Un Environnement visuel de programmation par flux de données pour soutenir l'apprentissage de la science des données

Charles Boisvert[1], Konstantinos Domdouzis[2], and Matthew Love[3]

[1] Sheffield Hallam University, City Campus, Howard Street, Sheffield S1 1WB, UK,
c.boisvert@shu.ac.uk
[2] k.domdouzis@shu.ac.uk
[3] m.love@shu.ac.uk

Résumé De nombreux outils visuels d'apprentissage de l'informatique se concentrent sur la programmation procédurale, orientée objet ou événementielle. Cet article présente un outil graphique et un langage pour rendre la programmation fonctionnelle accessible aux apprenants inexpérimentés, tout en permettant un accès ouvert aux données et aux résultats exécutables pour l'étude et le déploiement. Nous pensons que la disponibilité d'un autre paradigme et la conception ouverte des outils contribue à améliorer les moyens d'apprentissage.

Mots-clés enseignement de l'informatique, données ouvertes, flux de données, programmation

Abstract Many visual tools to learn computing focus on procedural, object-oriented and event-based programming. This paper describes a graphical tool and language which makes functional programming accessible to inexperienced learners, while also supporting open access to the data and executable results for study and deployment. We believe that both the availability of another paradigm and the open tool design contributes to improve learning tools.

Keywords Computer science education, open data, data science, workflow, functional programming

1 Introduction

Le domaine du *'big data'* qui va en s'élargissant présente de nouveaux défis aux apprenants. Cependant la communauté enseignante ne répond pas encore clairement à cette nouvelle problématique. Les nombreux cours en ligne et outils d'apprentissage restent en priorité attentifs à l'apprentissage de la programmation procédurale, événementielle ou orientée objet, avec de multiples environnements visuels basés sur des blocs comme ALICE [1], et des études comme celles de Kölling [6].

Par exemple, la collection d'exercices d'informatique recueillie par Nick Parlante ([11], [10]) contient cent-une propositions, rassemblées pour leur qualité,

mais dont seules sept incorporent une source de données réelles. On constate que la recherche d'une qualité d'outils et d'enseignement ne s'est pas encore répandue à l'analyse et à la manipulation des données.

Notre equipe de recherche *'Data Intelligence'* à Sheffield Hallam University, s'intéresse à l'adaptation de nos outils, de nos méthodes d'enseignement et de nos ressources pour faciliter l'accès et la manipulation des données par des apprenants de tous ages. Nous croyons aussi qu'une telle facilitation peut soutenir un public très large, des écoles à la société civile. Deux auteurs de cet article ont en particulier travaillé à l'accès aux données ouvertes avec un groupe de démocratie locale [8].

Pour poursuivre cette idée, nous développons à présent le projet *Open Piping*, un environnement de programmation fonctionnelle en logiciel libre doublé d'une interface de programmation par flux de données [4].

2 Motivations du projet

Open Piping est un système de *'pipes'* - de programmation fonctionnelle par la manipulation directe d'une représentation graphique du flux de données.

2.1 Des outils connus, mais restreints

La programmation visuelle par flux de données est une technique bien connue ([9], [7]), y compris certains outils en usage commercial et scientifique ([3], [4]). Cette technique est particulièrement adaptée aux architectures de services et aux applications de manipulation de données. Mais dans de nombreux cas, la valeur de ces outils est limitée par la faible transparence des processus et des technologies qu'ils implémentent - parfois même, délibérément.

Prenons le cas de Yahoo pipes [9], largement utilisé jusqu'à son abandon en 2015. N'importe quel usager pouvait définir, partager et exécuter des flux de données, mais pour choisir le système d'exécution du processus ainsi défini, l'usager devait en passer par un processus complexe et incertain : exporter le flux en JSON, puis le compiler avec un utilitaire comme *pipes2py* [2], et enfin trouver un hébergement approprié. Quand Yahoo a mis fin au projet, les utilisateurs n'ont pas eu d'autre choix, s'ils souhaitaient encore faire fonctionner leurs processus, que de les recréer ou de de réaliser ce travail d'exportation, de recompilation et de redéploiement.

Open piping tente de proposer une facilité d'utilisation comparable aux outils commerciaux, mais avec une architecture ouverte qui facilite la souplesse de développement et permette des échanges plus riches entre utilisateurs.

2.2 Réduire les barrières d'accès à la science des données

Notre ambition est de proposer un outil graphique qui prolonge la disponibilité de données ouvertes par un système de processus définis par les utilisateurs,

4. https ://github.com/boisvert/open-piping

qui inclurait dès la conception, la transparence et la souplesse nécessaire pour appliquer ces processus utilisateurs dans une variété de langages et d'environnements. *Open piping* est conçu pour être à la fois :

Ouvert. C'est à dire, un logiciel libre ; le code source du système est disponible sous licence GNU. Mais la notation utilisée pour définir les processus - simplement une expression symbolique qui définit une fonction, encodée en JSON pour la facilité - est elle-même ouverte. Le processus exécutable, défini à partir de la fonction, peut alors être offert en le transformant en l'un ou l'autre langage de programmation cible.

Interopérable. Le format JSON de spécification et d'échange d'expressions symboliques, ouvertement disponible, lisible, rend possible l'interopérabilité du système avec d'autres services, comme des interfaces utilisateurs alternatives, de nouveaux langages ou des outils d'exécution et d'hébergement de processus.

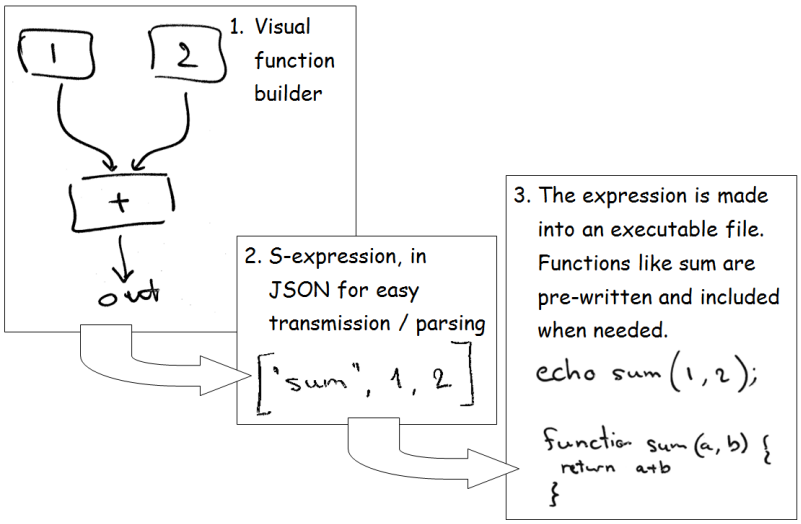
Simple d'utilisation. L'interface utilisateur rend facile la définition de flux de données et montre clairement la relation entre flux de données, l'expression symbolique résultante, et la fonctionnalité exécutable.

Facilité de déploiement des processus résultants. Le choix de langages et de conventions de services et d'intégration des données, doit faciliter la réutilisation de processus utilisateurs dans des environnements différents, par exemple dans des systèmes de gestion de contenus, en tant qu'objets intégrés au web ou à des applications, ou dans une architecture de services.

3 L'architecture *Open piping*

Le principe de fonctionnement d'*Open piping* est résumé par la fig. 1.

Figure 1. Le principe d'*Open piping*



Le flux défini par l'utilisateur est traduit en une expression symbolique encodée en JSON pour bénéficier du support de nombreux outils pour cette convention, ce qui permet de l'interpréter ensuite dans divers environnements d'exécution.

L'interprétation s'appuie sur un ensemble de fonctions de base pré-définies, dont le choix sert simultanément à définir les blocs graphiques élémentaires disponibles à l'utilisateur, fournissent l'accès aux capacités essentielles, et limitent, par sécurité, cet accès à des fonctionnalités déterminées.

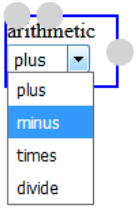
3.1 Définir et encoder un flux de données

La représentation graphique forme un graphe dont les sommets indiquent les opérations à effectuer, et les arêtes le flux des données d'une opération à l'autre.

Pour mettre à la disposition de l'utilisateur les blocs opérateurs nécessaires à la composition d'un tel graphe, l'interface est configurée grâce à une description des blocs sous forme de données JSON. Ainsi, le code indiqué en fig. 2 définit le bloc indiquant les opérations arithmétiques. Ce bloc requiert deux arguments en entrée et, par défaut, un seul en sortie.

Figure 2. Définition et représentation graphique d'un bloc de fonctions arithmétiques

```
"plus": {"args": 2, "group": "arithmetic"},
"minus": {"args": 2, "group": "arithmetic"},
"times": {"args": 2, "group": "arithmetic"},
"divide": {"args": 2, "group": "arithmetic"}
```



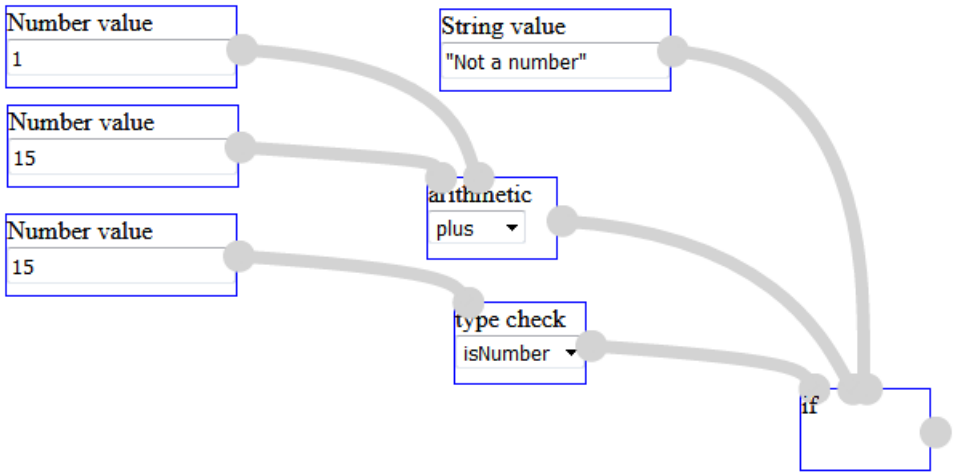
Un exemple de flux de données est représenté en fig. 3 (page suivante). L'interface a été développée grâce aux outils du projet JSPlumb [5] pour manipuler et représenter graphiquement des connections entre éléments. En parcourant récursivement le graphe, on obtient une expression symbolique. Par exemple, dans un langage tel que LISP, l'exemple fig. 3 se note par la structure :

```
(if (isNumber 15) (plus 1 15) "not a number")
```
[1]

L'encodage en JSON suit les règles suivantes :

— La notation n'utilise pas les *objets* JSON, mais seulement les *tableaux*, *chaînes*, *nombres* et les valeurs `true`, `false`, et `null`.
— Une valeur simple est une chaîne de caractères, un nombre, ou l'une des trois valeurs `true`, `false`, `null` ; elle respecte la syntaxe JSON.
— Une liste est représentée par un tableau en JSON. Chaque élément de la liste peut-être une valeur simple, ou bien une liste, et ainsi de suite récursivement.

Figure 3. Un flux de données



En respectant cette convention, le flux visible en fig. 3 s'écrit :

```
["if", ["isNumber",15], ["plus",1,15], "Not a number"]
```
[2]

L'avantage bien connu des expressions symboliques est que code exécutable et données suivent les mêmes conventions. Une expression comme la ligne [2] ci-dessus peut donc être transmise et analysée dans différents environnements sans incidence sur l'interprétation.

3.2 Interpréter une expression symbolique en langage exécutable

Pour permettre l'exécution d'une même expression dans des environnements divers, nous nous appuyons sur des caractéristiques présentes dans presque tous les langages de programmation - l'usage de variables, d'un moyen d'exécution conditionnelle, de fonctions - mais il faut fournir des données nécessaires à l'interprétation dans chaque langage. Ces données sont elles-même écrites en JSON.

Pour illustrer la méthode d'interprétation, nous allons étudier le cas de l'expression [2] précédente.

Commençant par une de ses sous-parties, pour interpréter en JavaScript et JQuery l'expression

```
["isNumber", 15]
```
[3]

la liste de fonctions prédéfinies en JavaScript inclut une définition de `isNumber` :

```
"isNumber": {"args": "n", "body": "return $.isNumeric(n);"}
```
[4]

A partir de ces données l'interprétation fournit à la fois la fonction requise et l'appel à son exécution :

```
function isNumber(n) {return $.isNumeric(n);}
process(isNumber(15));
```
[5]

Certains opérateurs simples sont interprétés en substituant simplement des chaînes de caractères pour former le code cible. C'est le cas des opérations arithmétiques, mais aussi de l'exécution conditionnelle, que nous interprétons par l'opérateur ternaire :

```
"if": {"args": "a,b,c", "js": "@a?@b:@c"},
"plus": {"args": "a,b", "js": "(@a+@b)"}
```
[6]

Les informations présentées en [4] et [6] sont toutes deux nécessaires pour interpréter l'exemple au complet :

```
["if", ["isNumber",15], ["plus",1,15], "Not a number"]
```
[7]

L'expression est interprétée récursivement. Les structures `if` et `plus`, sont remplacées par les opérateurs, et `isNumber` par la fonction correspondante. Le code JavaScript résultant est :

```
function isNumber(n) {return $.isNumeric(n);}
process(isNumber(15)?(1+15):"Not a number");
```
[8]

On voit que l'interprétation d'une fonction définie par l'utilisateur est réalisable simplement ; pour la rendre exécutable dans un langage donné, il suffit donc de pouvoir définir et exécuter de façon sûre les fonctions de base.

3.3 Etendre les possibilités du langage

Les caractéristiques définies plus haut assurent aux utilisateurs la possibilité de définir simplement les processus qu'ils souhaitent voir opérer sur les données, tout en conservant le contrôle pour utiliser ces processus dans des environnements de leur choix.

En particulier :

— Les limites d'exécution ne sont pas inhérentes au langage. Si nécessaire, l'environnement dans lequel les fonctions sont déployées peut, par exemple, limiter la durée de l'exécution.

— Par sécurité, le même environnement peut assurer l'interprétation de l'expression symbolique et son exécution. Ceci limite les risques d'injection de code et définit les fonctions élémentaires autorisées dans un contexte donné.

— La définition d'un processus est découplée de l'environnement d'exécution, en définissant visuellement une fonction dans un langage spécifié ouvertement ; ainsi les évolutions de l'interface visuelle, des langages cibles, et des environnements d'exécution restent indépendants.

Cependant, plusieurs autres structures sont nécessaires pour éviter des limites inhérentes au langage lui-même et offrir à l'utilisateur toutes les capacités d'un langage de programmation.

Fonctions, variables et assignation. Fonctions et variables sont définies en interprétant `defun` et `setq`. Toutes deux s'interprètent de la même façon que les structures LISP équivalentes.

Quelques fonctions élémentaires. Outre les primitives mathématiques et logiques, le langage requiert plusieurs fonctions de base. par exemple, pour manipuler des listes, `first` et `rest` séparent le premier élément des autres, `cons` ajoute un élément à une liste, `isEmpty` teste si une liste est vide.

Fonctions d'ordre supérieur. Pour permettre l'extension du langage il est indispensable de pouvoir manipuler les fonctions comme données. Cependant, les solutions adoptées récemment dans de nombreux langages avec les fonctions de première classe, complexifient grandement l'interprétation vers des langages multiples : pour rendre applicable n'importe quelle fonction du langage, il faudrait toutes les inclure dans un résultat exécutable, et les sécuriser. La solution adoptée par *Common LISP*, d'une fonction `apply` qui indique a l'interpréteur comment traiter le paramètre, est plus adaptée. Un exemple classique est la définition naïve de `map` sous cette forme :

```
["defun", "map",
         ["f", "arr"],
         ["if", ["isEmpty", "arr"],
                [],
                ["cons", ["apply", "f", ["first", "arr"] ],
                         ["map", "f", ["rest", "arr"] ]
                ]
         ]
]
```

Partant de l'interface utilisateur, l'usage d'une fonction comme donnée se définit en indiquant qu'un bloc - et non le résultat de l'évaluation d'un bloc - est donné en paramètre à un autre. Nous proposons donc de marquer que le paramètre est différent par le symbole `antislash` :

```
[ "map", "\isNumber", [1,2,3,"a","b","c"] ]
```

L'usage d'un symbole modifiant l'interprétation du paramètre correspond bien à la définition du processus par une manipulation graphique, et facilite l'interprétation tout en permettant la souplesse des fonctions d'ordre supérieur.

4 Conclusions et travaux à venir

A présent les outils permettent à un utilisateur de définir un flux de données, puis de produire l'expression symbolique correspondante, qui est traduite en JavaScript ou en PHP. Le code JavaScript permet de tester le résultat. Un ensemble d'expressions symboliques permet de tester le langage et le processus de compilation.

Le projet ne démontre la compilation des expressions symboliques constructivement qu'en deux langages, mais de multiples environnements sont envisageables. Les processus définis pourraient être exécutés, mais aussi déployés dans de nouveaux systèmes ; l'intégration avec des outils de gestion de contenus serait intéressante, ainsi que, pour le traitement de données plus massives, l'adaptation à des environnements parallèles.

Le langage montre comment des utilisateurs peuvent être soutenus par des outils visuels, mais souples, qui permettent l'accès au développement dans un langage fonctionnel. Nous pensons qu'il y a un besoin d'instruments qui offrent au public une plus grande diversité de paradigmes de programmation, et permettent aux apprenants d'essayer, de comparer, et de bénéficier des avantages de ces différents paradigmes pour des applications différentes. De plus, nous souhaitons soutenir l'accès ouvert non seulement aux données, mais au code source, aux conventions en usage, aux résultats exécutables, et aux environnements d'exécution, permettant l'apprentissage autonome grâce à l'accessibilité.

Références

1. S. Cooper, W. Dann, and R. Pausch. Alice : a 3-d tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, volume 15, pages 107–116. Consortium for Computing Sciences in Colleges, 2000.
2. G. Gaughan. A project to compile yahoo! pipes into python. https ://github.com/ggaughan/pipe2py. Accessed : 2016-10-10.
3. D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn. Taverna : a tool for building and running workflows of services. *Nucleic acids research*, 34(suppl 2) :W729–W732, 2006.
4. N. Instruments. Logiciel de conception de systemes labview. http ://www.ni.com/labview. Accessed : 2017-10-04.
5. I. JSPlumb. Jsplumb toolkit documentation. https ://jsplumbtoolkit.com/docs.html. Accessed : 2017-13-04.
6. M. Kölling. The problem of teaching object-oriented programming. *Journal of Object Oriented Programming*, 11(8) :8–15, 1999.
7. D. Le-Phuoc, A. Polleres, G. Tummarello, and C. Morbidoni. Deri pipes : visual tool for wiring web data sources. *)ˆ(Eds.) :'Book DERI pipes : visual tool for wiring web data sources'(2008, edn.)*, 2008.
8. M. Love, C. Boisvert, E. Uruchurtu, and I. Ibbotson. Nifty with data : Can a business intelligence analysis sourced from open data form a nifty assignment ? In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, pages 344–349, New York, NY, USA, 2016. ACM.
9. T. O'Reilly. Pipes and filters for the internet. http ://radar.oreilly.com/2007/02/pipes-and-filters-for-the-inte.html. Accessed : 2016-10-10.
10. N. Parlante. Nifty assignments. http ://nifty.stanford.edu. Accessed : 2016-01-12.
11. N. Parlante, J. Popyack, S. Reges, S. Weiss, S. Dexter, C. Gurwitz, J. Zachary, and G. Braught. Nifty assignments. In *ACM SIGCSE Bulletin*, volume 35, pages 353–354. ACM, 2003.