Un compilateur CAML → SYNDEX pour les applications de traitement de signal temps-réel distribuées

J. Sérot[1]

1: LASMEA - UMR 6602 CNRS
Campus des Cézeaux, 63177 Aubière cedex
Jocelyn.Serot@lasmea.univ-bpclermont.fr

Résumé

Ce papier décrit DROM, un sous-ensemble de CAML dédié à la spécification de graphes flot de données conditionnés. DROM a été développé initialement dans le but de fournir un frontal à l'outil SYNDEX, un logiciel d'aide à l'implantation d'applications de traitement de signal sur architectures parallèles. La principale originalite de DROM par rapport à d'autres systèmes de notation de graphe est son caractère "pleinement fonctionnel" et polymorphe, et notamment la possibilité d'y définir la notion de "schéma de graphe" sous la forme de fonctions d'ordre supérieur.

1. Introduction

Le logiciel SYNDEX [12], développé à l'INRIA dans le cadre du projet SOSSO, est un outil d'aide à l'implantation d'applications de traitement du signal et de l'image (TSI) temps-réel parallèles[1]. Il propose une méthodologie et un environnement de programmation permettant de faire du *prototypage rapide* de telles applications sur des architecture multi-processeurs dédiées (à base de DSP ou de processeurs généraux) ou banalisées (réseaux de stations ou PC sous Unix/Linux).

SYNDEX s'appuie sur un modèle unifié de graphe à la fois pour décrire l'architecture cible et spécifier l'algorithme à implanter. La figure 1 par exemple illustre une session SYNDEX dans le cas d'une application typique de traitement d'image. L'algorithme et l'architecture cible figurent dans la fenêtre de gauche. L'architecture cible y est modélisée sous la forme d'un graphe formé de quatre processeurs (DSP TMS320C40 ici) reliés par des liens de communication série selon une topologie en anneau. L'algorithme est exprimé sous la forme d'un *graphe flot de données conditionné* (GFDC), au sein duquel les sommets correspondent aux opérations élémentaires et les arcs aux dépendances de donnée entre ces opérations. Les sommets sans prédécesseur (resp. sans successeur) correspondent à l'interface avec l'environnement (réception des stimuli et production des réponses)[2].

Le GFDC n'implique en général qu'un ordre *partiel* sur l'exécution des opérations, d'où l'existence d'un parallélisme potentiel. La principale tâche de SYNDEX consiste à effectuer une réduction de ce parallélisme potentiel sur le parallélisme effectivement offert par l'architecture. Cette opération, dite d'*adéquation* revient à trouver une *distribution* des opérations sur les processeurs et un *ordonnancement* des opérations et des communications résultantes sur chaque processeur compatibles avec l'ordre

1. Par *temps-réel*, on entend ici la nécessité pour ces applications de réagir de manière *réactive* avec leur environnement, en produisant à chaque nouveau stimulus des réponses en un temps borné. Ces contraintes temporelles expliquent le recours au parallélisme.

2. L'application étant réactive, ce graphe correspond en fait à une factorisation finie (repliement) d'un motif infiniment répété (activé) [13]

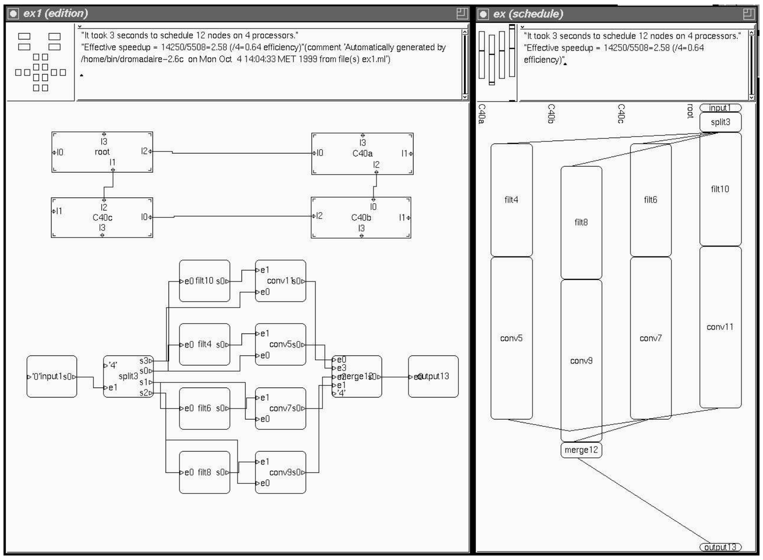


FIG. 1 – *Une session* SYNDEX

partiel et satisfaisant les contraintes temporelles de l'application. Pour cela — et à cause du caractère NP-complet de ce problème — SYNDEX s'appuie sur une heuristique de minimisation de la latence du graphe résultat (durée totale d'une activation, calculée à partir de la durée de chaque opération). La figure 1 (fenêtre de droite) montre le résultat d'une adéquation effectuée sur les graphes décrits à gauche. Il y a une colonne par processeur et dans chaque colonne, les opérations sont représentées par des boites de hauteur proportionnelle à leur durée. Les communications inter-processeurs apparaissent comme des lignes joignant une boite source à une (ou plusieurs) boite(s) destination. La hauteur totale du diagramme donne donc la latence globale de l'application. Sur la base de ce diagramme, l'utilisateur peut juger de l'efficacité de l'adéquation [3] et le cas échéant, tenter de l'améliorer en modifiant l'architecture, l'algorithme ou les deux avant de relancer l'heuristique d'adéquation.

Dès qu'une adéquation satisfaisante est trouvée, SYNDEX est en mesure de générer un *exécutif distribué optimisé* pour chaque processeur de l'architecture. Cet exécutif est généré sous la forme d'un macro-code (`m4`) faisant appel à un jeu de primitives couvrant toutes les fonctionnalités requises par le modèle d'exécution sous-jacent (décrit dans [11]): opérations arithmétiques et logiques, appel de fonctions externes, allocation et initialisation de mémoire, exécution en pseudo-parallélisme de séquences de calcul et de communication, synchronisations entre ces séquences, communication inter-processeur par bus ou lien, *etc.* La définition de ce jeu de primitives (le *noyau* d'exécutif) pour chaque type de processeur cible suffit alors pour produire alors le code compilable (puis téléchargeable) pour l'architecture cible, ce qui favorise la portabilité des solutions (seul ce noyau d'exécutif est à réécrire dans la cas d'un changement de processeur cible).

Signalons enfin, que l'usage d'un modèle d'exécution synchrone (analogue à celui du langage SIGNAL [1]) pour la spécification des GFDC permet à SYNDEX de garantir la *sûreté* des exécutifs générés (absence d'inter-blocage) et donc d'éliminer de fait la délicate phase de mise au point des applications (qui dans le cas d'architecture multi-DSP peut vite tourner au cauchemar, compte-tenu du caractère très bas-niveau des ressources manipulées et de la relative faiblesse des outils logiciels de mise au point).

En résumé, les facilités offertes par SYNDEX — exploration "interactive" de solutions, génération automatique d'exécutifs en C, suppression de la phase de "debug" — en font un outil extrêmement

3. En comparant par exemple l'accélération effective avec l'accélération maximale théorique, calculée par SYNDEX à partir du chemin critique du graphe logiciel

efficace lors de la phase d'*implantation*. Un usage régulier nous a toutefois conduit à souhaiter une amélioration de ses capacités au niveau de la *spécification* des applications, en particulier lors de la saisie du graphe de l'algorithme. Cette saisie peut se faire sous trois formes: graphiquement, en placant et connectant "à la souris" les opérations (c'est la solution la plus courante), textuellement, en entrant le code SYNDEX [4], "à la main" directement dans l'interpréteur figurant en haut à droite de la figure 2 (solution rarement utilisée) ou à partir d'une spécification en SIGNAL (une option du compilateur SIGNAL permet de générer du code SYNDEX). Les deux premières solutions ne sont pas pratiquement satisfaisantes dès la complexité du graphe dépasse quelques dizaines de nœuds, situation qui se produit fréquemment lorsque le *grain* de la description algorithmique diminue [5], comme illustré sur la figure 2.

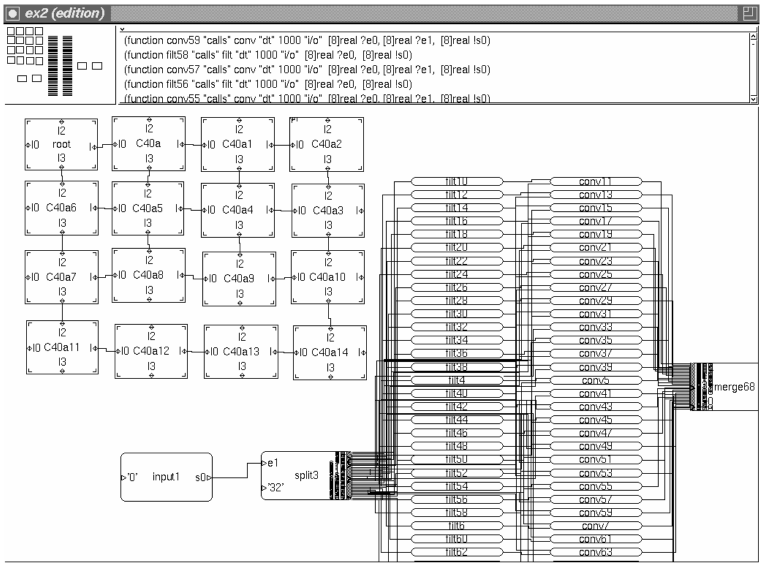


FIG. 2 – *Un graphe difficile à décrire et à manipuler sous* SYNDEX

La notion de *graphe factorisé*, introduite dans [13] — pendant de celle d'`array` (tableaux de processus) en SIGNAL — permet, en théorie, de s'affranchir des problèmes résultant d'une granularité trop fine. Malheureusement, les instructions correspondantes ne sont actuellement supportées ni par l'interface SIGNAL → SYNDEX (la traduction des `array` en graphes factorisés n'étant pas formalisée dans le cas général), ni par le générateur de code de SYNDEX (l'implantation efficace des graphes factorisés posant des problèmes particulièrement complexes vis à vis de l'heuristique de distribution-ordonnancement). En son absence, la spécification de "gros" graphes requiert donc une bonne dose de "copier-coller", que ce soit sous forme graphique (sous SYNDEX) ou textuelle (en SIGNAL). Par ailleurs, la notion de graphe factorisé, comme celle d'`array` en SIGNAL, reste réservée au codage de graphes présentant une "régularité" suffisante (répétition de motifs simples). Dans le cas général, la notion de *schéma de graphe* [6], *i.e* de motif "irrégulier" et polymorphique est hors de sa portée. De tels schémas apparaissent pourtant fréquemment lors de la parallélisation des algorithmes de TSI, sous la forme de "patrons" ou "squelettes" génériques [4]. Ils peuvent être vus comme l'analogue "graphique" de celle de fonction d'ordre supérieur présente dans les langages fonctionnels.

D'où l'idée d'un outil permettant de spécifier des GFDC sous forme textuelle mais dans un formalisme pleinement fonctionnel et polymorphe (par contraste, SIGNAL est d'ordre 0 et monomorphe). Ce papier traite donc du développement d'un frontal Caml à SYNDEX, *c.a.d* de la

4. Le "code" SYNDEXest une simple description textuelle, "a plat" du graphe logiciel, listant l'ensemble des sommets et des connexions entre ces sommets. Un extrait de ce code est visible sur la figure 2

5. Par exemple, lorsque l'on passe d'une description orientée "image" à une description orientée "pixel" en TI

6. La formulation est choisie ici par analogie avec celle de *schéma de type* par exemple

possibilité de décrire en CAML des graphes flot de données[7].

2. DROM$_0$: un premier compilateur CAML $\rightarrow$ GFDC

DROM$_0$ est un sous-ensemble du langage Caml Light [7] dédié à la description de graphes flot de données conditionnés. Comme CAML, DROM$_0$ est strict, polymorphique et supporte les fonctions d'ordre supérieur.

Le choix d'un langage à sémantique stricte (par opposition à ceux à sémantique "paresseuse" ou "lazy"), n'est pas fortuit: cette sémantique est celle qui s'accorde en effet le mieux avec le modèle d'exécution *data-driven* des GFDC, au sein duquel c'est la *disponibilité* des arguments d'une fonction qui déclenche son exécution[8].

Le choix de Caml Light lui-même répond à des exigences essentiellement pragmatiques: diffusion dans la communauté, existence de supports pédagogiques, accès au code source. A ces raisons s'ajoutent en fait une dernière, développée au paragraphe 5: la possibilité d'utiliser un interpréteur Caml Light "standard" pour exécuter des spécifications de GFDC.

Le langage DROM$_0$ décrit dans ce paragraphe constitue un noyau minimum qui, progressivement enrichi dans les paragraphes 3 et 4 conduira au langage DROM réellement utilisé pour les applications de TSI évoquées en introduction.

De Caml Light, DROM$_0$ ne retient que trois constructions — la liaison `let` (non récursive[9]), la définition de fonction via (une forme limitée de) filtrage et l'application de fonction — et trois types de base: les entiers (`int`), les flottants (`float`) et les booléens (`bool`). Sont omises — pour des raisons évidentes — toutes les constructions impératives (assignation, boucles `for` et `while`) et celles liées à la gestion des exceptions. Il n'y a pas non plus de déclaration de type ou de valeur (`type t` ou `val f : ..`). Par suite toutes les opérations apparaissant dans le graphe doivent correspondre à des identificateurs prédéfinis (primitives arithmétiques, ...).

La figure 3a donne un programme DROM$_0$ simple et le GFDC correspondant.

La syntaxe abstraite des programmes DROM$_0$ est très simple[10].

```
type expression =
    Variable of string
  | Fonction of motif * expression
  | Application of expression * expression
  | Let of definition * expression
  | Constante of constante

and motif =
    Motif_variable of string
  | Motif_produit of motif vect

and constante =
    Entier of int
  | Flottant of float
```

7. Insistons sur cette approche essentiellement *pragmatique* à opposer notamment à celle, plus théorique, qui viserait à donner une interprétation flot de données à un programme CAML quelconque (dans le but de proposer un compilateur CAML parallélisant par exemple)

8. Un langage à évaluation paresseuse s'accorde lui mieux d'un modèle d'exécution *demand-driven.*

9. Le cas des définitions récursives est traité au paragraphe 3.4

10. Les extraits de code donnés dans ce papier sont tirés d'une première implémentation largement inspirée du langage MINICAML décrit au chapitre 16 de [7]. On ne décrira donc ici que les structures de données et les fonctions spécifiques à notre implémentation.

```
  | Booleen of bool

and definition = {
    nom : string;
    exp : expression };;

type phrase =
    Definition of definition
  | Expression of expression;;

type programme == phrase list;;
```

2.1. Traduction

On décrit ici le principe de la traduction d'un programme DROM_0 complet en un GFDC. Cette traduction est effectuée phrase par phrase, au sein d'un environnement global composé d'un liste de liaisons `(nom,valeur)` et du graphe en cours de construction. Au début de la traduction, la liste contient les identificateurs prédéfinis (primitives) et le graphe est vide.

Ensuite, pour chaque phrase, le traducteur effectue successivement l'analyse syntaxique, la synthèse et vérification de type (l'AST est alors annoté) et l'évaluation de l'AST-typé[11]. Les deux premières phases sont classiques et ne seront pas détaillées ici. Le mécanisme d'évaluation d'une expression est décrit au paragraphe suivant. Il consiste en une mise à jour du graphe contenu dans l'environnement global. Par ailleurs, si la phrase traitée est une définition, l'environnement global est enrichi de la nouvelle liaison.

En fin de programme, une dernière phase (triviale) permet de produire une représentation concrète du graphe contenu dans l'environnement global. Dans notre cas, il s'agit de code SYNDEX, mais tout autre format est évidemment envisageable[12].

2.2. Evaluation

Le mécanisme d'évaluation d'une phrase constitue la principale originalité de DROM_0 par rapport à une implantation CAML "classique". L'idée de base est simple: seules les constantes et l'*application complète* d'une primitive correspondent à ("créent") un sommet (nœud) dans le graphe résultat. Les premières sont représentées par des sommets sans prédécesseur (étiquetés avec la valeur de la constante), les secondes par des sommets étiquetés avec le nom de la primitive et dont les arcs d'entrée représentent les arguments de la primitive. Un argument étant soit une constante, soit le résultat de l'application d'une autre primitive, tout arc connecte nécessairement la sortie d'un sommet à l'entrée d'un autre[13].

La fonction d'évaluation distingue donc trois types de valeurs, correspondant respectivement à la sortie d'un sommet, à une primitive partiellement appliquée, et aux fonctions (fermetures):

```
type valeur =
    Val_noeud of graphe__id_sommet
  | Val_primitive of primitive
  | Val_fermeture of fermeture
```

11. Pour des raisons de lisibilité, et du compte-tenu du fait que DROM se comporte sur ce point exactement comme CAML, toutes les informations de type ont été omises dans les fragments de codes reproduits ici.
12. Les graphes reproduits dans ce papier ont été obtenus à partir d'un module de sortie au format `.dot` [19] par exemple.
13. Le cas des sommets multi-sorties sera abordé au paragraphe 3.3

```
and fermeture = {
     defn : motif * expression;
     envn:  (string * valeur) list }

and primitive = { nom: string; arite: int; args: valeur list };;

type environnement = { graphe : graphe__graphe; liaisons: (string * valeur) list };;
```

A noter le codage classique des fermetures qui reprend la règle de portée statique des identificateurs. Cette règle est la seule admissible dans le cas d'une interprétation flot de données.

Le graphe lui-même est représenté classiquement sous la forme d'une liste de sommets. Chaque sommet, repéré par un identificateur unique, est étiqueté avec la nom de la primitive appliquée ou de la constante portée et contient la liste des arcs en entrée et en sortie. Les arcs pointent toujours sur un sommet (source ou destination) et sont étiquetés avec le type de la donnée portée.

```
type graphe = { mutable sommets : (id_sommet * sommet) list }
and  sommet = { s_desc: s_info; mutable entrees : arc list; mutable sorties : arc list}
and arc = { index: id_sommet; a_desc: types__type_simple}
and s_info =
    NInt of int
  | NFloat of float
  | NBool of bool
  | NPrim of string;;
val cree_graphe : unit -> graphe;;
val ajoute_sommet : graphe -> (s_info * arc list * arc list) -> id_sommet;;
val cherche_sommet : graphe -> id_sommet -> sommet;;
```

La fonction d'évaluation prend une expression, un environnement et retourne une valeur. Si cette expression contient une constante ou l'application complète d'une primitive, le sommet correspondant est créé et ajouté au graphe contenu dans l'environnement global. Le graphe est donc construit incrémentalement, "par effet de bord", lors de l'évaluation de chaque expression du programme.

```
let rec evalue_expression env expr = match expr with
    Variable id -> begin try assoc id env with Not_found -> erreur "id inconnu"  end
  | Fonction (motif,exp) -> Val_fermeture {defn=motif,exp; envn=env.liaisons}
  | Let (def,corps) -> evalue_expression (evalue_definition env def) corps
  | Constante c -> evalue_constante c
  | Application (fn,arg) ->
      let val_fn = evalue_expr env fn in
      let val_arg = evalue_expr env fn in
      evalue_application env val_fn val_arg

and evalue_definition env def = (def.nom, evalue_expr env def.exp) :: env

and evalue_constante c = match c with
    Entier n -> Val_noeud (ajoute_sommet env.graphe (NInt n, [], [])
  | Flottant n -> Val_noeud (ajoute_sommet env.graphe (Nfloat n, [], [])
  | Booleen b -> Val_noeud (ajoute_sommet env.graphe (NBool b, [], [])

and evalue_application env fn arg = match fn with
```

```
  | Val_primitive p ->
      let p' = { nom = p.nom; arite = p.arite; args = arg :: p.args } in
      if p'.arite = list_length p'.args then instancie_primitive env p'
      else Val_primitive p'
  | Val_fermeture ferm ->
      begin match ferm.defn with
           (motif,expr) ->
              let env_etendu = (env.graphe, filtrage arg motif @ ferm.envn) in
              evalue_expr env_etendu expr
          | _ -> erreur "application d'une valeur non-fonctionnelle" end

and instancie_primitive env p =
   Val_noeud (ajoute_sommet env.g (NPrim p.nom, rev p.args, []));;
```

Le langage DROM_0, malgré sa pauvreté apparente, permet déjà de décrire une large classe de graphes, notamment grâce à la pleine fonctionnalité offerte par la gestion "à la CAML" des fermetures. On peut ainsi décrire des graphes par simple succession de définition (dans le style des langages à assignation unique) (fig 3a), définir et utiliser des fonctions comme des sous-graphes (fig 3b), définir des fonctions par application partielle (fig 3c) et, *last but not least*, accéder à la notion de schéma de graphe via les fonctions d'ordre supérieur (fig 3d)

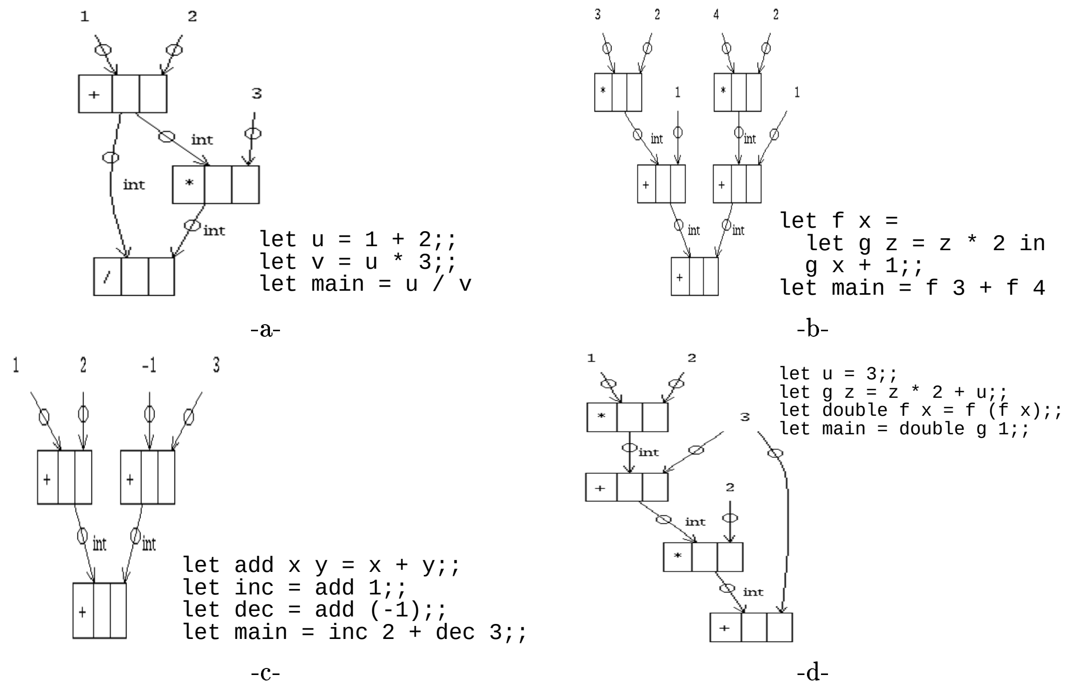


FIG. 3 – *Quelques programmes* DROM_0 *et les graphes correspondants*

On remarquera que la référence à un identificateur admet deux interprétations différentes selon que cet identificateur désigne une valeur fonctionnelle ou non. Dans le premier cas il y a *partage* de la valeur correspondante: sur la figure 3a par exemple, les deux arcs issus du sommet + désignent la *même* valeur. Dans le second cas, le sous-graphe correspondant est *répliqué* (instancié) à chaque référence (cas des fonctions `f` et `g` des figures 3b et 3d)[14].

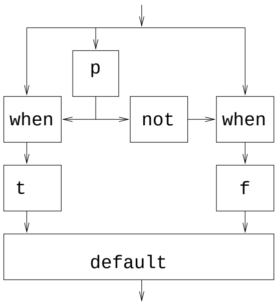
14. Un programme DROM peut donc – en théorie – générer un graphe exponentiellement plus grand que son texte

3. Le langage DROM$_1$

Si le langage DROM$_0$ permet de valider les bases du schéma de traduction CAML $\rightarrow$ GFDC, il lui manque un certain nombre de caractéristiques pour pouvoir être utilisable dans le cadre d'applications réalistes. Ces caractéristiques concernent l'expression de la conditionnelle, la déclaration de primitives externes, les opérations multi-sorties et le statut des définitions récursives.

3.1. Traduction de la conditionnelle

Dans le modèle flot de données synchrone, il n'y a pas de construction spécifique pour la conditionnelle: celle-ci s'exprime explicitement à l'aide des primitives `when` et `default`.




```
let cond p t f x =
  let px = p x in
  let tx = t (when px x) in
  let fx = f (when (not px) x) in
  default tx fx
```

FIG. 4 – *La conditionnelle dans le modèle flot de données synchrone*

Son expression en DROM ne requiert donc *a priori* que deux primitives `when: bool ->'a ->'a` et `default: 'a ->'a ->'a` et une fonctionnelle `cond`, comme illustré sur la figure 4. Cette approche purement "syntaxique" s'est révélée suffisante pour nos applications cibles, qui ne contiennent en général que peu de conditionnement. Signalons toutefois que l'introduction de fonctionnalités d'ordre supérieur au sein d'un formalisme synchrone pose dans le cas général des problèmes théoriques significatifs [15], dont l'exposé dépasse largement le cadre de ce papier. Le lecteur intéressé est renvoyé sur ce point aux travaux de Caspi et Pouzet sur la programmation synchrone en CAML[17].

3.2. Primitives et types externes

En pratique, les sommets des GFDC manipulés par SYNDEX ne sont pas de simples opérateurs, mais des fonctions séquentielles définies séparément (en C). Les identificateurs de ces fonctions doivent donc être reconnus comme des primitives [16] par DROM. Par ailleurs, ces fonctions manipulent parfois des types de donnée non-atomiques, *i.e* différents des simples `int`, `float` et `double` du C qui admettent un équivalent direct en SYNDEX. En l'absence d'un schéma de traduction systématique de types de C vers SYNDEX et de CAML vers SYNDEX, l'approche retenue ici consiste à encapsuler les types non-atomiques avec un `typedef` C et à les faire voir comme des types atomiques par DROM. En effet, seuls l'heuristique et le générateur de code SYNDEX requièrent la description de l'implémentation d'un type (pour l'estimation des volumes et donc des temps de transmission et pour la génération des fonctions

(un peu comme l'algorithme de synthèse de type de CAML peut produire des types exponentiellement complexes à partir de phrases polymorphiques simples (cf [7],17.9). Il s'agit là bien sur d'une "limitation" toute théorique puisque, comme souligné à la fin de la section 1, notre objectif n'est pas de traduire sous forme de graphe des programmes CAML *quelconques* mais de donner une *description* concise (par factorisation fonctionnelle justement) de "gros" graphes.

15. Il devient très facile notamment de construire des graphes ne respectant pas la propriété de synchronisme

16. Que l'on peut qualifier d' "externes", par opposition aux primitives internes prédéfinies.

de communication[17]).

On rajoute donc deux types de *déclaration* à la syntaxe DROM:

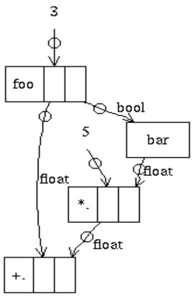
- `type t` déclare un type "opaque", vu comme un type atomique par DROM
- `value f : t1 ->t2 ->... ->tn : k "f_name"` déclare une primitive externe prenant $n-1$ arguments (de type `t1`,..,`tn-1`) et renvoyant un résultat de type `tn`. Au niveau de DROM, cette primitive possède le même statut qu'une primitive prédéfinie: appliquée à $n-1$ arguments, elle créera un sommet — de nom `f_name` et possédant $n-1$ arcs d'entrée dans le graphe résultat. Le format de la déclaration — et notamment le paramètre `k` (arité) — ne se justifie que par compatibilité avec la syntaxe Caml Light et avec les possibilités d'émulation séquentielle décrites au paragraphe 5.

En termes d'implantation, ces deux fonctionnalités ne posent pas de difficulté. Les types opaques sont représentés comme tous les types simples de Caml Light, par un `terme` (cf [7], 17.3) et les primitives externes exactement comme les primitives internes. La compilation d'une déclaration de type (resp. de primitive) conduit à mettre à jour les environnement de typage (resp. d'évaluation) pour la compilation des phrases suivantes. En pratique — et pour assurer la compatibilité avec un interpréteur CAML séquentiel —ces phrases sont données dans un (ou plusieurs) fichier(s) séparés (.mli).

3.3. Les opérateurs multi-sorties et les types produits

La nécessité d'un type produit (n-uplet) apparaît dès que le graphe comporte des opérations à plusieurs sorties[18]. La déclaration de la primitive correspondante est alors nécessairement de la forme `value f : t1 ->... ->tn * tn+1 * ... * tn+m`, où `m` est le nombre de sorties.

La présence de valeurs de tel type en sortie d'un sommet requiert alors l'usage de primitives de sélection pour accéder à une de leur composante. Ces primitives sont par exemple `val fst : 'a * 'b ->'a` et `val snd : 'a * 'b ->'b` dans le cas d'un opération à deux sorties. Pour DROM, de telles primitive de sélection sont nécessairement prédéfinies et interprétées de manière *ad-hoc* lors de l'évaluation car contrairement à toutes les autres elles ne créent pas de sommet dans le graphe résultat (cf fig 5).




```
(* Fichier appli.mli *)

value foo : int -> float * bool = 1 "foo";;
value bar : bool -> float = 1 "bar";;

(* Fichier appli.ml *)

let u = foo 3;;
let main = fst u +. 5.0 *. bar (snd u);;
```

FIG. 5 – *Primitives de sélection*

En termes d'implantation, la prise en compte des primitives multi-sorties et des types produit

17. Il y a en fait un cas où un type C non-atomique admet une représentation "transparente" sous DROM: c'est celui des tableaux, qui sont représentés par des `vect` en Caml Light et gérés explicitement par SYNDEX. Cet aspect ne sera toutefois pas détaillé ici. Signalons par ailleurs que La possibilité de déclarer des types externes n'étant apparue que dans la version 5 de SYNDEX, les implantations de DROM ciblant les versions antérieures utilisent un mécanisme *ad-hoc* d'encodage de ces types dans le nom des arcs.

18. Cas à distinguer bien sur de celui ou la *même* sortie d'un sommet d'opération est partagée, *i.e* diffusée vers plusieurs sommets successeurs.

amène essentiellement 1) ajouter un cas `| Val_produit of valeur vect` au type `valeur` manipulé par l'évaluateur 2) à ajouter un champ `num_de_sortie` aux valeurs de type `Val_noeud` 3) à intercepter l'application des primitives `fst, ...` dans la fonction `evalue_application` et enfin à prévoir une fonction de "mise à plat" des arguments dans la fonction `instancie_primitive` lorsque qu'un argument de type produit est passé "tel quel" à cette primitive.

3.4. Valeurs récursives

Le langage DROM$_0$ n'autorise pas la définition de valeur récursives (par la construction `let rec`). On peut se poser la question de l'interprétation d'une telle construction dans le modèle flot de données synchrone.

En CAML, et sauf cas particulier, seule la définition de *fonction* récursive à un sens (par exemple `let rec fact x = if x >0 then x * fact (x-1) else 1`. En DROM, cette définition pourrait être vue comme une définition récursive de *graphe* (figure 6). Le graphe correspondant à l'application de `fact` à un argument `n` peut alors obtenu en "dépliant" `n` fois ce graphe récursif. Ce dépliage n'est évidemment possible que si `n` désigne une constante, connue à compilation du graphe. Cette propriété étant impossible à assurer syntaxiquement, on doit renoncer aux fonctions récursives dans le cas général[19].

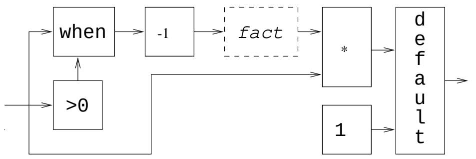


FIG. 6 – *Une graphe "récursif" pour la fonction* `fact`

La définition de valeur — au sens de "valeur de sortie d'une primitive du graphe" — récursive correspond elle à la création d'un cycle dans le graphe (fig 7a). Tel quel, ce cycle n'a pas de sens dans une interprétation flot de données synchrone: pour s'exécuter, `f` a besoin de `y`, valeur elle même produite par l'exécution de `f`, *etc.* En revanche, si l'on prend en compte l'aspect *réactif* des graphes décrits, *i.e* le fait qu'il ne représentent en fait qu'*une* activation parmi une infinité, la présence de cycle traduit la notion de *dépendance inter-activation*, *i.e* le fait que l'exécution de la fonction `f` à la k^{ieme} activation utilise le résultat produit par cette fonction à la précédente activation. Pour représenter cette dépendance inter-activation, on utilise un opérateur spécial (noté `mem` en SYNDEX, `$` en SIGNAL, `pre` en Lucid). Le graphe correspondant apparaît sur la figure 7b.

L'opérateur `mem` sera vu comme une primitive par DROM, de signature `val mem: 'a ->'a ->'a` et décrit par les relations de la figure 7c, où x_k désigne l'argument fourni à cet opérateur à l'itération k.

La primitive `mem` étant vue comme une primitive "standard" par DROM, elle ne pose aucun problème particulier en termes d'implantation (la notion de retard, et la mémoire interne que cela suppose seront gérés par SYNDEX).

19. En fait il est possible – et cette propriété sera exploitée au paragraphe 4 – de vérifier à la compilation si l'argument d'une fonction est une constante. On pourrait donc envisager de définir (au niveau de l'évaluateur) une fonctionnelle `recn` assurant une forme de dépliage récursif borné. De signature `val : recn : int ->('a ->'b) ->('a ->'b) ->('a ->'b)`, et appliquée de la manière suivante `let g = recn n f g` (où `n` est une constante entière, `f` une fonction récursive et `g` une fonction non-récursive) cette fonctionnelle instancierait récursivement `n` fois le graphe de `f` et remplacerait sa *dernière* occurrence par celui de `g`. N'ayant pas trouvé d'exemple d'utilisation probant à ce schéma, nous ne l'avons pas implanté toutefois.


```
let rec y = f x y        let rec y = f (mem y) x        mem z x0 = z
                                                        mem z xk = xk-1
```

(a) (b) (c)

FIG. 7 – *Définition récursive de valeur*

4. DROM$_2$: une extension de DROM pour le parallélisme

En TSI on est souvent amené à exprimer le parallélisme potentiel d'une application sous la forme d'un parallélisme de *donnée*, pour lequel une même fonction est appliquée indépendamment sur un ensemble de "blocs", résultant par exemple d'un découpage (partition) des données d'entrée. L'exemple de la figure 1 est typique de cette situation. On a vu à la fin du paragraphe 1 les problèmes de spécification que soulèvent cette approche quand la finesse du découpage (et donc la taille du graphe associé) augmente. Même en supposant appliquables et disponible les facilités de *factorisation régulière* évoquées alors, la mise en œuvre de tels schémas de parallélisation pose un problème pratique de codage au programmeur lorsque les fonctions de découpage et de fusion de données (`split` et `merge` dans l'exemple) sont fournies par celui ci. Ce dernier doit en effet alors déclarer autant de primitives qu'il y a de découpages envisageables. Pour un découpage des données en 2, 4, *etc* blocs, il faudrait par exemple déclarer à DROM l'ensemble des primitives suivantes:

```
type img;;
value split2 : img -> img * img = 1 "split2";;
value split4 : img -> img * img * img * img  = 1 "split4";;
value merge2 : img * img -> img  = 2 "split2";;
value merge4 : img * img * img * img -> img  = 4 "split4";;
...
```

et bien sur écrire en C toutes les fonctions correspondantes[20]:

```
typedef struct { ... } img;
void split2(/*in*/ img*, /*out*/ img*, /*out*/ img*);
void split4(/*in*/ img*, /*out*/ img*, /*out*/ img*, /*out*/ img*, /*out*/ img*);
void merge2(/*in*/ img*, /*in*/ img*, /*out*/ img*);
void merge4(/*in*/ img*, /*in*/ img*, /*in*/ img*, /*in*/ img*, /*out*/ img*);
...
```

Dans une optique de prototypage rapide — où on est amené à évaluer de nombreux schémas de découpage — cette approche devient rapidement lourde. On est alors tenté de définir deux fonctions de découpage/fusion génériques, dont les prototypes seraient par exemple

```
void split(/*in*/ int n, /*in*/ img *im, /*out*/ vararg(img*));
void merge(/*in*/ int n, /*in*/ vararg(img*), /*out*/ img *im);
```

où `n` est une constante spécifiée *à la compilation* et `vararg` la macro assurant le passage d'un nombre variable d'arguments[21]. Le problème est alors de déclarer la primitive correspondante en DROM. Une solution "naive" consisterait à traduire les `varargs` du C en listes CAML. Une primitive de signature `value split : int ->img ->img list = 2 "split"` donnerait alors un sommet à deux entrées, dont la première serait obligatoirement une constante entière `n` (règle R1) et à `n` sorties. Ce choix

20. On notera au passage l'interface requise par SYNDEX pour les primitives externes: type `void`, tous les arguments non-atomiques passés par adresse, tous les résultats retournés par adresse

21. La plupart des compilateurs C, et notamment tous ceux que nous avons utilisés en aval de SYNDEX, supportent cette construction, au prix de quelques appels de macros dans le code de la fonctions pour accéder individuellement aux arguments

risque toutefois de créer une confusion entre les “vraies” listes CAML (qui ne dont pas présentes dans DROM mais qui, pour le programmeur habitué à CAML ont une “saveur” plutôt séquentielle) et le type de sortie de fonctions comme `split`, qui correspond plus à un *produit* de taille variable. Nous avons donc préféré introduire un type spécifique, le *t-uplet* (`'a tuple`) pour qualifier les sorties des primitives renvoyant un nombre variable de résultats ou, symétriquement, prenant un nombre variable d'arguments[22]. Les primitives `split` et `merge` de l'exemple auront donc pour signature:

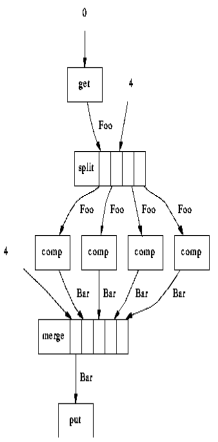
```
value split : int -> img  -> img tuple = 2 "split";;
value merge : int -> img tuple -> img = 2  "merge";;
```

Un `'a tuple` est donc un produit homogène (`t*...*t`) dont la taille — contrairement aux produits CAML — n'est pas fixée syntaxiquement, mais à la compilation (la règle R1, par exemple, est vérifiée par la fonction `instancie_primitive` lors de la traduction).

Outre l'insertion des primitives à nombre variable d'entrée-sorties, la notion de *t-uplet* permet une expression élégante du parallélisme de donnée, à condition de disposer d'une fonctionnelle `pmap`, permettant d'appliquer une même fonction à toutes les composantes d'un *t-uplet*

```
val pmap : ('a -> 'b) -> 'a tuple -> 'b tuple
```

La figure 8 illustre cette possibilité. On notera au passage la définition et l'usage de la fonctionnelle `scm` pour encapsuler le *schéma de graphe* sous-jacent, dit en *split-compute-merge*.




```
(* Fichier appli.mli *)

type foo;;
type bar;;
value get : int -> foo = 1 "get";;
value split : int -> foo -> foo tuple = 2 "split";;
value comp : foo -> bar = 1 "comp";;
value merge : int -> bar tuple -> bar = 2 "bar;;
value put : bar -> unit = "put";;

(* Fichier appli.ml *)

let scm n s c m x = m (n pmap c (s n x));;
let x = get 0;;
let y = scm 4 split comp merge x;;
let main = put y;;
```

FIG. 8 – *Les* t-uplets *et le parallélisme*

En termes d'implantation, on notera que contrairement au couple (`'a list`, `map`) en CAML, ni le type `'a tuple`, ni la fonctionnelle `pmap` ne sont exprimables dans le langage en DROM. Ils sont donc gérés de maniere *ad-hoc* par le synthétiseur de type et l'évaluateur. La fonction `instancie_primitive` par exemple, est modifiée de la manière suivante:

```
let instancie_primitive env p ty_res = (* ty_res est le type resultat de la primitive,
  match p.nom with                        infere par le synthetiseur de type *)
    "pmap" ->
       let f, xs = begin match p.args with f :: Val_tuple xs :: [] -> f, xs in
       let ys = map_vect (evalue_application env f) xs in
       Val_tuple ys
  | _ ->
     let n = ajoute_sommet env.g (NPrim p.nom, rev p.args, []) in
```

22. On verra au paragraphe suivant que dans une interprétation *séquentielle* des programmes DROM, les types `'a tuple` et `'a list` se confondent

```
match ty_res with
    types__terme("tuple", t)  ->
      let taille = (match hd (rev p.args) with
          Val_noeud (id,_) ->
            (match (graphe__cherche_sommet env.graphe id).s_info with
                NInt n -> n
              | _ -> erreur "le 1er argument de pmap doit etre une cte entiere")
         | _ -> erreur "le 1er argument de pmap doit etre une cte entiere") in
      Val_tuple (init_vect taille  (function i -> Val_noeud(n, i))
  | ...
```

5. Emulation séquentielle

Par émulation séquentielle, on entend la possibilité de donner une interprétation *séquentielle* — via un interpréteur Caml Light "standard" par exemple — à des GFDC décrits en DROM. Cette possibilité, qui se justifie théoriquement par le fait qu'une exécution séquentielle n'est qu'un cas particulier de l'ordre partiel imposé par le GFDC[23], est d'un grand intérêt pratique puisqu'elle permet de faire du prototypage de programme parallèles *sans disposer* de l'architecture cible, sur simple station de travail par exemple. Il est alors possible — comme souligné dans [3] — de vérifier la *logique* du programme et le comportement des fonctions séquentielles en dehors de toute exécution parallèle et de réserver les outils d'implantation parallèle à la mise au point des aspects *temporels*.

En pratique, les points à régler pour assurer la compatibilité entre le traducteur DROM et un interpréteur (ou un compilateur) Caml Light concernent les types et fonctionnelles "built-in" de DROM, l'interprétation des définitions récursives de valeur et — d'un point de vue plus technique — l'interface aux primitives externes écrites en C.

5.1. Les types et valeurs prédefinies DROM

Les primitives arithmétiques et logiques prédéfinies en DROM le sont aussi pour CAML et donc ne nécessitent aucune redéfinition. Pour tous les autres types et valeurs, on fournit un module `emul.ml` contenant un "équivalent séquentiel" en Caml Light. Ce module contient par exemple une (ré)interprétation des *t-uplets* introduits au paragraphe 3.3 sous la forme de listes et une réécriture de la conditionnelle définie au paragraphe 3.1:

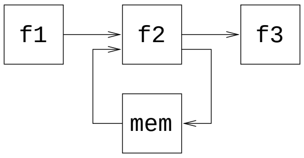
```
(* Fichier emul.ml *)

type 'a tuple = 'a list;;
let pmap = map;;
let cond p t f x = if p x then t x else f x;;
...
```

5.2. Les définitions récursives de valeur et la fonctionnelle `itermem`

On a vu au paragraphe 3.4 que les définitions de valeurs récursives se justifiaient dans un contexte d'exécution *itérative* du GFDC. Jusque là, cette hypothèse était restée complètement implicite dans les programmes DROM, et ce parce que le graphe final est toujours "implicitement itéré" pour SYNDEX. L'insertion de la primitive `mem` oblige au contraire à expliciter la notion d'itération temporelle au niveau du code DROM. Pour cela, on encapsule l'opérateur `mem` dans le schéma de graphe `itermem` (fig. 9).

23. A condition bien sur que ces primitives soient de fonctions *pures*, *c.a.d* sans effet de bord




```
let itermem f1 f2 f3 z x =
  let rec y = f2 (mem z (fst y), f1 x) in
  f3 (snd y);;
```

FIG. 9 – *La fonctionnelle* `itermem`

L'expression `itermem get loop put z0 x` décrit alors un graphe pour lequel, à chaque itération (activation), la donnée produite par la fonction `get` est passée à la fonction `loop` avec le résultat de l'itération précédente. Cette fonction produit d'une part le résultat (mémorisé) pour la prochaine itération et d'autre part une sortie pour la fonction `put`.

La définition *séquentielle* de cette fonctionnelle `itermem` (présente dans le fichier `emul.ml`) sera, elle, la suivante:

```
let itermem f1 f2 f3 z x =
  let rec h z =
    let z', y = f2 (z, f1 x) in f3 y; h z'
  in h z;;
```

Par souci d'homogénéité, on est amené à expliciter l'itération même dans les graphes sans opérateurs `mem`, en introduisant la fonctionnelle suivante `val iterate : ( 'a ->'b) ->'a ->'b` qui se définit respectivement comme `let iterate f x = f x` pour DROM et comme `let iterate f x = f x ; iterate f x` en CAML. De telle sorte que le programme de la figure 3a s'écrira en fait:

```
let main () =
  let u = 1 + 2 in
  let v = u * 3 in
  u / v;;
iterate main ();;
```

5.3. L'interface CAML / C / SYNDEX

L'appel à des primitives externes écrites en C depuis un interpréteur Caml Light suppose la génération d'un code d'interface (*stub-code*) adéquat afin de convertir les structures de données entre C et CAML. Les conventions imposées par SYNDEX sur l'interface de ces fonctions externes C contraignent de manière assez forte la structure de ce *stub-code*. Exception faite des valeurs atomiques (`int`,`float`,...) arguments et résultats sont en effet toujours des *pointeurs* sur des tampons alloués statiquement, par le générateur de code SYNDEX. Dans le cas d'une émulation séquentielle, ces tampons devront au contraire être alloués *dynamiquement* par le *stub-code* et, bien sur récupérés par le GC. S'agissant de types opaques, l'allocation ne peut se faire que dans le tas C. Le tas CAML ne contient alors que des pointeurs finalisés vers le tas C. Il faut alors fournir pour chaque type opaque une fonction d'allocation et de libération au *stub-code*. Afin d'automatiser la mise en œuvre — assez lourde dans le cas général — de ce mécanisme, nous avons donc développé une extension à l'outil `camlidl` de X. Leroy [15]. Cette extension propose des attributs `idl` permettant d'interfacer des fonctions opérant sur des tampons gérés de manière transparente par le stub-code lui-même. Elle est décrite dans [16].

6. Travaux en rapports

L'évidence d'une relation entre les formalismes de programmation de type flot de données et les langages fonctionnels remonte aux travaux de Dennis sur VAL [9]. Le langage VAL, et la plupart des projets qui s'en sont inspirés (notamment le dialecte de FP introduit par l'auteur dans [14]) sont toutefois d'ordre 0 et ne supportent donc pas la notion de schéma de graphe. La visualisation de programmes pleinement fonctionnels a fait elle l'objet d'investigations séparées. Citons par exemple les travaux de Reekie sur VISUAL HASKELL [10] ou de Kelso [6]. L'esprit de ces travaux est cependant assez éloigné du notre puisque qu'il s'agit de fournir une *représentation graphique* de programmes fonctionnels *quelconques* (dans le but de proposer des environnements de programmation purement graphiques par exemple). Les travaux de Launchburry sur HAWK [18] et Sheeram sur LAVA [2] sont, sous cet éclairage, plus proche des nôtres dans la mesure où ils utilisent les facilités d'abstraction d'un langage fonctionnel (Haskell) pour *décrire* concisément des graphes de systèmes matériels complexes (circuits intégrés). Ces outils sont avant tout dédiés à la spécification et à la vérification formelle de circuits (V)LSI (même si un des objectifs de LAVA et la production automatique de code VHDL synthétisable). Enfin, nos travaux présentent de nombreuses similarités avec le projet Lucid Synchrone [17], ne serait-ce que par une définition commune de la classe des systèmes cibles (réactifs, synchrones) et par le langage source utilisé (CAML). La principale caractéristique de Lucid Synchrone est de reposer sur une formalisation rigoureuse de la pleine fonctionnalité au sein du modèle synchrone. Notre approche est beaucoup plus pragmatique, donc plus difficile à justifier d'un point de vue théorique. Nous la justifions par le fait que, comme évoqué au paragraphe 3.1, les applications envisagées font un usage très restreint de l'hypothèse synchrone. Enfin, signalons que, de par son adossement à SYNDEX, DROM cible spécifiquement des architectures distribuées, alors que Lucid Synchrone – à notre connaissance – ne produit que du code CAML séquentiel.

7. Conclusion et perspectives

Une version expérimentale de DROM, écrite en Caml Light et inspirée du MINICAML décrit dans [7] est opérationnelle et a déjà servi à spécifier et implanter des applications de traitement d'images réalistes. Dans [5] par exemple on décrit une application d'étiquetage en composantes connexes temps réel implantée sur une architecture multi-processeurs et entièrement spécifiée en DROM.

Une réécriture est en cours sur la base des sources de Caml Light 0.74. Ce repositionnement, qui consiste uniquement à effectuer la traduction en GFDC directement à partir de l'AST généré par l'analyseur de syntaxe de Caml Light 0.74, permettra d'améliorer sensiblement la convivialité de l'outil (report des erreurs notamment). Une étape suivante consisterait à partir non plus de l'AST mais du code intermédiaire (λ-code décrit dans [8]). Cette approche se justifierait notamment dans le cadre d'une extension des types DROM aux types sommes de CAML, extension qui requiert un vrai mécanisme de compilation du filtrage afin de limiter la quantité de conditionnement dans le graphe final.

Enfin, l'interface à la version actuelle de DROM est assez rudimentaire: il s'agit d'un simple compilateur "batch", traitant une par une des phrases issues de un ou plusieurs fichiers (.ml ou .mli) passés sur la ligne de commande et produisant du code SYNDEX qui doit être relu depuis une session SYNDEX. Deux variantes sont donc à l'étude: d'une part, une version "interactive", intégrée à l'IHM de SYNDEX v5.0 et avec laquelle il serait possible d'entrer et de modifier des graphes d'algorithmes directement depuis SYNDEX (en redirigeant les entrée-sorties de l' interpréteur de commande SYNDEX vers un processus DROM; d'autre part, une version supportant la compilation séparée. Ce dernier point est assez délicat car il suppose une définition claire de la notion d' "édition de lien" dans le cas d' "objets" de type graphe.

Références

[1] A. Benveniste, P. Le Guernic, Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, Sep. 1991.

[2] P. Bjesse, K. Claessen, M. Sheeran, S. Singh. Lava: hardware design in Haskell. In *ACM Conf on Functional Programming*, Baltimore, USA, 1998.

[3] M. Danelutto, R. DiCosmo, X. Leroy, Susanna Pelagatti. Parallel functional programming with skeletons: the OCamlP3L experiment. In *Proceedings ACM workshop on ML and its applications.* Cornell Univ., 1998.

[4] D. Ginhac. *Prototypage rapide d'applications de vision artificielle par squelettes fonctionnels.* PhD thesis, Université B. Pascal, 1999.

[5] D. Ginhac, J. Sérot, J.P. Dérutin. Fast prototyping of image processing applications using functional skeletons on a MIMD-DM architecture. In *IAPR Workshop on Machine Vision and Applications*, Chiba, Japan, Nov 1998.

[6] J. Kelso Visual Representation for Functional Programs. Technical report CS-95-01, Dept of Computer Science, Murdoch University, 1995

[7] X. Leroy, P. Weis. *Le langage Caml.* Interéditions, 1993.

[8] X. Leroy. The ZINC experiment: an economical implementation of the ML language. RT inria no 117, Février 1990.

[9] J.R. Mc Graw. The VAL language: description and analysis *ACM Trans PLS*, 4:44–82, 1982.

[10] H.J. Reekie Visual Haskell: A first attempt. Technical report 94.5, University of Technology Sydney,

[11] Yves Sorel Thierry Grandpierre, Christophe Lavarenne. Modèle d'exécutif distribué temps réel pour SynDEx. RR inria no3476, Aout 1998.

[12] Y. Sorel. Massively parallel systems with real time constraints. The "Algorithm Architecture Adequation" Methodology. In *Proc. Massively Parallel Computing Systems*, Ischia Italy, 1994.

[13] Y. Sorel, C. Lavarenne. Modèle unifié pour la conception conjointe logiciel-matériel. *Traitement du Signal*, 14(6), 1997.

[14] J. Sérot, G. Quénot, B. Zavidovique. A Visual Dataflow Programming Environment for a Real-Time Parallel Vision Machine. *Journal of Visual Languages and Computing*, 6:327–347, 1995.

[15] `http://pauillac.inria.fr/caml/camlidl`.

[16] `http://wwwlasmea.univ-bpclermont.fr/Personnel/Jocelyn.Serot/soft/patch-camlidl`.

[17] `http://www-spi.lip6.fr/softs/lucid-synchrone.html`

[18] `http://www.cse.ogi.edu/PacSoft/projects/Hawk`

[19] `http://www.research.att.com/sw/tools/graphviz`.