

# Compilation : de C au langage machine

Roberto M. Amadio

Cours M1 Ingénierie Informatique  
Université Paris Diderot (Paris 7)  
AA 2013-2014

## Sommaire du cours

- Préliminaires.
- Spécification de la sémantique (opérationnelle).
- Spécification d'un compilateur jouet.

# Préliminaires

## Que fait un compilateur ?

- Que fait un **ordinateur** ? Il exécute des **commandes**.
- Que fait un **compilateur** ? Il **traduit** des commandes formulées dans un certain langage  $L$  (dit **langage source**) dans les commandes d'un langage  $L'$  (dit **langage objet**) compréhensible par l'ordinateur.

$$\mathcal{C} : L \rightarrow L'$$

- Avant de traduire, il faut s'assurer que la 'phrase' à traduire soit bien une phrase du langage source. C'est le but de l'**analyse syntaxique**.

$$w \in L ?$$

## Structure d'un compilateur (typiquement de C a un langage machine)

**Front-end** Analyse lexicale et syntaxique. Analyse statique.  
Génération d'un code intermédiaire.

**Langages Intermediaires** Optimisations sur le code  
intermédiaire. Sélection d'instructions.

**Back-end** Allocation de registres. Conventions d'appel.  
Linéarisation du code. Génération de code assembleur.

## Exemple d'un compilateur : gcc

**Front-end** C, C + +, Java, Ada, Fortran,:::

**Langages intermediaires** GIMPLE de type **Register Transfer Language** (RTL). A noter qu'il s'agit de **registres virtuels** et pas des registres du processeur. Nombreuses optimisations effectuées à ce niveau.

**Back-end** Depend du micro-processeur (de l'ordre de 100). Encore des optimisations à ce niveau (allocation de registres,:::)

**NB** Il y a beaucoup d'optimisations possibles et on peut toujours en trouver qui améliorent les précédentes (**full employment theorem for compiler writers**).

## Objectifs d'un compilateur

**Fiabilité** Doit préserver la sémantique du programme source et produire des messages d'erreur significatifs.

**Portabilité** Facile à maintenir et à re-diriger.

**Efficacité** Doit générer du code rapidement.

**Efficacité du code généré** En espace du code généré et en complexité (en temps et en espace) de l'exécutable.

## Exercice

- Consulter le man gcc.
- Utiliser les options pour visualiser le code assembleur produit par gcc en fonction du niveau d'optimisation choisi.
- Par exemple :  
`gcc -O3 -o example -v -save-temps example.c`  
produit (entre autres) un fichier `example.s` avec optimisations de 'niveau 3'.



## Objectifs du cours

### Notions d'intérêt general

- Sémantique opérationnelle.
- Transformations de programmes.
- Analyse du flot de données et du contrôle.
- Gestion de la mémoire.
- Fonctions d'ordre supérieur.

## Un compilateur optimisant pour C

- La **specification formelle** de C.
- Aspects spécifiques de gestion du **contrôle**, de l'**environnement** et de la **memoire**.
- Structuration des **optimisations** et du **back-end**.
- **Dependance** du processeur.

## Un projet de genie logiciel

- Manipulation d'un programme d'une certaine taille (de l'ordre de 10K lignes d'ocaml).
- Modularisation et Intégration (travail en équipe).
- Test et Mesures de performance.
- Questions légales (licences).

## Cours connexes

### Pre-requis (L3)

- Analyse Syntaxique et Compilation.
- Programmation fonctionnelle (ocaml).

### Cours du M1

- Architecture (S1).
- Calculabilité et Complexité (S1).
- Sémantique (S2).
- Programmation fonctionnelle avancée (S1).
- Preuves assistées par ordinateur (S2).

### Cours plus avances (M2)

- Parcours **P**rogrammation (M2-II)
- Parcours **S**emantique et/ou **V**erification (M2-MPRI)

## Emploi du temps

Cours R. Amadio. 2h/semaine.

TP/Projet P. Boutillier. 2h/semaine.

## Plan du cours

Introduction

Front end

$C \rightarrow \text{Clight} \rightarrow \text{Cminor} \rightarrow \text{RTLAbs}$

Langages assembleurs Les exemples de Mips et 8051.

Back end

$\text{RTLAbs} \rightarrow \text{RTL} \rightarrow \text{ERTL} \rightarrow \text{LTL} \rightarrow \text{LIN} \rightarrow \text{Mips}$

Complements

- Gestion de la mémoire.
- Fonctions d'ordre supérieur.

Revision

## Plan des TP/Projet

**TP** Présentation d'un **compilateur experimental** de C vers un langage assembleur (Mips) écrit en ocaml qui implémente une partie des idées présentées dans le cours.

**Projet** Adapter certaines parties du compilateur présenté. Par exemple :

- optimiser la **selection d'instructions**.
- mettre en oeuvre un algorithme pour l'**allocation de registres**.

## References

- Les **transparents du cours** seront disponibles sur la page du cours.
- Le **code commenté du compilateur** utilisé dans le projet sera disponible sur la page du projet.
- Deux **livres** qui couvrent le contenu du cours sont :
  - { A. Appel. **Modern compiler implementation** (in ML).  
Cambridge University Press.  
Le plus proche du cours.
  - { M. Muchnich. **Advanced Compiler Design and Implementation**. Morgan Kaufmann.  
A consulter un peu comme une encyclopedie.



## Contrôle des connaissances

- La **note finale** est déterminée par la note d'examen et la note de projet (qui ont le même poids).
- Le projet est **obligatoire**.
- A la deuxième session on **garde** la note de projet de la première session.
- Veuillez consulter la page du cours pour des amples informations sur ce qu'on considère comme un **plagiat** dans un environnement académique.

.

# Spécification de la sémantique (opérationnelle)

## Outils de specification

Probleme	Outils
Reconnaître le langage source	Grammaires
Spécifier la sémantique <b>dynamique</b>	Sémantiques (opérationnelle, dénotationnelle, axiomatique,...)
Spécifier la sémantique <b>statique</b>	Systèmes de typage, Sémantiques non-standard (int. abstraite...)

**NB** Dans la suite on se focalise sur la **Sémantique opérationnelle**.

## Semantique operationnelle

La **semantique operationnelle** d'un **langage de programmation** est constituée (au moins) d'un **systeme formel** (au sens de la logique mathématique) qui permet de décrire à un certain niveau d'abstraction les **executions possibles** des **programmes du langage**.

**NB** A partir d'une telle spécification, il y a un certain nombre de constructions (**traces, simulations, ::::**) qui permettent de dériver une notion d'**equivalence** ou d'**approximation** entre programmes.

## Exemple (automate a pile)

- Un **automate** (à pile)  $A$  peut être vu comme un programme qui prend un mot  $w$  en entrée et l'accepte ou le refuse.
- Dans ce cas la sémantique opérationnelle revient à **specifier les calculs de  $A$** .
- Typiquement, on décrit les **petits pas** par des règles :

$$(q; a \cdot w; X \cdot \ ) \vdash (q'; w; ' \cdot \ )$$

- On dérive le **resultat du calcul** :

$$w \in \mathcal{L}(A) \text{ ssi } (q_0; w; Z_0) \vdash^* (q; ; \ )$$

**NB** On applique la **même idee** aux langages de programmation.

## Un autre exemple : le langage Imp

$id$	$::= x \mid y \mid \dots$	(identi ants)
$n$	$::= 0 \mid -1 \mid +1 \mid \dots$	(entiers)
$v$	$::= n \mid \text{true} \mid \text{false}$	(valeurs)
$e$	$::= id \mid v \mid e + e$	(expressions)
$b$	$::= e < e$	(conditions booleennes)
$S$	$::= \text{skip} \mid id := e \mid S; S \mid \text{if } b \text{ then } S \text{ else } S \mid \text{while } b \text{ do } S$	(commandes)
$P$	$::= \text{prog } S$	(programmes)

## Syntaxe abstraite

- La **syntaxe abstraite** d'un programme est essentiellement un arbre étiqueté.
- Le fait de passer d'une représentation linéaire (un mot) à une **representation a arbre** permet d'éliminer certaines informations syntaxiques comme les parenthèses.
- Dans des langages à la ML, la représentation d'un arbre étiqueté est directe à l'aide de **declarations de type et de constructeurs**.

```
exp = Id of string | Cnst of int | Op of op * exp * exp
```

## Modele memoire

- Imp est un langage qui agit par **e** et **de bord** sur une mémoire.
- Dans notre exemple jouet, il suffit de voir la mémoire comme une **fonction totale**  $s$  des identificateurs aux entiers.

$$s : id \rightarrow \mathbb{Z}$$

- Notation pour la **mise-a-jour** :

$$(s[n=x])(y) = \begin{cases} n & \text{si } x = y \\ s(y) & \text{autrement} \end{cases}$$



## Evaluation des expressions (grands pas)

$$\begin{array}{c}
 \frac{}{(v, s) \Downarrow v} \qquad \frac{}{(x, s) \Downarrow s(x)} \\
 \\
 \frac{(e, s) \Downarrow v \quad (e', s) \Downarrow v'}{(e + e', s) \Downarrow (v +_{\mathbf{Z}} v')} \qquad \frac{(e, s) \Downarrow v \quad (e', s) \Downarrow v'}{(e < e', s) \Downarrow (v <_{\mathbf{Z}} v')}
 \end{array}$$

**NB** L'addition  $v +_{\mathbf{Z}} v'$  est définie seulement si  $v, v' \in \mathbf{Z}$  (et de même pour  $v <_{\mathbf{Z}} v'$ ).

## Evaluation de commandes et programmes (grands pas)

$$\begin{array}{c}
 \frac{}{(\text{skip}, s) \Downarrow s} \qquad \frac{(e, s) \Downarrow v}{(x := e, s) \Downarrow s[v/x]} \qquad \frac{(S_1, s) \Downarrow s' \quad (S_2, s') \Downarrow s''}{(S_1; S_2, s) \Downarrow s''} \\
 \\
 \frac{(b, s) \Downarrow \text{true} \quad (S, s) \Downarrow s'}{(\text{if } b \text{ then } S \text{ else } S', s) \Downarrow s'} \qquad \frac{(b, s) \Downarrow \text{false} \quad (S', s) \Downarrow s'}{(\text{if } b \text{ then } S \text{ else } S', s) \Downarrow s'} \\
 \\
 \frac{(b, s) \Downarrow \text{false}}{(\text{while } b \text{ do } S, s) \Downarrow s} \qquad \frac{(b, s) \Downarrow \text{true} \quad (S; \text{while } b \text{ do } S, s) \Downarrow s'}{(\text{while } b \text{ do } S, s) \Downarrow s'} \\
 \\
 \frac{(S, s) \Downarrow s'}{(\text{prog } S, s) \Downarrow s'}
 \end{array}$$

## Trois possibilites

On converge a une valeur/resultat

```
x:=1; while (0<x) do x:=x+(-1)
```

On diverge

```
x:=1; while (0<x) do x:=x+1
```

On est bloque (erreur)

```
x:=1; while (0<x) do x:=x>true
```

## Exercice (determinisme, erreurs et divergence)

1. Montrez que pour chaque programme  $P$  :

$$(P; s) \Downarrow s' \text{ et } (P; s) \Downarrow s'' \text{ implique } s' = s''$$

2. Ajoutez des règles de la forme

$$(S; s) \Downarrow \text{err}$$

pour spécifier les situations où une situation d'erreur est détectée pendant l'exécution.

3. Ajoutez des règles de la forme

$$(S; s) \uparrow$$

pour spécifier quand un calcul diverge.

**NB** Il convient de définir  $\uparrow$  comme le **plus grand ensemble de couples**  $(S; s)$  qui satisfait certaines conditions (définition co-inductive).

4. Vos règles devraient avoir la propriété que pour chaque couple  $(P; s)$  **exactement** une des situations suivantes est réalisée :

$$\exists s' \ (P; s) \Downarrow s' \quad (P; s) \Downarrow \text{err} \quad (P; s) \Uparrow$$

ce qui permet de conclure (avec le point 1) que la sémantique en question est **déterministe**.

## Semantique a petits pas

- La sémantique **a grands pas** a l'avantage de se focaliser sur le résultat final (ce qui est observable) et l'inconvenient d'être éloignée d'une mise-en-oeuvre.
- Dans la sémantique **a petits pas** la situation est **duale**.
- Nous considérons une sémantique à petits pas pour les **commandes** (qui repose sur une sémantique à grands pas pour les expressions et les conditions booléennes).
- Une **continuation**  $K$  est une liste de commandes qui termine avec un symbole spécial **halt** (c'est une représentation abstraite de ce qui reste à faire dans le calcul)

$$K ::= \text{halt} \mid S \cdot K$$

## Semantique a petits pas des commandes

$$(x := e, K, s) \rightarrow (\text{skip}, K, s[v/x]) \quad \text{si } (e, s) \Downarrow v$$

$$(S; S', K, s) \rightarrow (S, S' \cdot K, s)$$

$$(\text{if } b \text{ then } S \text{ else } S', K, s) \rightarrow \begin{cases} (S, K, s) & \text{si } (b, s) \Downarrow \text{true} \\ (S', K, s) & \text{si } (b, s) \Downarrow \text{false} \end{cases}$$

$$(\text{while } b \text{ do } S, K, s) \rightarrow \begin{cases} (S, (\text{while } b \text{ do } S) \cdot K, s) & \text{si } (b, s) \Downarrow \text{true} \\ (\text{skip}, K, s) & \text{si } (b, s) \Downarrow \text{false} \end{cases}$$

$$(\text{skip}, S \cdot K, s) \rightarrow (S, K, s)$$



## Toujours trois possibilites

On converge a une valeur/resultat

$$(S; \text{halt}; s) \xrightarrow{*} (\text{skip}; \text{halt}; s')$$

On diverge

$$(S; \text{halt}; s) \rightarrow \dots (S'; K'; s') \rightarrow \dots$$

On est bloque (erreur)

$$(S; \text{halt}; s) \rightarrow \dots (S'; K'; s') \not\rightarrow$$

et ou bien  $S' \neq \text{skip}$  ou bien  $K' \neq \text{halt}$ .

## Exercices

1. Donnez une définition à petits pas de la sémantique des expressions (et des conditions booléennes).
2. Expliquez pourquoi la sémantique à petits pas est déterministe.
3. A partir de la sémantique à petits pas, on peut définir une relation  $\Downarrow'$  par :

$$(S; s) \Downarrow' s' \quad \text{si} \quad (S; \text{halt}; s) \xrightarrow{*} (\text{skip}; \text{halt}; s')$$

Vérifiez dans des exemples que  $\Downarrow$  et  $\Downarrow'$  sont égales.

# Spécification d'un compilateur jouet

## Le langage Vm

- On définit une **machine virtuelle** Vm et son langage de programmation.
- Une machine virtuelle est essentiellement un **type de donnees** avec des structures de données (pile, tas, environnement, :::) et des opérations (les instructions).
- La machine Vm comprend :
  1. Un **code**  $C$  (une liste d'instructions),
  2. Un **compteur ordinal**  $pc$ ,
  3. Une **memoire**  $s$  (comme pour Imp).
  4. Une **pile** d'entiers .

## Instructions de la Vm (semantique informelle)

<code>cnst(n)</code>	pousser $n$ sur la pile
<code>var(x)</code>	pousser la valeur de $x$
<code>setvar(x)</code>	extraire 1 valeur et l'affecter à $x$
<code>add</code>	extraire 2 valeurs de la pile et pousser leur somme
<code>branch(k)</code>	sauter avec decalage $k$
<code>bge(k)</code>	extraire 2 valeurs et sauter si plus grand ou egal avec decalage $k$
<code>halt</code>	arrêt du calcul

## Instructions de la Vm (semantique formelle a petits pas)

Regle	$C[i] =$
$C \vdash (i, \sigma, s) \rightarrow (i + 1, n \cdot \sigma, s)$	<code>cnst</code> ( $n$ )
$C \vdash (i, \sigma, s) \rightarrow (i + 1, s(x) \cdot \sigma, s)$	<code>var</code> ( $x$ )
$C \vdash (i, n \cdot \sigma, s) \rightarrow (i + 1, \sigma, s[n/x])$	<code>setvar</code> ( $x$ )
$C \vdash (i, n \cdot n' \cdot \sigma, s) \rightarrow (i + 1, (n +_{\mathbf{Z}} n') \cdot \sigma, s)$	<code>add</code>
$C \vdash (i, \sigma, s) \rightarrow (i + k + 1, \sigma, s)$	<code>branch</code> ( $k$ )
$C \vdash (i, n \cdot n' \cdot \sigma, s) \rightarrow (i + 1, \sigma, s)$	<code>bge</code> ( $k$ ) et $n <_{\mathbf{Z}} n'$
$C \vdash (i, n \cdot n' \cdot \sigma, s) \rightarrow (i + k + 1, \sigma, s)$	<code>bge</code> ( $k$ ) et $n \geq_{\mathbf{Z}} n'$

On peut dériver une sémantique à grands pas comme dans le cas d'Imp (voir exercice) :

$$(C; s) \Downarrow s' \text{ si } C \vdash (0; ; s) \xrightarrow{*} (i; ; s') \text{ et } C[i] = \text{halt} :$$

## Compilation de Imp a Vm

Soient  $sz(e)$ ,  $sz(b)$ ,  $sz(S)$  le nombre d'instructions que la fonction de compilation  $\mathcal{C}$  associe a  $e$ ,  $b$ , et  $S$ , respectivement. On définit :

$$\begin{aligned}\mathcal{C}(x) &= \text{var}(x) & \mathcal{C}(n) &= \text{cnst}(n) & \mathcal{C}(e + e') &= \mathcal{C}(e) \cdot \mathcal{C}(e') \cdot \text{add} \\ \mathcal{C}(e < e', k) &= \mathcal{C}(e') \cdot \mathcal{C}(e) \cdot \text{bge}(k)\end{aligned}$$

$$\mathcal{C}(x := e) = \mathcal{C}(e) \cdot \text{setvar}(x) \quad \mathcal{C}(S; S') = \mathcal{C}(S) \cdot \mathcal{C}(S')$$

$$\begin{aligned}\mathcal{C}(\text{if } b \text{ then } S \text{ else } S') &= \mathcal{C}(b, k) \cdot \mathcal{C}(S) \cdot (\text{branch}(k')) \cdot \mathcal{C}(S') \\ \text{ou : } k &= sz(S) + 1, \quad k' = sz(S')\end{aligned}$$

$$\begin{aligned}\mathcal{C}(\text{while } b \text{ do } S) &= \mathcal{C}(b, k) \cdot \mathcal{C}(S) \cdot \text{branch}(k') \\ \text{ou : } k &= sz(S) + 1, \quad k' = -(sz(b) + sz(S) + 1)\end{aligned}$$

$$\mathcal{C}(\text{prog } S) = \mathcal{C}(S) \cdot \text{halt}$$

## Quelles propriétés pour $\mathcal{C}$ ?

- Dans ce cours on va travailler avec des langages **déterministes**.
- Ainsi, la **condition principale** est :

$$(P; s) \Downarrow s' \text{ implique } (\mathcal{C}(P); s) \Downarrow s'$$

si avec entrée  $s$  le résultat est  $s'$  alors le code objet avec la même entrée produit le même résultat.

- De plus on pourrait demander que la **divergence est préservée** :

$$(P; s) \Uparrow \text{ implique } (\mathcal{C}(P); s) \Uparrow$$

si avec entrée  $s$  le programme source diverge alors le code objet fait de même.



- Par contre il **n'est pas** raisonnable de demander la **preservation des erreurs** :

$$(P; s) \Downarrow \text{err} \text{ implique } (\mathcal{C}(P); s) \Downarrow \text{err}$$

en effet des optimisations peuvent éliminer des erreurs présentes dans le code source.

- Enfin, la fonction de compilation peut être une **fonction partielle** (pour certains programmes elle ne produit pas de code objet). Par exemple,  $\mathcal{C}(\text{true})$  n'est pas défini. Dans ce cas, on considère quand même que la fonction est correcte.

### Remarque (sur le non-determinisme)

- Même dans un langage séquentiel comme C, la spécification peut être **non-deterministe**. Par exemple, ANSI-C ne spécifie pas l'ordre d'évaluation d'une expression.
- Le non-déterminisme permet de laisser un **degre de liberte a la mise-en-oeuvre**.
- Dans ce cas, la **condition principale** doit être reformulée (Comment ? Il y a plusieurs possibilités:::).

## Verification du compilateur jouet

Voici les **propriétés** qu'on peut démontrer du compilateur jouet pour assurer la **condition principale** :

1. Si  $(e; s) \Downarrow v$  alors  $C \cdot \mathcal{C}(e) \cdot C' \vdash (i; \cdot; s) \xrightarrow{*} (j; v \cdot \cdot; s)$  où  $i = |C|$  et  $j = |C \cdot \mathcal{C}(e)|$ .
2. Si  $(b; s) \Downarrow \text{false}$  alors  $C \cdot \mathcal{C}(b; k) \cdot C' \vdash (i; \cdot; s) \xrightarrow{*} (j + k; \cdot; s)$  où  $i = |C|$  et  $j = |C \cdot \mathcal{C}(b; k)|$ .
3. Si  $(b; s) \Downarrow \text{true}$  alors  $C \cdot \mathcal{C}(b; k) \cdot C' \vdash (i; \cdot; s) \xrightarrow{*} (j; \cdot; s)$  où  $i = |C|$  et  $j = |C \cdot \mathcal{C}(b; k)|$ .
4. Si  $(S; s) \Downarrow s'$  alors  $C \cdot \mathcal{C}(S) \cdot C' \vdash (i; \cdot; s) \xrightarrow{*} (j; \cdot; s')$  où  $i = |C|$  et  $j = |C \cdot \mathcal{C}(S)|$ .

### Exercice (mise en oeuvre)

Mettez en oeuvre les sémantiques des langages  $\text{Imp}$  et  $\text{Vm}$  ainsi que la fonction de compilation  $\mathcal{C}$ .

## Exercices (preuves)

**Preuve** Essayez de montrer les propriétés énoncées.

**Hauteur de pile** Soit  $C = \mathcal{C}(P)$  un code qui vient de la compilation d'un programme. On suppose qu'au début du calcul la hauteur de la pile est 0.

- Donnez un **algorithme** qui pour chaque instruction  $i$  du code  $C$ , détermine la hauteur de la pile au moment de l'exécution de l'instruction  $i$ .
- Éventuellement, essayez de montrer la **correction** de votre algorithme.

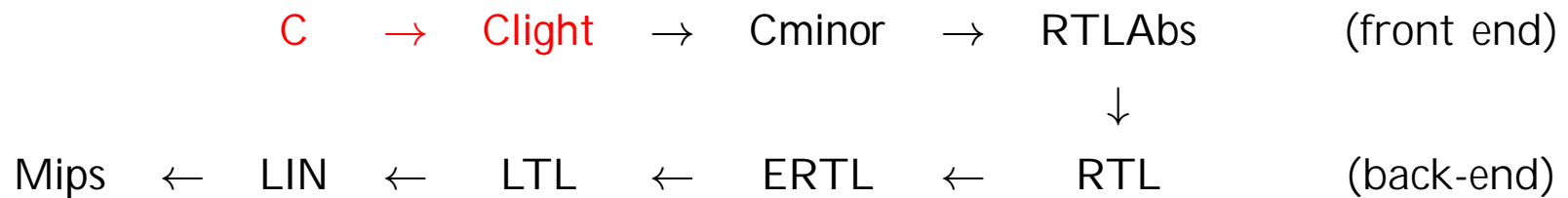
## Passage a l'echelle

- Ils existent maintenant plusieurs exemples de **compilateurs realistes** qui ont été vérifiés dans le sens mathématique qu'on vient de présenter.
- La construction de la preuve utilise un autre programme qu'on appelle **assistant de preuve**.
- L'**assistant de preuve** est un bureaucrate infatigable qui enregistre tous les pas de la preuve et qui vérifie qu'ils sont conformes aux règles d'un certain système logique (par exemple la logique du premier ordre).
- Notez que le **verificateur de la preuve** est un petit programme (quelques centaines de lignes) qui est conceptuellement simple (implémentation directe des règles logiques).

- Même si l'on a pas l'intention de faire une preuve formelle, il est intéressant d'utiliser les **outils de specification** présentés (sémantique opérationnelle).
- Ils permettent de **clarifier** ce qu'on doit implémenter.
- Ils permettent de **tester** le compilateur. En effet les spécifications sont **executables** et peuvent être implémentées presque sans effort dans un langage comme **ocaml**.

**NB** Dans le Projet chaque langage aura sa sémantique exécutable.

## Front end : de C à Clight





## Plan du cours

- Historique.
- CIL.
- Clight : syntaxe.
- Structures de données sémantiques.
- Clight : une idée de la sémantique.

# Historique

## Le compilateur CerCo

- Ce compilateur est développé dans le cadre d'un projet de recherche et hérite de ce projet son nom : CerCo.
- La structure de CerCo est proche de celle d'un autre compilateur développé dans le cadre du projet CompCert. Voir <http://compcert.inria.fr> . A propos de la licence :

The CompCert C compiler is not free software. This release can be used for evaluation, research and education purposes, but not for commercial purposes. See the INRIA Non-Commercial License Agreement for more information.

- A noter qu'une grande partie (de ClightT7 Td [(http:/3(pro)-5ateur)]TJ/F20

- Dans CerCo, la traduction de C à Clight utilise le front-end CIL (C intermediate language). Voir <http://sourceforge.net/projects/cil/>. CIL utilise la GNU General Public License (GNU-GPL). En particulier :

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice...
2. Redistributions in binary form must reproduce the above copyright notice...
- ...

- La traduction de Clight à RTL est en grande partie une simple transcription en `ocaml` des définitions Coq du compilateur CompCert (distribuable sous licence GNU-GPL).
- Le back-end réutilise le back-end d'un autre compilateur (de Pascal à Mips) inspiré par CompCert qui a été écrit par F. POTTIER à des fins pédagogiques (distribuable sous licence Creative Commons (Attribution Non-Commercial Share Alike) ; comme pour la licence INRIA, elle empêche une utilisation commerciale).

CIL

## CIL

CIL est un outil développé en ocaml qui comprend entre autres :

- Un **analyseur syntaxique** pour C.
- Un programme d'environ 5K lignes qui met les programmes C dans une **forme epuree** (qui est encore du C).

Cette forme épurée a servi de base pour la définition du langage Clight qui est le point de départ du projet.

CIL a été développé dans le cadre de la recherche sur l'**analyse de programmes C**.

CIL est **robuste** (e.g. il compile le noyau de LINUX) et il a été adopté dans d'autres projets de R&D (e.g. FRAMA-C).

## Transformations CIL

Voici certaines des transformations opérées par CIL. Voir <http://www.cs.berkeley.edu/~necula/cil/cil004.html>.

### Normalisation declarations de type

```
struct { int x; } s;
```

Ici *S* est une variable dont le type est une **struct** anonyme.

```
struct __anonstruct_s_1 { int x ; };  
struct __anonstruct_s_1 s ;
```

On donne un nom à la structure.



## Calcul longueur des tableaux

```
int a1[] = {1,2,3};  
int a2[sizeof(int) >= 4 ? 8 : 16];
```

`a1` a trois éléments et on sait que `int` prend 4 octets.

```
int a1[3] = {1,2,3};  
int a2[8] ;
```

## Elevation declarations imbriquées

```
int x = 5;
  int main() {
    int x = 6;
    { int x = 7;
      return x; }
    return x; }
```

Toutes les variables sont déclarées à l'entrée de la fonction.

```
int x = 5;
int main(void)
{ int x___0 ;
  int x___1 ;
  { x___0 = 6;
    x___1 = 7;
    return (x___1);
    return (x___0); } }
```

## Elimination d'edges de bord dans les expressions

```
int f (int x){return x+2;}
```

```
int main(){int x; x=3; return f(++x); }
```

On élimine les affectations et les appels de fonction.

```
int f(int x){return x + 2;}
```

```
int main(void){int x; int t; int t__1;  
  x = (int)3; t = x + 1; x = t; t__1 = f(t);  
  return t__1; }
```

**Traitement explicite de cast** On associe un type à chaque expression et des `cast` explicites sont insérés chaque fois qu'il faut promouvoir ou convertir d'un type à un autre.

Clight : syntaxe

## Trois niveaux de description possibles

**Papier**    Mechanised semantics for the Clight subset of the C language. S. Blazy, X. Leroy. *Journal of Automated Reasoning*.

**Assistant de preuve (Coq)**    Voir <http://compcert.inria.fr>.

**Programme (ocaml)**    Voir sources projet : `/src/clight`.

- Dans le cours on utilisera plutôt la **description programme** qui est celle que vous allez manipuler dans les TP et le projet. On n'utilisera jamais la deuxième, et la première de temps en temps (voir premier cours par exemple).
- Pour Clight la description papier est à grands pas et celle programme à petits pas. Au niveau de l'assistant de preuve on considère les deux descriptions et on démontre leur équivalence.

## Conventions

- La **syntaxe abstraite** est décrite dans des fichiers `langage.mli`. Il s'agit essentiellement de **declarations de type**. Un **commentaire** (`* ... *`) peut suggérer la fonction de l'opérateur. Parfois on mentionne la **syntaxe concrete** `[...]`. Par exemple :

```
type binary_operation =  
  ...  
  | Oand (* bitwise and [&]      *)  
  ...
```

- La **semantique** est décrite dans des fichiers `langageInterpret.ml`. Il s'agit essentiellement de la **définition par itrage** sur la syntaxe abstraite d'une fonction d'évaluation.
- Souvent, la définition de la sémantique nécessite l'introduction de **structures auxiliaires continuations, environnements,**

memoires,  $:::$  (cf. sémantique de `Imp`).



## Avertissement

- Accrochez-vous, le plus dur est au début !

La syntaxe et la sémantique des langages qui vont suivre dans la chaîne de compilation seront de plus en plus simples.

- De plus on pourra re-utiliser certaines structures auxiliaires définies pour Clight (par exemple le modèle mémoire).

## Syntaxe abstraite de Clight : Types

Voir `src/clight/clight.mli`.

```
type signedness = Signed | Unsigned
```

```
type intsize = I8 | I16 | I32
```

```
type ctype =
```

Tvoid	(* empty type *)
Tint of intsize*signedness	(* integers *)
Tpointer of ctype	(* pointers *)
Tarray of ctype*int	(* array *)
Tfunction of ctype list*ctype	(* function *)
Tstruct of ident*(ident*ctype) list	(* struct *)
Tunion of ident*(ident*ctype) list	(* union *)
Tcomp_ptr of ident	(* pointer to named struct/union *)

A noter que `int` (ou *big\_int* si nécessaire) est un entier ocaml et `ident` est représenté par le type `string` ocaml.

## Exemple

```
Tstruct( 's1', [( 'n', Tint(I32, Signed)); ( 'next', Tcomp_ptr( 's1' )) ] )
```

Le type d'une structure `s1` avec un champ `n` de type entier, 32 bits signé et `next` de type pointeur à `s1`.

**NB** Le type obtenu en remplaçant *comp\_pointer*(`s1`) par `s1` n'est pas légal.

## Syntaxe abstraite de Clight : Expressions

Voir `src/clight/clight.mli`.

```
type unary_operation =  
  | Onotbool (* boolean negation [!] *)  
  ...  
type binary_operation =  
  | Oadd (* addition [+] *)  
  ...  
  | Oand (* bitwise and [&] *)  
  ...  
  | Oeq (* comparison [==] *)  
  ...
```



## Syntaxe abstraite de Clight : Commandes

Voir `src/clight/clight.mli`.

```

type statement =
  | Sskip                                (* skip *)
  | Sassign of expr*expr                 (* assignment [lvalue = rvalue] *)
  | Scall of expr option*expr*expr list  (* function call *)
  | Ssequence of statement*statement     (* sequence *)
  | Sifthenelse of expr*statement*statement (* conditional *)
  |

```

## Syntaxe abstraite de Clight : Fonctions et Programmes

Voir `src/clight/clight.mli`.

```
type cfunction = {  
  fn_return :   ctype ;  
  fn_params  : (ident*ctype) list ;  
  fn_vars    : (ident * ctype) list ;  
  fn_body    : statement }
```

```
type fundef =  
  | Internal of cfunction  
  | External of ident*ctype list*ctype           (* not implemented *)
```

```
type init_data =  
  | Init_int8 of int  
  | Init_int16 of int  
  | ...  
  | Init_space of int  
  | Init_addr of ident*int (* address of symbol + offset *)  
  
type program = {  
  prog_funct: (ident * fundef) list ;  
  prog_main: ident option;  
  prog_vars: ((ident * init_data list) * ctype) list }
```



# Structures de données sémantiques

## Modele memoire

$[X \mapsto Y]$  est l'ensemble des fonctions (partielles) de  $X$  à  $Y$ .

**References** Un ensemble infini  $R$  (par exemple  $\mathbf{Z}$ ).

**Locations** Une location est une référence et un **o set** :

$$L = R \times \mathbf{N}$$

**Valeurs** Une union disjointe d'entiers, pointeurs,  $:::$

$$V = [\text{int} : \mathbf{Z}; \text{ptr} : L; :::]$$

**Memoire** Une fonction qui associe aux références deux entiers (les bornes du bloc de mémoire) et une fonction qui spécifie le contenu du bloc.

$$M = [R \mapsto (\mathbf{Z} \times \mathbf{Z} \times [\mathbf{N} \mapsto V])]$$

## Operations sur la memoire On peut :

- **Allouer** un bloc de mémoire à une **nouvelle** référence.
- **Liberer** (rendre inaccessible) un bloc.
- **Lire** un certain nombre d'octets consécutifs dans un bloc.
- **Modifier** un certain nombre d'octets consécutifs dans un bloc.

**NB** Ce modèle permet une arithmétique de pointeurs à l'intérieur d'un bloc mais il empêche tout débordement à l'extérieur d'un bloc. Dans ce dernier cas, la sémantique du langage source produit une erreur et le code compilé est libre soit de produire une erreur soit de générer un comportement imprévisible.

## Environnements

**Environnement global** Un couple de fonctions : la première associe des références aux variables globales et la deuxième associe les définitions aux noms de fonctions  $d$  :

$$G = [id \mapsto R] \times [d \mapsto Fd]$$

**Environnement local** Une fonction qui associe des références aux variables locales :

$$E = [id \mapsto R]$$

Operations sur les environnements On peut :

- Lire le **bloc** qui correspond à un identificateur.
- Lire la **définition de fonction** qui correspond à un identificateur de fonction.
- Construire l'**environnement initiale** à partir d'un programme.

Clight : sémantique

## Evaluation d'expressions

```
let rec eval_expr globalenv localenv m e = let Expr (ee,tt) = e in  
  match ee with  
  ...
```

- L'évaluation (grands pas) d'une expression *e* (avec son type) dépend d'un env. global *globalenv*, d'un env. local *localenv* et d'une mémoire : *m*.
- Les règles d'évaluations sont différentes selon que l'expression est à gauche (*lvalue*) ou à droite d'une affectation. Par exemple, dans *x=x+1* l'évaluation de *x* à gauche doit rendre une référence et celle de *x* à droite un entier.

## Modes d'execution de commandes

Les règles d'évaluation (petits pas) pour les commandes distinguent trois modes :

```
type state =  
  | State of cfunction*statement*continuation*localEnv*memory  
  | Callstate of fundef*Value.t list*continuation*memory  
  | Returnstate of Value.t*continuation*memory
```



## Continuations

Les continuations sont un peu plus compliquées que dans le langage Imp. En particulier, elles permettent de traiter les appels récursifs.

```
type continuation =  
  | Kstop  
  | Kseq of statement*continuation  
  | Kwhile of expr*statement*continuation  
  | Kdowhile of expr*statement*continuation  
  | Kfor2 of expr*statement*statement*continuation  
  | Kfor3 of expr*statement*statement*continuation  
  | Kswitch of continuation  
  | Kcall of (Value.t*ctype) option*cfunction*localEnv*continuation
```

## Regles a petits pas pour les commandes

```
let next_step globalenv = function
  | State(f,Sassign(a1,a2),k,e,m) ->
    ...
```

Pour calculer le prochain état d'une commande comme `Sassign(a1,a2)` on a besoin de :

- un env. global `globalenv`,
- le nom de la fonction en exécution `f`,
- la continuation `k`,
- l'environnement local `e`,
- la mémoire `m`.

## Sommaire

- Clight est un sous-ensemble de C. CIL s'occupe de l'analyse syntaxique et de compiler un programme C en Clight.
- Pour décrire la sémantique de Clight on a besoin d'un certain nombre de structures : une mémoire, un environnement global et local et une continuation.
- Les détails de la fonction d'évaluation se trouvent dans le fichier `/src/clight/clightInterpret.ml` qui fait environ 500 lignes de code ocaml.

## Front end : de Clight à Cminor



## Plan du cours

- Introduction à Cminor.
- Vers la sémantique de Cminor (exercices).
- Vers la compilation de Clight à Cminor (exercices).

## Différences principales entre Clight et Cminor (et tâches du compilateurs)

**Elimination de la surcharge** Par exemple, on distingue entre l'addition pour les entiers 32 bits et les flottants.

**Calcul d'adresse explicite** Par exemple, pour un tableau d'entiers 32 bits :

`a[i]` devient `load(int32; a + (i * 4))`

**Simplification des structures de contrôle** Cminor en a seulement 4 (plus `switch` et `goto`) :

`if_then_else; block; loop; exit n`

**Gestion explicite de la memoire** Chaque fonction alloue un bloc de mémoire d'une certaine taille où elle place les variables **non-scalaires** et les variables scalaires dont on prend l'adresse (opération &).

**Partitionnement des donnees locales** Les données locales d'une fonction sont :

- soit dans un environnement local qui est initialisé à l'appel de la fonction et éliminé à son retour.
- soit dans un bloc de mémoire qui est alloué à l'appel de la fonction et récupéré à son retour.

**Statut d'une variable** Une variable peut donc être :

- Globale.
- Dans l'environnement local.
- Dans le bloc local de mémoire.

## Syntaxe abstraite de Cminor (extrait)

```
type memory_q =  
  | MQ_int8signed | MQ_int8unsigned | MQ_int16signed | MQ_int16unsigned | ...  
  
type expression =  
  ...  
  | Mem of Memory.memory_q * expression (* memory read *)  
  
type statement =  
  ...  
  | St_store of Memory.memory_q * expression * expression (* memory write *)  
  ...  
  | St_tailcall of expression * expression list * AST.signature (* tail call *)  
  ...  
  | St_loop of statement (* loop... *)  
  | St_block of statement  
  | St_exit of int  
  ...
```



```
type internal_function =  
  { f_sig      : AST.signature ;  
    f_params   : AST.ident list ;  
    f_vars     : AST.ident list ;  
    f_stacksize : int ;           (* size memory allocation *)  
    f_body     : statement }
```

La syntaxe abstraite est encore proche de celle de C mais avec une gestion plus explicite de la mémoire.

## Exercice (continue et break)

On étend les commandes du langage Imp avec les commandes suivantes :

$$S ::= \dots \mid \text{continue} \mid \text{break}$$

D'après **C : a reference manual**, par S. HARBISON et G. STEELE, l'effet de ces commandes est le suivant :

**break** Causes execution of the smallest enclosing while (for, do, switch) statement to be terminated. Program control is immediately transferred to the point just beyond the terminated statement. It is an error for a break statement to appear where there is no enclosing iterative (or switch) statement.

**continue** Causes execution of the smallest enclosing while (for,do) statement to be terminated. Program control is immediately transferred to the end of the body, and the execution of the affected iterative statement continues from that point with a reevaluation of the loop test. It is an error for continue to appear where there is no enclosing iterative statement.

Le **probleme** est de définir une sémantique à grands pas de ces commandes.

**Suggestion** : Utiliser des jugements pour les commandes de la forme :

$$(S; s) \Downarrow (o; s')$$

où  $o \in \{nrm; brk; cnt\}$  indique le **mode du resultat** (*normal*, *break* ou *continue*).

## Exercice (loop, block et exit)

La sémantique informelle des instructions de contrôle de Cminor est la suivante :

$\text{loop}\{S\}$  Boucle infinie.

$\text{block}\{S\}$  Déclaration d'un bloc.

$\text{exit } n$  Termine les  $n + 1$  blocs qui contiennent la commande.

Le **probleme** est d'étendre la sémantique à petits pas de Imp.

**Suggestion** : Étendez la notion de continuation de la façon suivante :

$$K ::= \text{halt} \mid \text{endblock}(K) \mid S \cdot K :$$

Les jugements sont toujours de la forme :

$$(S; K; s) \rightarrow (S'; K'; s')$$

## Exercice (petits pas, grands pas)

Proposez une sémantique :

- **petits pas** pour break et continue.

**Suggestion** : Introduisez une continuation  $\text{endloop}(K)$  similaire à  $\text{endblock}(K)$ .

- **grands pas** pour loop, block et exit.

**Suggestion** : On introduit un jugement  $(S; s) \Downarrow (n; s')$  où  $n = 0; 1; 2; \dots$ . La terminaison ‘normale’ est 0.

# Compilation

## Exercice (fonctions et continuations)

- Dans le langage Imp, une **memoire** est une fonction totale  $s : id \rightarrow \mathbf{Z}$ .
- On appelle **memoire locale** une fonction partielle  $ls \in [id \mapsto \mathbf{Z}]$ .
- On écrit  $ls \cdot s$  pour la mémoire définie par :

$$(ls \cdot s)(x) = \begin{cases} ls(x) & \text{si } x \in dom(ls) \\ s(x) & \text{autrement} \end{cases}$$

- On considère une extension de **Imp** avec une notion rudimentaire de fonction. Un programme est maintenant une suite de **declarations de fonctions** de la forme :

$$\begin{aligned} f_1(x) &= S_1 \\ \dots \\ f_m(x) &= S_m \end{aligned}$$

où  $f_1; \dots; f_m$  sont des noms distincts différents des identificateurs, suivis par un paramètre formel et une commande (le corps de la fonction).

- La catégorie syntaxique des **commandes** devient :

$$S ::= \dots \parallel id := f(e) \parallel \text{return}(e)$$

- et celles des **continuations** devient :

$$K ::= \dots \text{returnto}(f; id; K; ls) :$$



- Proposez une **semantique a grands pas des expressions** dont le jugement a la forme :

$$(e; l; s) \Downarrow v$$

- Proposez une **semantique a petits pas des commandes** dont le jugement a la forme :

$$(f; S; K; l; s) \rightarrow (f'; S'; K'; l'; s')$$

## Remarques

La formalisation en Cminor est un peu plus compliquée :

- L'environnement local devient un **environnement local** et un **bloc de memoire**.

- On distingue les fonctions **internes** et **externes** (que nous ignorons).

- On distingue trois états **standard**, **call**, **return** avec transitions :

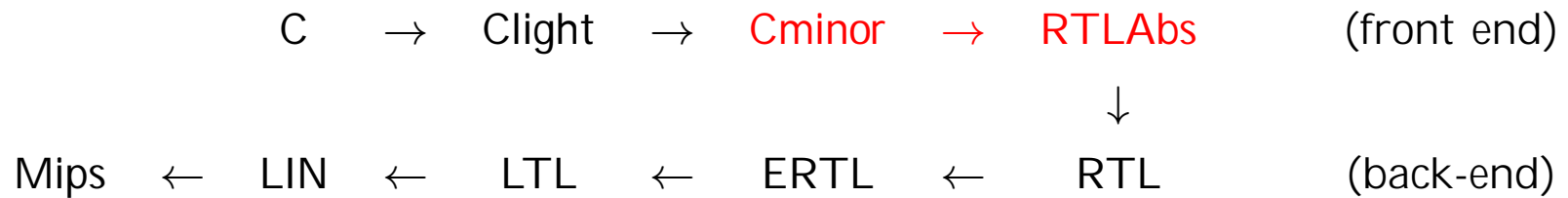
**appel** *standard*  $\rightarrow$  *call*  $\rightarrow$  *standard*.

**retour** *standard*  $\rightarrow$  *call*  $\rightarrow$  *standard*.

- On considère une forme spécialisée d'**appel terminal** (**tail call**). Dans ce cas, le **bloc d'activation** de la fonction appelée

écrase celui de la fonction appelante.

# Front end : de Cminor à RTLAbs



## Plan du cours

- RTLAbs : Syntaxe.
- Compilation de Cminor à RTLAbs (exercices).

## Différences principales entre Cminor et RTLAbs (et tâches du compilateur)

**Pseudo-registres** En RTLAbs on dispose d'une quantité arbitraire de **pseudo-registres**. On peut voir ces pseudo-registres comme des variables locales temporaires.

**Operations simples** En RTLAbs on n'a plus d'expressions. Les arguments des commandes sont des pseudo-registres.

## **Explicitation du Graphe du Flot de Contrôle En RTLabs**

chaque instruction a une adresse (symbolique) et on gère explicitement le passage d'une instruction à une autre.

Ainsi on peut associer à chaque fonction Cminor un **graphe du flot de contrôle** (ou CFG pour **control flow graph**).

## Syntaxe abstraite RTLAbs (extrait)

```
type addressing =
```

```
  | Aindexed of AST.immediate          (* r1 + offset          *)
  | Aindexed2          (* r1 + r2          *)
  | Aglobal of AST.ident * AST.immediate (* symbol + offset      *)
  | Abased of AST.ident * AST.immediate  (* symbol + offset + r1  *)
  | Ainstack of AST.immediate            (* stack_pointer + offset *)
```

```
type statement = ...
```

```
  | St_op of AST.op * Register.t * Register.t list * Label.t      (* operation *)
  | St_load of Memory.memory_q * addressing * Register.t list *
    Register.t * Label.t      (* memory load
  | St_store of Memory.memory_q * addressing * Register.t list *
    Register.t * Label.t      (* memory store
  | St_condition of AST.op * Register.t list * Label.t * Label.t (* branching *)
  | St_call_id of AST.ident * Register.t list * Register.t *
    AST.signature * Label.t (* function call
  | St_return of Register.t option      (* return *)
```



```
type internal_function =                                (* internal function *)
  { f_luniverse : Label.Gen.universe ;                 (* to generate labels *)
    f_runiverse  : Register.universe ;                 (* to generate pseudo-registe
    f_sig        : AST.signature ;
    f_result     : Register.t option ;
    f_params     : Register.t list ;
    f_locals     : Register.t list ;
    f_stacksize  : int ;
    f_code       : code ;
    f_entry      : Label.t }

type program =                                          (* program *)
  { vars      : (AST.ident * int (* size *)) list ;
    functs    : (AST.ident * function_def) list ;
    main      : AST.ident option }
```

## Exercice (introduction des temporaires)

On reprend le langage Imp étendu avec :

- Les instructions de **contrôle** de Cminor : block, loop, exit  $n$
- Une notion rudimentaire de **declaration de fonction** :

function  $f(y) = \text{local } z; S$

- Les instructions pour **l'appel et le retour** d'une fonction :

$S ::= \dots \parallel id := f(e) \parallel \text{return}(e)$

**NB**  $y, z, e, \dots$  sont des vecteurs.

- On suppose qu'un **programme** a la forme :

global  $x$ ;

function  $f_1(y_1) = \text{local } z_1; S_1;$

...

function  $f_n(y_n) = \text{local } z_n; S_n;$

$f_i()$

- On souhaite **compiler** de tels programmes dans des programmes (équivalents) qui satisfont les **restrictions syntaxiques** suivantes :

Forme standard	Forme restreinte
$e ::= n \mid id \mid (e + e)$	$e ::= id$
$b ::= (e < e)$	$b ::= (e < e)$
$S ::= id := e \mid \text{if } b \dots$	$S ::= id := e \mid id := n \mid id := (e + e) \mid \text{if } b \dots$

– **Suggestion :**

1. On suppose disposer d'une fonction *new* qui nous fournit un 'nouveau' identificateur.
2. On définit une fonction :

$$t : Expression \rightarrow Commande \times Identi\ cateur$$

telle que si  $t(e) = (S; x)$  alors après exécution de la commande  $S$  l'identificateur  $x$  contient la valeur de l'expression  $e$  et la commande  $S$  satisfait les restrictions.

3. On définit une fonction :

$$t : Commande \rightarrow Commande$$

telle que  $t(S)$  est une commande équivalente à  $S$  qui satisfait les restrictions.

4. Quid de la compilation des fonctions et du programme ?

## Un langage a la RTLAbs

- Un **programme** a toujours la forme :

global  $x$ ;

function  $f_1(y_1) = \text{local } z_1; G_1;$

...

function  $f_n(y_n) = \text{local } z_n; G_n;$

$f_i()$

- Par contre le code associé à chaque fonction est maintenant structuré comme un **graphe** (avec racine).
- Chaque noeud du graphe est identifié par une **etiquette**.

- A chaque étiquette on associe une **operation elementaire** et un nombre fini d'étiquettes où le calcul peut continuer (les arêtes du graphe) :

$$\begin{array}{lll} \ell : & \text{skip} & \rightarrow \ell' \\ \ell : & id := n & \rightarrow \ell' \\ \ell : & id := id & \rightarrow \ell' \\ \ell : & id := id + id & \rightarrow \ell' \\ \ell : & id < id & \rightarrow \ell', \ell'' \\ \ell : & \text{branch} & \rightarrow \ell' \\ \ell : & id := f(\vec{id}) & \rightarrow \ell' \\ \ell : & \text{return}(id) & - \end{array}$$

**NB** Le code d'un programme  $V_m$  est un CFG qui en plus a été **linearise** (ordre total entre les noeuds).



## Exercice (explicitation du flot de contrôle)

Décrivez une fonction de compilation du langage impératif avec temporaires au langage à la RTLAbs.

### Suggestions :

- On suppose disposer d'une fonction *new* qui nous donne une nouvelle étiquette  $\ell \in \text{Label}$ .
- On définit une fonction :

$$\mathcal{C} : \text{Commande} \times \text{Label} \rightarrow \text{Graphe} \times \text{Label}$$

avec l'intuition que si  $\mathcal{C}(S; \ell) = (G; \ell')$  alors le CFG  $G$  qui correspond à  $S$  a comme point d'entrée  $\ell'$  et comme point de sortie  $\ell$ .

- Pour le traitement des instructions **block** et **exit** il peut être utile de raffiner en

$$\mathcal{C} : \textit{Commande} \times \textit{Label} \times \textit{Label}^* \rightarrow \textit{Graphe} \times \textit{Label}$$

avec l'intuition que la liste d'étiquettes qu'on prend en argument permet des sorties directes avec l'exit.

- Si  $G$  et  $G'$  sont deux CFG avec ensembles de noeuds disjoints alors on dénotera par  $G \cup G'$  leur union.

## Sommaire

**Fait (front-end)** Chaque fonction `C` a été réduite à un CFG où chaque noeud effectue une opération sur des pseudo-registres.

**A faire (back-end)** Il faut viser un **processeur spécifique**. Les tâches principales seront :

- Sélection d'instructions.
- Explicitation des conventions d'appel (gestion de la pile).
- Allocation des pseudo-registres.
- Linéarisation du code.

Ces tâches dépendent du processeur. On va donc présenter d'abord les deux processeurs considérés dans ce cours.

# Langages Assembleurs

## Plan du cours

- Généralités.
- Mips.
- 8051.

## Generalites

- Un **langage assembleur** est un langage de programmation directement exécutable par un processeur (après conversion en binaire) sous une forme lisible par un humain (en caractères ASCII).
- On appelle aussi **assembleur** le programme qui transforme un programme assembleur en binaire et **desassembleur** celui qui fait l'opération inverse.
- Un langage assembleur peut contenir des **etiquettes** qui seront traduites en offsets numériques et des **macro-instructions** qui seront expansées.

**NB** La réalisation en matériel (hardware) d'un interprète pour le langage binaire est le sujet du **cours d'architecture**.

## Quelques criteres de classification (des processeurs et des langages assembleurs)

- Taille du **mot memoire** : 8, 16, 32, 64, :::
- Organisation de la **memoire** (registres, cache, ROM, RAM,:::)
- Séparation **code** et **donnees**.
- Instructions simples ou complexe (**RISC** (reduced instruction set computer ou **CISC** (complex instruction set computer)).
- **Modes d'adressage** (absolu, relatif, direct, immédiat, indirect à registre,:::)
- **Domaine d'application** générale ou spécifique.

## Mips (historique)

- Architecture **RISC 32** (et 64) bits développées par MIPS Tech. au début des années 80.
- L'objectif était d'exécuter en **pipeline** les instructions. Par exemple l'exécution d'une instruction d'addition pourrait se décomposer en :

Chargement Instruction    Decodage    Lecture    Calcul,    Ecriture

Chaque phase prend un **cycle d'horloge**. A régime on exécute une instruction par cycle d'horloge (les sauts compliquent les choses:::).

- Dans ce but, il convient d'avoir (autant que possibles) des **instructions simples et uniformes**.



- Compétitif dans le marché des ordinateurs de bureau (desktop computers) jusqu'aux années 90, Mips est encore utilisé dans les **systemes embarques**.
- Toujours très populaire dans les **cours d'architecture**.

## Mips (references)

**Tutorial Assemblers, linkers, and the SPIM simulator**, par J. Larus (copie sur la page du cours).

### Emulateur

- L'émulateur (X)SPIM est disponible sur <http://sourceforge.net>.
- Un autre émulateur est MARS (MIPS Assembler and Run-time Simulator).
- Un émulateur/évaluateur écrit en ocaml sera disponible dans `/src/ASM/MIPSInterpret.ml`.

**Compilateur** gcc peut générer du code Mips.

## 8051 (historique)

- Architecture **8 bits** développée par Intel dans les années 80.
- **Nombreuses variantes** ont été développées par une vingtaine d'autres constructeurs avec des noms comme MCS-51, 80C51, 8052.
- Utilisation massive dans le domaine des **systemes embarques** pour des applications qui nécessitent d'une quantité limitée de mémoire pour les données et le code.
- A titre de comparaison, pour trouver un **PC d'une puissance comparable** il faut remonter au Commodore 64 (**1982**). Un PC avec 64K de RAM qui était vendu avec un interprète pour BASIC et qui utilisait la télé (cathodique!) comme écran.

- Le 8051 est maintenant daté mais il y a toujours une demande pour des processeurs économiques et prédictibles. Par exemple voir la série **ARM Cortex-M**.

## 8051 (references)

**Tutorial** Disponible sur la page du cours (on peut ignorer les chapitres sur les **timers** et les **interrupts**).

### Emulateur

- Un émulateur disponible est le MCU 8051 IDE <http://sourceforge.net>.

**NB** Il a beaucoup de fonctionnalités mais il n'est **pas documenté** et il n'est **pas très** **able**.

- Un émulateur/évaluateur en **ocaml** sera disponible.

**Compilateurs** Il y a plusieurs compilateurs de C à 8051 :

**SDCC** Voir : <http://sdcc.sourceforge.net/>

**Commerciaux** Voir par exemple :

<http://www.keil.com/c51/c51.asp> ou

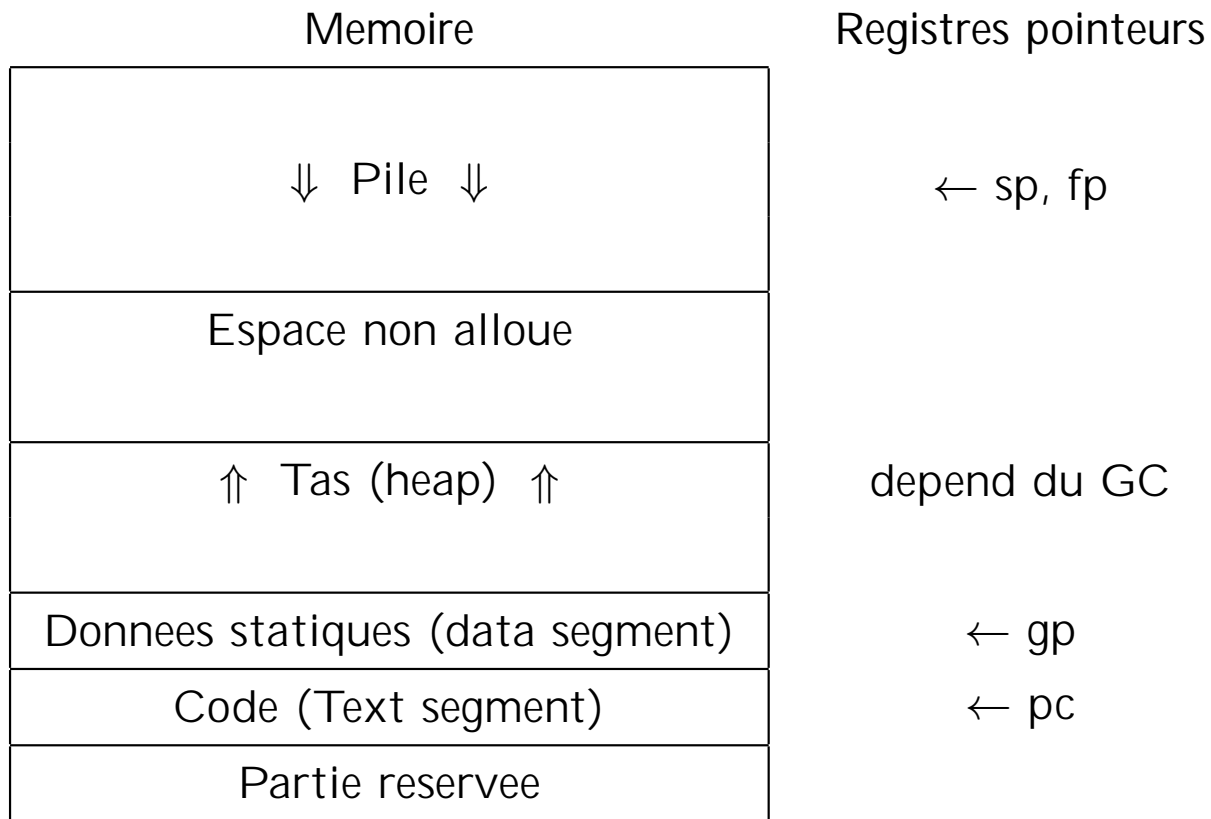
<http://www.iar.com/website1/1.0.1.0/244/1/>

**NB** En effet ces compilateurs considèrent des extensions de C qui permettent de préciser l'allocation de certaines données, la taille des pointeurs, la manipulation de certains registres, les conventions d'appel, :::

## Points principaux a comprendre (pour compiler RTLAbS)

- Architecture de la **memoire**.
- Organisation des **registres** de la CPU.
- Jeu d'**instructions** : transfert, calcul et saut.
- **Conventions d'appel**.
- Eventuellement les **entrees-sorties** (appels système), pour faire des tests.
- Notions sur l'**execution** (émulateur).

## Mips : architecture de la memoire





## Modes d'adressage

Seulement les instructions **load** et **store** peuvent accéder à la mémoire. On dispose des modes d'adressage suivants :

Format	Address Computation
(register)	contents of register
imm	immediate
imm (register)	immediate + contents of register
symbol	address of symbol
symbol $\pm$ imm	address of symbol + or – immediate
symbol (register)	address of symbol + contents of register
symbol $\pm$ imm (register)	(address of symbol + or – immediate) + contents of register

**NB** On adresse l'octet mais un mot mémoire occupe 4 octets (32 bits).

## Mips : registres et conventions d'utilisation

Nom	Nombre	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Expression evaluation and results of a function
v1	3	
a0-3	4	Arguments 1-4
t0-7	8-15	Temporaries (not preserved across call)
s0-7	16-23	Saved temporaries (preserved across call)
t8-9	24-25	Temporaries (not preserved across call)
k0-1	26-27	Reserved for OS kernel
gp	28	Pointer to global area
sp	29	Stack pointer
fp or s8	30	Frame pointer
ra	31	Return address (used by function call)

## Mips : jeu d'instructions

On distingue trois familles d'instructions :

**Transfert** Déplacer les données entre les registres et la mémoire.

**Calcul** Effectuer des calculs où les arguments sont dans des registres et le résultat est sauvé dans un registre.

**Saut** Branchement (conditionnel), Saut et retour d'une routine, Appel système.

## Exemples d'instructions de transfert

`lw Rdest; address`      *Load word*

Load the 32-bit quantity (word) at `address` into register `Rdest`.

`sw Rsrc; address`      *Store word*

Store the word from register `Rsrc` at `address`.

## Exemples d'instructions de calcul

`li Rdest; imm`      *Load immediate*

Move the immediate `imm` into register `Rdest`.

`addi Rdest;Rsrc; imm`      *Add immediate (with over ow)*

Put the sum of the integer from register `Rsrc` and of `imm` (16 bits) into register `Rdest`.

`add Rdest; Rsrc1;Rsrc2`      *Add (with overflow)*

Put the sum of the integer from registers `Rsrc1` and `Rsrc2` into register `Rdest`.

`slt Rdest; Rsrc1;Rsrc2`      *Set on less than*

Set register `Rdest` to 1 if register `Rsrc1` is less than `Rsrc2` and to 0 otherwise.

## Exemples d'instructions de saut

`j label`      *Jump*

Unconditionally jump to the instruction at the label.

`bgtz Rsrc;label`      *Branch on Greater Than Zero*

Conditionally branch to the instruction at the label if the contents of `Rsrc` are greater than 0.

`jal Rsrc`      *Jump and Link*

Unconditionally jump to the instruction whose address is in register `Rsrc` and save the address of the next instruction in register 31 (`ra`).

`jr Rsrc`      *Jump and Link*

Unconditionally jump to the instruction whose address is in register `Rsrc`.



`syscall`     *System call*

Register `v0` contains the number of the system call.

### Mips : appels systeme (extrait)

Service	System Call Code	Arguments	Result
print_int	1	\$a0 = integer	integer (in \$v0)
print_string	4	\$a0 = string	
read_int	5		
exit	10		

## Programme 'Hello world'

<code>.data</code>	<code>#Des données suivent</code>
<code>hello:</code>	<code>#L'adresse de la donnée suit</code>
<code>.ascii "hello world\n"</code>	<code>#Une chaîne terminée par 0</code>
<code>.text</code>	<code>#Des instructions suivent</code>
<code>main:</code>	
<code>li \$v0, 4</code>	<code>#Code appel print_string</code>
<code>la \$a0, hello</code>	<code>#Adresse chaîne dans a0</code>
<code>syscall</code>	<code>#Appel système</code>
<code>jr \$ra</code>	<code>#Fin du programme</code>

## Programme qui imprime l'entier lu

```
.text
main:
    li    $v0, 5          # Code read_int
    syscall               # Appel système
    move  $t2, $v0        # Place valeur lue dans $t2
    li    $v0, 1          # Code print_int
    move  $a0, $t2        # Place valeur à écrire dans $a0
    syscall               # Appel système
    jr    $ra             # Fin du programme
```

## Le factoriel recursif : un exemple de gestion de la pile en Mips

```
.text
main:
    li $v0, 5           # retrieve input from user
    syscall
    move $a0, $v0       # store user input as first argument
    jal fact            # compute factorial
    move $t0, $v0       # save result in t0
    move $a0, $t0       # display result
    li $v0, 1
    syscall
    li $v0, 10          # end
    syscall
```

```
fact:
    move $fp, $sp          # allocate space and save $ra and $a0
    addi $sp, $sp, -8
    sw $a0, 0($fp)
    sw $ra, -4($fp)
    bgtz $a0, fact_L1      # test if argument equals 0
    li $v0, 1
    j fact_return
fact_L1:                  # recursive call branch
    addi $a0, $a0, -1
    jal fact
    addi $fp, $sp, 8
    lw $a0, 0($fp)
    mult $a0, $v0          # result is in hi,lo registers of mult unit
    mflo $v0              # put low order result of mult in v0
    j fact_return
fact_return:              # return sequence
    lw $ra, -4($fp)       # restore $ra and $a0 registers
    lw $a0, 0($fp)
    move $sp, $fp
    jr $ra
```

### Exercice (recursion avec un seul pointeur a la pile)

- Cette programmation du factoriel s'inspire des conventions d'appel de Mips (à venir) et utilise deux pointeurs à la pile :  $\$sp$  et  $\$fp$ .
- Reprogrammez le factoriel en utilisant seulement le pointeur  $\$sp$ .

### Exercice (recursion)

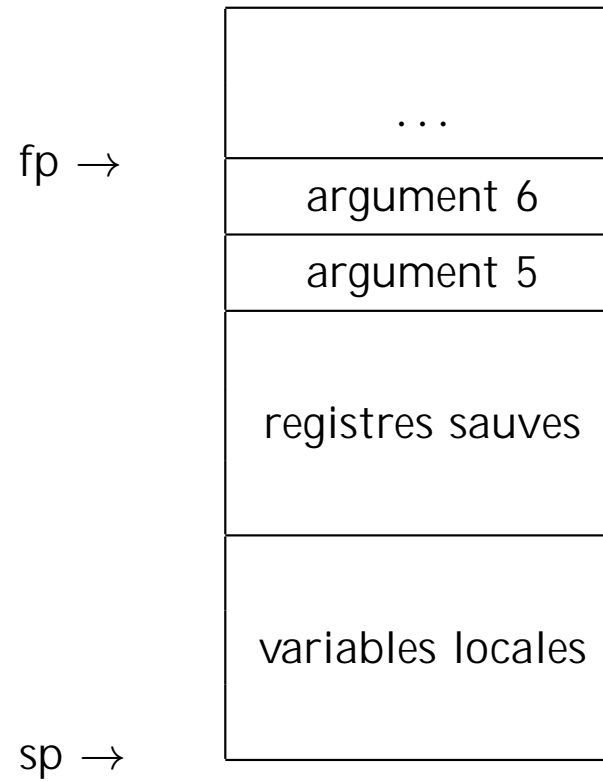
Écrire une routine `Fib` pour la fonction de FIBONACCI :

$$f(1) = 1; \quad f(2) = 1; \quad f(n+2) = f(n) + f(n+1)$$

en vous inspirant des conventions d'appel du factoriel.



## Mips : structure du bloc d'activation (frame)



## Notes sur les conventions d'appel

**fp** pointe au premier mot machine du bloc d'activation et **sp** au dernier (comme la pile croît vers le bas, **fp** est au dessus de **sp**).

- L'**appelant (caller)** effectue les opérations suivantes :
  - Les premiers 4 arguments sont passés dans les registres **a0–3**. Les autres, s'il y en a, vont sur la pile.
  - Il sauve les registres qui doivent être sauvés par l'appelant (**caller**) **t0–9** s'il les utilise, car l'appelé peut utiliser ces registres sans les sauver.

- Le code d'une **fonction appelee** commence par :
  - allouer la mémoire pour ses variables locales.
  - Sauver les registres sous le contrôle de l'**appele** (**callee**) `s0-7`, `fp`, `ra` si une utilisation de ces registres est envisagée.
  - Recalculer les valeurs de `fp`, `sp`.

- Au **retour** de l'appel, l'appelé doit :
  - Mettre le résultat dans le registre **v0**.
  - Restaurer les registres sauvés par l'appelé.
  - Recalculer la valeur de **sp**.
  - Sauter à l'adresse contenue dans le registre **ra**.

**NB** Certains compilateurs utilisent seulement **sp** et dans ce cas tous les calculs des offsets sont relatifs à **sp**.

### Remarque (allocation statique du bloc d'activation)

- Si le langage ne permet pas une suite cyclique d'appels :

$f$  appelle  $g \cdots$  appelle  $h$  appelle  $f$

alors à chaque instant du calcul la pile contient **au plus un bloc** d'activation pour la fonction  $f$ .

- Dans ce cas, on pourrait allouer **statiquement** un bloc d'activation pour chaque fonction.

## 8051 : organisation de la memoire

**Memoire Externe Donnees** 64 K octets (2 octets pour l'adresser dans  $DPTR=DPH+DPL$ ).

**Memoire Externe Code** 64 K octets (2 octets pour l'adresser dans PC).

**Memoire Interne** 256+128 octets (8 octets + mode d'adressage) (utilisation non-uniforme!).

**NB** Ici et dans la suite on se réfère au 8052 qui dispose d'une **extension** de 128 octets de la mémoire interne qui est adressable seulement de façon **indirecte**.

## 8051 : organisation de la memoire interne

**IRAM :0X00-0X2F** RAM interne adressable. Par défaut, les premiers 8 octets correspondent aux registres : R0-7. On peut adresser les **bits** des octets d'adresse 0X20-0X2F.

**IRAM :0X30-0X7F** Contient, entre autres, la **pile interne** qui est utilisée pour sauver le PC au moment de l'appel. Il faut faire attention à l'**initialisation** du registre SP.

**SFR : 0X80-0XFF** **Special Function Registers** comprennent : I/O, Timers, Interruptions, Pointeur à la pile (SP), Pointeur aux données (DPTR), Accumulateurs (A, B).

**IRAM additionnelle : 0X80-0XFF** Le 8052 contient 128 octets de mémoire interne additionnelle qui a en principe les mêmes adresses que les SFR. Pour faire la différence on utilise deux modes d'adressage différents : **direct pour les SFR** et **indirect pour la IRAM additionnelle**.

## Exemple adressage de la memoire interne

```
MOV A, 0X90          ; Écrit en A l'octet d'adresse 0X90 (SFR)

MOV R0, #0X90        ; Écrit en R0 la valeur 0X90
MOV A, @R0            ; Écrit en A l'octet de l'IRAM dont l'adresse est dans R0
```

**NB** L'adressage indirect s'applique toujours a l'IRAM (et jamais aux SFR).



## Registres principaux

Nom	Octets	Fonction
A	1	Accumulateur
B	1	Acc. pour mult. et division
SP	1	Pointeur a la pile initialise a 07x.
R0-7	1	Registres
PC	2	Compteur programme
DPTR	2	Pointeur aux donnees

**NB** Le SP est incrémenté de 1 avant d'insérer une valeur dans la pile.

## Modes d'adressage

Nom	Exemple
Immediat	MOV A, #0X20
Direct	MOV A, 0X30
Indirect	MOV A, @R0
Externe direct	MOVX A, @DPTR    MOVX @DPTR, A

**NB** Il y a aussi une forme d'adressage externe indirect qui est peu utilisé et qu'on ignore.

## 8051 : jeu d'instructions

**Format** [label :] mnemonic [operands] [; comment]

Nom	Exemple
Addition	ADD A, R1
Move	MOV R0, A
External Data Move	MOVX @DPTR, A
External Code Move	MOVC A, @A + DPTR
Jump if A=0	JZ Label
Long Jump	LJMP Label
Long Call	LCALL Label
Return	RET
Pop	POP R2
Push	PUSH R3

**NB** MOVX permet de lire/écrire la RAM externe des données alors que MOVC permet seulement de lire la RAM externe du code.

## Exemple de programme genere par SDCC

Programme `sum.c` :

```
int sum(int a)
{
    if(a == 0)
        return 0;
    else
        return a + sum(a-1);
}

int main()
{
    return sum(5);
}
```

Programme `sum.asm` généré par SDCC -S `sum.c` :

[123 lignes d'initialisation et commentaires omises !]

`_sum:`

`ar2 = 0x02`

`ar3 = 0x03`

`ar4 = 0x04`

`ar5 = 0x05`

`ar6 = 0x06`

`ar7 = 0x07`

`ar0 = 0x00`

`ar1 = 0x01`

`mov r2,dpl`

;on sauve l'argument (dph,dpl) dans (r3,r2)

`mov r3,dph`

`; sum.c:4: if(a == 0)`

`mov a,r2`

`orl a,r3`

; a= (a or r3) ou logique

`jnz 00102$`

; si a n'est pas zero on a un appel récursif

`; sum.c:5: return 0;`

`mov dptr,#0x0000`

; sinon on retourne le résultat 0

`ret`

00102\$:

; sum.c:7: return a + sum(a-1);

mov a,r2

add a,#0xff

mov dpl,a

mov a,r3

addc a,#0xff

mov dph,a ; (dph,dpl)=(r3,r2)-1

push ar2 ;

push ar3 ;on empile (r3,r2)

lcall \_sum

mov r4,dpl

mov r5,dph ; (r5,r4) est le resultat de l'appel récursif

pop ar3

pop ar2 ; (r3,r2) contiennent la valeur de l'argument

mov a,r4

add a,r2

mov dpl,a

mov a,r5

addc a,r3 ; (dph,dpl)=(r5,r4)+(r3,r2)

mov dph,a [nouvelle ligne] ret

```
_main:
; sum.c:12: return sum(5);
    mov dptr,#0x0005      ; on met l'argument 5 dans dptr
    ljmp _sum             ; on saute à _sum
```

## Remarques

- Le mode de compilation par défaut de SDCC est

`small memory model`

- Dans le cas en question, les entiers sont représenté sur **16 bits** et on utilise la **pile interne** pour les appels récursif.
- On remarquera que le code produit ne contient pas d'instructions d'échange avec la mémoire externe (**small memory=pas de memoire externe**).



## Exercice (large-memory)

Compilez le même programme `sum.c` avec l'option `--model-large` :

```
sdcc -S --model-large sum.c
```

Dans ce cas, le compilateur utilise aussi la **RAM externe**.  
Équipez-vous du **Tutorial** et cherchez à comprendre chaque instruction du code produit dans `sum.asm`.

Examinez aussi le cas de la fonction de FIBONACCI.

## 8051 : conventions d'appel

Il y en a pas::: ou il y en a trop ce qui revient au même !

- SDCC et Keil ne donnent pas de détails.
- Le 8051 IAR C/C++ Compiler (Reference Guide) en propose 6 qui peuvent être spécifiées par le programmeur.

A noter que l'utilisation d'appels récursifs dans les applications typiques du 8051 est **exceptionnelle**. En particulier on ne devrait pas les utiliser avec l'option par défaut `model-small`.

## Points non traites

**Directives du langage assembleur** Elles sont utilisées massivement dans le code compilé `.asm` et elles sont ensuite éliminées par l'assembleur. Exemples :

```
MY_VAL  DATA    0x44    ; RAM location
```

```
LIMIT   EQU      2000    ; LIMIT = 2000
```

```
COUNT   EQU      R5      ; COUNT synonyme pour R5
```

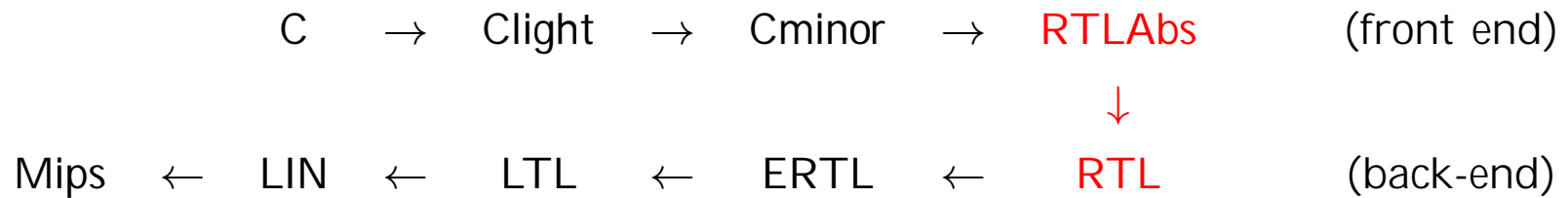
```
CSEG AT 0X1000          ; adresse prochaine instruction
```

**Entrees-sorties** Beaucoup plus compliqué qu'en Mips. Il faut programmer les **portes seriales** et cette programmation dépend de la **frequence** du processeur (SIC!).

**Interruptions** Elles peuvent être externes (par exemple, entrée disponible) ou internes (par exemple, débordement). Elles utilisent la **pile interne** et on peut leur associer une **priorite**.

**Emulateurs** Voir <http://mcu8051ide.sourceforge.net/>.

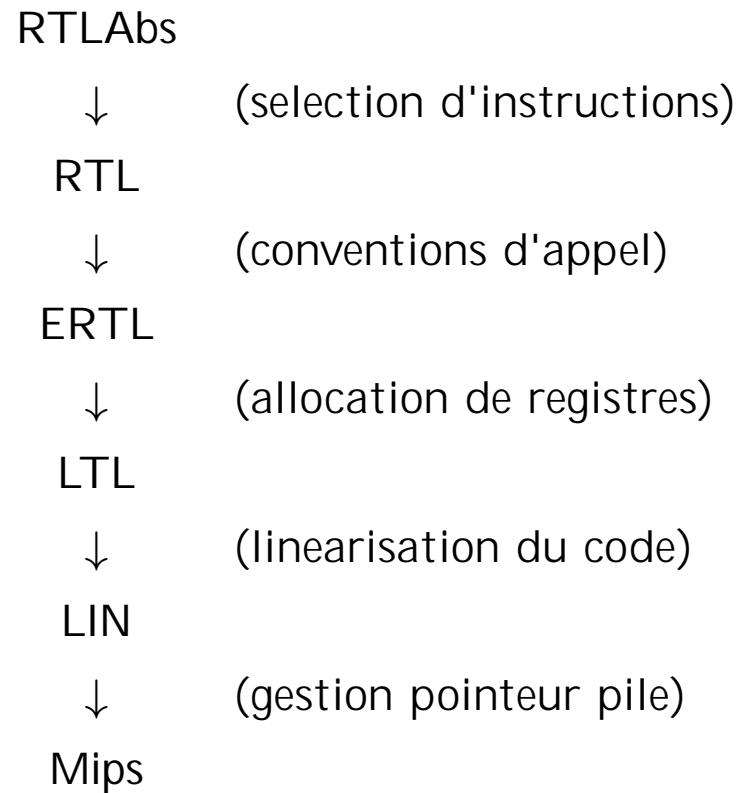
## Back-end : de RTLAbs à RTL



## Plan du cours

- Survol du back-end.
- Notions sur la sélection d'instructions.

## Les transformations du back-end





## sum en Cminor

Le programme source est celui qui calcule la somme des nombres naturels de 0 à  $a$ .

```
"sum" (a) : int -> int {  
  stack 0;                                (taille données dans la pile)  
  var t;                                  (temporaire)  
  if (a == 0) { return 0; }  
  else {t = "sum"(a - 1) : int -> int; (pas d'appels dans les expressions)  
    return a + t; } }
```

**NB** Le compilateur est en développement et les détails peuvent changer d'une version à la suivante.

## sum en RTLAbs : explicitation CFG

program:

```

"sum"(%0): int -> int
  locals: %2, %1, %3, %4, %5, %6, %7      (pseudo-registres (pr))
  result: %2                             (pr résultat)
  stacksize: 0
  entry: sum11                            (noeud d'entrée CFG)
  exit:  sum0                             (noeud de sortie CFG)
  sum11: imm_int 0, %6,      --> sum10
  sum10: imm_int 0, %7,      --> sum9
  sum9:  seq %5, %0, %7      --> sum8      (set on equal: %5 = (%7=%0))
  sum8:  beq %5, %6          --> sum4, sum6 (branch on equal- redondant!)
  sum6:  imm_int 0, %2,      --> sum0      (cas terminal)
  sum4:  imm_int 1, %4,      --> sum3      (cas récursif)
  sum3:  sub %3, %0, %4      --> sum2
  sum2:  call "sum", %3, %1  --> sum1      (appel récursif)
  sum1:  add %2, %0, %1      --> sum0      (on prépare le résultat)
  sum0:  return %2          (on retourne)

```

## sum en RTL : selection instructions Mips

```
function sum(%0) : %2                                (arg dans %0, res dans %2)
stacksize 0
var %0, %1, %2, %3, %4, %5, %6, %7, %8
entry sum20
sum20: li      %1, 0                                --> sum19          (initialisation optionnelle pr)
...    :      ...      ...                        --> ...
sum10: li      %8, 0                                --> sum9
sum9  : seq    %5, %0, %7                          --> sum8          (test traduit par test)
sum8  : beq    %5, %6                              --> sum4, sum6
sum6  : li     %2, 0                                --> sum0          (on traduit imm_int en li)
sum4  : li     %4, 1                                --> sum3
sum3  : sub    %3, %0, %4                          --> sum2          (a-1 dans %3)
sum2  : la     %8, sum                             --> sum13        (adresse sum dans %8)
sum13: call    %1, %8(%3)                         --> sum1          (valeur retour dans %1)
sum1  : add    %2, %0, %1                          --> sum0          (addition)
sum0  : return %2                                (retour)
```

**sum en ERTL : conventions d'appel**

```

procedure sum(1)
stacksize 0
var %0, %1, ..., %15, %16
entry sum30
sum30: newframe          --> sum29      (allocation bloc d'activation)
sum29: move  %16, $ra     --> sum28      (sauve adresse retour)
sum28: move  %15, $s6     --> sum27      (sauve s0-6)
sum27: ...    ..., ...   --> ...
sum22: move  %9, $s0      --> sum21
sum21: move  %0, $a0      --> sum20      (sauve argument)
sum20: li    %1, 0        --> sum19      (optionnel: init %0-8)
sum19: ...    ..., ...   --> ...
sum10: li    %7, 0        --> sum9
sum9 : seq   %5, %0, %7   --> sum8      (test)
sum8 : beq   %5, %6       --> sum4, sum6
sum6 : li    %2, 0        --> sum0      (résultat=0)
sum0 : j                                           --> sum41
sum4 : li    %4, 1        --> sum3      (cas récursif)
sum3 : sub   %3, %0, %4   --> sum2

```

sum2 : la	%8, sum	-->	sum13	
sum13: j		-->	sum44	(nop)
sum44: move	\$a0, %3	-->	sum43	
sum43: call	%8(1)	-->	sum42	(appel récursif)
sum42: move	%1, \$v0	-->	sum1	(sauve résultat)
sum1 : add	%2, %0, %1	-->	sum0	(résultat appel dans %2)
sum41: move	\$v0, %2	-->	sum40	(suite de retour)
sum40: move	\$ra, %16	-->	sum39	(restauration adresse retour)
sum39: move	\$s6, %15	-->	sum38	(restauration s0-6)
sum38: ...	..., ...	-->	...	
sum33: move	\$s0, %9	-->	sum32	
sum32: delframe		-->	sum31	(deallocation bloc d'activation)
sum31: jr	\$ra			(retour)

## sum en LTL : analyse de vivacite et allocation de registres

```

procedure sum(1)
var 8
entry sum30
sum30: newframe          --> sum29
sum29: lw    $ra, 4($sp)  --> sum28    (sauve adresse retour)
sum28: j     --> sum27    (suite de 'nop')
...   : j     --> ...
sum23: j     --> sum22
sum22: lw    $s0, 0($sp)  --> sum21    (sauve $s0)
sum21: move  $s0, $a0     --> sum20
sum20: j     --> sum19    (suite de 'nop')
...   : j     --> ...
sum11: j     --> sum10
sum10: li    $v0, 0       --> sum9
sum9  : seq  $v0, $s0, $v0 --> sum8
sum8  : beq  $v0, $zero    --> sum4, sum6
sum6  : li    $v0, 0       --> sum0    (résultat=0)
sum4  : li    $v0, 1       --> sum3    (cas récursif)
sum3  : sub  $a0, $s0, $v0 --> sum2

```

```
sum2 : la    $v0, sum    --> sum13
sum13: j      --> sum44    (encore des 'nop')
sum44: j      --> sum43
sum43: call   $v0        --> sum42
sum42: j      --> sum1
sum1 : add    $v0, $s0, $v0 --> sum0
sum0 : j      --> sum41    (suite de retour)
sum41: j      --> sum40
sum40: lw     $ra, 4($sp) --> sum39
sum39: j      --> sum38    (suite de 'nop')
...   : j      --> ...
sum34: j      --> sum33
sum33: lw     $s0, 0($sp) --> sum32
sum32: delframe          --> sum31
sum31: jr     $ra
```

**sum en LIN : linearisation**

```
procedure sum(1)
var 8
sum30:
newframe
sw    $ra, 4($sp)      (sauve $ra)
sw    $s0, 0($sp)      (sauve $s0)
move  $s0, $a0         (test)
li    $v0, 0
seq   $v0, $s0, $v0
beq   $v0, $zero, sum5
li    $v0, 0           (résultat = 0)
sum40:                 (suite de retour)
lw    $ra, 4($sp)
lw    $s0, 0($sp)
delframe
jr    $ra
sum5:                  (cas récursif)
li    $v0, 1
sub   $a0, $s0, $v0
```



```
la    $v0, sum
call  $v0
add   $v0, $s0, $v0
j     sum40
```

## sum en Mips

```
.align 2                                (mots allignes sur multiples de 4)
sum:
addi    $sp, $sp, -8                    (traduction newframe)
sw      $ra, 4($sp)
sw      $s0, 0($sp)
move    $s0, $a0
li      $v0, 0
seq     $v0, $s0, $v0
beq     $v0, $0, sum5
li      $v0, 0
sum40:                                    (suite retour)
lw      $ra, 4($sp)
lw      $s0, 0($sp)
addi    $sp, $sp, 8                      (traduction delframe)
jr      $ra
sum5:                                    (cas récursif)
li      $v0, 1
sub     $a0, $s0, $v0
la      $v0, sum
```

```
jalr    $v0  
add     $v0, $s0, $v0  
j       sum40
```

## Un petit exercice informel

Revisiter la compilation dans le back-end en supposant que le processeur cible est le 8051.

$\text{RTLAbs} \rightarrow \dots \rightarrow 8051$

Dans ce cas, on doit distinguer **deux types** :

**Donnees=1 octet** Un mot mémoire, un registre (= une adresse de la mémoire interne). Ceci suppose que toutes les données ont été décomposées en octets.

**Adresses=2 octets** Une adresse de la mémoire externe (données ou code).

On peut faire des **hypotheses** (qui semblent raisonnables).

- A un renommage près, on dispose en 8051 des registres (1 octet) : **s0-6**, **a0-4** (arguments) et **v0** (résultat).
- Par ailleurs, on dispose de couples de registres (2 octets) pour représenter **sp** (pile), **ra** (retour), **gp** (global), :::
- Les instructions Mips qui travaillent sur des données homogènes peuvent être réalisées par une **suite** d'instructions 8051.

**NB** On va marquer avec **ADR** les endroits où l'on manipule des adresses (couples d'octets).

**sum en 8051 (avant macro-expansion)**

```

.align 2                                (mots allignés sur multiples de 4)
sum:
addi    $sp, $sp, -8    ADR (traduction newframe)
sw      $ra, 4($sp)    ADR
sw      $s0, 0($sp)    ADR
move    $s0, $a0
li      $v0, 0
seq     $v0, $s0, $v0
beq     $v0, $0, sum5
li      $v0, 0
sum40:                                (suite retour)
lw      $ra, 4($sp)    ADR
lw      $s0, 0($sp)    ADR
addi    $sp, $sp, 8    ADR (traduction delframe)
jr      $ra            ADR
sum5:                                (cas récursif)
li      $v0, 1
sub     $a0, $s0, $v0
la      $v0, sum        ADR ERREUR DE TYPE v0! (en 8051 on peut utiliser LCALL)

```

```
jalr    $v0          ADR
add     $v0, $s0, $v0
j       sum40        ADR
```

## Ce qui resterait a faire pour generer un programme 8051

- Enlever l'erreur de type en utilisant LCALL
- Recalculer le décréement/incréement de la pile (3 au lieu de 8) et les offsets de l'argument et de l'adresse de retour.
- Faire une 'macro-expansion' des instructions Mips en supposant disposer d'un petit nombre de 'temporaires'.



# Sélection d'instructions

## Selection d'instructions en CerCo

- La sélection d'instructions de RTLabs à RTL est **tres simple** car la correspondance entre les instructions est pratiquement 1 à 1.
- *A contrario*, la sélection d'instructions dans SDCC pour le 8051 est un programme C de  $5K$  LOC.
- Dans ce qui suit nous allons nous placer dans un cadre un peu plus abstrait et examiner quelques méthodes pour “améliorer l'efficacité” du processus de sélection d'instructions (lire le **chapitre 9** de APPEL pour plus d'informations).

## Expressions langage intermediaire

- On suppose qu'on peut représenter une **expression** du langage intermédiaire comme un **arbre etiquete**.
- Par exemple, l'expression  $a[i+2]$  où :
  - $a$  est un **o set** par rapport à la mémoire **globale**.
  - $i$  est un **pseudo-registre**.revient à calculer  $(\$gp + a) + ((i + 2) * 4)$ , ce qu'on peut **abstraire** par l'arbre étiqueté :

$$\text{sum}(\text{sum}(\text{reg}, \text{cnst}), \text{prod}(\text{sum}(\text{reg}, \text{cnst}), \text{cnst}))$$

$\text{reg}$  = valeur contenue dans un registre,       $\text{cnst}$  = valeur constante

## Instructions de la machine

Comme instructions de la machine, on peut prendre, par exemple, un sous-ensemble de celles de Mips :

Instruction	Semantique
<code>addu rd,rs1,rs2</code>	$rd := rs1 + rs2$
<code>mul rd,rs1,rs2</code>	$rd := rs1 * rs2$
<code>addiu rd,rs,imm</code>	$rd := rs + imm, imm < 2^{16}$
<code>sll rd,rs,sh</code>	$rd := \text{shift}_{sh}(rs)$
<code>lw imm(rs)</code>	$rd := M[rs + imm]$
<code>move rd,rs</code>	$rd := rs$

## Construction des tuiles

- Une **tuile** est un fragment d'arbre étiqueté dans la notation du langage intermédiaire.
- Plus formellement, on peut associer une **signature** au langage intermédiaire, c'est à dire un ensemble de symboles avec une arité

$$\Sigma = \{\text{cnst}^0; \text{reg}^0; \text{sum}^2; \text{prod}^2\}$$

et voir une tuile comme un **terme du premier ordre** sur la signature dont toutes les variables sont différentes. Par exemple :

$$\text{sum}(\text{cnst}; \text{prod}(\text{x}; \text{y}))$$

On peut voir un tel terme comme un **ltre** (linéaire) dans le sense de **ocaml**. Un arbre (terme)  $t$  sur la signature  $\Sigma$  passe le filtre (**match**)  $f$  s'il y a une substitution  $S$  telle que  $t = S(f)$ .

- On peut associer à chaque instruction de la machine un **certain nombre de tuiles/ ltres** (filtres).
- Par exemple, en sachant qu'il y a un registre qui contient toujours la valeur 0, avec l'instruction **addiu** on pourrait associer les tuiles :

$f_1$	$f_2$	$f_3$	$f_4$	$f_5$	$f_6$
cnst	reg	sum(reg, cnst)	sum(cnst, reg)	sum(x, cnst)	sum(cnst, x)

## Selection d'instructions comme probleme d'optimisation

- On suppose avoir associé un **coût** à chaque tuile.
- Étant donné un arbre on cherche à déterminer :
  - s'il peut être recouvert par des **tuiles disjointes** (en général c'est le cas ou alors il y a un problème de conception du jeu d'instructions!),
  - et dans ce cas à déterminer un **recouvrement de coût minimal**.

### Remarque (sur le coût)

- On cherche à minimiser le coût **par rapport a un ensemble de tuiles** qu'il faut déterminer.
- Dans un processeur avec pipeline, cache, :: le coût d'une suite d'instructions **n'est pas** la somme des coûts de chaque instruction. Le modèle combinatoire est donc une approximation de la réalité.
- Si toutes les tuiles ont le même coût alors on peut essayer de **minimiser le nombre d'instructions generees**, ce qui revient à minimiser le nombre de tuiles qui recouvrent l'arbre.



## Remarque (sur la selection d'instructions dans CerCo)

- Dans le compilateur CerCo, on passe :
  - d'instructions 'abstraites' qui opèrent sur des **pseudo-registres**.
  - à des instructions Mips qui opèrent aussi sur des pseudo-registres !

C'est la transformation  $\text{ERTL} \rightarrow \text{LTL}$  qui s'occupe de remplacer les pseudo-registres par des 'vrais' registres.

- Au niveau RTLabs on a déjà perdu la **structure des expressions**. Si l'on souhaite exploiter cette structure, il convient de faire la sélection des instructions directement au niveau de Cminor.

## Exercice

On prend comme signature :  $\Sigma = \{a^0; g^2\}$ . On suppose les tuiles (filtres) et coûts suivants :

Tuile	Coût
$f_1 = a$	1
$f_2 = g(a, a)$	2
$f_3 = g(x, y)$	3
$f_4 = g(g(x, y), g(w, z))$	4

1. Montrez qu'il y **toujours une solution** au problème de recouvrement. Quels sont les **ensembles minimaux** de tuiles parmi celles considérées qui ont cette propriété ?
2. Considérez l'arbre  $t_0 = g(g(a; a); g(a; a))$ . Quels sont les **recouvrements possibles** et quels sont les **recouvrements de coût minimal** ?

## Algorithme glouton (maximal munch)

- On énumère les noeuds de la racine vers les feuilles (**top-down**).
- Au noeud  $n$  de l'arbre, on cherche la tuile qui couvre le **plus grand nombre de noeuds** de l'arbre de racine  $n$ .
- Ensuite, on applique récursivement la stratégie aux **sous arbres** qui ne sont pas couverts par la tuile (proposez une définition formelle).

**NB** Les instructions sont générées dans l'**ordre inverse** à celui de visite des noeuds.

### Exercice (suite)

- **Appliquez** l'algorithme glouton à l'arbre  $t_0$  considéré.
- Le résultat de l'algorithme glouton change-t-il si l'on prend comme critère d'optimisation local le **ratio** :  
*nombre noeud couverts / coût tuile?*

## Propriétés de l'algorithme glouton

- Si on trouve une solution alors il n'est pas possible d'en trouver une 'meilleure' (au sens du coût) en fusionnant des tuiles adjacentes (une propriété d'**optimalité locale**).
- C'est **facile à implémenter** dans un langage avec du filtrage (comme ocaml). Il faut veiller à ordonner les filtres de façon 'décroissante'.
- Bons résultats pour des architectures RISC avec **instructions simples**.

## Programmation dynamique (rappel)

- La programmation dynamique est une **technique d'optimisation** (et de programmation) qui consiste à chercher une solution d'un **probleme** en calculant d'abord les solutions de ses **sous-problemes**.
- La technique est 'intéressante' quand :
  1. le nombre de sous-problèmes est **polynomial** et
  2. il est **facile** de calculer la solution du problème à partir des solutions de ses sous-problèmes.

## Exemple de programmation dynamique

**Probleme** On veut savoir si une grammaire algébrique avec productions en **forme normale de Chomsky**

$$X \rightarrow YZ; \quad X \rightarrow a; \quad (X; Y; Z \text{ non-terminaux; } a \text{ terminal})$$

génère un mot  $w$ .

**Sous-problemes** On énumère tous les sous-mots de  $w$  de longueur 1, puis ceux de longueur 2, :::, puis ceux de longueur  $|w|$ . Pour chaque sous-mot on calcule l'ensemble de non-terminaux qui peuvent le générer.

**Nombre de sous-problemes** Soit  $n = |w|$ . Un sous-problème est déterminé par un couple  $(i; j)$  tel que  $1 \leq i \leq j \leq |w|$ . Donc on a  $O(n^2)$  sous-problèmes.

**Des sous-problemes au probleme** Soit  $w = a_1 \cdots a_m$  un mot de longueur  $m > 1$ .

- La dérivation du mot  $w$  commence forcément par une production  $A \rightarrow BC$ .
- On énumère les productions de la forme  $A \rightarrow BC$  et les nombres  $i$  tels que  $1 \leq i \leq (m-1)$  et on vérifie si  $B$  peut générer  $a_1 \cdots a_i$  et  $C$  peut générer  $a_{i+1} \cdots a_m$ .

**NB** La complexité est  $O(n^3)$ . Notez qu'il est essentiel d'**enumerer les problemes a partir des plus petits**, autrement on risque de re-calculer plusieurs fois la solution du même problème ce qui mène à une complexité exponentielle.



## Programmation dynamique appliquee au probleme du recouvrement

- On énumère les noeuds de l'arbre des feuilles vers la racine (**bottom-up**).
- Quand on est au noeud  $n$  on a déjà déterminé le coût minimum de tous les **descendants** de  $n$ .
- On énumère toutes les **tuiles qui sont compatibles** avec le noeud  $n$  (le sous-arbre de racine  $n$  matche le filtre). Pour chaque tuile  $t$  de coût  $c$  il y aura un certain nombre de sous-arbres de racine  $n_1; \dots; n_k$  ( $k \geq 0$ ) qui ne sont pas couverts par la tuile et dont le coût minimum est  $c_1; \dots; c_k$ .
- On dérive que  $c + \sum_{i=1, \dots, k} c_i$  est une **borne sup au coût** du sous-arbre  $n$ . Le **coût** de  $n$  est le minimum des bornes sup.

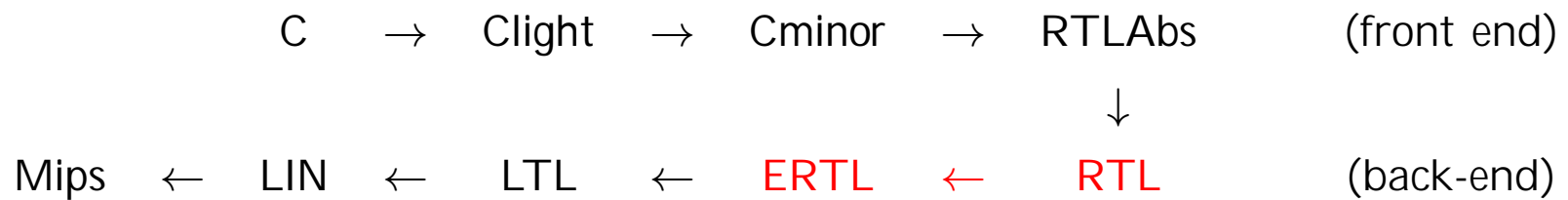
## Exercice (suite)

**Appliquez** la méthode de programmation dynamique à l'arbre  $t_0$ .

## Propriétés de l'algorithme par programmation dynamique

- Trouve une **solution optimale** (s'il y en a une).
- Toujours un algorithme **polynomial**. Et efficace en pratique.
- **Programmation lourde**. Selon APPEL, une spécification à l'aide de grammaires algébriques pour Motorola 68020 a 141 règles et 35 non-terminaux.
- On utilise des **generateurs de generateurs de code** (dans l'esprit de Yacc).

## Back-end : de RTL à ERTL



## Plan du cours

- Explicitation des conventions d'appel Mips.

## Conventions d'appel et allocation de registres

- L'**allocation** de registres depend des **conventions d'appel**.
- D'autre part, la **gestion de la pile** dans les conventions d'appel depend de l'**allocation de registres**.
- La solution adoptée dans le compilateur est d'expliciter les conventions d'appel avant l'allocation de registres en gardant une **representation abstraite** des opérations qui manipulent le pointeur à la pile.
- Ces opérations seront explicitées seulement au dernier pas de compilation (de LIN à Mips).

## Conventions d'appel en ERTL

- Arguments et résultat d'une fonction sont transmis par des **registres physiques** (nouvelle entité du langage) et/ou la pile.
- L'**adresse de retour** de l'appelant est gérée explicitement.
- Les registres physiques sont **sauvegardés** explicitement.
- On traite (de façon abstraite) l'allocation et l'élimination des **blocs d'activation**.

**sum en ERTL (rappel)**

```

procedure sum(1)
stacksize 0
var %0, %1, ..., %15, %16
entry sum30
sum30: newframe          --> sum29      (allocation bloc d'activation)
sum29: move  %16, $ra     --> sum28      (sauve adresse retour)
sum28: move  %15, $s6     --> sum27      (sauve s0-6)
sum27: ...    ..., ...   --> ...
sum22: move  %9, $s0      --> sum21
sum21: move  %0, $a0      --> sum20      (sauve argument)
sum20: li    %1, 0        --> sum19      (optionnel: init %0-8)
sum19: ...    ..., ...   --> ...
sum10: li    %7, 0        --> sum9
sum9 : seq   %5, %0, %7   --> sum8      (test)
sum8 : beq   %5, %6       --> sum4, sum6
sum6 : li    %2, 0        --> sum0      (cas terminal)
sum0 : j     --> sum41
sum4 : li    %4, 1        --> sum3      (cas récursif)
sum3 : sub   %3, %0, %4   --> sum2

```



sum2 : la	%8, sum	-->	sum13	
sum13: j		-->	sum44	(nop)
sum44: move	\$a0, %3	-->	sum43	
sum43: call	%8(1)	-->	sum42	(appel récursif)
sum42: move	%1, \$v0	-->	sum1	(sauve résultat)
sum1 : add	%2, %0, %1	-->	sum0	(résultat appel dans %2)
sum41: move	\$v0, %2	-->	sum40	(suite de retour)
sum40: move	\$ra, %16	-->	sum39	(restauration adresse retour)
sum39: move	\$s6, %15	-->	sum38	(restauration s0-6)
sum38: ...	..., ...	-->	...	
sum33: move	\$s0, %9	-->	sum32	
sum32: delframe		-->	sum31	(deallocation bloc d'activation)
sum31: jr	\$ra			(retour)

## Back-end : de ERTL à LTL

(L'analyse de vivacite et l'allocation de registres)



## Plan du cours

- Points fixes et Définitions (co-)inductives.
- Analyse de flot de données.
- Coloration de graphes et Allocation de registres.

# Point fixes et Définitions (co-)inductives

## Ordres partiels (rappel)

- Un **ordre partiel**  $(L; \leq)$  est un ensemble  $L$  équipé d'une relation réflexive, anti-symétrique et transitive.
- Soit  $X \subseteq L$  (éventuellement vide). Un élément  $y \in L$  est une **borne supérieure** pour  $X$  si  $\forall x \in X \ x \leq y$ .
- Un élément  $y \in L$  est le **sup** de  $X$  s'il est la plus petite borne supérieure.
- De façon **duale**, on définit la notion de **borne inférieure** et de **inf**.

## Treillis complets, fonctions monotones et points fixes

- Un **treillis complet** est un ordre partiel  $(L; \leq)$  tel que tout sous-ensemble de  $L$  a un sup.
- Une **fonction monotone**  $f$  sur un ordre partiel  $L$  est une fonction qui respecte l'ordre :

$$\forall x; y \ (x \leq y \text{ implique } f(x) \leq f(y))$$

- On dit que  $x$  est un **point fixe** de  $f$  si  $f(x) = x$ .

### Exercice (sur les treillis)

- Montrez que les **parties d'un ensemble** avec la relation d'inclusion comme relation d'ordre forment un treillis complet.
- Montrez que tout sous-ensemble d'un treillis complet a un **inf**.
- Un **treillis** est un ordre partiel tel que tout sous-ensemble **ni** a un sup. Montrez que :
  - tout sous-ensemble d'un treillis **ni** a un inf.
  - tout treillis fini est **complet**.

### Exercice (sur les points- fixes [Tarsky])

- Montrez que toute fonction monotone sur un treillis complet a un **plus grand** et un **plus petit point fixe** qui s'expriment respectivement par :

$$\sup\{x \mid x \leq f(x)\} \quad \text{et} \quad \inf\{x \mid f(x) \leq x\} :$$

- Soit  $(\mathbf{N} \cup \{\infty\}; \leq)$  l'ensemble des nombres naturels avec un élément maximum  $\infty$ ,  $0 < 1 < 2 < \dots < \infty$ . Montrez que toute fonction monotone  $f$  sur cet ordre admet un point fixe.



### Exercice (points xes par iteration)

- Soit  $(L; \leq)$  un treillis **ni** et  $f : L \rightarrow L$  une fonction monotone.
- Soit  $\perp$  ( $\top$ ) **le plus petit (le plus grand)** élément de  $L$ .
- Si  $x \in L$  alors soit  $f^n(x)$  l'iteration  $n$ -fois de  $f$  sur  $x$  ( $f^0(x) = x$ ).
- Montrez que **le plus petit point xe** de  $f$  s'exprime par
$$\sup\{f^n(\perp) \mid n \geq 0\}$$
- Énoncez et démontrez la propriété pour **le plus grand point xe**.

- Montrez que les assertions sont **fausses** si l'on supprime l'hypothèse que le treillis est **ni**.
- Un sous-ensemble  $X$  d'un ordre partiel est **dirige** si

$$\forall x, y \in X \quad \exists z \in X \quad (x \leq z) \text{ et } (y \leq z)$$

- Une fonction sur un treillis complet est **continue** si elle préserve les *sup* d'ensembles dirigés :

$$f(\sup(X)) = \sup(f(X)) \quad (\text{si } X \text{ dirigé})$$

- Montrez que **le plus petit point fixe** d'une fonction  $f$  **continue** s'exprime par

$$\sup\{f^n(\perp) \mid n \geq 0\}$$

## Sommaire

Dans notre cadre, les deux cas qu'on rencontre **le plus souvent** sont :

**Treillis** ni Pour trouver le plus petit (le plus grand) point fixe il suffit d'itérer un nombre fini de fois la fonction monotone à partir de l'élément le plus petit (le plus grand).

**Fonction continue** Le **plus petit point**  $x_e$  est le *sup* de l'itération (dénombrable) de la fonction à partir du plus petit élément.

## Definitions inductives (exemples)

Soit  $\mathbf{Z}$  l'ensemble des nombres entiers et *SUC* et  $+$  les opérations successeur et addition, respectivement. On pourrait définir :

Le plus petit sous-ensemble de  $\mathbf{Z}$  qui contient  $\{0;2\}$  et qui est stable par rapport à l'opération d'addition.

Il n'est pas si évident qu'une définition inductive définit bien un ensemble. Il faut d'abord s'assurer que *le plus petit ensemble* dont parle la définition *existe*.

- Pour ce faire on peut expliciter une fonction  $f : 2^{\mathbb{Z}} \rightarrow 2^{\mathbb{Z}}$

$$f(X) = \{0; 2\} \cup \{x + y \mid x, y \in X\}$$

telle que “ $X$  contient  $\{0; 2\}$  et  $X$  est stable par rapport à l’opération d’addition” ssi  $f(X) \subseteq X$ .

- Ensuite on remarque que  $2^{\mathbb{Z}}$  est un treillis complet et que  $f$  est monotone (et même continue ; exercice !). Donc le plus petit point fixe existe et s’exprime par :

$$\bigcap \{X \mid f(X) \subseteq X\} = \bigcup_{n \geq 0} f^n(\emptyset) :$$

### Exercice (de définition inductives)

- Soit  $R$  une **relation binaire** sur un ensemble.
- Sa **clôture réflexive et transitive**  $R^*$  est la plus petite relation qui contient la relation identité, la relation  $R$  et telle que si  $(x;y);(y;z) \in R^*$  alors  $(x;z) \in R^*$ .
- Montrez que  $R^*$  peut être vu comme un **ensemble de définition inductivement**.

## Exercice

On définit de façon informelle  $X$  comme le plus petit ensemble d'entiers tel que :

1.  $0 \in X$ ,
2. si  $x \in X$  alors  $\forall y \in X \ ( (x \equiv (y + 1)) \bmod 2 )$ .

Est-ce une bonne définition ?

## Exercice (de nition co-inductive (1/2))

- La notion de **de nition co-inductive** est obtenue par **dualisation** de la notion de définition inductive. Plutôt que de chercher à construire le plus petit ensemble tel que::: on cherche maintenant **le plus grand tel que:::**
- Soit  $M = (Q; \Sigma; q_0; F; )$  un **automate ni deterministe** avec fonction de transition  $\delta : \Sigma \times Q \rightarrow Q$ .
- Considérez  $f : 2^{(Q \times Q)} \rightarrow 2^{(Q \times Q)}$  défini par  $(q; q') \in f(R)$  si
  1.  $q \in F$  si et seulement si  $q' \in F$ .
  2.  $\forall a \in \Sigma \ ( (a; q); (a; q')) \in R$ .
- Montrez que  $f$  est **monotone**. Soit  $R$  l'ensemble co-inductif associé. Avez-vous déjà rencontré la relation  $R$  dans l'étude des automates finis ?



## Exercice (de nition co-inductive (2/2))

- Soit  $(S; \rightarrow)$  un ensemble d'états et une relation de transition  $\rightarrow \subseteq S \times S$ .
- On définit  $D$  comme le plus grand sous-ensemble de  $S$  tel que si  $s \in D$  alors :
  1.  $\exists s' \in S \ s \rightarrow s'$ ,
  2.  $\forall s' \ s \rightarrow s'$  implique  $s' \in D$ .
- Expliquez pourquoi il s'agit d'une bonne de nition.
- Calculez ensuite  $D$  si  $S_0 = \{1; 2; 3; 4\}$  avec transitions
$$1 \rightarrow 2; 3; 4 \quad 3 \rightarrow 1 \quad 4 \rightarrow 4$$
- Définissez  $D'$  comme le plus petit sous-ensemble de  $S_0$  qui satisfait les conditions ci-dessus et calculez-le.
- Avons-nous déjà utilisé ce type de définition pour le langage Imp ?

## Sommaire

- Une définition (co-)inductive est bien définie si la fonction associée est **monotone**.
- L'ensemble défini correspond à un plus petit (plus grand) **point xe**.

# Analyse de vivacité

## Exemple

Considérons ce segment de programme écrit dans un langage ERTL **imaginaire** (par simplicité on utilise un code **deja linearise**).

```
0 :   $c := M[add_c]$       (on lit de la memoire)
1 :   $a := 0$ 
2 :   $b := a + 1$ 
3 :   $c := c + b$ 
4 :   $a := b * 2$ 
5 :  if  $a < N$  then goto 2
6 :  return( $c$ )
```

Ce programme utilise 3 **pseudo-registres**  $a; b; c$  mais il se trouve qu'il est possible de l'exécuter en utilisant seulement 2 **registres**.

Par **quelle analyse** le compilateur peut-il arriver à cette conclusion ?

## Etape 1 : analyse de la vivacite des variables

- On considère le **CFG** associé au programme.
- On dit qu'un pseudo-registre (variable)  $a$  est **vivace** sur l'arête  $(i;j)$  s'il y a un chemin qui commence à l'arête  $(i;j)$  et qui mène à un noeud  $k$  qui a besoin de la valeur de  $a$  à l'arête  $(i;j)$ .
- En particulier, cela suppose que la variable  $a$  **n'est pas rede nie** le long du chemin.
- Cette définition est un cas particulier de **de nition inductive** et peut être effectivement calculée par une **iteration nie**.

- Le **resultat de l'analyse de vivacite** est résumé dans le tableau suivant :

Arêtes	<i>a</i>	<i>b</i>	<i>c</i>
(0, 1)	<i>N</i>	<i>N</i>	<i>Y</i>
(1, 2)	<i>Y</i>	<i>N</i>	<i>Y</i>
(2, 3)	<i>N</i>	<i>Y</i>	<i>Y</i>
(3, 4)	<i>N</i>	<i>Y</i>	<i>Y</i>
(4, 5)	<i>Y</i>	<i>N</i>	<i>Y</i>
(5, 6)	<i>N</i>	<i>N</i>	<i>Y</i>
(5, 2)	<i>Y</i>	<i>N</i>	<i>Y</i>

## Etape 2 : coloration du graphe d'interference

Au tableau on peut associer un graphe où :

- les **noeuds** sont les **variables**,
- il y a une **arête** entre les variables  $x$  et  $y$  ssi il y a une **ligne du tableau** où les **variables** sont **vivaces** (on dit qu'elles **interferent**).

Ensuite il s'agit de **colorier le graphe** avec un certain nombre de couleurs de façon à que deux noeuds adjacents aient toujours deux couleurs différents.

On appelle **nombre chromatique** le nombre minimum de couleurs nécessaires (dans notre cas il en faut 2).

### Etape 3 : allocation de registres et compilation

On sait maintenant qu'on peut exécuter notre (segment de) programme avec 2 registres seulement.

Par exemple, on peut affecter le registre  $R_1$  à la variable  $c$  et le registre  $R_2$  aux variables  $a; b$ . On obtient alors :

```
0 :  $R_1 := M[add_c]$   
1 :  $R_2 := 0$   
2 :  $R_2 := R_2 + 1$   
3 :  $R_1 := R_1 + R_2$   
4 :  $R_2 := R_2 * 2$   
5 : if  $R_2 < N$  then goto 2  
6 : return( $R_1$ )
```



### Remarque

- Il y a des **algorithmes efficaces** pour effectuer l'analyse de vivacité.
- Le problème de la **coloration de graphes** est **NP-complet**. Néanmoins il y a des **heuristiques** qui marchent bien dans la pratique de la compilation.
- Si le **nombre chromatique** trouvé est **supérieur** au **nombre de registres** de la machine on modifie le programme de façon à garder en mémoire certaines variables (**spill**) et on répète l'analyse. En pratique ce processus **converge vite**.

## Analyse de vivacite

- On effectue l'analyse sur un **CFG**  $G = (N; E)$ .
- Pour tout noeud  $n \in N$ , on définit **deux ensembles de variables** :
  - $Def(n)$  L'ensemble des variables **a ectees** au noeud  $n$ .
  - $Uses(n)$  L'ensemble des variables **lues** au noeud  $n$ .
- On rappelle qu'une variable  $a$  est **vivace** à l'arête  $(n; n')$  s'il y a un chemin dans  $G$  qui commence par  $(n; n')$ , qui mène à un noeud  $n''$  qui lit  $a$  et tel que  $a$  n'est pas définie le long du chemin.
- Pour chaque arête  $(n; n')$  on veut définir  $Live(n; n')$  comme l'**ensemble de variables vivaces** à l'arête  $(n; n')$ .

## Définition formelle vivacité

- On définit  $Live(n; n')$  pour  $(n; n') \in E$  comme **la plus petite** famille d'ensembles tel que :

$$Live(n; n') = Uses(n') \cup ((\bigcup_{n'' \in suc(n')} Live(n'; n'')) \setminus Def(n'))$$

où :  $suc(n') = \{n'' \mid (n'; n'') \in E\}$ .

- Par la théorie du point fixe on dérive un **algorithme itératif** pour calculer  $Live(n; n')$  :

$$Live_0(n; n') = \emptyset$$

$$Live_{k+1}(n; n') = Uses(n') \cup ((\bigcup_{n'' \in suc(n')} Live_k(n'; n'')) \setminus Def(n'))$$

$$Live(n; n') = \bigcup_{k \geq 0} Live_k(n; n')$$

### Exercice (sur la bonne définition de *Live*)

- Soient  $E$  un ensemble fini d'arêtes et  $V$  un ensemble fini de variables.
- Soit  $(E \rightarrow 2^V)$  l'ensemble des fonctions (totales) des arêtes aux parties de  $V$ .
- On peut voir  $(E \rightarrow 2^V)$  comme un **ordre partiel** en posant :
$$L \leq L' \text{ ssi } \forall (n; n') \in E \quad L(n; n') \subseteq L'(n; n') :$$
- Montrez que  $(E \rightarrow 2^V)$  est un **treillis complet**.
- Montrez que toute suite  $\{L_i\}_{i \geq 0}$  telle que  $L_i \leq L_{i+1}$  pour  $i \geq 0$  est **bornée**.

- On définit  $f : (E \rightarrow 2^V) \rightarrow (E \rightarrow 2^V)$  par

$$f(L)(n; n') = \text{Uses}(n') \cup \left( \left( \bigcup_{n'' \in \text{succ}(n')} L(n'; n'') \right) \setminus \text{Def}(n') \right)$$

- Vérifiez que  $f$  est **monotone** (et **continue**).
- Conclure que *Live* est le **plus petit point fixe** de  $f$  et que ce point fixe est atteint après un **nombre fini d'iterations**.
- Trouvez un exemple de programme qui montre que prendre le plus petit point fixe **n'est pas équivalent** à prendre le plus grand.
- Proposez une définition de *Live* qui est indexée sur les **noeuds** plutôt que sur les arêtes du programme.

## Vers un calcul efficace de la vivacité

- Le calcul de la vivacité peut être **specialise pour chaque variable**.
- Dans ce cas, on obtient une **equation booleenne** :

$$\begin{aligned}
 a \in Live(n, n') \\
 \Leftrightarrow \\
 (a \in Uses(n')) \vee \\
 ((\bigvee_{n'' \in suc(n')} (a \in Live(n', n'')) \wedge (a \notin Def(n')))
 \end{aligned}$$

- On utilise les **abbreviations** suivantes :

$$\begin{array}{lll}
 a \in Live(n, n') & \equiv & x_{n, n'} \quad (\text{variable booleenne}) \\
 a \in Uses(n) & \equiv & u_n \quad (\text{constante booleenne}) \\
 a \notin Def(n) & \equiv & nd_n \quad (\text{constante booleenne})
 \end{array}$$

- Dans ce cas, l'**equation logique** devient :

$$x_{n,n'} = u_{n'} \vee \left( \left( \bigvee_{n'' \in \text{suc}(n')} x_{n',n''} \right) \wedge nd_{n'} \right)$$

- On remarquera que dans ces équations il n'y a **pas de negation logique** (sinon on ne serait pas sûr de trouver une solution:::)
- On peut toujours récrire de tels systèmes en **forme canonique** avec des équations de la forme :

$$x = \bigwedge_{i \in I} x_i \quad x = \bigvee_{i \in I} x_i$$

où en particulier  $1 = \bigwedge \emptyset$  et  $0 = \bigvee \emptyset$

**NB** Chaque variable **paraît exactement une fois** à gauche d'une équation.

## Solution de systemes d'equations booleennes monotones

- Étant donné un système d'équations en forme canonique, on construit un **graphe dirige etiquete** où :
  - Les **variables** sont les **noeuds** du graphe.
  - Le noeud  $x$  est **etiquete** avec  $\wedge$  (ou  $\vee$ ) si l'équation associée à  $x$  est  $x = \bigwedge_{i \in I} x_i$  (ou  $x = \bigvee_{i \in I} x_i$ ).
  - De plus, dans ce cas on introduit une **arête** de  $x$  à  $x_i$ .

**NB** On sait que  $x$  est 1 si l'équation a la forme  $x = \bigwedge \emptyset$ . L'idée de l'algorithme est de visiter le graphe et de **propager les 1**.



## Une mise en oeuvre

- On maintient un vecteur  $st : V \rightarrow \mathbf{Z}$  (**strength**). Intuitivement  $st(x)$  est le nombre de successeurs de  $x$  qui doivent devenir 1 pour que  $x$  soit forcé à être 1 aussi.
- Initialement, on pose :

$$st(x) = \begin{cases} ]suc(x) & \text{if } lab(x) = \wedge \\ 1 & \text{if } lab(x) = \vee \end{cases} :$$

- Le vecteur  $st : V \rightarrow \mathbf{Z}$  induit un **vecteur booleen**  $\hat{st} : V \rightarrow \{0;1\}$  comme suit :

$$\hat{st}(x) = \begin{cases} 1 & \text{if } st(x) \leq 0 \\ 0 & \text{if } st(x) > 0 \end{cases}$$

- Au début  $A$  est l'ensemble des 1 a propager :

$$A = \{x \mid st(x) = 0\}$$

- Ensuite l'algorithme de propagation est le suivant :

```
while  $A \neq \emptyset$  do
  begin
    choose  $x \in A$ ;
     $A := A \setminus \{x\}$ ;
     $\forall w \in \text{pred}(x)$  do      (les predecesseurs)
      begin
         $st(w) := st(w) - 1$ ;
        if  $st(w) = 0$  then  $A := A \cup \{w\}$ 
      end
    end
  end
return( $\hat{st}$ )
```

- Le résultat  $\hat{st}$  est la plus petite solution.

### Exercice (sur l'algorithme de propagation)

- Appliquez l'algorithme au graphe associé au système :

$$x_1 = x_1 \wedge x_2; \quad x_2 = x_1 \vee x_2 \vee x_3; \quad x_3 = 1 ;$$

- Montrez que l'algorithme a une complexité en temps **lineaire** dans le **nombre d'arêtes** du graphe.
- Dérivez un borne supérieur pour la **complexite du calcul de la vivacite**.
- Adaptez l'algorithme pour qu'il calcule **la plus grande solution**.

## Variations sur l'analyse du **coût** de données

- Le calcul de vivacité est un exemple parmi d'autres d'**analyse de coût** de données.
- L'idée de base est d'analyser les dépendances entre **définitions** et **usages**.
- Nombreuses **variations** sont possibles. Nous en considérons deux dans la suite.

## Propagation de constantes

- Fixons une variable  $x$ . Pour chaque noeud  $n$  on détermine l'**ensemble**  $R(n)$  des noeuds  $n'$  tel que  $x \in Def(n')$  et il y a un chemin de  $n'$  à  $n$  où  $x$  n'est pas redéfini.
- Si  $n' \in R(n)$  et  $x \in Uses(n)$  alors la valeur de  $x$  est **susceptible** d'être celle définie dans  $n'$ .
- En particulier, si (i)  $R(n) = \{n'\}$  et (ii) l'opération associée au noeud  $n'$  est  $x := cst$  (**affectation d'une constante**) alors on sait que la valeur de  $x$  dans  $n$  sera  $cst$ .
- On peut donc envisager de **propager la constante**, à savoir remplacer l'occurrence de  $x$  dans  $n$  par  $cst$ .

## Elimination de code mort

- Supposons que le noeud  $n$  consiste en une **affectation**  $x := e$ .
- Si la valeur de  $x$  au noeud  $n$  n'est pas utilisée dans un noeud accessible depuis  $n$  alors on peut considérer que l'affectation est inutile ; on dit qu'il s'agit de **code mort**.
- Dans ce cas, on peut simplement **supprimer l'instruction** (à noter qu'il s'agit d'une optimisation qui peut éliminer des erreurs !)

## Exercice

Formaliser les analyses de flot de données qui permettent de mettre en oeuvre les techniques de **propagation de constantes** et d'**elimination de code mort** qu'on vient de décrire.

## Exercice (dominance)

La notion de **dominance** sert, par exemple, à identifier les boucles dans un CFG.

- Soit  $G = (N; A; r)$  un **graphe dirige** avec noeuds  $N$ , arêtes  $A$  et un noeud racine  $r \in N$ .
- On dit que le noeud  $n$  **domine** le noeud  $m$  et on écrit  $D(n; m)$  si tout chemin dirigé (y compris le chemin de longueur 0) de  $r$  à  $m$  passe par  $n$ .
- Il **suit** de cette définition que pour tout  $n \in N$ ,  $D(r; n)$  et  $D(n; n)$ . En plus,  $D(n; r)$  seulement si  $n = r$ .



1. Montrez que la relation  $D$  est **transitive**.
2. **Calculez**  $D(n; m)$  pour le graphe  $G = (\{1; \dots; 7\}; A; 1)$  avec arêtes  $A$  définies par :

$$A = \{ \begin{array}{lll} (1;2);(1;3) & (2;4) & (3;4) \\ (4;5) & (5;6);(5;7) & (6;1) \end{array} \} :$$

On présentera le résultat par une matrice  $7 \times 7$  à valeurs en  $\{0;1\}$  où  $D(n; m) = 1$  si et seulement si  $n$  domine  $m$ .

3. Soit  $Pred(n) = \{p \mid (p; n) \in A\}$ . **Montrez** que si  $n \neq m$ ,  $n \neq r$  et  $m \neq r$  alors

$$D(n; m) \text{ si et seulement si } \forall p \in Pred(m) \ D(n; p) :$$

4. Donnez un algorithme **iteratif** qui calcule une suite de matrices  $D_0; D_1; \dots$  qui converge à  $D$ . Faut-il prendre le plus petit ou le plus grand point fixe ?
5. Calculez le résultat de votre algorithme pour le **graphe**  $G = (\{1; \dots; 4\}; A; 1)$  avec arêtes  $A$  définies par  $A = \{(1;2); (2;3); (3;2); (3;4); (4;2); (4;3)\}$ . Cette fois on présentera le résultat par une matrice  $4 \times 4$  à valeurs en  $\{0;1\}$  où  $D(n;m) = 1$  si et seulement si  $n$  domine  $m$ .

6. On suppose maintenant que le graphe en question est **connecte** (chaque noeud est accessible à partir de la racine). Montrez que  $D(n; m)$  et  $D(n'; m)$  implique  $D(n; n')$  ou  $D(n'; n)$ .
7. Dérivez que chaque noeud  $m$  différent de la racine a un unique **dominateur immédiat**, à savoir il existe un noeud **unique**  $n$  tel que (i)  $n \neq m$ , (ii)  $D(n; m)$  et (iii) si  $D(n'; m)$  alors  $D(n'; n)$ .
8. Conclure qu'on peut associer à chaque graphe  $G$  un **arbre de domination** avec les mêmes noeuds que  $G$  et tel que il y a une arête du noeud  $n$  au noeud  $n'$  ssi  $n$  est le dominateur immédiat de  $n'$ .
9. Construire l'arbre de domination associé au **graphe**  $G$  suivant :

$1 \rightarrow 2;$      $2 \rightarrow 3, 4;$      $3 \rightarrow;$      $4 \rightarrow 2, 5, 6;$      $5 \rightarrow 7, 8;$      $6 \rightarrow 7;$

$7 \rightarrow 11;$      $8 \rightarrow 9;$      $9 \rightarrow 8, 10;$      $10 \rightarrow 5, 12;$      $11 \rightarrow 12$

### Lecture conseillée

Appel. Modern compiler implementation in ML (C, Java). Chapitre 10.

On pourra aussi lire la section 17.2 pour d'autres exemples d'analyse de flot de données et la section 18.1 pour plus d'informations sur la notion de dominance.

# Coloration du graphe d'interférence et Allocation de registres

## Graphe d'interference

On rappelle que le **graphe d'interference** est dérivé de l'analyse de vivacité du programme comme suit :

- Les **noeuds** sont les **variables** du programme.
- Deux noeuds (variables)  $x; y$  sont connectées par une **arête** (non-dirigée) ssi il y a une arête  $(n; n')$  du CFG du programme telle que  $x; y \in \text{Live}(n; n')$ .

## Coloration et nombre chromatique d'un graphe

- Soit  $G = (N; E)$  un **graphe non-dirigé**.
- Une **coloration** de  $G$  par  $n$  couleurs est une fonction  $c : N \rightarrow \{1; \dots; n\}$  telle que si les noeuds  $a$  et  $b$  sont adjacents alors  $c(a) \neq c(b)$ .
- Le **nombre chromatique** d'un graphe  $G$  est le plus petit nombre de couleurs qui permet de colorier  $G$  (notez que ce nombre est au plus égal au nombre de noeuds du graphe).

- Le problème de savoir si un graphe admet une coloration avec  $n$  couleurs est NP-COMPLET si  $n \geq 3$  (et dans PTIME si  $n = 1$  ou  $n = 2$ ).
- Un théorème célèbre (démontré pour la première fois par Appel et Haken en 1977 à l'aide d'un ordinateur) dit qu'un **graphe planaire** peut toujours être colorié avec 4 **couleurs**.



## Exercice (2-coloration)

Le but est de trouver un algorithme efficace pour vérifier si un graphe  $G$  non-dirigé a nombre chromatique égale à 2.

1. Quels sont les graphes avec nombre chromatique égal à 1 ?
2. Montrez que si  $G$  admet une 2-coloration alors  $G$  est un **graphe biparti**.
3. Montrez que si  $G$  admet une 2-coloration alors tous ses **cycles ont longueur pair**.
4. Montrez que si tous les cycles de  $G$  ont une longueur pair alors  $G$  admet une **2-coloration**.
5. Donnez un algorithme **polynomial** en temps qui vérifie si tous les cycles d'un graphe ont longueur pair.

### Exercice (graphes planaires)

1. Donnez un exemple de graphe **non-planaire**.
2. Donnez un exemple de graphe planaire qui ne peut pas être colorié avec **3 couleurs** (Appel et Haken ont montré en 1977 qu'un graphe planaire peut toujours être colorié avec 4 couleurs).

### Exercice (decomposition)

1. Montrez qu'un graphe  $G$  est  $k$ -coloriable si et seulement si  $G$  privé d'un sommet de degré strictement inférieur à  $k$  est  $k$ -coloriable.
2. Donnez un **exemple** d'un graphe  $G$  dont tous les sommets ont degré au moins  $k$  mais qui est néanmoins  $k$ -coloriable.

## Une heuristique

Soient  $P$  un programme à la ERTL et  $k$  le nombre de registres disponibles. L'heuristique se décompose en 4 phases :

Construction, Simplification, Sélection et Répétition.

**Construction** Cette phase se décompose en :

- Analyse de vivacité.
- Construction du graphe d'interférence.

Ensuite on va à **Simplification**.

**Simplification** On initialise une **pile**  $S$  vide. On cherche à colorier le graphe d'interférence avec  $k$  couleurs.

1. Si le nombre de pseudo-registres est inférieur égal à  $k$  on **Termine** (solution triviale).
2. Si le graphe est vide on va a **Selection**.
3. Sinon, on sélectionne un noeud  $v$  avec moins que  $k$  noeuds adjacents. On insère  $v$  dans  $S$ . On enlève  $v$  et les arêtes du graphe. On **itere Simplification**.
4. Sinon, on sélectionne un noeud  $v$ . On insère  $v$  dans  $S$  en le **marquant**. On enlève  $v$  et les arêtes du graphe. On **itere Simplification**.

**Selection** On reconstruit le graphe en dépilant les noeuds de la pile  $S$  (dans l'ordre inverse de leur insertion).

- Si la pile  $S$  est vide on **termine**.
- Si le noeud extrait **n'est pas marque** alors il a au plus  $k - 1$  noeud adjacents et donc on peut le colorier (par l'exercice sur la  $k$ -coloration). On **itere Selection**.
- Si le noeud est **marque** alors soit il est coloriable et dans ce cas on **itere Selection** soit il ne l'est pas et dans ce cas on **va a Repetition**.

**Repetition** On sélectionne une (ou plusieurs) variable  $v$  marquée dans la phase de Simplification et on lui affecte une adresse en mémoire  $add_v$ . On modifie le programme de façon à effectuer :

- un transfert mémoire  $\rightarrow$  pseudo-registre juste avant sa lecture et
- un transfert pseudo-registre  $\rightarrow$  mémoire juste après son écriture.

**Va a Construction**

## Exercice (application de l'heuristique)

On considère le programme :

1	$j := M[add_j]$	9	$c := e + 8$
2	$k := M[add_k]$	10	$d := c$
3	$g := M[j + 12]$	11	$k := m + 4$
4	$h := k - 1$	12	$j := b$
5	$f := g * h$	13	$M[add_d] := d$
6	$e := M[j + 8]$	14	$M[add_k] := k$
7	$m := M[j + 16]$	15	$M[add_j] := j$
8	$b := M[f]$		



1. Pour chaque arête du programme, calculez la vivacité des variables  $b, c, d, e, f, g, h, k, m$ .
2. Calculez le graphe d'interférence.
3. Appliquez l'heuristique présentée avec 4 couleurs.
4. Répétez avec 3 couleurs. Ceci devrait échouer. Essayez de modifier le programme de façon telle que l'heuristique puisse réussir.

## References

On trouve **nombreuses variantes** de cette heuristique dans la littérature. Entre autres, elles se distinguent par :

- le **choix du pseudo-registre** à ‘spiller’.
- le **choix de la couleur**.
- le traitement de **certaines instructions** comme **move** (on peut supprimer les instructions de la forme  $x := y$  si on affecte à  $x$  et  $y$  le même registre).

Lire le chapitre 11 de APPEL pour plus d’informations.

## Exercice (allocation de registres pour expressions)

- L'allocation de registres est beaucoup plus simple si l'on considère le code associé à l'**évaluation d'expressions**.
- Considérons des expressions de la **forme suivante** (*id* sont des pseudo-registres) :

$$e ::= id \parallel op(e; e)$$

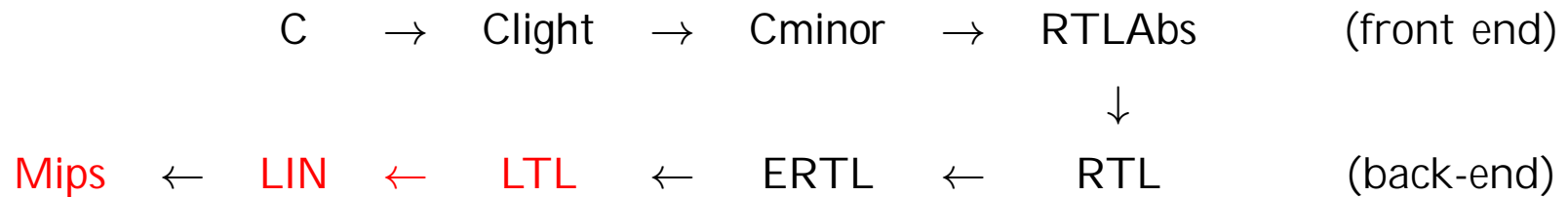
- Notre objectif est de compiler une expression comme une suite d'affectations élémentaires en utilisant **le moins de registres possible**.

- Par exemple, l'expression  $x + (y * z)$  pourrait être **compilée** comme suit (le résultat étant en  $r1$ ) :

$$r1 := M[add_x]; r2 := M[add_y]; r3 := M[add_z];$$
$$r2 := r2 * r3; r1 := r1 + r2;$$

- Proposez une fonction de **compilation** qui effectue cette tâche.
- Votre approche produit-elle toujours une **solution optimale** par rapport au nombre de registres utilisés ?

## Back-end : de LTL à LIN à Mips



## Plan du cours

- Linéarisation de code.
- Génération de code assembleur Mips

**sum en LTL (rappel)**

```

procedure sum(1)
var 8
entry sum30
sum30: newframe          --> sum29
sum29: lw    $ra, 4($sp)  --> sum28    (sauve adresse retour)
sum28: j      --> sum27    (suite de 'nop')
...   : j      --> ...
sum23: j      --> sum22
sum22: lw    $s0, 0($sp)  --> sum21    (sauve $s0)
sum21: move  $s0, $a0     --> sum20
sum20: j      --> sum19    (suite de 'nop')
...   : j      --> ...
sum11: j      --> sum10
sum10: li    $v0, 0       --> sum9
sum9  : seq  $v0, $s0, $v0 --> sum8
sum8  : beq  $v0, $zero    --> sum4, sum6
sum6  : li   $v0, 0        --> sum0    (résultat=0)
sum4  : li   $v0, 1        --> sum3    (cas récursif)
sum3  : sub  $a0, $s0, $v0 --> sum2

```

```
sum2 : la    $v0, sum    --> sum13
sum13: j      --> sum44    (encore des 'nop')
sum44: j      --> sum43
sum43: call   $v0        --> sum42
sum42: j      --> sum1
sum1 : add    $v0, $s0, $v0 --> sum0
sum0 : j      --> sum41    (suite de retour)
sum41: j      --> sum40
sum40: lw     $ra, 4($sp) --> sum39
sum39: j      --> sum38    (suite de 'nop')
...   : j      --> ...
sum34: j      --> sum33
sum33: lw     $s0, 0($sp) --> sum32
sum32: delframe          --> sum31
sum31: jr     $ra
```



## Compilation de LTL a LIN

- Le CFG est remplacé par une **suite lineaire d'instructions**.
- Le successeur de chaque instruction est **implicite** (sauf pour le branchement).
- On garde les **etiquettes** seulement pour les instructions cibles d'un branchement.

## Exercice (sur la linearisation)

- On suppose que le CFG comprend 4 **types d'instructions** :

$\ell : \text{op} \rightarrow \ell'$	(operation ou appel)
$\ell : j \rightarrow \ell'$	(saut non-conditionne)
$\ell : \text{bj} \rightarrow \ell', \ell''$	(saut conditionne)
$\ell : \text{ret}$	(retour)

- D'autre part les **instructions linearisees** ont la forme (l'étiquette à gauche de ' :' est optionnelle) :

$\ell : \text{op}$	(operation ou appel)
$\ell : j \rightarrow \ell'$	(saut non-conditionne)
$\ell : \text{bj} \rightarrow \ell'$	(saut conditionne)
$\ell : \text{ret}$	(retour)

- A partir d'une **enumeration**  $\ell_1 :: \dots :: \ell_n$  des noeuds, proposez une fonction  $\mathcal{C}$  qui étant donné le noeud  $\ell_i$  et le code linéarisé pour  $\ell_{i+1} :: \dots :: \ell_n$  produit le code linéarisé pour  $\ell_i :: \dots :: \ell_n$ .
- Un **critere de qualite** est que le code linéarisé ne contient pas un saut non-conditionné à l'instruction suivante.
- Appliquez votre algorithme au CFG :

$\ell_1 : \quad j \rightarrow \ell_2$

$\ell_2 : \quad \mathbf{bj} \rightarrow \ell_3, \ell_4$

$\ell_3 : \quad \mathbf{op} \rightarrow \ell_5$

$\ell_4 : \quad \mathbf{op} \rightarrow \ell_5$

$\ell_5 : \quad \mathbf{ret}$

## sum en LIN (rappel)

```
procedure sum(1)
var 8
sum30:
newframe
sw    $ra, 4($sp)      (sauve $ra)
sw    $s0, 0($sp)      (sauve $s0)
move  $s0, $a0         (test)
li    $v0, 0
seq   $v0, $s0, $v0
beq   $v0, $zero, sum5
li    $v0, 0           (résultat = 0)
sum40:                 (suite de retour)
lw    $ra, 4($sp)
lw    $s0, 0($sp)
delframe
jr    $ra
sum5:                  (cas récursif)
li    $v0, 1
sub   $a0, $s0, $v0
```

```
la    $v0, sum
call  $v0
add   $v0, $s0, $v0
j     sum40
```

## Compilation de LIN a Mips

- La **gestion des blocs d'activation** est réalisée par un incrément/décrément du registre *sp*.
- L'**accès a la pile** se fait par un décalage par rapport à *sp*.
- La notion de **fonction disparaît**.

## sum en Mips (rappel)

```
.align 2                                (mots allignes sur multiples de 4)
sum:
addi    $sp, $sp, -8                    (traduction newframe)
sw      $ra, 4($sp)
sw      $s0, 0($sp)
move    $s0, $a0
li      $v0, 0
seq     $v0, $s0, $v0
beq     $v0, $0, sum5
li      $v0, 0
sum40:                                    (suite retour)
lw      $ra, 4($sp)
lw      $s0, 0($sp)
addi    $sp, $sp, 8                    (traduction delframe)
jr      $ra
sum5:                                    (cas récursif)
li      $v0, 1
sub     $a0, $s0, $v0
la      $v0, sum
```

```
jalr    $v0  
add     $v0, $s0, $v0  
j       sum40
```



## Exercice (petits sauts et grands sauts)

- On considère un raffinement de la machine  $V_m$  (sans étiquettes!) dans laquelle les instructions de branchement  $\text{branch}(k)$  et  $\text{bge}(k)$  prennent **deux adresses consecutives** si le offset  $k$  est plus grand en valeur absolue qu'une certaine valeur  $b > 0$ .
- On appelle cela une **expansion** d'une instruction de branchement. Notez que l'expansion d'une instruction de branchement peut en causer d'autres **en cascade**.
- Le but est de concevoir et analyser un **algorithme** qui étant donné un programme  $C$  avec  $n$  instructions  $C[1]; \dots; C[n]$  :
  1. détermine quelles instructions de branchement doivent être **expansees** et
  2. **recalcule les o sets** des instructions de branchement.

- Soit  $x_i \in \{1;2\}$ , une variable qui dénote le nombre d'adresses associées à l'instruction  $C[i]$ ,  $i \in \{1;:::;n\}$ .
- Soit  $of_i$  l'offset associé à l'instruction  $i$  qui est défini comme suit :

$$of_i = \begin{cases} k & \text{si } C[i] = (\text{branch } k) \text{ ou } C[i] = (\text{bge } k) \\ 0 & \text{autrement} \end{cases}$$

1. Décrivez les **contraintes** (inégalités) que les variables  $x_i$  doivent satisfaire en fonction de  $of_i$ .

Une **solution** au système de contraintes est un vecteur  $(x_1; \dots; x_n) \in \{1; 2\}^n$  qui satisfait les contraintes.

Une solution est **optimale** si la valeur  $\sum_{i=1, \dots, n} x_i$  est la plus petite possible, c'est à dire si la longueur du code est minimum.

2. Montrez que votre système de contraintes a **toujours une solution** (laquelle?) et qu'il est possibles d'ordonner les vecteurs  $\{1;2\}^n$  de façon à que le calcul de la solution optimale se réduit au calcul d'un **plus petit point**  $\mathbf{x}_e$ .

En particulier, donnez :

- la valeur initiale de l'itération,
- le pas d'itération,
- l'argument qui montre que l'itération termine et
- l'argument qui montre que quand l'itération termine on a atteint la solution optimale.

# Récupération automatique de la mémoire

## Plan

1. La gestion du tas.
2. Marquage et balayage.
3. Comptage de références.
4. Récupération par copie.
5. Compléments.

## Problematique

- Le code exécutable généré par un compilateur est un processus qui tourne (souvent) au dessus d'un système d'exploitation.
- Le processus dispose d'un certain **segment de memoire virtuelle** qui doit être géré de façon économique.
- Les machines virtuelles des langages de programmation courants (C, Java, ML,...) distinguent **trois zones de memoire** :
  - statique** contient le code, les variables globales, les tampons d'entrée-sortie,...
  - pile** (ou **stack**) contient la pile des blocs d'activation des procédures (ou **frames**), le contexte d'évaluation.
  - tas** (ou **heap**) contient des données dont la durée de vie n'est **pas previsible** (facilement).

## La gestion de la pile

C'est simple :

- un pointeur au sommet de la pile (plus éventuellement 1, 2 pointeurs à la base de la pile et à la base du bloc d'activation),
- pour allouer un bloc de  $b$  cellules on incrémente le compteur de  $b$  en vérifiant qu'il n'y a pas de débordement,
- pour enlever un bloc de  $b$  cellules on décrémente le pointeur de  $b$ .



## La gestion du tas

Problème : quand peut-on récupérer la mémoire ?

**Option 1** Pas de récupération : évidemment **correct**, mais **danger de saturation** de la mémoire.

**Option 2** A la charge du programmeur (cf. C) : **incomplet** (on oublie de récupérer) et **incorrect** (on récupère avant l'heure).

**Option 3** A la charge d'un analyseur statique (cf. ML-Kit) : **correct** et **coût prédictible**. Par contre, l'efficacité de la récupération est assez **difficile à prévoir**.

**Option 4** La machine virtuelle appelle un programme dit ramasse-miettes (ou *garbage collector*) pour récupérer les cellules inaccessibles (cf ML, Java, :::). Algorithme certifié **correct** et éventuellement **complet**.

**NB** Dans la suite on se focalise sur l'option 4. On revient sur l'option 3 plus tard.

## Ramasse miettes (garbage collector)

Il convient d'abstraire le problème :

Mémoire	=	Graphe dirigé avec racines
Cellules de la mémoire	=	Noeuds du graphe
Pointeurs	=	Arêtes dirigées
Noeuds pointés par zone statique et pile	=	Racines du graphe

**Définition** Une cellule dans le tas est *recupérable* si elle n'est pas accessible à partir des racines.

### Remarque (sur l'accessibilite)

- La notion d'**accessibilite** de la mémoire est une sur-approximation de la réalité.
- A comparer, par exemple, avec la notion d'accessibilité dans le **graphe du flot de contrôle**.
- Il est possible qu'une instruction accessible dans le graphe du flot de contrôle ne soit **jamais executee**.
- De la même façon, il est possible qu'une cellule accessible dans le graphe de la mémoire ne soit **jamais visitee**.

## Principe de fonctionnement

1. Au début, toutes les cellules libres du tas sont connectées dans une liste (dite **liste libre**).
2. Quand une nouvelle cellule est nécessaire, on extrait un élément de la **liste libre**.
3. Quand on a plus de cellules dans la **liste libre** on appelle un programme **ramasse miettes** (*garbage collector*) pour vérifier si une partie de la mémoire du tas peut être récupérée et insérée à nouveau dans la liste libre.

**NB** Dans la suite on suppose que toutes les cellules ont la **même taille**. En général il faut considérer l'allocation de cellules de **taille variable** (par exemple pour l'allocation de tableaux).

## Marquage et balayage (mark and sweep)

On suppose que toutes les cellules comprennent un **bit de marquage** qui est initialement à 0. La méthode fonctionne en deux phases :

**Marquage** On visite le graphe en commençant par les racines et on met à 1 les bits de marquage de toutes les cellules accessibles.

**Balayage** On va parcourir toutes les cellules du tas et pour chaque cellule on effectue les opérations suivantes :

- Si le bit de marquage est à 0 alors on insère la cellule dans la liste libre.
- Si le bit de marquage est à 1 on le remet à 0.

## Marquage = Parcours du graphe

- La phase de marquage est normalement effectuée par une visite en **profondeur d'abord** du graphe.
- On appelle la procédure  $DF(v)$  (*Depth First*) sur chaque racine du graphe.
- Dans la suite on cherchera à programmer **sans appels recursifs**, car pour mettre en oeuvre la récursion on a besoin de mémoire:::
- Par exemple, la procédure  $DF$  suivante manipule explicitement une pile pour garder une trace des noeuds à visiter.

*Init* :  $sp := nil$ ;

**procedure**  $DF(v)$

**if**  $v$  points to heap and  $v.mark = 0$  **then**

**begin**

$push(v, sp)$ ;

**while**  $sp \neq nil$  **do**

**begin**

$v := pop(sp)$ ;

$v.mark := 1$ ;

$\forall w(w \text{ pointer in cell } v \text{ and } w.mark = 0)$  **do**

$push(w, sp)$ ;

**end**

**end**

### Exercice (facile)

Comment peut-on modifier les structures de données de cet algorithme pour qu'il visite le graphe *en largeur*?

Rappel : considerez un arbre binaire avec racine 1 dont les fils sont 2 et 3, et tel que les fils de 2 sont 4 et 5 et les fils de 3 sont 6 et 7. Dans une visite *en profondeur* (de gauche à droite) on visite les noeuds dans l'ordre 1,2,4,5,3,6,7 alors que dans une visite *en largeur* on visite les noeuds dans l'ordre 1,2,3,4,5,6,7.



## Implementation ramassage

On suppose que  $p$  pointe à la liste des cellules libres du tas.

*Init* :  $p := \text{'lower address of heap'}$ ;

```
while  $p < \text{'upper address of heap'}$  do  
  begin  
    if  $p.mark = 1$  then  $p.mark := 0$   
    else  $insert(p, fl)$ ;  
     $p := p + \text{'cell size'}$ ;  
  end
```

## Coût marquage et ramassage

- Soit  $R$  le nombre de cellules dans le tas qui sont accessibles à partir de la zone statique et de la pile.
- Soit  $H$  le nombre total de cellules disponibles dans le tas. Alors le coût est donné par :

$$c_1 R + c_2 H$$

où  $c_1, c_2$  sont des facteurs constants,  $c_1 R$  est le coût du marquage et  $c_2 H$  est le coût du ramassage du tas.

- Le **coût par cellule recuperee** est :

$$(c_1 R + c_2 H) / (H - R)$$

car  $(H - R)$  est exactement le nombre de cellules récupérées.

- Si  $H \approx R$  alors le coût est élevé et si  $H \gg R$  alors le coût s'approche de  $c_2$ .
- Donc il n'est pas très intéressant d'exécuter la méthode de ramasse miettes quand une grande partie du tas est accessible.
- Dans ce cas, la machine virtuelle passe son temps à essayer de récupérer un nombre réduit de cellules.
- Quand cette situation se vérifie, la machine virtuelle peut essayer d'obtenir de la mémoire virtuelle additionnelle du système d'exploitation.

## Comptage des references (reference counting)

- Un problème avec la méthode de marquage et ramassage est que son exécution provoque l'arrêt de l'exécution du programme pour **un temps proportionnel a la taille du tas**.
- Cet arrêt peut être inacceptable pour des programmes qui doivent respecter des **contraintes de `temps reel'**.
- La méthode du comptage des références règle **partiellement** ce problème.

## Ingredients du comptage des references

- Chaque cellule du tas comprend un **champ compteur** qui ‘compte’ le nombre de pointeurs à la cellule.
- Initialement le compteur est à 0.
- Pour chaque instruction, le **compilateur** génère un certain nombre d’instructions qui **maintiennent les compteurs à jour**.

**Exemple**

cell address	env	eld in cell	counter	eld in cell
$x$	$y$		$n_1$	
$y$	$nil$		$n_2$	
$p$	$nil$		$n_3$	

On suppose que le programme comprend l'instruction :

$$x:env := p$$

Le compilation doit générer la séquence suivante d'instructions où, comme dans la section précédente, on suppose que  $fl$  pointe à la liste de cellule libres dans le tas.

```
p.count := p.count + 1; x.env.counter := x.env.counter - 1;  
if x.env.counter = 0 then  
  begin  
    insert(x.env, fl);  
    recursively update counters of cells pointed by x.env  
  end  
x.env := p;
```

Maintenant la gestion du tas est **interlaccée** avec l'exécution du programme.

- On dit que la méthode du comptage de références est une méthode de ramasse miettes **incrementale**.
- Cependant, il faut noter que la méthode est **coûteuse** et **incomplete** (elle ne récupère pas toujours la mémoire disponible).
- Par exemple, considérez la configuration :

cell address	env	eld in cell	counter	eld in cell
$r$	$p$		1	
$p$	$x$		1	
$x$	$y$		2	
$y$	$x$		1	

avant d'exécuter le code associé à l'affectation  $r := nil$ .



**Exercice** Dans ce cas, le code généré peut-il récupérer toute la mémoire inaccessible ?

- Le point est que le comptage de références ne voit pas l'inaccessibilité de **structures avec cycles** et donc il peut ne pas récupérer des cellules qui ne sont plus accessibles.
- On remarquera aussi que la mise à jour récursive de compteurs est une opération qui dans le pire des cas peut être proportionnelle à la taille du tas, mais cette opération peut être **interlacee** avec l'exécution du programme.
- Donc même sans structures cycliques, il convient de voir le champ compteur d'une cellule comme une **borne superieure** au nombre de pointeurs à la cellule.
- En pratique, on utilise plutôt des 'méthodes incrémentales' basées sur des **algorithmes concurrents**.

## Recuperation par copie (copying collection)

En plus de la non-incrémentalité, la méthode de marquage et ramassage a deux problèmes additionnels :

1. On a besoin d'une pile pour le marquage dont la taille est bornée par le nombre de cellules du tas. Donc on a **besoin de beaucoup de memoire** juste quand elle est épuisée (on verra plus tard qu'on peut se passer de la pile par une technique d'**inversion de pointeurs**).
2. La mémoire récupérée peut être de plus en plus **fragmentee** ce qui est un problème si on a besoin d'allouer des données de taille variable sur des blocs de cellules contiguës.

Ces deux problèmes sont réglés par la méthode de **recuperation par copie**.

## Ingredients de la recuperation par copie

- Le tas est maintenant divisé en deux moitiés composées de cellules contiguës :

from\_space    |    to\_space

- Initialement, on alloue dans la zone ‘from\_space’.
- Quand ‘from\_space’ est saturée, on **traverse la partie accessible** de ‘from\_space’ et on construit une **copie isomorphe** dans un segment initial de la zone ‘to\_space’

## Algorithme de copie

L'algorithme qui fait la copie isomorphe est la partie intéressante de la méthode.

- La première fois qu'on arrive à une cellule accessible de la zone 'from\_space' on copie son contenu dans la première cellule disponible dans la zone 'to\_space'.
- La cellule dans la zone 'from\_space' est alors **marquee** et un pointeur à sa copie dans la zone 'to\_space' est inséré. Le marquage est important pour éviter que la cellule soit recopiée plusieurs fois.

**NB** Pas de **fragmentation** dans la zone 'to\_space'.

## Coût de la recuperation par copie

- Une fois que la phase de copie est complétée, on ne procède *pas* à une phase de ramassage. En effet, il suffit d'**invertir** simplement le rôle de 'from\_space' et de 'to\_space' et de continuer l'exécution du programme.
- Ceci veut dire que si  $R$  est le nombre de cellules accessibles dans la zone 'from\_space' alors le coût de la méthode est  $cR$  pour une constante  $c$  et le coût par cellule récupérée est  $cR=((H=2) - R)$ .
- Si  $H \gg R$  alors le coût approche 0, mais en pratique  $R$  est plutôt proportionnel à  $H$ .

## Implementation de l'algorithme de copie

On décrit maintenant l'algorithme qui copie la partie accessible de 'from\_space' dans 'to\_space'.

- On suppose que chaque cellule contient un champ  $f1$ .
- Il peut s'agir d'un champ spécial ou du premier champ pointeur de la cellule s'il y en a un.
- On suppose que *next* et *scan* sont deux pointeurs qui pointent initialement à l'adresse de base de 'to\_space'.

- D’abord on doit définir une procédure *Fwd* qui va créer une copie d’une cellule dans *to\_space* s’il n’y en a pas déjà une.

*function Fwd(p) = case*

*p* points to *from\_space* and *p.f1* points to *to\_space* : *p.f1*

*p* points to *from\_space* and *p.f1* does not point to *to\_space* :

*copy(p, next);*

*p.f1 := next;*

*increment(next);*

*p.f1*

*else : p*



- Soit  $r$  la racine du graphe ‘from\_space’. On exécute :

$Fwd(r);$             (this increments  $next$ )

**while**  $scan < next$  **do**

**begin**

$\forall$  pointer  $f$  in the cell pointed by  $scan$  **do**

$scan.f := Fwd(scan.f);$

$increment(scan);$

**end**

- Un point intéressant de l'algorithme est qu'il n'utilise **pas de mémoire additionnelle** pour visiter le graphe dans 'from\_space' (ce qui n'était pas le cas pour la méthode de marquage et ramassage).
- La raison est que les éléments à visiter sont mémorisés dans la zone 'to\_space' entre les pointeurs *scan* et *next*.

## Exemple recuperation par copie

En exécutant la méthode de ramassage par copie sur l'exemple :

Adresse	Champ $f1$	Champ $f2$
7	9	11
9	7	9
11	9	7

et en supposant que la racine  $r$  est 7 et que l'adresse de base de 'to\_space' est 12 on produit la copie suivante dans la zone 'to\_space' :

Adresse	Champ $f1$	Champ $f2$
12	13	14
13	12	13
14	13	12

### Exercice (compactage d'un tableau)

- On dispose d'un tableau qui contient des **blocs de taille variable**. Si  $p$  est l'adresse du premier mot d'un bloc alors on dénote avec  $p:statut$  son statut qui peut être libre ou occupé et avec  $p:long$  sa longueur.
- Décrivez un **algorithme lineaire** dans la taille du tableau qui permet de compacter la mémoire, c'est-à-dire de faire en sorte que les blocs occupés sont contigus et précèdent un bloc libre.

- Voici un **exemple de tableau** avant et après compactage où  $X$  dénote des informations non significatives mémorisées dans les blocs libres et  $a; b; c; :::$  dénotent des informations significatives mémorisées dans les blocs occupés.

1 :	(libre, 2)	1 :	(occupe, 2)
2 :	$X$	2 :	$a$
3 :	(occupe, 2)	3 :	(occupe, 3)
4 :	$a$	4 :	$b$
5 :	(libre, 1)	5 :	$c$
6 :	(occupe, 3)	6 :	(libre, 5)
7 :	$b$	7 :	$X$
8 :	$c$	8 :	$X$
9 :	(libre, 1)	9 :	$X$
10 :	(libre, 1)	10 :	$X$

Avant

Après

## Inversion de pointeurs

- Une méthode pour visiter en profondeur un graphe qui **n'utilise pas une pile** mais qui demande de réserver un petit nombre de bits pour chaque cellule du tas.
- Pour simplifier, on suppose que chaque cellule pointée par  $x$  contient **deux pointeurs** au tas qu'on désigne par  $x:f_0$  et  $x:f_1$ .
- En plus, chaque cellule contient un champ d'un bit *mark* et un champ de 2 bits *done* (en général, le nombre de bits dans ce champ est **logarithmique** dans le nombre de pointeurs au tas dans la cellule).

```
local current, pred, next, i;  
current := root; current.done := 0; current.mark := 1; pred := nil;  
while true do  
  i := current.done;  
  if i < 2  
  then  
    next := current.fi  
    if next.mark = 0  
    then  
      current.fi := pred; pred := current; (1)  
      current := next; current.mark := 1; current.done := 0;  
    else  
      current.done := i + 1;  
    end  
  else  
    next := current; current := pred;  
    if current = nil then STOP;  
    i := current.done; pred := current.fi;  
    current.fi := next; (2) current.done := i + 1;
```

- Dans (1), *current:fi* est sauvé dans *next* et il pointe ensuite à la cellule d'où *current* a été accédé.
- Dans (2), la valeur originale de *current:fi* est restaurée.



## Exemple

La trace d'exécution de l'algorithme sur le graphe

$$G = \{(1;2);(1;4);(2;3);(2;4);(3;1);(3;2);(4;1);(4;3)\}$$

current 1, 2, 3, 2, 4, 2, 1, *nil* ( $\rightarrow$  STOP)

next 2, 3, 2, 1, 3, 4, 3, 1, 4, 2, 4, 1

pred *nil*, 1, 2, 1, 2, 1, *nil*

	mark	done	$f_0$	$f_1$
1	0, 1	-, 0, 1, 2	2, <i>nil</i> , 2	4
2	0, 1	-, 0, 1, 2	3, 1, 3	4, 1, 4
3	0, 1	-, 0, 1, 2	2	1
4	0, 1	-, 0, 1, 2	3	1

## Notion de recuperation generationale

- Les cellules allouées récemment ont beaucoup **plus de chance** d'être récupérables.
- Division de la mémoire en **generations**.
- On récupère plus souvent les **dernieres** générations.
- On garde une trace de l'**age des cellules**.
- On doit accéder rapidement aux cellules des dernières générations sans avoir à traverser tout le tas. Dans ce but :
  - On garde une **liste de pointeurs** de l'ancienne génération vers la nouvelle.
  - On exploite le fait que souvent la nouvelle génération **ne pointe pas** vers l'ancienne.

## Notion de recuperation concurrente

Le problème est de faire tourner en ‘parallèle’ :

**Mutator** un programme qui modifie le graphe des noeuds accessibles.

**Collector** un programme qui marque les noeuds accessibles et insère ceux inaccessibles dans la liste libre.

### Propriete souhaitee

Un noeud non-accessible **au debut** d’une phase de récupération devrait être inséré dans la liste libre à la fin de la phase de récupération.

## Probleme

On suppose que  $A$  et  $B$  sont **accessibles** et que le **mutator** produit les configurations 1-5 alors que le **collector** est actif aux pas 3 et 5.

Pas	Mutator	Pile collector	Noeuds marqués
1	$A \rightarrow C \quad B$	$A; B$	-
2	$A \rightarrow C \leftarrow B$	$A; B$	-
3	$A \quad C \leftarrow B$	$B$	$A$
4	$A \rightarrow C \leftarrow B$	$B$	$A$
5	$A \rightarrow C \quad B$	-	$A; B$

On ne marque pas  $C$  alors qu'il est toujours **accessible**.

## Approche a trois couleurs (Dijkstra et al. 1972)

**Blanc** les noeuds qui n'ont pas encore été accédés (au début tous les noeuds sont blancs).

**Gris** les noeuds qui sont sur la pile (ou la queue), ils sont considérés comme accessibles mais on n'a pas encore exploré les fils.

**Noir** les noeuds qui ont été marqués et dont on a exploré les fils.

## Invariants

- Les seules transitions de couleur sont de clair vers sombre.

Pour la **terminaison**, il suffit donc de montrer que forcément on arrive à une transition de couleur.

- **Un noeud noir ne doit pas pointer a un noeud blanc.**

Si on veut rediriger un pointeur vers un noeud blanc (pas visité) alors automatiquement le noeud devient gris (il est inséré dans la pile).

## Methode revisee avec 2 couleurs (Ben-Ari 1982)

Chaque fois que le **mutator** redirige un pointeur vers un noeud  $i$ , la couleur de  $i$  devient noir.

Le **collector** maintient deux variables :  $BC$  (*black count*) et  $OBC$  (*old black count*).

1. Il **noircit les racines**.
2. Il parcourt tout le tas et **noircit les noeuds blancs** pointés par un noeud noir (**NB** l'inversion de pointeur ne peut pas être utilisée).
3. Il **compte le nombre de noeuds noirs** (mise à jour de  $BC$ ).
4. Si  $OBC < BC$  alors  $OBC := BC$ ; goto 2.
5. Il parcourt tout les tas, il **blanchit les noeuds noirs** et **met les noeuds blancs dans la liste libre**.

## Remarques

- On fait l'hypothèse que le pointeur à la **liste libre** et le **noeud *nil*** font partie des racines.
- Ainsi l'activité du **mutator** se résume à sélectionner deux noeuds **accessibles**  $i$  et  $j$ , à rediriger un pointeur de  $i$  vers  $j$  et à noircir  $j$ . **NB** Ce n'est **pas correct** d'inverser ces deux opérations!
- **Terminaison.** Tant que le **collector** n'arrive pas à l'étape 5 le nombre de noeuds noirs peut seulement **augmenter**. Ceci assure que *in ne* le collector arrive à l'étape 5.



- Démontrer qu'à l'étape 5

**tous les noeuds blancs sont inaccessibles**

est la partie **delicate** de la preuve.

- **Petit historique :**

1. Dijkstra et al. 1978. Algorithme à 3 couleurs.
2. Ben Ari 1981, Réduction à 2 couleurs pour avoir une preuve élégante (au sens mathématique).
3. Van de Snepscheut 1987, Preuve revisitée en utilisant (parfois) les principes de raisonnement de Owicki-Gries (logique de Hoare pour programmes parallèles).
4. Prensa-Nieto 2002, preuve de Van de Snepscheut et logique de Owicki-Gries formalisée et certifiée en Isabelle-HOL.

- **Completeness** Un noeud noirci peut devenir *garbage* à cause de l'activité du *mutator*. Un tel noeud sera récupéré à l'itération suivante du *collector*.

## Sommaire

### Methodes de base

- Marquage et ramassage.
  - Récupération par copie.
- et une méthode ‘historique’ (comptage des références).

**Desiderata** Pas besoin de mémoire additionnelle, compacte la mémoire, efficace, incrémental,:::

### Elaborations

- notion de génération pour améliorer la chance de récupération,
- récupération concurrente.

## Lecture conseillée

Appel. Modern compiler implementation in ML (C, Java). Chapitre 13.

# Interprétation et Compilation de Fonctions d'Ordre Supérieur

## Comment executer un programme d'un langage d'ordre superieur ?

- Chaque fonction/code doit calculer dans un **environnement** approprié.
- On est donc amené à manipuler un couple (**code**, **environnement**) qu'on appelle **clôture** (ou fermeture).
- On peut distinguer **deux approches**.

1. On construit un **interprete/machine abstraite** qui manipule efficacement les clôtures. L'interprète est obtenu par raffinement de la sémantique opérationnelle. Exemples : la machine SECD de Landin jusqu'à la machine abstraite de OCAML.
2. On **compile** le programme d'ordre supérieur vers un programme du premier ordre équivalent (par exemple C ou une version abstraite d'un langage assembleur). La compilation consiste alors à expliciter la notion de clôture. Exemples : les compilateurs autour de STANDARD ML-NJ.

## Plan

- Liaison statique et dynamique. Notion de clôture.
- Exemple de machine abstraite.
- Exemple de compilation vers un langage du premier-ordre.
- Gestion de la mémoire basée sur les régions.



# Liason statique et dynamique

## Un simple langage d'expressions

On analyse la différence entre liaison **statique** et **dynamique** dans le cadre d'un simple langage d'expressions :

$$e ::= \perp \mid n \mid x \mid \textit{let } x = e \textit{ in } e' \mid \textit{quote}(e) \mid \textit{unquote}(e)$$

- $\perp$  représente un calcul qui **diverge**,
- $n$  est un **entier**,
- *quote* permet de **bloquer** l'évaluation d'une expression et *unquote* permet de la **debloquer**.
- L'ensemble des **valeurs** est défini par

$$v ::= n \mid$$

## Liaison dynamique

**Environnement** : une fonction partielle (à domaine fini)

$: Id * Exp$

**Evaluation** : En **big-step** :  $(e; \eta) \Downarrow v$ .

$$\frac{}{(v, \eta) \Downarrow v}$$

$$\frac{(\eta(x), \eta) \Downarrow v}{(x, \eta) \Downarrow v}$$

$$\frac{(e, \eta) \Downarrow \text{quote}(e') \quad (e', \eta) \Downarrow v}{(\text{unquote}(e), \eta) \Downarrow v}$$

$$\frac{(e, \eta) \Downarrow n}{(\text{unquote}(e), \eta) \Downarrow n}$$

Par nom :

$$\frac{(e, \eta[e'/x]) \Downarrow v,}{(\text{let } x = e' \text{ in } e, \eta) \Downarrow v}$$

Par valeur :

$$\frac{(e', \eta) \Downarrow u \quad (e, \eta[u/x]) \Downarrow v}{(\text{let } x = e' \text{ in } e, \eta) \Downarrow v}$$

## Liaison statique

**Environnement**  $Env$  est le plus petit ensemble de fonctions partielles sur  $Id$  tel que la fonction à domaine vide est un environnement et

si  $e_1; \dots; e_n \in Exp$

$\sigma_1; \dots; \sigma_n \in Env$

$FV(e_i) \subseteq dom(\sigma_i)$

alors  $[(e_1; \sigma_1) = x_1; \dots; (e_n; \sigma_n) = x_n] \in Env$

On appelle aussi **clôture** un couple  $(e; \sigma)$  constitué d'un code (une expression dans notre cas) et d'un environnement.

## Evaluation En big-step

$$\frac{}{(v, \eta) \Downarrow (v, \eta)} \qquad \frac{\eta(x) \Downarrow (v, \eta')}{(x, \eta) \Downarrow (v, \eta')}$$

$$\frac{\begin{array}{c} (e, \eta) \Downarrow (\text{quote}(e'), \eta') \\ (e', \eta') \Downarrow (v, \eta'') \end{array}}{(\text{unquote}(e), \eta) \Downarrow (v, \eta'')} \qquad \frac{(e, \eta) \Downarrow (n, \eta')}{(\text{unquote}(e), \eta) \Downarrow (n, \eta')}$$

Par nom :

$$\frac{(e, \eta[(e', \eta)/x]) \Downarrow (n, \eta_1)}{(\text{let } x = e' \text{ in } e, \eta) \Downarrow (n, \eta_1)}$$

Par valeur :

$$\frac{\begin{array}{c} (e_1, \eta) \Downarrow (m, \eta_2) \\ (e, \eta[(m, \eta_2)/x]) \Downarrow (n, \eta_1) \end{array}}{(\text{let } x = e_1 \text{ in } e, \eta) \Downarrow (n, \eta_1)}$$

## Exercice

Donnez des exemples d'expressions qui distinguent les 4 combinaisons possibles :

1. Statique-Valeur.
2. Statique-Nom.
3. Dynamique-Valeur.
4. Dynamique-Nom.

## Solutions

- *let*  $x = \perp$  *in* 3 distingue **appel par nom** et **appel par valeur** dans les deux types de liaison. A savoir, l'évaluation converge par nom et diverge par valeur.

– Reste à comparer :

1. dynamique+nom et statique+nom et
2. dynamique+valeur et statique+valeur.

Soit :

$$e_1 \equiv \textit{let } x = 3 \textit{ in } e_2; \quad e_2 \equiv \textit{let } y = x \textit{ in } e_3; \quad e_3 \equiv \textit{let } x = 5 \textit{ in } y :$$

- En dynamique par nom,  $(e_1; \emptyset) \Downarrow 5$ .
- En dynamique par valeur,  $(e_1; \emptyset) \Downarrow 3$ .
- En statique par nom  $(e_1; \emptyset) \Downarrow (3; \emptyset)$ .
- En statique par valeur  $(e_1; \emptyset) \Downarrow (3; \emptyset)$ .



- Reste à comparer **dynamique et statique par valeur**.
  - Si l'on se restreint à des expressions sans *quote*, *unquote* alors les deux stratégies coïncident.
- En effet dans la liaison statique on va associer à une variable un nombre et donc l'environnement ne joue pas de rôle.

- On modifie donc l'exemple précédent comme suit :

$$\begin{aligned} e_1 &\equiv \text{let } x = 3 \text{ in } e_2, & e_2 &\equiv \text{let } y = \text{quote}(x) \text{ in } e_3, \\ e_3 &\equiv \text{let } x = 5 \text{ in } \text{unquote}(y) . \end{aligned}$$

Maintenant  $(e_1; \emptyset)$  s'évalue en 5 en dynamique par valeur et en  $(3; \emptyset)$  en statique par valeur.

# Un exemple de machine abstraite

## Un fragment de langage fonctionnel (le $\lambda$ -calcul)

$$\begin{aligned}
 M ::= & \text{ } id \quad (\text{variable}) \\
 & id:M \quad (\text{abstraction}) \\
 & (MM) \quad (\text{application})
 \end{aligned}$$

**NB** En ocaml,  $x:M$  s'écrit `function  $x \rightarrow M$ .`

## Environnements et clôtures dans le $\lambda$ -calcul

- On reprend la définition d'environnement en remplaçant les expressions  $e$  par des  $\lambda$ -termes  $M$ .
- Les clôtures  $c; c'; :::$  sont des couples (terme, environnement) qu'on dénote par  $M[ \ ]$ .
- Une **valeur-clôture** est une clôture de la forme  $(\lambda x.M)[ \ ]$ .

## Evaluation big-step par valeur sur les clôtures

$$\frac{}{vc \Downarrow_V vc} \qquad \frac{\eta(x) \Downarrow_V vc}{x[\eta] \Downarrow_V vc}$$

$$\frac{M[\eta] \Downarrow_V \lambda x.M_1[\eta'] \quad M'[\eta] \Downarrow_V vc' \quad M_1[\eta'[vc'/x]] \Downarrow_V vc}{(MM')[\eta] \Downarrow_V vc}$$

**NB** On peut donner des règles similaires pour l'appel par nom.

## Clôtures et substitutions explicites

- Les règles précédentes décrivent des réductions **faibles** ; on ne réduit pas sous un  $\lambda$ .
- Par ailleurs, une **clôture** peut être vue comme une façon d'implémenter/représenter explicitement l'opération de **substitution** :

$$\begin{aligned} (\lambda x:M) V &\rightarrow [V=x]M \\ ((\lambda x:M) V)[\sigma] &\rightarrow M[\sigma[V[\sigma]=x]] \end{aligned}$$

## Contexte d'evaluation et pile

- A chaque pas, l'évaluateur doit explorer le terme à la recherche d'un **redex** (= le sous-terme à simplifier)
- On peut raffiner la définition de l'évaluateur en introduisant une **pile** qui maintient une trace des valeurs calculées et des termes à évaluer.
- La pile correspond en effet au '**contexte d'evaluation**' (= la partie du terme autour du redex).

**Exemple** Dans  $3 + (5 * 7)$ , le redex est  $5 * 7$  et le contexte d'évaluation  $3 + [ ]$ .



## Contexte d'evaluation (definition formelle)

- Un **contexte d'evaluation** est un terme qui contient une occurrence d'un symbole spéciale  $[]$  qu'on appelle le 'trou' (*hole*).
- Pour le  $\lambda$ -calcul, les contextes d'évaluation qui correspondent à une stratégie d'évaluation **par valeur de gauche a droite** sont :

$$E ::= [] \mid EM \mid VE$$

- Si  $E$  est un contexte et  $M$  un terme alors  $E[M]$  est le terme qui resulte du **remplacement** en  $E$  du trou  $[]$  par le terme  $M$ .

## Machine abstraite pour l'appel par valeur

- Un contexte d'évaluation est maintenant la composition de contextes élémentaires de la forme  $[ ]N$  ou  $V[ ]$ .
- On abrège :

$$[ ]c \equiv r; c \quad r \text{ pour right}$$

$$vC[ ] \equiv l; vC \quad l \text{ pour left}$$

- La pile  $S$  a donc la forme :

$$S = m_1 : c_1 : \dots : m_n : c_n \quad \text{où } m \in \{l; r\}$$

- Le calcul est décrit par les règles suivantes :

$$\begin{array}{ll}
 (x[];s) & \rightarrow (\lambda (x);s) \\
 ((MM')[];s) & \rightarrow (M[];r : M'[] : s) \\
 (vc;r : c : s) & \rightarrow (c;l : vc : s) \\
 (vc;l : (\lambda x:M)[] : s) & \rightarrow (M[] [vc=x]);s)
 \end{array}$$

## Exercice

On ajoute au  $\lambda$ -calcul en appel par valeur des opérateurs  $op_1; \dots; op_m$  avec arité  $n_1; \dots; n_m$ ,  $n_j \geq 0$ .

1. Quels sont les nouveaux contextes d'évaluation ?
2. Comment faut-il modifier la machine abstraite ?

## Mise-en-oeuvre : structuration de la memoire

- On analyse (un petit peu) la mise en oeuvre de l'évaluateur pour l'appel par valeur.
- L'évaluateur gère une **memoire** qui est divisée en **trois parties** :
  1. Statique.
  2. Pile.
  3. Tas.

**Statique** Cette partie contient :

- Les **instructions** à exécuter.
- Un pointeur **pt\_code** à la prochaine instruction à exécuter. Initialement ce pointeur pointe à la première instruction.
- Un pointeur **pt\_stack** au sommet de la pile. Initialement ce pointeur pointe à la base de la pile.
- Un pointeur **pt\_env** à l'environnement courant (dans le tas) Initialement ce pointeur est **nil**.
- Un pointeur **pt\_free** à la première cellule libre du tas.

**Pile** Une zone contiguë de mémoire dont le sommet est pointé par `pt_stack`. Initialement la pile est vide.

**Tas (heap)** Une zone de mémoire qui est initialement liée pour former une liste de *cellules libres* et le premier élément de la liste est pointé par `pt_free`.

## Mise en oeuvre : representation des clôtures

Instructions, éléments de la pile et éléments du tas sont structurés comme des **enregistrements** :

**Code** 3 champs : **op** l'étiquette de l'instruction, **left** le pointeur gauche et **right** le pointeur droit.

**Pile** 3 champs : marqueur **m**, pointeur au code **code**, pointeur à l'environnement **env**.

**Tas** 4 champs : **var** nom de la variable, **code** pointeur au code, **env** pointeur à l'environnement et **next** pointeur au prochain élément du tas.



## Mise-en-uvre : interpretation

L'évaluateur est une boucle *while* qui lit la prochaine instruction et l'exécute. Par exemple, dans le cas de l'**application**

$$((MN)[\eta], s) \rightarrow (M[\eta], r : N[\eta] : s)$$

```
Eval :  case pt_code.op of
...
@ :      let p = push() in
          p.code := pt_code.right;
          p.env := pt_env;
          p.m := r;
          pt_code := pt_code.left;
          goto Eval
```

**NB** La duplication de l'environnement est en 'temps constant'.

## Exercice

Complétez la description de la mise-en-oeuvre de l'évaluateur à l'aide de pointeurs.

Exemple de compilation vers  
un langage du premier ordre

## Closure conversion + Hoisting

Dans sa forme la plus simple, la compilation se decompose en **deux phases** :

- Explicitation des clôtures (*Closure conversion*).
- Surélévement des définitions de fonction (*Hoisting*).

On obtient alors un programme qui pourrait être exprimé en Cminor.

En ajoutant deux transformations (**continuation passing style + nommage des valeurs**) on peut aussi compiler directement vers un langage assembleur abstrait comme RTLabs.

## Une extension du $\lambda$ -calcul

Pour exprimer les transformations il convient de considérer un langage un peu plus riche que le  $\lambda$ -calcul qui contient :

definitions let

$$\text{let } x = M \text{ in } N$$

construction et projection de tuples

$$(M_1 ; \dots ; M_n) \quad M.i$$

abstraction et application polyadique

$$x_1 ; \dots ; x_n . M \quad @ (M ; N_1 ; \dots ; N_n) \quad i \geq 1$$

## Explicitation des clôtures

- Si  $M$  est un terme,  $\text{fv}(M)$  (*free variables*) est l'ensemble des variables qui paraissent libres dans  $M$ .
- On définit une fonction qui transforme un terme  $M$  dans un terme  $\mathcal{C}(M)$  équivalent où **toutes les fonctions sont closes**.
- Ceci veut dire que pour tout sous-terme  $N$  de la forme  $x_1 :: \dots :: x_n : N'$  on a  $\text{fv}(N) = \emptyset$ .
- L'idée pour 'clôre' une fonction est de lui ajouter **un argument qui représente l'environnement** dans lequel elle doit être évaluée.

## La fonction qui `explicite les clôtures'

On écrit  $\text{let } (y_1; \dots; y_m) = N \text{ in } M$  pour

$$\text{let } y_1 = N:1 \text{ in } \dots \text{let } y_m = N:m \text{ in } M$$

$$\begin{aligned} \mathcal{C}(x_1; \dots; x_n; M) &= (e; x_1; \dots; x_n; \text{let } (y_1; \dots; y_m) = e \text{ in } \mathcal{C}(M); \\ &\quad (y_1; \dots; y_m)) \\ \text{si } \text{fv}(x_1; \dots; x_n; M) &= \{y_1; \dots; y_m\} \end{aligned}$$

$$\mathcal{C}(@ (M; N_1; \dots; N_n)) = \text{let } (f; e) = \mathcal{C}(M) \text{ in } @ (f; e; \mathcal{C}(N_1); \dots; \mathcal{C}(N_n))$$

$$\mathcal{C}(x) = x$$

$$\mathcal{C}((M_1; \dots; M_n)) = (\mathcal{C}(M_1); \dots; \mathcal{C}(M_n))$$

$$\mathcal{C}(M:i) = \mathcal{C}(M):i$$

$$\mathcal{C}(\text{let } x = M \text{ in } N) = \text{let } x = \mathcal{C}(M) \text{ in } \mathcal{C}(N)$$

## Exemple

$$\mathcal{C}(x: y:@(x;y)) = (e;x:\mathcal{C}(y:@(x;y));())$$

$$\mathcal{C}(y:@(x;y)) = (e;y:\text{let } (x) = e \text{ in } \mathcal{C}(@ (x;y));(x))$$

$$\mathcal{C}(@ (x;y)) = \text{let } (f;e) = x \text{ in } @(f;e;y)$$

**NB** L'abstraction  $y:@(x;y)$  devient un terme clos et une tuple qui contient la valeur de la variable libre  $x$ .



## Surelevement des définitions de fonction

- Un **terme simple** est un terme qui ne contient pas d'abstractions. On dénote ces termes par  $T; S; \dots$
- Un **contexte fonctionnel** est un contexte de la forme :

$$\text{let } x_1 = y_1^+ : T_1 \text{ in } \dots \text{let } x_k = y_k^+ : T_k \text{ in } [ ]$$

- On définit une **fonction**  $\mathcal{C}_h$  qui associe à un terme  $M$  avec **fonctions closes** un couple  $\mathcal{H}(M)$  constitué d'un **terme simple** et un **contexte fonctionnel**.

- Si  $\mathcal{H}(M) = (T; F)$  alors le **resultat de la compilation** est le terme  $F[T]$ .
- On reconnaît la **structure d'un programme**  $C$  : une liste de fonctions (le context  $F$ ) et une fonction main (le terme  $T$ ).

## La fonction $\mathcal{C}_h$ (1/2)

$$\mathcal{H}(x_1; \dots; x_n; M) = \text{let } (T; F) = \mathcal{H}(M) \text{ in let } x = \text{new}() \text{ in} \\ (x; F[\text{let } x = x_1; \dots; x_n; T \text{ in } []])$$

$$\mathcal{H}(@ (M; N_1; \dots; N_n)) = \text{let } (T; F) = \mathcal{H}(M); (T_1; F_1) = \mathcal{H}(N_1); \dots; \\ (T_n; F_n) = \mathcal{H}(N_n) \text{ in} \\ (@ (T; T_1; \dots; T_n); F[F_1[:::[F_n]:::]])$$

$$\mathcal{H}(x) = (x; [])$$

## La fonction $\mathcal{C}_h$ (2/2)

$$\mathcal{H}((M_1, \dots, M_n)) = \text{let } (T_1, F_1) = \mathcal{H}(M_1), \dots, (T_n, F_n) = \mathcal{H}(M_n) \text{ in } ( (T_1, \dots, T_n), F_1[\dots[F_n]\dots] )$$

$$\mathcal{H}(M.i) = \text{let } (T, F) = \mathcal{H}(M) \text{ in } ( T.i, F )$$

$$\mathcal{H}(\text{let } x = M_1 \text{ in } M_2) = \text{let } (T_1, F_1) = \mathcal{H}(M_1), (T_2, F_2) = M_2 \text{ in } ( T_2, F_1[\text{let } x = T_1 \text{ in } F_2] ) \quad M_1 \neq \lambda y^+.M$$

$$\mathcal{H}(\text{let } x = \lambda y^+.M_1 \text{ in } M_2) = \text{let } (T_1, F_1) = \mathcal{H}(M_1), (T_2, F_2) = M_2 \text{ in } ( T_2, F_1[\text{let } x = \lambda y^+.T_1 \text{ in } F_2] )$$

## Exercice

1. Calculer :

$$\mathcal{H}(\text{let } x_1 = (y_1) \text{ in let } x_2 = y_2:T_1 \text{ in } T_2)$$

2. Calculer :

$$\mathcal{H}(\text{let } x_1 = (y_1.\text{let } x_2 = y_2:T_2 \text{ in } T_1) \text{ in } T_0)$$

# Gestion de la mémoire basée sur les régions

## Gestion de la memoire basee sur les regions (1/2)

- Une alternative/complément au ramasse miettes consiste à garantir par une **analyse statique** que l'opération de deallocation (`free`) est 'sûre'.
- On décrit une approche basée sur la notion de **region** (*region based memory management*) dans le contexte d'un **langage fonctionnel** similaire à celui obtenu après explicitation des clôtures et surélévement des définitions.
- Néanmoins, l'approche n'est pas spécifique aux langages fonctionnels. Par exemple, elle a été appliquée aussi à C et à Java.

## Gestion de la memoire basee sur les regions (2/2)

Operation	Notation
Allouer une région	let all( $r$ ) in $\dots$
Allouer une donnée dans une région	let $x = (y_1; \dots; y_n) \text{at}(r)$ in $\dots$
Acceder une valeur	let $x = y:i$ in $\dots$
Passer une région en paramètre	let $x = r \dots \quad @ (s; r; \dots)$
Disposer une région	dis( $r$ ) in $\dots$



## Idees sur la mise-en-oeuvre

- La mémoire disponible est partitionnée en pages qui sont insérées dans la **liste libre**.
- Une **region** est un ‘pointeur’ à un descripteur qui contient un pointeur au premier et dernier élément d’une liste de pages et un compteur qui donne la quantité de mémoire disponible dans la dernière page de la liste.
- Pour **allouer une region** on initialise le descripteur.
- Pour **allouer une donnee** dans une région, on vérifie si la dernière page de la région dispose d’assez de mémoire. Sinon on prend une (ou plusieurs) pages de la liste libre et on les insère dans la liste associée à la région.

- Une **donnee** est simplement un pointeur à la mémoire et son **acces** est direct (on ne passe pas par la région).
- Pour **disposer une region** on concatène la liste libre à la liste des pages associées à la région.

**NB** Si la taille des données à allouer est connue/bornée au moment de la compilation alors toutes les opérations sont réalisées en **temps constant**.

## Systeme d'analyse statique

- Étant donné un programme (niveau RTL), le **compilateur** doit ajouter les instructions pour gérer la mémoire (points d'allocation/deallocation des regions, allocation des données aux regions, :::) )
- L'algorithme doit être **correct** :
  1. pas d'accès d'une donnée dans une région disposée et
  2. on dispose une région au plus une fois.
- La **formalisation** est délicate (et omise) !
  1. définir un système de **types** et **e** ets.
  2. montrer qu'un **programme bien type** ne produit pas **d'erreurs** d'accès à la mémoire.
  3. construire et résoudre des **systèmes de contraintes** pour inférer les positions où allouer/disposer les régions.

- Une **solution inefficace est toujours possible** : on alloue toutes les données dans une région qu'on dispose seulement à la fin du calcul.
- Dans les premières implémentations on a utilisé une **strategie a pile** : la région disposée est la dernière allouée.
- Pour certains programmes la strategie à pile est inefficace mais il est possible de **decoupler allocation et deallocation**. Voir Aiken *et al.* PLDI 1995 pour une évaluation expérimentale.
- On peut aussi **combiner** l'analyse statique avec le GC.

## Sommaire

La structure de données essentielle pour implémenter les fonctions d'ordre supérieur est celle de

**clôture = code + environnement :**

On peut :

- Manipuler explicitement la structure au niveau d'une machine abstraite.
- Simuler la structure dans un langage du premier ordre (avec pointeurs au code).

