes de langages impératifs, les effets de bord dans les fonctions ou la ré-affectation des variables (les transformant en ce que d'autres langages appellent des constantes, des variables finales ou encore des variables immutables).

Mais si l'on précise ce qui est prohibé par l'adoption relativement stricte de ce paradigme fonctionnel, il est juste d'également préciser ce qu'elle permet en contrepartie. Comme dans tous les langages (plus ou moins) fonctionnels, les fonctions y sont de première catégorie (on peut par exemple définir des variables contenant des fonctions). Mais Haskell va bien plus loin que cela et brille particulièrement dans les domaines de la (dé)composition de fonctions complexes à partir de fonctions simples, de l'évaluation partielle ou encore de l'implémentation de fonctions par application de motifs (*pattern matching*)…

---

1. En fait « d'interdit », il s'agit plutôt de comportements par défaut des fonctions et des variables. Mais il existe parfois des syntaxes explicites de transgression de ces comportements par défaut.

Langage compilé

Les programmes réalisés en Haskell sont généralement destinés à être compilés en exécutables natifs, à l'instar de langages comme le C ou le C++ et à la différence de langages interprétés comme le Javascript ou le PHP ou de langages compilés pour des machines virtuelles comme le Java ou le Scala.

Cette approche est plus lourde en termes d'outillage, car elle impose au programmeur ou à la programmeuse de disposer d'un compilateur et de savoir s'en servir, ou de disposer également d'un système de construction (*build*) qui sait s'en servir. Elle impose également de définir la plateforme ciblée par le programme (Linux, Window, Android…) ou de procéder à des compilations distinctes pour chacune des plateformes ciblées si l'on souhaite une diffusion sur plusieurs de ces plateformes.

Mais l'étape de compilation permet en contrepartie d'éliminer de nombreuses erreurs potentielles en amont de la diffusion du programme auprès de ses utilisateur·ices. De plus, les exécutables natifs qui en résultent constituent la forme la plus performante possible que puisse revêtir un programme.

Typage statique fort

Haskell est un langage **statiquement typé** et **fortement typé**.

Cela signifie que le respect des types pour valoriser des variables ou des arguments de fonctions y est contrôlé de façon stricte par le compilateur. En outre, le langage ne s'adonne pas à des transtypages automatiques comme l'on peut en voir dans d'autres langages, par exemple quand on affecte une valeur numérique à une variable chaîne de caractères (par exemple en Javascript) ou même, pour un transtypage moins flamboyant, un entier simple dans une variable déclarée comme un entier long (par exemple en Java ou en C).

Les conversions de valeurs d'un type vers un autre peuvent évidemment exister en Haskell. Mais elles doivent alors être explicites.

Par rapport à une approche plus dynamiquement typée, qui a également ses adeptes, le typage statique opte pour plus de rigueur et moins de souplesse. Le programmeur ou la programmeuse ne peut pas faire tout ce qu'il veut aussi simplement qu'il pourrait le désirer, mais cela l'empêche également de faire n'importe quoi par mégarde ou par ignorance.

Système de types très expressif

En plus d'être statique et fort, ajoutons que le système de types de Haskell est particulièrement expressif. Les types peuvent y être dotés de multiples constructeurs, être paramétrés avec d'autres types et se conformer à des interfaces communes à certains d'entre eux.

Haskell pousse naturellement à faire grand usage des types de données algébriques, concept particulièrement puissant.

À ce titre, on notera que les références n'ont pas systématiquement de valeur indéfinie associée, telle que la valeur `null` du langage C et de ses héritiers. On peut cependant utiliser le type paramétrable `Maybe` pour définir explicitement qu'une valeur peut être non définie.

Evaluation paresseuse (par défaut)

En Haskell les différentes valeurs manipulées par un programme, notamment celles résultant de l'exécution de fonctions, sont évaluées de façon paresseuse (*lazy evaluation*). C'est à dire qu'elles sont réellement évaluées (avec le coût de calcul que cela implique) non pas au moment où elles sont définies dans le programme, mais de façon différée au moment ou la valeur y est concrètement utilisée.

Par rapport à une évaluation plus classique (dite stricte), cela implique une autre forme de compromis entre l'utilisation de la ressource calcul et celle de la ressource mémoire :

- Une évaluation paresseuse peut consommer plus de mémoire, car des expressions intermédiaires doivent être mémorisées qui permettront l'évaluation le moment venu.

  Si l'on prend l'exemple d'une fraction, une évaluation paresseuse impliquera de mémoriser le numérateur et le dénominateur plutôt que le résultat de la division (donc deux nombres au lieu d'un seul) jusqu'à ce que l'évaluation différée de la division soit effectivement exécutée.

- En contrepartie, l'évaluation paresseuse peut économiser du temps de calcul. Le fait que l'évaluation d'une valeur soit différée au moment de son utilisation implique une optimisation par construction qui est que seule les valeurs effectivement utilisées sont évaluées.

  Imaginons par exemple un programme qui charge les données d'une liste de personnes et que l'une des propriétés de chacune de ces personnes soit son âge calculé à partir de sa date de naissance et de la date du jour. Avec un langage à évaluation stricte, à moins que l'on prenne un soin particulier à rédiger du code pour l'éviter, l'âge de chacune des personnes sera calculé au chargement de la liste, qu'il soit utilisé ou non par la suite dans l'exécution du programme. Avec son évaluation paresseuse, le même programme écrit en Haskell ne calculera l'âge que pour les personnes pour lesquelles on l'utilise effectivement (par exemple quand l'utilisateur·ice consulte la fiche d'une personne, une parmi des centaines peut être) et ce sans que l'on prenne de précaution particulière dans le code.

2. En quoi cette combinaison particulière de caractéristiques rend Haskell intéressant

Chacune des caractéristiques exposées dans la section précédente peut se révéler intéressante en tant que telle, considérée séparément des autres. Mais cela peut être sujet à débat, plusieurs d'entre elles présentant des compromis entre des aspects intéressants et des contraintes qui les accompagnent.

Mais c'est surtout le « cocktail » qui résulte de l'association de ces différentes caractéristiques qui rend Haskell particulièrement intéressant.

- L'association de la pureté fonctionnelle, du typage statique fort et de l'étape de compilation qui permet une détection précoce des erreurs en s'appuyant sur ces autres caractéristiques donne au langage un très fort penchant pour la **rigueur**.

Avec la plupart des langages de programmation, même compilés, quand un programme compile le programmeur ou la programmeuse expérimenté·e sait qu'il y a encore potentiellement loin de la coupe aux lèvres pour avoir un programme qui fonctionne correctement. On a juste passé l'étape de la résolution des erreurs qui pouvaient être détectées à la compilation, mais l'on sait qu'il reste à trouver toutes celles qui ne se manifestent qu'à l'exécution… et elles peuvent être nombreuses et sournoises.

Avec Haskell, le niveau de confiance que l'on peut avoir pour un programme qui compile est considérablement plus élevé (sans être absolu). La structure même du langage vous aura interdit de coder tout un tas d'erreurs qui peuvent être monnaie courante avec d'autres langages. Ensuite, la compilation vous aura détecté la plupart des petites incohérences qui auraient pu vous échapper.

Une fois la compilation achevée, il reste très peu de place pour l'occurence d'erreurs arithmétiques (comme par exemple le fait de déréférencer une référence non définie, cause de toutes les `NullPointerException` du monde Java). Restent alors (seulement) les erreurs fonctionnelles, qui résultent généralement d'une mauvaise compréhension par le programmeur ou la programmeuse ou par l'analyste du problème à résoudre. Mais contre cela, aucun langage de programmation ne peut rien.

Cette synergie de caractéristiques qui tend à considérablement réduire à la source le nombre de bugs pouvant survenir est probablement le principal point fort de Haskell, à une ère où la principale plaie de l'industrie informatique est justement qu'elle croule sous les bugs.

- La pureté fonctionnelle entraîne la transparence référentielle, qui à son tour permet la mémoïsation des appels de fonctions. Ainsi le moteur d'exécution peut de son propre chef mémoriser le résultat du premier appel d'une fonction avec un jeu donné de valeurs d'arguments pour le réutiliser en lieu et place des prochains appels similaires. Il s'agit d'une optimisation ici automatisée alors qu'elle nécessite habituellement un travail spécifique du programmeur ou de la programmeuse utilisant des langages impératifs.

  Associons à cette mémoïsation l'évaluation paresseuse et nous obtenons des programmes avec un niveau déjà avancé d'optimisation, sans que cela ait coûté quoi que ce soit en temps de développement.

3. Domaines d'application

Haskell jouit d'une communauté non négligeable d'adeptes qui en sont généralement des pratiquants particulièrement convaincus. Mais ce n'est très clairement pas le langage de programmation le plus pratiqué au monde.

Notons en outre que si, comme tout langage *turing complete*, il peut théoriquement tout faire, il brille cependant particulièrement dans certains domaines d'application ; moins dans d'autres. Mais c'est là le lot de tous les langages de programmation.

3.1. Domaines dans lesquels Haskell brille particulièrement

Les domaines suivants sont ceux dans lesquels Haskell brille le plus :

Programmation serveur pour le web

Les bibliothèques standards de Haskell comprennent le module `Network.Wai`. WAI pour « Web Application Interface ».

Il s'agit d'un module fournissant le support de divers protocoles, au premier rang desquels HTTP(S), ainsi que des fonctions pour les exploiter à un bas niveau d'abstraction, donc avec un grand contrôle. Ce module s'accompagne de `wai-extra`, qui apporte une pléthore de fonctions complémentaires pour des sujets connexes comme l'interface CGI, la compression des flux réseau avec Gzip ou encore l'utilisation de Jsonp.

Il s'accompage également de Warp, un serveur HTTP embarquable dans l'application réalisée. Il s'agit d'un serveur HTTP **extrêmement performant**, comparable sur ce point à un Nginx, utilisant des ***green threads***.

Avec tout cela il est donc possible de réaliser en Haskell un programme serveur autonome qui, une fois l'exécutable lancé, écoute les requêtes HTTP lui parvenant sur un port spécifié et y répond. Il peut s'agir d'un serveur de pages répondant du HTML ou encore de services web répondant du JSON. D'ailleurs, la sérialisation et désérialisation de structure de données Haskell en JSON sont grandement facilitées par le module Aeson.

En outre, Haskell est un langage pris en compte par OpenAPI, ce qui permet de s'en servir pour réaliser des services web selon la norme **Swagger**. Mais le générateur OpenAPI étant ce qu'il est, on préfèrera probablement se tourner vers une approche *code first* en s'appuyant sur le module Haskell Servant.

Utilitaires en ligne de commande

Comme tout langage compilé nativement, Haskell est parfaitement adapté à la réalisation de programmes à lancer en ligne de commandes. L'exécutable produit à l'issue de la compilation peut en effet directement être lancé, sans qu'il soit nécessaire d'installer un interpréteur ou une machine virtuelle.

Haskell, dans son approche fonctionnelle pure, impose un encadrement strict des entrées et sorties, lesquelles sont un élément central de la ligne de commandes. Ceci tend à structurer les programmes d'une façon qui peut plaire ou non, mais qui a le mérite de clarifier la situation.

C'est l'occasion pour le programmeur ou la programmeuse de s'adonner au **style monadique**.

On peut citer le programme de conversion de documents Pandoc comme exemple de programme en lignes de commandes réalisé en Haskell au succès retentissant.

3.2. Domaines pour lesquels Haskell est peu adapté

Haskell se révèle peu adapté pour les domaines suivants :

Programmation système bas niveau (par exemple pilotes pour matériels)

Haskell est un langage compilé en natif qui présente typiquement, sur des applications de gestion de relativement haut niveau d'abstraction, des performances similaires à celles que l'on pourrait attendre de langages comme le C ou le C++.

En revanche, entre sa gestion automatique de la mémoire et son évaluation paresseuse (par défaut), il est moins adapté que ces deux derniers à la programmation système très bas niveau, par exemple pour écrire les pilotes d'un matériel (imprimante, carte graphique…). Dans ce domaine, il demeure en effet généralement important de pouvoir maîtriser très précisément l'allocation et la libération de la mémoire (ce qui se fait au prix d'appels de routines spécifiques qui deviennent extrêmement fastidieuses dans des applications de plus haut niveau d'abstraction).

Applications graphiques natives

Il existe des adaptateurs (*bindings*) pour Haskell pour différentes bibliothèques graphiques, notamment GTK et Qt.

Mais :

- Ces adaptateurs sont, par nécessité, écrits partiellement dans d'autres langages que Haskell.

  Cela tient au fait que les bibliothèques graphiques ciblées sont elles-mêmes écrites dans d'autres langages, par exemple C pour GTK ou C++ pour Qt. On remarquera d'ailleurs qu'il n'y a pas de *toolkit* graphique écrit directement en Haskell comme il en existe pour ces deux derniers langages.

- Même parmi les langages exogènes aux grands *toolkits* graphiques, Haskell ne figure pas parmi les plus populaires (par exemple quand on le compare à des langages comme Python, Go ou Rust). Les adapteurs Haskell pour ces bibliothèques graphiques ne figureront donc pas forcément parmi les plus complets et les mieux maintenus.

3.3. Domaines dans lesquels Haskell peut présenter un potentiel intéressant, mais qui reste à développer

Enfin, il est des domaines dans lesquels Haskell n'est peut être pas (encore) « au top », mais propose des choses présentant des potentiels intéressants, sous des formes parfois encore expérimentales :

Applications pour navigateurs web

Si, comme évoqué plus haut, Haskell est capable de briller particulièrement sur la partie serveur du web, du côté du client (le navigateur web) les choses sont un peu plus compliquées.

Les adeptes de Haskell sont rarement de grands amateurs du langage Javascript, incontournable pour doter les applications web de comportements riches exécutés directement au sein du navigateur web. Ce désamour n'est pas étonnant, tant les caractéristiques des deux langages sont différentes (paradigmes différents, compilé pour l'un et interprété pour l'autre, typage fort d'un côté et très faible de l'autre…). Cela fait d'ailleurs l'objet d'un article (en anglais) qui assimile dans son titre le langage Javascript à un problème à résoudre plutôt qu'un langage à utiliser.

L'une des réponses de la communauté Haskell à ce problème est d'explorer la voie du **transpilage**. Il s'agit d'un concept voisin de celui de la compilation. La démarche proposée est de traduire automatiquement du code Haskell non pas en code binaire natif de la machine comme le ferait un compilateur classique, mais en Javascript. L'idée est donc d'éviter d'avoir à écrire directement du Javascript en écrivant à la place du Haskell qui sera ensuite traduit (transpilé) en Javascript.

Plusieurs projets Haskell explorent donc cette voie. Un certain nombre d'entre eux y ajoute la notion de **programmation fonctionnelle réactive**, une approche qui a le potentiel de libérer le développeur du *callback hell* typique d'un emploi poussé de Javascript.

Parmi ces technologies, citons le transpileur GHCSJS et la bibliothèque Reflex.Dom.

Applications « graphiques » en mode texte

Il est un type d'applications qui reçoit peu les faveurs de la mode, c'est celui des applications proposant des interfaces « graphiques » en mode texte et devant donc être exécutées dans un terminal. Pourtant, certaines de ces applications sont des perles dans leurs genres. On peut par exemple citer l'éditeur de texte Vim, dont une version réellement graphique (gVim) existe, mais qui peut être utilisé en mode texte presque sans rien y perdre. On peut également citer des gestionnaires de fichiers comme Ranger ou Vifm ou bien encore des lecteurs musicaux comme Cmus…

Haskell peut être un langage intéressant à utiliser pour réaliser de telles applications. La bibliothèque brick est un outil qui peut être utilisé à cette fin.