Habilitation à Diriger des Recherches

Université de Nice - Sophia Antipolis

Présentée par **Christèle Faure**

le 22 mars 2002

Titre

Dérivation de Programme Assistée par Ordinateur
Tome I : Synthèse des travaux

Jury

Christian Bischof
Alain Deutsch
André Galligo
Andreas Griewank (Rapporteur)
François Irigoin (Rapporteur)
Gilles Kahn
Mohamed Masmoudi
Olivier Pironneau (Rapporteur)

Résumé

Ce manuscrit décrit les travaux de recherches que j'ai effectués dans le domaine de la dérivation de programme. Appliquée à la main, la dérivation de programme est un travail de programmation long, fastidieux, et difficilement exempt d'erreur. La Dérivation de Programme Assistée par Ordinateur (DAO) aussi appelée différentiation automatique a pour but d'automatiser ce processus : produire automatiquement un programme qui calcule des dérivées exactes (aux erreurs d'arrondi près) à partir d'une fonction représentée par un programme informatique. Ce manuscrit décrit de façon synthétique mes travaux en les replaçant dans des axes de recherche que j'ai cherché à rendre indépendants, après une introduction donnant les bases minimales nécessaires à la compréhension du reste du document. Pour rendre compte de tous les travaux de recherche en DAO, j'ai développé un modèle de programme plus général que les modèles classiques utilisés. De plus, j'ai unifié les modes de dérivation (direct, inverse), les différentes stratégies de dérivation, les différentes forme d'implémentation de la DAO dans une théorie globale pour permettre une décomposition de la DAO en trois problèmes indépendants : l'obtention des matrices Jacobienne locales à partir du programme original et l'évaluation de la complexité de cette tâche, la composition de ces matrices par l'intermédiaire de l'application d'algorithmes de dérivations et l'évaluation de la complexité pratique de ces algorithmes, et enfin l'optimisation du programme résultat. Je présente finalement la spécification d'une plate-forme permettant le développement rapide d'outils plus efficaces de dérivation de programme et le décris succintement des sujets de recherches associés à la DAO mais pouvant être étudiés indépendamment.

Mots clés : différentiation automatique, dérivation de programme, transformation de programme, calcul de dérivées, dérivées exactes, compilation.

Abstract

This manuscript describes my research in the area of differentiation of programme. Applied by hand differentiation of program is painful, and error prone. Computational or Automatic Differentiation aims at automating this process: automatically generating the program that computes the exact derivative of the function implemented within a program. This manuscript introduces first some notions on Automatic Differentiation necessary to the understanding of the rest of the documents, and thereafter synthetically describes my work putting them in a general framework where research axes are as independent as possible. First, I developed a new model of program that enables the description of all differentiation new results. Then, I unify all the differentiation modes, the differentiation strategies and diverse kind of implementation of Automatic Differentiation in a general theory. From this I extract the three main problems encountered in differentiation of program: computation of all local Jacobian matrices from the original program and evaluation of its complexity, composition of all these matrices using derivation algorithms and evaluation of the complexity of these algorithms, and then the optimization of the generated program and the evaluation of its practical complexity. Finally a general and open platform is presented, and some interesting research subjects are sketched.

Keywords: automatic differentiation of program, program transformation, exact derivatives, computation of derivatives, compilation.

Table des matières

Première partie

Dossier d'habilitation

Chapitre 1

Introduction

Mes travaux portent sur deux problèmes de recherche différents : le Calcul Formel Multi-valué et la Dérivation de Programme Assistée par Ordinateur.
Durant la période 1989-1994 (thèse 1989-1992 et stage post-doctoral 1992-1994) j'ai défini les bases d'un outil de calcul formel dont les résultats contiendraient leur domaine de validité. Les systèmes de calcul formel standards donnent des réponses génériques aux problèmes qui leur sont posés. Ces résultats génériques (par exemple $1/x$) sont valides sauf sur un ensemble fini de valeurs de leurs paramètres ($x = 0$). Pour pouvoir effectuer des calculs complexes en toute sécurité, il faudrait que les systèmes fournissent non plus le résultat générique, mais tous les résultats possibles associés à leurs domaines de validité. Mon travail de thèse et de stage post-doctoral a consisté à définir une forme d'expression générale appelée expression multi-valuée qui permette d'exprimer ces résultats complexes. J'ai défini une arithmétique sur ces expressions offrant à la fois sécurité et puissance de calcul puisque toutes les branches de calcul sont explorées à moins d'une demande explicite de l'utilisateur qui apparaît dans le résultat final. Ces travaux ont été poursuivis après 1992 dans l'équipe SAFIR-CAFÉ de l'INRIA [DGH96] et par James Davenport à l'Université de Bath. Cette partie de mes travaux est décrite brièvement en fin de ce document dans le Chapitre 5 (page 85).
Dans ce document, j'approfondis la deuxième partie de mes travaux de recherche (1994-2001) qui concerne la Différentiation Automatique de Programme.

1.1 Dérivation de Programme Assistée par Ordinateur

Les simulations numériques de phénomènes physiques ou économiques comportent presque toujours des phases d'assimilation de données, d'estimation de paramètres, d'optimisation de formes ou de calculs de sensibilité,

qui toutes nécessitent l'évaluation de dérivées partielles d'une ou plusieurs fonctions coût. Si ces fonctions étaient disponibles sous une forme analytique, il serait possible d'utiliser des outils de calcul symbolique (ou Calcul Formel) pour les dériver. Mais elles sont accessibles par l'évaluation d'un programme décrivant un modèle discrétisé du phénomène étudié, ce qui impose la dérivation du programme.

Trois méthodes sont couramment utilisées pour calculer la valeur de la dérivée d'un programme : les différences finies, la méthode de l'adjoint discret, la dérivation de programme.
La méthode des différences finies est conceptuellement la plus simple, mais conduit aux approximations les plus grossières. L'obtention d'approximations fines se fait au prix de nombreux essais si le contrôle de la qualité des dérivées est effectué à la main ou de calculs très complexes si ce contrôle est automatisé. Si les dérivées sont le résultat final recherché (pour une étude de sensibilité par exemple) le contrôle de la qualité des dérivées peut être très fin. Par contre si ce sont des résultats intermédiaires utilisés par un programme (assimilation de données, optimisation), le coût en temps de calcul de cette précision apparaît exorbitant.

La méthode dite de l'adjoint discret [Tal91] consiste à discrétiser un problème dont les inconnues sont les dérivées cherchées, elle donne de bons résultats mais n'est applicable que dans les cas où le modèle original est mathématiquement adéquat, elle ne permet donc pas de traiter tous les problèmes. D'autre part elle fait parfois apparaître une incohérence entre problèmes original (direct) et adjoint, qui entraîne des instabilités numériques. Ces problèmes dus à la non commutation de la discrétisation par rapport à la dérivation ont plus ou moins d'importance suivant les applications, mais apparaissent fréquemment.

La dérivation de programme apporte l'exactitude des dérivées par rapport à l'approximation obtenue par différences finies, et la cohérence entre programme original et programme dérivé par rapport à l'adjoint discret. Appliquée à la main, la dérivation de programme est un travail de programmation long, fastidieux, et difficilement exempt d'erreurs. La Différentiation Automatique de programme a pour but d'automatiser ce processus : produire automatiquement un programme qui calcule des dérivées exactes (aux erreurs d'arrondi près) à partir d'une fonction représentée par un programme informatique. L'automatisation permet d'effectuer cette transformation fastidieuse de façon sûre et rapide pour chaque nouvelle version du programme original. La Différentiation Automatique existe depuis les années 80 comme discipline académique, mais n'a toujours pas de contours clairs. Ceci peut être expliqué par sa forte inter-disciplinarité, ainsi que par la difficulté à développer des outils permettant de démontrer son efficacité sur des programmes de taille industrielle.
Les documents de synthèse présentant le domaine sont peu nombreux : les trois proceedings [GC91, BBCG96, CFG$^+$01] de la conférence intitulée Inter-

national Workshop on Computational Differentiation et le livre d'Andreas Griewank [Gri00].

L'interdisciplinarité de ce domaine s'explique schématiquement par le fait que les besoins en types de programmes dérivés sont définis par des numériciens et des ingénieurs, alors que les moyens sont offerts par des informaticiens ou des mathématiciens. Les axes de recherche autour desquels cette discipline est structurée démontrent cette interdisciplinarité :

1. L'étude de méthodes numériques nouvelle et l'amélioration des méthodes actuelles grâce à une meilleure utilisation de dérivées de programmes, et la définition des types de programmes dérivés ainsi que de leurs contraintes de performance associées.
2. La définition et l'étude de stratégies de dérivation de plus en plus efficaces (en temps et en espace) sur des modèles mathématiques de programmes de plus en plus proches des programmes réels,
3. Le développement d'outils informatiques complexes impliquant à la fois des techniques proches de la compilation et des méthodes propres à la dérivation de programme.

La Différentiation Automatique ayant pour visée pratique la production automatisée d'un programme dérivé (qui calcule les dérivées), le développement d'outils est fondamental. Or celui-ci est très rapide tant qu'il s'agit de traiter des cas d'école, mais devient aussi complexe que le développement d'un compilateur pour traiter des problèmes réels. Les principaux outils de Différentiation Automatique qui ont été développés (`ADIFOR [BCC+92, BCK+98]`, `ADIC [BRM97]`, `ADOL-C [GJM+96]`, `Odyssée [FP98]`, `TAMC [Gie97]` or `TAF [Fas00]`, `PADRE2 [Kub96]`) ont atteint la maturité suffisante pour traiter des programmes de taille et de complexité industrielle. Mais sur ces programmes réels, des problèmes de passage à l'échelle des programmes dérivés peuvent apparaître. La Différentiation Automatique appliquée à des composants du programme original, associée à une structuration manuelle du programme dérivé, reste parfois le seul moyen pour obtenir un programme dérivé efficace. L'automatisation totale du calcul des dérivées d'un programme industriel, en particulier en mode inverse, reste hors de portée des outils de Différentiation Automatique actuels.

Pour toutes ces raisons, je préfère utiliser le terme **Dérivation de Programme Assistée par Ordinateur** à la place du terme consacré de Différentiation Automatique. DAO rend mieux compte à la fois des différents domaines de compétences nécessaires à son développement et de la composition de phases manuelles et automatiques nécessaires à son utilisation dans certains cas.

Ma contribution se décompose suivant les trois axes de recherche de la DAO.
J'ai établi des collaborations scientifiques ou industrielles avec des utilisateurs potentiels de la DAO. Ces contacts m'ont permis d'identifier les besoins

réels des utilisateurs en terme de dérivées tels que

- des dérivées directionnelles calculées $Jdx_1 \cdots Jdx_n$,
- des gradients $dx_1^\star J^\star \cdots dx_n^\star J^\star$,
- des dérivées d'ordre supérieurs : Hessien, développement de Taylor, etc.

mais aussi leurs contraintes : la facilité d'utilisation, la limitation en temps et/ou en mémoire. J'ai aussi étudié l'apport de dérivées spécialisées pour les processus itératifs ou la composition de dérivées directionnelles et de gradients sur des programmes industriels particuliers (Dassault, Alenia, Essilor). De nombreuses méthodes numériques ont été développées sous l'hypothèse que les dérivées étaient peu fiables puisque seulement des approximations ou coûteuses en temps de développement pour avoir les dérivées exactes. C'est pour cela que les dérivées sont en fait assez peu utilisées dans les méthodes numériques. Il est très important de re-visiter ces méthodes sous l'hypothèse que la DAO fourni des dérivées exactes sans sur-coût en temps de développement. Ma contribution dans cette axe est plus de l'ordre de la communication que de la réalisation. Je ne la présente donc pas directement, mais décris les travaux de recherche qu'elle a entraîné dans les deux autres axes.

J'ai unifié les modes de dérivation direct et inverse, les différentes stratégies de mode inverse ainsi que les deux types implémentations de la DAO. En effet, tant que cette unification n'est pas réalisée, il est difficile d'isoler des problématiques assez clairement pour pouvoir attirer des chercheurs d'autres disciplines.
D'autre part, j'ai développé l'outil `Odyssée [FP98]` en ajoutant au noyau de dérivation, les fonctionalités permettant le traitement interprocédural, de nouveaux algorithmes en mode inverse et des algorithmes d'optimisation des programmes dérivés. De cette manière, `Odyssée [FP98]` a pu être appliqué à des programmes de taille et de complexité industrielle par des non-spécialistes.

1.2 Dérivée d'un programme

Cette section présente les bases de la DAO nécessaires à la compréhension du reste de ce document. La dérivation de programme a été définie sur un modèle de programme simplifié. Sur ce modèle, les deux modes de DAO ont été démontrés et la complexité théorique en nombre d'opérations qui fonde la DAO a pu être évaluée.

1.2.1 Modèle de programme

La dérivation de programme a été introduite théoriquement dans des travaux déjà anciens [OVB71, Spe80, Iri84, KNC84, Saw84, IK87]. Ceux-ci sont basés sur un modèle de programme numérique simple appelé *modèle*

scalaire. Dans ce modèle, un programme est une séquence d'affectations de variables réelles (*straight-line program, SLP*) dont la valeur n'est calculée qu'une seule fois (*single assignment program, SAP*).

On peut généraliser ce modèle à un *modèle vectoriel* pour se rapprocher des programmes réels. Dans le modèle vectoriel, un programme est composé d'une séquence d'affectations qui peuvent chacune modifier plusieurs variables (scalaires) réelles pour permettre la prise en compte des appels de sous-programmes. C'est le modèle vectoriel que nous utilisons dans le reste de cette section.

Considérons un programme scalaire qui utilise N variables scalaires dont n sont des variables d'entrées et p sont des variables de sortie. Pour le représenter suivant le modèle vectoriel, il suffit de grouper les N variables scalaires dans un vecteur et de dupliquer le vecteur à chaque étape du calcul.

Chaque instruction originale a_i correspond à une fonction e_i

$$e_i \ : \mathbb{R}^{n_i} \longrightarrow \mathbb{R}^{p_i},$$

qui peut être étendue à une fonction f_i de $\mathbb{R}^N$ dans $\mathbb{R}^N$ par une composition avec l'injection $\mathcal{I}_{p_i N}$ et la projection $\mathcal{S}_{N n_i}$ canoniques.

$$f_i \ : \mathbb{R}^N \overset{\mathcal{S}_{N n_i}}{\hookrightarrow} \mathbb{R}^{n_i} \xrightarrow{e_i} \mathbb{R}^{p_i} \overset{\mathcal{I}_{p_i N}}{\hookrightarrow} \mathbb{R}^N$$

Un programme de longueur m dont chaque affectation calcule une fonction f_i calcule alors la fonction composée f suivante :

$$f \ : \mathbb{R}^n \overset{\mathcal{I}_{nN}}{\hookrightarrow} \mathbb{R}^N \xrightarrow{f_1} \dots \xrightarrow{f_m} \mathbb{R}^N \overset{\mathcal{S}_{Np}}{\hookrightarrow} \mathbb{R}^p \tag{1.1}$$

où $\mathcal{I}_{nN}$ et $\mathcal{S}_{Np}$ désignent l'injection et la projection canoniques.

Soient $x \in \mathbb{R}^n$ le vecteur d'entrée, $y \in \mathbb{R}^p$ le vecteur de sortie et $\{x_i \in \mathbb{R}^N, i = 1 \cdots m\}$ les vecteurs intermédiaires, le programme P qui calcule f s'écrit de la manière suivante :

$$\begin{array}{rcccll} 0 & : & x_0 & = & \mathcal{I}_{nN}(x) & \\ i & : & x_i & = & f_i(x_{i-1}) & i = 1 \cdots m \\ m+1 & : & y & = & \mathcal{S}_{Np}(x_m) & \end{array}$$

La Figure 1.1 montre la correspondance entre programme vectoriel et fonction mathématique sur un exemple simple. La fonction mathématique f composée de deux fonctions élémentaires f_1 et f_2 est associée à un programme élémentaire P constitué de deux instructions exécutées en séquence.

1.2.2 Programme dérivé

Considérons un programme P représentant la fonction f décrite dans l'Identité 1.1. Chaque fonction f_i modifie un petit nombre p_i de variables,

$$\begin{array}{rcl} \mathbb{R}^2 & \rightarrow & \mathbb{R} \\ (x,y) & \rightarrow & x * y^2 \end{array}$$

(a) e_1

$$\begin{array}{rcl} \mathbb{R}^2 & \rightarrow & \mathbb{R} \\ (y,z) & \rightarrow & y * z^2 \end{array}$$

(b) e_2

$$\begin{array}{rcl} \mathbb{R}^4 & \rightarrow & \mathbb{R}^4 \\ (x,y,z,w) & \rightarrow & (x,y,x * y^2,w) \end{array}$$

(c) f_1

$$\begin{array}{rcl} \mathbb{R}^4 & \rightarrow & \mathbb{R}^4 \\ (x,y,z,w) & \rightarrow & (x,y,z,y * z^2) \end{array}$$

(d) f_2

```
Z = X * Y**2
W = Z**2 * Y
```

(e) P

$$f = f_2 \cdot f_1$$

(f)

FIG. 1.1 – *Programme vu comme une composition de fonctions*

sa matrice Jacobienne totale Df_i (formée par toutes les dérivées partielles) se calcule donc facilement en utilisant les règles élémentaires de dérivations d'expression, telle que "la dérivée d'une somme est la somme des dérivées".

On appelle *programme dérivé* du programme P, le programme P' qui calcule les instructions dérivées. L'application de la règle de dérivation des fonctions composées à l'Identité 1.1 permet d'obtenir l'Identité 1.2.

$$Df(x) = D\mathcal{S}_{Np}(x_m) \cdot Df_m(x_{m-1}) \cdots Df_1(x_0) \cdot D\mathcal{I}_{nN}(x) \tag{1.2}$$

Notons * l'opérateur de transposition, en transposant Identité 1.2 on obtient l'Identité 1.3.

$$Df^*(x) = D\mathcal{I}^*_{nN}(x) \cdot Df^*_1(x_0) \cdots Df^*_m(x_{m-1}) \cdot D\mathcal{S}^*_{Np}(x_m) \tag{1.3}$$

Les identités 1.2 et 1.3 définissent respectivement le *mode direct* et le *mode inverse* de DAO. L'injection $\mathcal{I}_{nN}$ et la projection $\mathcal{S}_{Np}$ canoniques sont des opérateurs linéaires tels que $\mathcal{I}_{nN}\mathcal{S}_{Nn} = \mathbf{1}$ donc $D\mathcal{I}_{nN} = \mathcal{I}_{nN}$ $D\mathcal{S}_{Np} = \mathcal{S}_{Np}$ et $D\mathcal{S}_{Np}{}^\star = \mathcal{I}_{p,N}$ et $D\mathcal{I}_{nN}{}^\star = \mathcal{S}_{N,n}$. Par la suite je simplifierai les équations en utilisant les opérateurs dérivés.

Mode direct

Soient x,y,$x_1, \cdots x_m$ définis comme précédemment et soient $dx \in \mathbb{R}^n$ la direction d'entrée, $dy \in \mathbb{R}^p$ la direction de sortie et $\{dx_i \in \mathbb{R}^N, i = 1 \cdots m\}$ les directions intermédiaires. Le programme dérivé P' qui calcule une dérivée directionnelle de P dans la direction dx en *mode direct* s'écrit de façon

immédiate comme suit :

$$\begin{array}{rcrcll} 0 & : & dx_0 & = & D\mathcal{I}_{nN}(x)dx & \\ i & : & dx_i & = & Df_i(x_{i-1})dx_{i-1} & i = 1 \dots m \\ m+1 & : & dy & = & D\mathcal{S}_{Np}(x_m)dx_m & \end{array}$$

A chaque étape i du calcul, le programme calcule la nouvelle direction dx_i en faisant le produit de la direction dx_{i-1} par la matrice Jacobienne locale $Df_i(x_{i-1})$. Cette nouvelle direction sera elle aussi propagée en avant vers l'entrée de l'instruction suivante $i+1$ et ainsi de suite. Il faut noter que l'ordre d'évaluation original est conservé par la dérivation et que l'évaluation de l'instruction i nécessite la valeur $Df_i(x_{i-1})$ qui est calculée à partir de la valeur originale x_{i-1}. Le calcul original et le calcul dérivé peuvent donc être combinés au niveau de l'instruction.

Le programme P' qui fournit au vol les valeurs originales nécessaires à l'évaluation des matrices Jacobienne locales et propage la dérivée directionnelle est présenté dans la Figure 1.2. Cet algorithme appelé *linéaire tangent* est implanté dans tous les outils de dérivation par transformation de programme.

On peut noter que ce calcul sur une direction dx s'étend facilement au calcul de k directions indépendantes $dx^1, \cdots dx^k$. Cet algorithme appelé *multi-tangent parallèle* est présenté dans la Figure 1.3. Dans ce programme dérivé, le calcul de chaque variable originale x_i, ainsi que celui de chaque Jacobienne locale Df_i est effectué une seule fois. Cet algorithme est implanté dans `ADIFOR [BCC+92, BCK+98]` et l'ensemble des directions $\{dx^1 \cdots dx^k\}$ est alors appelé *seed matrix*.

De même, le calcul de k directions dépendant les unes des autres peut être réalisé (voir la Figure 1.4) en ne calculant les valeurs originales et les valeurs des matrices Jacobienne locales qu'une seule fois en utilisant l'algorithme appelé *multi-tangent séquentiel*. Il n'est pas nécessaire de conserver les valeurs des x_i, mais les valeurs des $Df_i(x_{i-1})$. Cet algorithme n'a été implanté dans aucun outil de DAO.

Si les valeurs des matrices Jacobiennes ne peuvent pas être conservées (tracées), elles peuvent être recalculées au vol à partir des valeurs originales. Si les valeurs originales non plus ne peuvent pas être conservées, elles peuvent être re-calculées. Ceci donne lieu à de nombreux compromis qui seront développés ultérieurement.

Mode inverse

Soient x,y,$x_1, \cdots x_m$ définis comme précédemment et soient $dx^* \in \mathbb{R}^n$ le vecteur adjoint d'entrée, $dy^* \in \mathbb{R}^p$ le vecteur adjoint de sortie, et $\{dx_i^* \in \mathbb{R}^N, i = 1 \cdots m\}$ les vecteurs intermédiaires. Le programme adjoint qui calcule le gradient de P obtenu en appliquant l'Identité 1.3 s'écrit de façon

$$
\begin{array}{lcll}
(x_0,dx_0) & = & (\mathcal{I}_{nN}(x),\mathcal{I}_{nN}(dx)) & \\
(x_i,dx_i) & = & (f_i(x_{i-1}),Df_i(x_{i-1})dx_{i-1}) & i = 1 \dots m \\
(y,dy) & = & (\mathcal{S}_{Np}(x_m),\mathcal{S}_{Np}(dx_m)) &
\end{array}
$$

FIG. 1.2 – *Programme linéaire tangent*

$$
\begin{array}{lcll}
(x_0,dx_0^1,\cdots dx_0^k) & = & (\mathcal{I}_{nN}(x),\mathcal{I}_{nN}(dx^1),\dots,\mathcal{I}_{nN}(dx^k)) & \\
J_i & = & Df_i(x_{i-1}) & \rceil \quad i = 1 \dots m \\
(x_i,dx_i^1,\cdots dx_i^k) & = & (f_i(x_{i-1}),J_i dx_{i-1},\dots,J_i dx_{i-1}^k) & \rfloor \\
(y,dy^1,\cdots,dx^k) & = & (\mathcal{S}_{Np}(x_m),\mathcal{S}_{Np}(dx_m^1),\cdots,\mathcal{S}_{Np}(dx_m^k)) &
\end{array}
$$

FIG. 1.3 – *Programme linéaire multi-tangent parallèle*

$$
\begin{array}{lcll}
x_0 & = & \mathcal{I}_{nN}(x) & \\
J_i & = & Df_i(x_{i-1}) & \rceil \quad i = 1 \dots m \\
x_i & = & f_i(x_{i-1}) & \rfloor \\
y & = & \mathcal{S}_{Np}(x_m) &
\end{array}
$$

$$
\begin{array}{lcll}
dx_0^1 & = & \mathcal{I}_{nN}(dx^1) & \\
dx_i^1 & = & J_i dx_{i-1}^1 & i = 1 \dots m \\
dy^1 & = & \mathcal{S}_{Np}(dx_m^1) & \\
 & \vdots & & \\
dx_0^k & = & \mathcal{I}_{nN}(dx^k) & \\
dx_i^k & = & J_i dx_{i-1}^k & i = 1 \dots m \\
dy^k & = & \mathcal{S}_{Np}(dx_m^k) &
\end{array}
$$

FIG. 1.4 – *Programme linéaire multi-tangent séquentiel*

immédiate comme suit :

$$\begin{array}{lcll} 0 & : \; dx_m^* & = & D\mathcal{S}_{Np}{}^\star(x_m)dy^* \\ m-i+1 & : \; dx_{i-1}^* & = & Df_i^\star(x_{i-1})dx_i^* \quad i = m \ldots 1 \\ m+1 & : \; dx^* & = & D\mathcal{I}_{nN}{}^\star(x_0)dx_0^* \end{array}$$

La comparaison de cette séquence d'instructions et de celle du programme original, montre que la valeur du vecteur x_{i-1} utilisée à l'instruction i dans le programme original est nécessaire à l'instruction $m-i+1$ du programme dérivé. Les valeurs x_i calculées de 1 à m dans le programme original sont donc utilisées de m à 1 pour l'évaluation des Jacobiennes locales. Un pré-calcul de toutes les valeurs x_i combiné avec un calcul au vol des matrices Jacobiennes locales est la méthode la plus simple pour implémenter ce calcul. Cet algorithme appelé *linéaire cotangent* est utilisé par `Odyssée [FP98]`, `TAMC [Gie97]` et `ADOL-C [GJM+96]`. Comme en mode direct, les calculs *parallèles* ou *séquentiels* de k gradients remplacent le re-calcul des valeurs originales $x_1, \cdots x_m$ et des matrices Jacobiennes locales par la conservation de leurs valeurs comme le montre respectivement les Figures 1.6 et 1.7. La remarque sur le remplacement de la mémorisation par du calcul vaut ici comme pour le mode direct.

Choix des dérivées

Nous avons considéré les programmes dérivés qui calculent la matrice Jacobienne $\frac{\partial y}{\partial x}$ du vecteur des variables de sortie y par rapport à celui des variables d'entrée x du programme. Or en général les utilisateurs veulent la sensibilité d'un sous-ensemble de variables de sorties par rapport à un paramètre donné ou le gradient d'une variable de sortie par rapport à un ensemble de paramètres particuliers. On appelle *variable indépendante* toute variable de sortie dont une dérivée est souhaitée, et *variable dépendante* toute variable d'entrée par rapport à laquelle la dérivée est calculée. Pour choisir un sous-ensemble des dérivées partielles calculées, il faut sélectionner ν éléments du vecteur dérivé d'entrée et π éléments du vecteur dérivé de sortie. Pour réaliser ce choix théoriquement, il suffit d'ajouter une fonction d'injection sur le vecteur dérivé d'entrée et une fonction de projection sur le vecteur dérivé de sortie comme présenté en mode direct dans l'Identité 1.4 et en mode inverse dans l'Identité 1.5.

$$Df(x) = \mathcal{S}_{p\pi} \cdot \mathcal{S}_{Np} \cdot Df_m(x_{m-1}) \cdot \cdots \cdot Df_1(x_0) \cdot \mathcal{I}_{nN} \cdot \mathcal{I}_{\nu n} \tag{1.4}$$

$$Df^\star(x) = \mathcal{S}_{n\nu} \cdot \mathcal{S}_{Nn} \cdot Df_1^\star(x_0) \cdot \cdots \cdot Df_m^\star(x_{m-1}) \cdot \mathcal{I}_{pN} \cdot \mathcal{I}_{\pi p} \tag{1.5}$$

Il est clair que le calcul de la matrice Jacobienne complète est alors inutile puisque certaines composantes seront "oubliées" lors du produit matrice/vecteur. Nous verrons par la suite qu'il existe un moyen plus astucieux

$$
\begin{array}{lcll}
x_0 & = & \mathcal{I}_{nN}(x) & \\
x_i & = & f_i(x_{i-1}) & i = 1 \dots m \\
y & = & \mathcal{S}_{Np}(x_m) & \\
 & & & \\
dx^*_m & = & \mathcal{I}_{pN}(dy^*) & \\
dx^*_{i-1} & = & Df^\star_i(x_{i-1}) dx^*_i & i = m \dots 1 \\
dx^* & = & \mathcal{S}_{Nn}(dx^*_0) &
\end{array}
$$

FIG. 1.5 – *Programme linéaire cotangent*

$$
\begin{array}{lcll}
x_0 & = & \mathcal{I}_{nN}(x) & \\
x_i & = & f_i(x_{i-1}) & i = 1 \dots m \\
y & = & \mathcal{S}_{Np}(x_m) & \\
 & & & \\
(dx^{\star\,1}_m, \cdots, dx^{\star\,k}_m) & = & (\mathcal{I}_{pN}(dy^{\star 1}), \dots, \mathcal{I}_{pN}(dy^{\star k})) & \\
J^\star_i & = & Df^\star_i(x_{i-1}) & \rceil \quad i = m \dots 1 \\
(dx^{\star\,1}_{i-1}, \cdots, dx^{\star\,k}_{i-1}) & = & (J^\star_i dx^{\star 1}_i, \dots, J^\star_i dx^{\star k}_i) & \rfloor \\
(dx^{\star 1}, \cdots, dx^{\star k}) & = & (\mathcal{S}_{Nn}(dx^{\star 1}_0), \dots, \mathcal{S}_{Nn}(dx^{\star k}_0)) &
\end{array}
$$

FIG. 1.6 – *Programme linéaire multi-cotangent parallèle*

$$
\begin{array}{lcll}
x_0 & = & \mathcal{I}_{nN}(x) & \\
J^\star_i & = & Df^\star_i(x_{i-1}) & \rceil \quad i = 1 \dots m \\
x_i & = & f_i(x_{i-1}) & \rfloor \\
y & = & \mathcal{S}_{Np}(x_m) & \\
 & & & \\
dx^{\star\,1}_m & = & \mathcal{S}_{Nn}(dy^{\star 1}) & \\
dx^{\star\,1}_{i-1} & = & J^\star_i dx^{\star 1}_i & i = m \dots 1 \\
dx^{\star 1} & = & \mathcal{I}_{pN}(dx^{\star 1}_0) & \\
 & & & \\
dx^{\star\,k}_m & = & \mathcal{S}_{Nn}(dy^{\star k}) & \\
dx^{\star\,k}_{i-1} & = & J^\star_i dx^{\star k}_i & i = m \dots 1 \\
dx^{\star k} & = & \mathcal{I}_{pN}(dx^{\star k}_0) &
\end{array}
$$

FIG. 1.7 – *Programme linéaire multi-cotangent séquentiel*

que l'implantation de l'injection et la surjection pour réaliser cela en pratique (voir Section 2.3). Cette méthode est basée sur le calcul des *variables actives* du programme qui sont influencées par les variables indépendantes et influencent les variables dépendantes.

1.2.3 Complexité du programme dérivé

Cette section décrit la complexité en nombre d'opérations arithmétiques du programme dérivé qui calcule une dérivée directionnelle en mode direct ou un gradient en mode inverse. Pour évaluer la complexité du programme dérivé par rapport au programme original, le *modèle 3-adresses* est utilisé. Le modèle 3-adresses est une restriction du *modèle vectoriel* pour laquelle chaque fonction élémentaire utilise au plus deux variables d'entrée ($n_i = 2$) et calcule une seule variable de sortie ($p_i = 1$) en effectuant une seule opération. Pour obtenir ce modèle à partir du modèle vectoriel, il suffit d'introduire des variables intermédiaires ce qui ne change pas la complexité en nombre d'opérations arithmétiques.

Toutes les fonctions intrinsèques (tels que *abs*,*sign*) des langages de programmation peuvent être ré-écrites à partir des opérations arithmétiques de base $+, -, *, /$. Étudier ces quatre opérations suffit donc à évaluer le coût supplémentaire en nombre d'opérations du programme dérivé par rapport au programme original associé.

Les Figures 1.8, 1.9 et 1.10 présentent les dérivées en *mode direct* et *inverse* des instructions cible $y = x_1 + x_2$, $y = x_1 - x_2$, $y = x_1 * x_2$ et $y = x_1/x_2$ par rapport à leurs variables d'entrée x_1 et x_2.

Le nombre maximum d'opérations effectuées lors de l'évaluation de la dérivée d'une opération cible est atteint pour le quotient. Il vaut 5 si toutes les opérations sont comptées et 3 si seules les divisions sont considérées. Si on note N le nombre d'opérations coûteuses, des travaux théoriques [BS83, Mor85] ont prouvé que la complexité d'un *programme 3-adresses* vérifiait l'Identité 1.6.

$$\frac{N(P,P')}{N(P)} \leq 3 \tag{1.6}$$

Cette borne théorique prend en compte les opérations arithmétiques mais pas les opérations mémoires, et reste donc valable par introduction de variables intermédiaires. Elle est donc valide par extension du modèle de programme de 3-adresses, au modèle scalaire ou au modèle vectoriel. Cette complexité en nombre d'opérations ne peut être extrapolée en temps de calcul qu'en considérant que le programme est exécuté sur une machine à mémoire infinie avec des accès mémoires "gratuits", ce qui n'est pas réaliste. Je présente par la suite les bases d'un calcul pratique de la complexité des programmes dérivés.

$$\begin{aligned} y &= x_1 + x_2 \\ dy &= dx_1 + dx_2 \end{aligned}$$

(a)

$$\begin{aligned} y &= x_1 + x_2 \\ dx_1^\star &= dx_1^\star + dy^\star \\ dx_2^\star &= dx_2^\star + dy^\star \end{aligned}$$

(b)

FIG. 1.8 – *Dérivées directionnelle et gradient d'une somme*

$$\begin{aligned} y &= x_1 * x_2 \\ dy &= x_1 * dx_2 + dx_1 * x_2 \end{aligned}$$

(a)

$$\begin{aligned} y &= x_1 * x_2 \\ dx_1^\star &= dx_1^\star + x_1 * dy^\star \\ dx_2^\star &= dx_2^\star + x_2 * dy^\star \end{aligned}$$

(b)

FIG. 1.9 – *Dérivées directionnelle et gradient d'un produit*

$$\begin{aligned} y &= x_1/x_2 \\ s_1 &= y * dx_2 \\ s_2 &= dx_1 - s_1 \\ dy &= s_2/x_2. \end{aligned}$$

(a)

$$\begin{aligned} y &= x1/x2 \\ s_1 &= dy^\star/x_2 \\ dx_1^\star &= dx_1^\star + s_1 \\ dx_2^\star &= dx_2^\star - y * s_1 \end{aligned}$$

(b)

FIG. 1.10 – *Dérivées directionnelle et gradient d'un quotient*

1.3 Dérivée d'une instruction

1.3.1 Variable, valeur et case mémoire

Dans le modèle vectoriel présenté Section 1.2.1, chaque instruction élémentaire modifie un vecteur de valeurs de taille N où N représente le nombre total de variables scalaires du programme. Ce modèle mathématique ne permet pas de représenter un programme réel car contrairement aux *variables mathématiques* dont la valeur est constante durant tout le calcul, la valeur d'une *variable informatique* est couramment modifiée durant l'exécution du programme.

Une *variable informatique* est un nom associé par commodité à un emplacement mémoire (à partir d'une certaine adresse et sur une certaine longueur dépendant de son type numérique). Dans cet emplacement, différentes valeurs parfois sémantiquement indépendantes peuvent être rangées. On appelle *instance d'une variable* toute valeur qui peut être prise par la variable et *écrasement d'une variable* la modification de sa valeur. La DAO réplique la longueur des variables originales sur les variables dérivées puisque chaque variable dérivée est définie avec le même type numérique que la variable originale.

Un *chemin* est un accès à une case mémoire construit à partir de variables, d'éléments de tableaux, de composantes de structures, associés à une arithmétique de pointeurs. Plusieurs chemins peuvent mener à (pointer vers) une même case mémoire, ils sont alors dits *aliasés* et permettent indifféremment de lire ou de modifier le contenu de cette case. La variété de ces chemins et le nombre d'alias possibles dépend entièrement de l'expressivité du langage source considéré. En `Fortran 77` par exemple seules les variables peuvent être aliasées alors qu'en `C` l'arithmétique de pointeurs permet d'aliaser n'importe quelles cases mémoire. La DAO réplique la structure des chemins originaux sur les chemins dérivés puisque ces derniers traversent les variables ou structures dérivées qui ont le même type que les originales.

Les instructions dérivées sont induites directement par le modèle vectoriel décrit Section 1.2.1 si les variables informatiques ne sont jamais écrasées et si les chemins ne sont pas aliasés. Par contre, si l'instruction considérée utilise des variables informatiques quelconques, l'écriture des instructions dérivées n'est pas immédiate. Le respect de la structure des variables et des chemins originaux entraîne des contraintes sur l'écriture des instructions dérivées. Les Sections 1.3.2 et 1.3.3 montrent sur des exemples simples comment les règles standard de dérivations doivent être adaptées si plusieurs instances d'une même variable, ou plusieurs alias d'une même case mémoire, sont utilisées dans une même instruction.

1.3.2 Dérivée sans alias

Cette section montre comment écrire les instructions dérivées à partir des calculs vectoriels dérivés $d_{i+1} = J_i d_i$ et $d^\star_{i+1} = J^\star_{m+1-i} d^\star_i$ où $J_i = Df_i(x_{i-1})$.

Chaque instruction originale correspond à une fonction f_i construite par la composition d'une fonction élémentaire e_i avec l'injection $\mathcal{I}_{p_i N}$ et la projection $\mathcal{S}_{N n_i}$ canoniques.

$$f_i \ : \mathbb{R}^N \overset{\mathcal{S}_{N n_i}}{\hookrightarrow} \mathbb{R}^{n_i} \xrightarrow{e_i} \mathbb{R}^{p_i} \overset{\mathcal{I}_{p_i N}}{\hookrightarrow} \mathbb{R}^N$$

Les matrices Jacobienne locales J_i et $J^\star_i$ de f_i sont présentées dans les Identités 1.7 et 1.8.

$$J_i = \mathcal{S}_{N n_i} \cdot De_i \cdot \mathcal{I}_{p_i N} \tag{1.7}$$

$$J^\star_i = \mathcal{S}_{N p_i} \cdot De^\star_i \cdot \mathcal{I}_{n_i N} \tag{1.8}$$

Soit a_1 (Figure 1.11(a)) une instruction non itérative utilisant les variables `X` `Y` et `Z`. On note `dX`, `dY` et `dZ` les variables dérivées respectivement associées aux variables originales `X`, `Y` et `Z`. Comme le montre la Figure 1.11, la formulation mathématique présentées dans les Identités 1.7 et 1.8 peut être mise en pratique directement sur les instructions non itératives. Pour que les séquences d'instructions dérivées Figures 1.11(c) et 1.11(d) soient optimales, il suffit de supprimer les instructions inutiles de la forme `dX = dX`.

Considérons maintenant le cas des instructions itératives. Comme les différentes instances d'une variable sont dénotées de la même manière, toutes les lectures d'une variable dans une instruction réfèrent à la même valeur. Le seul type d'instructions qui utilise plusieurs instances d'une variable sont les instructions itératives qui modifient une variable qu'elles lisent. Soit l'instruction itérative a_2 (Figure 1.12(a)) fabriquée à partir de l'instruction a_1 en remplaçant `Z` par `Y` pour que la variable `Y` soit lue et écrite.

La Figure 1.12 présente les instructions dérivées de a_2 dans les Figures 1.12(c) et 1.12(d). Les instructions dérivées en *mode direct* Figures 1.11(c) et 1.12(c) sont semblables que l'instruction soit itérative ou non. Soit R_i l'ensemble des variables lues (partie droite) par l'affectation i et W_i l'ensemble des variables modifiées par i, la règle de dérivation d'une instruction en mode direct est présentée dans l'Identité 1.9.

$$dy = \sum_{x \in R_i} \frac{\partial y}{\partial x} dx \quad \forall y \in W_i \tag{1.9}$$

Cette règle est optimisée en prenant en compte le sous-ensemble de variables lues ou écrites ou lieu de l'ensemble de toutes les variables.

Par contre, en mode inverse les séquences d'instructions Figures 1.11(d) et 1.12(d) sont différentes et ne peuvent être déduites l'une de l'autre par un simple re-nommage. Le repérage d'une instruction itérative se fait sur

```
Z = X * Y**2
```

(a) a_1

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ Y^2 & 2XY & 0 \end{pmatrix}$$

(b) J_1

```
dX = dX
dY = dY
dZ = Y**2 dX + 2XY dY
```

(c) $d_o = J_1 d_i$

```
dX = dX + Y**2 dZ
dY = dY + 2XY dZ
dZ = 0
```

(d) $d_o = J_1^\star d_i$

FIG. 1.11 – *Dérivée d'une instruction standard*

```
Y = X * Y**2
```

(a) a_2

$$\begin{pmatrix} 1 & 0 \\ Y^2 & 2XY \end{pmatrix}$$

(b) J_2

```
dX = dX
dY = Y**2 dX + 2XY dY
```

(c) $d_o = J_2 d_i$

```
dX = dX + Y**2 dY
dY =  2XY dY
```

(d) $d_o = J_2^\star d_i$

FIG. 1.12 – *Dérivée d'une instruction itérative*

$$dx^\star = dx^\star + \sum_{y \in W_i} \frac{\partial y}{\partial x} dy^\star \quad \forall x \in R_i \setminus W_i \tag{1.10}$$
$$dx^\star = 0 \quad \forall x \in W_i \setminus R_i \tag{1.11}$$
$$dx^\star = dx^\star \sum_{y \in W_i} \frac{\partial y}{\partial x} \quad \forall x \in R_i \cap W_i \tag{1.12}$$

TAB. 1.1 – *Règles de dérivation d'une instruction sans alias*

la syntaxe de l'instruction : l'instruction est itérative ssi $R_i \cap W_i \neq \emptyset$. Il suffit donc de définir des règles de dérivation différentes pour une instruction itérative ou non, comme présenté dans la Figure 1.1.

Ces règles de dérivation des instructions itératives ou non, mais sans alias sont utilisées dans les systèmes standard et présentées dans les publications [GK98, Gri00].

1.3.3 Dérivée en présence d'alias

Je définis un chemin comme une variable, un accès à une composante d'un chemin et le de-référencement d'un chemin. Deux chemins sont *aliasés* si ils correspondent à la même case mémoire physique, et il est alors possible de modifier cette case mémoire par l'un ou l'autre chemin. En présence d'alias les règles précédentes ne peuvent être utilisées. Pour simplifier l'écriture des exemples, je me place dans le cadre particulier d'alias entre variables. Si les variables `Z` et `Y` sont aliasées, l'évaluation de l'instruction a_1 a pour effet de modifier simultanément non seulement `Z`, mais aussi `Y`. L'instruction a_1 est une instruction itérative déguisée en instruction standard, il faut donc la dériver en utilisant les règles de dérivations itératives.

Si lors de l'évaluation des instructions originales les variables `Z` et `Y` sont aliasées, les variables `dZ` et `dY` sont aliasées lors de l'évaluation des instructions dérivées. L'évaluation des instructions dérivées Figure 1.11(c) est compatible avec les alias dérivés alors que les instructions présentées dans la Figure 1.11(d) calculent une dérivée fausse pour `dZ` et `dY`. Cette instruction doit donc être dérivées comme l'instruction itérative qu'elle est. Ceci implique de connaître les alias pour générer le programme dérivé correct.

La règle théorique dénotée *RULE 8* (page 76) dans [Gri00] affirme que la dérivation d'une instruction est indépendante du contexte d'exécution d'une instruction. En effet, en pratique il est possible d'éviter toute analyse d'alias en appliquant les deux règles pratiques suivantes :

1. Toute variable d'un programme doit être considérée comme active. En effet après l'exécution de l'instruction a_1, Y est actif alors que cette activité n'apparaît pas statiquement.
2. Les instructions correctes de dérivation sont obtenues en introduisant des variables temporaires permettant de séparer les calculs (appliqués

aux variables temporaires) des effets de bords (appliqués aux variables originales). La Figure 1.13(b) montre comment les variables temporaires sont introduites pour permettre la dérivation par l'algorithme standard de a_1 présenté dans la Figure 1.13(a).

Comme le montre la Figure 1.13, cette solution théoriquement parfaite ne peut être mise en oeuvre que sur des programmes de petite taille car si l'introduction de variables supplémentaires (utilisées localement) n'entraîne pas un sur-coût trop important pour une affectation, le traitement d'un appel qui peut utiliser de nombreuses variables devient arbitrairement coûteux en mémoire. De même considérer une variable scalaire comme active alors qu'elle ne l'est pas n'est pas trop pénalisant alors que la même sous-optimalité appliquée à un tableau peut suffire à rendre le programme dérivé inutilisable.

Aucune étude approfondie de la dérivation en présence d'alias n'a encore été réalisée et les outils de DAO actuels considèrent uniquement les programmes sans alias même si ils ne peuvent pas vérifier qu'un programme n'utilise pas d'alias. Je n'ai fait ici qu'introduire le problème de la dérivation d'une instruction en présence d'alias. Je propose dans la Section 3.2.3 quelques pistes de réflexion sur les problèmes de détection des alias, de règles de dérivation des instructions compatibles avec les alias, etc.

1.4 Mes sujets de recherche

La Section 1.2 présente le modèle de programme qui permet les calculs de complexité qui fondent la DAO. La dérivation d'un programme est étudiée comme une application directe de la dérivation des fonctions composées à un straight line program.

La distance existant entre ce modèle de dérivation et la dérivation d'un programme réel ne permet pas de rendre compte de tous les travaux de recherche en DAO. Faute de ce cadre général de description et de comparaison, chaque méthode, chaque algorithme est décrit comme un cas particulier.

Dans ce document, je défini un modèle de programme plus proche d'un programme général que le modèle classique. À partir de ce modèle, je montre qu'on peut étudier la construction de la trace de l'exécution d'un programme, sa dérivation et l'optimisation du programme dérivé indépendamment. Résoudre ces trois problèmes séparément permet d'obtenir des programmes dérivés efficaces utilisant les modes de dérivation déjà connus et surtout d'imaginer d'autres types de programmes dérivés encore plus efficaces.

1.4.1 Trace de l'exécution

Nous avons vu dans la Section 1.2 que pour évaluer la dérivée d'un programme vectoriel, il fallait connaître les matrices Jacobienne locales de

```
Z = X * Y**2
```

(a) a_1

```
S1 = X
S2 = Y
S3 = Z
S4 = S1 * S2**2
Z = S4
```

(b) a_3

```
dS1 = dX
dS2 = dY
dS3 = dZ
dS4 = dS1 * S2**2 + 2S1S2 * dS2
dZ = dS4
```

(c) $d_o = J_3 d_i$

```
dS1 = 0
dS2 = 0
dS3 = 0
dS4 = 0

dS4 += dZ dS1 += S2**2 * dS4
dS2 += 2S1S2 * dS4
dS4 = 0

dZ += dS3
dS3 = 0
dY += dS2
dS2 = 0
dX += dS1
dS1 = 0
```

(d) $d_o = J_3^\star d_i$

FIG. 1.13 – *Dérivée de l'instruction a_1 en présence d'alias*

toutes les affectations. De même, pour un programme général avec des structures de contrôle complexes, il faut composer ces matrices de façon cohérente par rapport au programme original.

Je dénomme *trace de l'exécution* la description abstraite qui contient toute l'information nécessaire à l'évaluation et la composition des matrices Jacobienne. Cette trace peut contenir soit la valeur des matrices elles-même, soit les valeurs originales et les opérations élémentaires dérivées à partir desquelles ces matrices peuvent être évaluées. De plus, il est possible de conserver les opérations originales au lieu des opérations dérivées puisque les règles de dérivation des opérations élémentaires sont bien connues. Ces différentes possibilités peuvent être combinées pour obtenir une trace optimale.

Les dérivées sont calculées par une interprétation de cette trace : en mode direct par interprétation direct (*First In First Out,FIFO*) ou en mode inverse par interprétation inverse (*Last In First Out,LIFO*). De même, les deux types d'outil de DAO (par surcharge d'opérateur ou transformation source-à-source) construisent et utilisent une telle trace. Cette vision découplée des problèmes de construction et d'interprétation de la trace permet donc d'unifier les deux modes et les deux implémentation de la DAO.
La DAO niveau machine basée sur l'interprétation d'un programme comme une suite d'opérations élémentaires peut être décrite et implémentée de façon simple, mais n'est pas toujours praticable sur des programmes de taille industrielle.
Une vision plus structurée du programme permet l'application de stratégies de dérivation ou d'optimisation globales (de niveau supérieur à l'instruction) qui rendent la DAO applicable même à des programmes volumineux.

1.4.2 Dérivation Multi-niveaux

Le modèle de programme multi-niveaux que je propose offre une vision structurée du programme original. Il s'agit d'une représentation hiérarchique dont les *blocs élémentaires* sont des programmes vectoriels.

Complexité pratique des programmes dérivés La modélisation classique permet une description simple de la DAO, et les calculs de complexité associés permettent sa promotion : si on peut calculer la fonction et une dérivée directionnelle ou un gradient en un coût d'au plus trois fois celui de la fonction, alors pourquoi s'en priver. Mais, cette description ne rend pas compte de la complexité pratique des programmes générés car le coût des opérations mémoire est négligé. J'ai défini des paramètres génériques associés au programme original à partir desquels la complexité pratique en temps et en espace du programme dérivé peut être calculée.

Algorithmes de dérivation multi-niveaux Les algorithmes de dérivation des affectations peuvent être directement déduits du modèle théo-

rique comme ceux d'une séquence d'affectations. Par contre, la dérivation d'un programme structuré offre de nombreuses possibilités de composition des dérivés des blocs, entraînant des complexités différentes pour le programme dérivé. En mode inverse de nombreux algorithmes peuvent être utilisées, et l'efficacité du programme dérivé dépend de l'adéquation du choix de l'algorithme avec les caractéristiques du programme original. Cette structure *multi-niveaux* permet de prendre en compte les dérivées optimisées pour certains types de blocs particuliers tels que les boucles itératives [Gri92, Gil92, GPRS96] ou parallèles [GK98, HFH01]. Mon travail a été de proposer des descriptions statique ou dynamique des programmes permettant de décrire et de classifier les différents types de programmes dérivés.

1.4.3 Optimisation du programme dérivé

Dans la description théorique de la DAO faite dans les sections précédentes, seule la dérivation de toutes les sorties du programme par rapport à toutes ses entrées est considérée. Si seules certaines dérivées sont nécessaires nous avons vu qu'il faut théoriquement introduire une injection et une projection canoniques (voir les Identités 1.4 et 1.5). Appliquer ces identités correspond à : *(1)* calculer toutes les matrices Jacobienne locales et donc toutes les dérivées, puis *(2)* extraire celles qui sont demandées par l'utilisateur. Ceci n'est pas raisonnable sur un programme industriel car la quantité de calculs inutiles serait trop importante. Je présente ici deux méthodes permettant d'implanter ces fonctionnalités de façon praticable par l'optimisation au vol du programme dérivé. Ceci réalise du *program slicing* sur le programme dérivé total.

Diminution du nombre de dérivées calculées

Il est fondamental que le programme dérivé ne calcule que les dérivées nécessaires. On appelle *variable indépendante* une variable par rapport à laquelle le programme est dérivé. Toute variable intermédiaire qui ne dépend pas d'une variable indépendante aboutit à une contribution nulle par construction qui ne doit pas être calculée.

On appelle *variable dépendante* une variable dont la dérivée est demandée par l'utilisateur. Pour implanter cette projection, il faut faire l'optimisation duale de la précédente en ne conservant que les variables intermédiaires qui influencent les variables dépendantes.

Diminution du nombre de valeurs dans la trace

Pour calculer les valeurs dérivées, nous avons vu qu'il faut fournir la valeur des variables originales. Par défaut on peut considérer que toute valeur calculée par le programme original est utilisée par sa dérivée. De même que

précédemment, cette stratégie naïve ne peut pas être mise en pratique sur un programme industriel. J'appelle *valeur nécessaire* toute valeur calculée par le programme originale et utilisée par le programme dérivé. Ce terme est traduit du vocable *required* utilisé par Ralph Giering et Thomas Kaminsky dans [GK01]. Les valeurs à tracer dans le programme original sont donc le sous-ensemble des valeurs nécessaires qui sont modifiées entre le moment où elles sont calculées et celui où elles sont utilisées dans le programme dérivé.

1.4.4 Développement d'outils

Les outils de DAO peuvent être classés par leur mode de fonctionnement : ceux qui agissent par *surcharge d'opérateur* et ceux qui effectuent une transformation *source-à-source*. `ADOL-C [GJM+96]` est le précurseur et le plus utilisé des outils basés sur la surcharge d'opérateur, mais il en existe plusieurs autres : `TADIFF`, `autodiff::Fad<T_float>`, etc. L'avantage de cette approche est la simplicité et la flexibilité de son implémentation et son principal inconvénient est la faible efficacité des programmes dérivés obtenus à partir d'applications de taille industrielle en particulier en mode inverse.

Par contre les outils qui effectuent une transformation source-à-source sont d'une complexité d'implémentation comparable à un compilateur. Ceci explique qu'il n'en existe qu'un petit nombre. `ADIFOR [BCC+92, BCK+98]` est le précurseur de ces outils encore développé (contrairement à `GRESS`) et a été développé initialement pour mettre en oeuvre le mode direct. `PADRE2 [Kub96]`, `TAMC [Gie97]` et `Odyssée [FP98]` sont nés peu après et ont eux été développés dés le départ pour mettre en oeuvre le mode inverse. L'avantage de cette approche est la potentielle efficacité des programmes dérivés obtenus, ainsi que la flexibilité d'utilisation du programme dérivé.

Mon travail de développement a consisté à étendre le spectre des programmes pouvant être dérivés par `Odyssée [FP98]` (outil source-à-source). Les programmes dérivés initialement se composaient d'un unique sous-programme alors que la version courante n'a plus de limitations sur la structure de programme. Pour cela, j'ai développé des algorithmes de dérivation interprocédurale permettant de dériver n'importe quelle structure de programme et des algorithmes d'optimisation spécifiques aux programmes dérivés. De plus, j'ai mis au point un algorithme de mode inverse utilisant une trace dynamique permettant le traitement de n'importe quel graphe de flot de contrôle. Grâce à cela, `Odyssée [FP98]` a atteint un niveau d'applicabilité et de fiabilité qui permet son utilisation en milieu industriel.

Chapitre 2

Synthèse des travaux

La dérivation de programme se décompose en problèmes pouvant être considérés séparément : *(1)* obtenir la trace du programme original, *(2)* construire le programme dérivé et *(3)* optimiser le programme dérivé. La construction d'un outil de DAO est le quatrième axe et permet la mise en pratique des trois précédents. Un programme dérivé peut-être considéré (voir Section 1.2.2) comme une suite d'affectations dérivées vectorielles de la forme :

$$d_{i+1} = J_i d_i \quad \text{ou} \quad d^\star_{i+1} = J^\star_{m+1-i} d^\star_i \quad \text{pour} \quad i = 1..m$$

où $J_i = Df_i(x_{i-1})$ est la matrice Jacobienne locale de l'instruction originale i et d_i, d_{i+1} sont les dérivées directionnelles locales et $d^\star_i$ et $d^\star_{i+1}$ sont les dérivées adjointes locales.
Cette description élémentaire montre les différents types d'informations nécessaire à l'exécution du programme dérivé :

1. les opérations permettant le calcul des matrices Jacobiennes locales,
2. les valeurs sur lesquelles portent ces opérations,
3. l'ensemble des instructions dérivées,
4. l'ordre d'exécution des instructions dérivées,
5. les valeurs d'entrée des dérivées.

Les opérations permettant le calcul des matrices Jacobiennes locales sont directement déductibles des opérations originales en utilisant les règles simples de dérivation des expressions. De même, l'ordre d'exécution des instructions dérivées est directement déductible de l'ordre original : en mode direct l'ordre dérivée est l'ordre original, alors qu'en mode inverse il est inversé. Les informations nécessaire à l'exécution du programme dérivé se décomposent alors en deux classes :

- les informations provenant du programme original :
    1. les opérations permettant le calcul original,
    2. les valeurs sur lesquelles portent ces opérations,
    3. l'ordre d'exécution des instructions originales.

– les informations propres au programme dérivé :

1. l'ensemble des instructions dérivées,
2. les valeurs d'entrée des dérivées.

L'évaluation du programme dérivé se fait donc en trois temps : *(1)* l'évaluation des valeurs originales, *(2)* l'évaluation des matrices Jacobienne, *(3)* l'exécution des instructions dérivées dans l'ordre adéquat par rapport à l'ordre original.

Pour récupérer les informations du programme original, il faut en exécuter une copie augmentée d'instructions de trace. Le modèle de DAO basé sur la construction dynamique de la trace complète de l'exécution est réalisé par *surcharge d'opérateur* et utilisé dans un outil tel qu'`ADOL-C` [GJM+96], `TADIFF`, `autodiff::Fad<T_float>`. La *trace de l'exécution* construite dynamiquement est appelée *tape* et contient toutes les opérations effectuées ainsi que les valeurs sur lesquelles elles ont porté, dans l'ordre d'exécution original. Le tape d'un programme de taille réel est très volumineux, puisque sa longueur est proportionnelle au produit du nombre d'opérations de base effectivement réalisées par le nombre de variables effectivement utilisées. Le programme dérivé n'est pas construit comme un programme indépendant mais évalué par interprétation du *tape* : calcul des matrices Jacobiennes locales et réalisation des produits matrice × vecteur. Le volume du *tape* et l'impossibilité d'appliquer des optimisations globales rendent ce modèle de DAO difficilement praticable sur un code de taille industrielle.

Dans le *tape* d'un programme, les instructions exécutées et les valeurs sur lesquelles elles portent sont considérées de la même manière. Or il existe une grande différence entre ces deux types d'information : les instructions exécutées peuvent être les mêmes pour deux jeux de données différents, alors que les valeurs sur lesquelles portent ces instructions seront différentes.
Pour tenir compte de cela, il est possible de faire une exécution abstraite du programme original pour connaître toutes ses exécutions possibles et une exécution concrète pour connaître l'exécution suivie et les valeurs sur lesquelles portent les opérations réellement effectuées. Ce modèle de DAO correspond à ce que font les outils basés sur la transformation *source-à-source*. Ces outils construisent par une exécution abstraite du programme original, un programme dérivé qui pour toutes les exécutions possibles du programme original, évalue la séquence des instructions dérivées. En contrepartie de cette généralité, le programme construit doit déterminer les paramètres permettant de choisir une exécution particulière du programme dérivé. Dans ce modèle, le programme dérivé peut contenir les évaluateurs des matrices Jacobiennes locale, la *trace de l'exécution* ne contient alors pas les opérations originales mais uniquement les valeurs sur lesquelles elles portent et les valeurs de contrôle de l'exécution. Évaluer les valeurs dérivées se fait dans ce cas aussi en deux temps : évaluation du programme original augmenté de traces et évaluation du programme dérivé qui lit la trace.

Dans cette première section j'ai considéré le programme original comme un programme straight-line : la suite des instructions exécutées. La dérivation n'intervient alors qu'au niveau de l'instruction ce qui ne permet l'application d'aucune stratégie ou optimisation globale.
Un programme général représente plusieurs programmes straight-line. Le choix du programme straight-line à exécuter se fait par la donnée de paramètres d'entrée du programme dont la valeur se propage lors des calculs pour permettre tous les choix intermédiaires. Le flot de contrôle de chaque exécution est abrégé par l'utilisation de structures de contrôle complexes qui correspondent à des pattern de contrôle souvent utilisés. Cette structuration des programmes traduit souvent le sens du programme, en effet les unités sémantiques de calcul sont traduites en blocs d'instructions plus ou moins indépendants.
La perte de la structure original à travers la dérivation est donc certainement dommageable pour l'efficacité du programme dérivé. La dérivation de programmes multi-niveaux réintroduit cette structure et ouvre ainsi la voie à de nombreuses possibilités d'optimisation.

2.1 Trace de l'exécution d'un programme

Le traitement (construction et accès) de la trace de l'exécution est le problème fondamental de la DAO en modes direct et inverse. Certaines valeurs calculées par le programme original sont indispensables au calcul des valeurs des dérivées et sont calculées par une exécution tracée du programme original.

Ces valeurs originales n'existent en général plus au moment où elles sont utilisées par le calcul dérivé et doivent donc être mémorisées puis restaurées ou recalculées. Il est possible de s'affranchir de ce problème pour certains algorithmes particuliers de mode direct ou inverse, mais le problème général demeure. Cette section présente les différents problèmes théoriques associés à la définition et au traitement de la trace.

2.1.1 Trace des valeurs nécessaires

Dans cette section, je considère le problème suivant: comment obtenir les valeurs permettant le calcul des Jacobiennes locales et les valeurs permettant de choisir une exécution particulière du programme dérivé. Appelons *valeur nécessaire* une valeur utilisée dans une instruction dérivée s' et qui est calculée à partir de valeurs utilisées dans son instruction originale s associée. J'appelle *trace des valeurs nécessaires* d'un programme, l'ensemble des valeurs nécessaires de ce programme.

La Figure 2.1 illustre les trois classes de *valeurs nécessaires* sur des exemples d'instructions simples A, B, C et leurs instructions dérivées par rapport à x appelées respectivement $direct(A)$, $direct(B)$, $direct(C)$ ou

```
z = k**2*y(i)*x(2)
```

(a) A

```
z = x(i+j)
```

(b) B

```
if (z .lt. y(j))
then A
else B
```

(c) C

```
zttl = k**2*y(i)*xttl(2) +
       k**2*yttl(i)*x(2)
```

(d) direct (A)

```
zttl = xttl(i+j)
```

(e) direct (B)

```
if (z .lt. y(j))
then direct(A)
else direct(B)
```

(f) direct (C)

```
xccl(2) += 2*k*y(i)*zccl
zccl     = 0.
```

(g) adjoint (A)

```
xccl(i+j) += zccl
zccl       = 0.
```

(h) adjoint (B)

```
if (z .lt. y(j))
then adjoint(A)
else adjoint(B)
```

(i) adjoint (C)

FIG. 2.1 – *Exemples élémentaires de valeurs nécessaires*

$adjoint(A)$, $adjoint(B)$ et $adjoint(C)$ en supposant que x, y et z ne sont pas aliasés.

Ces trois classes de valeurs nécessaires se définissent à partir du type de calcul dérivé dans lequel elles interviennent :

la Jacobienne locale Dans les Exemples 2.1(d) et 2.1(g) qui illustre ce cas, la valeur de $2 * k * y(i)$ est une dérivée partielle de la matrice Jacobienne locale nécessaire au calcul des instructions directes et adjointes de A notées $direct(A)$ et $adjoint(A)$ respectivement.

l'indexe des variables dérivées Dans les Exemples 2.1(e) et 2.1(h), la valeur $i + j$ est utilisée comme indexe des variables directe $xttl$ et adjointe $xccl$ nécessaire aux calcul de $direct(B)$ et $adjoint(B)$.

le contrôle Dans les Exemples 2.1(f) et 2.1(i), la valeur $z.lt.y(j)$ ou tout indicateur de la branche effectivement suivie par le programme original, est nécessaire au choix de la branche de programme dérivé ($diff(A)$ ou $diff(B)$) à exécuter dans le calcul de $diff(C)$ où $diff$ dénote *direct* ou *adjoint*.

Comme nous l'avons vu dans l'Exemple 2.1, les valeurs nécessaires sont le plus souvent calculées à partir de valeurs calculées par le programme original. Elles peuvent être tracées de différentes manières dont les deux extrêmes sont appelés *(1)* le mode *gros grain* et *(2)* le mode *grain fin*.
En mode *gros grain*, les valeurs nécessaires sont elles-mêmes tracées. En mode *grain fin*, les valeurs des variables originales sont tracées. La trace

gros grain des valeurs nécessaires de l'instruction C contient les valeurs $\{2 * k * y(i)\}$, $\{i + j\}$, $\{z.lt.y(j)\}$, alors qu'en mode grain fin elle contient les valeurs $\{k,y,i\}$,$\{i,j\}$,$\{z,y,j\}$.

On appelle *longueur de la trace* le nombre de valeurs scalaires qu'elle contient. Pour connaître sa taille physique, il faut prendre en compte la taille physique de chacune de ses valeurs. Une trace grain fin semble plus longue qu'une trace gros-grain puisque pour chaque valeur nécessaire, la trace contient plusieurs valeurs au lieu d'une et qu'elle peut contenir des tableaux complets (la valeur de tous les éléments). Par contre la trace grain fin autorise des optimisations car, si une même dérivée partielle (pour fixer les idées) peut difficilement être nécessaire pour plusieurs sous-traces de la même instruction ou plusieurs instructions dérivées, les valeurs utilisées pour calculer cette dérivée partielle peuvent l'être. Dans l'exemple ci-dessus, la trace gros-grain de l'instruction C contient 3 valeurs scalaires, alors que sa trace grain fin contient 6 valeurs scalaires et deux tableaux. Une fois optimisée par suppression des doublons, la trace grain fin de C ne contient plus que 5 valeurs dont un tableau puisque les valeurs de i, j et y ne sont conservées qu'une fois. Si une trace mixte, gros-grain sur les case de tableaux et grain fin sur les scalaires est utilisées, 5 valeurs scalaires sont tracées puisque les cases $y(i)$ et $y(j)$ sont tracées à la place du tableau y.

Dans les sections suivantes chaque composante est décrite de façon plus précise.

2.1.2 Composantes de la trace

Je considère ici les sous-traces indépendantes formées des valeurs nécessaires de chaque classe. Nous allons voir que les complexités en nombre d'opérations et en longueur de la trace sont très différentes d'une sous-trace à l'autre et qu'elles devraient donc être traitées différemment. Je dénote par `push` et `pop` les opérations d'accès en écriture et lecture des valeurs de la trace.

Trace des indexes

Dans les implémentations actuelles, la variable dérivée associée à chaque variable originale est de même type numérique et de même taille que cette dernière. Ceci signifie en particulier qu'un tableau du programme original est associé à un tableau dérivé de même taille. Ce choix n'est pas optimal puisque même si seule une composante de ce tableau est dérivée, toutes les autres composantes seront à la fois calculées, mais cette solution semble de beaucoup le plus simple. L'autre méthode consisterait à associer à chaque variable ou case de tableau actif, un indexe dans un espace mémoire (ou disque) global mais il faut alors gérer dynamiquement la correspondance entre variable originale et indexe dans cette zone des dérivées.

```
v=i+j;
push(v);        pop(v);
z = x(v)        xccl(v) += zccl
```

(a) B_1^s (b) $B_1'^r$

FIG. 2.2 – *Trace gros grain de B*

```
push(i);        pop(j);
push(j);        pop(j);
z = x(i+j)      xccl(i+j) += zccl
```

(a) B_2^s (b) $B_2'^r$

FIG. 2.3 – *Trace grain fin de B*

Sous cette hypothèse, la valeur des indexes des variables dérivées est la même que celle des indexes des variables du programme original. Utiliser une trace *grain fin* implique donc des re-calculs systématiques alors qu'une trace *gros grain* élimine ce re-calcul. Les Figures 2.2 et 2.3 illustrent ce problème sur l'instruction B de l'Exemple 2.1(h). En mode *gros grain*, l'instruction B est ré-écrite en B_1^s pour conserver la valeur $i + j$ et l'instruction adjointe correspondante est $B_1'^r$. En mode grain-fin, l'instruction B est ré-écrite en B_2^s pour conserver les valeurs i et j et son instruction adjointe est $B_2'^r$. Dans cet exemple, la trace gros-grain permet d'éliminer trois opérations (une addition, une opération *pop*, et une opération *push*) et de libérer l'espace mémoire d'une sauvegarde. Il semble donc plus efficace d'utiliser une trace gros grain pour la sous-trace des indexes des dérivées.

Trace de la matrice Jacobienne

Les valeurs des éléments des Jacobiennes locales ne sont en général pas calculées par le programme original (sauf pour les calculs linéaires). Par contre, elles sont évaluées à partir de sous expressions partagées entre l'instruction originale et l'instruction dérivée. Pour cette sous-trace, conserver les valeurs des variables du programme original (grain fin) permet donc de minimiser la longueur de la trace par partage à la fois entre les éléments d'une même Jacobienne locale et entre plusieurs Jacobiennes locales consécutives.

Il semble donc plus efficace d'utiliser une trace grain fin pour la sous-trace des Jacobiennes. `ADIFOR [BCC+92, BCK+98]` utilise une trace gros-grain locale prônée dans [GLVM91] pour factoriser le calcul des Jacobiennes lors de la propagation de plusieurs dérivées directionnelles. Mais cette approche n'a été implantée dans aucun système réel pour le mode inverse.

Trace du contrôle

Pour conserver les paramètres de contrôle, la méthode la plus simple est de conserver explicitement la suite des instructions exécutées. Nous avons vu que ceci était faisable (voir *tape*) mais entraînait une trace volumineuse. Si un repère (indexe) est associé à chaque affectation, tracer la suite de ces indexes suffit à connaître de façon exacte le chemin d'exécution du programme. Cette méthode parfaitement équivalente à la première solution reste très volumineuse. Pour diminuer la longueur de cette *trace des instructions*, il est possible de grouper les séquences d'affectations en *blocs de bases*. Si un indexe est associé à chaque bloc de base, la longueur de la *trace des blocs de base* effectivement traversés sera en moyenne nettement inférieure à la trace des instructions. L'avantage majeur de cette méthode est de traiter n'importe quel type de contrôle : qu'il soit explicite `do j=1,n`, `if-then-else`, etc ou implicite `goto`, `do while`, etc. L'inconvénient de cette approche est l'introduction de valeurs supplémentaires par rapport au programme original. J'ai implanté cette méthode dans `Odyssée [FP98]` afin de dériver des programmes dont le graphe de flot n'était pas explicite (voir la Publication 18 référencée page 76) sans les modifier.

La méthode statique consiste à conserver la valeur des paramètres de contrôle explicites tels que les tests des branchements, les indices de début et de fin de boucles, pour reproduire les instructions originales de contrôle dans le programme dérivé. Par cette méthode, un branchement du programme original est transformé en un branchement dans le programme dérivé, une boucle en une boucle, etc. Les valeurs de contrôle nécessaires sont alors les mêmes dans le programme original et dans le programme dérivé. Dans ce cas, nous avons pu vérifier sur la trace des indexes qu'une trace gros grain semble optimale car elle minimise à la fois le nombre d'opérations et la longueur de la trace. Cette méthode utilisable uniquement sur un programme dont le contrôle est explicite est implémentée dans la plupart des outils actuels de DAO : `TAMC [Gie97]`, `PADRE2 [Kub96]`, `Odyssée [FP98]`, `ADIFOR [BCC+92, BCK+98]`.

2.1.3 Complexité de la trace

Cette section décrit la *complexité spatiale et temporelle* de la trace d'exécution. Notons V le nombre de variables lues ou écrites par un programme, et N le nombre d'opérations arithmétiques effectivement effectuées. On peut remarquer que dans un modèle sans écrasement $V = N$ alors que dans un programme réel $V \ll N$.

La Table 2.4 présente pour chaque composante de la trace : le *Type* de ses éléments, sa *Long*ueur et le nombre d'*Op*érations arithmétiques à effectuer pour calculer les valeurs originales et les valeurs nécessaires à partir des valeurs conservées dans la trace. Les trois formes de trace de contrôle sont

	Gros Grain			Grain Fin		
	Type	Long	Ops	Type	Long	Ops
Index	*Integer*	N	N	*Integer*	$N * V$	$2 * N$
Jacobienne	*Real*	$N * V^2$	$3 * N$	*Any*	$N * V$	$3 * N$
Contrôle (1)	*Boolean*	N	N	*Any*	$N * V$	$2 * N$
Contrôle (2)	*Integer*	N	0			
Contrôle (3)				*String*	$N * V$	$2 * N$

FIG. 2.4 – *Complexité de la trace*

présentées : (1) la trace des valeurs de contrôle, (2) la trace des indexes des blocs de base, (3) la trace des opérations originales effectuées. Ce calcul théorique reflète bien ce qui se passe si les trois sous-traces sont traitées séparément. Par contre, elle ne prend pas en compte le partage qui peut exister à l'intérieur d'une sous-trace ou entre deux sous-traces. Comme nous l'avons vu dans la Section 2.1.1, une valeur de la trace grain-fin est en général nécessaire à une instruction pour plusieurs raisons et peut même être nécessaire à plusieurs instructions.

Dans les outils de DAO standards, une trace de l'exécution n'est utilisée qu'en mode inverse car il existe pour le mode linéaire tangent une optimisation permettant le calcul et l'utilisation des éléments de la Jacobienne locale en flux tendu. `ADIFOR [BCC+92, BCK+98]` utilise une trace gros-grain locale à chaque instruction pour implémenter son linéaire tangent multiple. En mode inverse, le mode de trace choisi est le mode grain fin pour les indexes et les matrices Jacobiennes : la trace est composée d'un ensemble de valeurs de variables du programme original. Par contre il existe plusieurs représentations de la trace de contrôle : la trace des instructions dans `ADOL-C [GJM+96]`, la trace grain fin des valeurs de contrôle dans `TAMC [Gie97]`, `PADRE2 [Kub96]`, et `Odyssée [FP98]`. `Odyssée [FP98]` implante aussi la trace des indexes des blocs de base permettant de gérer un graphe de flot quelconque.

2.2 Dérivation multi-niveaux

Cette section présente la dérivation de programmes généraux basée sur un *modèle multi-niveaux* beaucoup plus général que le *modèle vectoriel*. J'introduis ensuite une représentation graphique des programmes multi-niveaux permettant la comparaison syntaxique des programmes dérivés. Puis je définis la complexité pratique en temps de calcul et en taille mémoire des algorithmes de dérivation et leur évaluation sur les programmes dérivés. Enfin ces résultats de complexité sont appliqués aux algorithmes standards

de dérivation.

2.2.1 Modèle de programme multi-niveaux

Un programme *multi-niveaux* est un programme structuré en *blocs d'instructions* liés par des structures de contrôle quelconques. Ces blocs peuvent être une séquence d'instructions élémentaires (bloc de base), une instruction complexe (boucle), un sous-programme, etc. Ce modèle étend le modèle présenté dans [OVB71] car il s'agit d'une hiérarchie de blocs quelconques au lieu d'une hiérarchie de sous-programmes. Contrairement à un programme vectoriel, un programme multi-niveaux est potentiellement associé à plusieurs exécutions en fonction de la valeur des paramètres de contrôle. Pour clarifier la description des programmes, j'associe un nom à chaque occurrence d'un bloc et je considère dans cette section une seule exécution de chaque programme. Les résultats de complexité obtenus sur une exécution peuvent être généralisés au programme, en prenant en compte toutes les exécutions possibles.

Cette section présente les deux descriptions d'un programme multi-niveaux que j'utilise par la suite : l'*arbre d'appel* qui donne une vision hiérarchique des blocs d'instructions constituant un programme et le *chemin d'exécution* qui donne une vision linéaire ou séquentielle du programme.

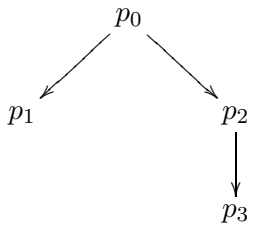
L'arbre d'appel se lit de haut en bas et de gauche à droite : il existe une flèche du bloc p_i vers le bloc p_j si p_i appelle p_j. Le chemin d'exécution se lit en partant du label START et en suivant le chemin jusqu'au label END. La ligne continue signifie l'exécution des instructions du programme et la ligne pointillé signifie l'appel-au et le retour-du bloc. Ces représentations sont utilisées par la suite pour décrire de façon visuelle les différents programmes dérivés constructibles.

La Figure 2.5 présente l'*arbre d'appel* et le *chemin d'exécution* de l'exécution E de P constituée de quatre blocs $p_0, \cdots p_3$. Le bloc p_0 effectue une partie de ses calculs notée p_0^1, exécute p_1, effectue une deuxième partie de ses instructions p_0^2, exécute p_2 et exécute les instructions de la dernière partie de p_0 notée p_0^3.

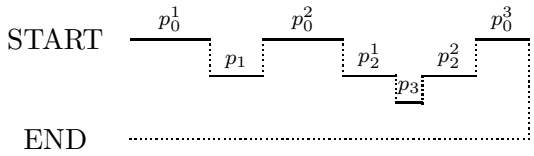
2.2.2 Complexité des algorithmes de dérivation

La représentation graphique d'une exécution d'un programme par son arbre d'appel et son chemin d'exécution permet une vision structurelle (syntaxique) du programme, mais ne rend pas compte de ses caractéristiques dynamiques (occupation mémoire et temps de calcul). Cette section présente la *complexité spatiale et temporelle des algorithmes de dérivation*. Pour mesurer le caractéristiques dynamiques des programmes dérivés, j'ai défini sept paramètres génériques :

α le taux d'activité en nombre d'opérations arithmétiques,



(a) Arbre d'appel



(b) Chemin d'exécution

FIG. 2.5 – *Description de l'exécution E de P*

β le taux d'activité en mémoire,

γ le nombre d'opérations de trace des valeurs nécessaires,

δ la longueur de la trace des valeurs nécessaires,

ϕ le nombre d'opérations de gestion des valeurs supplémentaires,

χ le nombre des valeurs supplémentaires,

ψ le facteur de re-calcul de la fonction initial.

Tous ces paramètres dépendent du programme original. Les paramètres α, β et γ dépendent des variables actives sélectionnées, alors que les paramètres δ, ϕ, χ et ψ dépendent de la stratégie de dérivation choisie.
Les paramètres α et β mesurent les taux d'activité du programme dérivé par rapport au programme original : si une variable est active, elle est associée à une variable dérivée ce qui augmente β et si une opération est active, elle engendre une ou plusieurs opérations dérivées ce qui augmente α. Les paramètres γ et δ mesurent la complexité de la mémorisation des variables nécessaires. Certaines stratégies de dérivation nécessitent des valeurs autres que les valeurs nécessaires (le contexte d'appel des sous-programmes par exemple) appelées ici valeurs supplémentaires. La complexité du traitement de ces valeurs supplémentaires est mesurée par les paramètres ϕ en temps et χ en espace. D'autre part, le (re-)calcul nécessaire à la production des valeurs nécessaires ou supplémentaires est mesuré par le paramètre en temps ψ et par aucun paramètre en espace puisqu'il n'implique pas de besoin en espace mémoire supplémentaire.

Soient T (respectivement T') le temps d'exécution du programme original (respectivement du programme dérivé) non optimisé, M (respectivement M') la mémoire utilisée, T_s le temps d'accès (lecture ou écriture) à un élément de la trace, et M_s l'espace nécessaire à la mémorisation d'une valeur scalaire. On peut décrire la complexité pratique du programme dérivé calculant une dérivée directionnelle ou un gradient obtenu en utilisant n'importe quelle stratégie de dérivation par les deux équations suivantes :

$$T' = \alpha\ T + \gamma\ T_s + \phi T_s + \psi T \quad (2.1)$$

$$M' = \beta\ M + \delta M_s + \chi M_s \quad (2.2)$$

Ce schéma de complexité pratique permet de prendre en compte tous les algorithmes de dérivations connus.

Évaluation de la complexité pratique

Pour transformer la complexité pratique décrite précédemment en un moyen de choix du meilleur algorithme de dérivation pour un programme P et un ensemble donné de variables actives, il faut connaître la valeur des sept paramètres génériques : α, β, γ et δ, ϕ, χ, ψ.
Les paramètres α, β et γ peuvent être obtenus en instrumentant le programme original pour calculer les nombres d'opérations et de variables dérivées et de valeurs scalaire à tracer. On peut de même évaluer les paramètres δ, ϕ, χ et ψ pour chaque algorithme de dérivation à partir de mesures élémentaires calculées pour chaque bloc.

À partir de la valeur des paramètres génériques, des caractéristiques T et M du programme P et des caractéristiques T_s et M_s de la librairie de gestion de la trace choisie, il est possible d'évaluer la complexité pratique de chaque programme dérivé. La comparaison de ces mesures de complexité donne à l'utilisateur final le moyen de choisir l'algorithme de DAO le mieux adapté à son programme pour une machine donnée. La complexité pratique d'un programme doit être évaluée pour chaque choix de variables actives et chaque jeu de données.

Influence du compilateur sur la complexité pratique

Le dernier facteur à prendre en compte lors de l'extrapolation de la complexité pratique du programme dérivé à partir de mesures sur le programme original, est la capacité d'optimisation du compilateur utilisé.
On note T_o et T'_o les temps de calcul du programme original et de sa dérivée compilés en utilisant un compilateur optimisant. Si on note $C = T'/T$ le ratio en temps de calcul de la fonction et de sa dérivée non optimisés, et $C_o = T'_o/T_o$ le ratio obtenu si les programmes sont compilés en mode optimisé, un défaut d'optimisation du compilateur peut apparaître $C_o \leq C$.

Ce défaut d'optimisation vient de différents facteurs et en particulier de l'augmentation importante du nombre de lignes et de variables qui inhibent certaines optimisations. Par exemple, la distance entre les instructions partageant des sous-expressions communes ayant augmenté, le compilateur peut ne plus reconnaître et mettre en oeuvre ce partage.

2.2.3 Dérivées de programmes multi-niveaux

Les algorithmes de dérivation en mode direct décrits dans Section 1.2.2 sont des algorithmes assez standards qui sont implantés dans tous les systèmes actuels. Par contre, il existe de nombreuses façons d'écrire l'adjoint d'un programme multi-niveaux répertoriées dans [Gri00] et dans la Publication 7 référencée page 75. Parmi ces différentes possibilités, deux sont très utilisées : le *mode séparé* et le *mode joint*. Je traduits les termes *split* et *joint* définis pas Andreas Griewank dans [Gri00]. Les programmes adjoints écrits à la main utilisent le *mode séparé*, alors que ceux générés automatiquement utilisent le *mode joint*. Comme je l'ai montré dans la Section 1.2.2, les modes joints et séparés sont applicables au mode direct autant qu'au mode inverse. Dans cette section, je décrits graphiquement les algorithmes qui utilisent les modes joint ou séparé en mode direct ou inverse, par les arbres d'appel et les chemins d'exécution pour en permettre une approche immédiate.

On associe à chaque bloc p_i du programme original, plusieurs *blocs dérivés* permettant de composer le *programme dérivé* :

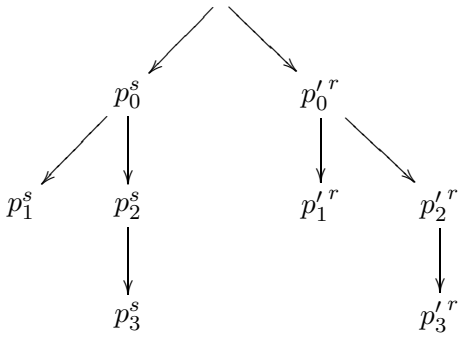
p_i^s qui exécute les mêmes calculs que p_i et mémorise la *trace de cette exécution*,

$p_i'^r$ qui restaure la trace gérée en mode *FIFO* et calcule les dérivées dans le sens de l'exécution originale,

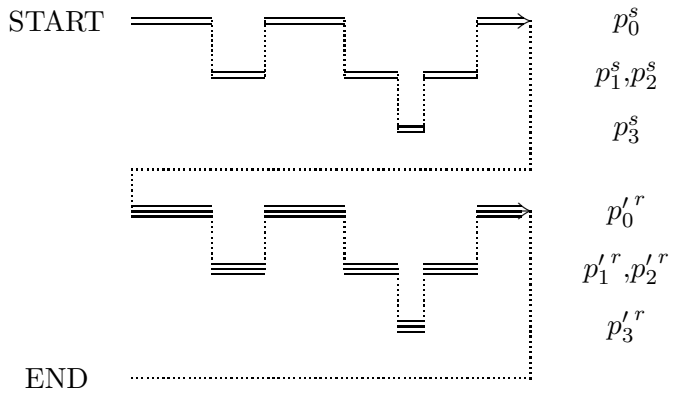
$p_i^{\star r}$ qui restaure la trace gérée en mode *LIFO* et calcule les variables adjointes dans le sens rétrograde.

Les Figures 2.6(a) et 2.6(b) et Figures 2.8(a) et 2.8(b) présentent les graphes d'appel et les chemins d'exécution des dérivées de P construites en *mode séparé*. Sur ces représentations du programme le sens du terme séparé apparaît clairement : les blocs ${p_1}^s$ et ${p_1'}^r$ ou $p_1^{\star r}$ sont loin l'un de l'autre lors de l'exécution.
Les Figures 2.7(a) et 2.7(b) et Figures 2.9(a) et 2.9(b) présentent les graphes d'appel et les chemins d'exécution des dérivées de P construites en *mode joint*. Dans les chemins d'exécution, les symboles $\bullet$ et $\blacktriangle$ signifient le stockage du *contexte d'exécution* des blocs courant et suivant et $\circ$ et $\triangle$ signifient la restauration de ceux-ci.

À la lecture de ces graphes, on constate que le *mode séparé* nécessite une trace beaucoup plus importante que le *mode joint*. En contrepartie, le *mode joint* conserve les *contextes d'exécution* et re-calcule même les blocs plusieurs fois pour construire l'adjoint.

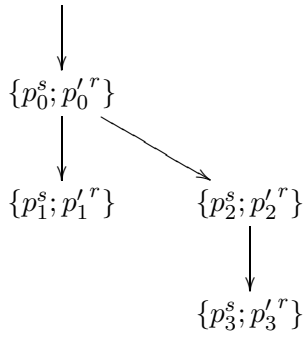


(a) Arbre d'appel

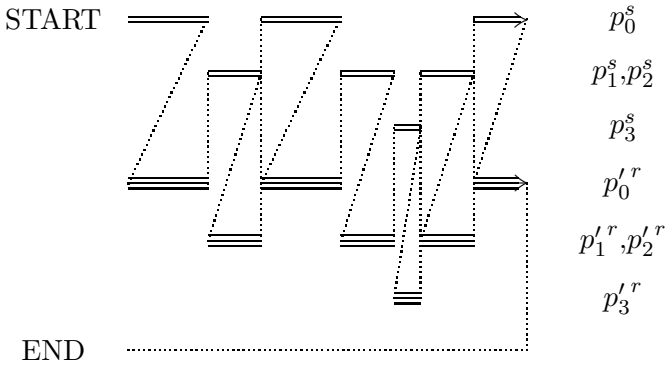


(b) Chemin d'exécution

FIG. 2.6 – *P' construit en mode séparé*

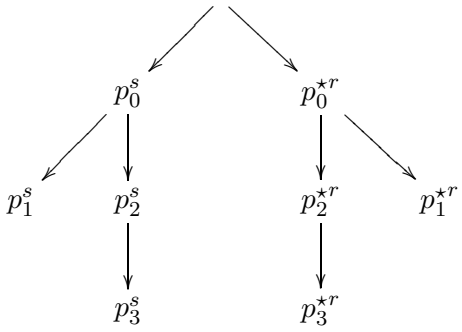


(a) Arbre d'appel

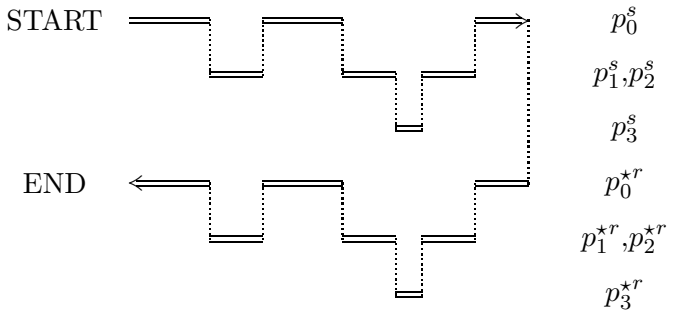


(b) Chemin d'exécution

FIG. 2.7 – *P' construit en mode joint*

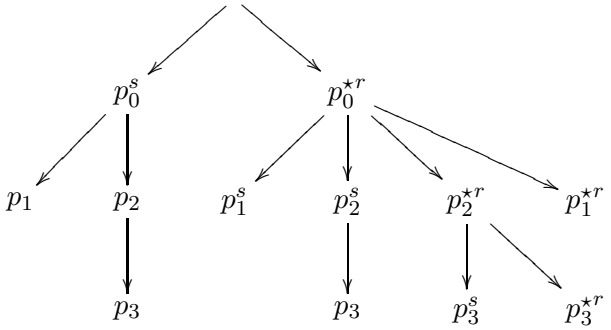


(a) Arbre d'appel

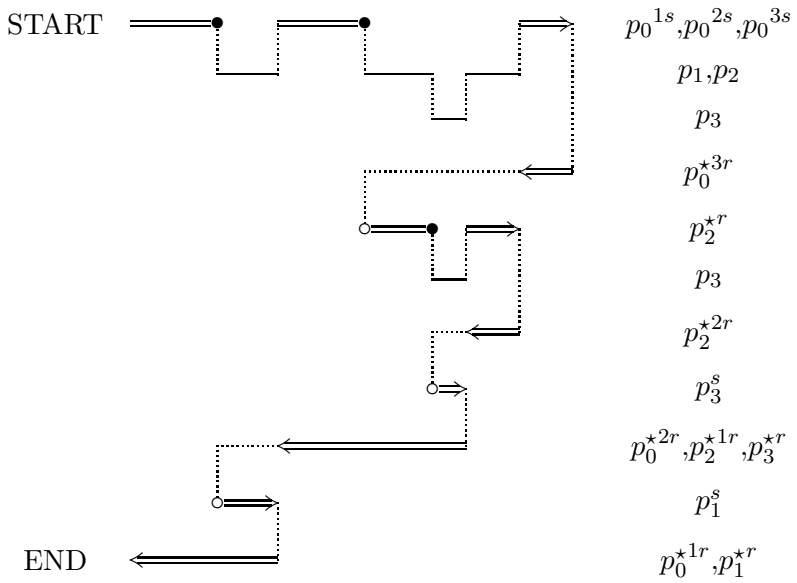


(b) Chemin d'exécution

FIG. 2.8 – *$P^\star$ construit en mode séparé*



(a) Arbre d'appel



(b) Chemin d'exécution

FIG. 2.9 – $P^\star$ *construit en mode joint*

L'utilisation des complexités en temps 2.1 et en mémoire 2.2 permet de comparer les stratégies de dérivation de façon plus précise.

En mode *séparé*, il n'y a pas de calcul de la fonction originale supplémentaire par rapport au calcul de la trace d'exécution $\delta = 0$ et il n'y a aucune valeur supplémentaire à la trace à conserver $\phi = 0$ et $\chi = 0$. De plus, la trace complète doit être conservée avant d'être utilisée, ce qui implique que $\gamma = \delta = \sum_{i \in P} l_i$ où l_i est la longueur de la trace sur un bloc i du programme. Ceci signifie que la complexité pratique du linéaire tangent et de l'adjoint en mode séparé sont de la même forme :

$$T'_{\text{séparé}} = T^{\star}_{\text{séparé}} = \alpha\ T + 2 \sum_{i \in P} l_i T_s \tag{2.3}$$

$$M'_{\text{séparé}} = M^{\star}_{\text{séparé}} = \beta\ M + \sum_{i \in P} l_i M_s \tag{2.4}$$

En *mode direct joint*, la trace de chaque bloc est utilisée avant que celle du bloc suivant soit mémorisée, la longueur de la trace est donc la longueur maximum de la trace d'un bloc calculé sur tous les blocs du programme.

$$T'_{\text{joint}} = \alpha\ T + 2 \sum_{i \in P} l_i T_s \tag{2.5}$$

$$M'_{\text{joint}} = \beta\ M + \max_{i \in P} l_i M_s \tag{2.6}$$

En *mode inverse joint*, la trace de l'exécution est utilisée de façon non linéaire ce qui signifie que $\delta \ll \gamma$ puisque $\delta = \max_{b \in P} \sum_{i \in b} l_i$ et $\gamma = \sum_{i \in P} l_i$ où l_i est la longueur de la trace du bloc i du programme, b est une branche du programme, $\phi = \chi = \sum_{i \in P} c_i$ où c_i est la longueur du contexte d'exécution de p_i. De plus, chaque bloc original est re-calculé un nombre n_i de fois égal à sa profondeur dans l'arbre d'appel total, si son temps d'exécution est noté t_i le temps total est donc $\psi = \sum_{i \in P} n_i * t_i$. En résumé, la complexité de l'adjoint obtenu en mode joint est de la forme :

$$T^{\star}_{\text{joint}} = (\alpha + \psi)T + 2 \sum_{i \in P} (l_i + c_i) T_s \tag{2.7}$$

$$M^{\star}_{\text{joint}} = \beta\ M + \max_{b \in P} \sum_{i \in b} (l_i + c_i) M_s \tag{2.8}$$

On peut extrapoler ces complexités pratiques des algorithmes qui calculent une dérivée aux algorithmes qui calculent plusieurs fois une même dérivée à partir de valeurs différentes des variables indépendantes de façon *parallèle ou séquentielle*.
Le coût de n dérivées calculées séquentiellement correspond à n applications du calcul pour une dérivée avec partage du calcul des valeurs nécessaires et supplémentaires: la taille mémoire est donc la même que pour une seule dérivée, alors que le coût en temps est n fois celui du calcul d'une dérivée

$$\psi = \sum_{i \in P} n_i * t_i \qquad \alpha'' = 1 + n(\alpha - 1) \qquad \beta'' = 1 + n(\beta - 1)$$

FIG. 2.10 – *Notations*

	Calcul séquentiel	Calcul parallèle
$T'^{\star}_{\text{séparé}}$	$n\alpha T + (n+1)\sum l_i T_s$	$\alpha'' T + 2\sum l_i T_s$
$M'^{\star}_{\text{séparé}}$	$\beta M + \sum l_i T_s$	$\beta'' M + \sum l_i M_s$
T'_{joint}	$n\alpha T + (n+1)\sum l_i T_s$	$\alpha'' T + 2\sum l_i T_s$
M'_{joint}	$\beta M + \max l_i M_s$	$\beta'' M + \max l_i M_s$
$T^{\star}_{\text{joint}}$	$n(\alpha+\psi)T + (n+1)\sum(l_i + c_i)T_s$	$(\alpha''+\psi)T + 2\sum(l_i + c_i)T_s$
$M^{\star}_{\text{joint}}$	$\beta M + \max\sum(l_i + c_i)M_s$	$\beta'' M + \max\sum(l_i + c_i)M_s$

FIG. 2.11 – *Calcul de n dérivées*

pour sa composante indépendante de la trace, alors que chaque élément de la trace est écrit une fois et relu n fois, ce qui le temps de traitement de la trace.
Le coût de n dérivées calculées parallèlement correspond à une application du calcul pour une dérivée avec pour chaque bloc n calculs indépendants de dérivée : la composante de la taille mémoire et du temps de calcul concernant les dérivées est donc multipliée par n alors que les autres composantes sont les mêmes que pour le calcul d'une seule dérivée. La Table 2.11 présente les complexités du calcul de n dérivées dans la première colonne pour le calcul séquentiel et dans la deuxième colonne pour le calcul parallèle.

Ces complexités pratiques permettent de comparer les stratégies, mais ne suffisent pas pour choisir la stratégie la mieux adaptée à un programme donné car pour faire cela il faut connaître α,β et $\{l_i,c_i,d_i,p_i\}_{i \in P}$. En mode inverse, j'ai réalisé des études pratiques qui montrent que la taille de la trace totale du mode inverse joint $\max_{b \in B}\sum_{i \in b}(l_i + c_i)$ est parfois du même ordre de grandeur que la trace du mode inverse séparé $\sum_{b \in B}\sum_{i \in b} l_i$ (voir la Publication 27 referencée page 77). En effet, en pratique la taille des contextes est très importante par rapport à la taille de la trace pure puisque les tableaux ne peuvent être traités que globalement. De même, une étude sur un programme thermo-hydraulique 3D présentée dans la Publication 10 referencée page 75 fait apparaître le coût en temps de calcul du traitement dynamique (mémorisation et restauration) de la trace et des contextes d'appel. Dans ce cas, seule une évaluation des paramètres pratiques correspondants au programme étudié peut permettre de choisir entre elles pour un programme

donné. Cette section présente le cas de programmes dérivés construits en utilisant le même mode pour tous les blocs du programmes, mais la combinaison des modes direct-inverse, joint-séparé est absolument nécessaire à la dérivation de programmes particuliers tels que les programmes itératifs. Ceci est encore plus vrai lorsqu'il s'agit de dérivées d'ordre supérieur comme l'a montré le travail qu'Andreas Griewank et moi avons présenté dans la Publication 6 référencée page 75. Ceci démontre l'importance de l'existence d'un module d'évaluation à priori de la complexité pratique de chaque algorithme sur un programme donné en associant les paramètres de dérivation à chaque bloc du programme dans les futurs outils de DAO.

2.3 Optimisation du programme dérivé

La Section 2.2.2 présente les paramètres intrinsèques à un programme qui permettent de contrôler les caractéristiques dynamiques du programme dérivé. Cette section présente des optimisations à effectuer pour diminuer la valeur de ces paramètres en fonction de l'ensemble des variables indépendantes et dépendantes spécifiques à une dérivée et ainsi diminuer la complexité pratique des programmes dérivés quelle que soit la stratégie de dérivation appliquée

Ces optimisations sont possibles par l'utilisation d'informations déduites d'une exécution abstraite du programme par analyse statique. Cette exécution déroule toutes les branches de calcul possibles et donne une approximation supérieure des informations en utilisant une approche conservative. Les analyses permettant l'optimisation des programme dérivés sont basées sur les mêmes techniques que celles utilisées pour les optimisations dans les compilateurs, mais peuvent être réalisées de deux façons différentes puisqu'il s'agit d'optimiser non pas le programme analysé mais un programme issu de ce dernier avant sa génération : *(1) à priori*, ou *(2) à posteriori*. Dans le premier cas *(1)*, les informations sont calculées sur le programme original P et utilisées pour ne pas générer les instructions supplémentaires à P à priori inutiles dans P'. Dans le deuxième cas *(2)* le programme est dérivé, puis les informations sont calculées sur le programme dérivé P' et utilisées pour supprimer à posteriori les instructions inutiles de P' [GK01]. La première approche offre l'avantage de faire une analyse sur un programme relativement petit, mais peut donner un résultat moins précis du fait de l'extrapolation des informations de P à P'. La seconde approche est plus précise, mais beaucoup plus coûteuse puisqu'elle implique une analyse sur un programme au moins deux fois plus volumineux en nombre de lignes ou de variables. Les deux approches doivent être combinées pour permettre la génération de programmes dérivés efficaces puisque les analyses à priori permettent uniquement de supprimer des instructions et des variables dérivées alors que les analyses à posteriori permettent de supprimer des instructions et des

variables originales ou dérivées. Une analyse de *variables vivantes* associé à un algorithme de *program slicing* permet de réaliser cela : appliquée au programme originale elle ne conserve que les calculs qui influencent les variables dépendantes et permet de faciliter la dérivation tout en optimisant le programme dérivé, alors qu'appliquée au programme dérivé elle ne conserve que les calculs qui influencent les dérivées des variables dépendantes par rapport aux variables indépendantes et optimise le programme dérivé. Ces analyses de variables vivantes sont générales et ne sont donc pas décrites ici, par contre celles utilisées dans la première approche sont spécifiques à la DAO et sont présentées dans cette section.

2.3.1 Diminution du facteur d'activité

Si le programme est dérivé par rapport à toutes ses variables, chaque variable réelle et chaque instruction du programme original est dérivée. Sous cette hypothèse, chaque variable réelle est associée à deux variables dans le programme dérivé, ce qui entraîne un facteur β de 2. Chaque opération élémentaire du programme original est active, elle est donc théoriquement associée à au plus 3 opérations dans le programme dérivé, le facteur α vaut donc au plus 3.

En général, les utilisateurs recherchent la dérivée de certains paramètres de sortie appelés *variables dépendantes* par rapport à certains paramètres d'entrée appelés *variables indépendantes*. Les variables intermédiaires sont dites *actives* si elles dépendent des variables indépendantes et influencent les variables dépendantes alors que les autres variables sont dites *passives*. Les variables passives apportent des contributions toujours nulles aux dérivées, alors que les variables actives peuvent entraîner des contributions non nulles.

La génération des dérivées provenant uniquement de calculs originaux impliquant au moins une variables active est la façon optimale de réaliser les injections $\mathcal{I}_{\nu n}$ et $\mathcal{I}_{\pi p}$ présentées dans les Identités 1.4 et 1.5 de la description formelle de la dérivation. Pour déterminer toutes les variables actives, une analyse statique du programme qui propage les variables actives à travers le programme est nécessaire.

Cette section décrit de façon synthétique l'algorithme de *propagation de variables actives*. Cette analyse bi-directionnelle sur le graphe de flot de contrôle se décompose en deux analyses mono-directionnelles : $\overrightarrow{A}$ qui propage en avant les variables influencées par les variables indépendantes et $\overleftarrow{A}$ qui propage en arrière les variables qui influencent les variables dépendantes.

Considérons qu'il existe un ordre partiel sur les instructions qui correspond à l'ordre d'exécution. Je note $\overrightarrow{A}_i^-$ l'ensemble des variables influencées par les variables indépendantes avant l'instruction i et $\overrightarrow{A}_i^+$ l'ensemble des variables après l'instruction i. On peut décomposer la propagation des va-

riables actives en suivant le modèle standard des analyses de programmes :

1. Calcul de l'information élémentaire :
   Cette étape répond à la question suivante: Supposons qu'on connaisse $\overrightarrow{A}_i^-$ l'ensemble des variables avant l'instruction i, quel est l'ensemble $\overrightarrow{A}_i^+$ des variables après l'instruction i?
   Notons I_i et O_i l'ensemble des variables d'entrée, respectivement de sortie de l'instruction i et un prédicat $dep_i(x,y)$ qui $\forall(x,y) \in I_i \times O_i$ renvoie vrai si la valeur de x influence celle de y à travers i.
   Ce calcul élémentaire est décrit par l'équation suivante :

$$\overrightarrow{A}_i^+ = \overrightarrow{A}_i^- \cup \{y \in O_i | \exists x \in (\overrightarrow{A}_i^- \cap I_i) \& dep_i(x,y) = true\} \quad (2.9)$$

2. Propagation en avant de l'information élémentaire :
   Cette étape répond à la question suivante: Supposons qu'on connaît $\overrightarrow{A}_j^+$ pour tous les prédécesseurs possibles de l'instruction i dans le graphe de flot, quel est $\overrightarrow{A}_i^-$ l'ensemble des variables actives à l'entrée de l'instruction i?
   Notons $pred_i$ l'ensemble des prédécesseurs possibles de i, la propagation d'une instruction à la suivante se fait par l'équation suivante :

$$\overrightarrow{A}_i^- = \cup_{j \in pred_i} \overrightarrow{A}_j^+ \quad (2.10)$$

3. Calcul de l'information complète :
   Pour connaître l'ensemble des variables actives pour chaque instruction, les deux équations précédentes son appliquées pour toutes les instructions en suivant le graphe de flot. Au point d'entrée du programme (qui n'a pas de prédécesseur), l'ensemble des variables actives est celui donné par l'utilisateur pour spécifier la dérivée. Cette information est propagée jusqu'à son point de sortie. En faisant cela, les arcs de retour du graphe de flot de contrôle n'ont pas été parcourus, il faut donc itérer ce calcul jusqu'à l'obtention d'un point fixe.

Le prédicat dep_i et les fonctions I_i, O_i et $pred_i$ sont pré-calculés de façon approchée pour toutes les instructions i du programme. Ce calcul peut être fait avant les itérations du point fixe puisque ces informations sont intrinsèques à l'instruction.
Cet algorithme cache de nombreux problèmes d'implantation (point fixe, calcul d'information flot sensitive, calcul interprocédural, analyse de tableaux) que je n'aborde pas ici pour des raisons de concision.

La propagation arrière de $\overleftarrow{A}$ se décrit de façon analogue en remplaçant les équations par :

$$\overleftarrow{A}_i^- = \overleftarrow{A}_i^+ \cup \{x \in I_i | \exists y \in (\overleftarrow{A}_i^+ \cap O_i) \& dep_i(x,y) = true\} \quad (2.11)$$

$$\overleftarrow{A}_i^+ = \cup_{j \in succ_i} \overleftarrow{A}_j^- \quad (2.12)$$

De façon analogue, il est possible de faire une analyse arrière qui détermine les dérivées impliquées dans le calcul des variables actives de sortie (dépendantes). $\overleftarrow{A}$ implémente de manière efficace les projections $\mathcal{S}_{n\nu}$ et $\mathcal{S}_{p\pi}$ introduites dans les Identités 1.4 et 1.5.

Finalement, l'ensemble des variables actives d'une instruction i se calcule par la combinaison des informations $\overrightarrow{A}_i^-$ et $\overleftarrow{A}_i^-$:

$$A_i = \overrightarrow{A}_i^- \cap \overleftarrow{A}_i^- \tag{2.13}$$

2.3.2 Diminution de la longueur de la trace d'exécution

De même qu'il est possible par une analyse statique de programme de diminuer le nombre de dérivées partielles calculées, on peut diminuer la longueur de la trace *grain fin* (en terme de valeurs de variables) par une analyse statique. Une variable est *absolument nécessaire* si sa valeur est *nécessaire* et si elle risque d'être modifiée. L'analyse des variables absolument nécessaires propage en avant la portée des variables nécessaires dans le programme.
La portée d'une variable v se définit par un intervalle $i..j$ tel que i est l'indice de l'instruction qui calcule v et j l'indice de l'instruction qui calcule à nouveau v. La variable v doit être tracée après l'instruction i et avant l'instruction k si elle est nécessaire à l'instruction k et $k <= j$.

L'algorithme de *propagation de variables absolument nécessaires* se décompose donc en deux étapes : *(1)* propagation en avant des portées des variables nécessaires, et *(2)* choix de l'instant où entrer la valeur d'une *variable nécessaire* dans la trace. Notons L_i l'ensemble des variables pour lesquelles l'instruction i appartient à l'intervalle de portée courant. La première étape se décompose de la même manière que précédemment en deux équations sur lesquelles itérer jusqu'à obtenir un point-fixe. Notons $N_i \subset I_i$ l'ensemble des variables nécessaires dans l'instruction i et O_i l'ensemble des variables de sortie de l'instruction i.

$$L_i^+ = (L_i^- \cup N_i) \setminus O_i \tag{2.14}$$
$$L_i^- = \cup_{j \in pred_i} L_j^+ \tag{2.15}$$

A partir de l'information L_i, on connaît la portée de toutes les variables du programme. Si les variables sont tracées avant chaque modification, il reste à calculer l'information T_i qui donne la liste des valeurs à conserver avant l'instruction i en utilisant l'équation :

$$T_i = L_i^+ \cap O_i \tag{2.16}$$

Comme pour l'analyse précédente, les fonctions N_i, O_i et $pred_i$ peuvent être pré-calculés pour toutes les instructions i du programme. Cette analyse peut être modifiée pour tracer les variables nécessaires avant la première

$\tilde{N}(x,false)$	=	$\emptyset$		
$\tilde{N}(x,true)$	=	$\{x\}$		
$\tilde{N}(add(e_1,e_2),false)$	=	$\cup_{i\in[1..2]}\tilde{N}(e_i,false)$	si	$A(e_1) \wedge A(e_2)$
	=	$\tilde{N}(e_1,false)$	si	$A(e_1) \wedge \overline{A}(e_2)$
	=	$\tilde{N}(e_2,false)$	si	$\overline{A}(e_1) \wedge A(e_2)$
	=	$\emptyset$	si	$\overline{A}(e_1) \wedge \overline{A}(e_2)$
$\tilde{N}(add(e_1,e_2),true)$	=	$\cup_{i\in[1..2]}\tilde{N}(e_i,true)$		
$\tilde{N}(minus(e1),false)$	=	$\tilde{N}(e1,false)$	si	$A(e_1)$
	=	$\emptyset$	si	$\overline{A}(e_1)$
$\tilde{N}(minus(e1),true)$	=	$\tilde{N}(e1,false)$		
$\tilde{N}(mul(e_1,e_2),false)$	=	$\cup_{i\in[1..2]}\tilde{N}(e_i,true)$		
$\tilde{N}(mul(e_1,e_2),true)$	=	$\cup_{i\in[1..2]}\tilde{N}(e_i,true)$		
$\tilde{N}(app(op,[e_1,\cdots e_n]),false)$	=	$\cup_{i\in[1..n]}\tilde{N}(e_i,true)$		
$\tilde{N}(app(op,[e_1,\cdots e_n]),true)$	=	$\cup_{i\in[1..n]}\tilde{N}(e_i,true)$		

TAB. 2.1 – *Règles de construction de* $\tilde{N}$

utilisation ou pour grouper les valeurs à tracer en modifiant les règles de calcul de L_i. Cette stratégie permet de diminuer le coût en temps de calcul des accès en lecture et écriture de la trace et n'a été implantée dans aucun système. Cette analyse peut être réalisée indépendamment sur chaque composante de la trace en donnant trois définitions à N_i et à L_i suivant que l'analyse concerne les variables nécessaires aux indexes, aux matrices Jacobiennes, ou au contrôle. Pour la trace du contrôle ou des indexes, les fonctions N_i et L_i sont confondues, par contre la fonction N_i est complexe à définir syntaxiquement pour la trace des matrices Jacobiennes. La définition exacte de la fonction N_i donnée mathématiquement par l' Identité 2.17 ne peut être directement utilisée sauf en appliquant au vol une dérivation du second ordre des instructions.

$$N_i \quad = \quad \{x \in I_i | \exists y \in O_i \frac{\partial^2 y}{\partial^2 x} \neq 0\} \tag{2.17}$$

Des sur-approximations plus ou moins précises de N ne nécessitant pas une dérivation du second ordre peuvent être utilisées : la plus simple consiste à considérer toutes les variables lues dans l'instruction i comme nécessaires. On note $A(e)$ le prédicat qui est vrai si e est une expression active c.a.d qu'elle contient au moins une variable active. Je présente dans la Table 2.1 une approximation plus fine notée $\tilde{N}$ qui utilise l'active-linéarité et qui est définie récursivement sur la syntaxe des expressions. Cette analyse n'a été implantée dans aucun système existant.

2.3.3 Diminution de la taille du contexte

On appelle *contexte d'exécution* d'un bloc l'ensemble des valeurs des variables avant l'exécution qui une fois restaurées assurent la même exécution du bloc. Lors de la dérivation d'un programme multi-niveaux en *mode joint*, le contexte d'exécution de tous les blocs doit être conservé. La définition naïve du contexte d'exécution c_i du bloc p_i est l'ensemble des valeurs des variables globales ou des paramètres lus par p_i. Comme le contexte d'exécution de chaque bloc est sauvé, une première optimisation consiste à définir le contexte c_i comme l'ensemble des entrées directement modifiée par p_i.

Cette définition est optimale pour les variables scalaires, mais pas pour les tableaux. En effet, même en faisant une analyse de régions de tableaux lues et écrites [CI96], il est en général impossible de savoir de façon certaine quelles composantes d'un tableau sont lues ou écrites. En particulier, l'utilisation dans le programme original de fonctions compliquées ou de tableaux d'indirection éliminent toute possibilité d'analyse statique. La seule information certaine concerne donc le plus souvent le tableau complet et non une case particulière : c'est à dire que si une case du tableau t est lue ou écrite, toutes les cases du tableau t sont considérées comme lues ou écrites. Calculer le contexte statiquement donne donc toujours des résultats sous-optimaux sur les variables à valeur tableau.

Une méthode plus précise pour calculer le contexte d'exécution c_i de p_i consiste à le calculer dynamiquement. Pour cela, il faut modifier le programme p_i pour qu'il sauvegarde la valeur de chaque variable (ou case de tableau) lue avant sa première modification. Le bloc ainsi modifié n'effectue pas tous les calculs de p_i, mais seulement ceux effectués avant le premier calcul de la dernière variable calculée. Le programme qui calcule c_i peut être généré par transformation de programme à partir de p_i et d'une information calculée par analyse statique. Ici encore, l'information sur le tableau est peu précise, mais l'exécution corrigera cette imprécision. Cette imprécision entraîne au pire une exécution complète du bloc. Cette méthode semble excellente mais n'a encore été implémentée dans aucun système connu et doit être évaluée sur des programmes de taille réelle.

2.4 Développement et exploitation d'outils

Il existe deux types d'implantations d'outils de DAO : ceux qui sont basés sur la *surcharge d'opérateur* et ceux qui fonctionnent par transformation *source-à-source*. Un outil basé sur la surcharge d'opérateurs consiste en un module qui redéfinit chaque opérateur de base du langage $+$, $\times$, $-$, $/$, *abs*, *cos* pour qu'il réalise l'opération initiale ainsi que des opérations supplémentaires. Ce mode d'implémentation est très flexible car un outils est constitué de modules écrits dans le même langage que le programme source ou une extension de ce langage (Fortran 90 pour Fortran 77, C++

pour C), mais il ne permet pas l'intégration simple du programme dérivé (qui est donc un exécutable) dans une chaîne logiciel.

De plus, la surcharge d'opérateur ne permet qu'une vision locale du programme puisque chaque opération élémentaire est traitée indépendamment. Le *mode direct* s'implante naturellement par la surcharge des opérateurs de base $+, \times, -, /, abs, cos$ en des opérateurs $\oplus, \otimes, \ominus$ qui réalisent l'opération originale sur les valeurs originales et les opérations dérivées sur les valeurs dérivées puisque nous avons vu que pour certains algorithmes de mode direct les opérations originales et dérivées pouvaient être combinées. Par exemple l'opérateur $+ : \mathbb{R} \to \mathbb{R}$ sur les réels peut être redéfini $\oplus : \mathbb{R}^2 \to \mathbb{R}^2$ sur les couples réel original et réel dérivé pour calculer une dérivée directionnelle. La grande flexibilité de la surcharge apparaît ici clairement puisqu'il est aisé de définir un module pour le calcul d'une ou plusieurs dérivées directionnelles, d'un développement de Taylor, ou de dérivées d'ordre supérieur.
Le module qui implante le *mode inverse* pour sa part se décompose en deux parties : la re-définition des opérateurs de base qui permet l'écriture du *tape*, et le dérivateur proprement dit qui interprète le *tape* à l'envers pour calculer les dérivées. Là encore, le *mode séparé* s'implante naturellement puisque le dérivateur relit le tape linéairement et à l'envers. Par contre, il est très difficile d'implanter le *mode joint* car dans ce mode le tape est parcouru de façon non linéaire (en faisant des aller-retour).

La surcharge d'opérateur permet en mode direct et inverse d'implanter certains algorithmes de façon naturelle et c'est alors le moyen le plus immédiat de développer un outil de DAO. Mais il est extrêmement complexe de développer un outils implantant de cette manière tous les algorithmes de DAO connus. Le développement d'un algorithme quelconque nécessite le codage d'informations sur la structure du programme dans le *tape*. Les inconvénients majeurs de cette approche sont donc la difficulté à implanter des algorithmes de dérivation qui utilisent la structure du programme original, mais aussi la difficulté à opérer des optimisations globales du programme.

2.4.1 Dérivation source-à-source

La dérivation *source-à-source* autorise l'implantation de toutes les stratégies connues, et de toutes les optimisations globales basées sur des informations statiques. De plus, le programme dérivé est un programme source du même type que le programme original et peut donc être inclus dans n'importe quelle chaîne logiciel. L'inconvénient majeur de l'approche source-à-source est la complexité de sa mise en oeuvre. En effet, un tel outil peut être décrit comme un *front-end* et un *back-end* de compilateur associés à un *middle-end* spécifique à la dérivation. Cette comparaison montre le poids des développements à réaliser avant de pouvoir mettre en oeuvre ou tester le middle-end. La dérivation par transformation de programme traduit le

programme original P en un programme dérivé P' par les étapes suivantes :

Phase 1 analyse lexicale et syntaxique du programme original P et traduction de P dans un format interne (arbre de syntaxe abstraite, table des symboles, graphe de flot de contrôle, ...),

Phase 2 normalisation de la représentation interne pour simplifier la phase d'analyse statique (ajout de variables intermédiaires, etc),

Phase 3 analyse statique (*variables dépendantes*, *variables actives*, *variables nécessaires*),

Phase 4 construction de la représentation interne du programme dérivé P' en utilisant l'algorithme choisi,

Phase 5 traduction du format interne de P' en un programme écrit dans le langage original.

Le front-end et le back-end effectuent respectivement les Phases 1 et 5 et sont des modules communs à tous les outils de transformation de programme, alors que le middle-end qui correspond aux Phases $2-4$ est spécifique à la DAO. À la complexité de mise en oeuvre commune aux outils de transformation de programme, s'ajoutent donc des problèmes intrinsèques à la DAO provenant de l'ajout de calculs supplémentaires dans un programme existant : *(1)* l'ajout de variables et d'instructions non triviales dans le programme original, et *(2)* l'optimisation du programme dérivé *à priori* (c.a.d à partir d'informations calculées sur le programme original).

2.4.2 Travail personnel

`Odyssée [FP98]` est un outils de DAO construit sur le modèle source-à-source qui dérive un programme `Fortran 77` en un programme `Fortran 77`. Soit P un programme qui calcule m sorties $Y = \{y_1, \ldots, y_m\}$ à partir de n entrées $X = \{x_1 \ldots x_n\}$. `Odyssée [FP98]` prend en entrée un programme P et un sous-ensemble $X_a \subseteq X$ d'entrées actives et rend un nouveau programme P' qui calcule les dérivées de toutes les variables de sortie $y_i \in Y$ par rapport aux variables $x_j \in X_a$.

J'ai développé les dernières versions de l'outil `Odyssée [FP98]` $\tilde{1}.7$ à partir d'un front-end et d'un back-end déjà existant dans `Odyssée [FP98]` $\tilde{1}.5$. Ceux-ci étaient suffisamment robustes et généraux pour que je n'ai pas à les modifier en profondeur, j'ai donc principalement étendu le middle-end d'`Odyssée [FP98]`.

Phase 2 J'ai développé une transformation permettant de rendre *3-adresses* un programme quelconque et de normaliser les appels de sous-programme.

Phase 3 L'implantation interprocédurale de l'analyse de dépendances et de la propagation des variables actives que j'ai réalisée a permis la dérivation de programmes de grande taille (60 000 lignes). L'analyse

de variables nécessaire a montré son efficacité sur des programmes de taille industrielle, mais reste à l'état de prototype.

Phase 4 J'ai développé un algorithme de mode inverse utilisant la trace des blocs au lieu des valeurs de contrôle permettant de dériver n'importe quel graphe de flot.

Tous ces travaux m'ont permis d'étendre `Odyssée [FP98]` à une plus grande classe de programmes et d'optimiser les programmes générés de façon significative. Finalement, la version courante d'`Odyssée [FP98]` permet de dériver automatiquement de gros programmes industriels en générant des programmes exécutables (c.a.d de complexité pratique raisonnable).

`Odyssée [FP98]` a été utilisé lors de nombreux travaux contractuels dans le cadre de projets européens AD-CAPE (avec Elf, BP, etc) et DECISION (avec Dassault, NAG, etc) et de contrats industriels (Dassault, Aérospatiale, Alenia, Essilor, EDF, etc). Ces évaluations de l'apport de la DAO à des problèmes spécifiques ont nécessité de ma part la formation, l'encadrement des personnes qui les ont réalisées et l'interprétation des résultats avec les clients.
De 1995 à 2000 L'exécutable d'`Odyssée [FP98]` a été accessible gratuitement sous contrat pour des utilisations non commerciales, j'ai donc mis `Odyssée [FP98]` à disposition des entreprises ou universités contractantes. Le source d'`Odyssée [FP98]` est depuis mai 2000 en libre accès par ftp anonyme.

2.4.3 Travail d'enseignement et d'encadrement

DAO de *programmes parallèles*

Ce travail a été réalisé par Patrick Dutto dans le cadre d'un contrat avec Dassault. Nous avons étendu `Odyssée [FP98]` aux programmes parallélisés utilisant MPI en considérant la librairie MPI comme une librairie externe. Nous avons utilisé la possibilité offerte par `Odyssée [FP98]` de décrire des sous-programmes externes, et n'avons donc pas modifié le noyau du dérivateur. Cette extension consiste en *(a)* un fichier de description des sous-programmes de la librairie MPI, *(b)* des fichiers `Fortran 77` contenant les définitions des dérivées des commandes MPI. Le fichier de description est lu par `Odyssée [FP98]` avant la dérivation et permet une dérivation cohérente de tout programme contenant des commandes MPI. Les fichiers de définition `Fortran 77` compilés et liés au programme généré permettent d'exécuter la dérivée. Une première phase de ce travail présentée dans la Publication 30 référencée page 77 a permis de valider cette extension en mode direct et une seconde présentée dans la Publication 29 référencée page 77 nous a permis de valider cette extension en mode inverse. Cette seconde étude a montré la difficulté d'interprétation des dérivées calculées en mode inverse pour un programme parallèle, par rapport à celles calculées en mode

direct comme le montre la Publication 15 referencée page 76.

DAO pour l'optimisation

Une première étude a été réalisée par Serge Fantino dans le cadre d'un contrat avec Essilor en *optimisation optique*. Le problème était de déterminer les focales d'astigmatisme par lancé de trois rayons : un lancé permettant le calcul du rayon moyen et deux lancés pour calculer le trajet de deux rayons voisins du rayon moyen.
Cette étude avait pour but de valider le calcul des focales d'astigmatisme par la dérivée d'un unique lancé de rayon calculé par différentiation automatique (par `Odyssée [FP98]`). Cette dérivée représente le calcul des focales accompagné de leur variations par rapport à une variation du rayon à l'entrée.

Une deuxième étude a été réalisée par Yves Papegay dans le cadre d'un contrat avec Alenia et concernait l'optimisation d'une forme 2-D d'ailes d'avions avec écoulement de fluide visqueux ou non. Nous avons modifié l'optimiseur pour que le gradient des contraintes et de la fonction objectif soient calculés non pas par différences divisées, mais par DAO. Ceci nous a amené à réfléchir à une méthodologie pour propager les variables actives à travers des lectures et écritures sur fichiers.

Dans les deux cas, les résultats obtenus ont permis de valider l'approche par un gain en précision des résultats et en temps de calcul et ont amené l'industriel client à utiliser la DAO.

Développement d'outils de DAO

Vladimir Vyskocil et Serge Fantino ont développé l'outil ODYPer pour répondre au problème de propagation d'erreur d'arrondi par le développement d'un mode inverse spécifique. Le logiciel ODYPer est un dérivateur de code `Fortran 90` particulièrement adapté à l'analyse de la propagation des erreurs d'arrondi en mode inverse. Il utilise une extension à `Fortran 90` du parseur `Fortran 77` du logiciel Odyssée mais implémente de nouvelles structures de données pour représenter et analyser le programme original et de nouveaux algorithmes de DAO en mode inverse.

Les caractéristiques principales du logiciel ODYPer le démarquant du programme `Odyssée [FP98]` sont les suivantes :

- Structures de données internes plus sophistiquées et unifiées permettant une mise en place plus rapide et aisée de nouveaux algorithmes de DAO.
- Sauvegarde des dérivées partielles intermédiaires nécessaires pour l'analyse de la propagation des erreurs d'arrondi.
- Nouvelle interface avec coloration syntaxique, exploration interactive du graphe d'appel des unités FORTRAN et langage de script basé sur TCL.

Formation à la DAO

La DAO est une nouvelle technique prometteuse mais souvent méconnue du monde industriel. Comme pour le Calcul Formel avant son enseignement en école préparatoire, il n'existe que peu de cursus qui l'intègrent. J'ai essayé par des formations et des exposés lors d'écoles d'été ou de formations spécifiques de la promouvoir :

1998 : Formation à la DAO au CNES (Cours, 12h).

1997 : École d'été du CEMRACS "Shape Optimization and Automatic Differentiation" (Cours, 3h).

1996 : École d'été INRIA "Techniques de développement pour programmes numériques" (Cours, 3h).

1996 : École d'été d'analyse numérique CEA - INRIA - EDF (TD, 60h).

J'ai aussi fait quelques cours en milieu académique :

Licence d'informatique (UNSA) Architecture du système de calcul formel Maple (40h), langage Le-Lisp (50h), responsable J. Chazarain, 1989, 1990.

DEA Mathématique et Informatique (UNSA) La DAO, un exemple de transformation de programme (9h), 1995, 1996, 1997.

Chapitre 3

Bilan et perspectives

3.1 Plate-forme de DAO

La recherche en DAO est très grandement freinée par le coût initial exorbitant du développement d'un outil général. Chaque idée nouvelle validée théoriquement est évaluée sur une application particulière par une transformation manuelle, mais n'est ensuite pas toujours automatisée dans un outil de DAO. Même quand ce travail est fait, entre le développement de la théorie et sa mise en application pratique dans un outil, il s'écoule souvent beaucoup trop de temps. Raccourcir ce processus permettrait d'accélérer à la fois la recherche en DAO et sa dissémination dans le monde industriel puisque la réactivité aux besoins nouveaux est un point crucial à la survie de tout outil.
Le développement d'une plate-forme ouverte construite de façon modulaire et permettant la fabrication d'exécutables dédiés permettrait de remédier à ce problème. Cette plate-forme doit associer la généralité d'un outil source-à-source et la flexibilité de la surcharge d'opérateur. Combiner ces deux approches permet de tirer parti à la fois des informations statiques extraites du programme source et des informations dynamiques présentes lors de l'exécution instrumentée du code. Dans ce contexte, la surcharge peut porter sur le programme dérivé pour calculer des développement de Taylor par exemple, comme sur le programme original pour construire la trace de l'exécution. Aucun outil ne permet cette combinaison pour l'instant, bien qu'elle semble très prometteuse.

Le développement d'une telle plate-forme doit se faire de façon indépendante du front-end et du back-end nécessaires à tout outil source-à-source. De cette façon l'outil résultat reste à la pointe de la recherche puisqu'il profite à la fois des avancées du front-end et du back-end, et de celles réalisées sur la plate-forme elle-même. De plus, l'association d'une telle plate-forme et de front-ends et back-ends dédiés permet de dériver différents langages source. Utiliser le front-end et le back-end d'un compilateur existant faciliterait la

construction d'outils industriels de DAO en phase avec l'état de l'art, alors que l'intégration à une plate-forme de développement d'outil de compilation (SUIF) permettrait d'obtenir un outil de recherche dans lequel les études théoriques seraient mises en oeuvre immédiatement.
Le front-end et le back-end choisis doivent permettre la représentation de programmes sous une forme similaire aux compilateurs : graphe d'appel de sous-programmes, arbre de syntaxe abstraite, table des symboles, graphe de flot, etc.

La plate-forme en elle-même peut être décomposée en six modules indépendants :

un module d'analyse statique qui permet l'implantation d'analyses sur la représentation standard des programmes. Pour permettre une implantation rapide, des algorithmes génériques de propagation d'information en avant ou en arrière par rapport au flot d'informations, avec ou sans point fixe, doivent être étudiés et développés. Ce module doit permettre le bouchonnage : assurer la cohérence des analyses à partir d'une description formelle des sous-programmes dont le source n'est pas disponible. Les analyses standards (propagation de variables actives, de valeurs nécessaires, etc) peuvent être implantées pour permettre la définition rapide d'algorithmes de dérivation. Au fur et à mesure de l'utilisation de la plate-forme, d'autres analyses seront développées pour l'optimisation des programmes dérivés.

un noyau de dérivation qui permet la dérivation de programmes multi-niveaux : la représentation de programmes multi-niveaux, la dérivation des blocs, la composition des blocs dérivés. La représentation de programmes par une hiérarchie de blocs, déclarée par l'utilisateur ou calculée sur le programme, est une extension de la représentation standard dans laquelle les blocs sont des sous-programmes. Les paramètres de dérivation sont associés à chaque bloc de programme, pour permettre la prise en compte les cas particuliers tels que les boucles itératives ou parallèles. La modularité des algorithmes qui permet leur combinaison est basée sur une division du travail bien formalisée par des règles simples. Des exemples de ces règles sont : *(1)* la dérivée d'un bloc est responsable du traitement de sa trace locale et du calcul de ces dérivées propres, *(2)* la composition de la dérivée des blocs est responsable de l'ordonnancement des blocs dérivés et du traitement de la trace globale. De telles règles doivent être étudiées pour assurer la composition cohérente des algorithmes et maximiser la possibilité de combinaison. Les algorithmes standards de dérivation doivent être implantés pour définir le comportement par défaut de l'outils.

une bibliothèque interne de gestion de la trace ou de toute autre fonctionalité nécessaire à l'implantation des algorithmes de dérivation qui présente différentes implantations. Chaque outil automatique et cha-

que équipe a développé sa propre bibliothèque de gestion de la trace par sauvegarde : en mémoire ou sur fichier, avec ou sans bufferization, etc. Le choix de l'implantation à utiliser pour une application donnée dépend de la longueur de la trace, de la stratégie de dérivation et de la machine cible choisies. Pour satisfaire tous les besoins, il faut donc fournir une grande variété de ces implantations groupées dans un module particulier.

une bibliothèque externe qui propose des descriptions formelles et des dérivées pré-définies pour les sous-programmes de bibliothèques standard. Les bibliothèques numériques (BLAS, NAG) ou les bibliothèques de parallélisation (MPI, PVM) sont de plus en plus utilisées dans les programmes industriels. Le source de ces bibliothèques n'est en général pas fourni, il est donc fondamental de fournir leur description formelle (bouchon) pour permettre la dérivation cohérente des programmes les utilisant. Il est tout aussi important de fabriquer une fois pour toutes des dérivées performantes pour les sous-programmes qu'elles contiennent.

un module de calcul de complexité qui permet l'estimation de la complexité pratique des différents programmes dérivés constructibles, ainsi que les tests automatiques associés.
La complexité pratique d'un programme dérivé peut être calculée à partir des paramètres élémentaires de complexité du programme et de la complexité pratique de l'algorithme de dérivation choisi. Les paramètres élémentaires de complexité de chaque bloc peuvent être évalués par l'instrumentation du programme original (surcharge d'opérateur).

un module de génération de tests des programmes dérivés. Lorsqu'un programme est dérivé entièrement automatiquement, il est cohérent et donc correcte. Par contre s'il se compose de modules générés et de modules écrits à la main ou extraits de bibliothèques, cette cohérence n'est plus assurée. Il est donc important de pouvoir tester les programmes dérivés par comparaison entre plusieurs dérivées ou par comparaison aux différences finies. Ces programmes de test doivent être construits automatiquement pour réaliser des tests au niveau du programme (global), du bloc, de l'instruction (local), ...

3.2 Problèmes ouverts

Dans cette section, je présente des problèmes de recherches incontournables pour mettre en application la DAO sur des programmes industriels.

Nous avons vu qu'une grande variété d'algorithmes de DAO pouvait naître d'une vision des programmes plus proche de leur structure réelle. Certains problèmes doivent être résolus au préalable pour faciliter la mise

en application de ces nouveaux algorithmes.

1. L'augmentation du nombre de programmes dérivés constructibles à partir d'un même programme original incitent au développement de méthodes de choix automatique du programme le plus efficace, mais aussi de structuration optimale.
2. Pour une stratégie donnée, nous avons vu que certaines optimisations du programme dérivé permettait d'améliorer sensiblement l'efficacité du programme dérivé. Les optimisations que j'ai présentées sont basées sur une analyse statique du programme, l'ajout d'informations dynamiques peuvent certainement l'améliorer encore cette efficacité. Ces calculs supplémentaires effectués lors de l'évaluation des valeurs dérivées doivent être minimaux pour ne pas ralentir le processus complet.
3. Jusqu'à présent les outils de DAO excluent de leur champ d'application les programmes qui utilisent des alias, mais cette restriction ne peut être conservée car de plus en plus de programmes numériques les utilisent. La dérivation doit donc être ré-étudiée sous l'hypothèse que certains chemins du programme original sont aliasés.

De plus, les programmes industriels qui seront développés dans les années à venir sont différents des programmes actuels pour plusieurs raisons. Les programmes industriels deviennent de plus en plus volumineux et de structure complexe : ils sont couplés entre eux, se composent de modules écrits dans différents langages, et finalement utilisent la parallélisation pour rester utilisables en pratique. Toutes ces extensions des programmes doivent être prises en compte par les outil de DAO pour conserver une bonne adéquation entre les outils et les programmes cibles.

3.2.1 Dérivée optimale

L'augmentation du nombre de programmes dérivés constructibles nécessite certainement la définition d'outils permettant de prévoir et de comparer leurs complexités pratiques pour un programme donné. Il est impossible de faire ce travail à la main, un utilisateur ne peut donc pas choisir le meilleur programme dérivé de son programme avant de l'avoir construit et testé. J'ai défini quelques caractéristiques du programme original à partir desquelles la complexité des différentes dérivées constructibles est extrapolée.

Le découpage du programme original en blocs influence énormément la complexité pratique des dérivées par l'intermédiaire de la profondeur et de la largeur de l'arbre d'appel qu'il induit. De façon générale, les algorithmes de DAO transposent la structure du programme original au programme dérivé. Pour l'instant, il n'existe que deux niveaux (ou types de blocs) utilisés automatiquement par les outils de DAO : l'instruction, le sous-programme. L'utilisateur peut forcer les outils à reconnaître certains blocs particuliers permettant l'application d'algorithmes optimisés : boucles parallèles, boucles

itératives, parties de code linéaires, etc. La recherche automatique de ces blocs particuliers permettant un traitement optimisé serait un apport important mais difficile à mettre en oeuvre car suppose la recherche statique d'informations sémantiques.

Les blocs dérivés sont évalués dans un ordre comparable (le même ou l'inverse) à l'ordre original. Ces blocs dérivés peuvent certainement être re-ordonnés de manière à rapprocher des instructions partageant des sous-expressions (minimiser les calculs) ou à augmenter la localité de la trace (minimiser la mémoire). Par exemple, si deux blocs de calcul B_1 et B_2 peuvent être exécutés en parallèle dans le programme original, il est possible de calculer leurs adjoints $B_1^\star$ et $B_2^\star$ en parallèle. Sous cette hypothèse, B_1 et $B_1^\star$ d'une part et B_2 et $B_2^\star$ d'autre part peuvent être rapprochés pour rendre les traces locales (voir [HFH01]).

De façon générale, si on re-structure le programme dérivé à partir de son équivalent straight-line, il est possible de construire un programme dérivé plus efficace que la transposition exacte de la structure originale. Ce problème peut être traité comme le re-ordonnancement du programme dérivé. Le re-ordonnancement n'est pas simple en lui-même et se complique ici par le fait que l'ordre doit être décidé *à priori*: il faut prévoir la dérivée optimale à partir du programme original et des informations statiques qui peuvent en être extraites. Cette solution statique doit être rapprochée de la solution dynamique (ou runtime) développées dans les travaux [Nau01] sur l'élimination optimale des arcs dans le graphe de calcul. La recherche d'une dérivée optimale n'a jusqu'à présent été considérée que sur des cas particuliers mais il est important d'en faire une étude exhaustive.

3.2.2 Optimisation statique et/ou dynamique

J'ai montré Section 2.3 que l'analyse statique permettait d'extraire des informations fondamentales à l'optimisation du programme dérivé. Cette technologie a l'avantage de pouvoir être utilisée au temps de la compilation et peut donc être coûteuse en temps si elle permet d'accélérer l'évaluation des dérivées elles-même. Mais pour cette même raison, l'information calculée correspondent au cas le pire c'est à dire à la combinaison (maximum, union) de l'information élémentaire calculée pour toutes les exécutions possibles. Cette information est une approximation supérieure de l'information réelle, elle est donc correcte mais peut-être raffinée pour une exécution particulière.

A la différence des outils de compilation, la DAO crée un programme dérivé qui exécute au moins une fois toutes les instructions originales. Pour améliorer les résultats des analyses statiques, il est donc possible d'utiliser des informations dynamiques extraites du programme original. Le programme dérivé pourrait par exemple calculer le sous-ensemble des variables statiquement actives, réellement (dynamiquement) actives, de manière à supprimer des opérations inutiles. Une telle combinaison permettrait de tirer

au mieux partie des informations extraites du programme original : qu'elles soient dynamiques ou statiques. Ceci pourrait être réalisé par la génération par un outil source-à-source d'un programme dérivé surchargé. La difficulté de cette approche est la contrainte de temps puisqu'il faut que l'évaluation de ces informations supplémentaires ne ralentisse pas l'évaluation des dérivées.

Aucun outil existant ne réalise cette combinaison. En effet, pour l'instant les deux types d'implantation sont souvent opposées au lieu d'être associée.

3.2.3 Dérivation en présence d'alias

Les outils standards de DAO font l'hypothèse forte que les programmes qu'ils traitent n'utilisent pas d'alias (`TAMC [Gie97]`, `Odyssée [FP98]`, `ADIFOR [BCC+92, BCK+98]`). Pour vérifier cette hypothèse, ils considèrent que les programmes écrits vérifient la norme ou ils excluent les instructions qui permettent de les créer. Malheureusement, il est possible de créer des alias sans utiliser les instructions standards pour les créer. Par exemple, en `Fortran 77`, il est possible de créer des alias par appel de sous-programme comme je l'ai montré dans la Publication 39 référencée page 78.

Il n'est pas nécessaire de connaître les alias si les variables sont protégées et si aucune optimisation statique n'est effectuée. Ses deux contraintes ne permettent pas de construire un programme dérivé efficace.
Dans l'optique d'une extension du domaine d'applicabilité de la DAO, la seule solution possible est donc la prise en compte des alias lors de la dérivation. La prise en compte des alias pose quatre problèmes différents : *(1)* le calcul des alias sur le programme original, *(2)* l'adaptation des algorithmes d'analyse statique pour calculer les informations modulo ces alias, *(3)* la dérivation des instructions de calcul des pointeurs (souvent à valeur entière), *(4)* la dérivation des instructions de calcul sur des chemins aliasés.

Le calcul précis des alias *(1)* est en lui-même un problème complexe. Ne pas calculer les chemins aliasés oblige (comme dans `ADIC [BRM97]`) à considérer que toute variable est aliasée à toute autre variable visible au même point de programme. Si le calcul précis des chemins aliasés est infaisable, il est possible de détecter les chemins non-aliasée ou de tester dynamiquement les alias réels. Mais la détection de tous les alias reste la seule solution à long terme.
Les algorithmes d'analyse statiques *(2)* peuvent être adaptés en considérant les classes d'équivalences modulo la relation d'alias : un chemin est actif si un de ses alias est actif, un chemin est nécessaire si un de ses alias est nécessaire. Si aucune analyse précise d'alias n'est réalisée, toute variable réelle sera active et nécessaire ce qui aboutira à un programme dérivé souvent inutilisable.
La dérivation en mode direct des instructions du programme original *(3)* qui calculent des pointeurs est simple si elles sont reconnues statiquement. La difficulté vient du typage car ces instructions opèrent des calculs en nombre entier et donc passifs pour la DAO. Par exemple $pt = malloc(15)$ doit être

dérivé en $pt' = malloc(15)$ en mode direct alors que la fonction *malloc* calcule un entier à partir d'un entier. La dérivation en mode inverse de ce type d'instructions est encore plus compliquée : un *malloc* est dérivé en un $free$ et vice versa.
Le problème *(4)* est lui aussi plus simple en mode direct qu'en mode inverse. La dérivée en mode direct d'une affectation est la même qu'on prenne en considération ou non les alias des variables qu'elle lit et écrit. Par contre, en mode inverse, la dérivée d'une instruction dépend de ces alias. Il semble qu'il n'existe aucune façon d'écrire la dérivée d'une instruction qui soit correcte modulo tout pattern d'alias entre ses variables.
Chacun de ses sous-problèmes est un sujet de recherche à part entière et seule la combinaison de leurs résultats peut amener une solution général et efficace.

3.2.4 Dérivation et parallélisation ou couplage

Le couplage ou la parallélisation de programmes pose les mêmes problèmes par rapport à la DAO. Ces problèmes sont dus à l'existence de plusieurs exécutables, au fait que l'exécution du programme ne dépend pas uniquement des données d'entrée, et à la transmission de données d'un exécutable à l'autre. L'existence de plusieurs exécutables implique qu'il existe plusieurs exécutions différentes qui contribuent au résultat final. Chaque exécution prise indépendamment n'est différente d'une exécution séquentielle que par les instructions de communication qu'elle contient. Je présente ici comment tracer, dériver et optimiser ces instructions de communication.

La trace d'une instruction de communication contient tous ses paramètres : le type de communication, l'origine, la destination, le type et la valeur transférée s'il y a lieu. Il est à noter que les données peuvent être re-formatées entre l'exécutable d'origine et celui de destination, ce qui complique la lecture et l'écriture des données dans la trace.
Si les communications sont bloquantes, l'ordre d'exécution est connu statiquement, par contre si les communications sont non-bloquantes leur ordre d'exécution est connu uniquement dynamiquement. L'ordre d'évaluation de ces instructions doit donc lui aussi être conservé dans une trace spécifique.

La transmission des données entre les exécutables doit être dérivée pour transmettre les données dérivées correspondantes. Les transmissions de données peuvent être modélisées comme des lectures/écritures de variables fictives. Une transmission de donnée peut être dérivée en une transmission simultanée de la valeur originale et de la valeur dérivée ou en deux transmissions séparées. En mode inverse, les transmissions doivent être inversées dans le temps, et transposées dans l'espace (origine et destination inversée) puisque une écriture devient une lecture et vice-versa.

L'analyse statique des variables actives ou nécessaires indispensable à la construction d'une dérivée optimisée est très complexe sur un programme

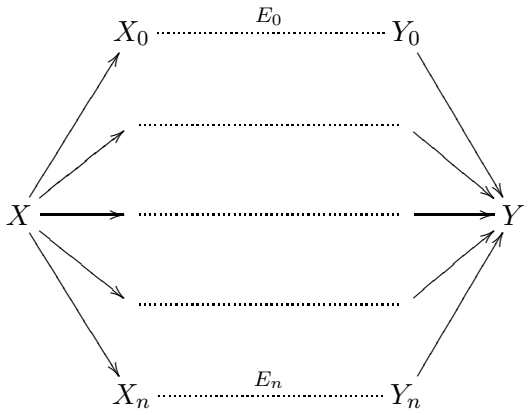


FIG. 3.1 – *Schéma de partitionnement*

parallèle ou couplé. La solution la plus simple est d'éviter cette analyse et d'associer à chaque donnée transmise une donnée dérivée. Cette solution n'est pas optimale puisque la donnée dérivée peut être nulle, le coût des communications est alors doublé sans raison. Cette approximation conduit toute valeur transmise à être active après transmission quel que soit son état avant, ce qui par propagation des variables actives force des calculs dérivés inutiles. Cette simplification n'est donc pas applicable à un programme de taille industrielle. La modélisation des transmissions de données par des lecture/écriture de variables fictives permet un calcul effectif des transmissions actives par analyse statique. Cette analyse est compliquée par la présence de communications non-blocantes dont le moment d'exécution n'est connu statiquement que par un intervalle de temps entre deux barrières.

Chaque phase de la dérivation pose des problèmes particuliers qu'il faudrait formaliser et résoudre de façon générale pour éviter de cherche s solutions pour chaque librairie nouvelle.

3.2.5 Dérivation de partitionnement de données

Je présente ici un problème qui apparaît effectivement lors de la dérivation de programmes industriels parallèles. Comme présenté dans Figure 3.1, le calcul de Y à partir de X par n exécutables se fait en trois phases : *(1)* le partitionnement de la donnée X en n parties X_1,...,X_n, *(2)* le calcul par l'exécutable E_n de la solution partielle Y_n à partir de la donnée partielle X_n, *(3)* la construction de la solution totale Y à partir des solutions partielles Y_1,...,Y_n. Le programme numérique qui calcule Y n'est pas construit suivant ce schéma pour des raisons d'efficacité. La répartition des donnée est faite une fois pour toute et stockées sur fichiers, les programmes parallèles lisent ses fichiers et construisent la solution. La partie du programme sur laquelle

porte la DAO ne contient donc pas en général l'algorithme de partitionnement, alors qu'il contient l'algorithme de construction de la solution à partir des solutions calculées par chaque exécutable.

Ceci implique qu'en mode direct il manque le partitionnement de la direction de dérivation dX en $dX_0 \cdots dX_n$, et la construction de l'adjoint dX à partir de $dX_0 \cdots dX_n$ en mode inverse. En mode direct comme en mode inverse, seule la connaissance de l'algorithme de partitionnement peut permettre le calcul des dérivées. Si le programme de partitionnement ne peut être traité par dérivation de programme, est-il possible d'en faire une description formelle qui permettrait de générer automatiquement le morceau de programme dérivé manquant ?

3.2.6 Dérivation multi-langages

De plus en plus de programmes sont écrits en plusieurs langages pour des raisons d'efficacité par exemple (C est plus efficace que `Fortran 77` pour gérer l'allocation dynamique) ou des raisons historiques (des modules existants performants ou sûrs sont ré-utilisés pour un nouveau développement). De même, la résolution de problèmes complexes est faite par le couplage de programmes indépendants potentiellement écrits dans des langages différents.

Il n'est pas possible de dériver ces programmes multi-langages en utilisant un unique outil de DAO puisque chaque outil existant est dédié à un langage. Soit un programme écrit en plusieurs langages tels qu'il existe un outil de DAO pouvant dériver chaque module. La première solution pour dériver le programme complexe est de dériver chaque module séparément et de composer la dérivée totale à la main. Cette approche est possible mais ne permet de calculer qu'une dérivée maximale puisque chaque module calcule une dérivée totale. Le programme dérivé ainsi obtenu sera le plus souvent inutilisable car trop gourmand en temps et en espace. Des optimisations restent faisables à la main, mais sont difficilement contrôlables ce qui peut aboutir à un programme dérivé incohérent.

Un méta-outil de DAO est donc indispensable à la généralisation de l'utilisation des outils de DAO aux programmes écrits en plusieurs langages. Cet outil généralise le bouchonnage : il demande à chaque outil dédié des informations sur son module associé. Puis il compose ces informations en considérant chaque module comme un méta-bloc. Ensuite il demande aux système dédiés la dérivée de chaque module et génère le programme qui compose ces dérivées pour calculer la dérivée totale. Le protocole client-serveur par lequel l'outil maître communique avec les outils dédiés est à définir. De cette manière il est possible d'utiliser des outils très différents de manière uniforme. Il est clair que l'intégration de ce mécanisme à la plate-forme décrite en Section 3.1 est simplifié par le fait que le même outil peut en changent de front-end dériver différents langages. Les informations

nécessaires à la dérivation d'une module sont donc les mêmes pour tous les langages et le protocole est alors très simple.

Chapitre 4

Parcours professionnel

FAURE Christèle
Née le 4 Mars 1964 à Toulouse (31),
Situation de famille : Deux enfants (1992, 1995),
12 Traverse du Collet, 06650 Le Rouret.
Tel personnel : 04 93 09 45 48.

Compétences :

- Recherche et développement d'outils logiciels.
- Élaboration et suivi de contrats industriels de *R&D*.
- Encadrement d'équipes et formation d'ingénieurs.
- Promotion internationale d'outil logiciel.

Profession actuelle (depuis Septembre 2000) :

Ingénieur *R&D* pour *Polyspace Technologies*.
Application de l'interprétation abstraite à la détection automatique d'erreurs d'exécution basée sur des techniques d'interprétation abstraite.

4.1 1994-2000 : Chercheur contractuel

J'assure à la fois la recherche et le développement de l'outil de Différentiation Automatique (DA) appelé `Odyssée [FP98]` ainsi que sa promotion internationale. `Odyssée [FP98]` est maintenant reconnu dans le monde industriel, mais aussi dans le monde académique comme l'un des trois outil de DA les plus performants.

Cette mission se décompose en trois actions menées en parallèle.

4.1.1 Développement d'un outil industriel de DA

Objectif à atteindre : Réaliser pour EDF un outil applicable à des programmes industriels de plus en plus généraux.

Réalisations : Je développe l'outil `Odyssée [FP98]`.

1994-1996 Je participe à la reprise de la base logiciel déjà existante.

1996-2000 Je deviens responsable de la recherche et du développement de l'outil.

Résultats : EDF est satisfait des résultats obtenus sur les programmes cibles. Le client a renouvelé le contrat durant six ans et a décidé d'utiliser `Odyssée [FP98]` de façon autonome.

L'intérêt industriel et académique pour `Odyssée [FP98]` est démontré par son utilisation :

20 sites à l'étranger (DERA, NCAR, Los Alamos Nat. Laboratory, NCEP, Alenia, etc),

20 sites en France (IFP, Orstom, CEA Saclay, etc),

et de futures utilisations à NASA Langley Research Center, Mitsubishi Heavy Industries.

4.1.2 Définition et coordination d'une Action de *R&D*

Objectif à atteindre : Pour permettre une utilisation générale de l'outil, l'espace mémoire utilisé par le programme adjoint généré automatiquement doit être diminué. J'ai identifié ce problème sur des programmes opérationnels d'EDF.

Réalisations : J'ai soumis à la Direction Scientifique de l'INRIA une action de *R&D* d'`Odyssée [FP98]` visant à résoudre ce problème (*(a)* Définition des objectifs techniques, *(b)* Choix de l'équipe pluridisciplinaire, *(c)* Spécification d'un calendrier et d'un budget sur 30 mois).

Le projet a été accepté et s'est déroulé de **Juin 1996-1999**. J'ai coordonné l'action jusqu'à son terme (*(a)* Coordination de l'équipe, *(b)* Recrutement de post-doctorants, *(c)* Organisation de réunions d'avancement, *(d)* Gestion du calendrier et du budget (900 KF)).

Résultats : Gain de 75% en espace mémoire utilisé par le programme généré sur des programmes opérationnels. Pour parvenir à ce résultats, une méthode issue de la compilation a été implémentée dans une version prototype d'`Odyssée [FP98]`.

4.1.3 Animation de la recherche

– J'organise des manifestations internationales :

2000 : "AD2000: 3rd International Conference on AD" à Nice (100 personnes).

1998 : Une session DA à IMACS'ACA, Prague, Czech Republic (20 personnes).

1997 : Une réunion internationale des utilisateurs d'`Odyssée [FP98]` (35 personnes).

1996 : Une session DA à IMACS'ACA, Maui, Hawai (20 personnes).

– J'encadre ou co-encadre des étudiants :

2000 : ESSI 3 (6 mois), Karine Berra et Christophe Prevot,
Développement d'un site Web Didactique présentant les travaux de recherche de l'INRIA de façon synthétique et à travers des publications grand publique.

1999 : Post-Docs (1 an), Patrick Dutto et Stefka Fidanova : Dérivation de programme parallèle et parallélisation de programme dérivé.
Uwe Naumann : Optimisation du programme dérivé en mode inverse par suppression des sauvegardes inutiles.

1999-1996 : Doctorant, Mohamed Tadjouddine, Analyse de Dépendances de Jacobiennes Creuses pour la Différentiation Automatique [Tad99].

1997 Post-Doc (6 mois), Mohammed Ghémirès, Réalisation d'un programme dérivé en mode adjoint utilisant le *mode séparé* pour diminuer le nombre de re-calcul des sous-programmes.

1997 : Stage DEA (3 mois), Serge Fantino, Application de la Différentiation Automatique à l'optimisation surfacique.

1997 : Post-Doc au CNES (18 mois), Sandrine Philoreau, Utilisation du Calcul Formel (CIRCE : package Maple) pour la vérification des exigences fonctionnelles d'instruments d'optique et la réalisation de bilans de performance.

1995 : Stage DEA (3 mois), Arnaud Vallois, Simplification d'expressions Formelles Booléennes.

– Je réalise deux cours :

Licence d'informatique Architecture du système de calcul formel Maple (40h), langage Le-Lisp (50h), responsable J. Chazarain, 1989, 1990.

DEA Mathématique et Informatique La Différentiation Automatique, un exemple de transformation de programme (9h), 1995, 1996, 1997.

4.2 Liens Recherche / Industrie

– J'élabore, réalise et/ou encadre des contrats industriels (Dassault, Aérospatiale, Nag, Essilor, Alenia).

DECISION (EP 25 058), 1997-2000 ESPRIT, HPCN, Integrated Optimization Strategies for Increased Engineering Design Complexity,
NAG, DASSAULT AVIATION, MESSET, NOKKA TUME, INRIA, VTT, University of Jyvaskyla. Développement d'une plate-forme d'optimisation offrant un choix d'optimiseurs basés sur des technologies variées (génétique, hybride) et ouverts au calcul automatique de gradients par Différentiation Automatique.

AD-CAPE (24023), 1998-1999 ESPRIT IV, Software Technologies, Trial Applications Automatic Differentiation for Computer Aided Process Engineering, SIMULOG, BELSIOM, PROSIM, BP, ELf, SCHERING. Application d'`Odyssée [FP98]` aux programmes CAPE. Mon rôle dans ce projet a été la mise à disposition et l'aide à l'utilisation d'`Odyssée [FP98]`.

GENIE, 1997-1998 Contrat Industriel, DASSAULT AVIATION, AEROSPATIALE, INRIA, SIMULOG. Ce projet comportait de nombreux thèmes. Il s'est agi pour nous d'étendre d'`Odyssée [FP98]` aux *programmes parallèles MIMD*.

Alenia, 1988 Contrat Industriel, ALENIA AERONAUTICA. Etude de l'apport de la différentiation automatique à l'optimisation de forme d'un profile (2-D) d'aile d'avion dans des écoulements de fluide visqueux ou non.

– Je réalise des formations d'ingénieurs :

1998 : École d'été de DA au CNES (Cours, 12h).

1997 : École d'été du CEMRACS "Shape Optimization and Automatic Differentiation" (Cours, 3h).

1996 : École d'été INRIA "Techniques de développement pour programmes numériques" (Cours, 3h).

1996 : École d'été d'analyse numérique CEA - INRIA - EDF (TD, 60h).

4.3 Formation : Mathématique et Informatique

1992-1994 : Post doc. en Grande Bretagne (14 mois) à l'Université de Bath.

1992 : *Doctorat de Mathématique* (UNSA) "Simplification en Calcul Formel"

1988 : DESS Informatique et Sciences de l'Ingénieur et DEA Mathématique et Informatique (UNSA).

4.4 Liste de publications

Édition de Proceedings

1. [CFG+01] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation of Algorithms: From Simulation to Optimization*. Computer and Information Science, Springer, New York, 2002.
2. C. Faure and U. Naumann, editors. *AD'2000 abstracts: From Simulation to Optimization*. INRIA, Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France), 2000.
3. C. Faure, editor. *Automatic Differentiation for Adjoint Code Generation, Proceedings of IMACS'ACA, Automatic Differentiation Session*, Rapport de recherche 3555, INRIA, November 1998.

Rédaction de mémoires

4. C. Faure. *Quelques aspects de la Simplification en Calcul Formel*. PhD thesis, Université de Nice Sophia-Antipolis, 1992.

Dans des journaux internationaux avec comité de lecture

5. C. Faure. Automatic Differentiation Platform: Design. *Mathematical Modelling and Numerical Analysis*, 36(5):783-792, 2002.
6. A. Griewank and C. Faure. Reduced Functions, Gradients and Hessians from fixed-point iterations for state equations. *Numerical Algorithms*, 30:113-139, 2002.
7. C. Faure. Adjoining strategies for multi-layered programs. *Optimisation Methods and Software*, 17(1):129-164, 2002.
8. D. Elizondo, B. Cappelaere, and C. Faure. Automatic versus manual model differentiation to compute sensitivities and solve non-linear inverse problems. *Computers & Geosciences*, 28(3):47–64, 2002.
9. C. Faure and I. Charpentier. Comparing Global Strategies for Coding Adjoints. *Scientific Programming*, 9(1):1–11, 2001.
10. C. Faure. Generating Adjoints of Industrial Codes with Limited Memory. *Flow, Turbulence and Combustion*, 65(3-4):453–467, 2000.
11. J.H. Davenport and C. Faure. The "unknown" in computer algebra. *Programming and Computer Software*, 20:1–5, 1994. Traduction from (de) Programmirovanie.
12. J.H. Davenport and C. Faure. The "unknown" in computer algebra (english. russian summary). *Programmirovanie*, 1:4–10, 1994.
13. C. Faure, A. Galligo, J. Grimm, and L. Pottier. The extensions of the sisyphe computer algebra system : `ulysse` and `athena`. In J. Fitch,

editor, *Design and Implementation of Symbolic Computation Systems*, volume 721 of *LNCS*, pages 44–55. Springer-Verlag, 1992.

Dans des proceedings de conférences avec comité de lecture

12. B. Cappelaere, D. Elizondo, and C. Faure. `Odyssée [FP98]`- versus Hand- Differentiation of a Terrain-Modeling Application. [CFG+01], Chapter 7, pages 75-83.
13. C. Faure and U. Naumann. Minimizing the Tape Size. [CFG+01], Chapter 34, pages 293-299.
14. E. Soulié, C. Faure, and T. Berclaz. Electron Paramagnetic Resonance, Optimization and Automatic Differentiation. [CFG+01], Chapter 10, pages 99-109.
15. C. Faure, P. Dutto, and S. Fidanova. `Odyssée [FP98]` and parallelism : Extension and Validation. In *Procceedings of The 3rd European Conference on Numerical Mathematics and Advanced Applications, Jyväskylä,*
Finland, July 26-30, 1999. pages 478-485, World Scientific, 2000.
16. D. Elizondo, B. Cappelaere, and C. Faure. Using the Odyssée automatic differentiation tool for inverse modeling. *Geophysical Research Abstracts*, 1(2):306, 1999. (24th General Assembly of the EGS - Hydrology, Oceans and Atmosphere, The Hague, April 1999).
17. M. Tadjouddine, C. Faure, and F. Eyssette. Sparse Jacobian Computation in Automatic Differentiation by Static Program Analysis. In G. Levi, editor, *Static Analysis*, volume 1503 of *Lecture Notes in Computer Science*, pages 311–326. Springer-Verlag, September 1998.
18. C. Faure and C. Duval. Automatic differentiation for sensitivity analysis. A test case. In K. Chan, S. Tarantola, and F. Campolongo, editors, *Proceedings of Second International Symposium on Sensitivity Analysis of Model Output (SAMO'98)*, volume 17758. EN, Luxembourg, 1998.
19. C. Faure. Splitting of Algebraic Expressions for Automatic Differentiation. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation : Techniques, Applications, and Tools*, pages 117–127. SIAM, Philadelphia, Penn., 1996.
20. C. Duval, P. Erhard, C. Faure, and J.C. Gilbert. Application of the automatic differentiation tool Odyssée to a System of Thermohydraulic Equations. In J.-A. Désidéri, P. Le Tallec, E. Oñate, J. Périaux, and E. Stein, editors, *Proc. of ECCOMAS'96*, volume Numerical Methods in Engineering'96, pages 795–802. John Wiley & Sons, September 1996.
21. C. Faure. A meta simplifier. In S. Watanabe and M. Nagata, editors, *Proceedings of ISSAC*, page 290. ACM Press, 1990.

Dans des proceedings de conférences sans comité de lecture

22. A. Griewank and C. Faure. Piggyback Differentiation and Optimization. To Appear in *Lecture Notes in Computer Science*, 2000.
23. C. Faure. Automatic generation of adjoint codes. In F Bertin, editor, *Journées scientifiques sur l'assimilation d'observations de la chimie atmosphérique dans les modèles*, pages 23–25. CNFGG, Académie des Sciences, 1999. available at http://www.omp.obs-mip.fr/cnfgg/.
24. C. Faure. Oscillation in the derivatives. In C.H. Bischof, P. Eberhard, and P.D. Hovland, editors, *Second Argonne Theory Institute on Differentiation of Computational Approximations to Functions*, volume ANL/MCS-TM-236, page 13. Argonne National Laboratory, June 1998.

Rapports et publications internes

24. C. Faure, J.H. Davenport, and H. Naciri Multi-Valued Computer Algebra. Rapport de recherche 4001, INRIA, September 2000.
25. D. Elizondo, C. Faure, and B. Cappelaere Automatic- versus Manual-differentiation for non-linear inverse modeling. Rapport de recherche 3981, INRIA, July 2000.
26. C. Faure, E. Soulié, and T. Berclaz. Résonance paramagnétique électronique, optimisation et différentiation automatique. Rapport de recherche 3907, INRIA, March 2000.
27. C. Faure and I. Charpentier. Comparing Strategies for Coding Adjoints. Rapport de recherche 3811, INRIA, November 1999.
28. C. Faure. Adjoining Strategies for Multi-Layered Programs. Rapport de recherche 3781, INRIA, October 1999.
29. C. Faure and P. Dutto. Extension of Odyssée to the MPI library - Reverse mode-. Rapport de recherche 3774, INRIA, October 1999.
30. C. Faure and P. Dutto. Extension of Odyssée to the MPI library - Direct mode-. Rapport de recherche 3715, INRIA, June 1999.
31. C. Faure. Le gradient de THYC3D par Odyssée. Rapport de recherche 3519, INRIA, October 1998.
32. C. Faure and Y. Papegay. `Odyssée [FP98]` User's Guide. Version 1.7. Rapport technique 0224, INRIA, September 1998.
33. C. Faure and C. Duval. Application d'Odyssée au logiciel de thermohydraulique THYC. Rapport de recherche HT-13/97/038/A, EDF/DER, December 1997.
34. C. Faure and Y. Papegay. `Odyssée [FP98]` Version 1.6. The language reference manual. Rapport technique 0211, INRIA, November 1997.
35. F. Eyssette, J.-C. Gilbert, C. Faure, and N. Rostaing-Schmidt. Applicabilité de la différentiation automatique à un système d'équations aux dérivées partielles régissant les phénomènes thermohydrauliques

dans un tube chauffant. Rapport de recherche 2795, INRIA, February 1996.

36. F. Eyssette, J.-C. Gilbert, C. Faure, and N. Rostaing-Schmidt. Applicabilité de la différentiation automatique à un système d'équations aux dérivées partielles régissant les phénomènes thermohydrauliques dans un tube chauffant. Rapport de recherche HT-13/96/001/A, EDF/DER, February 1996.
37. C. Faure. Objets conditionnels et objets inconnus. Rapport de recherche 2298, INRIA, July 1994.
38. C. Faure. Towards a meta simplifier. Rapport de recherche 1402, INRIA, April 1991.

Non publié

39. C. Faure. Compilation aspects of Automatic Differentiation by Examples. In Preparation.

Bibliographie

[BBCG96] M. Berz, C.H. Bischof, G.F. Corliss, and A. Griewank, editors. *Computational Differentiation: Applications, Techniques, and Tools*. SIAM, Philadelphia, 1996.

[BCC+92] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[BCK+98] C. Bischof, A. Carle, P. Khademi, A. Mauer, and P. Hovland. ADIFOR 2.0 User's Guide. Technical Report ANL/MCS-TM-192/CRPC-TR95516-S, Argonne National Laboratory Technical Memorandum and CRPC Technical Report, 1998.

[BRM97] Christian Bischof, Lucas Roh, and Andrew Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software: Practice and Experience*, 27(12):1427–1456, 1997.

[BS83] W. Baur and V. Strassen. The complexity of partial derivatives. *Theoretical Comp. Sci.*, 22:317–330, 1983.

[CFG+01] G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, editors. *Automatic Differentiation of Program: From Simulation to Optimization*. Springer Verlag, 2001.

[CI96] B. Creusillet and F. Irigoin. Exact vs. approximate array region analyses. *Languages and Compilers for Parallel Computing*, August 1996.

[DGH96] S. Dalmas, M. Gaetano, and C. Huchet. A Deductive Database for Mathematical Formulae. In J. Calmet and C. Limongelli, editors, *Design and Implementation of Symbolic Computation Systems, International Symposium, DISCO'96*, volume 1128 of *LNCS*. Springer Verlag, 1996.

[Fas00] FastOpt. *TAF: Transformation of Algorithms in Fortran*, 2000. `http://www.FastOpt.de`.

[FP98] C. Faure and Y. Papegay. Odyssée User's Guide. Version 1.7. Rapport technique 0224, INRIA, September 1998.

[GC91] A. Griewank and G.F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications*. SIAM, Philadelphia, 1991.

[Gie97] R. Giering. *TAMC: Tangent linear and Adjoint Model Compiler, Users manual*, 1997. `http://puddle.mit.edu/~ralf/tamc`.

[Gil92] J-C. Gilbert. Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1:13–21, 1992.

[GJM+96] A. Griewank, D. Juedes, H. Mitev, J. Utke, O. Vogel, and A. Walther. Adol-c: A package for the automatic differentiation of algorithms written in c/c++. *ACM TOMS*, 22:131–167, 1996.

[GK98] R. Giering and T. Kaminski. Recipes for Adjoint Code Construction. *ACM Trans. On Math. Software*, 24(4):437–474, 1998.

[GK01] R. Giering and T. Kaminski. Generating recomputations in reverse mode. In *Automatic Differentiation of Algorithms: From Simulation to Optimization*, pages 283–290. Springer-Verlag, 2001.

[GLVM91] J-C. Gilbert, G. Le Vey, and J. Masse. La Différentiation Automatique de fonctions représentées par des programmes. Rapport de recherche 1557, INRIA, November 1991.

[GPRS96] J. Grimm, L. Pottier, and N. Rostaing-Schmidt. Optimal time and minimum space-time product for reversing a certain class of programs. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Applications, Techniques, and Tools*, pages 95–106. SIAM, 1996.

[Gri92] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in Reverse Automatic Differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[Gri00] A. Griewank. *Principles and Techniques of Algorithmic Differentiation*. SIAM, 2000.

[HFH01] L. Hascoet, Stefka Fidanova, and Christophe Held. Adjoining Independent Computations. In *Automatic Differentiation of Algorithms: From Simulation to Optimization*, pages 299–304. Springer-Verlag, 2001.

[IK87] M. Iri and K. Kubota. Methods of fast automatic differentiation and applications. Research memorandum rmi 87-02, Department of Mathematical Engineering and Instrumentation Physics, Faculty of Engineering, University of Tokyo, 1987.

[Iri84] M. Iri. Simultaneous computation of functions, partial derivatives and estimates of rounding errors, complexity and practicality. *Japan Journal of Applied Mathematics*, 1:223–252, 1984.

[KNC84] K.V. Kim, Yu.E. Nesterov, and B.V. Cherkasskiĭ. An estimate of the effort in computing the gradient. *Soviet Math. Dokl.*, 29:384–387, 1984.

[Kub96] K. Kubota. PADRE2 - FORTRAN precompiler for automatic differentiation and estimates of rounding errors. In M. Berz, C.H. Bischof, G.F. Corliss, and A. Griewank, editors, *Computational Differentiation: Applications, Techniques, and Tools*, pages 463–471, Philadelphia, 1996. SIAM.

[Mor85] J. Morgenstern. How to compute fast a function and all its derivatives, a variation on the theorem of Baur-Strassen. *Sigact News*, 16:60–62, 1985.

[Nau01] U. Naumann. General Elimination Techniques for Cheap Jacobians. In *Automatic Differentiation of Algorithms: From Simulation to Optimization*, pages 247–253. Springer-Verlag, 2001.

[OVB71] G.M. Ostrovskii, Yu.M. Volin, and W.W. Borisov. Uber die berechnung von ableitungen. *Wiss. Z. Tech. Hochschule fur Chimie*, 13:382–384, 1971.

[Saw84] J.W. Sawyer. First partial differentiation by computer with an application to categorial data analysis. *The American Statistician*, 38:300–308, 1984.

[Spe80] B. Speelpening. *Compiling fast partial derivatives of functions given by algorithms*. PhD thesis, University of Illinois, Urbana-Champaign, 1980.

[Tad99] M. Tadjouddine. *Analyse de Dépendances de Jacobiennes Creuses pour la Différentiation Automatique*. PhD thesis, Université de Nice-Sophia Antipolis, 1999.

[Tal91] O. Talagrand. The use of adjoint equations in numerical modeling of the atmospheric circulation. In A. Griewank and G.F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Applications*, pages 169–180. SIAM, 1991.

Index

Chapitre 5

Recherches doctorales et post-doctorales

Les logiciels de Calcul Formel utilisent des moyens divers pour prendre en compte les calculs ayant plusieurs résultats. Mon travail doctoral a eu pour but de spécifier une forme d'expressions "générales" qui permette un traitement unifié de ce type de calculs.

L'une des grandes forces du Calcul Formel est de traiter non pas un problème particulier, mais une famille de problèmes en un seul calcul. De ce fait toute expression entrée dans un système de Calcul Formel contient des variables qui représentent soit les données du problème, soit les paramètres généraux à tous les problèmes de la même famille. On souhaiterait pouvoir définir l'influence de ces paramètres à la lecture du résultat.

L'intégrale suivante fournit un exemple pertinent de ce type de calculs :

$$\begin{array}{rcll} \int \frac{dx}{ax^2+bx+c} & = & \frac{1}{\sqrt{b^2-4ac}} \log \frac{2ax+b-\sqrt{b^2-4ac}}{2ax+b+\sqrt{b^2-4ac}} & (b^2 > 4ac) \\ & = & \frac{2}{\sqrt{4ac-b^2}} \operatorname{atan} \frac{2ax+b}{\sqrt{4ac-b^2}} & (b^2 < 4ac) \\ & = & \frac{-2}{2ax+b} & (b^2 = 4ac) \end{array}$$

On veut pouvoir lire sur le résultat les différentes formes possibles de l'expression en fonction des domaines de valeurs des paramètres. Ceci n'est pas possible dans les systèmes actuels (voir l'article intitulé "Multi-valued results in Computer Algebra" fourni) car pour la plupart ils rendent un résultat unique. Les stratégies qu'ils utilisent sont les suivantes :

minimale le système choisit la forme de la solution,

proviso le système choisit la forme de la solution et mémorise les contraintes,

implicite le système fournit un résultat de forme unique, mais contenant un opérateur multi-valué (`sign`),

explicite le système fournit un résultat multiple sous forme de liste (donc sans les contraintes), ou sous forme de `If`.

Toutes ces stratégies peuvent être unifiées, car sont basées sur une unique forme de calcul : le calcul *conditionnel*. J'ai donc étudié une nouvelle classe d'expressions qualifiées de *conditionnelles* qui permet d'exprimer des résultats multiples en donnant pour chaque valeur son domaine de validité sous forme de contraintes sur les valeurs des paramètres.

Ce travail prolonge mon travail de thèse. En effet, il s'agissait alors d'analyser l'influence des propriétés algébriques (dépendant du type de l'expression) sur la forme des expressions et de définir un mécanisme de calcul utilisant au mieux ces propriétés. Ce premier travail était basé sur du typage "formel" associé à de la réécriture équationnelle.

5.1 Expressions conditionnelles

On définit une expression conditionnelle comme une liste de couples condition/valeur, qui signifie sous telle condition l'expression a telle valeur. La complexité du calcul sur les expressions conditionnelles vient du calcul des conditions car celui sur les valeurs est (mono-valué) déjà existant dans les systèmes. Le calcul essentiel à effectuer sur les conditions est la détection des tautologies et des contradictions qui permet de diminuer la taille des expressions conditionnelles par suppression des couples $False \rightarrow val$. Si on considère que les expressions conditionnelles sont des extensions de leurs domaines de valeurs, on veut que les propriétés soient conservées lors de l'extension, donc que les opérations de ce domaine soient elles aussi étendues.

Les conditions sont des expressions booléennes générales formées des opérateurs booléens courants ($\wedge$,$\vee$ ...) des constantes $True$,$False$ et de "variables" booléennes plus complexes que de simples symboles. Par exemple, pour prendre en compte une condition de la forme $b^2 < 4ac$, il faut définir des conditions élémentaires de la forme $negative(b^2 - 4ac)$ ... Ces conditions expriment des contraintes sur les paramètres de l'expression conditionnelle, par l'intermédiaire de prédicats appliqués à des objets de la même forme que les valeurs. On peut remarquer que ces conditions élémentaires ne sont pas indépendantes contrairement aux variables des expressions booléennes.

La preuve qu'une condition est une tautologie peut se faire par réfutation (prouver que la négation de l'expression est fausse quelles que soient les variables) ou tout autre méthode (BDD ...) en supposant que les conditions élémentaires sont indépendantes. Mais ceci limite fortement la capacité de preuve, par exemple $negative(a) \wedge null(a)$ ne peut être ainsi calculé en $False$.

Pour gérer les dépendances entre conditions élémentaires il faut introduire d'autres informations qui sont propres à chaque ensemble de prédicats ainsi qu'au domaine de valeurs des paramètres. Dans notre exemple, une partie des dépendances entre les conditions élémentaires peut s'exprimer

sous forme de règles de transformation, telles que :

$$\begin{array}{lll} \forall X & negative(X) & \Leftrightarrow \quad \neg positive(X) \wedge \neg null(X) \\ \forall P_i & null(\prod_i P_i) & \Leftrightarrow \quad \wedge_i null(P_i) \\ \ldots & & \end{array}$$

De telles règles existent certainement pour toutes les classes de conditions élémentaires.

Ces règles génériques peuvent être "instanciées" en utilisant les capacités spécifiques du Calcul Formel telles que `solve` ...
Par exemple la deuxième règle est instanciée de la manière suivante :

1. transformer $ab - a$ en $a(b-1)$ par factorisation,
2. utiliser la règle sur le signe des produits pour en déduire que $\forall a,b$ $null(ab-a) \Leftrightarrow null(a) \vee null(b-1)$.

D'autres fonctions pré-définies dans les systèmes de Calcul Formel peuvent être utilisée pour générer ces équivalents des conditions élémentaires (pour cet exemple `solve, eval, sign` ...).

Pour prouver que $null(ab-a) \wedge positive(a)$ est contradictoire, on prouve alors que :

$$(null(ab-a) \wedge positive(a)) \wedge (null(ab-a) \leftrightarrow null(a) \vee null(b-1)).$$

L'utilisation de ces règles allie donc à la fois les capacités propres du calcul formel, et celles du calcul booléen général. Les instanciations effectuées par les systèmes peuvent être coûteuses, elles sont donc conservées dans un contexte qui ne contient donc que des tautologies. On peut remarquer que le choix des instanciations à effectuer (car il ne s'agit pas de tout générer) est très important.

La méthode de réfutation à utiliser doit donc prendre en compte un contexte (contenant des tautologies), ainsi pour prouver que e est contradictoire on prouve que e est contradictoire dans le contexte $Context_e$ associé à e par le système, c'est à dire que $Context_e \wedge e$ contient une contradiction.
Ce contexte peut aussi être utilisé globalement pour permettre à l'utilisateur de le visualiser, et même en ajouter que le système ne sait pas générer.

Dans le calcul d'une expression conditionnelle, un contexte unique est utilisé pour toutes les conditions, ainsi la cohérence des calculs est assurée. Le gain en calculs nécessaires à l'instanciation est donc très important car le scindage d'une formule se fait à partir d'un test unique ($b^2 - 4ac > 0$, $b^2 - 4ac = 0$, $b^2 - 4ac < 0$).

5.2 Application : l'intégrateur d'Axiom

L'intégrateur du système de Calcul Formel `Axiom` a été modifié pour rendre non plus un ensemble de solutions, mais une expression conditionnelle contenant toutes les solutions telles qu'elle sont décrites dans les tables d'intégrales.

Pour permettre cela, le domaine des expressions conditionnelles `ConditionalExpression` (correspondant à la description précédente) a été défini. Pour ce domaine les conditions sont des expressions booléennes `ExpressionCondition` sur les comparaison à zéro de polynômes de paramètres, et les valeurs sont des expressions générales.

Les modifications apportées à l'intégrateur d'`Axiom` pour qu'il retourne non pas la liste de certaines valeurs possibles, mais une expression conditionnelle donnant toutes les valeurs sont de deux ordres : le calcul des conditions pour lesquelles les valeurs étaient préalablement rendues, le calcul des points critiques ainsi que des valeurs qui n'apparaissaient pas dans le résultat standard.

Le calcul des conditions associées aux valeurs rendues a été dans ce cadre assez simple, car elles étaient indiquées dans le programme. Dans l'exemple la discrimination se fait par $\delta = b^2 - 4ac$, l'ancien intégrateur calculait donc les solutions pour $\delta > 0$ et $\delta < 0$.
Le calcul des point critiques a été ajouté : ce sont dans notre cas les solutions de $\delta = 0$ réalisé par `solve`. Pour chaque point critique, la fonction à intégrer est évaluée et l'intégrale calculée par (re-)intégration.

Pour calculer l'intégrale par rapport à x de $a/(x**2 - a**2 + 1)$, notre intégrateur calcule les deux valeurs principales , puis détermine les cas critiques équivalents à $a^4 - a^2 = 0$, c'est à dire $a - 1 = 0$,$a = 0$,$a + 1 = 0$ et calcule la valeur de l'intégrale pour chacun de ces cas. Pour que le système puisse vérifier que l'expression conditionnelle construite est complète, donc que toutes les valeurs possibles sont énumérées, l'intégrateur déclare que $a^4 - a^2 = 0 \Leftrightarrow (a - 1 = 0)$ or $(a = 0)$ or $(a + 1 = 0)$.

Voici alors le calcul de l'intégrale :

```
(4) -> a/(x**2-a**2+1) :: EXPR INT

                a
   (4)   -----------
                2    2
         x  - a  + 1
                Type: Expression Integer
(5) -> integrate(%,x)
                                                    1
   (5)                                  ((a - 1=0) -> - -)
                                                    x
                                           ((a=0) -> 0)
                                                    1
                                        ((a + 1=0) -> -)
                                                    x
                                                        x
                                             a atan(-----------)
                                                    +--------+
                                                    |   2
                                      4    2        \|- a  + 1
                                   ((a  - a <0) -> -------------------)
                                                  +--------+
                                                  |   2
                                                 \|- a  + 1
                                               +------+
                          2    2      | 2             2                2    2
   4    2       a log((x  + a  - 1)\|a  - 1  + (- 2a  + 2)x) - a log(x  - a  + 1)
((a  - a >0) -> ------------------------------------------------------------------)
                                                 +------+
                                                 | 2
                                                2\|a  - 1
                Type: ConditionalExpression Integer
```

La transformation de l'intégrateur a donc été simple. Et les capacités du

calcul sur les conditions a été utilisé pour mémoriser que $null(a^4 - a^2)$ était équivalent à $null(a) \vee null(a-1) \vee null(a+1)$. Ceci serait utilisé dans le calcul de `% + abs (a-1)` par exemple pour ne garder que les valeurs atteignables. Le calcul multi-valué a été utilisé pour le calcul des limites sans révéler de difficultés particulières.

5.3 Conclusion

L'introduction d'objets conditionnels dans un système de Calcul Formel permet non seulement d'exprimer des résultats de manière complète, mais aussi de pousser plus loin les calculs des systèmes de calcul formel.

Par exemple, si $abs(n)$ est transformé en $\begin{array}{ll} n<0 & -n \\ n=0 & 0 \\ n>0 & n \end{array}$, $abs(abs(n))$ est automatiquement (sans introduire de règles spécifique) égal à $abs(n)$, et $abs(n) - abs(-n)$ est automatiquement calculé en 0.

Dans les sections précédentes, nous avons décrit des expressions complètes, c'est à dire telles que les contraintes qu'elles contiennent recouvrent tout le domaine de valeurs des paramètres. Or la manipulation de telles expressions complètes à la place des expressions générales paraît très coûteuse. Pour limiter ce problème on peut utiliser les contextes définis sur les conditions pour "contrôler" les calculs. Si l'utilisateur connaît des informations sur les paramètres, il les déclare et coupe certaines branches de calcul pour ne conserver que les valeurs qu'il désire. Ce même contexte peut aussi être utilisé par le système lui-même à la manière du `lastproviso` de `Maple`, c'est à dire en gardant la trace des contraintes qu'il a choisi. La notion de complétude est donc étendue à la complétude modulo un contexte, ce qui correspond pour la disjonction des conditions à être toujours vrais modulo le contexte. Une autre stratégie consisterait à retarder les calculs en calculant les résultats généraux et en retardant les calculs à faire aux points critiques. Ceci impliquerait de définir ce qui signifie le calcul conditionnel paresseux. De cette manière, on pourrait recouvrir toutes les stratégies mises en oeuvre dans les systèmes existants en paramétrant le calcul multi-valué.

Dans un premier temps je me suis attachée à l'étude des expressions multi-valuées dont la forme dépend des paramètres d'entrées, ou plutôt des valeurs possibles de ces paramètres.
La définition des objets conditionnels que j'ai utilisée jusqu'à présent ne tient donc pas compte de la notion de formule erronée (par exemple $1/x$ pour $x = 0$). La gestion de ces "cas d'erreur" (division par zéro ...) pose des problèmes différents telles que la "disparition" par simplification des expressions présentant des erreurs. Par exemple l'expression xy/x est automatiquement simplifiée en y alors que pour $x = 0$ ce n'est plus vrai.
De plus, à partir de la forme simplifiée, il n'est possible de re-générer ni les

contraintes, ni les valeurs correspondantes. Ce problème apparaît clairement pour la simplification, mais aussi pour la résolution de systèmes d'équations (`solve`) ainsi que d'autres opérations pré-définies dans les systèmes.
Pour remédier à cela, il faut redéfinir ces fonctions pour qu'elles rendent les contraintes sur les paramètres en même temps que leurs résultats. Ceci semble signifier la re-définition du "coeur" des systèmes ...

Les deux points que nous venons d'évoquer sont sous-tendus par le même problème qui est de savoir ce qu'est un paramètre par rapport à une inconnue. Intuitivement, l'expression $1/x$ peut être conservée sous la forme $1/x$ si x est une inconnue, mais doit être transformée en *si* $x = 0$ *then error else* $1/x$ si on considère x comme un paramètre. Ce problème me semble être le point crucial à étudier pour mieux comprendre les spécificités du Calcul Formel et donner une structure au calcul multi-valué.