La Différentiation Automatique dans Eva-Flo

Laurent Hascoët

June 20, 2007

1 Quelles dérivées pour l'évaluation d'erreurs?

On suppose qu'on a un "bout de code à étudier", qui implémente une fonction $f : x \in I\!R^m \mapsto y \in I\!R^{n=1}$. Très probablement n vaut 1 et en tout cas $n << m$.

Pour les besoins de l'évaluation d'erreurs, on veut évaluer $f(X)$, l'intervalle vrai dans lequel se trouve $f(x)$ quand $x \in X$ où X est un multi-intervalle petit donné autour de x_0.

L'expansion de Taylor et le théorème de la valeur intermédiaire nous disent que:

$$\forall x_2 \in X, f(x_2) = f(x_0) + f'(\xi \in X) \times (x_2 - x_0)$$

et donc que, en passant aux intervalles, on obtient la surestimation suivante de $f(X)$:

$$f(X) \subset f(x_0) + (X - x_0) \times f'(X)$$

où les notations X, $f(X)$, $X-x_0$, $f'(X)$ désignent des intervalles. Au deuxième ordre, on obtient la surestimation suivante, encore meilleure que la précédente:

$$f(X) \subset f(x_0) + (X - x_0) \times f'(x_0) + \frac{(X - x_0)^2}{2} \times f''(X)$$

où à nouveau, X, $f(X)$, $X - x_0$ désignent des intervalles, alors que $(X - x_0)^2$, et $f''(X)$ désignent des quoi?.

Mais en réalité, on veut des intervalles exacts garantis, et donc même les calculs de $f(x)$ etc doivent être effectués sur des intervalles du genre $[x] \doteqdot [x - ulp, x + ulp]$. On obtient donc au premier ordre:

$$f(X) \subset f([x_0]) + (X - [x_0]) \times f'(X)$$

avec des intervalles partout !

2 Qu'attend-on de l'outil de DA?

Qu'attend-on comme interaction entre le noyau central d'Eva-Flo et le peripherique de Différentiation Automatique?

Dans le noyau central, on a une représentation du code en XML (pour faciliter les transferts avec d'autres modules). Plus particulièrement, cette représentation semble associer des lambda-expressions et des affectations, avec la restriction supplémentaire qu'on ne peut pas écraser la valeur d'une variable (Single Assignment). On pense donc qu'on pourra facilement transférer ce code vers Tapenade, sous la forme d'un petit sous-programme simple. Exemple:

```
x,y ↦ r
--------------
a = x*x
b = y*y
r = 2.0*a + b**2.0 + x
```

Comme on a deux entrées et qu'on veut le gradient $\left(\frac{\partial r}{\partial x}, \frac{\partial r}{\partial y}\right)$ on préfère le mode inverse. De plus comme on fait l'hypothèse SSA, on n'a pas besoin de se préoccuper de sauvegarder les valeurs intermédiaires écrasées. Sur l'exemple, on attend le code suivant (ce code est simplifié par rapport à ce que produit Tapenade: il y avait une variable d'entrée supplémentaire $\overline{\mathtt{r}}$ qui par définition vaut 1.0 . On l'a simplifiée.)

```
x,y ↦ x̄, ȳ
--------------
a = x*x
b = y*y
r = 2*a + b**2 + x
x̄ = 1.0
b̄ = 2*b
ā = 2.0
r̄ = 0.0
ȳ = 2.0*y*b̄
x̄ = x̄ + 2.0*x*ā
```

On peut le simplifier encore en détectant que la première instruction du code d'origine, ainsi que la dernière, sont en fait inutiles pour le code différentié (`a` et `r` n'interviennent pas dans les dérivées). Aussi, la valeur finale de $\overline{\mathtt{r}}$ est inutile. Tapenade effectue ces simplifications en standard. On arrive à:

```
x,y ↦ x̄, ȳ
--------------
b = y*y
x̄ = 1.0
b̄ = 2*b
ā = 2.0
ȳ = 2.0*y*b̄
x̄ = x̄ + 2.0*x*ā
```

On peut imaginer que l'outil propage statiquement l'information supplémentaire que `x` et `y` sont des intervalles. Ca revient à faire de la surcharge d'opérateurs, mais statiquement, à compile-time, ce qui donne un code bien plus efficace. En revanche les constantes immédiates `1.0` et `2.0` sont exactes. On va noter ici les intervalles et les opérations sur les intervalles avec un chapeau comme dans $\hat{x}$ ou $\hat{+}$. On obtient sur l'exemple:

$$\begin{array}{l}
\hat{\mathtt{x}},\hat{\mathtt{y}} \mapsto \hat{\bar{\mathtt{x}}},\ \hat{\bar{\mathtt{y}}} \\
\texttt{--------------} \\
\hat{\mathtt{b}} \mathrel{\hat{=}} \hat{\mathtt{y}} \mathbin{\hat{*}} \hat{\mathtt{y}} \\
\bar{\mathtt{x}} = 1.0 \\
\hat{\bar{\mathtt{b}}} \mathrel{\hat{=}} 2.0 \mathbin{\hat{*}} \hat{\mathtt{b}} \\
\bar{\mathtt{a}} = 2.0 \\
\hat{\bar{\mathtt{y}}} \mathrel{\hat{=}} 2.0 \mathbin{\hat{*}} \hat{\mathtt{y}} \mathbin{\hat{*}} \hat{\bar{\mathtt{b}}} \\
\hat{\bar{\mathtt{x}}} \mathrel{\hat{=}} \bar{\mathtt{x}} \mathbin{\hat{+}} 2.0 * \bar{\mathtt{a}} \mathbin{\hat{*}} \hat{\mathtt{x}}
\end{array}$$

L'étape suivante est de remettre ce programme différentié sous une forme de lambda-expression (et SSA) pour le renvoyer vers le noyau d'Eva-Flo. Pour passer en SSA, on introduit le clone $\overline{\mathtt{x1}}$ de $\bar{\mathtt{x}}$. Du point de vue notation, toute variable $\overline{x}$ est dorénavant notée x`b`. Cela pourrait donner:

$$\begin{array}{l}
\lambda(\hat{\mathtt{x}},\hat{\mathtt{y}})\ . \\
\quad \mathtt{let}\ \hat{\mathtt{b}} \mathrel{\hat{=}} \hat{\mathtt{y}} \mathbin{\hat{*}} \hat{\mathtt{y}}\ \mathtt{in} \\
\quad\quad \mathtt{let\ xb1\ =\ 1.0\ in} \\
\quad\quad\quad \mathtt{let}\ \widehat{\mathtt{bb}} \mathrel{\hat{=}} 2.0 \mathbin{\hat{*}} \hat{\mathtt{b}}\ \mathtt{in} \\
\quad\quad\quad\quad \mathtt{let\ ab\ =\ 2.0\ in} \\
\quad\quad\quad\quad\quad \mathtt{let}\ \widehat{\mathtt{yb}} \mathrel{\hat{=}} 2.0 \mathbin{\hat{*}} \hat{\mathtt{y}} \mathbin{\hat{*}} \widehat{\mathtt{bb}}\ \mathtt{in} \\
\quad\quad\quad\quad\quad\quad \mathtt{let}\ \widehat{\mathtt{xb}} \mathrel{\hat{=}} \mathtt{xb1} \mathbin{\hat{+}} 2.0 * \mathtt{ab} \mathbin{\hat{*}} \hat{\mathtt{x}}\ \mathtt{in} \\
\quad\quad\quad\quad\quad\quad\quad (\widehat{\mathtt{xb}},\ \widehat{\mathtt{yb}})
\end{array}$$

On peut aussi simplifier les `let` utilisés une seule fois. En fait sur l'exemple courant, cela s'applique à tous les `let`. Cela montre seulement que l'exemple est trop simple. En général certains `let` devront rester. (Pour l'anecdote, c'est seulement sur des exemple plus gros, sur lesquels les `let` ne peuvent pas tous se simplifier, que l'on pourra vraiment montrer la supériorité du mode inverse sur la différentiation tangente)

Après simplification sur l'exemple, la lambda-expression différentiée devient:

$$\lambda(\hat{\mathtt{x}}\ ,\ \hat{\mathtt{y}})\ .\ \ (4.0 \mathbin{\hat{*}} \hat{\mathtt{x}} \mathbin{\hat{+}} 1.0\ ,\ 2.0 \mathbin{\hat{*}} \hat{\mathtt{y}} \mathbin{\hat{*}} (2.0 \mathbin{\hat{*}} \hat{\mathtt{y}} \mathbin{\hat{*}} \hat{\mathtt{y}}))$$

Cela soulève aussi la question du réordonnancement optimal des opérations pour maximiser les opérations exactes entre scalaires (non-intervalles). Il semble plus malin de faire cela avant le passage aux lambda. En tout cas on pourrait chercher à obtenir:

$$\lambda(\hat{\mathtt{x}}\ ,\ \hat{\mathtt{y}})\ .\ \ (4.0 \mathbin{\hat{*}} \hat{\mathtt{x}} \mathbin{\hat{+}} 1.0\ ,\ 4.0 \mathbin{\hat{*}} \hat{\mathtt{y}} \mathbin{\hat{*}} \hat{\mathtt{y}} \mathbin{\hat{*}} \hat{\mathtt{y}})$$