Representat° Intermediaire

Entre le haut niveau et le bas niveau.

Un peu de vocab

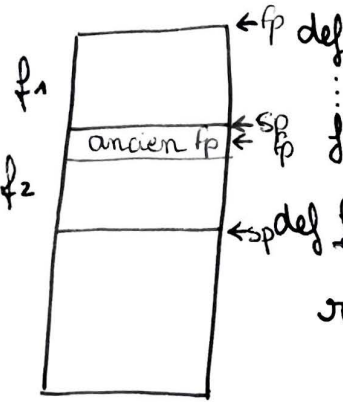
front end (nom) front-end (adjective)
↳ analyse du language, generat° de l'ast, optimisation.
middle end → traitement generique
back end → product° de l'assembleur, allocat° des registres

registre vectoriel → registre + grand qui permet des collect° d'operat°

Passer par un langage intermediaire:
Avantages:
- Insert° / Suppress° de nouveaux langages
- Correct° de bugs facile.

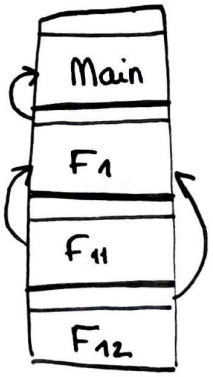
Il est trop complexe de trouver une intersect° entre les langages sans faire de preference. De ce fait on a non pas 1 mais plutôt une succession de representations intermediaire, leur nombre varie selon le langage
HIR → High Intermediade Representat°
LIR → Low " "
Chaque representat° est en realité une transformation de l'AST.
qui linearise 1 feature par 1 feature (ex: recurt°, object, λ express°).
Il est pratique que LIR et HIR € à la ⁂ famille de langages (fonctionnel, impératif, ...)
Choix du IR:
- fonctionel ? Imperatif ? Basé s/ les registres ? / la pile ?
Variables temporaires: stocke les étapes de calcul.
Triplet, quadruplet → 2 ou 3 registres.

Block d'activat°: stocke les variables locales, certains registres, adresse de retour.



←fp def f1(...){
⋮
f2(...); }
←sp fp
←sp def f2(...){
return; }

(See the slides 60 → 64) w/ alloca. ≠ solut°:



Static Link:
Ici on imagine que tt les liens statique pointe vers leurs parents syntaxique.

Règles: on ne doit avoir accès qu'aux variables de ses parents (pr pouvoir gerer les variables en echappement).

fp = frame pointer
sp = stack pointer

On va traduire ≠mt des instruct° selon leur utilisation
lorsqu'on essaye de linéariser un programme on va regarder si
deux s/s arbres de e sont commutables pour pouvoir limiter la sauvegarde
dans des temporaires. Un s/s arbre est dit "pur" si les variables manipulées
sont const ou en lecture seule. Si l'un des deux s/s arbres est non
pur on ~~fa~~ sauvegarde les var ds des temporaires. ex:
$t + (t := 42, 0) \Rightarrow t_0 = t \quad t = 42 \quad t_0 + 0$.

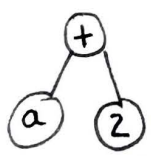
Quand on a une condition on va sauter sur un bloc si faux et sur
un autre si vrai.
Comment linéariser ça ?
On va construire des blocs independants commençant par un label et finissant
par un jump.
Lorsqu'on organise les blocs après un cjump on colle en priorité le label false
Si ni le label true ni le label false ne sont dispos on crée un jump
sur le label false.
Le but est de realiser le moins de jump possible.

Cisc : avantage : très haut niveau.
desavantages : les instructions ont des coûts variables.
Revoir l'histoire des cycles d'execution en Système 1.
risc → : désavantage : on peut vouloir utiliser une variable avant
qu'elle ai été executée.

Sélection d'instruction. ⟹ REVOIR
For now, on considere qu'on a un nombre illimité de registres
On va colorer les noeuds de l'arbre.

---

a+2 : addi t_1, a, 2.
on ajoute le resultat dans une temporaire



revoir les couleurs.

Liveness Analysis
Control Flow Analysis.
A l'étape precedente on a produit un texte et on veut construire
le graph correspondant au flow du texte. Pour ça on va devoir
fournir des metadonnées pr les sauts et les labels. On colore les
arcs en f° de l'utilisat° des variables.

Coloration des arcs :

On part d'une utilisat° de a. Il faut que ses predecesseurs la fournisse. Si le pred utilise a on remonte, si le pred defini a on colore l'arc et on s'arrête. on colore l'arc.

Ici il est compliqué de travailler avec des arcs.

Du coup on va essayer de raisonner par rapport aux noeuds.

On va regarder les lives in et lives out de chaque noeuds (flèches entrantes colorees et flèches sortantes colorées). Et les lives out du noeud n sont l'union des lives in de ses successeurs.

Pour chaque noeuds on a des use et des define.

in = use ∪ (out / def)

out = ∪ in des successeurs

On arrête qd on obtient 2 fois de suite les mm in and out.

On commence par le bas du flow pour faire moins d'iterations.

ALLOCATION DES REGISTRES.

1er à ê coloré = de + contraint.

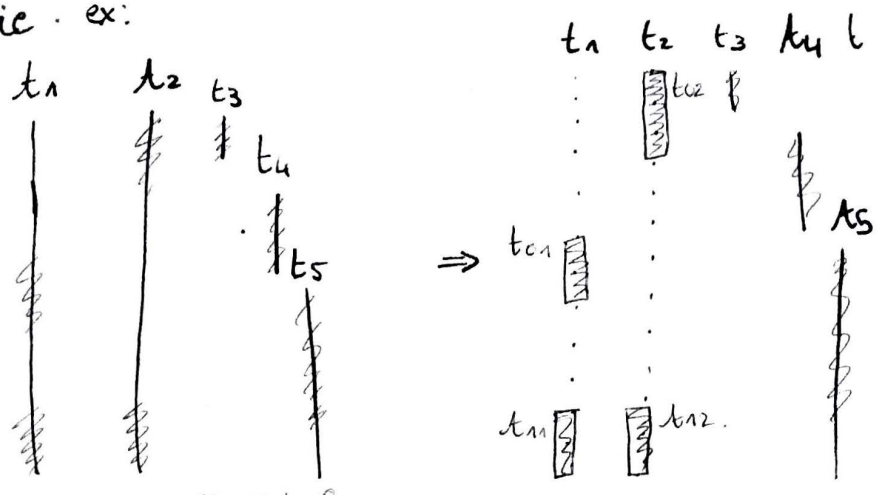
Puis colore les noeuds ac une strat?

Pb: pas assez de registre.

↳ ~~on utilise des temporaires pr~~

on va essayer d'utiliser la stack.

Lorsqu'on met une variable sur la stack on restreint sa durée de vie. ex:



↳ zone d'utilisat° de la variable.

Comment choisir quelle est la variable qu'on met s/ la stack.

⇒ Priorité aux variables des boucles.

Si on a $t_1 := t_2$ on dit qu'ils ont un lien d'affinité et on va essayer de les mettre dans le m registres

On a le droit de fusionner deux noeuds si ils ont des voisins identx et si leur d° n'est pas significatifs (degre > nb de registres - 1)

Sur le graphe les variables st liées aux variables ~~avec~~ qui qui vivent en m tmps qu'elles. On a une sorte de pile qui va nous permettre de faire la colorat°. On empile un noeud n'ayant pas un d° significatif puis une fois tt les variables empilées on colore en dépilant.
On scinde les noeuds fusionnés lors de l'empilement devant la colorat°.

Nœud Précolorés des registres matériels sont précolorés.

Lorsque la colorat° n'est pas possible on met une var s/ la stack.
Pour la choisir on fait un tableau de score d'utilisat° comptant les colonnes suivante : ① définit°/utilisation en dehors d'une boucle
② " " dans une boucle
③ degre du noeud
- Total ⇒ = (① + ② x10) / ③

On sacrifie la variable avec le + petit score.
Une fois le graphe coloré on remplace les variables par leur registres puis on va retirer tt les lignes inutile (ex : r3 := r3).

SCHEDULING

L'ordre des instructions peut influencer le temps d'execution s'il existe des dépendances entre ces dernières.
Si b dépend de a, a doit ê executé avant b.

≠ types de dépendance :

1 : écrit dans @, b.
2 : add dans c, @ et d.) RAW

1 : a = b + c
2 : a = d + e) WAW → si on inverse a n'aura pas la bonne valeur pr la suite.

Graph de dépendance

On crée un noeud par instruction puis on relie les noeuds selon leurs dépendance RAW, WAR et WAW.
On obtient un graph qui peut ê traité comme un diagramme de Gantt de poid des arcs dépend du type de dépendance (0 = WAW et WAR) (2 = RAW) (1 = RAR).

On obtient le chemin critique du graph.
On execute en premier les noeuds sans predecesseurs et qui ont le plus gros poid.
Lorsque 2 noeuds on la même poids on prend celui dont les predecesseurs sont le + éloigné.

→ Revoir le loop unrolling dans les slides.
⇒ traiter 5 par 5 permet de remplir les cycles de gène
⇒ Mais pk on deroule pas les 100 instruct°? Pas assez de registres.
⇒ On peut avoir besoin d'un prologue et d'un epilogue pour caller la taille du tableau ex: ~~to~~ for (1 to 99) ⇒ for (1 to 4) + for (5 to 95) + for (95 to 99)

* Le compilateur doit également se soucier des acces memoire. Il va essayer de traiter les données contiques en même temps pour qu'elles soient chargées en même temps dans le cash.

```
A[1024]; B[1024]
for (int i=0; ... )
   A[i] = B[i] * a
```

⇒

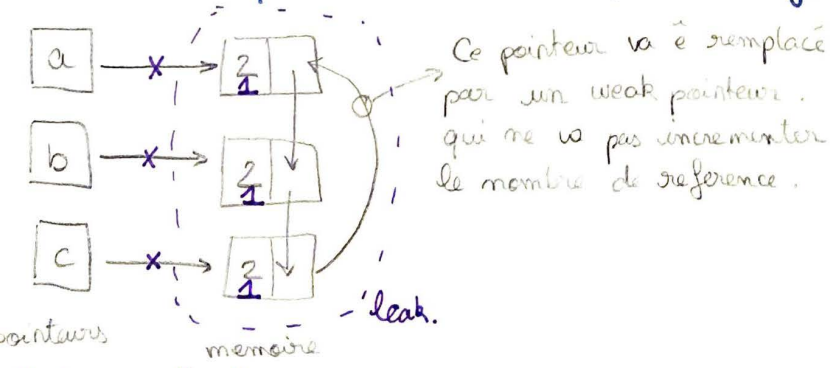
```
Struct {int A; int B}
R[1024]
for (int i ... )
   R[i].A = R[i].B * 42.
```

On a donc obtenu du code assembleur qu'on transforme en binaire et qu'on passe au microprocesseur.

GARBAGE COLLECTOR.

Permet d'eviter les pointeurs qui se baladent, les oublis de free et les reflexions sur le "quand liberer mon objet"

≠ strats:
Count pointer: on compte les refs des objets et on supprime l'objet quand le compteur passe à 0. Pb: References cyclique.



Autre methode:
Pour trouver les objets on prend un Root Set et on marque recursivement ses successeurs. Ceux qui ne sont pas marqué sont inateignable et donc desalloué. On marque à l'aide d'un booleen et on set alternativement à T ou F

Pb: segmentat° (comme un malloc sans merge).
On pourrait regrouper les zones mémoires utilisées mais ⚠ d'actualiser les references. On va copier les " " " bout à bout dans une zone 2 et après l'update des refs on clear la zone 1. On swap les deux zones et on alloue dans la partie liberee de zone 1.

La mémoire est divisée en 3 parties : Eden, Survivors, Tenured.
Les variable les + vieilles vont descendre progressivement vers des espaces mémoire ou le GC passe moins souvent.