

Compilation : de C au langage machine

Cours M1 Ingénierie Informatique

Université Paris Diderot (Paris 7)

AA 2011-2012

Roberto M. Amadio

1

Préliminaires

3

Sommaire du cours

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

2

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.

$$C : L \rightarrow L'$$

- 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**.

$$w \in L ?$$

4

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

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

Langages Intermédiaires Optimisations sur le code intermédiaire. Sélection d'instructions.

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

5

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.

7

Exemple d'un compilateur : gcc

Front-end C, C++, Java, Ada, Fortran,...

Langages intermédiaires 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*).

6

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'.

8

Objectifs du cours

Notions d'intérêt général

- Sémantique opérationnelle.
- Transformations de programmes.
- Analyse du flot de données et du contrôle.
- Gestion de la mémoire.

9

Un projet de génie 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).

11

Un compilateur optimisant pour C

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

10

Cours connexes

Pré-requis (L3)

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

Cours du M1

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

Cours plus avancés (M2)

- Parcours **Programmation** (M2-II)
- Parcours **Sémantique et/ou Vérification** (M2-MPRI)

12

Emploi du temps

Cours R. Amadio. 2h/semaine.

TP/Projet N. Ayache. 2h/semaine.

13

Plan des TP/Projet

TP Présentation d'un *compilateur expérimental* 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. En particulier il s'agira :

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

15

Plan du cours

Introduction

Front end

C → Clight → Cminor → RTLAbs

Langages assembleurs Les exemples de Mips et 8051.

Back end

RTLAbs → RTL → ERTL → LTL → LIN → Mips

Compléments

- Gestion de la mémoire.

Révision

14

Références

- Les **transparentes 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. Voir <http://www.pps.jussieu.fr/~ayache>
- 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 encyclopédie.

16

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 et sur ses conséquences.

17

Spécification de la sémantique (opérationnelle)

18

Outils de spécification

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

Sémantique opérationnelle

La *sémantique opérationnelle* d'un *langage de programmation* est constituée (au moins) d'un *système formel* (au sens de la logique mathématique) qui permet de décrire à un certain niveau d'abstraction les *exécutions 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'**équivalence** ou d'**approximation** entre programmes.

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

19

20

Exemple (automate à 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 **spécifier les calculs de A** .
- Typiquement, on décrit les **petits pas** par des règles :

$$(q, a \cdot w, X \cdot \gamma) \vdash (q', w, \gamma' \cdot \gamma)$$

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

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

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

21

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 **représentation à 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 **déclarations de type et de constructeurs**.

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

23

Un autre exemple : le langage Imp

id	$::= x \mid y \mid \dots$	(identifiants)
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 booléennes)
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)

22

Modèle mémoire

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

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

- Notation pour la **mise-à-jour** :

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

24

Évaluation des expressions (grands pas)

$$\frac{}{(v, s) \Downarrow v} \quad \frac{}{(x, s) \Downarrow s(x)}$$

$$\frac{(e, s) \Downarrow v \quad (e', s) \Downarrow v'}{(e + e', s) \Downarrow (v +_{\mathbf{Z}} v')} \quad \frac{(e, s) \Downarrow v \quad (e', s) \Downarrow v'}{(e < e', s) \Downarrow (v <_{\mathbf{Z}} v')}$$

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

25

Trois possibilités

On converge a une valeur/résultat

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

On diverge

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

On est bloqué (erreur)

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

27

Évaluation de commandes et programmes (grands pas)

$$\frac{}{(\text{skip}, s) \Downarrow s} \quad \frac{(e, s) \Downarrow v}{(x := e, s) \Downarrow s[v/x]} \quad \frac{(S_1, s) \Downarrow s' \quad (S_2, s') \Downarrow s''}{(S_1; S_2, s) \Downarrow s''}$$

$$\frac{(b, s) \Downarrow \text{true} \quad (S, s) \Downarrow s'}{(\text{if } b \text{ then } S \text{ else } S', s) \Downarrow s'} \quad \frac{(b, s) \Downarrow \text{false} \quad (S', s) \Downarrow s'}{(\text{if } b \text{ then } S \text{ else } S', s) \Downarrow s'}$$

$$\frac{(b, s) \Downarrow \text{false}}{(\text{while } b \text{ do } S, s) \Downarrow s} \quad \frac{(b, s) \Downarrow \text{true} \quad (S; \text{while } b \text{ do } S, s) \Downarrow s'}{(\text{while } b \text{ do } S, s) \Downarrow s'}$$

$$\frac{(S, s) \Downarrow s'}{(\text{prog } S, s) \Downarrow s'}$$

26

Exercice (déterminisme, 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.

28

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).

29

Sémantique à petits pas

- La sémantique à **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 à **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)

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

31

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**.

30

Sémantique à petits pas des commandes

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

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

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

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

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

32

Toujours trois possibilités

On converge a une valeur/résultat

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

On diverge

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

On est bloqué (erreur)

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

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

33

Spécification d'un compilateur jouet

35

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.

34

Le langage Vm

- On définit une **machine virtuelle** Vm et son langage de programmation.
- Une machine virtuelle est essentiellement un **type de données** 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 **mémoire** s (comme pour lmp).
 4. Une **pile** d'entiers σ .

36

Instructions de la Vm (sémantique informelle)

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

37

Compilation de Imp à Vm

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

$$\mathcal{C}(x) = \text{var}(x) \quad \mathcal{C}(n) = \text{cnst}(n) \quad \mathcal{C}(e + e') = \mathcal{C}(e) \cdot \mathcal{C}(e') \cdot \text{add}$$

$$\mathcal{C}(e < e', k) = \mathcal{C}(e') \cdot \mathcal{C}(e) \cdot \text{bge}(k)$$

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

$$\mathcal{C}(\text{if } b \text{ then } S \text{ else } S') = \mathcal{C}(b, k) \cdot \mathcal{C}(S) \cdot (\text{branch}(k')) \cdot \mathcal{C}(S')$$

où : $k = sz(S) + 1$, $k' = sz(S')$

$$\mathcal{C}(\text{while } b \text{ do } S) = \mathcal{C}(b, k) \cdot \mathcal{C}(S) \cdot \text{branch}(k')$$

où : $k = sz(S) + 1$, $k' = -(sz(b) + sz(S) + 1)$

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

39

Instructions de la Vm (sémantique formelle à petits pas)

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

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

$$(C, s) \Downarrow s' \text{ si } C \vdash (0, \epsilon, s) \xrightarrow{*} (i, \epsilon, s') \text{ et } \mathcal{C}[i] = \text{halt} .$$

38

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.

40

- Par contre il **n'est pas** raisonnable de demander la **préservation 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.

41

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

43

Remarque (sur le non-déterminisme)

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

42

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 σ au moment de l'exécution de l'instruction i .
- Éventuellement, essayez de montrer la *correction* de votre algorithme.

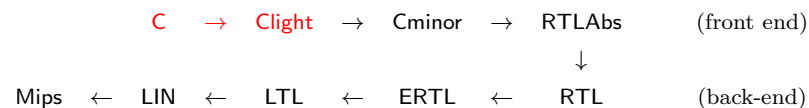
44

Passage à l'échelle

- Ils existent maintenant plusieurs exemples de **compilateurs réalistes** 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 **vérificateur de la preuve** est un petit programme (quelques centaines de lignes) qui est conceptuellement simple (implémentation directe des règles logiques).

45

Front end : de C à Clight



47

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

NB Dans le Projet chaque langage aura sa sémantique exécutable.

46

Plan du cours

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

48

Historique

- 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...

...

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 Clight à l'assembleur PowerPc) du compilateur CompCert a été formalisé et vérifié dans un assistant de preuve (Coq). D'ailleurs, le programme ocaml est extrait automatiquement des preuves.

- 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

53

Transformations CIL

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

Normalisation déclarations 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.

55

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 épurée** (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).

54

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] ;
```

56

Élévation déclarations 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); }
```

57

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.

59

Élimination d'effets 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; }
```

58

Clight : syntaxe

60

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.

61

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).

63

Conventions

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

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

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

62

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.

64

Syntaxe abstraite de Clight : Expressions

Voir `src/clight/clight.mli`.

Exemple

```
struct(s1, (n, int(I32, signed))(next, comp_pointer(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.

65

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

66

Syntaxe abstraite de Clight : Commandes

Voir `src/clight/clight.mli`.

```
type expr =
  | Expr of expr_descr*ctype
and expr_descr =
  | Econst_int of int (* integer *)
  | Evar of ident (* variable *)
  | Ederef of expr (* pointer dereference [*] *)
  | Eaddr of expr (* address-of operator [&] *)
  | Eunop of unary_operation*expr (* unary operation *)
  | Ebinop of binary_operation*expr*expr (* binary operation *)
  | Ecast of ctype*expr (* type cast [(ty)e] *)
  | Econdition of expr*expr*expr (* conditional [e1 ? e2 : e3] *)
  | ...
  | Esizeof of ctype (* size of a type *)
  | Efield of expr*ident (* access field struct/union *)
```

NB `float` n'est pas implémenté.

67

```
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 *)
  | Swhile of expr*statement (* [while] loop *)
  | ...
  | Sbreak (* [break] statement *)
  | Scontinue (* [continue] statement *)
  | Sreturn of expr option (* [return] statement *)
  | Sswitch of expr*labeled_statements (* [switch] statement *)
  | Slabel of label*statement (* labelled statement *)
  | Sgoto of label (* goto *)
and labeled_statements =
  | LSdefault of statement (* champ default obligatoire *)
  | LScase of int*statement*labeled_statements
```

68

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 *)
```

69

```
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 }
```

70

Structures de données sémantiques

Modèle mémoire

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

Références Un ensemble infini R (par exemple \mathbf{Z}).

Locations Une location est une référence et un *offset* :

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

Valeurs Une union disjointe d'entiers, pointeurs,...

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

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

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

71

72

Opérations sur la mémoire On peut :

- **Allouer** un bloc de mémoire à une *nouvelle* référence.
- **Libérer** (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.

73

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 *fid* :

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

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

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

74

Opérations 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

75

76

Évaluation 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.

77

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*cstype) option*cfunction*localEnv*continuation
```

79

Modes d'exécution 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
```

78

Règles à 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`.

80

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.

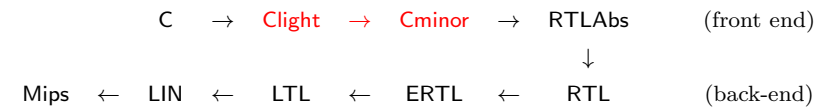
81

Plan du cours

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

83

Front end : de Clight à Cminor



82

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

Élimination 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`) :

`it_then_else, block, loop, exit n`

84

Syntaxe abstraite de Cminor (extrait)

Gestion explicite de la mémoire 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 données 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.

85

```
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
  ...
```

86

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.

```
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.

87

88

Exercice (loop, block et exit)

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

`loop{S}` Boucle infinie.

`block{S}` Déclaration d'un bloc.

`exit n` Termine les $n + 1$ blocs qui contiennent la commande.

Le **problème** 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 :

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

89

90

Exercice (petits pas, grands pas)

Proposez une sémantique :

– **petits pas** pour `break` et `continue`.

Suggestion : Introduisez une continuation `endloop(K)` similaire à `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.

91

Exercice (compilation vers loop, block et exit)

Proposez une **compilation** du langage `Imp` avec `break` et `continue` dans le langage `Imp` avec `loop`, `block` et `exit`.

Suggestion : on peut faire l'hypothèse que :

$$\mathcal{C}(\text{break}) = \text{exit } 1$$

$$\mathcal{C}(\text{continue}) = \text{exit } 0$$

92

Exercice (fonctions et continuations)

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

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

93

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

$$(e, ls, s) \Downarrow v$$

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

$$(f, S, K, ls, s) \rightarrow (f', S', K', ls', s')$$

95

- On considère une extension de **Imp** avec une notion rudimentaire de fonction. Un programme est maintenant une suite de **déclarations 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 :

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

- et celles des **continuations** devient :

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

94

Remarques

La formalisation en **Cminor** est un peu plus compliquée :

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

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

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

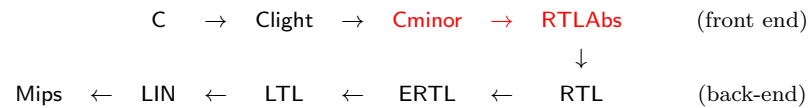
$$\mathbf{appel} \quad \textit{standard} \rightarrow \textit{call} \rightarrow \textit{standard}.$$

$$\mathbf{retour} \quad \textit{standard} \rightarrow \textit{call} \rightarrow \textit{standard}.$$

- 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.

96

Front end : de Cminor à RTLAbs



97

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.

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

99

Plan du cours

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

98

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 du contrôle** (ou CFG pour *control flow graph*).

100

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 *)

```

101

```

type internal_function = (* internal function *)
{ f_luniverse : Label.Gen.universe ; (* to generate labels *)
  f_runiverse : Register.universe ; (* to generate pseudo-registers *)
  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 }

```

102

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 **déclaration de fonction** :

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

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

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

NB \vec{y} , \vec{z} , \vec{e} , ... sont des vecteurs.

103

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

$$\begin{aligned}
 &\text{global } \vec{x}; \\
 &\text{function } f_1(\vec{y}_1) = \text{local } \vec{z}_1; S_1; \\
 &\dots \\
 &\text{function } f_n(\vec{y}_n) = \text{local } \vec{z}_n; S_n; \\
 &f_i()
 \end{aligned}$$

104

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

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

105

Un langage à la RTLAbs

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

```

global  $\vec{x}$ ;
function  $f_1(\vec{y}_1) = \text{local } \vec{z}_1; G_1;$ 
...
function  $f_n(\vec{y}_n) = \text{local } \vec{z}_n; G_n;$ 
 $f_i()$ 

```

107

- *Suggestion* :

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

$$t : Expression \rightarrow Commande \times Identificateur$$

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 ?

106

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

108

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

$l : \text{skip} \quad \rightarrow \ell'$
 $l : id := n \quad \rightarrow \ell'$
 $l : id := id \quad \rightarrow \ell'$
 $l : id := id + id \quad \rightarrow \ell'$
 $l : id < id \quad \rightarrow \ell', \ell''$
 $l : \text{branch} \quad \rightarrow \ell'$
 $l : id := f(\vec{id}) \quad \rightarrow \ell'$
 $l : \text{return}(id) \quad -$

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

109

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 $\ell \in Label$.
- On définit une fonction :

$$C : Commande \times Label \rightarrow Graphe \times Label$$

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

110

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

$$C : Commande \times Label \times Label^* \rightarrow Graphe \times Label$$

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.

111

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.

112

Langages Assembleurs

113

Généralités

- 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 **désassembleur** celui qui fait l'opération inverse.
- Un langage assembleur peut contenir des **étiquettes** 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**.

115

Plan du cours

- Généralités.
- Mips.
- 8051.

114

Quelques critères de classification (des processeurs et des langages assembleurs)

- Taille du **mot mémoire** : 8, 16, 32, 64, ...
- Organisation de la **mémoire** (registres, cache, ROM, RAM, ...)
- Séparation **code** et **données**.
- 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.

116

Mips (historique)

- Architecture **RISC 32** (et 64) bits développées par MIPS Tech. au début des années 80.
- L’objectif était d’exécuter en **pipeline** le cycle de chargement, décodage, exécution, accès mémoire, écriture registres.
- Dans ce but, il convient d’avoir (autant que possibles) des **instructions simples et uniformes** qui peuvent être exécutée dans un cycle d’horloge.
- Compétitif dans le marché des ordinateurs de bureau (desktop computers) jusqu’aux années 90, Mips est encore utilisé dans les **systèmes embarqués**.
- Toujours très populaire dans les **cours d’architecture**.

117

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 **systèmes embarqués** 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.

119

Mips (références)

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

Émulateur

- L’émulateur (X)SPIM est disponible sur <http://sourceforge.net>.
- Un émulateur/évaluateur écrit en ocaml sera disponible dans `/src/ASM/MIPSInterpret.ml`.

Compilateur gcc peut générer du code Mips.

118

8051 (références)

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

Émulateur

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

120

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,...

121

Mips : architecture de la mémoire

Mémoire	Registres pointeurs
↓ Pile ↓	← sp, fp
Espace non alloué	
↑ Tas (heap) ↑	dépend du GC
Données statiques (data segment)	← gp
Code (Text segment)	← pc
Partie réservée	

123

Points principaux à comprendre (pour compiler RTLABs)

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

122

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 ± imm	address of symbol + or - immediate
symbol (register)	address of symbol + contents of register
symbol ± imm (register)	(address of symbol + or - immediate) + contents of register

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

124

Mips : registres et conventions d'utilisation

Nom	Nombre	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Expression evaluation and
v1	3	results of a function
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)

125

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`.

127

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.

126

Exemples d'instructions de calcul

`li Rdest, imm` *Load immediate*

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

`addi Rdest, Rsrc, imm` *Add immediate (with overflow)*

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

128

Exemples d'instructions de saut

`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.

129

`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.

130

`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`.

131

`syscall` *System call*

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

132

Mips : appels système (extrait)

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

133

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
```

135

Programme 'Hello world'

```
.data
hello:
    .asciiz "hello world\n"
.text
main:
    li $v0, 4      #Code appel print_string
    la $a0, hello  #Adresse chaîne dans a0
    syscall        #Appel système
    jr $ra        #Fin du programme
```

134

Le factoriel récursif : 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
```

136

```

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

```

137

Exercice (récursion)

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.

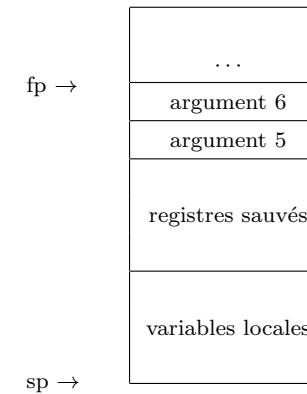
139

Exercice (récursion avec un seul pointeur à 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 $\$sp$.

138

Mips : structure du bloc d'activation (frame)



140

Notes sur les conventions d'appel

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

- L'**appelant** 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.

141

- 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**.

143

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

142

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 la fonction f .

144

8051 : organisation de la mémoire

Mémoire Externe Données 64 K octets (2 octets pour l'adresser dans DPTR=DPH+DPL).

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

Mémoire 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**.

145

Exemple adressage de la mémoire 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 à l'IRAM (et jamais aux SFR).

147

8051 : organisation de la mémoire 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**.

146

Registres principaux

Nom	Octets	Fonction
A	1	Accumulateur
B	1	Acc. pour mult. et division
SP	1	Pointeur à la pile
R0-7	1	Registres
PC	2	Compteur programme
DPTR	2	Pointeur aux données

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

148

8051 : jeu d'instructions

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

Modes d'adressage

Nom	Exemple
Immédiat	MOV A, #0X20
Direct	MOV A, 0X30
Indirect	MOV A, @RO
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.

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 IRAM_address
Push	PUSH IRAM_address

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

149

Exemple de programme généré 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);
}
```

151

150

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
```

152

```

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

```

153

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 mémoire externe*).

155

```

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

```

154

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.

156

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`.

157

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

Timers Le processeur dispose de *timers programmables* pour mesurer le passage du temps. Plus en général, pour chaque instruction, chaque constructeur spécifie le **nombre de cycles** nécessaires à son exécution. En connaissant la fréquence du processeur, on peut déterminer le temps nécessaire à l'exécution d'une suite d'instructions.

NB Ceci n'est pas forcément vrai pour des processeurs avec une architecture plus compliquée (pipeline, cache, ...)

159

Points non traités

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
```

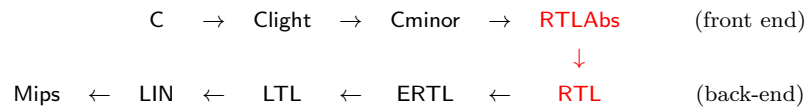
158

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 **priorité**.

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

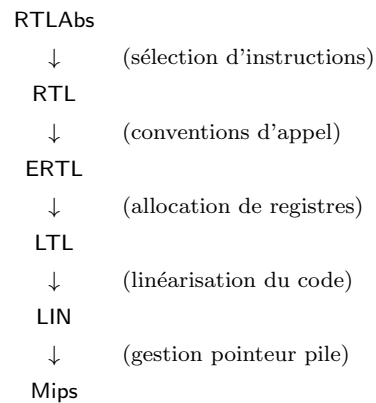
160

Back-end : de RTLAbs à RTL



161

Les transformations du back-end



163

Plan du cours

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

162

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.

164

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)
```

165

sum en RTL : sélection 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)
```

166

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
```

167

```
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 --> sum31                (retour)
```

168

sum en LTL : analyse de vivacité 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
```

169

```
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
```

170

sum en LIN : linéarisation

```
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
```

171

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

172

sum en Mips

```
.align 2           (mots alignes 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
```

173

Un petit exercice informel

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

RTLAbs → ... → 8051

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

Données=1 octet Un mot mémoire, un registre (= une adresse de la mémoire interne). Ceci suppose que toutes les données ont été décomposées en octets (voir traduction du premier TP).

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

175

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

174

On peut faire des **hypothèses** (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).

176

sum en 8051 (avant macro-expansion)

```
.align 2                (mots alignés 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
```

177

178

Ce qui resterait à faire pour générer un programme 8051

- Enlever l’erreur de type en utilisant LCALL
- Recalculer le décrémentation/incrémentation 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

179

180

Sélection d'instructions en CerCo

- La sélection d'instructions de RTLabs à RTL est **très simple** car la correspondance entre les instructions est pratiquement 1 à 1.
- 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).

181

Instructions de la machine

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

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

183

Expressions langage intermédiaire

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

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

182

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 :

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

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

184

- On peut associer à chaque instruction de la machine un **certain nombre de tuiles/filtres** (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
<code>cnst</code>	<code>reg</code>	<code>sum(reg, cnst)</code>	<code>sum(cnst, reg)</code>	<code>sum(x, cnst)</code>	<code>sum(cnst, x)</code>

185

Remarque (sur le coût)

- 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 générées**, ce qui revient à minimiser le nombre de tuiles qui recouvrent l'arbre.

187

Sélection d'instructions comme problème 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**.

186

Remarque (sur la sélection 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 $ERTL \rightarrow LTL$ 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.

188

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**?

189

Exercice (suite)

- **Appliquez** l'algorithme glouton à l'arbre t_0 considéré.
- Est-ce que le resultat de l'algorithme glouton change si l'on prend come critère d'optimisation local le ratio :
$$\text{nombre noeud couverts} / \text{coût tuile}?$$

191

Algorithme glouton (*maximal munch*)

- On énumère les noeuds de la racine vers le 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 instruction sont générées dans l'**ordre inverse** à celui de visite des noeuds.

190

Propriétés de l'algorithme glouton

- Si on trouve une solution alors il n'est pas possible d'en trouver une 'meilleure' (au sens du nombre d'instructions générées) en fusionnant deux 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**.

192

Exemple de programmation dynamique

Programmation dynamique (rappel)

- La programmation dynamique est une **technique d’optimisation** (et de programmation) qui consiste à chercher une solution d’un **problème** en calculant d’abord les solutions de ses **sous-problèmes**.
- 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.

193

Des sous-problèmes au problème 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’énumérer les problèmes à partir des plus petits, autrement on risque de re-calculer plusieurs fois la solution du même problème ce qui mène à une complexité exponentielle.

195

Problème 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-problèmes 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-problèmes 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.

194

Programmation dynamique appliquée au problème 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.

196

Exercice (suite)

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

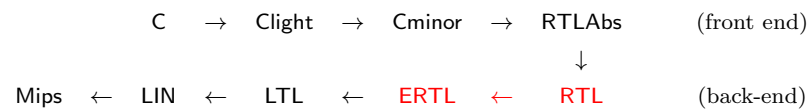
197

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 **générateurs de générateurs de code** (dans l'esprit de Yacc).

198

Back-end : de RTL à ERTL



199

Plan du cours

- Explicitation des conventions d'appel Mips.

200

sum en RTL (rappel)

```
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)
```

201

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**.

203

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'explicitier les conventions d'appel avant l'allocation de registres en gardant une **représentation abstraite** des opérations qui manipulent le pointeur à la pile.
- Ces opérations seront explicitées seulement au dernier pas de compilation (de LIN à Mips).

202

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
```

204

```

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)

```

205

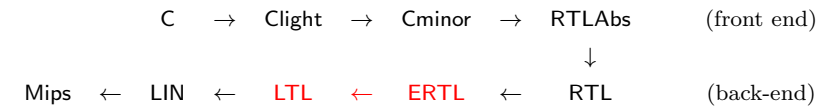
Plan du cours

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

207

Back-end : de ERTL à LTL

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



206

Point fixes et Définitions (co-)inductives

208

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**.

209

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 **fini** a un sup. Montrez que :
 - tout sous-ensemble d'un treillis **fini** a un inf.
 - tout treillis fini est **complet**.

211

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 $f(x) = x$.

210

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 ∞ , $0 < 1 < 2 < \dots < \infty$. Montrez que toute fonction monotone f sur cet ordre admet un point fixe.

212

Exercice (points fixes par itération)

- Soit (L, \leq) un treillis **fini** 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'**itération** n -**fois** de f sur x ($f^0(x) = x$).
- Montrez que **le plus petit point fixe** de f s'exprime par
$$\sup\{f^n(\perp) \mid n \geq 0\}$$
- Énoncez et démontrez la propriété pour **le plus grand point fixe**.

213

Sommaire

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

Treillis fini 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 fixe** est le \sup de l'itération (dénombrable) de la fonction à partir du plus petit élément.

215

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

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

- 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

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

214

Définitions 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*.

216

- 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 :

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

217

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 $\forall y \in X$ ($(x \equiv (y + 1)) \text{ mod } 2$).

Est-ce une bonne définition ?

219

Exercice (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 défini inductivement**.

218

Exercice (définition co-inductive (1/2))

- La notion de **définition 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 $M = (Q, \Sigma, q_0, F, \delta)$ un **automate fini déterministe** 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. $\forall a \in \Sigma$ $(\delta(a, q), \delta(a, q')) \in R$.
- 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 ?

220

Exercice (définition 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 définition**.
- **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 ?

221

Analyse de vivacité

223

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 fixe**.

222

Exemple

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

```
0 :  c := M[addc]    (on lit de la mémoire)
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 ?

224

Étape 1 : analyse de la vivacité 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 redéfinie** le long du chemin.
- Cette définition est un cas particulier de **définition inductive** et peut être effectivement calculée par une **itération finie**.

225

Étape 2 : coloration du graphe d'interférence

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 **interfèrent**).

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).

227

- Le **résultat de l'analyse de vivacité** est résumé dans le tableau suivant :

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

226

Étape 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( $R_1$ )
```

228

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**.

229

Définition formelle vivacité

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

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

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

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

$$\begin{aligned} Live_0(n, n') &= \emptyset \\ Live_{k+1}(n, n') &= Uses(n') \cup \left(\bigcup_{n'' \in suc(n')} Live_k(n', n'') \setminus Def(n') \right) \\ Live(n, n') &= \bigcup_{k \geq 0} Live_k(n, n') \end{aligned}$$

231

Analyse de vivacité

- 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 **affectées** 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') .

230

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 **finie**.

232

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

$$f(L)(n, n') = Uses(n') \cup \left(\bigcup_{n'' \in suc(n')} L(n', n'') \right) \setminus Def(n')$$

– 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'itérations**.

– 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.

233

– Dans ce cas, l'**équation logique** devient :

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

– On remarquera que dans ces équations il n'y a **pas de négation 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.

235

Vers un calcul efficace de la vivacité

– Le calcul de la vivacité peut être **spécialisé pour chaque variable**.

– Dans ce cas, on obtient une **équation booléenne** :

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

– On utilise les **abréviations** suivantes :

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

234

Solution de systèmes d'équations booléennes monotones

– Étant donné un système d'équations en forme canonique, on construit un **graphe dirigé étiqueté** où :

– Les **variables** sont les **noeuds du graphe**.

– Le noeud x est **étiqueté** avec \bigwedge (ou \bigvee) 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**.

236

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 booléen** $\hat{st} : V \rightarrow \{0, 1\}$ comme suit :

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

237

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 **linéaire** dans le **nombre d'arêtes** du graphe.
- Dérivez un borne supérieur pour la **complexité du calcul de la vivacité**.
- Adaptez l'algorithme pour qu'il calcule **la plus grande solution**.

239

- Au début A est l'ensemble des 1 à 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 prédécesseurs)
      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**.

238

Variations sur l'analyse du flot de données

- Le calcul de vivacité est un exemple parmi d'autres d'analyse de flot 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.

240

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 $R(n) = \{n'\}$ et 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 .

241

Exercice

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

243

Élimination 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!)

242

Exercice (dominance)

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

- Soit $G = (N, A, r)$ un graphe dirigé 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 $n = r$.

244

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 :

$$A = \{ (1, 2), (1, 3) \quad (2, 4) \quad (3, 4) \\ (4, 5) \quad (5, 6), (5, 7) \quad (6, 1) \} .$$

On présentera le résultat par une matrice 7×7 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

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

245

6. On suppose maintenant que le graphe en question est **connecté** (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 :

$$1 \rightarrow 2; \quad 2 \rightarrow 3, 4; \quad 3 \rightarrow; \quad 4 \rightarrow 2, 5, 6; \quad 5 \rightarrow 7, 8; \quad 6 \rightarrow 7; \\ 7 \rightarrow 11; \quad 8 \rightarrow 9; \quad 9 \rightarrow 8, 10; \quad 10 \rightarrow 5, 12; \quad 11 \rightarrow 12$$

247

4. Donnez un algorithme **itératif** 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. Pour valider votre intuition, 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 4×4 à valeurs en $\{0, 1\}$ où $D(n, m) = 1$ si et seulement si n domine m .

246

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.

248

Coloration du graphe d'interférence

et Allocation de registres

249

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).

251

Grphe d'interférence

On rappelle que le **graphe d'interférence** 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 $x, y \in Live(n, n')$.

250

- 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 Hanken en 1977 à l'aide d'un ordinateur) dit qu'un **graphe planaire** peut toujours être colorié avec 4 **couleurs**.

252

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.

253

Exercice (décomposition)

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.

255

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 Hanken ont montré en 1977 qu'un graphe planaire peut toujours être colorié avec 4 couleurs).

254

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**.

256

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 **Sélection**.
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 **itère 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 **itère Simplification**.

257

Répétition 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 sont écriture.

Va à Construction

259

Sélection 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 marqué** alors il a au plus $k - 1$ noeud adjacents et donc on peut le colorier (par l'exercice sur la k -coloration). On **itère Sélection**.
- Si le noeud est **marqué** alors soit il est coloriable et dans ce cas on **itère Sélection** soit il ne l'est pas et dans ce cas on **va à Répétition**.

258

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]
```

260

Références

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.

261

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) :
$$e ::= id \mid op(e, e)$$
- Notre objectif est de compiler une expression comme une suite d’affectations élémentaires en utilisant **le moins de registres possible**.

263

262

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

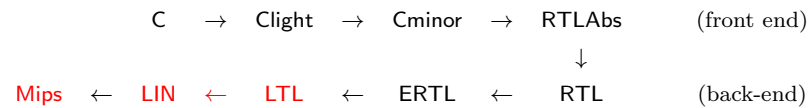
$$\begin{aligned} r1 &:= M[add_x]; r2 := M[add_y]; r3 := M[add_z]; \\ r2 &:= r2 * r3; r1 := r1 + r2; \end{aligned}$$

- Proposez un **algorithme** qui effectue cette affectation.
- Votre approche produit-elle toujours une **solution optimale** par rapport au nombre de registres utilisés ?

264

Back-end : de LTL à LIN à Mips

Plan du cours



- Linéarisation de code.
- Génération de code assembleur Mips

265

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
  
```

267

266

```

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
  
```

268

Compilation de LTL à LIN

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

269

- A partir d'une énumération l_1, \dots, l_n des noeuds, proposez une fonction \mathcal{C} qui étant donné le noeud l_i et le code linéarisé pour l_{i+1}, \dots, l_n produit le code linéarisé pour $l_i \dots, l_n$.
- Un **critère de qualité** est que le code linéarisé ne contient pas un saut non-conditionné à l'instruction suivante.

271

Exercice (sur la linéarisation)

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

$l : \text{op} \rightarrow l'$ (opération ou appel)
 $l : j \rightarrow l'$ (saut non-conditionné)
 $l : \text{bj} \rightarrow l', l''$ (saut conditionné)
 $l : \text{ret}$ (retour)

- D'autre part les instructions linéarisées ont la forme (l'étiquette est optionnelle) :

$l : \text{op}$ (opération ou appel)
 $l : j \rightarrow l'$ (saut non-conditionné)
 $l : \text{bj} \rightarrow l'$ (saut conditionné)
 $l : \text{ret}$ (retour)

270

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
```

272

Compilation de LIN à Mips

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

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

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 Vm (sans étiquettes!) dans laquelle les instructions de branchement $\text{branch}(k)$ et $\text{bge}(k)$ prennent **deux adresses consécutives** 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 **expansées** et
 2. **recalcule les offsets** des instructions de branchement.

277

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.

279

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

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

278

2. Montrez que votre système de contraintes a toujours une solution (laquelle?) et qu'il est possible d'ordonner les vecteurs $\{1, 2\}^n$ de façon à ce que le calcul de la solution optimale se réduise au calcul d'un plus petit point fixe.

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.

280

3. On suppose maintenant avoir construit un **graphe dirigé** G tel que :

- Les noeuds correspondent aux instructions de branchement.
- Il y a une arête du noeud i au noeud j si une expansion du noeud j affecte le offset de l’instruction i .

Par ailleurs, on associe à chaque noeud i une valeur nof_i (**new offset**) qui à la terminaison de l’algorithme doit correspondre à la valeur de l’offset pour l’instruction i . On appelle **degré** du graphe le nombre d’arêtes qui peuvent pointer vers un noeud. Donnez une borne au degré de G en fonction de b .

281

Récupération automatique de la mémoire

283

4. Donnez la structure d’un algorithme qui calculerait la solution optimale en s’appuyant sur le graphe G et déterminez sa complexité asymptotique en temps.

Suggestion : on peut s’inspirer de l’algorithme de ‘propagation des 1’ pour la solution de systèmes d’équations booléennes monotones.

282

Plan

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

284

Problématique

- Le code exécutable généré par un compilateur est un processus qui tourne au dessus d'un système d'exploitation.
- Le processus dispose d'un certain *segment de mémoire virtuelle* qui doit être géré de façon économique.
- Les machines virtuelles des langages de programmation courants (C, Java, ML, ...) distinguent *trois zones de mémoire* :

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 prévisible*.

285

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 Analyse statique (cf. ML-Kit) : peut être *incomplet* (mais *efficace*).

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.

287

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 .

286

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 *récupérable* si elle n'est pas accessible à partir des racines.

288

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).

289

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 récursifs*, 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.

291

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.

290

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
  end
```

292

Exercice (facile)

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

Rappel : considérez 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.

293

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 :

$$c_1R + c_2H$$

où c_1, c_2 sont des facteurs constants, c_1R est le coût du marquage et c_2H est le coût du ramassage du tas.

- Le *coût par cellule récupérée* est :

$$(c_1R + c_2H)/(H - R)$$

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

295

Implémentation ramassage

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

Init : $p :=$ 'lower address of heap';

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

294

- 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.

296

Comptage des références (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 à la taille du tas*.
- Cet arrêt peut être inacceptable pour des programmes qui doivent respecter des *contraintes de 'temps réel'*.
- La méthode du comptage des références règle *partiellement* ce problème.

297

Exemple

cell address	env field in cell	counter field in cell
<i>x</i>	<i>y</i>	<i>n</i> ₁
<i>y</i>	<i>nil</i>	<i>n</i> ₂
<i>p</i>	<i>nil</i>	<i>n</i> ₃

On suppose que le programme comprend l'instruction :

$$x.env := p$$

299

Ingrédients du comptage des références

- 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*.

298

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.

```
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;
p.count := p.count + 1
```

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

300

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

cell address	env field in cell	counter field in cell
<i>r</i>	<i>p</i>	1
<i>p</i>	<i>x</i>	1
<i>x</i>	<i>y</i>	2
<i>y</i>	<i>x</i>	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 ?

301

Récupération 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 mémoire* 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 *fragmentée* 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 *récupération par copie*.

303

- 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 *interlacée* avec l'exécution du programme.
- Donc même sans structures cycliques, il convient de voir le champ compteur d'une cellule comme une *borne supérieure* au nombre de pointeurs à la cellule.
- En pratique, on utilise plutôt des 'méthodes incrémentales' basées sur des *algorithmes concurrents*.

302

Ingrédients de la récupération 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 construit une *copie isomorphe* dans un segment initial de la zone 'to_space'

304

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 *marquée* 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'.

305

Implémentation 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'.

307

Coût de la récupération 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 .

306

- 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
```

308

– Soit r la racine du graphe ‘from_space’. On exécute :

```

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

while  $scan < next$  do
  begin
     $\forall$  pointer field  $f$  in the cell pointed by  $scan$  do
       $scan.f := Fwd(scan.f)$ ;
    increment( $scan$ );
  end

```

309

Exemple récupération 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

311

- 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 $next$.

310

Exercice

- 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 linéaire 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.

312

- 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, \dots dénotent des informations significatives mémorisées dans les blocs occupés.

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

Avant

Après

313

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 $x.f0$ et $x.f1$.
- 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).

314

```

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;
  else
    next := current; current := pred;
    if current = nil then STOP;
    i := current.done; pred := current.fi;
    current.fi := next; (2) current.done := i + 1;

```

315

- 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.

316

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 (→ 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, nil, 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

317

Notion de récupération 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.

Propriété souhaitée

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

319

Notion de récupération générationnelle

- Les cellules allouées récemment ont beaucoup *plus de chance* d'être récupérables.
- Division de la mémoire en *générations*.
- On récupère plus souvent les *dernières* 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.

318

Problème

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	$A \rightarrow C$	B	A, B
2	$A \rightarrow C \leftarrow B$	A, B	-
3	A	$C \leftarrow B$	B
4	$A \rightarrow C \leftarrow B$	B	A
5	$A \rightarrow C$	B	-
			A, B

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

320

Approche à trois couleurs

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.

321

Sommaire

Méthodes 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,...

Élaborations

- notion de génération pour améliorer la chance de récupération,
- marquage à 3 couleurs pour une récupération concurrente.

323

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 à 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).

322

Lecture conseillée

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

324

Conclusion

Notions d'intérêt général Sémantique opérationnelle.

Transformations de programmes. Analyse du flot de données et du contrôle. Gestion de la mémoire.

Un compilateur optimisant pour C Spécification formelle.

Gestion du contrôle, de l'environnement et de la mémoire.

Structuration des optimisations et du back-end. Dépendance du processeur.

Un projet de génie logiciel Manipulation d'un programme

d'une certaine taille. Modularisation et Intégration. Test et

Mesures de performance.