QuickFill, QuickMixte : approches par blocs pour la réduction du nombre de programmes en synthèse de programmes

Vanessa Fokou, Peggy Cellier, Maurice Tchuente, Alexandre Termier

► To cite this version:

Vanessa Fokou, Peggy Cellier, Maurice Tchuente, Alexandre Termier. QuickFill, QuickMixte : approches par blocs pour la réduction du nombre de programmes en synthèse de programmes. CARI 2022 - Colloque Africain sur la Recherche en Informatique et en Mathématiques Appliquées, Oct 2022, Yaoundé, Cameroun. pp.1-10. hal-03717860

HAL Id: hal-03717860
https://inria.hal.science/hal-03717860

Submitted on 10 Jul 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

QuickFill, QuickMixte : approches par blocs pour la réduction du nombre de programmes en synthèse de programmes

Vanessa Fokou[1], **Peggy Cellier**[2], **Maurice Tchuente**[1,*], **Alexandre Termier**[2]

[1]Université de Yaoundé I, Département d'Informatique, BP 812 Yaoundé Cameroun
[1]vanfokou@gmail.com, maurice.tchuente@gmail.com
[2]Univ Rennes, Inria, INSA Rennes, CNRS, IRISA - UMR 6074, France
[2]prenom.nom@irisa.fr
[*]IRD, Sorbonne Université, UMMISCO, F-93143, Bondy, France

Résumé

Les tâches répétitives sont le plus souvent fastidieuses ; afin de faciliter leur exécution, les approches de synthèse de programmes ont été développées. Elles consistent à inférer automatiquement des programmes qui satisfont l'intention d'un utilisateur. L'approche la plus connue en synthèse de programmes est *FlashFill* intégrée au tableur Excel qui permet le traitement des chaînes de caractères. Dans *FlashFill* l'intention de l'utilisateur est représentée par des exemples i.e, des couples $(entrée, sortie)$. *FlashFill* explore un très grand espace de programmes et peut donc nécessiter un temps d'exécution important et inférer beaucoup de programmes dont certains fonctionnent sur des exemples donnés mais ne capturent pas l'intention de l'utilisateur. Dans cet article, nous proposons deux approches *QuickMixte* et *QuickFill* basées sur les blocs qui visent à guider l'exploration de l'espace de programmes de *FlashFill* en enrichissant les spécifications fournies par l'utilisateur. Ces approches demandent à l'utilisateur de fournir des associations entre les sous-parties de la sortie et de l'entrée pour affiner les spécifications. Les expérimentations menées sur une série de 12 jeux de données montrent que *QuickMixte* et *QuickFill* permettent de réduire considérablement l'espace de programmes de *FlashFill*. Nous montrons qu'avec ces approches, il est souvent possible de donner moins d'exemples qu'avec l'algorithme *FlashFill* original pour une plus grande proportion de programmes corrects.

Mots-Clés

Synthèse de programmes, programmation par l'exemple, manipulation des chaînes de caractères, tâches répétitives, approche par blocs.

I INTRODUCTION

La synthèse de programmes (*en anglais, Program Synthesis*) consiste à trouver automatiquement, à partir d'un langage de programmation sous-jacent, des programmes qui satisfont l'intention de l'utilisateur, souvent exprimée par des exemples (entrée, sortie). Elle a été utilisée dans plusieurs applications comme la réparation de code [6], la modélisation probabiliste [7], la suggestion de code [5] ou encore le traitement des données [8]. Le pré-traitement des données est l'une des tâches les plus chronophages d'un processus d'analyse de données. On estime que les ingénieurs/scientifiques des données consacrent 80% de leur temps au pré-traitement des

données [9]. Cette tâche demande souvent de faire des transformations non triviales de formats, qui sont trop complexes pour être effectuées par les outils existants, mais trop simples pour mériter l'attention soutenue d'un analyste. En pratique, la solution classique pour effectuer les tâches de pré-traitement de données est l'écriture de petits scripts dans des langages de programmation généralistes (par exemple Python). La difficulté est que de nombreux utilisateurs ne maîtrisent pas les bases de la programmation, et n'ont donc pas la possibilité d'écrire de tels scripts.

Pour pallier ce problème, les techniques de synthèse de programmes, en particulier la programmation par l'exemple, sont intéressantes. C'est un sous-domaine de la synthèse de programmes, où la spécification se présente sous la forme d'exemples entrée-sortie. L'approche la plus connue en synthèse de programmes est *FlashFill* [4], elle est intégrée à toutes les versions récentes du tableur Excel pour le traitement automatique de chaînes de caractères. *FlashFill* considère des lignes dans le tableur, et apprend des programmes prenant en entrée le contenu d'une ou plusieurs colonnes de la ligne par exemple une colonne des noms et une colonne des prénoms, et dont le résultat est stocké dans une autre colonne par exemple une colonne contenant la concaténation des noms et prénoms. L'algorithme original *FlashFill*, tel que présenté dans GULWANI [4], doit explorer un grand espace de programmes, et peut donc nécessiter un temps d'exécution important. De plus, beaucoup de programmes peuvent être inférés dont certains fonctionnent sur les exemples donnés mais ne capturent pas l'intention de l'utilisateur, et donneront des résultats faux sur d'autres entrées. Ce problème de généralisation est dû au fait que les programmes sont souvent spécifiques aux exemples donnés.

Dans cet article nous présentons *QuickMixte* et *Quickfill* deux améliorations de *FlashFill*. Ces deux approches réduisent l'espace des programmes à explorer par *FlashFill* en enrichissant les spécifications fournies par l'utilisateur. Pour chaque exemple qui consiste en couple (entrée, sortie), l'utilisateur est invité à donner le *matching* entre les sous-parties de l'entrée et les sous-parties de la sortie. Le but de ces deux approches est de montrer que l'ajout de certaines interactions avec l'utilisateur peut aider à réduire le nombre de programmes de *FlashFill*. Les expériences menées ont permis de vérifier cette hypothèse.

Le reste du papier est organisé comme suit : la section 2 présente quelques travaux antérieurs sur le traitement de données via la synthèse de programmes, la section 3 présente les approches *QuickMixte* et *QuickFill*, la section 4 présente les expérimentations et la section 5 conclut le papier.

II ÉTAT DE L'ART SUR LA PROGRAMMATION PAR L'EXEMPLE

La synthèse de programmes comprends beaucoup de sous-domaines parmi lesquels la programmation par l'exemple et la programmation par démonstration, elles peuvent automatiser de nombreuses tâches courantes - à l'exemple de l'édition de texte - qui tendent à consommer une fraction frustrante du temps des utilisateurs. En programmation par l'exemple, l'idée est d'apprendre des programmes qui correspondent aux actions de l'utilisateur à partir des exemples qu'il fournit par interaction avec un agent logiciel. L'agent peut ensuite généraliser ces programmes afin qu'ils puissent fonctionner dans d'autres situations similaires, mais pas nécessairement identiques, aux exemples sur lesquels ils sont appris [1]. En 2003, LAU, WOLFMAN, DOMINGOS et WELD [2] formalisent l'algèbre des espaces de version, une méthode permettant de composer des espaces de version plus simples pour construire des espaces plus complexes et l'appliquent à la programmation par démonstration dans le domaine de l'édition de texte en l'occurrence *SMARTedit*, un système qui apprend les procédures répétitives d'édition de texte par

l'exemple. En 2009, LAU [3] explique pourquoi les systèmes de programmation par démonstration échouent ; elle y mentionne que la facilité d'utilisation reste un obstacle majeur pour ces systèmes ; *SMARTedit* étant un cas d'étude. La démonstration fournit par l'utilisateur consiste en une séquence de l'état de l'éditeur après chaque action primitive expliquant comment effectuer la transformation. Par ailleurs, le langage de *SMARTedit* n'est pas aussi expressif que ce qui est nécessaire dans le cadre des feuilles de calcul. Les outils de type tableurs comme Microsoft Excel sont particulièrement adaptés à la manipulation et la transformation des données numériques. De ce fait, on ne dispose que de fonctions élémentaires pour la manipulation des chaînes de caractères. En 2011, GULWANI [4] propose l'approche *FlashFill* pour le traitement automatique de chaînes de caractères dans les tableurs.

FlashFill est un synthétiseur de programmes pour les transformations syntaxiques de chaînes de caractères à partir d'un ensemble de couples d'exemples *(entrée, sortie)* fournis par l'utilisateur. Il prend en entrée un ensemble d'exemples de type (σ, s) où $\sigma = (\sigma_1, ..., \sigma_n)$ est un tuple dont chaque élément est une chaîne de caractères, et où s est une chaîne de caractères. Dans un tableur comme Excel, chaque ligne est un exemple, σ est un ensemble de cellules de la ligne contenant des données, et s est le résultat d'une transformation de chaîne de caractères sur les cellules σ. Par exemple $((A1 : "Jean", B1 : "Dupont"), C1 : "Jean\ Dupont")$ permet d'inférer qu'il faut concaténer le contenu des colonnes A et B et ranger le résultat dans la colonne C. Pour un exemple (σ, s) donné, *FlashFill* traite chaque sous-partie de s à partir de l'entrée σ pour trouver les sous-programmes associés, et combine par la suite ces sous-programmes pour obtenir l'ensemble des programmes permettant d'obtenir s à partir de σ. Ce procédé conduit à un grand nombre de programmes car le chevauchement qu'il y a entre les sous-parties de s implique que certaines sous-parties sont traitées à plusieurs reprises, ce qui augmente le nombre de programmes de *FlashFill*. Dans la suite, nous présentons deux approches par blocs qui permettent de réduire le nombre de programmes de *FlashFill*.

III APPROCHES PAR BLOCS : *QUICKMIXTE, QUICKFILL*

Dans cette section, nous rappelons tout d'abord les principales notions de *FlashFill* avant d'introduire les deux approches *QuickMixte* et *QuickFill*.

3.1 Explications sur *FlashFill*

Pour résoudre le problème de transformation syntaxique de chaînes de caractères dans les tableurs, *FlashFill* s'appuie sur deux principaux éléments : un langage qui contient des primitives simples pour de la manipulation de chaînes de caractères et un algorithme de synthèse qui utilise ce langage spécifique pour construire des programmes de transformation. Ce langage contient des expressions telles que : *ConstStr* pour les constantes, *SubStr* pour l'extraction de sous-chaînes, *Concatenate* pour la concaténation, etc. Les éléments dudit langage sont expliqués en Annexe. Étant donné S un ensemble de couples d'exemples entrée-sortie (σ, s) avec $\sigma = (\sigma_1, ..., \sigma_n)$, le but de *FlashFill* est d'apprendre des programmes de transformation de chaînes de caractères cohérents avec tous les couples (σ, s) $\in S$ i.e, capables de transformer chaque entrée σ en sa sortie s. Sa première étape consiste à calculer pour chaque couple (σ, s) $\in S$ indépendamment les *programmes* permettant de transformer σ en la sortie s. La deuxième étape consiste à construire l'ensemble des programmes cohérents avec tout couple (σ, s) $\in S$ par intersection des programmes individuels si tous les exemples sont de même forme ou par partitionnement des exemples puis intersection des programmes par partition sinon. Pour ce qui est de la troisième étape, elle intervient uniquement lorsque le partitionnement a lieu ; nous ne

nous y intéressons pas dans ce papier. Comme mentionné précédemment, la grande taille de l'espace de programmes qu'explore *FlashFill* est due au fait qu'il traite chaque sous-partie de s ; pour chacune d'elle, il se sert de toute l'entrée σ pour construire les sous-programmes correspondant, le nombre de tels sous-programmes est influencé par la longueur de σ. Notre but est de réduire l'espace de programmes à explorer par *FlashFill* via les approches par blocs *QuickMixte* et *QuickFill* qui diminuent le nombre de sous-parties à traiter et empêchent d'utiliser toute l'entrée en demandant à l'utilisateur des associations entre les sous-parties de l'entrée et celles de la sortie. Il est à noter que *QuickMixte* et *QuickFill* partent du même problème initial que *FlashFill* i.e, à partir d'un ensemble d'exemples, il faut apprendre un ensemble des programmes pouvant transformer l'entrée σ en la sortie s.

3.2 QuickMixte

QuickMixte demande à l'utilisateur, pour chaque couple d'exemple entrée-sortie $(\sigma = (\sigma_1, ..., \sigma_n), s)$, d'identifier dans s l'ensemble de sous-chaînes $\{s_1, ..., s_k\}$ impactées par l'entrée σ, et pour chaque sous-chaîne s_i, d'identifier l'élément de l'entrée lui étant associée : une sous-chaîne σ_j^{sub} $\in \sigma_j$. Ce *matching* entre les sous-parties s_i de s et les chaînes σ_j^{sub} de σ associées représente *les couples de blocs*. Les sous-chaînes de s n'étant pas impactées par l'entrée σ sont des *constantes*. Un couple de blocs (σ_j^{sub}, s_i) est tel que s_i est une sous-partie de s, σ_j^{sub} une sous-chaîne de σ pouvant permettre de construire s_i. Les couples de blocs lui étant fournis, *QuickMixte* construit les sous-programmes permettant de construire chaque sous-chaîne s_i par appel de *FlashFill* et à la fin procède à une combinaison de sous-programmes pour obtenir des programmes permettant de construire s à partir de σ. Les sous-programmes obtenus sur chaque couple de blocs, sont de la forme d'un programme retourné par *FlashFill* i.e, une formule de concaténation (*Concatenate*). Le but étant d'obtenir des programmes cohérents avec l'exemple $(\sigma = (\sigma_1, ..., \sigma_n), s)$, on va combiner les sous-programmes dans l'ordre des s_i pour obtenir les programmes finaux. Ces couples de blocs fournis permettent de guider plus finement l'espace de recherche dans la mesure où le traitement d'une partie de la sortie ne se fait pas en exploitant toute l'entrée mais une partie précise de cette dernière. Par exemple, soit un couple d'exemple *(σ = "Data wrangling", s = "Dg")*; on a deux couples de blocs à savoir *("Data", "D")* et *("wrangling", "g")*. Avec le premier couple de blocs, on trouve les sous-programmes permettant d'obtenir *"D"* dans *"Data"* (extraction du premier caractère) et le deuxième couple de blocs permet de trouver les sous-programmes permettant d'obtenir *"g"* dans *"wrangling"* (extraction du dernier caractère) ; les concaténations de ces sous-programmes donnent les programmes permettant d'obtenir *"Dg"* à partir de *"Data wrangling"*. En d'autres termes, *QuickMixte* est une combinaison des résultats d'exécutions de *FlashFill* sur les couples de blocs. Pour plusieurs couples d'exemples donnés, l'ensemble de programmes est l'intersection des programmes obtenus pour chaque exemple.

3.3 QuickFill

L'approche *QuickMixte* précédente permet de réduire l'espace de programmes car elle laisse certaines sous-chaînes de s inexplorées et pour chaque sous-chaîne de s traitée, elle exploite une sous-partie σ_j de σ. Cependant, l'algorithme de *FlashFill* est entièrement conservé sur le traitement de chaque couple de blocs car, chaque couple de blocs (σ_j, s_i) est traitée par appel de *FlashFill*. Cette pratique peut empêcher une réduction stricte de l'espace de programmes, car comme l'algorithme original, par conséquent,*QuickMixte* hérite de certains défauts de *FlashFill*. Pour chaque couple (σ_j, s_i), *QuickFill* se sert du langage de *FlashFill* pour construire les programmes correspondant. Au lieu de traiter individuellement chaque sous-partie de s_i pour trouver les sous-programmes, *QuickFill* cherche directement la chaîne s_i pour construire les

sous-programmes . Il faut noter que dans *QuickFill*, la constructions de blocs se fait de la même façon que dans *QuickMixte* à la seule différence que pour un couple de blocs (σ_j^{sub}, s_i), s_i doit être une sous-chaîne consécutive de la sous-partie de σ_j^{sub} associée ; pourtant avec *QuickMixte*, s_i n'est pas nécessairement une sous-chaîne consécutive de σ_j^{sub}. Par exemple, avec *QuickMixte* on peut avoir pour couple de blocs *("Docteur", "Dr")* mais avec *QuickFill* ce n'est pas possible car *"Dr"* n'est pas une sous-chaîne de *"Docteur"* ; par contre, *("Docteur", "D")* est valide comme couple de blocs que ce soit pour *QuickMixte* ou pour *QuickFill*. Il est important de noter que dans de nombreux cas, *QuickMixte* et *QuickFill* demandent le même nombre de couples de blocs et que *QuickFill* obtenir de meilleurs résultats comme le montrent les expérimentations.

Une chaîne s de longueur n a au total $\frac{n(n+1)}{2}$ sous-chaînes possibles. Étant donné (σ, s) un couple de blocs, *FlashFill* construit les sous-programmes associés à chaque sous-chaîne de s à partir de toute l'entrée σ. *QuikFill* évite d'explorer toutes les sous-chaînes de s et n'utilise que les parties de σ associées aux sous-chaînes de s à explorer. Contrairement à *quickMixte*, Les sous-programmes obtenus avec *QuickFill* sont élémentaires i.e, sans formules de concaténation. les programmes finaux s'obtiennent par construction des formules *Concatenate* dans l'ordre des s_i comme dans *QuickMixte*. La section suivante présente les expérimentations.

IV EXPÉRIMENTATIONS

Il est important de noter que dans cet article, nous nous basons sur l'algorithme *FlashFill* tel que défini dans [4], et pas sur la version qui peut être testée dans Excel 365. Par ailleurs, notre but étant de tester l'intérêt d'ajouter certaines interactions avec l'utilisateur et pas de reproduire *FlashFill* dans toute sa généralité, nous ne considérons pas le constructeur $Loop$ du langage de manipulation de chaînes de caractères. Nous ne considérons pas non plus le constructeur $Switch$, qui permet de gérer des conditions dans les programmes lorsque les exemples sont de différentes formes i.e lorsque les exemples se partitionnent en plusieurs "types", chaque type étant résolu par une forme de programme de transformation de chaînes différent [1].

Afin de montrer l'intérêt de *QuickFill* et *QuickMixte*, nous avons mené une série d'expérimentations dans laquelle nous évaluons le nombre de programmes inférés par *QuickFill* et *QuickMixte* par rapport à *FlashFill*, le taux de programmes corrects - un programme est dit correct lorsqu'il capture bien l'intention de l'utilisateur i.e, produit de bonnes sorties sur toutes les entrées - dans *QuickFill et QuickMixte* par rapport à *FlashFill*. Nous évaluons aussi le temps de construction des programmes et l'espace mémoire qu'ils occupent.

Les expérimentations ont été menées sur une machine Core i3 2.1 GHz dotée d'une mémoire RAM de 8Go. Les méthodes *QuickMixte, QuickFill* et *FlashFill Light* ont été implémentées en Python. Le code et les données sont disponibles sur github [2].

Jeux de données. Les jeux de données utilisés [3] ont été obtenus en adaptant les cas traités dans l'article de *FlashFill* [4]. La génération de ces jeux de données s'est faite via un générateur aléatoire de mots à partir d'expressions régulières [4]. Chaque jeu de données contient 30 éléments dont les premiers couples $(entrée, sortie)$ représentent les exemples d'entraînement (i.e., à partir desquels les programmes sont appris) et le reste représente les exemples de tests (i.e., les

1. Nous travaillons dans la suite avec une version simplifiée de *FlashFill* que nous avons appelé *FlashFill Light*.
2. https://github.com/vanes11/FlashFill_QuickFill_Mixte
3. https://github.com/vanes11/FlashFill_QuickFill_Mixte/tree/master/Benchmarks
4. https://onlinerandomtools.com/generate-random-data-from-regexp

entrées sur lesquelles les programmes générés sont testés). Les jeux de données couvrent plusieurs types de tâches comme l'extraction des prénoms à partir des adresses mails, l'extraction des jours dans les dates ou encore l'extraction des initiales d'un sigle.

Évaluation du nombre de programmes. On cherche à comparer le nombre total de programmes et le nombre de programmes corrects produits par *FlashFill* par rapport à *QuickFill* et *QuickMixte*. Les programmes sont obtenus sur chaque jeu de données pour un nombre d'exemples d'apprentissage allant de 1 à 4, le tableau 1 présente les résultats pour 1 exemple d'apprentissage sur nos 12 jeux de données. La colonne $Total$ représente la totalité des programmes générés et la colonne $Correct$ représente le nombre de programmes corrects dans la totalité des programmes. On observe que *QuickMixte* permet de réduire le nombre total de programmes générés de plus de 75% pour 10 jeux de données. Toutefois, de part la réutilisation directe de l'algorithme *FlashFill* sur chaque couple de blocs par *QuickMixte*, il reste 2 jeux de données où aucune réduction n'est observée. Notre deuxième contribution *QuickFill*, qui élague plus l'espace de recherche, permet d'obtenir une réduction supérieure à 81% sur l'intégralité des des jeux de données.

	FlashFill		**QuickMixte**		**QuickFill**	
Benchmarks	**Total**	**Correct**	**Total**	**Correct**	**Total**	**Correct**
B1	6482	2	6482 (0.0%)	57	6 (99.91%)	3
B2	27398	18302	3158 (88.47%)	2102	22 (99.92%)	22
B3	196	9	49 (75.0 %)	9	36 (81.63%)	9
B4	13744	6	672 (95.11%)	51	60 (99.56%)	15
B5	2089550	2	4244 (99.8%)	2	18 (99.99%)	2
B6	96914	96914	1521 (99.43 %)	1521	1444 (98.51%)	1444
B7	2106048	756899	11520 (98.45 %)	9920	14 (99.99%)	14
B8	473170	471396	4489 (99.05%)	4356	4356 (99.08%)	4356
B9	2897578	982278	343 (99.99%)	54	216 (99.99%)	54
B10	783760	779076	20497 (97.38%)	20196	20196 (97.42%)	20196
B11	32	22	32 (0.0%)	22	6 (81.25%)	6
B12	14486	228	25 (99.83%)	21	16 (99.89%)	1

TABLE 1 – Nombre de programmes générés par *FlashFill*, *QuickMixte* et *QuickFill* ; le pourcentage entre parenthèse représentent le pourcentage de réduction par rapport à *FlashFill*.

Évaluation du taux de programmes corrects.

Le tableau 1 donne une idée de la réduction en nombre de programmes générés qu'effectuent *QuickMixte* et *QuickFill*. Nous évaluons ensuite la variation des taux de programmes corrects dans *QuickFill, QuickMixte* et *FlashFill* en fonction du nombre d'exemples d'apprentissage allant de 1 à 4 pour chaque jeu de données. La figure 1 se limite à deux jeux de données ; des résultats similaires sont observés sur les autres jeux de données. Cette expérience montre que pour un nombre d'exemples donné, *QuickFill* et *QuickMixte* ont, dans la majorité des cas, de meilleurs taux de programmes corrects par rapport à *FlashFill*. En pratique, cela signifie qu'il faut souvent moins d'exemples pour découvrir les programmes capturant l'intention de l'utilisateur. Par ailleurs, on remarque aussi que *QuickMixte* n'est pas aussi efficace que *QuickFill* en terme en réduction du nombre de programmes et de meilleur taux de programmes corrects.

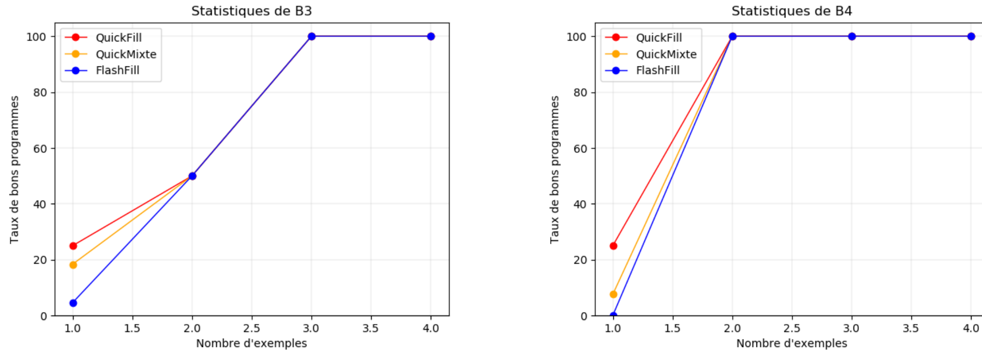

FIGURE 1 – Taux de programmes corrects dans les *benchmarks* $\{B3, B4\}$.

Évaluation du temps de construction des programmes et de l'occupation mémoire. Nous observons ici le comportement de *QuickFill, QuickMixte* et *FlashFill* en terme de temps de construction des programmes et d'occupation mémoire en fonction du nombre d'exemples d'apprentissage (de 1 à 4). Ces deux métriques nous permettent de vérifier l'hypothèse selon laquelle, réduire le nombre de programmes permettrait de gagner en temps d'exécution et en occupation mémoire.

La figure 2 présente la variation du temps de construction des programmes dans *QuickFill, QuickMixte* et *FlashFill* sur les jeux de données $\{B5, B6\}$. La légende est la suivante : *t-QuickFill* = temps de *QuickFill*, *t-Mixte* = temps de *QuickMixte* et *t-FlashFill* = temps de *FlashFill*. Nous observons sur cette figure que le temps de construction des programmes dans *FlashFill* est toujours supérieur à celui de *QuickFill et QuickMixte*.

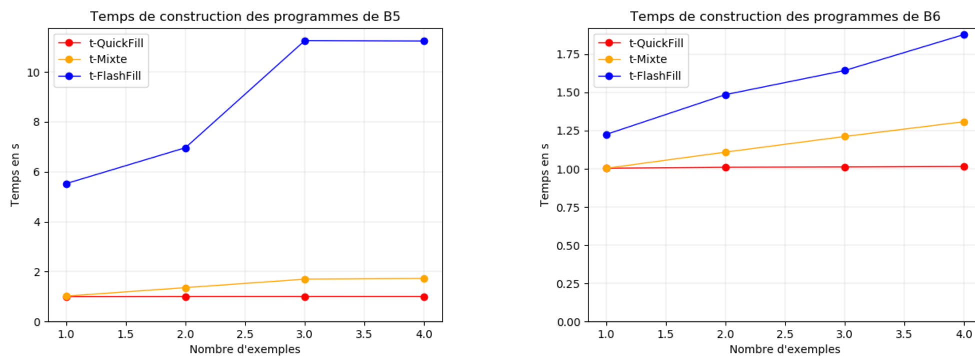

FIGURE 2 – Temps de construction des programmes dans les *benchmarks* $\{B5, B6\}$.

La figure 3 présente la variation de l'occupation en mémoire des programmes dans *QuickFill, QuickMixte* et *FlashFill* sur les jeux de données $\{B4, B5\}$. La légende est la suivante : *t-QuickFill* = mémoire de *QuickFill*, *t-Mixte* = mémoire de *QuickMixte* et *t-FlashFill* = mémoire de *FlashFill*. Comme pour le temps de construction de programmes, les programmes inférés par *FlashFill* requièrent plus d'espace mémoire que ceux de *QuickFill, QuickMixte* car ils sont plus nombreux.

V CONCLUSION

La synthèse de programmes est une solution de plus en plus utilisée pour décharger l'utilisateur de tâches fastidieuses et répétitives comme certaines transformations de données. La difficulté

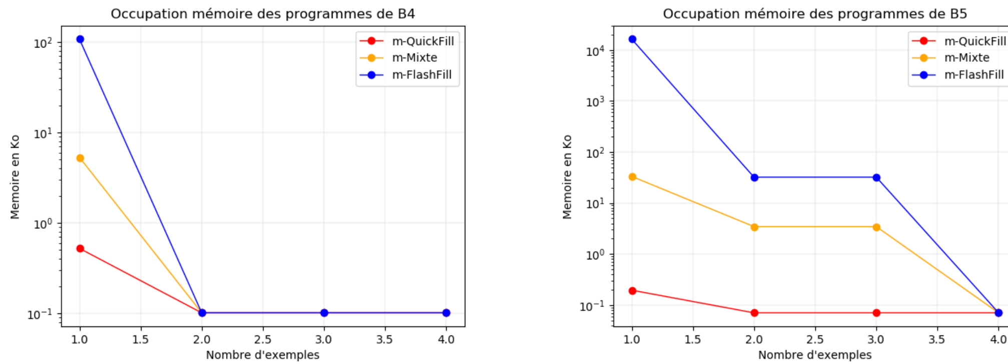

FIGURE 3 – Occupation mémoire des programmes dans les *benchmarks* $\{B4, B5\}$.

de ce domaine est de naviguer dans un espace de recherche immense, tout en limitant au maximum les interactions avec l'utilisateur (en général il est juste autorisé de lui demander quelques exemples). Plus l'espace de programmes est grand, plus il peut être coûteux de trouver le(s) programme(s) cohérent(s) avec l'intention de l'utilisateur. Dans cet article, nous avons présenté *QuickMixte et QuickFill*, deux approches qui demandent à l'utilisateur des interactions plus soutenues sur les exemples qu'il fournit, mais en contrepartie réduisent dans la majorité des cas le nombre total de programmes, peuvent arriver plus vite à des programmes correspondant à son intention : à la fois en moins de temps de calcul, et via moins d'exemples.

Une perspective d'amélioration du temps demandé à l'utilisateur par *QuickMixte et QuickFill* serait de fournir les blocs uniquement dans les sous-chaînes de sorties ; ce qui empêche l'utilisateur de fournir les correspondances dans les chaînes d'entrées et réduit ainsi le nombre de blocs à fournir.

RÉFÉRENCES

[1] H. LIEBERMAN. *Your wish is my command : Programming by example*. Morgan Kaufmann, 2001.

[2] T. LAU, S. A. WOLFMAN, P. DOMINGOS et D. S. WELD. « Programming by demonstration using version space algebra ». In : *Machine Learning* 53.1 (2003), pages 111-156.

[3] T. LAU. « Why programming-by-demonstration systems fail : Lessons learned for usable ai ». In : *AI Magazine* 30.4 (2009), pages 65-65.

[4] S. GULWANI. « Automating string processing in spreadsheets using input-output examples ». In : *ACM Sigplan Notices* 46.1 (2011), pages 317-330.

[5] D. PERELMAN, S. GULWANI, T. BALL et D. GROSSMAN. « Type-directed completion of partial expressions ». In : *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 2012, pages 275-286.

[6] H. D. T. NGUYEN, D. QI, A. ROYCHOUDHURY et S. CHANDRA. « Semfix : Program repair via semantic analysis ». In : *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pages 772-781.

[7] A. V. NORI, S. OZAIR, S. K. RAJAMANI et D. VIJAYKEERTHY. « Efficient synthesis of probabilistic programs ». In : *ACM SIGPLAN Notices* 50.6 (2015), pages 208-217.

[8] S. GULWANI. « Programming by examples ». In : *Dependable Software Systems Engineering* 45.137 (2016), pages 3-15.

[9] S. GULWANI, O. POLOZOV, R. SINGH et al. « Program synthesis ». In : *Foundations and Trends® in Programming Languages* 4.1-2 (2017), pages 1-119.

A ANNEXE

Un langage est principalement représenté par sa syntaxe et sa sémantique ; sa syntaxe décrit la forme propre de ses programmes, tandis que sa sémantique indique ce que ses programmes signifient i.e, ce que fait chaque programme lorsqu'il est exécuté. Nous présentons ci-dessous la syntaxe et la sémantique du langage moteur de *FlashFill*, pour plus de details, se référer à [4].

Syntaxe

FlashFill manipule des ensembles de programmes de chaînes de caractères. Ainsi, l'élément principal du langage est une expression de programme qui représente un programme de transformation de chaînes de caractères capables de transformer des chaînes d'entrée σ en des chaînes de sortie s. La figure 4 représente la syntaxe d'une expression de programme de transformation de chaînes de caractères.

$$\begin{array}{rcl}
\text{String expr } P & := & \texttt{Switch}((\mathrm{b}_1, \mathrm{e}_1), \cdot\cdot, (\mathrm{b}_n, \mathrm{e}_n)) \\
\text{Bool b} & := & \mathrm{d}_1 \vee \cdot\cdot \vee \mathrm{d}_n \\
\text{Conjunct d} & := & \pi_1 \wedge \cdot\cdot \wedge \pi_n \\
\text{Predicate } \pi & := & \texttt{Match}(v_i, \mathrm{r}, k) \;\mid\; \neg\, \texttt{Match}(v_i, \mathrm{r}, k) \\
\text{Trace expr e} & := & \texttt{Concatenate}(\mathrm{f}_1, \cdot\cdot, \mathrm{f}_n) \\
\text{Atomic expr f} & := & \texttt{SubStr}(v_i, \mathrm{p}_1, \mathrm{p}_2) \\
 & \mid & \texttt{ConstStr}(s) \\
 & \mid & \texttt{Loop}(\lambda w : \mathrm{e}) \\
\text{Position p} & := & \texttt{CPos}(k) \;\mid\; \texttt{Pos}(\mathrm{r}_1, \mathrm{r}_2, \mathrm{c}) \\
\text{Integer expr c} & := & k \;\mid\; k_1 w + k_2 \\
\text{Regular Expression r} & := & \texttt{TokenSeq}(\mathrm{T}_1, \cdot\cdot, \mathrm{T}_m) \\
\text{Token T} & := & C+ \quad\mid\quad [\neg C]+ \\
 & \mid & \text{SpecialToken}
\end{array}$$

FIGURE 4 – Syntaxe d'une expression de programme [4]

Les règles de production de cette grammaire sont les éléments de la forme $A := B$, le non-terminal P qui est une expression de programme de transformation de chaînes de caractères représente l'axiome de cette grammaire. Soit C la classe de caractères constituée de l'ensemble des caractères : numériques, alphabétiques (majuscule, minuscule), alphanumériques, d'autres caractères spécifiques comme les ponctuations et $\neg C$ la classe des éléments qui ne sont pas dans C ; nous présentons ci-dessous les éléments de la figure 4.

- p := *Cpos(k)*, *Pos(r_1, r_2, c)* est une expression de position, où c := k | $k_1 w + k_2$ est une expression entière,
- T := C + | $[\neg C]$ + |$SpecialToken$ est un token, $SpecialToken$ étant un *StartTok* ou un *EndTok* et r := *TokenSeq($T_1, .., T_m$)* est une séquence d'expressions régulières,
- f := SubStr(v_i, p_1, p_2) | ConstStr(s) | Loop(λw : w) est une expression atomique, e := *Concatenate(f_1,...,fn)* est une *"trace expression"*,
- π := Match(v_i, r, k) | ¬ Match(v_i, r, k) est un prédicat,
- P := *Switch((b_1, e_1),..,(b_n, e_n))* est une expression de programme et $b := d_1 \wedge .. \wedge d_n$ est expression booléenne.

Sémantique

Cette section présente la signification des opérateurs du langage précédent.

- ϵ dénote une chaîne de caractères vide, $\perp$ dénote une valeur indéfinie i.e, si l'un des arguments d'un constructeur est $\perp$, il retourne $\perp$.
- *ConstStr* est l'expression de constante ; soit *s* une sous-chaîne,*ConstStr(s)* = *s*.
- *Concatenate(f_1,...,fn)* = $f_1 f_2 ... f_n$.
- *TokenSeq($T_1, .., T_m$)* est une séquence de tokens $T_1, .., T_m$, tout comme l'opération de concaténation. Les Tokens sont représentés par leurs noms représentatifs ; par exemple *AlphTok* renvoie à une séquence de caractères alphabétiques, *NumTok* renvoie à une séquence de caractères numériques.
- *SubStr(v_i, p_1, p_2)* retourne la sous-chaîne de vi allant des positions p_1 à p_2, la chaîne étant numéroté de 0 à $n - 1$, où n est la longueur de vi. Par exemple, soit vi = "IBM" alors *SubStr(v_i, 1 ,2)* = "BM".

- *Cpos(k)* et *Pos*(r_1, r_2, *c)* sont des expressions de positions. Une position peut être obtenue en parcourant la chaîne de la gauche vers la droite (selon que k soit positif pour $Cpos(k)$ et c positif pour *Pos*(r_1, r_2, *c)*) ou de la droite vers la gauche (selon que k soit négatif ou c négatif). Pour une chaîne *s* donnée, nous avons les définitions suivantes :

  $$[\![\text{CPos}(k)]\!]\ s \quad = \quad \begin{cases} k & \text{if} \quad k \geq 0 \\ \text{Length}(s) + k & \text{otherwise} \end{cases}$$

  Par exemple, soit la chaîne de caractères s = "IBM", nous pouvons représenté quelques positions comme suit : Cpos(0) représente l'indice du premier caractère ("I") de *s* à partir de la gauche, Cpos(-1) représente l'indice du dernier caractère ("M") de *s* à partir de la droite i.e Cpos(0) = 0 et Cpos(-1) = 2.

  $$[\![\text{Pos}]\!]\ s = t, \exists t_1, t_2, 0 \leq t_1 < t \leq t_2, s[t_1 : t-1]\ \ match\ \ r_1, s[t : t_1]\ \ match\ \ r_2.$$

  Pos(r_1, r_2, *c)* retourne l'indice de la $c^i eme$ occurrence de la sous-chaîne qui match $r_1 r_2$ dans s à partir de la gauche si $c\, positif$ et à partir de la droite $sinon$.
  Par exemple, soit la chaîne de caractères s = "Vanes10Laure11" :
  - Pos(AlphTok, NumTok, 1) = 5 : indice de début de la chaîne *NumTok* dans la première occurrence de "AlphaTokNumtok" ("vanes10") à partir de la gauche ; il s'agit de l'indice de début de "10".
  - Pos(AlphTok, NumTok, -1) = 12 : indice de début de la chaîne *NumTok* dans la première occurrence de "AlphaTokNumtok" ("laure11") à partir de la droite ; il s'agit de l'indice de début de "11".
  - Si r_2 = ϵ, par convention ϵ n'a ni indice de début ni indice de fin, par conséquent, *Pos*(r_1, r_2, *c)* retourne l'indice de fin de $c^i eme$ occurrence de r_1 dans s.
- Une expression de programme P est un constructeur $Switch$ dont les arguments sont les paires disjointes (b_1, e_1),..,(b_n, e_n) où les b_i sont des expressions booléennes et les ei sont les *"trace expression"*.
- Le constructeur *Loop(*λ *w : e)* sur une entrée σ permet de générer la concaténation des chaînes obtenues en exécutant de manière répétitive l'expression e tout le long de σ. *Loop(*λ *w : e)* produit la concaténation d'une séquence $e_1, ...e_n$ où e_i est le $i^e me$ résultat d'exécution du $Loop$ sur σ lorsque le paramètre w vaut i.

FlashFill manipule des ensembles d'expressions de programmes pour des transformations de chaînes de caractères. Pour cela, il faut une structure de données permettant de manipuler aisément de grands ensembles de données donc, les opérations de construction d'expressions de programmes doivent pouvoir prendre des ensembles de données en arguments. Pour se faire, le langage correspondant n'est rien d'autre qu'une généralisation du langage présenté précédemment. En d'autres termes, il s'agit du langage précédent à la différence que les constructeurs ont été surchargés pour prendre en argument des ensembles de données.