Implémentation OCaml d'un langage pour la différenciation automatique

Toulouse Solal

Le but de ce stage de L3 est d'implémenter une version réduite du langage Dex décrit dans l'article *Getting to the point*. Ce langage permet d'effectuer la différentiation automatique expliquée dans un second article des mêmes auteurs : *You Only Linearize Once* ou de manière plus détaillée dans le livre de Griewank et Walther *Evaluating Derivatives* en *forward mode* et en *reverse mode* à l'aide d'une combinaison d'opérations simples expliquées dans la suite. Les objectifs sont donc d'écrire un interprète et un typeur pour ce langage, d'implémenter les différentes transformations dont résulte la différenciation automatique, et finalement, de construire un générateur aléatoire de programmes source afin de tester ces transformations. Ce rapport tentera d'expliquer, d'une part, de manière synthétique, les notions de l'article, en y ajoutant certaines précisions omises par les auteurs, et d'autre part, les choix d'implémentation que nous avons faits.

Table des matières

1 Le langage Linear A

L'article utilise une version extrèmement réduite du langage *Dex*. Dans un premier temps, nous nous intéresserons à un langage intermédiaire appelé *Linear A*. Ce langage contient les éléments de base d'un langage de programmation : déclaration de fonctions, de variables, opérations unaires et binaires. Sa spécificité est l'existence de deux sortes de variables :

- les variables non-linéaires qui peuvent être utilisées sans restrictions et qui peuvent subir toutes les opérations binaires et unaires implémentées.
- les variables linéaires qui ne peuvent subir qu'un nombre restreint d'opérations :
  - l'addition de deux variables linéaires
  - la multiplication d'une variable linéaire par une variable non-linéaire

— la duplication d'une variable linéaire pour en obtenir deux nouvelles (*dup*)

— l'abandon d'une variable linéaire (*drop*)

Ces variables ne peuvent être initialisées qu'à zéro. Ces opérations permettent donc uniquement d'évaluer des variables linéaires à la valeur zéro pour un programme clos du langage *Linear A*. Ce n'est pas un soucis, car ce qui nous importera sera simplement de connaître les différentes opérations subies par ces variables, et non leur valeur finale. Leur autre particularité est qu'elles ne doivent être utilisées qu'une et une seule fois, d'où l'existence des deux dernières opérations. Le *drop* tel que décrit dans l'article peut prendre une expression quelconque en argument et non seulement une variable, son rôle est alors de détruire l'intégralité des variables linéaires libres de cette expression. En réalité, pour les utilisations qui en sont faites dans l'article, nous pouvons restreindre son application à des variables.

Une autre contrainte est qu'il est impossible d'imbriquer des opérations. Il est donc uniquement possible d'effectuer une opération sur des variables. Pour enchaîner plusieurs étapes d'un calcul, il est nécessaire de le décomposer en une suite d'opérations élémentaires.

Finalement, un programme de ce langage est une suite de déclarations de fonctions.

Ainsi une fonction du langage *Linear A* peut être vue comme l'implémentation d'une application linéaire de l'espace de ses entrées linéaires dans l'espace de ses résultats linéaires. Les variables non-linéaires correspondent alors aux scalaires utilisés par la fonction.

La fonction linéaire associée à la matrice $\begin{pmatrix} 1 & 1 \\ 2 & 0 \end{pmatrix}$ peut par exemple être représentée par le programme suivant :

```
1 def f(;x1 : real, x2 : real) =
2   let (;x1', x1'' : real, real) = dup(x1) in
3   let (a : real;) = 2 in
4   let (; y1 : real) = x1' + x2 in
5   let (; y2 : real) = a * x1'' in
6   (; y1, y2)
```

D'un point de vue pratique, lors de la déclaration ou de l'appel de variables, nous séparerons à l'aide d'un point-virgule les variables non-linéaires (à gauche) des variables linéaires (à droite).

Les types présents dans *Linear A* sont définis récursivement :

— les réels (`real`)

— les tuples de types (`[t1, t2, ...]`)

Un tuple de variables ne doit pas mélanger les variables linéaires et les variables non-linéaires afin d'obtenir un objet soit linéaire soit non-linéaire. Avoir des objets non mixtes sera, par la suite, une condition pour pouvoir appliquer l'opération d'unzipping.

L'existence de tuples entraîne la présence d'une opération supplémentaire, appelée *Unpack* dans l'article. Elle permet d'extraire les différentes composantes d'un tuple de taille n dans n variables.

2 Linéarisation

Cette première opération correspond au *forward mode*. Elle transforme un programme exclusivement non linéaire en un programme mixte où à chaque entrée non-linéaire x du programme initial est associée une entrée linéaire $\dot{x}$ et à chaque sortie non-linéaire y est associée une sortie linéaire $\dot{y}$. $\dot{x}$ peut être interprété comme une variation de x. Ainsi, $x = x_1 * x_2$ devient, par exemple, $\dot{x} = (\dot{x_1} * x_2) + (x_1 * \dot{x_2})$.

Cette transformation prend, en plus de l'expression `e` à linéariser, une une table d'association, qui à une variable de `e` associe la variable linéaire correspondante. En réalité,

comme les variables linéaires ne peuvent apparaître qu'une fois (là où l'utilisation des variables de `e` n'est pas restreinte), cette table associe à chaque variable de `e` une liste de variables linéaires fraîches correspondantes.

Dans nos programmes, nous écrirons `x'` la variable linéaire associée à `x`. Voici un exemple de linéarisation pour une fonction de $\mathbb{R}$ dans $\mathbb{R}$ qui à un réel x associe le tuple $[x^2]$:

```
1 def f(x : real;) =
2   let (y : real;) = x * x in
3   [y]
```

⇓ *Linearize*

```
1 def f'(x : real; x' : real) =
2   let (y1 : real;) = x * x in
3   let (y : [real];) = [y1] in
4   let (; x1', x2' : real, real) = dup(x') in
5   let (;y1' : real) = x * x1 in
6   let (;y2' : real) = x * x2 in
7   let (;y3' : real) = y1' + y2' in
8   let (;y' : [real]) = [y3'] in
9   (y; y')
```

Comme x apparaît deux fois dans le programme source, la variable linéaire associée $\dot{x}$ apparaîtra également à deux reprises dans le programme linéarisé. Etant donné qu'une variable linéaire ne peut être utilisée qu'une seule fois, `x'` doit être dupliqué (ligne 4). Plus généralement, lors de la déclaration d'une variable dans le programme source, il est nécessaire de compter ses occurences dans la suite afin de définir le bon nombre de variables linéaires correspondantes à l'aide de `dup`. L'exemple suivant illustre le cas où une variable x apparaît trois fois dans la suite du programme (`e1` est une expression se linéarisant en `e1'`) :

```
1 let (x : real;) = e1 in (x, x, x;)
```

⇓ *Linearize*

```
1 let (x : real; x' : real) = e1' in
2 let (; x1' : real, x2' : real) = dup(x') in
3 let (; x3' : real, x4' : real) = dup(x1') in
4 (x, x, x; x2', x3', x4')
```

La syntaxe de ces chaînes de déclarations est lourde, il pourrait donc être intéressant d'ajouter un dup^n, qui aurait pour rôle de dupliquer n fois une variable. Cela recouvrirait, par la même occasion, les utilisations de *drop*, grâce au cas $n = 0$. Comme nous le verrons en partie 5, l'opération complémentaire de *dup* est l'addition, il serait donc également judicieux d'avoir la possibilité d'additionner n variables en une seule opération.

3 Unzipping

Cette opération a pour but de transformer un programme mixte en deux programmes, l'un traitant les variables linéaires, et l'autre les variables non-linéaires. La difficulté réside dans le fait que les opérations sur les variables linéaires peuvent utiliser une variable non-linéaire. Plus précisément, c'est lors de la multiplication linéaire que l'on a également besoin d'une variable non-linéaire. Il ne suffit donc pas de séparer le programme en deux, il faut aussi conserver la déclaration des variables non-linéaires afin qu'elles puissent à

la fois être utilisées dans la partie linéaire et dans la partie non-linéaire, sans toutefois dupliquer les calculs. Pour ce faire, nous utilisons un contexte. Un contexte est une chaîne de déclarations de variables où la dernière déclaration comporte un trou, dans lequel il suffit de placer une expression que l'on souhaite évaluer afin de la rendre close.

$$let\ (x_1, x_2, ...) = e_1\ in\ let\ (y_1, y_2, ...) = e_2\ in\ ...\ in\ [\ ]$$

Pour la fonction linéarisée de l'exemple précédent, voilà le résultat de l'opération d'unzipping (le résultat est simplifié, les `let (;) = () in ...` ont par exemple été retirés) :

```
1  def f'(x : real; x' : real) =
2    let (y1 : real;) = x * x in
3    let (y : [real];) = [y1] in
4    let (; x1', x2' : real, real) = dup(x') in
5    let (;y1' : real) = x * x1 in
6    let (;y2' : real) = x * x2 in
7    let (;y3' : real) = y1' + y2' in
8    let (;y' : [real]) = [y3'] in
9    (y; y')
```

$$\Downarrow\ Unzip$$

```
1  def f'_nl (x : real;) =
2    let (y1 : real;) = x * x in
3    let (y : [real];) = [y1] in
4    (y, x;)
5
6  def f'_l (x : real; x' : real) =
7    let (;x1', x2' : real, real) = dup(x') in
8    let (;y1' : real) = x * x1 in
9    let (;y2' : real) = x * x2 in
10   let (;y3' : real) = y1' + y2' in
11   let (;y' : [real]) = [y3'] in
12   (; y')
```

Comme le montre l'exemple ci-dessus, lors de la déclaration d'une fonction f, la fonction non-linéaire f_{nl} est construite en la partie non-linéaire de e (le corps de f) dans le contexte résultant du unzipping de e. À cela, on ajoute en sortie les résultats non-linéaires intermédiaires de f_{nl} utilisés par f_l. L'appel de la fonction initiale f, est donc transformé en deux appels : Un premier appel de f_{nl} avec les arguments non-linéaires de f. Ainsi qu'un appel de la partie linéaire f_l avec les arguments linéaires de f ainsi que les sorties utiles de f_{nl}. `let (y; y') = f(x; x')` est, par exemple, transformé en :
`let (y, z;) = f_nonlin(x;) in let (; y') = f_lin(z; x')`, dans le cas où `f_lin` utilise de la variable non-linéaire `z`.

Pour qu'un contexte fonctionne, il est nécessaire qu'il ne contienne pas deux définitions d'une même variable, afin d'éviter que la seconde ne masque la première. Pour cela, les programmes subissent une première transformation qui modifie le nom des variables dont les noms entrent en collision.

4 Transposition

Cette troisième et dernière transformation transforme une fonction f en sa fonction "transposée" f^T. Si f est une fonction de E dans F, alors f^T est une fonction de F dans E. Les variables linéaires d'entrée de f seront notées avec un point : $\dot{x}$, celles de sortie de f, donc d'entrée de f^T seront notées avec deux points : $\ddot{x}$. Cette opération prend une expression `e`, une liste de variables représentant les entrées de `e`, ainsi qu'une seconde liste

de variables fraîches associées aux sorties de `e`, qui est la liste des variables libres de la transposée de `e`.

En s'appuyant sur la vision matricielle d'un programme (*cf. Partie 2*), la transposition correspond à la transposition matricielle. Sur l'exemple simple de l'addition linéaire, la matrice associée est $\begin{pmatrix}1 & 1\end{pmatrix}$, sa transposée : $\begin{pmatrix}1\\1\end{pmatrix}$ correspond à la duplication d'une variable linéaire. La transposée de `x = x1 + x2` est donc `(y1, y2) = dup(y)`.

De manière réciproque, un *dup* se transpose en une addition. L'article semble dire que *dup* peut prendre en argument une variable d'un type quelconque, alors que l'addition ne peut s'effectuer qu'entre des réels. Il a donc fallu étendre l'addition à tous les types pour pouvoir obtenir une transposée complète.

Les règles telles qu'elles sont données dans l'article ne s'appliquent qu'à des programmes purement linéaires ou purement non-linéaires, afin d'obtenir des règles simplifiées. C'est pour cela qu'il est nécessaire d'appliquer l'opération d'unzipping avant de transposer un programme.

La règle pour la déclaration de variables linéaires est assez complexe. En voici un exemple simple :

```
let (; x1 : real) = x2 in x1 + x3
```

$$v_{in} = [x_2; x_3]$$
$$v_{out} = [y]$$
$$\Downarrow \textit{Transpose}$$

```
let (; x1', x3' : real, real) = dup(y) in (;x1', x3')
```

Ici, `x1 + x3` est transformé en `dup(y)`, où `y` est l'entrée de l'expression transposée, et `x2` est transformé en `x1'`, car dans l'expression initiale, `x1` est la sortie de l'expression `x2`. Par la suite, on ajoute `x3'` aux sorties de l'expression transposée, car la variable n'a pas été utilisée.

Le principe général est que les deux expressions e_1 et e_2 de *Let* $(...) = e_1$ *in* e_2 sont interverties, et que les variables déclarées correspondent à toutes les variables libres de e_2. Il faut ensuite ajouter toutes celles d'entre elles qui n'ont pas été utilisées en sortie de e_1^T. Nous devons cependant faire attention à ce que les variables soient déclarées dans l'ordre des résultats de e_2^T. Pour cela, nous utilisons la convention que, pour une expression e, les résultats de e^T apparaissent en fonction de l'ordre d'apparition des variables libres dans e.

La règle de transposition du *Unpack* n'est pas spécifiée dans l'article. Nous avons donc introduit la règle suivante qui traite un cas simple :

```
let ([x1, x2,...] : [t1, t2, ...]) = x in (x1, x2, ...)
```

$\Downarrow$ Transpose

```
[x1', x2', ...]
```

C'est en fait la règle symétrique de celle proposée dans l'article pour traiter les tuples.

Pour le cas général, nous avons effectué la transformation suivante, afin de se ramener au cas précédent et d'utiliser la règle du *Let* pour le reste :

```
let ([x1, x2,...] : [t1, t2, ...]) = x in e
```

$\Downarrow$

```
let (x1, x2, ... : t1, t2, ...) = let ([x1', x2' , ...] : [t1,
    t2, ...]) = x in (x1', x2', ...) in e'
```

Ci-dessous, voilà le résultat de la transposée sur l'exemple des transformations précédentes :

```
def f'_nl (x : real;) =
  let (y1 : real;) = x * x in
  let (y : [real];) = [y1] in
  (y, x;)

def f'_l (x : real; x' : real) =
  let (;x1', x2' : real, real) = dup(x') in
  let (;y1' : real) = x * x1 in
  let (;y2' : real) = x * x2 in
  let (;y3' : real) = y1' + y2' in
  let (;y' : [real]) = [y3'] in
  (; y')
```

⇓ *Transpose*

```
def f'_nl_t (x : real;) =
  let (y1 : real;) = x * x in
  let (y : [real];) = [y1] in
  (y, x;)

def f'_l_t (x : real; y'a : [real]) =
  let (;x1'a, x2'a : real, real) =
  let (;x2'b, y1'a : real, real) =
  let (;y1'b, y2'a : real, real) =
  let (;y3'a : real) =
  let (;y'b : [real]) = y'a in
  let ([y3'b] : [real]) = y'b in y3'b
  in dup(y3'a)
  in let (;z1 : real) = x * y2'a in
  (z1, y1'b)
  in let (;z2 : real) = x * y1'a in
  (z2, x2'b)
  in x1'a + x2'a
```

Un programme transposé est généralement plus gros que le programme initial, car à de nombreux moments, il est indispensable d'ajouter des *Let* pour conserver un programme bien formé. Notemment dans la règle du *Let*, où l'étape qui consiste à ajouter des variables en sortie d'une expression e nécessite de déclarer chaque résultat e dans une variable.

Comme la transposée matricielle, cette transformation doit vérifier la propriété suivante (où (...|...) est le produit scalaire canonique de $\mathbb{R}^n$) : $\forall(\dot{X_1}, X_2, \ddot{X_3}), (\dot{X_1}|\ddot{e}[X_2; \ddot{X_3}]) = (\ddot{X_3}|\dot{e}[X_2; \dot{X_1}])$. Cette dernière permet donc de vérifier la correction de cette opération.

5 Générateur de programmes

Pour pouvoir efficacement tester les différentes transformations, nous avons eu besoin d'engendrer un grand nombre de programmes dans le langage *Linear A*. En réalité, nous avons utilisé un langage plus souple, dont nous avons traduit les programmes en *Linear A*. Dans ce langage, il est possible d'effectuer des opérations sur des expressions et non juste sur des variables. Aussi, il ne contient pas de variables linéaires, car ses programmes sont destinés à être testés sur le *Forward mode* et sur le *Reverse mode* (c'est lors de la linéarisation qu'apparaissent les variables linéaires).

Ce générateur prend en argument un budget n (ordre de grandeur de la taille du programme souhaité), ainsi qu'un type t (également généré aléatoirement), et renvoie un programme dont le type de retour est t. À chaque étape, il choisit aléatoirement un nombre, qui le dirige vers une règle de formation d'un programme, et construit récursivement les expressions nécessaires à la bonne formation du programme avec des budgets dont la somme vaut $n - 1$.

Un point délicat est la déclaration et l'utilisation de variables, car il n'est pas possible de savoir à l'avance les variables dont le générateur va avoir besoin. Pour cela, nous générons aléatoirement des variables en les stockant avec leur type, puis lorsque le générateur tente d'appeler une variable, nous regardons s'il en existe une du type souhaité, et si ce n'est pas le cas, nous relançons le générateur.

Nous avons utilisé ce générateur pour tester nos transformations. À chaque étape, nous avons vérifié la bonne formation et le bon typage des progammes obtenus. Il nous a permis de relever plusieurs problèmes dans les transformations, notemment dans la règle *Let* de la transposition, où nous avons compris que l'ordre des variables définies devait obéïr à une certaine convention.

6 *Forward Mode* et *Reverse Mode*

Comme expliqué précédemment, le *Forward mode* est implémenté par la linéarisation. Le *Reverse mode* est, quant à lui, la composition des trois opérations précédentes. Nous avons étudié un second cas de ce dernier sur l'exemple du *Lighthouse* donné dans le livre de Griewank et Walther (page 16) d'une fonction de $\mathbb{R}^4$ dans $\mathbb{R}^2$. Le résultat des transformations succesives est donné en annexe. Bien que ce second exemple soit très gros, il nous a permis de comparer les résultats avec ceux du livre, et ainsi de vérifier manuellement, à défaut de vérification automatique, que notre *Reverse Mode* fonctionne.

7 Conclusion

Nous avons atteint l'objectif d'implémenter les trois opérations : la linéarisation, le unzipping et la transposition, puis de les tester sur des programmes générés aléatoirement. Ainsi, nous avons accès à une implémentation fonctionnelle du *Reverse Mode*.

Ce travail a été fait sur une version très simplifiée du langage *Dex*. Un prolongement serait donc naturellement d'étendre notre implémentation au langage complet qui contient les tableaux, les boucles, les branchements...

Une dernière étape aurait pu être de vérifier automatiquement sur des exemples que les trois opérations sont correctes. Pour la transposition en utilisant la propriété du produit scalaire énoncée précédemment, pour le unzipping, en s'assurant que les deux programmes obtenus renvoient bien les mêmes résultats que le programme initial, et pour la linéarisation, en calculant l'écart entre $f(x+h\dot{x})$ et $y+h\dot{y}$ (où : $f^{lin}(x;\dot{x}) = (y;\dot{y})$) pour différentes valeurs de h et en s'assurant qu'il est petit. Par manque de temps, nous n'avons que pu vérifier manuellement que les résultats étaient bien ceux attendus sur plusieurs exemples.

8 Annexe

```
1 def f(x1, x2, x3, x4 : real, real, real, real) =
2   let (x5 : real) = x3 * x4 in
3   let (x6 : real) = sin(x5)/cos(x5) in
4   let (y1 : real) = (x1 * x6)/(x2 - x6) in
5   let (y2 : real) = (x1 * x2 * x6)/(x2 - x6) in
6   (y1, y2)
```

⇓ *Reverse Mode* (écriture simplifiée)

```
def f'23'nonlin(x1, x2, x3, x4; ) =
  let (x5; ) = (x3 * x4; ) in
  let (v'44; ) = cos(x5) in
  let (v'4; ) = (sin(x5); ) in
  let (v'48; ) = -1.000000 * sin(x5) in
  let (v'3; ) = (cos(x5); ) in
  let (v'54; ) = -1.000000 in
  let (v'60; ) = 1.000000 / (v'3 * v'3) in
  let (x6; ) = (v'4 / v'3; ) in
  let (v'8; ) = (x1 * x6; ) in
  let (v'72; ) = -1.000000 in
  let (v'7; ) = (x2 - x6; ) in
  let (v'77; ) = -1.000000 in
  let (v'83; ) = 1.000000 / (v'7 * v'7) in
  let (y1; ) = (v'8 / v'7; ) in
  let (v'16; ) = (x1 * x2; ) in
  let (v'14; ) = (v'16 * x6; ) in
  let (v'101; ) = -1.000000 in
  let (v'13; ) = (x2 - x6; ) in
  let (v'106; ) = -1.000000 in
  let (v'112; ) = 1.000000 / (v'13 * v'13) in
  let (y2; ) = (v'14 / v'13; ) in
  (y1, y2, x4, x3, v'44, v'48, v'3, v'4, v'54, v'60, x6, x1, v'
    72, v'7, v'8, v'77, v'83, x2, v'16, v'101, v'13, v'14, v'106,
     v'112; )

def f'23'lin'(x4, x3, v'44, v'48, v'3, v'4, v'54, v'60, x6, x1,
   v'72, v'7, v'8, v'77, v'83, x2, v'16, v'101, v'13, v'14, v'
    106, v'112; v'131, v'130) =
  let (; v'132, v'133, v'134, v'135, v'136) = let (; v'137, v'
    138, v'139, v'140, v'141, v'142) = let (; v'143, v'144, v'
    145, v'146, v'147, v'148, v'149) = let (; v'150, v'151, v'
    152, v'153, v'154, v'155) = let (; v'166, v'167, v'168, v'
    169, v'170, v'171, v'172) = let (; v'173, v'174, v'175, v'
    176, v'177, v'178) = let (; v'213, v'214, v'215, v'216, v'
    217, v'218, v'219) = let (; v'220, v'221, v'222, v'223, v'
    224, v'225, v'226, v'227) = let (; v'228, v'229, v'230, v'
    231, v'232, v'233, v'234, v'235, v'236) = let (; v'237, v'
    238, v'239, v'240, v'241, v'242) = let (; v'289, v'290, v'
    291) = let (; v'314, v'315) = let (; v'325, v'326) = let (; v
    '327, v'328) = let (; v'331, v'332) = let (; v'333) = let (;
    v'337) = (; v'130) in
  v'112 *. v'337 in
  dup(v'333) in
  let (; v'342) = v'106 *. v'331 in
  (; v'342, v'332) in
  let (; v'343) = v'14 *. v'327 in
  (; v'343, v'328) in
  let (; v'344) = v'13 *. v'326 in
  (; v'344, v'325) in
  let (; v'316, v'317) = let (; v'318, v'319) = let (; v'320) =
    (; v'315) in
  dup(v'320) in
  let (; v'324) = v'101 *. v'319 in
  (; v'324, v'318) in
  (; v'316, v'317, v'314) in
  let (; v'292, v'293) = let (; v'304, v'305) = let (; v'306, v'
```

```
    307) = let (; v'308) = (; v'291) in
41  dup(v'308) in
42  let (; v'312) = v'16 *. v'306 in
43  (; v'312, v'307) in
44  let (; v'313) = x6 *. v'305 in
45  (; v'313, v'304) in
46  let (; v'294, v'295) = let (; v'296, v'297) = let (; v'298) =
     (; v'292) in
47  dup(v'298) in
48  let (; v'302) = x1 *. v'296 in
49  (; v'302, v'297) in
50  let (; v'303) = x2 *. v'295 in
51  (; v'303, v'294, v'293, v'289, v'290, v'131) in
52  let (; v'243, v'244, v'245) = let (; v'256, v'257) = let (; v'
     267, v'268) = let (; v'269, v'270) = let (; v'273, v'274) =
     let (; v'275) = let (; v'279) = (; v'242) in
53  v'83 *. v'279 in
54  dup(v'275) in
55  let (; v'284) = v'77 *. v'273 in
56  (; v'284, v'274) in
57  let (; v'285) = v'8 *. v'269 in
58  (; v'285, v'270) in
59  let (; v'286) = v'7 *. v'268 in
60  (; v'286, v'267) in
61  let (; v'258, v'259) = let (; v'260, v'261) = let (; v'262) =
     (; v'257) in
62  dup(v'262) in
63  let (; v'266) = v'72 *. v'261 in
64  (; v'266, v'260) in
65  (; v'258, v'259, v'256) in
66  let (; v'246, v'247) = let (; v'248, v'249) = let (; v'250) =
     (; v'245) in
67  dup(v'250) in
68  let (; v'254) = x1 *. v'248 in
69  (; v'254, v'249) in
70  let (; v'255) = x6 *. v'247 in
71  (; v'255, v'246, v'243, v'244, v'237, v'238, v'239, v'240, v'
     241) in
72  let (; v'345) = v'234 +. v'235 in
73  (; v'345, v'228, v'229, v'230, v'231, v'232, v'233, v'236) in
74  let (; v'346) = v'220 +. v'223 in
75  (; v'346, v'221, v'222, v'224, v'225, v'226, v'227) in
76  let (; v'347) = v'213 +. v'215 in
77  (; v'347, v'214, v'216, v'217, v'218, v'219) in
78  let (; v'179, v'180) = let (; v'185, v'186) = let (; v'193, v'
     194) = let (; v'195, v'196) = let (; v'199, v'200) = let (; v
     '201) = let (; v'205) = (; v'173) in
79  v'60 *. v'205 in
80  dup(v'201) in
81  let (; v'210) = v'54 *. v'199 in
82  (; v'210, v'200) in
83  let (; v'211) = v'4 *. v'195 in
84  (; v'211, v'196) in
85  let (; v'212) = v'3 *. v'194 in
86  (; v'212, v'193) in
87  let (; v'187) = let (; v'190) = (; v'186) in
88  v'48 *. v'190 in
89  (; v'187, v'185) in
90  let (; v'181) = let (; v'182) = (; v'180) in
```

```
  v'44 *. v'182 in
  (; v'181, v'179, v'174, v'175, v'176, v'177, v'178) in
  let (; v'348) = v'166 +. v'167 in
  (; v'348, v'168, v'169, v'170, v'171, v'172) in
  let (; v'156, v'157) = let (; v'158, v'159) = let (; v'160) =
   (; v'150) in
  dup(v'160) in
  let (; v'164) = x3 *. v'158 in
  (; v'164, v'159) in
  let (; v'165) = x4 *. v'157 in
  (; v'165, v'156, v'151, v'152, v'153, v'154, v'155) in
  let (; v'349) = v'148 +. v'149 in
  (; v'349, v'143, v'144, v'145, v'146, v'147) in
  let (; v'350) = v'137 +. v'141 in
  (; v'350, v'138, v'139, v'140, v'142) in
  let (; v'351) = v'135 +. v'136 in
  (; v'351, v'132, v'133, v'134)
```