# Compilation de `pH` pour une machine parallèle *multi-contextes*

Guy Tremblay

Juin/juillet 1997

## 1 Introduction: Les grandes lignes du processus de compilation

Ce document décrit le processus de compilation de programmes `pH` pour des machines parallèles multi-contextes (*multi-threaded parallel computer*). Les grandes lignes de ce processus sont présentées à la figure 1. Les principales étapes sont décrites brièvement dans les pages qui suivent (pp. 1–5) et certaines d'entre elles sont décrites plus en détail dans les sections subséquentes (sections 2 à 4).

Il est à noter que les étapes de quadification et *threadification* sont réalisées à l'aide de l'outil `sorcerer` [Par94, Par96]. Les formes intermédiaires décrites plus loin pour ces étapes sont donc en fait toutes représentées sous forme d'arbres `sorcerer`.

### Front-end

Le *front-end* a été développé par L. Augustsson et nous a été fourni par les gens du MIT. En entrée, le *front-end* reçoit des programmes écrits dans la notation décrite dans [NAH+95] et effectue un certain nombre de transformations et optimisations:

- *Desugaring/pattern matching*
- *Map/reduce optimization*
- *Redundant case elimination*
- *Wrapper generation and inlining*
- *Basic beta reduction and abstraction*
- *Lambda Lifting*

En sortie, le *front-end* produit un fichier contenant une représentation intermédiaire du programme, représentation à la LISP décrite dans [Zho94]. Les éléments clés de cette représentation sont les suivants:

- L'identification unique des identificateur a été effectuée.
- Les types utilisés dans le programme sont indiqués. Pour chaque type, on a son arité (polymorphisme), son nom et les constructeurs qui lui sont associés. Pour chaque constructeur, on ne connaît que l'arité.

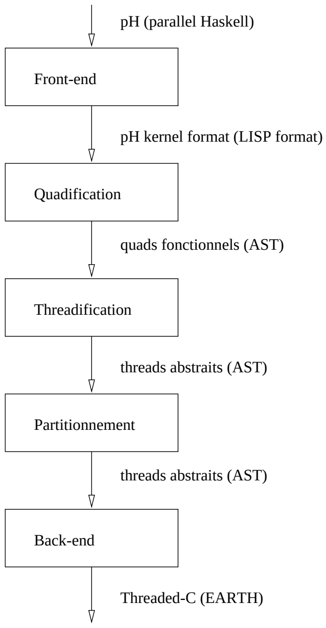


Figure 1: Les grandes lignes du processus de compilation de `pH`

- La valeur (expression ou fonction) associée à chacun des identificateurs est indiquée. Des informations de types sont parfois, mais pas toujours, associées aux valeurs.
- Les vérifications de type ont été effectuées et le programme est supposément bien typé. Malheureusement, certaines informations de types sont manquantes. Une nouvelle analyse étape d'analyse de type (après quadification) a été développée.
- Les fonctions locales à d'autres fonctions ont été *lambda-liftées*.

# Quadification

Le processus de quadification transforme la forme intermédiaire reçue du *front-end* dans une forme intermédiaire style *quads fonctionnels* [Tra88a, AA89]. Essentiellement, ceci signifie qu'après cette étape toute sous-expression est associée à un nom unique; en d'autres mots, il n'y a donc pas d'expressions dont les sous-expressions sont elles-mêmes des expressions complexes. Par exemple, soit une expression de la forme suivante:

```
let
   x = y * (w + z)
   r = x + (2 * y)
in
   x + r
```

Cette expression est alors transformée de la façon suivante (rappelons qu'une expression `let`, en `pH`, introduit des variables qui sont possiblement mutuellement récursives):

```
let
  t1 = w + z
  x  = y * t1
  t2 = 2 * y
  r  = x * t2
  t3 = x + r
in
  t3
```

Le processus de quadification est présenté plus en détails à la section 2.

## Threadification

L'objectif du compilateur en cours de développement est de générer un programme pouvant être exécuté sur une machine parallèle multi-contextes tout en respectant la sémantique indulgente (non-stricte mais non-paresseuse) du langage `pH`. Le processus de *threadification* peut être vu comme une transformation naïve du programme quadifié de façon à obtenir un programme défini en termes de *thread parallèles* et assurant le respect de la sémantique indulgente.

Plus précisément, à l'étape de *threadification*, chaque *quad* conduit à la génération de 1 ou plusieurs *threads*. Les *threads* ainsi obtenus sont donc de granularité très fine et sont, à toute fin pratique, semblables aux *threads* obtenus dans une compilation pour une machine parallèle à flux de données [Nik93].

L'exemple permettant d'illustrer le plus clairement ce processus est celui pour un appel de fonction. Soit $f$ une fonction définie par le programme et ayant $n$ arguments, qu'on suppose tous non-stricts. Soit alors un *quad* de la forme suivante:

```
x = f(x1, ..., xn)
```

Puisque l'appel doit se faire de façon non-stricte, donc que le résultat doit pouvoir être retourné de façon possiblement indépendante des arguments, un tel appel générera, *grosso modo*, les *threads* suivants, où `av` dénote la structure par l'intermédiaire de laquelles les arguments seront communiqués à la fonction:

```
THREAD  av, x <== NS_APPLY[n] f
THREAD  ENVOYER_ARG av, 1, x1
  ...
THREAD  ENVOYER_ARG av, n, xn
```

L'indication "`av, x <==`" signifie que deux résultats seront retournés par l'appel de fonction indiqué et que dans les deux cas ces résultats seront retournés de façon non-immédiate via une forme de *continuation*. Tout d'abord, lorsque la structure devant être utilisée pour transmettre les arguments aura été créée par l'appelé, un signal inter-activation (nécessairement, la fonction appelée n'est pas dans la même activation que celle où l'appel est effectué) approprié sera envoyé, permettant ensuite aux *threads* servant à transmettre les arguments de s'exécuter. De même, lorsque le résultat de la fonction sera disponible, une continuation appropriée sera activée (dénotée localement par `x`).

Un appel de fonction `x = f(x1, x2)` peut donc être représenté graphiquement tel qu'illustré à la figure 2, où la partie du bas représente la forme après *threadification*.

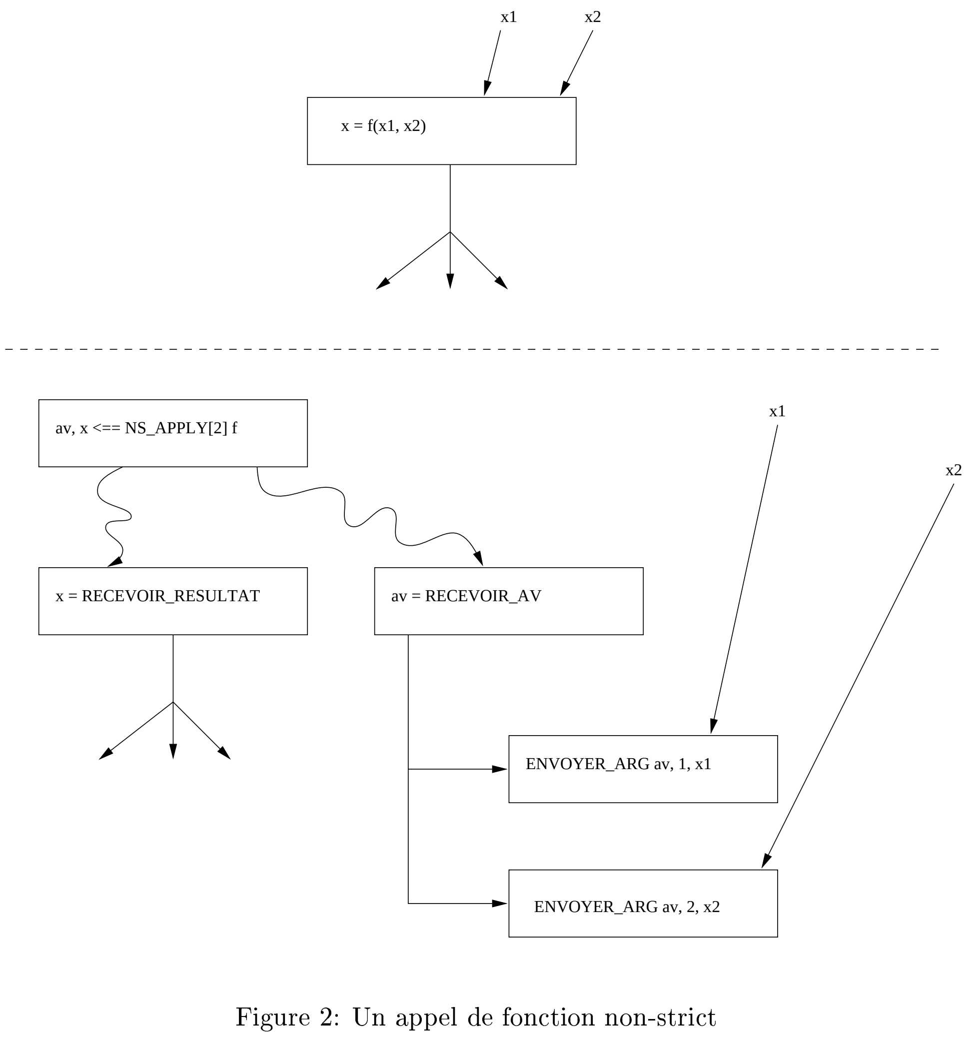


Figure 2: Un appel de fonction non-strict

Le processus de *threadification* effectue aussi les ajouts nécessaires pour assurer la terminaison correcte des instructions et des blocs de façon à réaliser la déallocation des blocs d'activation pour les appels de fonction (e.g., vecteurs pour le transfert des arguments). Plus de détails sont présentés à la section 3.

## Partitionnement

L'objectif du processus de partitionnement est d'obtenir des *threads* de taille supérieure tout en préservant la sémantique indulgente. En d'autres mots, l'objectif est de réduire les coûts de synchronisation dynamique en diminuant le nombre de *threads* et en introduisant des synchronisations statiques via l'ordonnancement des instructions à l'intérieur d'un *thread*.

Par exemple, soit les deux *threads* suivants:

```
THREAD t1 = x + y
THREAD t2 = 2 * t1
```

Clairement, la synchronisation dynamique entre ces deux *threads* (le premier signale le second lorsque `t1` est disponible) est inutile et pourrait être remplacée par un ordonnancement statique des instructions associées à l'intérieur d'un même *thread*. Le processus de partitionnement pourrait donc combiner ces deux *threads* de façon à obtenir le *thread* suivant:

```
THREAD t1 = x + y; t2 = 2 * t1
```

Évidemment, il faut aussi ajuster les signaux associés: le nouveau *thread* doit maintenant signaler les *threads* qui utilisent `t1` ainsi que ceux qui utilisent `t2`.

Le processus de partitionnement est présenté plus en détails à la section 4.

## Génération de code (*back-end*)

À court terme, nous prévoyons développer un générateur de code pour la machine parallèle multi-contextes `EARTH` [HMa95]. L'interface avec cette machine se fait via le langage `THREADED-C`, qui est essentiellement du `C` augmenté d'un ensemble de macros pour les aspects liés aux *threads*.

# 2 La quadification

## 2.1 Syntaxe abstraite après quadification

La figure 3 présente la syntaxe abstraite de la forme intermédiaire produite par le processus de *quadification*.

### 2.1.1 Quelques points à souligner/expliquer

- Pour l'instant, on suppose qu'un module est un programme complet et indépendant, i.e., un ensemble de définition de fonctions et un ensemble d'instructions.
- Le module/programme conserve les définitions de type de données. Dans le cas de ces définitions de type, on a:

$p \in \mathit{ProgramQ}$

  où $p ::= \texttt{module}\ [dt_1;\ \ldots;\ dt_l] < f_1 = lam_1; \ldots; f_m = lam_m > ins^{par(1)}$

$dt, dt_i \in \mathit{DefnType}$

  où $dt_i ::= \ t_i = K_{i1}^{a_{i1}} \mid \ldots \mid K_{im_i}^{a_{im_i}}$

$lam, lam_j \in \mathit{Lambda\text{-}expression}$

  où $lam ::= \ \backslash x_1. \cdots \backslash x_n.\ b$

$ins^{par(m)}, ins_i^{par(m)} \in \mathit{InstrQ}^{par(m)}$

  où $ins^{par(m)} ::=$

    $ins_1; \ldots; ins_k \ (k \geq m)$

$ins, ins_i \in \mathit{InstrQ}$

  où $ins ::=$

    $x = e \ \mid$

    $ins_1^{par(1)}$ — $\ldots$ — $ins_k^{par(1)}$ $(k \geq 2)$

$e, e_i \in \mathit{Expr}$

  où $e ::=$

    $se \ \mid$

    $p_i(se_1, \ldots, se_{n_i}) \ \mid$

    $\mathit{IVSTORE}(x_1, se, x_2) \ \mid$

    $\mathit{IVFETCH}(x_1, se) \ \mid$

    $K_{ij}^{a_{ij}}(x_1, \ldots, x_{a_{ij}}) \ \mid$

    $f_i(x_1, \ldots, x_{n_i}) \ \mid$

    $\texttt{case}\ x\ \texttt{of}\ (K_{ij}^{a_{ij}}\ x_1\ \cdots\ x_{a_{ij}}) \rightarrow b_j \ldots \ \mid$

    $\texttt{case}\ x\ \texttt{of}\ (K_{ij}^{a_{ij}}\ x_1\ \cdots\ x_{a_{ij}}) \rightarrow b_j \ldots\ \texttt{else}\ b \ \mid$

    $\texttt{from}\ x\ \texttt{fetch}\ (K_{ij}^{a_{ij}}\ x_1\ \cdots\ x_{a_{ij}})\ \texttt{in}\ b$

$b, b_i \in \mathit{BlocQ}$

  où $b ::= \ \{\ ins^{par(0)}\ \texttt{in}\ se\ \}$

$se, se_i \in \mathit{SimpleExpr}$

  où $se ::= \ c \ \mid \ x$

| | |
|---|---|
| $c \in \mathit{Const}$ | Constantes (divers types) |
| $x, x_i \in \mathit{VarId}$ | Identificateurs de variables |
| $f, f_i \in \mathit{FuncId}$ | Identificateurs de fonctions |
| $t, t_i \in \mathit{TypeId}$ | Identificateurs de types |
| $K_{ij}^{a_{ij}} \in \mathit{ConstrId}$ | Constructeurs de types algébriques |
| $p_i \in \mathit{PrimId}$ | Opérateurs primitifs stricts |

Figure 3: Syntaxe abstraite du langage `kpH` après la quadification

- Une définition de type $dt_i$ introduit un nouveau type identifié par $t_i$ et possédant $m_i$ constructeurs ($m_i \geq 0$): $K_{i1}, \ldots, K_{im_i}$.
- À chaque constructeur $K_{ij}$ est associé son arité $a_{ij}$.

- À cause de la présence d'instructions avec effets de bord, une forme d'ordonnancement et séquentialisation explicite des instructions est présente dans le langage `pH`. Par défaut, toutes les instructions sont implicitement parallèles — $ins_1; \ldots; ins_k$ — et (possiblement) mutuellement récursives. Par contre, il est possible d'indiquer qu'une série d'instructions devront être évaluées dans un ordre spécifique. Ceci est indiqué par des instructions de la forme:

$$ins_1^{par(1)} \text{ — } \ldots \text{ — } ins_k^{par(1)} \ (k \geq 2)$$

  Une telle construction signifie que l'instruction $ins_{j+1}$ ne pourra commencer à s'exécuter que lorsque *toute* activité à l'intérieur de $ins_j$ sera terminée. [1]

- Les opérateurs primitifs $p_i$ sont tous des opérateurs dont les arguments sont stricts. L'ensemble contient les éléments définissant les opérateurs arithmétiques (entiers et flottants), les opérateurs logiques, ainsi des primitives permettant de créer/allouer des I/M-structures.

- Certains opérateurs primitifs possèdent des arguments stricts mais doivent être traités de façon spéciale par le processus de *threadification*.

  1. Les opérateurs exécutés uniquement pour leur effet de bord.
  2. Les opérateurs *split-phase*, pour lesquels le résultat n'est pas disponible immédiatement mais seulement via une continuation.

  Dans ce qui suit, seuls les opérateurs `IVSTORE` et `IVFETCH` seront considérés, puisque les autres opérateurs seraient traitées de façon tout à fait équivalente.

- Une instruction `case` permet d'examiner une valeur `x` provenant d'un type algébrique et, selon le constructeur associée à cette valeur, de choisir d'exécuter une branche ou une autre ($b_j$); de plus, ceci permet aussi d'introduire un identificateur, local au bloc choisi, associé à chacun des champs de la structure. Un tel identificateur peut être "_", auquel cas cela signifie que le champ associé n'est pas requis.

- Le *front-end* contient des expression `while`. Après la quadification, ces expressions sont supprimées/éliminées et remplacées par des fonctions récursives.

# 3 La *threadification*

## 3.1 Le modèle d'exécution

Pour mieux comprendre le processus de *threadification* et partitionnement, il est important de tout d'abord comprendre le modèle dynamique d'exécution multi-contextes.

Un programme `pH` étant un programme parallèle, son exécution donne lieu, non pas à une pile d'activation, mais plutôt à un *arbre* d'activation. Dans le sous-ensemble de `pH` que nous utilisons, chaque activation correspond à un appel de fonction définie par

[1] Pour une présentation plus détaillée et formelle de cette notion, voir [AAS95].

le programme. Le *bloc* d'activation d'une fonction contient l'espace pour les variables locales, les arguments, les *slots* de synchronisation, l'adresse de retour, etc. L'idée d'arbre d'activation indique qu'à un instant donné plusieurs activations peuvent contenir des instructions prêtes à s'exécuter.

Un *thread* consiste en une séquence d'instructions, instruction pour lesquelles l'ordre d'exécution est déterminée de façon statique (ordre total). Par contre, l'ordre d'exécution des divers *threads* est définie de façon dynamique, en fonction des dépendances (ordre partiel) et synchronisations.

Une activation peut aussi contenir, à un instant donné, plusieurs *threads* actifs. Un *thread* est prêt à s'exécuter lorsque toutes ses données sont disponibles et qu'il pourra donc s'exécuter sans s'arrêter. Les *threads* locaux à une activation peuvent communiquer/se synchroniser entre eux via les variables et les *slots* de synchronisation locales à l'activation (coût faible). Quant aux *threads* provenant d'activations différentes, ils communiquent entre eux via l'envoi de messages (coût plus élevé); ces envois de message vont correspondre, plus loin, à des instructions d'appel de fonction (création d'une nouvelle activation), accès à une structure de données dynamique non-locale, etc.

Les structures de données dynamiques manipulées par un programme `pH` sont, ultimement, toutes représentées par des I/M-structures [ANP89]. Le tas associé à l'ensemble de ces structures est effectivement réparti/distribué entre les divers processeurs. Toutefois, ceci reste invisible au programmeur `pH` (modèle logique = mémoire partagée; modèle physique = mémoire distribuée). La principale particularité d'une I-Structure est qu'elle permet un accès avec synchronisation implicite, idéal pour un modèle d'exécution parallèle. Ainsi, un accès à une composante se fait de façon *split-phase*: une demande est faite, laquelle indique une continuation où devra être retournée le résultat lorsque disponible; si l'élément demandé n'est pas disponible, la demande est mise en attente jusqu'à ce que l'information demandée devienne disponible.

Quant à la notion de bloc mentionnée plus bas, elle correspond à la notion statique telle qu'on la retrouve dans tous les langages structurés modernes. Formellement, du point de vue dynamique, tous les *threads* associés à une même activation peuvent communiquer entre eux tel qu'indiqué plus haut, indépendamment qu'ils soient dans le même bloc statique ou non. Toutefois, comme on le verra, une distinction sera quand même introduite dans le but d'expliciter certaines informations utiles pour le partitionnement et le respect de la sémantique non-stricte.

## 3.2 Les instructions abstraites

### 3.2.1 Présentation des instructions

Les diverses instructions abstraites sont les suivantes:

- $y = se$

  Copie ou association d'une constante à un identificateur.

- $y = p_i(se_1, \ldots, se_{n_i})$

  Appel d'une fonction primitive stricte et non *split-phase*.

- $av, y_1, y_3 \Leftarrow \texttt{NS\_apply}[c]\ f_i\ (x_1, \ldots x_k)$

  Appel non-strict (*NS*) d'une fonction définie par le programme:

  - La fonction appelée est $f_i$;
  - $x_1, \ldots, x_k$ indiquent des arguments stricts;

- En plus des arguments stricts indiqués, la fonction utilise $c$ arguments non-stricts ($c \geq 0$).
- $av$ dénote la continuation pour recevoir le vecteur utilisé pour transmettre les arguments;
- $y_1$ dénote la continuation pour recevoir le résultat;
- $y_3$ dénote la continuation pour recevoir le signal indiquant que toute activité à l'intérieur de la fonction $f_i$ est terminée.

- `Envoyer_av` $x$

  Permet de retourner à la fonction appelante la structure $x$ qui devra être utilisée pour transmettre les arguments à la fonction appelée.

- `Envoyer_arg` $av, i, x_2$

  Permet à l'appelant de transmettre l'argument $x_2$ comme $i^{\text{ème}}$ argument non-strict de la fonction ayant retourné $av$ comme vecteur d'arguments.

- `Envoyer_res` $x_1$

  Permet à la fonction appelée de retourner $x_1$ comme résultat de la fonction.

- `Signaler_fin`

  Permet à la fonction appelée d'indiquer à l'appelant que toute activité est terminée.

- `Evaluer` $(K_{i1}^{a_{i1}} : b_1, \ldots, K_{im_i}^{a_{im_i}} : b_{m_i})[x_1, x_2]$ `selon` $x_3$

  Examine le constructeur associée à $x_3$ et, en fonction de ce constructeur, exécute l'un des blocs indiqués. Pour le bloc sélectionné, $x_1$ indiquera la variable où devra être transmis le résultat alors que $x_2$ indiquera ce qui devra être signalé pour indiquer que l'activité du bloc est terminée.

- `Envoyer_var` $x_1$ `a` $(K_{i1}^{a_{i1}} : b_1, \ldots, K_{im_i}^{a_{im_i}} : b_{m_i})$ `selon` $x_2$

  Examine le constructeur associée à $x_2$ et, en fonction de ce constructeur, envoie la variable $x_1$ au bloc sélectionné.

- `Envoyer_res_bloc`$[x_2]$ $x_1$

  Retourne la valeur de $x_1$ comme résultat du bloc courant. (Le signal inter-bloc est envoyé à $x_2$.)

- `Signaler_fin_bloc`$[x]$

  Signale que toute activité à l'intérieur du bloc courant est maintenant terminée. (Le signal inter-bloc est envoyé à $x$.)

- $y =$ `IConstr` $K_{ij}^{a_{ij}}$

  Associe à $y$ une structure de données dynamique vide (I-structure) dont le constructeur associé est $K_{ij}^{a_{ij}}$.

- `IVStore` $x_1, se, x_2$

  Range la valeur de $x_2$ dans la case $se$ de $x_1$ (effet de bord sur IStructure).

- $y \Leftarrow$ `IVFetch` $x, se$

  Obtient la case $se$ de la structure $x$ et retourne le résultat via la continuation $y$.

- `IFree` $x$

  Détruit la structure de donnée dynamique associée à $x$.

- `IntPrint` $x$

  Imprime la valeur entière associée à $x$.

- `Stop`

  Termine l'exécution du programme.

### 3.2.2 Instructions *split-phase* et pseudo-instructions

- Certaines instructions produisent un/des résultat mais pas de façon immédiate, e.g., `IFETCH`, `NS_APPLY`. Ces instructions sont des instructions *split-phase* qui sont caractérisées, entre autres, par des continuations qui indiquent où devra être envoyé le résultat.

- Conceptuellement, la réception d'un résultat provenant d'une instruction *split-phase* peut être dénotée à l'aide d'une pseudo-instruction de la forme suivante:

  x = RECEVOIR_XXX

  Une telle pseudo-instruction est utilisée comme une sorte de *placeholder* pour recevoir des signaux associées au résultat produit/retourné par l'instruction *split-phase* correspondante. Or, comme on le verra plus loin, parce qu'un *thread* doit toujours pouvoir se compléter lorsque son exécution s'amorce, cela signifie que de telles pseudo-instructions ne peuvent jamais apparaître au milieu d'un *thread* et doivent toujours apparaître au *début du thread.* On peut donc éliminer de telles pseudo-instructions et les remplacer, comme nous le ferons plus bas, par un attribut du *thread.* En d'autres mots, ces instructions de réception seront implicites dans notre représentation utilisée pour les *threads.*

## 3.3 Ce qu'est un *thread*

### 3.3.1 Caractéristiques générales d'un *thread*

1. L'exécution du *thread* ne peut s'amorcer que lorsque tous les signaux appropriés ont été reçus, i.e., que lorsque l'on est certain que toutes les dépendances (données et contrôles) sont effectivement satisfaites.

2. Lorsque l'exécution d'un *thread* s'amorce, alors elle peut se poursuivre sans interruption jusqu'à la fin. Après qu'un *thread* a été exécuté, tous les cotés gauche sont effectivement définis et disponibles.

3. Les signaux indiquant que les cotés gauches sont disponibles ne sont envoyés qu'à la toute fin du *thread*, jamais en cours d'exécution du *thread.*

### 3.3.2 Spécification d'un *thread*

Soit $t$ un *thread* défini par:

$\texttt{thread}\{x_1, \ldots, x_n\}\{y_1, \ldots, y_n\}\langle s\rangle\ ins_1; \ldots, ins_n$

Les attributs caractérisant ce *thread* sont les suivants:

- $\{x_1, \ldots, x_n\}$: Indique les signaux, provenant de l'activation locale de la procédure, requis avant que le *thread* puisse s'exécuter.

  Ces signaux sont utilisés pour signaler/indiquer les dépendances entre les variables locales des blocs d'une même activation. Par exemple: transmission des variables libres dans un bloc associé à l'une des branches d'une expression conditionnelle; envoi du résultat produit par la branche sélectionnée pour une instruction conditionnelle. Ils sont aussi utilisés pour les signaux locaux de terminaison (cf. plus bas).

- $\{y_1, \ldots, y_n\}$: Indique les signaux provenant d'autres activations, i.e., ne provenant pas de l'activation courante. Par exemple, ces signaux sont utilisés pour les tâches suivantes:

  – Recevoir (dans la fonction appelante) l'adresse du vecteur par lequel les arguments seront transmis à l'appelée.
  – Recevoir les arguments (dans la fonction appelée);
  – Recevoir le résultat (dans la fonction appelante);
  – Recevoir le signal de fin d'activation (dans la fonction appelante).
  – Recevoir le résultat provenant d'une instruction d'accès *split-phase* (pour les I-structures).

- $s$: si présent, indique qu'un signal de terminaison doit être généré. Divers cas sont possibles:

  – $s = \star$: Indique qu'un signal de terminaison devra être envoyé au *thread* $t$ tel que $\star \in \mathit{signauxInterBloc}(t)$. Ce dernier *thread* devra donc attendre que tous les *threads* $t_i$ contenant $\star$ dans $\mathit{signauxTerminaison}(t_i)$ aient terminé pour pouvoir être activé.
  – $s \neq \star$: Dans ce cas, il s'agit d'un signal de terminaison plus spécifique, utilisé pour les instructions séquentielles (avec barrières). Dans ce cas, un signal unique demande a être reçu.

- $ins_1; \ldots; ins_n$: la séquence d'instructions associées au *thread*.

## 3.4 Syntaxe abstraite après *threadification* et partitionnement

La figure 4 présente la syntaxe abstraite du langage `kpH` après par les processus de *threadification* et *partitionnement*.

### Explication/motivation du langage abstrait

- Soit un *thread* de la forme suivante, *thread* qui ne contient aucune instruction et où `x` dénote le nom du `j`$^{ième}$ argument de la fonction dans lequel ce *thread* apparaît:

  `thread`$\{\}\{x\}\langle\rangle$

  Un tel *thread* peut être interprété, conceptuellement/intuitivement, comme un *thread* de la forme suivante:

```
THREAD  x = RECEVOIR_ARG[j]
```

$p \in \mathit{ProgramT}$

où $p ::= \texttt{module}\,[dt_1;\ \ldots;\ dt_l]\ < f_1 = b_1;\ \ldots;\ f_n = b_n >\ b_0$

$dt, dt_i \in \mathit{DefnType}$

où $dt_i ::=\ t_i = K_{i1}^{a_{i1}} \mid \ldots \mid K_{im_i}^{a_{im_i}}$

$b, b_i \in \mathit{BlocT}$

où $b ::=\ thr_0\ thr_1\ \ldots\ thr_k$

$thr, thr_i \in \mathit{Thread}$

où $thr ::=\ \texttt{thread}\{x_1, \ldots, x_n\}\{y_1, \ldots, y_m\}\langle s\rangle\ ins_1; \ldots; ins_k$

$ins, ins_i \in \mathit{InstrT}$

où $ins ::=$

$y = se\ \mid$
$y = p_i(se_1, \ldots, se_{n_i})\ \mid$
$av, y_1, y_3 \Leftarrow\ \texttt{NS\_apply}[c]\ f_i\ (x_1, \ldots\ x_k)\ \mid$
$\texttt{Envoyer\_av}\ x\ \mid$
$\texttt{Envoyer\_arg}\ av, i, x_2\ \mid$
$\texttt{Envoyer\_res}\ x_1\ \mid$
$\texttt{Signaler\_fin}\ \mid$
$\texttt{Evaluer}\ (K_{i1}^{a_{i1}} : b_1, \ldots, K_{im_i}^{a_{im_i}} : b_{m_i})[x_1, x_2]\ \texttt{selon}\ x_3\ \mid$
$\texttt{Envoyer\_var}\ x_1\ \texttt{a}\ (K_{i1}^{a_{i1}} : b_1, \ldots, K_{im_i}^{a_{im_i}} : b_{m_i})\ \texttt{selon}\ x_2\ \mid$
$\texttt{Envoyer\_res\_bloc}[x_2]\ x_1\ \mid$
$\texttt{Signaler\_fin\_bloc}[x]\ \mid$
$y = \texttt{IConstr}\ K_{ij}^{a_{ij}}\ \mid$
$y \Leftarrow \texttt{IVFetch}\ x, se\ \mid$
$\texttt{IVStore}\ x_1, se, x_2\ \mid$
$\texttt{IFree}\ x\ \mid$
$\texttt{IntPrint}\ x\ \mid$
$\texttt{Stop}$

$se, se_i \in \mathit{SimpleExpr}$

où $se ::=\ c\ \mid\ x$

| | |
|---|---|
| $c \in \mathit{Const}$ | Constantes (divers types) |
| $x, y, x_i, y_i \in \mathit{VarId}$ | Identificateurs de variables |
| $s \in \mathit{VarId} \cup \{\star\}$ | Identificateurs de terminaison |
| $av, av_i \in \mathit{VarId}$ | Identificateurs pour vecteur d'arguments |
| $f, f_i \in \mathit{FuncId}$ | Identificateurs de fonctions |
| $t, t_i \in \mathit{TypeId}$ | Identificateurs de types |
| $K_{ij}^{a_{ij}} \in \mathit{ConstrId}$ | Constructeurs |
| $p_i \in \mathit{PrimId}$ | Opérateurs primitifs stricts et immédiats |

Figure 4: Syntaxe abstraite du langage `kpH` après *threadification* et partitionnement

- Soit une instruction, avant *threadification*, de la forme suivante:

  ```
  x = IVFETCH y, i
  ```

  Une telle instruction nécessite tout d'abord une instruction pour amorcer le *IVFetch*. Toutefois, comme cette instruction est *split-phase*, le résultat devant être associé à `x` ne sera pas immédiatement disponible. Dans ce cas, deux *threads* seront générés:

  `thread`$\{\}\{\}\langle\rangle$ $x \Leftarrow$ `IVFetch` $y, i$
  `thread`$\{\}\{x\}\langle\rangle$

  Le rôle du deuxième *thread* est de servir de *placeholder* pour la réception du signal indiquant la complétion du `IVFetch`. L'endroit où envoyer le résultat (continuation) est indiqué par "$x \Leftarrow$".

  Ces *threads* peuvent donc être interprétés comme suit:

  ```
  THREAD  x <== IVFETCH y, i
  THREAD  x = RECEVOIR_IVFETCH
  ```

- Soit une instruction, avant *threadification*, de la forme suivante:

  $x = f(x_1, \ldots, x_k, y_1, \ldots, y_n)$

  Supposons pour simplifier que $x_1, \ldots, x_k$ sont des arguments non-stricts alors que $y_1, \ldots, y_n$ sont des arguments stricts. Alors cette instruction génerera les *threads* suivants:

  `thread`$\{\}\{\}\langle\rangle$ $av, x, fin \Leftarrow$ `NS_apply`$[k]$ $y_1, \ldots, y_n$ $()$
  `thread`$\{\}\{av\}\langle\rangle$
  `thread`$\{\}\{\}\langle\rangle$ `Envoyer_arg` $av, 1, x_1$
  $\ldots$
  `thread`$\{\}\{\}\langle\rangle$ `Envoyer_arg` $av, k, x_k$
  `thread`$\{\}\{x\}\langle\rangle$
  `thread`$\{\}\{fin\}\langle\rangle$ `IFree` $av$

  Les diverses continuations sont les suivantes:

  – `av`: Vecteur via lequel les arguments non-stricts seront transmis.
  – `x`: Résultat produit/retourné par la fonction.
  – `fin`: Signal de fin d'exécution de la fonction (qui assure que toute activité est terminée).

  Le *thread* recevant `x` — `thread`$\{\}\{x\}\langle\rangle$ — peut être intuitivement interprété comme une réception du résultat:

  ```
  THREAD  x = RECEVOIR_RES
  ```

### 3.4.1 Activation d'un *thread*

De façon à être activé, un *thread* $t$ doit recevoir les signaux suivants:

- Tous les signaux indiquant que les variables dans *signauxInterActivation*$(t)$ sont disponibles ont été reçus.

- L'ensemble de signaux permettant d'assurer que toutes les variables correspondant à des dépendances inter-bloc sont disponibles, i.e., des dépendances envers des variables non-locales au bloc dans lequel le *thread* est défini.
- L'ensemble des signaux provenant du bloc local assurant que toutes les variables référencées dans les instructions du *thread* sont effectivement disponibles.

### 3.5 Quelques points supplémentaires

Rappelons tout d'abord que l'évaluation indulgente, bien qu'elle semble *spéculative* au niveau des arguments relativement à une approche paresseuse, n'est pas du tout spéculative au niveau des instructions conditionnelles. Ce sont donc les instructions `if` et `case` qui déterminent ce qui devra ou ne devra pas être exécuté.

Par exemple, soit la fonction `pH` suivante:

```
f x y z =
  if x == 0
    then 2 * y + z
  else
    f (x-1) y (z+1)
```

Dans ce cas, l'expression "`2 * y + z`" ne devra être évaluée que lorsque `x` est 0.

La figure 5 illustre deux approches possibles pour la réalisation des instructions conditionnelles. On suppose que, dans les deux cas, c'est le bloc de gauche qui est sélectionné pour exécution. La partie du haut représente la première approche — sans instruction `envoyer/recevoir_var` — alors que celle du bas représente la deuxième approche — où des instructions explicites d'envoi/réception des variables libres sont introduites. Les *blobs* représentent des blocs alors que les cercles à l'intérieur de ces blocs représentent des *threads* utilisant les variables indiquées. Quant aux rectangles à l'extérieur, ils représentent les *threads* qui définissent ces variables.

## 4 Le partitionnement

Cette section présente les principales stratégies de partitionnement introduites pour les langages indulgents. Essentiellement, il s'agit de travaux faits pour la compilation du langage Id [Sch91, SCvE91, TCS92, Coo94, Sch94, SCG95].

### 4.1 Objectif du partitionnement

L'objectif du partitionnement est de regrouper les *threads* de base (ayant presque toujours une seule instruction par *thread*) dans des *threads* les plus gros possibles. Ceci a pour effet de diminuer le nombre de changements de contexte et, donc, de réduire les coûts de synchronisation.

Une particularité importante des *threads* qu'on doit préserver est que lorsqu'un *thread* amorce son exécution, il doit pouvoir continuer, sans interruption ni necessité de recevoir de signaux additionnels, jusqu'à ce que toutes les instructions du *thread* soit exécutées. Ceci signifie entre autres que les instructions d'un *thread* seront ordonnés de façon statique alors que les aspects de synchronisation et/ou ordonnancement dynamique seront restreints aux interaction *entre threads*.

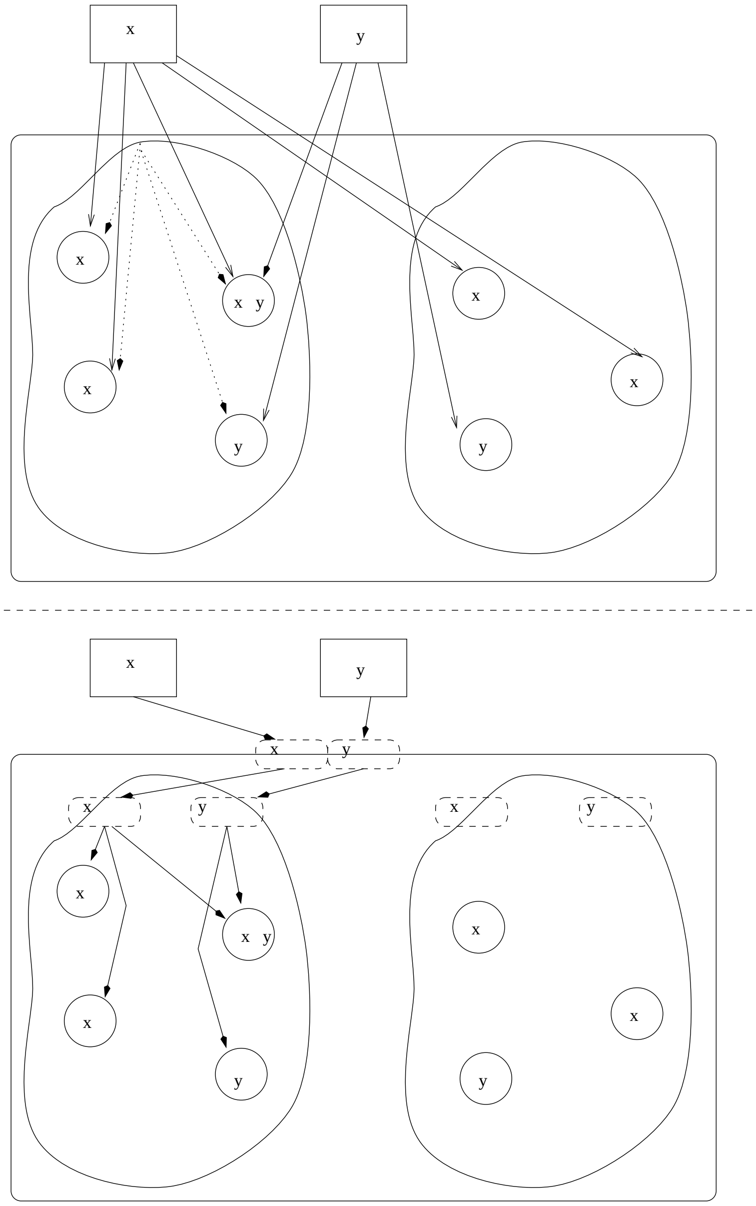


Figure 5: Deux approches pour les instructions conditionnelles

### 4.1.1 Nécessité de l'ordonnancement dynamique

La nécessité de synchronisation et d'ordonnancement dynamique découle de deux aspects:

1. Les communications inter-processeurs (envoi d'arguments, accès aux structures de données dynamique, donc des opérations avec des temps de latence imprévisibles) ainsi que les synchronisation (style producteur/consommateur) associées aux manipulations des structures de données parallèles/distribuées (I-/M- structures).
2. La non-stricticité qui rend parfois impossible de déterminer de façon statique l'ordre d'exécution des instructions.

Dans le cas de la non-stricticité, le contexte dynamique d'exécution peut donc influer sur l'ordonnancement des instructions. Un exemple simple illustrant cela est le suivant:

```
f u v = (u*u, v+v)

g z = let (s, t) = f z s in t

h z = let (s, t) = f t z in s

p z = f z z
```

Selon le contexte d'exécution, le calcul de `u*u` dans `f` peut parfois devoir être fait avant ou après le calcul de `v+v`. Par exemple, dans le contexte de l'appel à `f` fait dans `g`, il faut tout d'abord calculer `u*u` (avec `u=z`), alors que dans le cas de `h` on doit tout d'abord calculer `v+v` (avec `v=z`). Par contre, dans le cas de l'appel fait dans `p`, les deux expressions peuvent être évaluées dans n'importe quel ordre.

Cette possibilité de *feedback* (utilisation d'une partie du résultat d'une fonction comme argument à cette même fonction) est ce qui fait la force (en termes de pouvoir expressif) des langages non-stricts (paresseux ou non). Toutefois, c'est aussi cette possibilité qui entraîne la nécessité de synchronisations dynamique et qui, comme on le verra, rend beaucoup plus difficile la tâche de partitionnement: en l'absence de toute information contredisant une telle possibilité, on doit supposer qu'une telle dépendance existe toujours, d'où la nécessité de synchronisations dynamiques supplémentaires.

### 4.1.2 Le rôle de l'analyse des dépendances

La tâche de l'analyse des dépendances est de déterminer, de façon la plus précise (mais, malheureusement, non-exacte) possible, les dépendances pouvant exister entre les diverses instructions du programme. Divers types de dépendances peuvent être distinguées:

- Dépendance directe certaine: une telle dépendance est présente dans tous les contextes possibles d'exécution et est strictement locale à un bloc.

  **Exemple** La dépendance de `x` vers `y` dans le segment de code suivant est une dépendance certaine directe:

  ```
  y = x + 1
  ```

- Dépendance indirecte certaine: une telle dépendance est aussi présente dans tous les contextes dynamiques possibles. Toutefois, il s'agit d'une dépendance qui est possiblement non-locale au bloc, i.e., qui passe par l'intermédiaire d'instructions situées dans d'autres blocs, ou qui est réalisée via une opérations avec un temps de latence non-prévisible de type *split-phase*.

  **Exemple** Soit le segment de code (avant *thread*ification) suivant:

  ```
  x = IVFetch y, 1

  r = f a x
  ```

  Après *thread*ification, ce code sera transformé (en ignorant les signaux de terminaison) dans le code suivant:

  ```
  s1:: THREAD{}{ }  x <= IVFetch y, 1
  s2:: THREAD{}{x}

  s3:: THREAD{}{  }  av, r <= NS_APPLY[0] f
  s4:: THREAD{}{av}
  s5:: THREAD{}{  }  ENVOYER_ARG av, 1, a
  s6:: THREAD{}{  }  ENVOYER_ARG av, 2, x
  s7:: THREAD{}{r }
  ```

  Les dépendances certaines indirectes seraient alors les suivantes:

  - De `s1` vers `s2`
  - De `s3` vers `s4`
  - De `s3` vers `s7`

  Quant aux dépendances certaines directes, on aurait:

  - Du *thread* définissant `a` vers `s5`
  - De `s4` vers `s5`
  - De `s4` vers `s6`
  - De `s2` vers `s6`

  Graphiquement, ceci pourrait être représenté tel qu'indiqué à la figure 6, où les lignesn en zig-zag représentent des dépendances indirectes.

- Dépendance indirecte potentielle: comme pour une dépendance indirecte certaine, une telle dépendance passe par l'intermédiaire d'instructions d'autres blocs. Toutefois, à la différence des deux précédentes formes, une telle dépendance peut n'exister que dans certains contextes d'utilisation/invocation, donc peut parfois ne pas exister.

  **Exemple** Soit la fonction suivante, vue précédemment:

  ```
  f u v = (u*u, v+v)
  ```

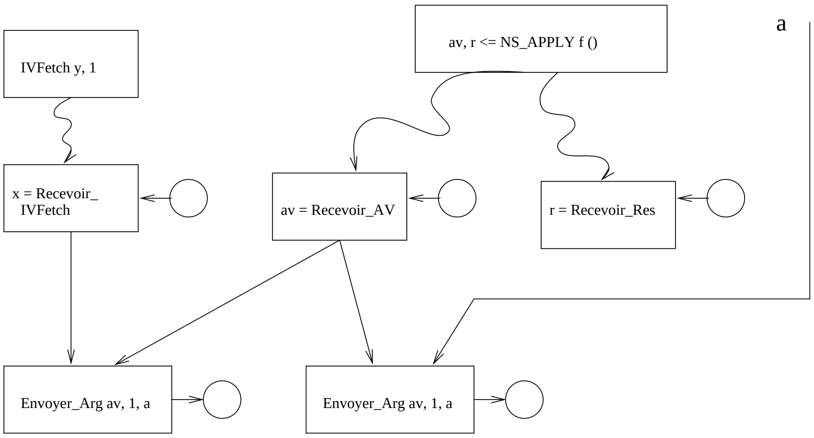


Figure 6: Exemple illustrant les diverses formes de dépendances certaines

En termes de *threads* de base, `f` serait représentée comme suit (en ignorant les signaux de terminaison):

```
s1:: THREAD{}{u}
s2:: THREAD{}{v}
s3:: THREAD{}{ } t1 = u*u
s4:: THREAD{}{ } t2 = v+v
s5:: THREAD{}{ } t3 = IConstr |(,)|
s6:: THREAD{}{ } IVStore t3, 1, t1
s7:: THREAD{}{ } IVStore t3, 2, t2
s8:: THREAD{}{ } ENVOYER_RES t3
```

Graphiquement, on aurait alors la représentation illustrée à la figure 7. Dans cette représentation, les petits cercles représentent des *inlets/outlets*. Un *inlet* représente la cible d'une dépendance qui n'est pas connue statiquement, alors qu'un *outlet* est la source d'une telle dépendance. [2] Intuitivement, les *inlets/outlets* peuvent donc être vus, dans la représentation graphique, comme des sortes d'indicateurs de continuation hors-page.

La tâche de l'analyse des dépendances, dans le cas des dépendances indirectes potentielles, peut être vue comme celle d'annoter ces indicateurs de continuation de façon à déterminer les dépendances effectives pouvant exister. Si une dépendance certaine qui *contredit* une telle dépendance potentielle ne peut pas être inferée, alors on devra supposer que la dépendance potentielle existe.

Dans notre exemple, en l'absence de toute information sur le contexte exacte d'utilisation, on aurait alors les dépendances potentielles suivantes:

- De `s6` vers `s2`
- De `s7` vers `s1`

[2] Dans notre représentation textuelle des *threads*, les *inlets* sont associés aux éléments apparaissant dans l'ensemble *signauxInterActivation* (deuxième ensemble de symboles suivant le mot-clé `THREAD`); quant aux *outlets*, ils sont associés aux opérations `IVStore` ainsi que `ENVOYER_XXX`.

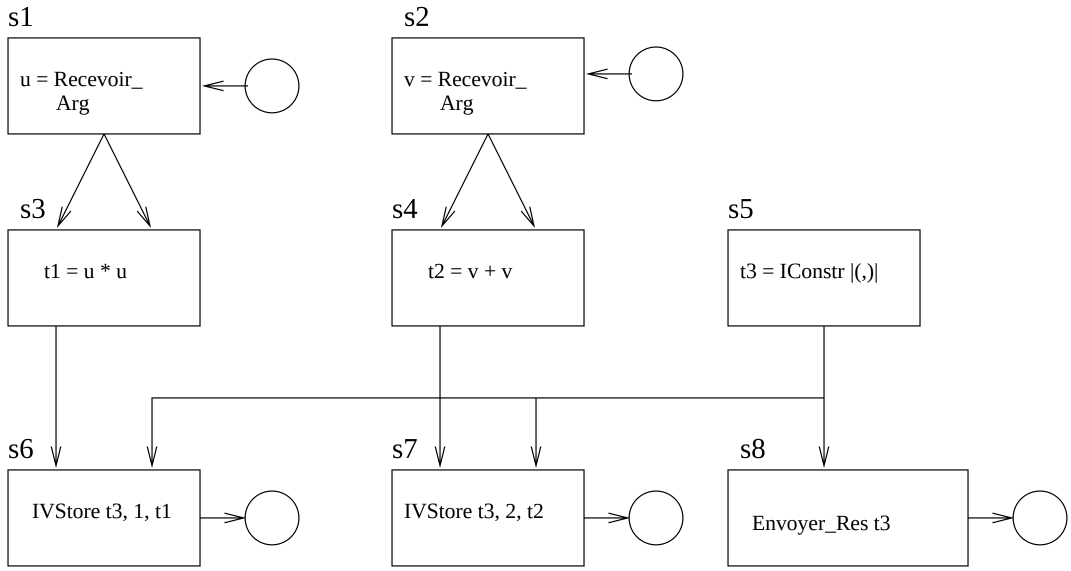


Figure 7: Autre exemple illustrant les diverses formes de dépendances

– De `s8` vers `s1` et `s2`

Certaines dépendances potentielles peuvent être éliminées parce qu'elles sont contredites par des dépendances certaines allant en sens inverse. Par exemple, la dépendance potentielle allant de `s6` vers `s1` peut être ignorée, puisqu'il existe une dépendance (transitive) certaine allant de `s1` vers `s6`. Une telle dépendance peut être ignorée parce que, si elle survient, alors elle conduira à un *deadlock*; or, le pire qui peut arriver lorsqu'on ignore certaines dépendances potentielles est de générer des *thread*s pouvant conduire à des *deadlocks*.

En d'autres mots, un partitionnement correct doit assurer qu'aucune *nouvelle* dépendance cyclique n'est créée. Ainsi, un *thread* `t0` nouvellement créé (obtenu par fusion d'autres *thread*s) ne doit pas dépendre d'un résultat produit par `t0`, sous peine d'en arriver à une situation de *deadlock*. Évidemment si avant le partitionnement le programme contenait déjà un tel *deadlock*, le partitionnement ne peut rien y faire.

Supposons que le seul appel à `f` dans le programme traité soit celui associé à la fonction `g` (p. 16). On pourrait alors éliminer les quatre (4) dépendances potentielles et les *remplacer* par deux dépendances indirectes certaines allant de `s8` vers `s1` et de `s6` vers `s2`. Une analyse (interprocédurale) tenant compte du contexte d'utilisation des fonctions peut donc permettre d'éliminer certaines dépendances potentielles associées à la non-strictícité des fonctions.

Ce sont les dépendances indirectes, potentielles ou certaines, qui obligent à séparer des *thread*s et qui conduisent à la nécessité de synchronisation dynamique entre *thread*s. Règle générale, de façon à obtenir un partitionnement correct, i.e., qui respecte la sémantique du programme initial, il est nécessaire de faire les approximations suivantes:

- Surestimer les dépendances potentielles indirectes: ceci oblige à séparer des *thread*s, ce qui empêche de former des *threads* "*trop gros*" introduisant des cycles qui conduisent à des *deadlocks* qui n'étaient pas présents dans le programme initial.

- Sous-estimer l'ensemble des dépendances certaines indirectes introduites en remplacement des dépendances potentielles: moins de dépendances indirectes certaines signifie moins de DPI pouvant être contredites, donc surestimation des DPI. Évidemment, on parle ici des dépendances indirectes certaines introduites en remplacement de dépendances potentielles; les dépendances indirectes certaines initialement présentes dans le programme (cf. figure 6) doivent être préservées.

## 4.2 Partitionnement des blocs de base

Le premier niveau de partitionnement consiste à générer des *threads* de façon locale, pour chaque bloc de base, indépendamment de toute information provenant d'autres blocs. Rappelons qu'un bloc de base est un ensemble de *threads*/instructions qui sont toutes associées aux mêmes conditions initiant leur exécution, e.g., bloc de base du haut niveau d'une fonction, bloc associé à une branche d'une conditionnelle, etc.

Pour illustrer les diverses stratégies, le même exemple que dans [Sch94] sera utilisé:

```
f a b = (a+b, a*b);
```

En termes de notre représentation des *threads*, cette fonction serait définie comme suit (en ignorant les signaux de terminaison):

```
f = b1

b1::
  THREAD{}{a}
  THREAD{}{b}
  THREAD{}{ }  t1 = a+b
  THREAD{}{ }  t2 = a*b
  THREAD{}{ }  t3 = IConstr |(,)|
  THREAD{}{ }  IVStore t3, 1, t1
  THREAD{}{ }  IVStore t3, 2, t2
  THREAD{}{ }  ENVOYER_RES t3
```

### 4.2.1 Stratégie de base (à la *dataflow*)

La stratégie la plus simple de partitionnement, qui ne nécessite aucun analyse complexe, consiste simplement à intégrer un noeud qui ne possède qu'un seul prédécesseur avec ce prédécesseur. De plus, tous les *threads* qui n'ont aucun prédécesseur peuvent être mis ensemble.

Pour cette stratégie, la fonction résultante pour notre exemple serait la suivante:

```
f = b1

b1::
  THREAD{}{a}
  THREAD{}{b}
  THREAD{}{ } t3 = IConstr |(,)|;
              ENVOYER_RES t3
  THREAD{}{ } t1 = a+b;
  THREAD{}{ } IVStore t3, 1, t1
  THREAD{}{ } t2 = a*b;
  THREAD{}{ } IVStore t3, 2, t2
```

### 4.2.2 Stratégie basée sur les dépendances *inlets/outlets*

Les prochaines stratégies déterminent le partitionnement en fonction des dépendances certaines, directes et indirectes, envers les *inlets* et/ou *outlets*, selon que l'on va de l'avant vers l'arrière et/ou de l'arrière vers l'avant pour déterminer les dépendances.

Dans le cas du partitionnement local, sans aucune information globale ou inter-procédurale, on suppose initialement que tous les *inlets/outlets* sont annotés de façon distincte les uns des autres; ceci signifie alors qu'aucune dépendance certaine *inlet/outlet* n'est connue, donc que toutes les dépendances potentielles (sauf celles contredites par des dépendances certaines en sens inverse) existent.

#### Dépendances envers les *inlets*

Lorsque tous les noeuds d'un possèdent des dépendances envers le même ensemble d'*inlets* — i.e., il existe un chemin allant de chacun de ces *inlets* vers chacun des noeuds considérés — alors ces noeuds peuvent être regroupés en un même *thread*.

Pourquoi un tel regroupement est-il acceptable? Soit deux noeuds $n_1$ et $n_2$ qui dépendent exactement du même ensemble d'*inlets*. Ceci signifie (voir plus loin, fig. 9, p. 22) que ces deux noeuds sont potentiellement influencés par les mêmes *outlets*. Or, s'il existe une dépendance indirecte potentielle entre $n_1$ et $n_2$, une telle dépendance doit nécessairement passer par un de ces *outlets*. Si c'est le cas, alors le noeud $n_1$ (ou $n_2$) s'influence donc lui-même via ce même *outlet*; en d'autres mots, il existe déjà un cycle potentiel!

Dans notre exemple, on aurait alors les *threads* suivants:

```
f = b1

b1::
  THREAD{}{a}
  THREAD{}{b}
  THREAD{}{ }  t3 = IConstr |(,)|
               ENVOYER_RES t3
  THREAD{}{ }  t1 = a+b
               t2 = a*b
               IVStore t3, 1, t1
               IVStore t3, 2, t2
```

Graphiquement, le graphe résultant du calcul des dépendances envers les *inlets* peut être représenté tel qu'illustré à la figure 8. Ce résultat est obtenu en propageant, vers l'avant, les identificateurs d'*inlets*.

Cette propagation des *inlets* permet de déterminer les dépendances certaines allant des inlets vers tous les noeuds, y compris ceux associés à des *outlets*. A priori, on doit supposer qu'il existe une dépendance potentielle allant de n'importe quel *outlet* vers n'importe quel *inlet*. Toutefois, les dépendances certaines, en sens inverse, nous permettent d'éliminer certaines des dépendances potentielles: si un noeud contenant un *outlet* dépend certainement d'un *inlet*, alors il n'y a pas de dépendance potentielle allant du *outlet*; donc, il existe une dépendance potentielle d'un *outlet* vers un *inlet* si le *outlet* ne dépend pas directement du *inlet*. Ceci est illustré à la figure 9.

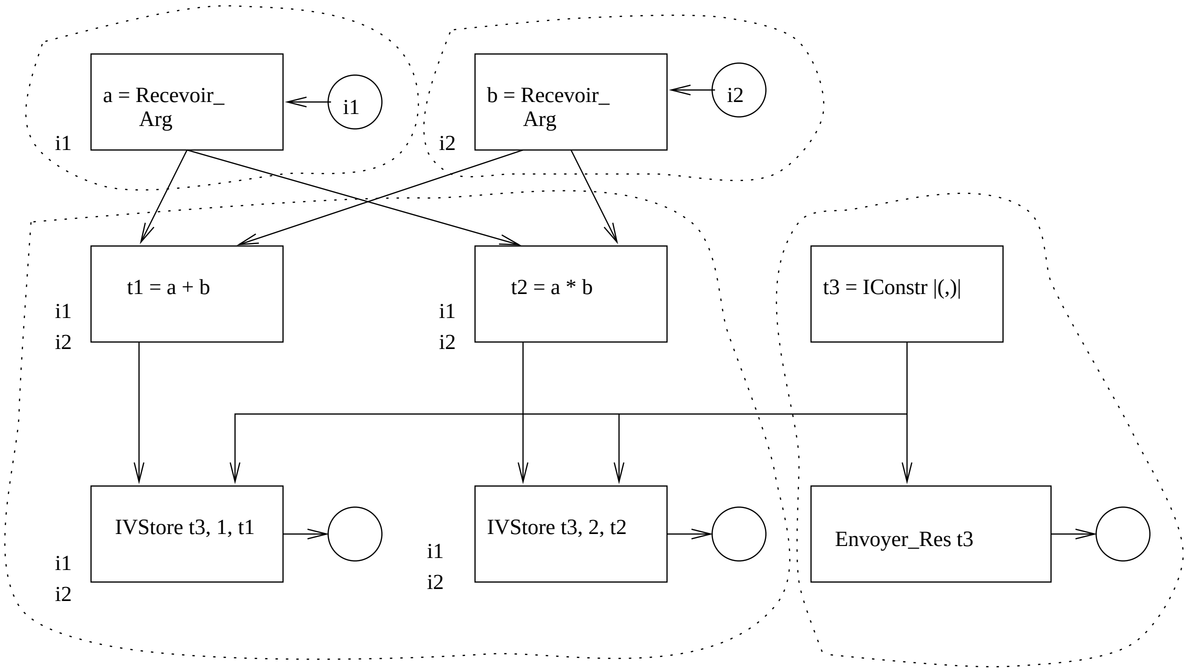


Figure 8: Partitionnement obtenu via la propagation des *inlets*

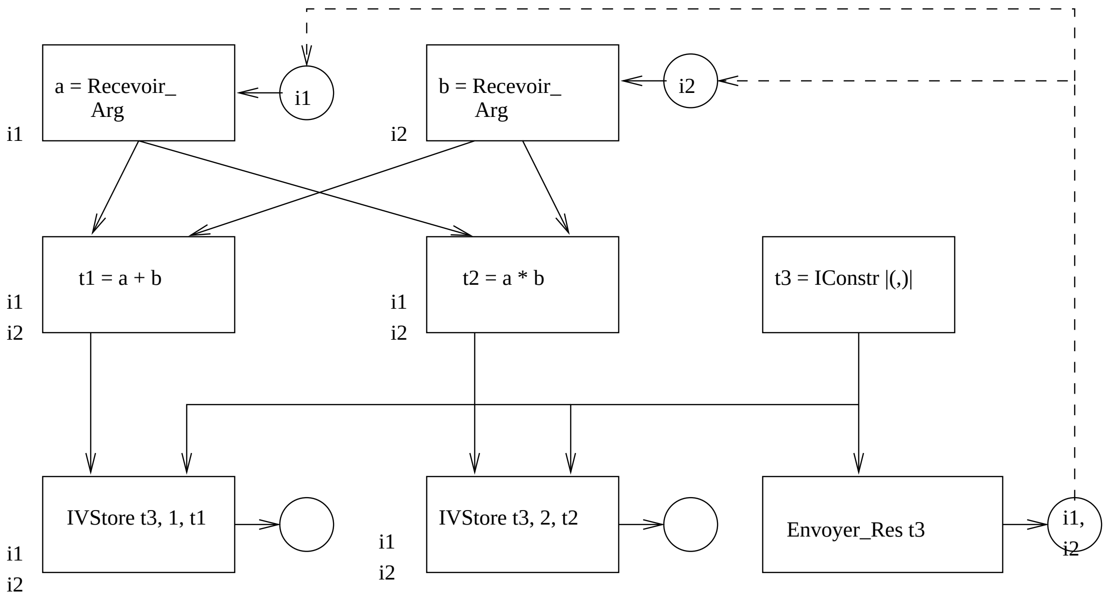


Figure 9: Dépendances potentielles résultant de la propagation des *inlets*

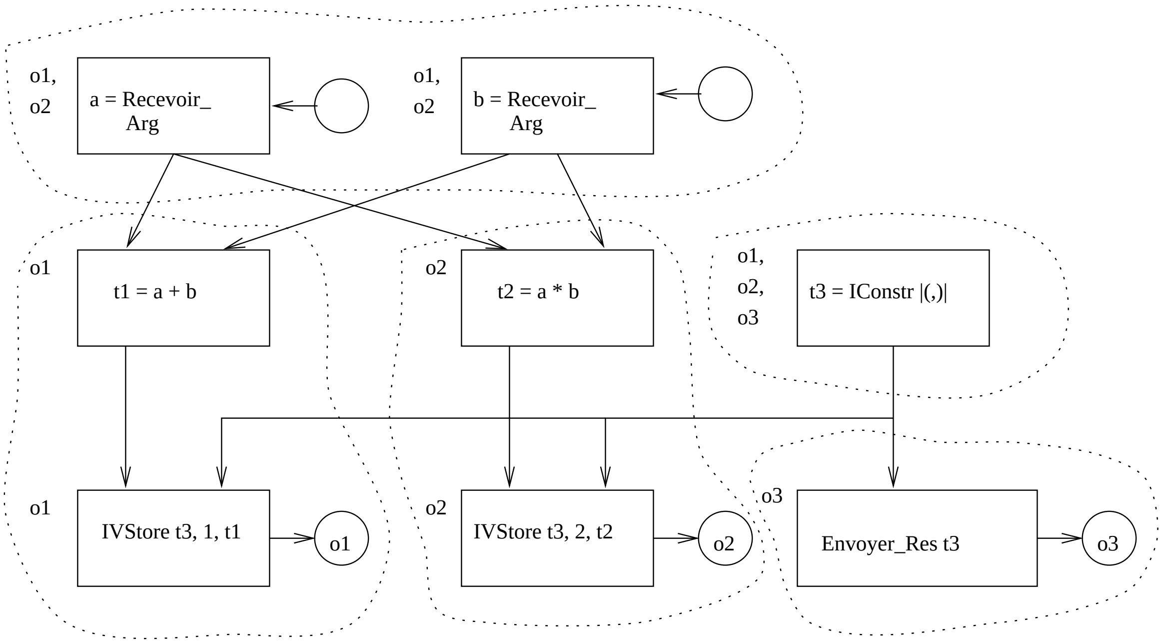


Figure 10: Partitionnement obtenu via la propagation des *outlets*

### Dépendances envers les *outlets*

Lorsqu'un groupe de noeuds possède tous des dépendances inverses envers les mêmes *outlets*, i.e., il existe un chemin du noeud vers chacun des *outlet* de l'ensemble — on dit aussi que les noeuds *dominent* tous le même ensemble d'*outlets* — alors ces noeuds peuvent être regroupés en un même *thread*.

Le raisonnement sur la validité de cette approche est le même que pour les *inlets*, mais en sens inverse.

```
f = b1

b1::
  THREAD{}{a,b}
  THREAD{}{   }  t3 = IConstr |(,)|;
                 ENVOYER_RES t3
  THREAD{}{   }  t1 = a+b
                 IVStore t3, 1, t1
  THREAD{}{   }  t2 = a*b;
                 IVStore t3, 2, t2
```

Graphiquement, les partitions résultantes sont telles qu'illustrées à la figure 10.

### Sous-partitionnement

Il est important de signaler que, pour ces stratégies ainsi que pour la suivante (itérative), une étape ultérieure de *sous-partitionnement* doit être appliquée, de façon à assurer que deux noeuds liés par une dépendance *certaine indirecte* ne soient dans la même partition; ces stratégies traitent toutes les dépendances certaines de façon identique, qu'elles soient directes ou non.

#### Stratégie itérative

Finalement, il est possible de combiner les deux stratégies précédentes, en itérant de l'une à l'autre, pour obtenir des résultats encore plus précis, et en travaillant à chaque itération sur la version réduite du graphe.

Une telle stratégie appliquée à notre exemple permettrait d'obtenir les *threads* suivants:

```
f = b1

b1::
  THREAD{}{   }    t3 = IConstr |(,)|;
                   ENVOYER_RES t3
  THREAD{}{a,b}    t1 = a+b
                   IVStore t3, 1, t1
                   t2 = a*b;
                   IVStore t3, 2, t2
```

### 4.2.3 Partitionnement basé sur les contraintes de séparation

La stratégie introduite par [Sch94, SCG95] est intuitivement plus simple que celle basée sur l'approche itérative. De plus, elle est censée produire de meilleurs résultats. Toutefois, le coût est, semble-t-il, supérieur, d'où la suggestion d'utiliser les deux stratégies de façon combinée, celle itérative étant utilisée pour les blocs de base plus gros et sans appels de fonction.

L'idée est simplement de former des *threads* de la façon suivante:

```
Identifier les contraintes de séparation
TANTQUE il reste des noeuds qui n'ont aucune contrainte de séparation FAIRE
  Choisir deux noeuds n'ayant aucune contrainte de séparation
  Combiner les deux noeuds
  Mettre à jour les dépendances et contraintes de séparation
```

Les contraintes de séparation sont comme suit: deux noeuds doivent être séparés si et seulement si il existe une dépendance indirecte (potentielle ou certaine) entre les deux. En d'autres mots, deux noeuds peuvent être mis dans le même *thread* si et seulement si il n'existe aucune dépendance indirecte certaine et aucune dépendance indirecte potentielle (surestimé). Ces contraintes peuvent donc être calculées, *grosso modo*, à partir du complément de la fermeture transitive des dépendances certaines. Pour notre exemple, cette approche produirait le même groupe de *thread* que l'approche itérative.

Notons que, pour des programmes complexes, toute l'astuce dans l'utilisation efficace de cette approche est évidemment de choisir une heuristique permettant de bien choisir les noeuds à combiner. De telles heuristiques sont décrites dans [Sch94].

### 4.2.4 Influence des types

Les approches de partitionnement mentionnée plus haut diffèrent de notre présentation en ce sens qu'elles supposent toutes que les fonctions peuvent recevoir $n$ arguments et retourner $m$ résultats. Ceci est utile, dans leur cadre, dans la mesure où les structures de données ne sont pas véritablement traitées. Ainsi, seuls [Tra88b, Coo94] mentionnent, encore que très brièvement et de façon très limitée, le traitement des structures de données (essentiellement des tuples). Les dépendances potentielles associées au *feedback* de la non-stricticité sont donc toujours obtenue via des scalaires, ce qui n'est pas le cas dans nos exemples, où un seul résultat est retourné par une fonction, lequel peut évidemment être un tuple (ou pourrait être toute autre structure de données). Dans nos exemples, il existe donc des dépendances potentielles qui ne sont pas présentes dans les approches précédentes: celles allant du résultat (type tuple) vers les arguments (type entier). Or, il est clair que si le programme est correctement typé une telle dépendance est impossible/inadmissible, un argument de type tuple ne pouvant pas être transmis en une position nécessitant un entier. Dans notre exemple, ceci nous permettrait donc d'éliiner les deux dépendances potentielles restantes, ce qui permettrait d'obtenir le partitionnement suivant:

```
f = b1

b1::
  THREAD{}{a,b}    t3 = IConstr |(,)|;
                   ENVOYER_RES t3
                   t1 = a+b
                   IVStore t3, 1, t1
                   t2 = a*b;
                   IVStore t3, 2, t2
```

En d'autres mots, pour cet exemple, toute possibilité de *feedback* peut être ignorée par le partitionnement parce que de toute façon, s'il existait un tel *feedback*, alors le programme générerait assurément un *deadlock*.

## 4.3 Partitionnement global et inter-procédural

Alors que le partitionnement local traite chaque bloc de façon isolée, le partitionnement global tente de propager les informations de dépendances d'un bloc vers un autre.

Les effets de cette propagation sont les suivants, selon que l'on propage d'un bloc appelant vers un bloc appelé, ou vice-versa:

- Propagation de l'appelé vers l'appelant: ici, les dépendances certaines identifiées dans le corps d'une procédure sont utilisées pour introduire, au niveau de l'appelant, des dépendances certaines, éliminant ainsi certaines dépendances potentielles.

  Par exemple, soit le code suivant:

  ```
  f x y = if x == 0 then 0 else y
  r = f a b
  ```

  Une partie du graphe associé à l'appel de `f` dans `g` est présentée à la figure 11. Initialement, puisqu'on a aucune information concernant les dépendances possibles associées à l'appel de `f`, on doit supposer qu'il existe une dépendance potentielle possible allant de chacun des *inlets* vers le *outlet*.

  Après analyse de `f`, on peut déterminer qu'il existe, dans `f`, une dépendance directe allant du premier argument vers le résultat produit par `f` (`f` est strict en son premier argument). Cette information peut alors être propagée au contexte

a

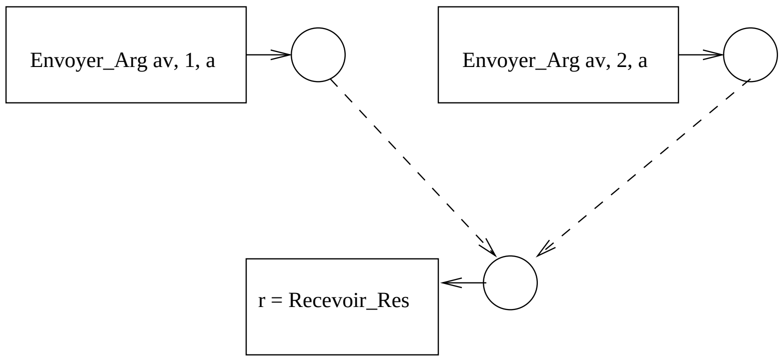


Figure 11: Dépendances potentielles associées à l'appel de `f` avant propagation interprocédurale

appelant et permet d'introduire une dépendance indirecte certaine allant du noeud transmettant le premier argument vers le noeud recevant le résultat, ce qui a pour effet de contredire, donc éliminer, l'une des dépendances potentielles. Ceci est illustré à la figure 12.

- Propagation de l'appelant vers l'appelé: ici, il s'agit de propager l'information associée aux contextes d'appel pour, entre autres, indiquer que certaines dépendances potentielles de *feedback* (d'un résultat vers un argument, tel que permis par la non-stricticité) ne surviennent pas, donc que certaines dépendances potentielles sont certaines de ne pas exister.

  Par exemple, soit la fonction `f` suivante vue précédemment:

  ```
  f u v = (u*u, v+v)
  ```

  Le graphe associé à cette fonction fut présentée à la figure 7. En l'absence de toute information associée aux contextes d'appel, le partitionnement illustré à la figure 13 sera produit: les deux composantes du tuple doivent être produites séparément, puisqu'il est tout à fait possible (cf. exemples vus précédemment) qu'il existe une dépendance de l'une des composantes vers l'autre. Toutefois, si l'on peut déterminer que dans tous les contextes d'appel de `f` il n'y a aucune telle dépendance, alors ces deux *threads* pourront être regroupés en un seul.

### 4.3.1 Approche de Schauser et al.

Dans l'approche de Schauser et al. [TCS92, Sch94, SCG95], l'analyse globale et interprocédurales des dépendances et le partitionnement se font de façon intégrée, via un processus itératif procédant, *grosso modo*, comme suit:

```
Soit b_1 un bloc appelant et b_2 un bloc appelé
Soit b l'un des deux bloc, disons b_2
REPETER
  Partitionner le bloc b
```

a

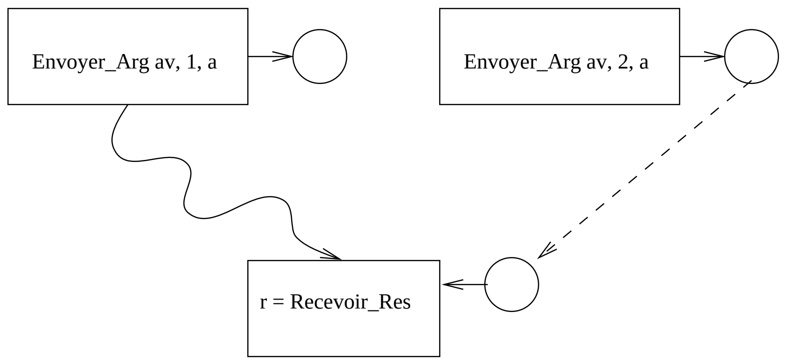


Figure 12: Dépendances potentielles associées à l'appel de `f` après propagation inter-procédurale

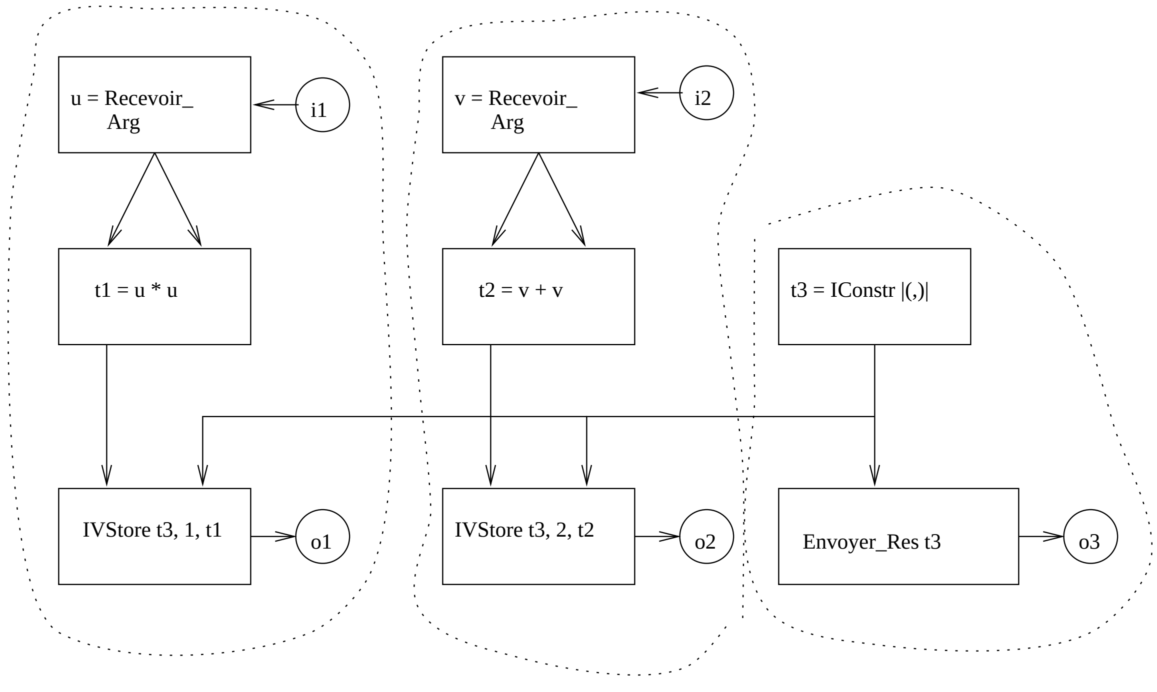


Figure 13: Partitionnement local de `f`

a

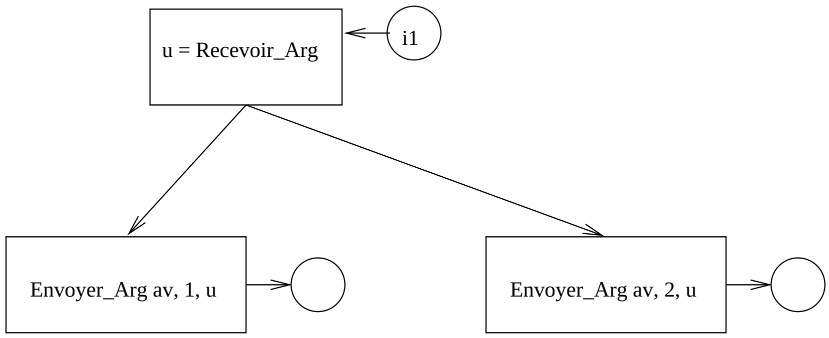


Figure 14: Contexte appelant pour `f`

```
    Propager les informations du bloc b vers l'autre bloc
    b = l'autre bloc
  JUSQUE il n'y ait plus de modifications
```

Dans cette approche, ce sont les annotations des *inlets/outlets* qui sont propagées d'un bloc à l'autre. Dans la figure 13, chaque *inlet/outlet* est annoté de façon unique, indiquant la possibilité de n'importe quelle dépendance potentielle, sauf lorsque contredite par une dépendance certaine. Toutefois, supposons que cette fonction ne soit appelée que dans le contexte suivant, illustré à la figure 14:

```
g u = f u u
```

Dans ce contexte, à cause des dépendances certaines allant du noeud contenant le *inlet* annoté par `i1` vers les noeuds `Envoyer_Arg`, les deux *inlets* de la fonction `f` pourraient alors être annotés par le même identificateur. On obtiendrait alors les partitions illustrés par la figure 15.

### Traitement de la récursion

Cette stratégie ne permet pas le traitement correct de la récursion. Pour traiter les fonctions récursive et/ou mutuellement récursives, une approche différente doit être utilisée. Cette approche, présentée brièvement dans [Sch94], possède les caractéristiques suivantes:

- L'information entre blocs n'est propagée que de l'appelé vers l'appelant. Il semblerait que la propagation de l'un à l'autre, dans le cas des fonctions récursives, ne soit pas valide à cause du fait que l'analyse et le partitionnement soient intégrés. En d'autres mots, l'information utilisée à un certain point peut changer en cours de route dûe à certains regroupements effectués par le partitionnement.

- Un traitement via un calcul de point fixe est utilisé pour analyser et partitionner les fonctions récursives. Une fois cette information obtenue et le partitionnement effectué, elle est propagée vers l'appelant.

- Le traitement des fonctions récursives n'est décrit qu'assez brièvement et, contrairement à l'autre approche, n'est pas décrit et prouvé formellement. Je ne suis

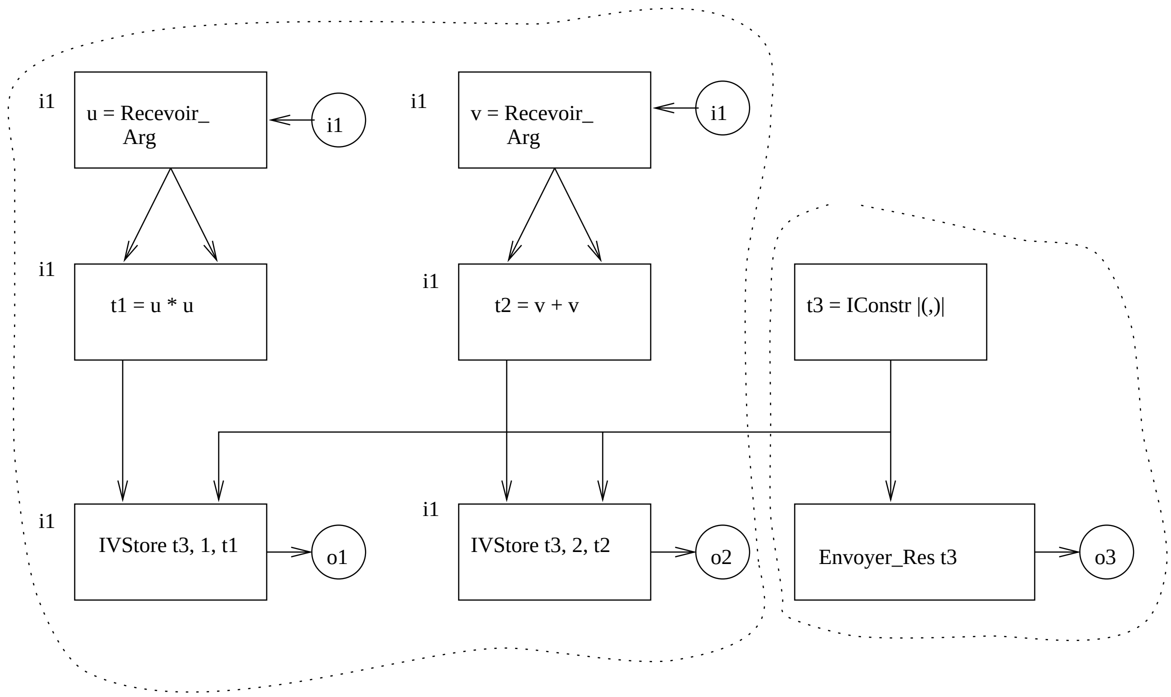


Figure 15: Partitionnement de `f` en tenant compte du contexte d'appel

pas convaincu que les résultats obtenus soient toujours corrects car l'approximation initiale utilisée me semble trop optimiste et semble conduire à trouver le plus grand point fixe plutôt que le plus petit.

## 4.4 L'analyse des chemins de Coorg

Un des problèmes de l'approche précédente mentionné par Coorg [Coo94, p. 36] est le suivant:

> One problem with DD is that is tries to derive dependence information and do partitioning in a single framework. This gives rise to difficulties when dealing with recursion. We separate the two parts: dependence information is computed and represented using *paths* and this information is used in the partitioning algorithm.

### 4.4.1 Partitionnement local

Dans le cas local, cette approche est essentiellement la même que l'approche itérative vue précédemment, bien qu'il utilise une approche de propagation des *ensembles de dominance* pour les *inlets* plutôt que la propagation des ensembles de dépendances.

### 4.4.2 Partitionnement global et interprocédural

**La notion de chemin**

L'analyse de Coorg est inspiré des travaux de Hudak, Bloss et Young [HY86, Blo89]. Essentiellement, il s'agit d'une approche d'interprétation abstraite (de type *forward analysis*) qui permet de déterminer, pour une fonction donnée, les divers ensembles de dépendances envers les arguments d'une fonction, dépendances exprimées via les chemins d'exécution possibles. [3] Par exemple, soit la fonction suivante:

```
f a b c d = if a == 0 then 0 if a == 1 then b else c + d
```

À cette fonction serait associé l'ensemble des chemins suivants: $\{\{a\},\{a,b\},\{a,c,d\}\}$. De par ces chemins, on peut déduire les informations suivantes:

- La fonction `f` est stricte en son premier argument, puisque `a` apparaît dans tous les chemins possibles.
- Bien que `f` ne soit pas stricte en `c` et `d`, ces deux arguments peuvent toujours être évalués dans le même *thread*, puisqu'ils apparaissent toujours ensemble dans tous les chemins.

**Propagation globale**

Dans cette approche, ce ne sont pas directement les ensembles de dépendances, représentés par des annotations de *inlet/outlet* qui sont propagés, mais plutôt des noms de chemin. Lorsque plusieurs chemins sont possibles, le bloc où l'information est propagée est instantié pour tenir compte des informations propagées et les informations résultantes sont ensuite combinées pour n'inclure que ce qui est commun à tous les chemins.

[3] Contrairement aux chemins de [Blo89], ceux de Coorg ne sont pas ordonnés, même s'il utilise une notation qui indique des séquences.

Une des différences, mentionnée par Coorg, entre l'approche de Coorg et celle de Schauser est que la propagation se fait essentiellement de l'appelant vers l'appelé, et ce que les blocs traités soient associés à des fonctions récursives ou non. [4]

### Extensions

Certaines extensions à son approche sont esquissées par Coorg:

- L'utilisation d'information concernant les contextes d'appel, i.e., la propagation de l'appelant vers l'appelé.

  Dans ce cas, comme pour l'approche de Schauser et al., ceci implique, dans le cas des fonctions récursives, que la fonction doit être repartitionnée après chaque itération, de façon à tenir compte du contexte lié aux appels récursifs.

- Une approche pour le traitement des structures de données est brièvement présentée. Toutefois, seuls les produits (tuples) sont traités. Aucun traitement n'est présenté pour les structures de données plus générales (récursives) style listes, arbres, etc.

# 5 Conclusion

### État du compilateur

Les deux principaux morceaux qui restent à développer dans notre compilateur sont l'analyse des dépendances + partitionnement et le générateur de code pour la machine `EARTH`. Le module de *threadification* est en cours de développement. Le développement du générateur de code ne devrait pas créer de difficulté conceptuelle majeure. Le développement de l'analyse de dépendances + partitionnement, par contre, pose de nouveaux problèmes. Divers points reliés à ce problème sont présentés dans les paragraphes qui suivent.

### Quelques éléments pour le partitionnement

- C'est véritablement le traitement des structures de données qui semble être le point faible des diverses approches. Comme l'indique Schauser [SCG95]:

  > *Using separation partitioning we have reduced the control overhead such that a large fraction of the remaining inefficiency is due to accesses to synchronizing I-structures. Better data structure analysis would reduce this overhead further and enhances the ability of interprocedural analysis to derive larger threads.*

- Bien que l'approche, toujours utilisée dans les travaux mentionnés précédemment, de considérer des fonctions avec $n$ arguments et $m$ résultats semble générale, elle ne semble pas vraiment intéressante lorsqu'on tente de généraliser à des structures de données arbitraires. Dans ce cadre, cela semble relativement facile (cf. [Coo94]) de généraliser à des structures de données qui sont des tuples (longueur finie et fixe) mais pas aux autres types de données.

[4] Ceci signifie, semble-t-il, qu'il n'est pas possible de représenter des corrélations entre *outlets*.

- La stratégie utilisée par Coorg de séparer de façon plus claire le calcul d'information interprocédurale du processus de partitionnement lui-même semble plus modulaire, plus *clean*. Pour le traitement des structures de données, qui nécessiteront une analyse de dépendances plus fouillée, ce sera très certainement la voie à suivre.
- Tel que brièvement illustré précédemment, le traitement des structures de données associé à l'utilisation d'informations de type devrait permettra d'éliminer certaines dépendances potentielles.
- Il serait intéressant d'examiner comment le partitionnement local — par exemple, via propagation des dépendances envers les *inlets*/*outlets* — pourrait se faire à l'aide d'une grammaire attribuée. Intuitivement, il semble que cela pourrait se faire relativement facilement. Par contre, il reste encore à voir comment l'analyse interprocédurale des dépendances, principalement pour traiter les structures de données, pourra se faire. Une approche basée sur une interprétation abstraite — donc avec calcul de point fixe — me semble, pour l'instant, la voie à examiner. Quelques possibilités à examiner:
  - Adapter/augmenter l'analyse des chemins de Coorg pour traiter des structures de données autres que les tuples: ceci serait très probablement utile pour la propagation "par en avant".
  - Adapter/augmenter mon approche de propagation abstraites de demandes pour représenter des dépendances plus générales: ceci serait utile pour la propagation d'information "par en arrière".

# Références


[AA89] Z. Ariola and Arvind. P-TAC: A parallel intermediate language. In *Proc. of the Fourth International Conf. on Funct. Prog. Lang. and Comp. Arch.*, pages 230–242. ACM Press, 1989.

[AAS95] S. Aditya, Arvind, and J. Stoy. Semantics of barriers in a non-strict, implicitly-parallel language. In *FPCA '95*, 1995.

[ANP89] Arvind, R.S. Nikhil, and K.K. Pingali. I-Structures: Data structures for parallel computing. *TOPLAS*, 11(4):598–632, October 1989.

[Blo89] A.G. Bloss. *Path Analysis and the Optimization of Non-strict Functional Languages*. PhD thesis, Yale University, May 1989.

[Coo94] S.R. Coorg. Partitioning non-strict languages for multi-threaded code generation. Master's thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1994.

[HMa95] H.J. Hum, O. Maquelin, and al. A design study for the EARTH multiprocessor. In *PACT '95*, pages 27–29. IFIP, 1995.

[HY86] P. Hudak and J. Young. Higher-order strictness analysis for untyped lambda-calculus. In *Proc. of the 13th Annual ACM Symp. on Principles of Prog. Lang.*, pages 97–109, 1986.

[NAH+95] R.S. Nikhil, Arvind, J. Hicks, S. Aditya, L. Augustsson, J.-W. Meassen, and Y. Zhou. pH language reference manual, version 1.0 — preliminary. Technical Report 369, Laboratory for Computer Science, MIT, Jan. 1995.

[Nik93] R.S. Nikhil. A multithreaded implementation of Id using P-RISC graphs. In *Languages and Compilers for Parallel Computing*. Springer-Verlag, LNCS-768, 1993.

[Par94] T.J. Parr. An overview of SORCERER: A simple tree-parser generator. In *Compiler Construction (CC '94)*. Springer-Verlag, LNCS-786, 1994.

[Par96] T.J. Parr. *Language Translation Using PCCTS and C++*. Automata Publishing Company, 1996.

[SCG95] K.E. Schauser, D.E. Culler, and S. Goldstein. Separation constraint partitioning — A new algorithm for partitioning non-strict programs into sequential threads. In *POPL '95*, 1995.

[Sch91] K.E. Schauser. Compiling dataflow into threads. Master's thesis, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, 1991.

[Sch94] K.E. Schauser. *Compiling Lenient Languages for Parallel Asynchronous Execution*. PhD thesis, Computer Science Division, University of California, Berkeley, May 1994.

[SCvE91] K.E. Schauser, D.E. Culler, and T. von Eicken. Compiler-controlled multithreading for lenient parallel languages. In *FPCA '91*, pages 50–72. Springer-Verlag, LNCS-523, 1991.

[TCS92] K.R Traub, D.E. Culler, and K.E. Schauser. Global analysis for partitioning non-strict programs into sequential threads. In *Proc. of the 1992 ACM Conf. on Lisp and Funct. Prog.*, pages 324–334, 1992.

[Tra88a] K.R. Traub. Compilation as partitioning: A new approach to compiling non-strict functional languages. CSG Memo 291, Lab. for Comp. Science, MIT, 1988.

[Tra88b] K.R. Traub. Sequential implementation of lenient programming languages. Technical Report MIT/LCS/TR-417, Laboratory for Computer Science, MIT, 1988.

[Zho94] Y. Zhou. `pH` kernel format. Laboratory for Computer Science, MIT, Nov. 1994.