Les amalgames : un mécanisme pour la structuration et la construction incrémentielle de programmes déclaratifs

Olivier Michel
LRI u.r.a. 410 du CNRS, Bâtiment 490, Université de Paris-Sud,
F-91405 Orsay Cedex, France.
email: michel@lri.fr

Résumé : 81/2 est un langage déclaratif permettant la définition de collections par des équations. Afin de structurer les programmes, un mécanisme primaire d'encapsulation permet de regrouper ces équations en *systèmes*. Ce mécanisme est insuffisant quand il s'agit de structurer de grands programmes, de partager ou d'extraire et de réutiliser des parties de programmes déjà écrites, ou bien encore quand on veut construire un programme comme résultat du calcul d'un autre programme.

Les *amalgames* généralisent la notion de système afin de permettre des systèmes « incomplets ». Deux opérations seulement sont permises sur les amalgames : la première étend la sémantique de l'opération d'accès à un élément dans un système et la seconde correspond à un opérateur de *concaténation* des amalgames.

Nous présentons une sémantique opérationnelle des amalgames dans le style SOS. Nous illustrons ensuite le calcul sur les amalgames en proposant un codage des fonctions arithmétiques totales. Appliqué au langage 81/2, ce formalisme permet de structurer un ensemble d'équations dans un style objet ou bien encore de modéliser la construction incrémentielle de programme.

1 Les systèmes en 81/2

La gestion des espaces de noms est un problème crucial pour les nouveaux langages de programmation. Par exemple les problèmes de modélisation de l'*héritage* dans les langages à objets [1], de la *délégation* dans les langages à prototypes [2], des *pages jaunes* ou autres mécanismes de nommage dans un environnement distribué [3], des *applets* dans les programmes construits dynamiquement [4], des mécanismes de *call-back*, etc., sont tous clairement reliés à la gestion des noms. Les sémantiques formelles des langages de programmation identifient habituellement la notion d'espace de noms à celle d'environnement [5]. De plus, le mécanisme de gestion des noms présent dans un compilateur ou un interprète est généralement «câblé» dans la définition du langage. Dans cet article, nous présentons une approche possible pour le calcul et la manipulation par le programmeur des espaces de noms, dans le cadre du langage déclaratif 81/2.

Le langage déclaratif 81/2 permet de définir des collections par des équations de la forme «*identificateur* = *expression*». Par exemple

$$\text{int } iota\,[10] \;=\; 0 \,\#\, (1 + iota : [9])$$

définit un vecteur de 10 entiers par une expression récursive. L'opérateur «#» représente la concaténation de deux vecteurs et «: [9]» représente la troncature à 9 éléments du vecteur argument. L'élément d'indice i du vecteur ainsi défini a pour valeur i (l'indexation des éléments commence à 0). On peut définir d'autres types de collections que des vecteurs, comme par exemple des suites infinies de valeurs, cf. [6].

Journées du GDR Programmation.
20, 21 et 22 novembre 1996.
Orléans.

1.1 Les systèmes

Un programme 81/2 est un ensemble d'équations. Afin de structurer cet ensemble, on peut regrouper les équations en paquet appelé *système* et correspondant à un *environnement*. Par exemple, le programme suivant

$$\begin{array}{rcl} hasard & = & \{ \begin{array}{rcl} intervalle & = & 2^{32}, \\ germe & = & 536, \\ valeur & = & (\$valeur * germe) \bmod intervalle, \ \ldots \ \} \end{array} \\ poisson & = & \{ \begin{array}{rcl} integre & = & hasard\,.\,valeur + \$integre, \ \ldots \ \} \end{array} \end{array}$$

regroupe les équations en deux paquets logiques : le premier système, *hasard*, correspond aux calculs nécessaires pour générer un nombre aléatoire uniformément distribué ; le deuxième système, *poisson*, utilise ce nombre aléatoire uniformément distribué pour calculer une autre variable aléatoire (qui suit par exemple une loi de Poisson). Dans cet exemple, on ne définit pas des vecteurs mais des suites de valeurs et l'opérateur « $ » permet d'accéder à l'élément précédent dans une suite.

Remarquons qu'un système est défini par une équation $id = exp$ de la même manière qu'une autre entité du langage : un système est une nouvelle sorte de collection, au même titre qu'un vecteur ou une suite infinie. Cependant, les opérations sur les systèmes sont très réduites : on ne peut créer un système que par énumération de ses éléments et on ne peut qu'accéder à un de ses éléments.

Une expression en dehors d'un système peut faire référence à une entité définie dans ce système, en utilisant l'opérateur « . » avec une référence au système et le nom de l'entité. On peut imbriquer les systèmes et redéfinir un nom ; les règles d'accès suivent la structure de blocs. On obtient ainsi un mécanisme rudimentaire de structuration correspondant à un mécanisme de visibilité des identificateurs.

1.2 Les insuffisances de la notion de système

Les systèmes comme mécanisme de structuration sont nettement insuffisants. On aimerait pouvoir

- paramétrer un système afin de factoriser du code ;
- utiliser un système comme environnement de liaison pour évaluer une expression arbitraire ;
- enrichir un système par l'ajout de définitions.

Cela nous amènera à construire un système comme le résultat d'un calcul, puisqu'un système peut être vu comme une collection d'un type particulier.

1.2.1 Paramétrer un système en admettant les références libres

On aimerait pouvoir définir le processus de Poisson indépendement du générateur aléatoire, afin d'utiliser *poisson* avec des variables aléatoires différentes. Une solution simple serait de permettre des références libres, i.e. des identificateurs qui ne réfèrent à aucune définition, et de fournir un mécanisme permettant de lier ces références libres à une définition.

C'est ce qui est possible dans un langage à liaison dynamique, contrairement aux langages à liaison lexicale. Par exemple, dans un langage à liaison lexicale comme Caml Light [7], la définition de fonction suivante :

$$\texttt{let}\ F = \texttt{let}\ y = 1\ \texttt{in}\ \texttt{let}\ f\ x = x + y + a\ \texttt{in}\ f$$

dans un environnement où a n'est pas défini provoque l'erreur « `variable` a `is unbound` », alors que dans un langage comme CommonLisp [8], une définition analogue provoquera la liaison lexicale de y et une liaison *dynamique* de a. Ainsi, en CommonLisp,

$$(\texttt{progn} \quad (\texttt{setq}\ a\ 1) \quad (\texttt{setq}\ y\ 666) \quad (F\ 1))$$

aura pour valeur 3 (la variable y apparaissant dans l'expression f étant liée syntaxiquement, la redéfinition de y par ($\texttt{setq}\ y\ 666$) n'a pas d'effet sur l'évaluation).

Remarquons que la construction «`let ... in ...`» utilisée pour la liaison syntaxique n'est pas la même que celle utilisée pour la liaison dynamique : `setq`. La structure du lieur n'est d'ailleurs pas la même car la liaison syntaxique reflète la portée par bloc dans le texte du programme, alors que la liaison effectuée par un `setq` est valide dans le temps jusqu'au prochain `setq`. Une autre différence est qu'un identificateur dans un lieur à porté syntaxique est une *variable muette*, c'est à dire sujette à α-conversion, alors que dans un lieur dynamique le nom est pertinent, ce qui permet la liaison «plus tard», indépendement de toute construction explicite reliant l'endroit de définition et l'endroit d'utilisation de l'identificateur. Si ce mode de gestion des noms admet une sémantique opérationnelle relativement simple à décrire, il a fallu attendre des travaux récents pour obtenir une théorie équationelle d'un mécanisme de capture de noms [9, 10, 11, 12].

Nous voulons adopter une stratégie analogue de résolution des références. Cependant, nous ne voulons pas distinguer références liées syntaxiquement et références liées dynamiquement : en effet, nous ne voulons pas redéfinir de liaison au cours du temps, mais simplement retarder le moment de cette liaison. Il n'est donc pas nécessaire de distinguer deux lieurs, mais simplement de permettre des expressions non closes. Ainsi le langage garde la propriété d'assignation unique (modulo la visibilité des identificateurs).

Avec une telle stratégie de gestion des liaisons, on peut résoudre facilement le problème de la paramétrisation du processus *poisson*. On définit *poisson* en laissant une référence libre à *hasard* et on veille à ce que cette référence soit dans la portée d'une définition de *hasard* quand *poisson* est utilisé :

$$
\begin{array}{rcl}
\mathit{uniform1} & = & \{\ldots\}, \\
\mathit{uniform2} & = & \{\ldots\}, \\
\mathit{poisson} & = & \{\mathit{integre} = \mathit{hasard}\,.\,\mathit{valeur} + \$\mathit{integre}, \quad \ldots \quad \} \\
\mathit{usage1} & = & \{\mathit{hasard} = \mathit{uniforme1}, \quad x = \mathit{poisson}\,.\,\mathit{integre}, \quad \ldots \quad \} \\
\mathit{usage2} & = & \{\mathit{hasard} = \mathit{uniforme2}, \quad y = \mathit{poisson}\,.\,\mathit{integre}, \quad \ldots \quad \}
\end{array}
$$

Dans le contexte de *usage1*, l'occurrence de *poisson* trouve une définition de *hasard* ce qui permet de calculer *integre* avec la variable aléatoire spécifiée par *uniform1*. Dans *usage2*, c'est la définition de *uniform2* qui sera référée. On voit que le mécanisme permettant de fournir une définition manquante peut simplement reposer sur la portée définie par les systèmes.

Notons qu'il existe une grande différence entre les fonctions et les expressions ouvertes : les arguments sont explicitement abstraits dans une fonction, alors que les références libres d'une expression ouverte le sont implicitement. Par suite, l'application de fonction doit être explicite, alors que ce n'est pas le cas pour la liaison des références. Le mécanisme de *curryfication* du λ-calcul et des langages fonctionnels permet de fournir une partie seulement des arguments attendus, mais il implique un ordre strict suivant lequel ces arguments doivent être fournis. Ce n'est que récemment que des extensions du λ-calcul ont été proposées [13, 9] pour s'affranchir de cette contrainte, et cela au prix d'une théorie assez lourde. À l'opposé, la fermeture des systèmes est naturellement incrémentielle et non ordonnée.

1.2.2 Calculer les lieurs

Le mécanisme précédent permet de fournir une définition manquante, mais il présente le désavantage suivant : le résultat ne correspond pas au système initial dans lequel certaines références ont été résolues, car il comporte des définitions supplémentaires.

Pour pallier cet inconvénient, il suffit d'étendre l'opération d'accès à un élément d'un système, à la sémantique d'un lieur comme `let rec`. Ainsi,

$$\{a = 1,\, b = 2\}\,.\,(a + b + x) \quad \text{est analogue à l'expression} \quad \texttt{let rec}\ a = 1\ \texttt{and}\ b = 2\ \texttt{in}\ (a + b + x)$$

et s'évaluera en $3 + x$ (x est une référence libre dans l'environnement $\{a = 1,\, b = 2\}$). Remarquons que la nouvelle sémantique de **.** étend la sémantique précédente : l'évaluation de a dans l'environnement $\{a = 1,\, b = 2\}$ correspond bien à l'accès à l'entité nommée a dans le système.

Cependant, contrairement à une sélection, les définitions dans un `let rec` ne sont pas des objets du calcul. En effet, dans l'expression : `let rec` $\diamond$ `in` $\star$, les définitions $\diamond$ et l'expression $\star$ ont une nature différente : une définition n'est pas une expression, et les définitions sont textuellement rassemblées par le programmeur.

Dans l'expression $\diamond\,\mathbf{.}\,\star$, les définition représentées par le système $\diamond$ sont de même nature que $\star$: elles peuvent donc être calculées. On a ainsi «internalisé» dans le langage, c'est à dire rendu accessible et modifiable par le calcul, la notion d'environnement. De ce point de vue, la construction $\mathbf{.}$ correspond plutôt à la construction `with` sur les enregistrements du langage Pascal.

1.2.3 Enrichir les systèmes

Nous ne disposons pas d'opérateur permettant de «rajouter» des définitions à un système. C'est pourquoi nous introduisons l'opérateur # qui permet de faire l'union de deux systèmes :

$$\{a = 1,\, b = c\} \,\#\, \{c = a,\, d = b\} \qquad \text{va se réduire en} \qquad \{a = 1,\, b = c,\, c = a,\, d = b\}$$

puis s'évaluer en $\{a = 1,\, b = 1,\, c = 1,\, d = 1\}$. L'opérateur # est analogue à l'extension symétrique d'un enregistrement, modulo le fait que les références libres dans chacun des système se lient avec les définitions apportées par l'autre système.

2 Une définition formelle des amalgames

Nous appelons *amalgames* les systèmes généralisés afin d'admettre les références libres dans une expression, ainsi qu'une notion d'accès étendue à la notion d'évaluation et une opération d'extension. Nous proposons ci-dessous une sémantique opérationnelle du calcul sur les amalgames, dans un style à la SOS [14].

2.1 Le langage Σ des amalgames

Nous ne nous intéressons qu'au «calcul pur» sur les amalgames, sans faire intervenir d'autres constructions. Le langage qui nous intéresse se réduit alors à :

$$\Sigma ::= x^n \mid x_n \mid e\,\mathbf{.}\,e' \mid e \,\#\, e' \mid \{x = e,\, x' = e', \ldots\}$$

où x appartient à un ensemble de noms $\textsc{Id}$, n est un entier, e, e' des éléments de Σ. On suppose que tous les noms utilisés en partie droite des équations d'un système sont distincts.

Références libres et liées. x^n et x_n sont respectivement des références libres et liées. Une référence liées x_n correspond à un identificateur référant à une définition apparaissant dans le n^{eme} scope englobant. Cette notation est une extension de la numérotation de De Bruijn [15], l'index étant complété par un nom. Une référence libre x^n correspond à l'attente d'une définition devant apparaître dans un scope de niveau q, avec $q \geq n$. Un exposant « ^{n} » permet donc de s'échapper d'un scope. Un scope est créé par la définition d'un système ou par un opérateur d'accès. Par exemple, dans le terme suivant, les flèches indiquent à quelle définition une référence est liée :

$$\{a = 1,\ b = \{a = 2\}\,\mathbf{.}\,\{a = 3,\ t = a_0,\ u = a_1,\ v = a_2,\ w = a^3\}\}$$

2.2 Les termes corrects : $\Sigma_{\mathcal{C}}$

Tous les termes de Σ ne représentent pas des termes correctement liées : des références libres sont peut-être dans la portée d'une définition et des références sont peut-être indûment liées. Nous définissons un prédicat permettant de décider si un terme de Σ est correct.

Avant de définir ce qu'est un terme correct, il est nécessaire de définir un certain nombre de fonctions et de prédicats auxiliaires [1] et nous spécifierons ensuite la notion de correction.

Définition 1 **(Scope, $\mathtt{Dom}()$ et $\mathcal{E}_{\mathrm{L}}$)** Un *scope* est un élément de $\textsc{Scope} = \{\star^n, \star_n\} \cup \mathcal{P}(\textsc{Id})$. Un *environnement de liaison* est un élément de $\mathcal{E}_{\mathrm{L}} = \mathtt{List}(\textsc{Scope})$. La fonction $\mathtt{Dom}()$ de Σ dans $\textsc{Scope}$ est définie

1. Nous utilisons, pour la manipulation de listes, les conventions suivantes : $\mathtt{List}(X)$ représente l'ensemble des listes d'éléments de X ; la liste vide est notée $<>$; la concaténation d'un atome h et d'une liste t se note $h{::}t$; l'accès au n^{eme} élément d'une liste l se note $\mathtt{nth}(n, l)$ la liste privée de son premier élément se note $\mathtt{tl}(l)$. Par convention, $\mathtt{tl}(<>) = <>$ et les itérés de $\mathtt{tl}()$ se définissent par : $\mathtt{tl}^0(l) = l$, $\mathtt{tl}^1(l) = \mathtt{tl}(l)$ et $\mathtt{tl}^n(l) = \mathtt{tl}^{(n-1)}(\mathtt{tl}(l))$.

par :

$$
\begin{array}{lcl}
\mathtt{Dom}(l, x^n) & = & \star^n \\
\mathtt{Dom}(l, x_n) & = & \star_n \\
\mathtt{Dom}(l, \{x = e, \dots\}) & = & \{x, \dots\} \\
\mathtt{Dom}(l, e \,\#\, e') & = & \mathtt{Comb}(\mathtt{Dom}(e), \mathtt{Dom}(e')) \\
\mathtt{Dom}(l, e \,\centerdot\, e') & = & \mathtt{Dom}(e')
\end{array}
$$

où la fonction $\mathtt{Comb}()$: $\textsc{Scope} \mapsto \{\star^n, \star_n\}$ définit la combinaison des constantes $\star^n$ et $\star_n$ pour calculer le domaine d'une concaténation :

$$
\begin{array}{lcl}
\mathtt{Comb}(\star_n, p) & = & \star_n \\
\mathtt{Comb}(p, \star_n) & = & \star_n \\
\mathtt{Comb}(\{\dots\}, \{\dots\}) & = & \star_n \\
\mathtt{Comb}(p, p') & = & \star^n
\end{array}
$$

La constante $\star_n$ est utilisée quand on ne connaît pas les liaisons, car des réductions doivent encore avoir lieu. La constante $\star^n$ est utilisée quand on ne connaît pas les liaisons, car il y a «un trou» dans l'expression, i.e. une référence libre. On ne distingue généralement pas les constantes $\star_n$ et $\star^n$. On note alors ces constantes par $\star$ qui désigne indifféremment un de ces deux éléments.

Nous avons besoin de chercher récursivement dans un $\mathcal{E}_{\mathrm{L}}$ un élément x. Si l'élément n'est pas trouvé, ou si une constante $\star$ est rencontrée, alors on renvoie $\star$, sinon on renvoie la position dans la liste de l'ensemble qui contient x.

Définition 2 **(La fonction index())** La fonction totale $\mathtt{index}() : \mathcal{E}_{\mathrm{L}} \times \textsc{Id} \times \mathbb{N} \mapsto \{\star\} \cup \mathbb{N}$ est définie par :

$$
\begin{array}{lcl}
\mathtt{index}(<>, x, n) & = & \star \\
\mathtt{index}(\star :: t, x, n) & = & \star \\
\mathtt{index}(h :: t, x, n) & = & \mathtt{if}\, x \in h \,\mathtt{then}\, n \,\mathtt{else}\, \mathtt{index}(t, x, n+1)
\end{array}
$$

Définition 3 **(Le prédicat correct())** Le prédicat $\mathtt{correct}(e) : \mathcal{E}_{\mathrm{L}} \times \Sigma \mapsto \textsc{Bool}$ est une fonction totale définie par :

$$
\begin{array}{lcl}
\mathtt{correct}(l, x^p) & = & (\mathtt{index}(\mathtt{tl}^p(l), x, p) == \star) \\
\mathtt{correct}(l, x_p) & = & (x \in \mathtt{nth}(p+1, l)) \\
\mathtt{correct}(l, \{x = e, \dots\}) & = & \bigwedge \mathtt{correct_B}(\{x, \dots\} :: l, e) \\
\mathtt{correct}(l, e \,\#\, e') & = & \mathtt{correct}(l, e) \wedge \mathtt{correct_B}(l, e') \\
\mathtt{correct}(l, e \,\centerdot\, e') & = & \mathtt{correct}(l, e) \wedge \mathtt{correct_B}(\mathtt{Dom}(e) :: l, e')
\end{array}
$$

et $\mathtt{correct}(e)$ abrège $\mathtt{correct}(<>, e)$. L'expression $\bigwedge \dots$ doit s'interpréter comme la conjonction sur toutes les parties droites e des définitions du système.

Définition 4 **(Le langage $\Sigma_{\mathcal{C}}$)** Nous définissons le langage $\Sigma_{\mathcal{C}}$, sous-langage de Σ tel que chaque élément de $\Sigma_{\mathcal{C}}$ est correct : $\Sigma_{\mathcal{C}} = \{x \mid x \in \Sigma \wedge \mathtt{correct}(x)\}$.

2.3 La liaison des expressions

La fonction de liaison $\mathtt{bind}(e)$ effectue la transformation de certaines références libres en références liées.

Définition 5 **(La fonction bind())** La fonction totale $\mathtt{bind}()$ de $\mathcal{E}_{\mathrm{L}} \times \Sigma \mapsto \Sigma$ est spécifiée par :

$$
\begin{array}{lcl}
\mathtt{bind}(l, x_n) & \equiv & x_n \\
\mathtt{bind}(l, x^n) & \equiv & \mathtt{if}(\star == \mathtt{index}(\mathtt{tl}^n(l), x, n))\,\mathtt{then}\, x^n \,\mathtt{else}\, x_{\mathtt{index}(\mathtt{tl}^n(l), x, n)} \\
\mathtt{bind}(l, \{x = v, \dots\}) & \equiv & \{x = \mathtt{bind}(\{x, \dots\} :: l, v), \dots\} \\
\mathtt{bind}(l, e \,\#\, e') & \equiv & \mathtt{bind}(l, e) \,\#\, \mathtt{bind}(l, e') \\
\mathtt{bind}(l, e \,\centerdot\, e') & \equiv & \mathtt{bind}(l, e) \,\centerdot\, \mathtt{bind}(\mathtt{Dom}(e) :: l, e')
\end{array}
$$

Il apparaît que la stratégie de liaison est lexicale retardée. La liaison d'une référence libre x^n s'effectue en remontant *au moins* n scopes englobant puis la liaison est effectuée avec la première définition ayant pour nom x. Si la constante $\star$ est rencontrée, alors c'est qu'une liaison future reste possible et donc la référence doit rester libre. La liaison d'une sélection respecte ce schéma : la liaison de $e \,\centerdot\, e'$ effectue la liaison de e' dans les définitions de $\mathtt{Dom}(e)$.

2.4 L'environnement d'évaluation

La relation de réduction sur les termes de $\Sigma_\mathcal{C}$ procède par substitution des définitions aux références liées et par contraction des concaténations et des sélections. Il est nécessaire, quand on réduit une sous-expression, de connaître les définitions qui apparaissent dans un scope englobant. Pour disposer de cette information, la relation de réduction utilise un *environnement d'évaluation*. Comme les expressions peuvent être imbriquées, nous utilisons une structure de liste pour l'environnement où chaque élément de la liste est une fonction partielle qui associe à un identificateur l'expression correspondante.

DÉFINITION 6 **(Environnement et environnement d'évaluation $\mathcal{E}_\mathrm{E}$)** Un *environnement* est une fonction partielle σ de signature $\textsc{Id} \rightarrow \Sigma$. On note $[x \mapsto e]$ la fonction qui associe l'expression e à l'identificateur x et qui est indéfinie pour tous les autres identificateurs. Si σ est un environnement, alors $\sigma' = \sigma \uplus [x' \mapsto e']$ est l'environnement tel que : $\sigma'(x) = e'$ si $x = x'$ et $\sigma(x)$ sinon. L'écriture $[x \mapsto e, \ldots, x' \mapsto e']$ est une abréviation de $[x \mapsto e] \uplus \cdots \uplus [x' \mapsto e']$. Un *environnement d'évaluation*, noté ρ, est un élément de $\mathcal{E}_\mathrm{E} = \mathtt{List}((\textsc{Id} \rightarrow \Sigma) \cup \{\star_n, \star^n\})$. Les constantes spéciales $\star$ correspondent à un environnement inconnu pour un scope donné.

Pour injecter une expression dans un scope autre que celui où elle est définie, il faut mettre à jour les indices des références liées. La fonction $\mathtt{Lift}()$ appliquée à un terme doit changer les références qui réfèrent à l'extérieur de ce terme mais pas celles qui sont liées *dans* le terme.

DÉFINITION 7 **(La fonction $\mathtt{Lift}()$)** On définit la fonction totale $\mathtt{Lift}()$ de Σ dans Σ : $\mathtt{Lift}(e) = \mathtt{Lift}(0, e)$ où la fonction $\mathtt{Lift}(n, e)$ propage le lift sur l'expression e en incrémentant la valeur de n qui correspond au nombre de scopes rencontrés, à chaque fois que cela est nécessaire. La fonction $\mathtt{Lift}()$ de $\mathbb{N} \times \Sigma$ dans Σ est définie par :

$$
\begin{array}{lcl}
\mathtt{Lift}(n, x_p) & \equiv & x_q \qquad \text{avec } q = \mathtt{if}\, p \geq n\, \mathtt{then} (p+1)\, \mathtt{else}\, p \\
\mathtt{Lift}(n, x^p) & \equiv & x^p \\
\mathtt{Lift}(n, \{x = v, \ldots\}) & \equiv & \{x = \mathtt{Lift}(n+1, v), \ldots\} \\
\mathtt{Lift}(n, e \,\#\, e') & \equiv & \mathtt{Lift}(n, e) \,\#\, \mathtt{Lift}(n, e') \\
\mathtt{Lift}(n, e \,\mathbf{.}\, e') & \equiv & \mathtt{Lift}(n, e) \,\mathbf{.}\, \mathtt{Lift}(n+1, e')
\end{array}
$$

Une référence n'est incrémentée que si elle réfère à un scope qui est supérieur au nombre de scopes traversés. On est ainsi assuré que seules les liaisons qui réfèrent une définition hors de leur expression, seront mises à jour.

La fonction $\mathtt{Nenv}(\rho, e)$ permet de construire un nouvel environnement de réduction à partir de l'environnement existant ρ et de l'expression e.

DÉFINITION 8 **(La fonction $\mathtt{Nenv}()$)** La fonction $\mathtt{Nenv}()$ de $\mathcal{E}_\mathrm{E} \times \Sigma$ dans $\mathcal{E}_\mathrm{E}$ est définie par :

$$
\begin{array}{lcl}
\mathtt{Nenv}(\rho, \{x = e, \ldots\}) & = & [x \mapsto e, \ldots] :: \mathtt{Lift}_\mathtt{E}(\rho) \\
\mathtt{Nenv}(\rho, e) & = & \mathtt{Dom}(e) :: \mathtt{Lift}_\mathtt{E}(\rho) \\
\mathtt{Lift}_\mathtt{E}(<>) & = & <> \\
\mathtt{Lift}_\mathtt{E}(\star :: t) & = & \star :: \mathtt{Lift}_\mathtt{E}(t) \\
\mathtt{Lift}_\mathtt{E}(h :: t) & = & (\lambda x.\, \mathtt{Lift}(h(x))) :: \mathtt{Lift}_\mathtt{E}(t)
\end{array}
$$

La liaison d'une expression nécessitera de connaître les définitions d'un système, définitions qui apparaissent dans un environnement d'évaluation. La fonction $\mathtt{Dom}_\mathtt{E}()$ permet de récupérer les identificateurs liés dans un environnement d'évaluation.

DÉFINITION 9 **(Les fonctions $\mathtt{Dom}_\mathtt{E}()$ et $\mathtt{Def}()$)** On définit les fonctions $\mathtt{Dom}_\mathtt{E}() : \mathcal{E}_\mathrm{E} \mapsto \mathcal{E}_\mathrm{L}$ et $\mathtt{Def}() : (\textsc{Id} \rightarrow \Sigma) \rightarrow \mathcal{P}(\textsc{Id})$ par :

$$
\begin{array}{lcl}
\mathtt{Dom}_\mathtt{E}(<>) & = & <> \\
\mathtt{Dom}_\mathtt{E}(\star :: t) & = & \star :: \mathtt{Dom}_\mathtt{E}(t) \\
\mathtt{Dom}_\mathtt{E}(\sigma :: t) & = & \mathtt{Def}(\sigma) :: \mathtt{Dom}_\mathtt{E}(t) \\
\mathtt{Def}([x \mapsto e, x' \mapsto e', \ldots]) & = & \{x, x', \ldots\}
\end{array}
$$

2.5 La garde de l'évaluation

Intuitivement, une terme est complètement réduit quand il ne comporte plus de références liées et quand les concaténations et les sélections ont été réduites autant que possible. C'est ce qu'exprime le prédicat $\mathtt{C}()$. La relation entre le processus de réduction et le prédicat $\mathtt{C}()$ est précisée par un théorème dans la section suivante.

DÉFINITION 10 **(Le prédicat $\mathtt{C}()$)** Le prédicat $\mathtt{C}() : \mathcal{E}_{\mathrm{L}} \times \Sigma \mapsto \textsc{Bool}$ est défini par induction structurelle sur la structure des termes :

$$\begin{array}{lcl}
\mathtt{C}(l, x_n) & = & \mathtt{false} \\
\mathtt{C}(l, x^n) & = & \star_n \notin \mathtt{tl}^n(l) \\
\mathtt{C}(l, \{x = v, \dots\}) & = & \bigwedge \mathtt{C}(\mathtt{Dom}(\{x = v, \dots\}) :: l, v) \\
\mathtt{C}(l, e \,\#\, e') & = & ((e \notin \textsc{Sys}) \vee (e' \notin \textsc{Sys})) \wedge \mathtt{C}(\mathtt{Dom}(e) :: l, e') \\
\mathtt{C}(l, e \,.\, e') & = & (e \notin \textsc{Sys}) \wedge \mathtt{C}(l, e) \wedge \mathtt{C}(l, e')
\end{array}$$

2.6 Spécification de l'évaluateur

La relation ternaire $\rho \vdash e \rightarrow e'$ de $\mathcal{E}_{\mathrm{E}} \times \Sigma \times \Sigma$ exprime que e se réduit en un pas de réduction en e' dans un environnement d'évaluation ρ. On note ϵ l'environnement d'évaluation vide : $<>_{\mathcal{E}_{\mathrm{E}}}$. Si $\epsilon \vdash e \rightarrow e'$, alors on note plus simplement $e \rightarrow e'$.

$e \downarrow \rho$ est le prédicat qui est faux si $\exists\, e' \in \Sigma, \rho \vdash e \rightarrow e'$ et qui est vrai sinon. On dit que e est un terme en *ρ-forme normale* car il n'est plus possible de le réduire dans ρ. Un terme est en *forme normale* s'il est en ϵ-forme normale. On abrège $e \downarrow \epsilon$ par $e \downarrow$.

$\rho \vdash e \rightarrow_n e'$ représente la *composition* de n réductions. Autrement dit, $\rho \vdash e \rightarrow_n e'$ signifie qu'il existe $e_0, \dots, e_n \in \Sigma$ où $e_0 \equiv e$ et $e_n \equiv e'$ avec $\rho \vdash e_0 \rightarrow e_1, \rho \vdash e_1 \rightarrow e_2, \dots, \rho \vdash e_{n-1} \rightarrow e_n$. Par convention, $\rho \vdash e \rightarrow_0 e'$ signifie que $e \downarrow \rho$ et $e \equiv e'$.

On dit que le calcul de e a pour résultat e' et on note $e \twoheadrightarrow e'$ si et seulement s'il existe n tel que $e \rightarrow_n e'$ et $e' \downarrow$.

La figure (1) définit une sémantique opérationnelle des amalgames. L'évaluation se fait par induction sur la structure d'un terme. La sémantique que nous proposons est « partiellement stricte » : un programme qui termine correspond à un terme qui possède une forme normale. Certains termes peuvent posséder une forme normale alors qu'un de ses sous-termes n'en possède pas (le sous-terme donne lieu à une série infinie de réductions). Par exemple, cela peut être le cas si le sous-terme se trouve à gauche d'une opération de sélection.

3 Trois propriétés importantes de l'évaluation

La spécification des règles de réductions et des prédicats utilisés nous permet la démonstration de trois propriétés sur l'évaluation. Les démonstrations sont développées dans [16].

Déterminisme de l'évaluation : $\forall e \in \Sigma, \quad (e \twoheadrightarrow e') \wedge (e \twoheadrightarrow e'') \;\Rightarrow\; e' \equiv e''$

Le résultat suivant est important car il permet d'évaluer efficacement $e \downarrow \rho$ ou bien de déterminer si $p \neq 0$ dans $\rho \vdash e \rightarrow_p e'$ à l'aide d'un critère syntaxique : $\mathtt{C}()$.

$\mathtt{C}()$ et forme normale : $\forall \rho \in \mathcal{E}_{\mathrm{E}}, \forall e \in \Sigma, \quad \mathtt{correct}(\mathtt{Dom}_{\mathtt{E}}(\rho), e) \;\Rightarrow\; \mathtt{C}(\mathtt{Dom}_{\mathtt{E}}(\rho), e) = e \downarrow \rho$

Remarque : Voici un exemple tel que $e \downarrow \rho \neq \mathtt{C}(\mathtt{Dom}_{\mathtt{E}}(\rho), e)$: l'expression $x_1 \downarrow \epsilon$ est vraie. En effet, la fonction $\mathtt{nth}(2, \epsilon)$ n'est pas définie (et donc aucune règle ne peut s'appliquer), et par définition, $\mathtt{C}(\mathtt{Dom}_{\mathtt{E}}(\rho), x_1)$ est faux. On vérifie bien que $\mathtt{correct}(\mathtt{Dom}_{\mathtt{E}}(\epsilon), x_1)$ est faux.

Enfin, on peut montrer que l'évaluation d'un terme de $\Sigma_{\mathcal{C}}$ a pour résultat un terme de $\Sigma_{\mathcal{C}}$. Cette propriété est analogue à la propriété de *subject reduction* d'un type qui demande à ce que le type d'une expression se conserve lors des réductions de l'expression. On peut d'ailleurs voir les éléments de $\Sigma_{\mathcal{C}}$ comme les habitants d'un type et le prédicat $\mathtt{correct}()$ comme l'inférence de ce type.

L'évaluation préserve la correction : $\forall e \in \Sigma_{\mathcal{C}}, \quad e \twoheadrightarrow e' \;\Rightarrow\; e' \in \Sigma_{\mathcal{C}}$

$$Ref^n \quad : \quad \frac{}{\rho \vdash x^n \to x^n} \quad \star_n \in \mathtt{tl}^n(\rho)$$

$$Ref_C \quad : \quad \frac{u = \mathtt{nth}(n+1, \rho)(x) \quad u' = \mathtt{bind}(\mathtt{Dom_E}(\rho), u)}{\rho \vdash x_n \to u'} \quad \mathtt{C}(\mathtt{Dom_E}(\rho), u)$$

$$Ref_{\neg C} \quad : \quad \frac{u = \mathtt{nth}(n+1, \rho)(x)}{\rho \vdash x_n \to x_n} \quad \neg\, \mathtt{C}(\mathtt{Dom_E}(\rho), u)$$

$$\{\dots\} \quad : \quad \frac{\begin{array}{c} \dots \\ \rho' = \mathtt{Nenv}(\rho, \{x = u, \dots\}) \quad \rho' \vdash u \to_n u' \end{array}}{\rho \vdash \{x = u, \dots\} \to \{x = u', \dots\}} \quad n \in \{0,1\} \wedge \sum n \geq 1$$

$$\#_1 \quad : \quad \frac{s = \mathtt{bind}(\mathtt{Dom_E}(\rho), \{x = u, x' = u', \dots\})}{\rho \vdash \{x = u, \dots\} \,\#\, \{x' = u', \dots\} \to s} \quad \begin{cases} x \neq x' \\ \mathtt{C}(\mathtt{Dom_E}(\rho), \{x = u, \dots\}) \\ \mathtt{C}(\mathtt{Dom_E}(\rho), \{x' = u', \dots\}) \end{cases}$$

$$\#_2 \quad : \quad \frac{\rho \vdash u \to_n u' \quad \rho \vdash v \to_m v'}{\rho \vdash u \,\#\, v \to u' \,\#\, v'} \quad n, m \in \{0,1\} \wedge n + m \geq 1$$

$$\centerdot_{S_1} \quad : \quad \frac{}{\rho \vdash \{x = u, \dots\} \,\centerdot\, v \to v} \quad \mathtt{C}(\mathtt{Dom_E}(\mathtt{Nenv}(\rho, \{x = u, \dots\})), v)$$

$$\centerdot_{S_2} \quad : \quad \frac{\rho \vdash u \to_n u' \quad \mathtt{Nenv}(\rho, u') \vdash v \to v'}{\rho \vdash u \,\centerdot\, v \to u' \,\centerdot\, v'} \quad n \in \{0,1\} \wedge u \in \textsc{Sys}$$

$$\centerdot_{\neg S_1} \quad : \quad \frac{\rho \vdash u \to u' \quad v' = \mathtt{bind}(\mathtt{Dom_E}(\mathtt{Nenv}(\rho, u')), v)}{\rho \vdash u \,\centerdot\, v \to u' \,\centerdot\, v'} \quad u \notin \textsc{Sys} \wedge u' \in \textsc{Sys}$$

$$\centerdot_{\neg S_2} \quad : \quad \frac{\rho \vdash u \to_m u' \quad \mathtt{Nenv}(\rho, u') \vdash v \to_n v'}{\rho \vdash u \,\centerdot\, v \to u' \,\centerdot\, v'} \quad \begin{cases} u \notin \textsc{Sys} \wedge u' \notin \textsc{Sys} \\ n, m \in \{0,1\} \wedge n + m \geq 1 \end{cases}$$

FIG. 1 – *La sémantique opérationnelle de l'évaluateur partiellement strict des amalgames.* SYS *représente l'ensemble des termes construits avec* $\{\dots\}$.

4 Exemples

La sémantique précédente a été directement traduite en règles Mathematica [17], en utilisant la connection entre C() et forme normale. Nous disposons ainsi d'un évaluateur qui a servi à calculer les exemples de cette section. D'autres exemples sont donnés dans [18, 19]

4.1 Le codage des fonctions arithmétiques totales

Nous allons illustrer le fait qu'on peut traduire les fonctions arithmétiques récursives[2] en amalgames. Cela montre la puissance formelle d'expression du calcul sur les amalgames. Nous nous restreindrons à la classe des *fonctions totales*. En effet, le schéma de traduction que nous proposons n'assure pas la non-terminaison du calcul de l'amalgame associé à l'application d'une fonction sur des arguments n'appartenant pas au domaine de définition. Cependant, rien ne prouve qu'il n'est pas possible de spécifier une autre notion de définissabilité par amalgame qui permettrait de faire coïncider fonctions partielles et expressions n'ayant pas de forme normale.

2. La définition de la récursion primitive que nous utilisons diffère légèrement de la définition usuelle mais elle est équivalente. Nous adoptons cette définition car elle est plus pratique à traduire. La fonction $\boldsymbol{U_i^p}$ représente la i^{eme} projection, $\boldsymbol{S}$ la fonction successeur et $\boldsymbol{Z}$ la fonction qui retourne zéro.

4.1.1 Définissabilité par amalgames

DÉFINITION 11 **(Représentation des entiers dans Σ)** Pour chaque $n \in \mathbb{N}$ un terme $\ulcorner n \urcorner \in \Sigma$ est défini de la manière suivante :

$$\ulcorner 0 \urcorner \equiv \{b = vrai^0\}$$
$$\ulcorner n+1 \urcorner \equiv \{p = \ulcorner n \urcorner, b = faux^0\}$$

DÉFINITION 12 **(Définissabilité par amalgames)** Soit φ une fonction numérique d'arité p. On dit que φ est *définissable par amalgame*, ou encore a-définissable, s'il existe un système s tel que :

$$\forall \vec{n}, \quad s[\ulcorner n_1 \urcorner, \ldots, \ulcorner n_p \urcorner] \equiv \{a1 = \ulcorner n_1 \urcorner, \ldots, ap = \ulcorner n_p \urcorner\} \centerdot s \twoheadrightarrow \{valeur = \ulcorner \varphi(\vec{n}) \urcorner, \ldots\}$$

Les ... dans le résultat indiquent que s doit être un système qui doit spécifier au moins une définition pour *valeur* et qui peut contenir d'autres définitions si cela est nécessaire. Les noms $a1, a2, \ldots$ sont par convention des noms donnés aux arguments de φ et qui ne sont pas utilisés ailleurs (s peut dépendre de ces noms).

4.1.2 Le codage de l'arithmétique

Les fonctions arithmétiques de base sont définissables par amalgames : il suffit de prendre les termes

$$\begin{array}{lcl} \boldsymbol{U_i^p} & \equiv & \{valeur = a\mathrm{i}\} \\ \boldsymbol{S} & \equiv & \{valeur = \{p = a1,\, b = faux^0\}\} \\ \boldsymbol{Z} & \equiv & \{valeur = \ulcorner 0 \urcorner\} \end{array}$$

La fonction numérique P tel que $P(n+1) = n$ est définissable par amalgame, on prend :

$$\boldsymbol{P} \equiv \{valeur = a1 \centerdot p\}$$

Les fonctions définissables par amalgames sont closes par composition. En effet, soient $\phi, \psi_1, \ldots, \psi_n$ a-définis respectivement par $G, H1, \ldots, H\mathrm{m}$, alors

$$\varphi(\vec{n}) = \phi(\psi_1(\vec{n}), \ldots, \psi_m(\vec{n}))$$

est défini par

$$\begin{array}{lrcl} \{ & valeur & = & args \centerdot G, \\ & args & = & \{a1 = H1 \centerdot valeur, \ldots, a\mathrm{m} = H\mathrm{m} \centerdot valeur\} \quad \} \end{array}$$

L'idée maintenant est définir une forme conditionnelle : $\lambda x, y, z.$(si $x = 0$ alors y sinon z). Cela se fait grâce au champ b de la représentation d'un nombre qui nous permet d'extraire une référence libre servant d'aiguillage. Mais attention, il ne faut pas que cette référence libre soit capturée et devienne liée. Par convention, le terme $\mathrm{ifZ}_{x,y,z}$ représente

$$\begin{array}{lrcl} \{ & codeCond & = & x \centerdot b \\ & codeVrai & = & y, \\ & codeFaux & = & z, \\ & valIf & = & aiguillage^0 \centerdot codeCond_1 \quad \} \end{array}$$

sachant que *aiguillage* sera défini un jour comme :

$$\text{Branchement} \equiv \{vrai = stop^0 \centerdot codeVrai^2,\ faux = stop^0 \centerdot codeFaux^2\}$$

une fois que l'on aura extrait *valIf* de ifZ. La référence libre *stop* a pour but d'empêcher la liaison des références libres $vrai^0$et $faux^0$qui apparaissent dans *codeVrai* et *codeFaux* (une fois que ceux-ci auront été liés puis substitués). On l'élimine en fournissant une définition inoffensive à *stop*, par exemple $stop = \{\}$.

L'idée pour définir un schéma d'appel récursif est de créer une expression correspondant au terme traduisant la fonction mais ne contenant aucune référence liée, puis de fournir la liaison de ces références au moment de l'appel de la fonction.

Nous pouvons maintenant montrer que les fonctions définissables par amalgames sont closes par récursion primitive. Nous allons détailler le schéma pour une fonction φ à deux variables, en utilisant la conditionnelle et le schéma récursif : soit la fonction φ définie par $\varphi(0,y) = \phi(y)$ et $\varphi(x+1,y) = \psi(x, \varphi(x, \psi_1(y)))$ avec ϕ a-défini par F, ψ a-défini par G et ψ_1 a-défini par H. Alors φ est a-défini par (x et y sont les noms utilisés pour les arguments de φ) :

$$\begin{array}{rcl}
\{\mathit{valeur} & = & \mathit{value} \centerdot \mathit{finalval} \\
\mathit{value} & = & \{\ \ \mathit{finalval} = \{\mathit{stop} = \{\}\} \centerdot (\mathit{recval} \centerdot \mathit{valIf}), \\
& & \quad \mathit{fct} = \mathrm{ifZ}_{x^0,\ G[y^0],\ F[x^0, \mathit{parameter} \centerdot (\mathit{call} \centerdot \mathit{valIf})]} \\
& & \quad \mathit{recval} = \{\mathit{call} = \{X = x^1 \centerdot p, Y = H[y^1]\}, \mathit{aiguillage} = \text{Branchement}\} \centerdot \\
& & \qquad\qquad (\{x = \ulcorner n_1 \urcorner, y = \ulcorner n_2 \urcorner, \mathit{parameter} = \{x = X, y = Y\}\} \centerdot \mathit{fct})\}\}
\end{array}$$

De la façon dont nous avons présenté la traduction de la récursion primitive, il doit être évident au lecteur qu'il nous importe peu de décrémenter x ou de l'incrémenter dans le schéma de la récursion primitive : cela donne lieu naturellement à l'implémentation de la minimisation.

4.1.3 Exemple de l'addition

La définition de l'addition prend la forme d'une récursion primitive :

$$\begin{array}{l}
\mathit{add}(0,n) = n \\
\mathit{add}(n+1,m) = \mathit{add}(n, S(m))
\end{array}$$

d'où les fonctions $\phi = U_1^1, \psi = U_2^2, \psi_1 = S$. La traduction de $\mathit{add}(2,1)$ donne le terme (les variables Tmp$x$ sont des méta-variables utilisées pour décrire le terme par morceaux, ce ne sont pas des identificateurs de Id) :

$$\begin{array}{rcl}
\mathit{add}(2,1) \equiv \{\mathit{valeur} & = & \text{Tmp1} \\
\mathit{a82} & = & \{p = \{p = \{b = \mathit{vrai}^0\}, b = \mathit{faux}^0\}, b = \mathit{faux}^0\}, \\
\mathit{a83} & = & \{p = \{b = \mathit{vrai}^0\}, b = \mathit{faux}^0\}\}
\end{array}$$

$$\begin{array}{rcl}
\text{Tmp1} \equiv \{\mathit{recval} & = & \text{Tmp2}, \\
\mathit{finalval} & = & \{\mathit{stop} = \{\}\} \centerdot \mathit{recval}_1, \\
\mathit{fct6} & = & \{\mathit{valif} = \mathit{aiguillage}^0 \centerdot \mathit{codeCond}_1, \\
& & \quad \mathit{codeCond} = x^0 \centerdot b^0, \\
& & \quad \mathit{codeVrai} = \{\mathit{val36} = \mathit{a86}_0, \mathit{a86} = y^0\} \centerdot \mathit{val36}_0, \\
& & \quad \mathit{codeFaux} = \{\mathit{val37} = \mathit{a88}_0, \mathit{a87} = x^0, \mathit{a88} = \mathit{call}^0 \centerdot \mathit{retvalue}^0\} \centerdot \mathit{val37}_0\}\} \centerdot \mathit{finalval}_0
\end{array}$$

$$\begin{array}{rcl}
\text{Tmp2} \equiv \{\mathit{call} & = & \{X = x^1 \centerdot p^0, Y = \mathit{Tmp3}, \mathit{retvalue} = \mathit{param}^0 \centerdot \mathit{fct6}^0 \centerdot \mathit{valif}^0\}, \\
\mathit{aiguillage} & = & \{\mathit{vrai} = \mathit{stop}^0 \centerdot \mathit{codevrai}^1, \mathit{faux} = \mathit{stop}^0 \centerdot \mathit{codefaux}^1\}\} \\
& & \qquad \centerdot \{x = \mathit{a82}_3, y = \mathit{a83}_3, \mathit{param} = \{x = X^0, y = Y^0\}\} \centerdot \mathit{fct6}_2 \centerdot \mathit{valif}^0
\end{array}$$

$$\begin{array}{rcl}
\text{Tmp3} \equiv \{\mathit{val34} & = & \{\mathit{comp6} = \mathit{a85}_1, \mathit{val35} = \{p = \mathit{comp6}_1, b = \mathit{faux}^0\}\} \centerdot \mathit{val35}_0, \\
\mathit{a84} & = & x^1, \\
\mathit{a85} & = & y^1\} \centerdot \mathit{val34}_0
\end{array}$$

qui s'évalue en 37 pas d'évaluation et environ 7 minutes sur une `HP 9000/770` avec Mathematica version 2.1 en le terme attendu :

$$\begin{array}{rcl}
\{\mathit{valeur} & = & \{p = \{p = \{p = \{b = \mathit{vrai}^0\}, b = \mathit{faux}^0\}, b = \mathit{faux}^0\}, b = \mathit{faux}^0\}, \\
\mathit{a82} & = & \{p = \{p = \{b = \mathit{vrai}^0\}, b = \mathit{faux}^0\}, b = \mathit{faux}^0\}, \\
\mathit{a83} & = & \{p = \{b = \mathit{vrai}^0\}, b = \mathit{faux}^0\}\}
\end{array}$$

Évidement, ce codage n'est pas très efficace, mais le codage de l'arithmétique dans les amalgames purs consiste essentiellement en un exercice formel.

4.2 Exemple de structuration dans le style objet

Les notions de système et de concaténation de systèmes permettent la manipulation d'environnements. La composition de systèmes permet de définir des enregistrements extensibles. De plus, les expressions ouvertes et la capacité à fournir des définitions manquantes vont nous permettre de définir un comportement similaire à ceux que l'on trouve dans les langages orientés objets. Il est possible de définir et composer des fragments de programme en exhibant un comportement de *classe* et d'*instanciation de classe* que l'on trouve dans ces langages. Nous allons montrer à travers un exemple comment «émuler» un style programmation proche de ceux des langages à objets.

Un système représente à la fois les notions de *classe* et de *constructeur* qui sont utilisées pour créer une instance d'une classe. Les arguments nécessaires au constructeur sont les références libres du système. L'*instanciation* d'une classe correspond à la concaténation du système avec les arguments requis par le constructeur. L'ajout de définitions supplémentaires, par l'utilisation de la concaténation, correspond au mécanisme d'*héritage*.

Un système complet (sans références libres) correspond à un objet comme c'est le cas dans les langages à objets. Mais pour ce qui concerne 81/2, les valeurs de l'objets sont des collections : il n'existe pas de notion de message ni de méthodes. Le modèle objet qui correspond le mieux à l'esprit de ce qui est faisable en 81/2 est le modèle *embedding based* où toutes les informations qui concernent l'objet se trouvent dans celui-ci. Il est évident que les mécanismes évolués de protection et d'encapsulation qui sont proposés par les langages à objets sont absents dans le modèle que nous proposons.

Pour illustrer ce style de programmation, on se propose de définir en suivant un style proche des langages orientés objets, une modélisation de la trajectoire d'une planète en mouvement circulaire uniforme autour d'une étoile, elle- même en mouvement rectiligne uniforme. Pour ce faire, on définit une classe *Mobile* d'objets animés d'un mouvement. La classe *Mobile* est représentée par un système avec deux références libres : *initial* qui représente la position initiale de l'objet, et *dp* qui représente les déplacements élémentaires de l'objet. À l'aide de ces références libres, qui sont des vecteurs de deux éléments correspondant aux axes Ox et Oy, le système *Mobile* définit une *position*:

$$\begin{array}{lcl} \mathit{Mobile}@0 & = & \{\mathit{position} = \mathit{initial}\}; \\ \mathit{Mobile} & = & \{\mathit{position} = \$\mathit{Mobile} \,.\, \mathit{position} + \mathit{dp}\}; \end{array}$$

Le champ *position* d'un *Mobile* est une suite de valeur représentant la trajectoire du mobile au cours du temps.

Une fois que *Mobile* est définit, nous pouvons définir une nouvelle classe d'objets : les objets mobile dotés d'une vitesse uniforme. La classe *U−Trajectoire* attend une position initiale (requis par la classe *Mobile* dont il hérite) et un vecteur *vitesse* pour pouvoir s'instancier :

$$\mathit{U{-}Trajectoire} \quad = \quad \mathit{Mobile} \,\#\, \{\mathit{dp} = \mathit{vitesse}\}$$

Le système *U−Trajectoire* est un système avec toutes les définitions du système *Mobile* car c'est un système étendu par les définitions de *dp* utilisées pour calculer les déplacements élémentaires avec une trajectoire uniforme (on suppose que *vitesse* est une constante et est égale à la différence de la trajectoire uniforme entre deux éléments consécutifs de la suite). L'opération de concaténation rassemble ces deux systèmes et effectue les liaisons entre la référence libre *dp* qui apparaît dans *Mobile* et la définition de *dp* dans le système anonyme complétant les définitions de *U−Trajectoire*.

On poursuit avec cet exemple en utilisant *Mobile* pour représenter la trajectoire circulaire d'une planète en révolution autour d'une étoile (le soleil par exemple) qui se déplace suivant une trajectoire uniforme. La classe *C−Trajectoire* attends un rayon, un centre et une vitesse angulaire :

$$\begin{array}{lcllcl} \mathit{C{-}Trajectoire} & = & \mathit{Mobile} \,\#\, \{ & \mathit{initial} & = & \{\mathit{centre} \,.\, 0, \mathit{angle} + \mathit{centre} \,.\, 1\}, \\ & & & \mathit{dp} & = & \{\mathit{dx}, \mathit{dy}\}, \\ & & & \mathit{ot} & = & \$t, \\ & & & t & = & \mathit{ot} + \mathit{vitesse\_angulaire}, \\ & & & \mathit{dx} & = & \ldots, \\ & & & \mathit{dy} & = & \ldots\}; \end{array}$$

Il ne nous reste plus qu'à instancier les classes pour décrire le « système solaire » :

$$\begin{array}{lcl} \mathit{Soleil} & = & \mathit{U{-}Trajectoire} \,\#\, \{\mathit{vitesse} = \{1.0, 1.0\}, \mathit{initial} = \{0.0, 0.0\}\}; \\ \mathit{Terre} & = & \mathit{C{-}Trajectoire} \,\#\, \{\mathit{angle} = 1.0, \mathit{centre} = \mathit{Soleil} \,.\, \mathit{position}\} \end{array}$$

La figure (2) illustre la trajectoire résultante de la terre (une trajectoire circulaire se déplaçant autour d'un centre, le soleil, qui lui est animé d'une translation uniforme).

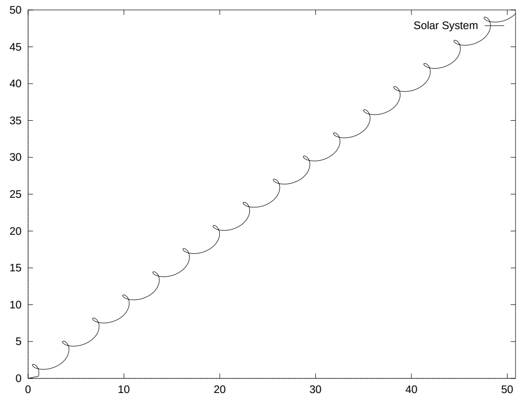

FIG. 2 – *Simulation d'un mouvement circulaire uniforme autour d'un centre lui-même en mouvement rectiligne uniforme.*

4.3 Construction dynamique et incrémentielle de programme

Nous avons récemment assisté à l'émergence d'une nouvelle classe d'application : les applications distribuées dont les composants ne sont pas nécessairement connu au lancement du programme. Un exemple est donné par la notion d'*applets* introduits par les navigateurs [4, 20]. Un applet est une portion de code qui est chargée dynamiquement, *à la demande*, lors de l'acquisition de données faisant référence à cet applet. Le chargement dynamique permet l'ajout de fonctionnalités, la modification du comportement d'un navigateur, etc.

Ce mécanisme trouve une traduction immédiate avec les amalgames. Un applet est modélisé sous la forme d'un terme ouvert. Les références libres d'un applet sont complétées dans son environnement d'évaluation, implicitement, sur la base de leur nom.

Par exemple, supposons qu'un applet doit être composé de deux parties A et B, chacune développée séparément mais pouvant faire référence l'une à l'autre. La partie A utilise un certain nombre de services, a priori inconnus du site où on assemble A et B, mais qui se trouvent tous dans une bibliothèque L_A. Cependant, si le site fournit sa propre implémentation d'un service, dans une bibliothèque L_S, on aimerait que ce soit cette implémentation là qui soit utilisée. Il en va de même pour B. Dans le langage des amalgames, la construction *dynamique et incrémentielle* du programme consiste en le terme

$$\mathit{prog} = L_A \,.\, (L_B \,.\, (L_S \,.\, (A \,\#\, B)))$$

avec A, B, L_A et L_B obtenus sur un site distant, par exemple par

$$A = \mathit{appletA} \,@\, \texttt{ftp.lri.fr}$$

où $x \,@\, \texttt{s}$ va chercher la définition de x sur le site $\texttt{s}$. Nous disons que la construction est dynamique car elle est calculée lors de l'exécution et qu'elle est incrémentielle car elle se fait à partir de parties de code déjà programmées.

5 Discussion

Nous avons vu que les amalgames définissaient une notion de nom (les éléments d'un système sont identifiés par des noms et deux identificateurs de même nom référencent la même expression) et une notion d'environnement (en effet, le premier argument d'une sélection est utilisé comme environnement pour l'évaluation du second argument)

La notion de nom est différente de la notion classique de variable : la seule opération permise sur un nom est le test de l'égalité avec un autre nom [21] alors qu'une variable est sujette à α-renommage afin d'éviter les phénomènes de *capture de variable* [22]. Cette notion de capture de variable est centrale dans l'approche des amalgames comme dans d'autres travaux. Par exemple, la notion de nom est essentielle dans le π-calcul [23] (« The pervasive notion we seize upon is *naming* » dans [23]) ainsi que dans les travaux concernant le traitement dynamique de la notion d'environnement [24, 25, 26, 27, 28, 29]

L'accès aux expressions redéfinies est possible dans les amalgames par l'utilisation des indices. Par exemple, dans l'expression $\{a = 1, b = \{a = 2, c = a_1\}\}$, l'indice permet à a_1 de référencer la définition englobante de a malgré que celle-ci soit redéfinie localement au système b. On trouve un mécanisme similaire permettant l'accès à des redéfinitions dans une extension du λ-calcul, le λN-calcul [9]. Cette extension introduit la notion de nom et une notation similaire aux indices de De Bruijn pour l'accès aux redéfinitions, suivant un mécanisme analogue au nôtre. Cette extension du λ-calcul permet la modélisation de certains traits caractéristiques des langages à objets. Cependant, l'introduction d'une opération *explicite* de fermeture est nécessaire alors qu'elle est *implicite* dans les amalgames. Le λN-calcul sert de cadre formel à la définition d'enregistrements extensibles (ce qui est fondamental pour la formalisation de la notion d'héritage des langages à objets). L'utilisation des opérations de concaténation et de sélection des amalgames permettent d'obtenir un comportement similaire, mais Dami définit un système de typage, qui n'a pas d'équivalent dans les amalgames.

Dans ce papier, nous avons mis en évidence le point de vue «incrémentiel» des amalgames. Ce mécanisme de construction de programme repose totalement sur le phénomène de *capture de variables*, et peut aussi être étudié dans le cadre du λ-calcul. Dans [12], la capture de variable se fait par l'utilisation d'une nouvelle entité, le *contexte*. Un contexte représente un λ-terme avec des trous, l'opération de remplissage de ces trous se faisant par une nouvelle opération d'application où la capture des variables est désirée. Il faut alors introduire une nouvelle sorte d'abstraction, notée Φ, ainsi qu'un nouveau mécanisme de réduction des expressions, car la cohabitation des λ-termes avec les contextes n'est pas triviale. Une conséquence des difficultés de cohabitation de ces deux types dans le λ-calcul est la nécessité de séparer l'espace de noms des variables abstraites, des noms soumis à capture. La proposition étudiée dans [30] effectue une tentative similaire, mais en résolvant le problème des noms par un système de typage qui assure l'absence de collision des deux espaces. Les amalgames n'ont pas ce problème de collision de noms de deux types différents, dans la mesure où toutes les références sont sujettes à capture (il n'y a pas de variable, que des noms). Le mécanisme est par conséquent plus simple. Par suite, on notera que la notion de fonction est nécessairement absente du calcul sur les amalgames, mais l'exemple du codage des fonctions récursives primitives totales permet de penser que l'expressivité formelle des amalgames seuls est largement suffisante.

Parmi les travaux à mener, nous aimerions développer d'autres versions des opérateurs de sélection et de concaténation. Par exemple, la concaténation n'introduit pas de nouveau scope, ce qui serait nécessaire si on veut qu'une référence libre de l'un des arguments puisse se lier à une définition présente dans l'autre argument, en présence d'une définition englobante.

Par ailleurs, sélections et concaténations admettent une interprétation très simple en terme de construction de graphe de flot de données : la concaténation correspond à l'union des graphes des arguments avec « recollage » des arcs pendants (les références libres), et la sélection correspond à l'extraction (suivie éventuellement d'une complétion) d'un sous-graphe dans un graphe donné. Ces opérations correspondent donc à des combinateurs puissants et il est nécessaire d'étudier la géométrie qu'ils induisent sur les graphes.

Références

[1] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Sciences*, pages 321–346. Springer-Verlag, April 1994.

[2] David Ungar and Randall B. Smith. Self: The power of simplicity. *ACM SIGPLAN Notices*, 22(12):227–242, December 1987.

[3] Christian Queinnec. DMEROON, overview of a distributed class-based causally-coherent data model. *Lecture Notes in Computer Science*, 1068:297–309, 1996.

[4] François Rouaix. *Caml Applets User Guide*. INRIA Rocquencourt, April 1996.

[5] Suresh Jagannathan and Gul Agha. A reflective model of inheritance. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 350–371, Utrecht, The Netherlands, June 1992. Springer-Verlag.

[6] J.-L. Giavitto and J.-P. Sansonnet. 81/2 : data-parallélisme et data-flow. *Techniques et Sciences de l'Ingénieur*, 12 - Numéro 5, 1993. Numéro spécial Langages à Parallélisme de Données.

[7] Xavier Leroy. The caml light system, release 0.6 documentation and user's manual. Technical report, INRIA, September 1993.

[8] Guy L. Steele, Jr. An overview of common LISP. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 98–107. ACM, ACM, August 1982.

[9] Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches*. Ph.D. thesis, University of Geneva, 1994.

[10] Laurent Dami. Functional programing with dynamic binding. Technical report, CUI Genève, 1994.

[11] Jacques Garrigue. Dynamic binding and lexical binding in a transformation calculus. In *Proceedings of the Fuji International Workshop on Functional and Logic Programming*. World Scientific, 1995.

[12] Shinn-Der Lee and Daniel P. Friedman. Enriching the lambda calculus with contexts: toward a theory of incremental program construction. In *International Conference on Functional Programming*. ACM, May 1996.

[13] Jacques Garrigue and H. Aït-Kaci. The typed polymorphic label-selective lambda-calculus. In *Principles of Programming Languages*, Portland, 1994.

[14] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report 19, Aarhus University, 1981.

[15] Nicolas G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.

[16] Olivier Michel. Représentations dynamiques de l'espace dans un langage déclaratif de simulation. Technical report, Université de Paris-Sud, centre d'Orsay, December 1996. (à paraître).

[17] Stephen Wolfram. *Mathematica*. Addison-Wesley, Redwood City, CA, 1988.

[18] Olivier Michel. Design and implementation of 81/2, a declarative data-parallel language. Technical Report 1012, Laboratoire de Recherche en Informatique, December 1995.

[19] Olivier Michel. Introducing dynamicity in the data-parallel language 81/2. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *EuroPar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Sciences*, pages 678–686. Springer-Verlag, August 1996.

[20] Sun Microsystems. *The Java^{TM} Language Specification*, 1.0 beta edition, October 1995.

[21] Martin Odersky. A functional theory of local names. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 48–59, Portland, Oregon, January 17–21, 1994. ACM Press.

[22] Henk Barendregt. *The Lambda Calculus: its Syntax and Semantics*. North-Holland, 1981 (1st ed) revised 84.

[23] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh, October 1991.

[24] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Austin, TX*, pages 348–355, New York, NY, 1984. ACM.

[25] R. Burstall and B. Lampson. A kernel language for modules and abstract data types. Technical Report 1, DEC SRC, September 1984.

[26] John Lamping. A unified system of parameterization for programming languages. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 316–326. ACM, ACM, July 1988.

[27] Shinn-Der Lee and Daniel P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 479–492, Charleston, South Carolina, January 1993.

[28] Suresh Jagannathan. Metalevel building blocks for modular systems. *ACM Transactions on Programming Languages and Systems*, 16(3):456–492, May 1994.

[29] Christian Queinnec and David DeRoure. Sharing code through first-class environments. *ACM SIGPLAN Notices*, 31(6):251–261, June 1996.

[30] Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. Technical Report RIMS-1098, Research Institute for Mathematical Sciences, Kyoto University, August 1996.