ANALYSE SEMANTIQUE

VERIFICATION DE TYPE

Qu’est ce qu’un type ?

- Définition sujette à débat…

- *Dénote d’un groupement de valeurs*
- *Et d’un ensemble d’opérations sur ces dernières*

- Les erreurs de type arrivent quand les opérations sont effectuées sur des valeurs ne les supportant pas

Types de vérification de type

- Vérification de type statique
  - Analyse effectuée au moment de la compilation
  - Prouve l'absence d'erreur avant l'exécution

- Vérification de type dynamique
  - Vérification des opérations en cours d'exécution
  - Plus précis que la vérification statique mais souvent bien moins efficace

Les systèmes de typage

- Les systèmes de typage forts
  - N'autorisent jamais une erreur de typage
  - Ex: Java, Python, ADA, Haskell…
- Les systèmes de typage faible
  - Autorisent des erreurs de typage à l'exécution
  - Ex: C
    - Ex : manipulation des pointeurs, débordements, cast…

La guerre des types

- Débat sans fin sur le meilleur système de typage
- Les systèmes de typage dynamique facilitent le prototypage, les systèmes de typage statique ont moins de bugs
- Les langages fortement typés sont souvent plus robuste, les faiblement typés sont souvent plus rapides…

Ingrédients du typage

- Un langage de termes
- Un langage de types
- Un système de typage
  - Quel type a quel terme ?

Termes et types

- Les termes dénotent de valeurs
  - On les évalue pour obtenir leur dénotation
- Les termes ont des types
- Evaluer ne change pas le type d'un terme
  - Les types sont des invariants
  - ➢ Vérification statique envisageable

Langage de termes

- Les expressions élémentaires
  - Constantes et variables
    - "douze", "12", 12, 0x12, douze
- Les expressions composées
  - Opérations, appels, accès structures, tableaux
    - 0x12+12, douze(12), douze.XII, douze[12]
- Les fonctions, procédures
- Les programmes…

Langage de types

- Pas forcément explicite dans le langage source
  - Si explicite, on peut garder la syntaxe abstraite, sinon en inventer une
  - Situation intermédiaire possible

- Lisp, Scheme
  - Complètement implicite
- ML, Javascript, Scala
  - Explicite optionnel, tous les types ont une expression
- C, C++, Java
  - Explicite obligatoire

Langage de types

- Exemple avec des constructions classiques
  - Langage procédural / fonctionnel

Types atomiques

- bool, int, float, char…
- unit
  - Type des termes avec une seule valeur possible
  - Utile en fonctionnel pour des fonctions ne renvoyant pas vraiment de valeur (void en C…).
- void
  - Le type des termes avec valeur pas représentable

Type tableau

- array(Ind, Val)
  - Retourne un terme de type *Val* quand on l'indexe avec un terme de type *Ind*

- Ex :
  - en C, float t[12] dénote array(int, float)

Le type fonction

- fun(From, To)
  - Retourne un terme de type *To*
  - Quand on l'appelle avec un terme de type *From*

- Ex : *fun(float, int)*
  - En C : int f(float)
  - En ML : int -> float

- Note : on pense à une fonction exécutable
  - Pas nécessaire : mémoïsation
  - Correspondance directe entre paramètre et résultat (cache)

Le type structure

- struct(A1:T1, …, Ai:Ti, …, An:Tn)
  - Retourne un terme de type Ti
  - Quand on lui applique le sélecteur Ai

- Ex: struct(i:int, f:float)
  - En C : struct { int i; float f } ;

Le type union

- union(A1:T1, …, Ai:Ti, …, An:Tn)
  - Retourne un terme de type Ti
  - Quand on lui applique le sélecteur Ai
- Différence avec structure ?
  - Structure : tous les sélecteurs sont définis tout le temps
  - Union : un seul sélecteur est défini à un moment donné
- Warning : un sélecteur mais lequel ???
  - Ex : C++ : std::variant => union avec vérification de type à l'exécution

Le type pointeur

- ptr(A)
  - Retourne un terme de type A
  - Quand on le déréférence

- Ex : ptr(int)
  - En C : int *

Expressions de type

- En C
  - struct list { char car ; struct list * suivant }
- Dénote
  - list : struct(car:char, suivant:ptr(list))

- En ML
  - (A->B)->(B->C)->(A->C)
- Dénote
  - fun(fun(A,B), fun(fun(B,C), fun(A,C)))

Curryfication

- fun(A, fun(B,C)) peut se lire fun(A,B,C)

- Il n'est pas nécessaire de prévoir un type de fonction n-aire
  - Ex : l'opérateur + sur les entiers : int X int -> int
    - i.e. fun(int, int, int)
  - Mais
    - (A,B)->C $\Leftrightarrow$ A->(B->C)
  - Donc + a le type fun(int, fun(int, int))

Système de typage

- Un système logique reposant sur
  - Des jugements
  - Des axiomes
  - Des règles de déduction

Jugement de type

- t : T
  - Le terme *t* a le type *T*

- Peut être manifestement vrai ou faux, ou demander vérification
  - *12.5 : int* ?
  - *12 : int* ?
  - *f(12) : int* ?

Règles de déduction

- Forme générale d'une règle

$$\frac{hypothèse1 \ldots hypothèseN}{Conclusion}$$

  - Si toutes les hypothèses sont vraies alors la conclusion l'est

- Axiome

$$\frac{}{Conclusion}$$

  - La conclusion est toujours vraie

Notion d'arbre de preuve

- Arbre de preuve
  - Nœuds : instances de règles de déduction
  - Racine : un jugement à prouver

- Analogie avec l'arbre de dérivation

- Un arbre de preuve est une preuve ssi toutes ses feuilles sont des axiomes

➢ *Un système de déduction est une grammaire de preuve*

Système de typage

Système de typage

=

Règles de déduction dont les jugements sont des types

Axiomes

- Liés aux constantes

$$\frac{}{Notation\ d'entier : int} \qquad \frac{}{Notation\ de\ booléen : bool}$$

…

- Les variables

$$\frac{}{ident(X) : T}\ si\ decl\ X\ T$$

Règles de déduction

- Tableau

$$\frac{t : array(T1, T2) \quad i : T1}{t[i] : T2}$$

- Fonction

$$\frac{f : fun(T1, T2) \;\; x : T1}{f(x) : T2}$$

LA règle de déduction de type

Règles de déduction

- Structure

$$\frac{x : struct(a1 : T1, \ldots, ai : Ti, \ldots, an : Tn)}{x.ident(ai) : Ti}$$

- Union

$$\frac{x : union(a1 : T1, \ldots, ai : Ti, \ldots, an : Tn)}{x.ident(ai) : Ti}$$

Règles de déduction

- Pointeurs

$$\frac{x{:}ptr(T)}{*x{:}T} \qquad \frac{x{:}T}{\&x{:}ptr(T)}$$

Quelques concepts

- Surcharge :
  - Symboles de type différent mais de même nom
    - + : fun(int, fun(int, int))
    - + : fun(float, fun(float, float))

      +, -, *, /, =, == sont des symboles très surchargés…
  - Les langages acceptant la surcharge (java, c++ etc…) acceptent cela pour les fonctions définies par l'utilisateur

Quelque concepts

- Programme bien typé
  - Un programme p est bien typé
    si un jugement *p : T* peut être prouvé.
  - Vérifier le bon typage d'un programme consiste à chercher une preuve
    - Dans les faits, c'est plus simple qu'il n'y parait (enfin…)

Quelques concepts

- Vérification de type statique / dynamique
  - *Statique* : la vérification de type est statique si elle est réalisée sans exécuter le programme (i.e. par le compilateur)
    - Ex : CAML, Scala
  - *Dynamique* : la vérification se fait à l'exécution du programme
    - Ex : Javascript
  - Combinaison possible : une partie en statique, l'autre en dynamique
    - Ex : Java

Quelques concepts

- Propriété du typage sain
  - Un système de typage est dit sain (*sound*) si un programme bien typé statiquement ne peut pas causer d'erreur de type dynamique

➢ Propriété recherchée mais rare
  - Ex : CAML, ML, Scala

Mise en œuvre

- En utilisant une grammaire attribuée
  - Utiliser deux attributs
    - Type : le type de l'expression
    - Ok : vrai si pas d'erreur de typage, faux sinon
  - *Vérification en cours d'analyse syntaxique*

- Sur un arbre de syntaxe abstraite
  - Même principe, deux attributs associés aux nœuds de l'arbre
  - *Vérification après l'analyse syntaxique*

- *Attention : dans la « vraie » vie, il faut aussi pouvoir émettre un message d'erreur pertinent… (avec numéro de ligne…)*

Ex: mise en œuvre en GA (1/3)

- Attributs
  - TS : table des symboles
    - isFunction : permet de savoir si un symbole est une fonction
    - type : retourne le type associé au symbole
  - Ok : vrai si le typage est correct
  - type : le type de l'expression

- Fonction utilisées
  - to(FunctionType) => retourne ne type de retour de la fonction
  - from(FunctionType) => retourne le type du paramètre de la fonction

Ex: mise en œuvre en GA (2/3)

- expr -> Symbol ‘(‘ expr1 ‘)’

{
expr1.TS = expr.TS;
expr.ok = expr1.ok && expr.TS.isFunction(Symbol) &&
from(expr.TS.type(symbol))==expr1.type;
expr.type = to(expr.TS.type(symbol));
}

$$\frac{f : fun(T1, T2) \quad x : T1}{f(x) : T2}$$

- expr -> var

{ var.TS=expr.TS; expr.type = var.type ; expr.ok = var.ok; }

$$\frac{}{ident(X) : T \text{ si decl } X\ T}$$

- expr -> cstInt

{expr.type = cst.type; expr.ok = true; }

$$\frac{}{Notation\ d'entier\ :\ int}$$

Ex : mise en œuvre en GA (3/3)

- De manière schématique la partie droite de la règle va apporter les hypothèses et la partie gauche sera étiquetée avec la conclusion

  – Si pas de règle de déduction applicable dans le contexte

  – Pas de jugement de type

  ➢ *Erreur de typage*

Pour aller plus loin…

- Dans certains langages
  - Déclaration de type absentes en partie…
    - Ex : fonction génériques / lambda fonctions en C++, Java…
  - …ou totalement
    - Ex : CAML

➢ Il faut reconstituer les déclarations en analysant le programme : **inférence de type**

Inférence de type

- Point de vue logique
  - Polymorphisme
    - Ex : $\forall T, length: fun(list(T), int)$
      length est une fonction retournant la longueur d'une liste quel que soit le type des éléments de cette liste
  - Inférence
    - *f* est utilisé dans *f(x), on en déduit que*

$$\exists T, f : fun(TX, T)$$

- Du point de vue opérationnel, on ajoute des variables de type :
  - $f: fun(T_X, var(T))$ où $var(T)$ est une variable de type

Retour sur l'exemple précédent

- expr -> Symbol '(' expr1 ')'

{

expr.ok = expr1.ok && expr.TS.isFunction(Symbol) ;

if (expr.ok

&& unifType(expr.TS.Type(Symbol),
makeFunction(Expr1.type, new varType))

then {expr.type = varType}

else {expr.type=error; expr.ok=false}

}

Procédure d'unification

- unifType(fun(int, int), fun(int, var(A)))
  - OK, var(A) <- int
- unifType(fun(int, int), fun(var(V), var(A)))
  - OK, var(V) <- int, var(A) <- int
- unifType(fun(int, float), fun(var(A), var(A)))
  - Not OK
- unifType(fun(var(X), var(X)), fun(int, var(A)))
  - OK, var(X) <- int, var(A) <- int

Procédure d'unification

- unifType(fun(var(X), var(X)), fun(var(V), var(A)))
  - OK, var(X) <- var(V), var(A) <- var(V)
- unifType(fun(var(X), var(Y)), fun(var(V), var(A)))
  - OK, var(X) <- var(V), var(A) <- var(Y)
- unifType(fun(var(X), var(X)), fun(fun(var(A), var(A)), var(A)))

➢ *L'unification dans le cas général est assez complexe à réaliser*

Conclusion

- Le type est vu comme une propriété
- Typage bien modélisé par déduction logique
- Vérification = recherche de preuve
  - facile à réaliser en GA
- Inférence = recherche de témoin de preuve (∃)
  - plus difficile, mais faisable

CODE INTERMÉDIAIRE

La sortie du front-end

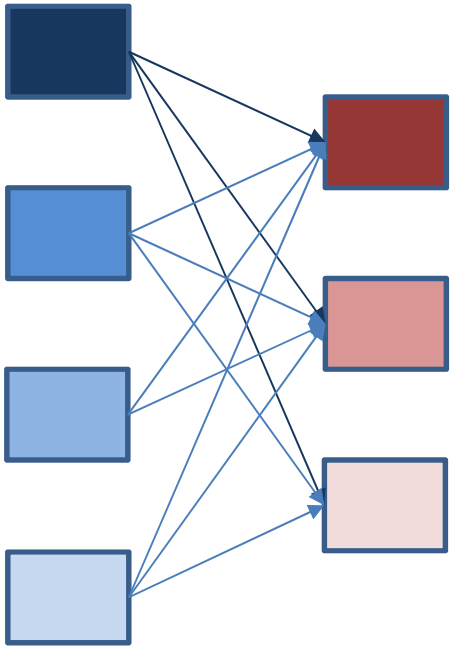
- Le programme lu appartient bien au langage
  - Lexicalement
  - Syntaxiquement
  - Sémantiquement

- Il est représenté par un arbre de syntaxe abstraite décoré
  - Table des symboles
  - Types
  - …

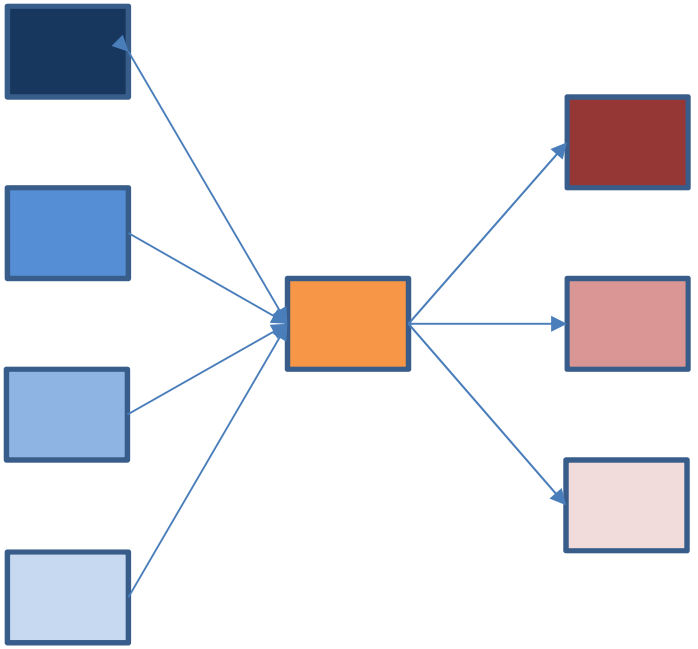
La production de code

- Etape 1 : produire du code intermédiaire pour une large cible
  - Viser des familles de machines cible
  - Sans contrainte de ressources
    - Registres sans limite
    - Types de données sans restriction
- Etape 2 : traduire le code intermédiaire en code exécutable pour une cible précise
  - Registres en nombre limité
  - Types de données : ceux de la cible
  - Modes d'adressage spécifiques

Intérêt du code intermédiaire



3 frontend
4*3 optimiseurs
4*3 générateurs de code



3 frontend
1 optimiseurs
3 générateurs de code

Différents types de code intermédiaire

- Le code intermédiaire doit être facile à produire, facile à transformer en code machine
  - Une sorte d'assembleur universel
  - Ne doit pas contenir de paramètres spécifiques à une machine / un processeur

- La nature du code intermédiaire est souvent dépendante de l'application
  - AST, Quadruplets, triplets…

- Une forme communément utilisée : Static Single Assignment form (SSA)
  - Facilite la transformation de programme
  - Ex : propagation de constantes…

Code intermédiaire

LE CODE 3 ADRESSES

Code 3 adresses

- Les instructions sont très simples
- Il y a une cible, au plus deux sources et un opérateur
- Les sources peuvent être des variables, des registres ou des constantes
- Les cibles sont des registres ou des variables

- Exemple : a+b*c-d/(b*c) se traduit en

```
t1 = b*c
t2 = a+t1
t3 = b*c
t4 = d/t3
t5 = t2-t4
```

Représentations

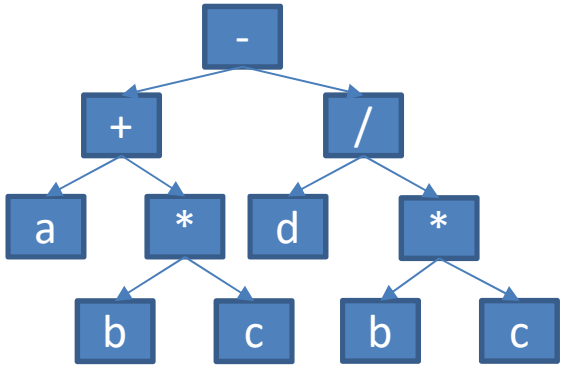
```
t1 = b*c
t2 = a+t1
t3 = b*c
t4 = d/t3
t5 = t2-t4
```

Quadruplets

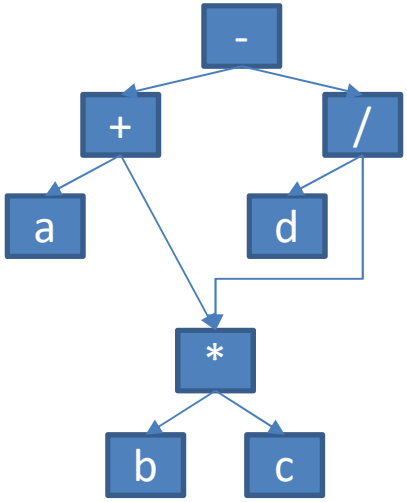
op	res	arg1	arg2
*	t1	b	c
+	t1	a	t1
*	t3	b	c
/	t4	d	t3
-	t5	t2	t4

Triplets

	op	arg1	arg2
1	*	b	c
2	+	a	(1)
3	*	b	c
4	/	d	(3)
5	-	(2)	(4)



Arbre de syntaxe abstraite



Graphe acyclique orienté

Code 3 adresses : instructions

- Instructions avec assignation
  - a = b
    - Copie b dans a
  - a = *unop* b
    - applique l'opérateur unaire *unop* sur b et stocke le résultat dans a
    - Ex : -, !, ~
  - a = b *biop* c
    - Applique l'opération *biop* avec b et c pour opérandes et stocke le résultat dans a
    - Ex : +, -, *, /, &, |, <, >
- Instructions de saut
  - goto L : saut inconditionnel au label L
  - if t goto L : si t est vrai sauter à L
  - Note : sur le if, il peut y avoir beaucoup de variantes
    - Ex : ifnz, ifz…

Code 3 adresses : instructions

- Les fonctions
  - func begin *<name>*
    - Déclare le début de la fonction nommée *<name>*
  - func end
    - La fin de la fonction
  - Return
    - Retourne à la fonction appelante
  - return *a*
    - Retourne la valeur *a* à la fonction appelante
  - param *p*
    - Place la paramètre p sur la pile
  - R = call *<name> n*
    - Appelle la fonction *<name>* avec les *n* paramètres en sommet de pile

Code 3 adresses : instructions

- Les tableaux
  - a = b[i]
    - Stocke la valeur de la *i*^{eme} case du tableau *b* dans *a*
  - b[i] = a
    - Stocke la valeur de *a* dans la *i*^{eme} case du tableau *b*

- Les pointeurs
  - a = &b
    - Stocke l'adresse de la variable b dans a
  - (*a)=b
    - Stocke la valeur de b à l'adresse désignée par a
  - a = (*b)
    - Stocke dans *a* la valeur à stockée l'adresse mémoire *b*

Code 3 adresses : remarques

- Généralement les variables du code 3 adresses sont
  - Les variables déclarées par l'utilisateur
  - Eventuellement quelques variables ajoutées par une transformation de code
  - ➢ *Toutes présentes dans la table des symboles*

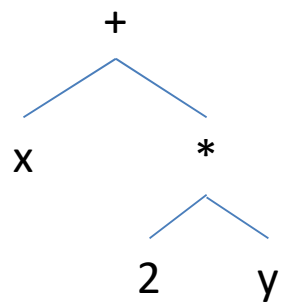
- Les résultats de calculs intermédiaires
  - Stockés dans des registres
  - Code 3 adresses : théoriquement une infinité de registres
  - Généralement un registre stocke un seul résultat
    - i.e. écrit une fois, lu *n* fois
  - ➢ *Facilite la transformations de code*

Représentation intermédiaire

GÉNÉRATION DE CODE 3 ADRESSES

Production de code 3A

- Entrée : un arbre de syntaxe abstraite
- Sortie : un code 3 adresses



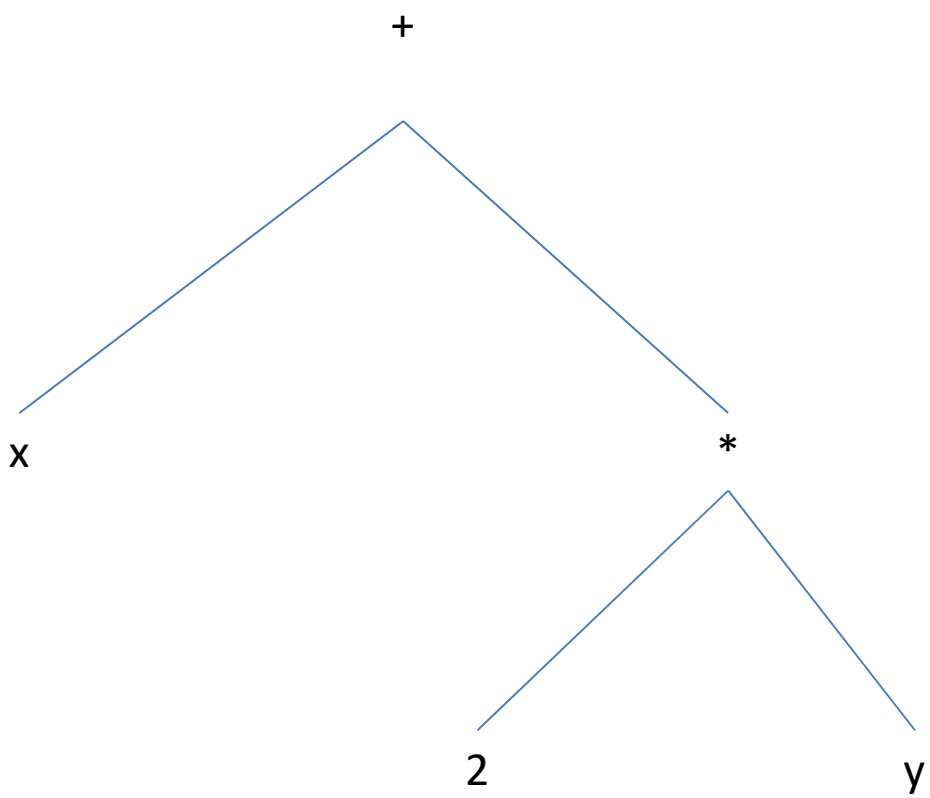
```
R0 = x
R1 = 2
R2 = y
R3 = R1 * R2
R4 = R0 +R3
```

Stratégie générale

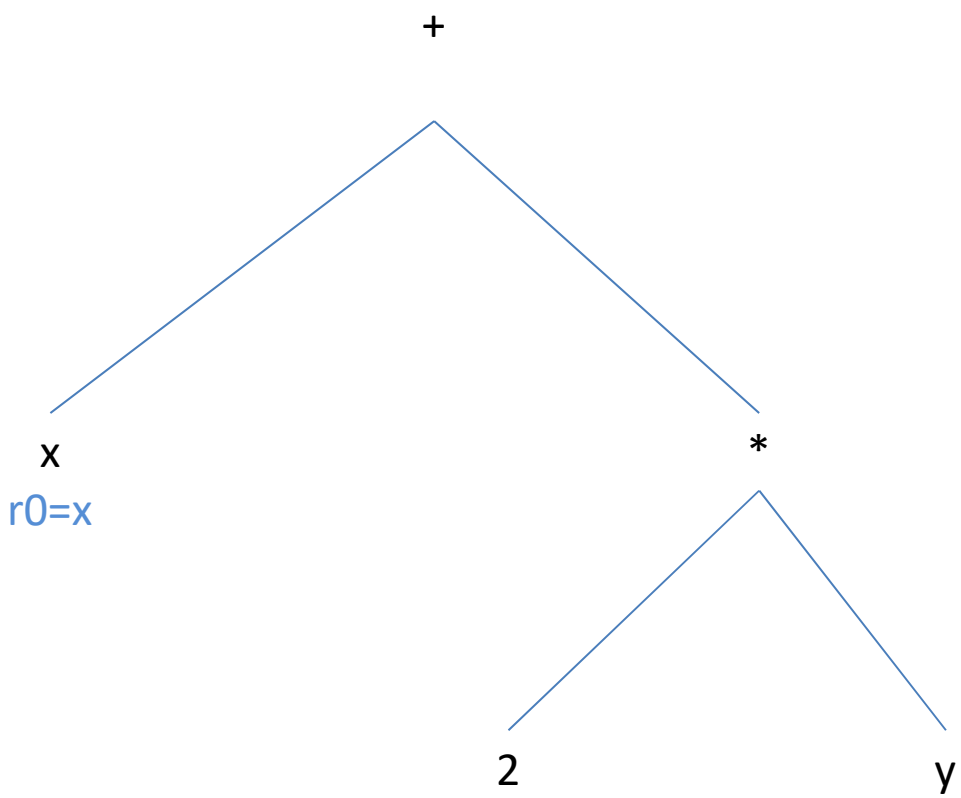
- Chaque nœud de l'arbre de syntaxe abstrait
  - Produit des bribes de code selon un patron prédéfini
    - *Sans avoir connaissance de ce que font les autres nœuds*
  - Peut composer un nouveau code à partir des codes des nœuds fils

- En utilisant une grammaire attribuée
  - Code = attribut synthétisé

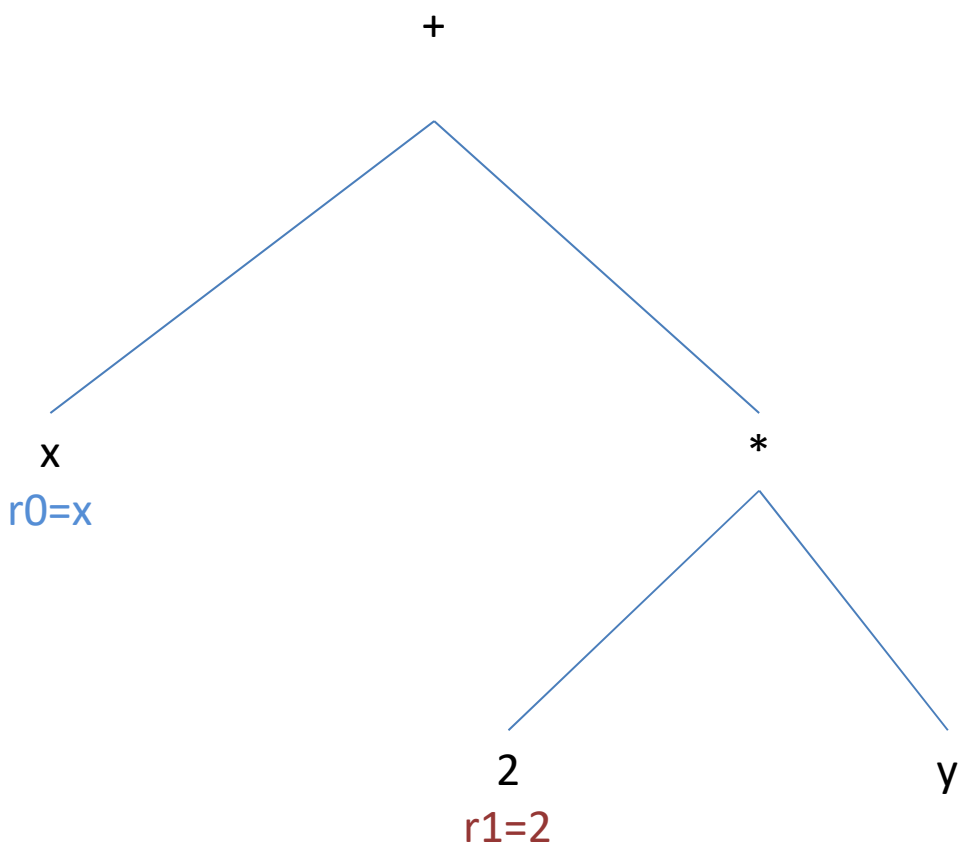
Retour sur l’exemple



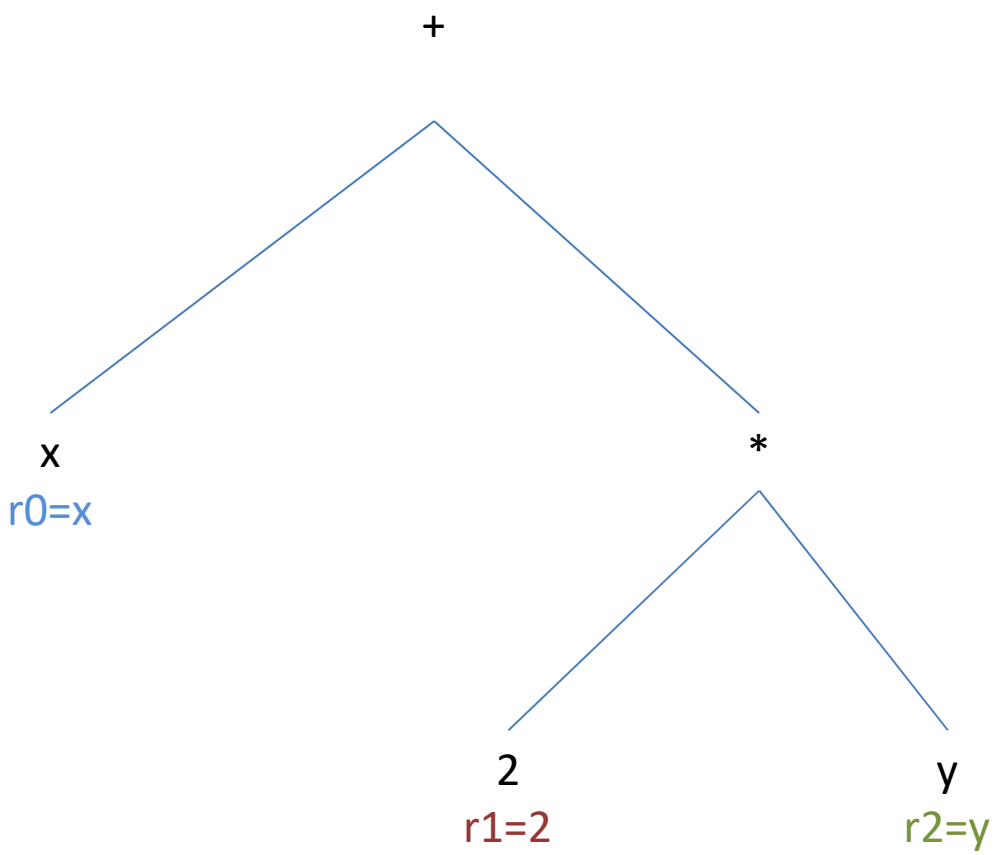
Retour sur l’exemple



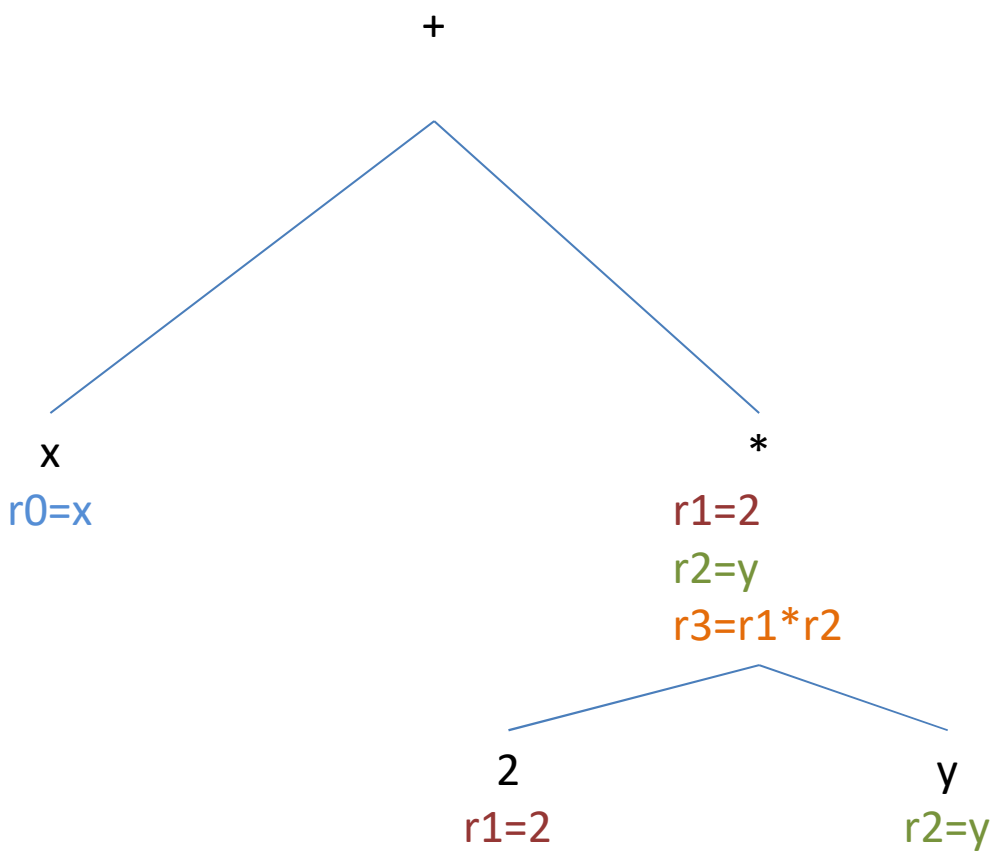
Retour sur l’exemple



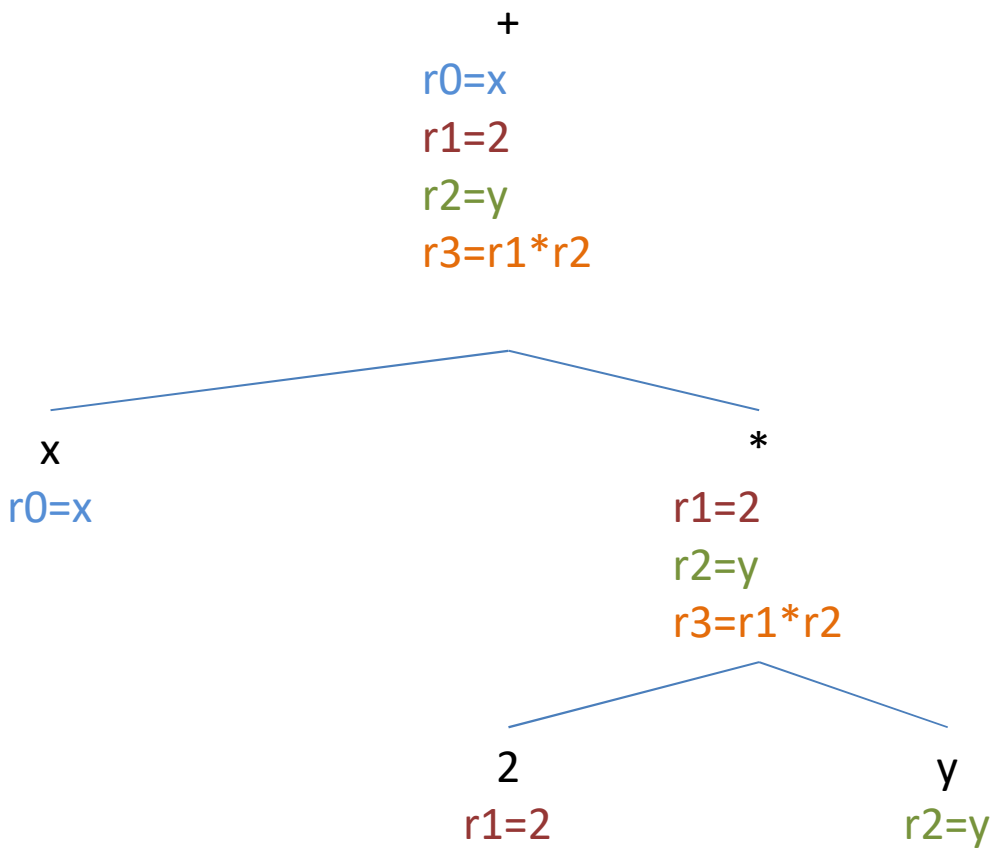
Retour sur l’exemple



Retour sur l'exemple



Retour sur l'exemple



Exemple de structure de contrôle

```
if(a<b)
{
    b=3*a;
}
else
{
    a=3*b
}
```

```
R0=a
R1=b
R2=R0<R1
ifz R2 goto false_label_0
R3=3
R4=a
R5=R3*R4
b=R5
goto end_if_label_1
false_label_0:
R6=3
R7=b
R8=R6*R7
a=R8
End_if_label_1:
```

Remarques

- Le code généré est naïf
  - Contient beaucoup de variables intermédiaires (registres)
  - Certains traitements sont « peu » utiles
- Des passes de transformation de code seront utilisées pour réduire l'utilisation des registres (entre autre)
  - Propagation de copie, élimination de code mort…
  - Algorithme d'allocation de registres

Code 3 adresses et runtime du langage

- Le langage source requiert des fonctionnalités
  - Souvent codées directement dans le langage cible
  - Lors de la production du code 3 adresses, utilisation de fonctions au nom prédéfini
  - Fonctions présentes dans la bibliothèque de runtime du langage

- Ex: l'opérateur new en C++
  - Partie intégrante du langage
  - Compilation en code 3 adresses

```
param size
R0 = call _new
```

  - Le registre R0 contient l'adresse de la mémoire allouée

TRANSFORMATION OPTIMISANTES

Le challenge de l'optimisation

- Un bon optimiseur
  - Ne doit pas changer le comportement observable du programme
  - Devrait produire un code intermédiaire aussi efficace que possible
  - Ne devrait pas prendre trop de temps de calcul

- Cependant
  - Les optimiseurs ratent souvent des optimisations « faciles » de part les limitations des algorithmes
  - Presque toutes les optimisations intéressantes sont NP-complètes voire indécidables…

Que pouvons nous optimiser ?

- Temps d'exécution
  - essayer de rendre le programme le plus rapide possible
  - Souvent au détriment de la mémoire et de la consommation énergétique
- L'occupation mémoire
  - Essayer de minimiser l'occupation mémoire du code produit
  - Souvent au détriment de la vitesse d'exécution et de la consommation énergétique
- La consommation énergétique
  - Essayer de réduire la consommation énergétique
  - Souvent au détriment de la vitesse et de l'occupation mémoire

Optimisation de code intermédiaire vs optimisation de code

- La distinction n'est pas toujours claire…
- Typiquement:
  - L'optimisation de code intermédiaire essaye de réaliser des simplifications valables pour toutes les machines
  - L'optimisation de code essaye d'améliorer les performances en connaissant les spécificités de la machine cible
- Parfois, des optimisations sont au milieu…
  - Ex : remplacer x/2 par x*0.5

Optimisations préservant la sémantique

- Une optimisation conserve la sémantique si elle ne change pas la sémantique du programme original

- Exemples
  - Suppression de variables temporaires inutiles
  - Calculer des valeurs qui sont connues au moment de la compilation
    - Ex : int a = 2*3 peut être remplacé par int a = 6
  - Sortir des invariants de boucles

- Contre exemple
  - Remplacer un tri à bulles par un quick sort…
    - Ne préserve pas la sémantique du programme original

Notion de bloc de base

- Un bloc de base est une séquence d'instructions de code intermédiaire telle que
  - Il y a exactement un point d'entrée à la séquence et s'il y a un contrôle entrant dans la séquence, il entre au début de cette dernière
  - Il y a exactement un endroit où le contrôle quitte la séquence et cet endroit est la fin de la séquence

➢ La séquence d'instruction s'exécute toujours en groupe

Graphe de flot de contrôle

- Un graphe de flot de contrôle est un graphe contenant les blocs de base d'une fonction
- Un arc orienté signale que le contrôle peut passer de la fin d'un bloc de base au début d'un autre bloc de base
- Il y a un nœud dédié au début et à la fin de la fonction

Exemple : calcul de pgcd

```
        R0 = call _readInt
        a = R0
        R1 = call _readInt
        b = R1
loop:
        R2 = a
        R3 = b
        R4 = R2 % R3
        r = R4
        ifz r goto end_if
        R5 = a
        b = R5
        R6 = r
        a = R6
        goto loop
end_if:
        param b
        call _printInt
```

Exemple : calcul de pgcd
Découpage en bloc de base

```
        R0 = call _readInt
        a = R0
        R1 = call _readInt
        b = R1
```

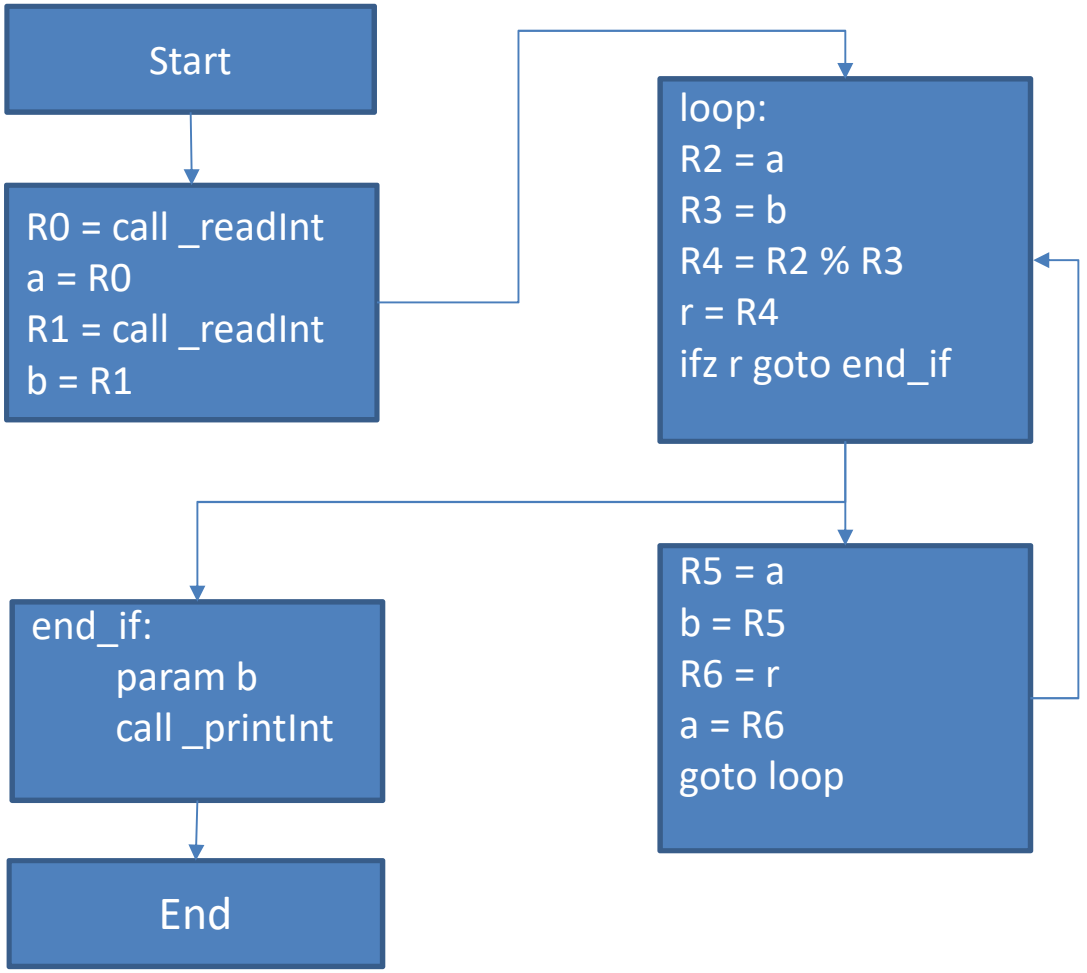
```
loop:
        R2 = a
        R3 = b
        R4 = R2 % R3
        r = R4
        ifz r goto end_if
```

```
        R5 = a
        b = R5
        R6 = r
        a = R6
        goto loop
```

```
end_if:
        param b
        call _printInt
```

Exemple : calcul de pgcd
Graphe de flot de contrôle

```
        R0 = call _readInt
        a = R0
        R1 = call _readInt
        b = R1
loop:
        R2 = a
        R3 = b
        R4 = R2 % R3
        r = R4
        ifz r goto end_if
        R5 = a
        b = R5
        R6 = r
        a = R6
        goto loop
end_if:
        param b
        call _printInt
```



Les types d'optimisation

- Une optimisation est dite *locale* si elle travaille sur un bloc de base
- Une optimisation est dite *globale* si elle travaille sur le graphe de flot de contrôle
- Une optimisation est dite *interprocedurale* si elle travaille sur les graphes de flot de contrôle de plusieurs fonctions
  - Non abordé dans ce cours…

Transformation optimisantes

OPTIMISATIONS LOCALES

Un code exemple

o1 = i ∗ 4

b = a – d

a = t[o1] + d

o2 = i ∗ 4

c = a – d

t[o2] = c

Elimination de sous expressions communes

```
o1 = i * 4
b = a – d
a = t[o1] + d
o2 = i * 4
c = a – d
t[o2] = c
```

```
o1 = i * 4
b = a – d
a = t[o1] + d
o2 = o1
c = b
t[o2] = c
```

Conditions d'application ?

Elimination de sous expression communes

- Remplacer la séquence suivante
  - (1) t = a op b
  - (2) …
  - (3) … = a op b
- Par
  - (1) t = a op b
  - (2) …
  - (3) … = t
- Si
  - t, a et b ne sont pas modifiés dans la suite d'instructions (2)
  - si (1) est toujours exécuté quand (3) l'est : toujours vrai dans un bloc de base

Propagation de copies

```
o1 = i * 4
b = a – d
a = t[o1] + d
o2 = o1
c = b
t[o2] = c
```

```
o1 = i * 4
b = a – d
a = t[o1] + d
o2 = o1
c = b
t[o1] = b
```

Conditions d'application ?

Propagation de copies

- Remplacer la séquence suivante :
  - (1) x = y
  - (2) …
  - (3) … x …
- Par
  - (1) x = y
  - (2) …
  - (3) … y …
- Si
  - x et y ne sont pas modifiés dans la suite d'instructions (2)
  - si (1) est toujours exécuté quand (3) l'est : toujours vrai dans un bloc de base

Elimination de code mort

```
o1 = i * 4
b = a – d
a = t[o1] + d
o2 = o1
c = b
t[o1] = b
```

```
o1 = i * 4
b = a – d
t[o1] = b
```

Conditions d'application ?

Elimination de code mort

- Remplacer
  - (1) x = E
  - (2) …
- Par
  - (2) …
- Si x n'est pas utilisé dans (2)

- La valeur rangée dans x en un point (P) n'est pas lue après le point (P)
  - On dit que x n'est pas vivante après ce point

- Attention : cet algorithme nécessite de connaitre les variables vivantes à la fin d'un bloc de base
  - La détermination des variables vivantes en fin d'un bloc de base nécessite l'analyse du graphe de flot de contrôle
  - Algorithme global…

Appliquer des optimisations locales

- Les optimisations précédentes prennent en compte des petites optimisations
  - La suppression des sous expressions communes élimine des calculs inutiles
  - La propagation de copies aide à identifier le code mort
  - L'élimination de code mort supprime des affectations inutiles

- Pour obtenir un effet maximum, il faut souvent appliquer ces optimisations plusieurs fois

Exemple

b = a * a

c = a * a

d = b + c

e = b + b

b = a * a

c = b

d = b + c

e = b + b

Elimination de sous expressions communes

Exemple

```
b = a * a
c = b
d = b + c
e = b + b
```

```
b = a * a
c = b
d = b + b
e = b + b
```

Propagation de copies

Exemple

```
b = a * a
c = b
d = b + b
e = b + b
```

```
b = a * a
c = b
d = b + b
e = d
```

Elimination de sous expressions communes

Autres optimisations locales

- Simplification de sous expressions constantes
  - Ex : x=4*5 peut se réécrire en x=20

- Simplification arithmétiques
  - $E * 2 \Leftrightarrow E + E \Leftrightarrow \text{shift-left}_1\ E$
  - $E * 7 \Leftrightarrow (\text{shift-left}_3\ E) - E$
  - $E / 4 \Leftrightarrow \text{shift-right}_2\ E$

Implémentation de l'optimisation locale

- Expressions disponibles
  - L'élimination des expressions communes (CSE) et la propagation de copie (CP) reposent sur les expressions disponibles à un endroit dans le code
  - Une expression est disponible si une variable contient la valeur de cette expression
  - Dans l'élimination des expressions communes on remplace une expression par la variable contenant sa valeur
  - Dans la propagation de copie, on remplace une variable par l'expression qui lui est associée

Recherche des expressions disponibles

- Initialement, aucune expression de disponible
- Quand une instruction du type a = b op c est exécutée
  - Toute expression utilisant a est invalidée
  - L'expression a = b op c devient disponible
  - (de même pour a = op b ou encore a = b)

- Algorithme : itérer sur le bloc de base depuis le début vers la fin en calculant les expressions disponibles

Exemple

a = b

c = b

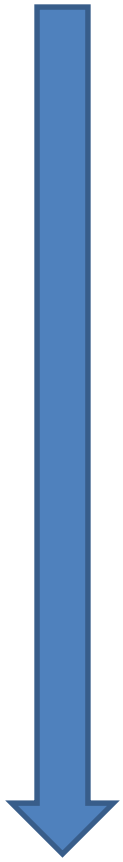
d = a + b

e = a + b

d = b

f = a + b

Expressions disponibles



{}

a = b

{a=b}

c = b

{a=b, c=b}

d = a + b

{a=b, c=b, d=a+b}

e = a + b

{a=b, c=b, d=a+b, e=a+b}

d = b

{a=b, c=b, d=b, e=a+b}

f = a + b

{a=b, c=b, d=b, e=a+b, f=a+b}

Elimination des sous expressions communes

{}

a = b

{a=b}

c = b -> c=a

{a=b, c=b}

d = a + b

{a=b, c=b, d=a+b}

e = a + b -> e = d

{a=b, c=b, d=a+b, e=a+b}

d = b -> d = a

{a=b, c=b, d=b, e=a+b}

f = a + b -> f = e

{a=b, c=b, d=b, e=a+b, f=a+b}

Elimination des sous expressions communes

a = b

c = a

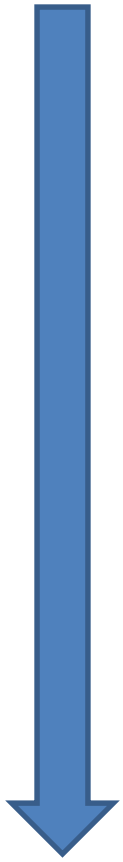
d = a + b

e = d

d = a

f = e

Expressions disponibles



{}

a = b

{a=b}

c = a

{a=b, c=a}

d = a + b

{a=b, c=a}

e = d

{a=b, c=a, e=d}

d = a

{a=b, c=a, d=a}

f = e

{a=b, c=a, d=a, f=e}

Propagation de copie

{}

a = b

{a=b}

c = a -> c = b

{a=b, c=a}

d = a + b -> d = b + b

{a=b, c=a}

e = d

{a=b, c=a, e=d}

d = a -> d = b

{a=b, c=a, d=a}

f = e

{a=b, c=a, d=a, f=e}

Propagation de copie

a = b

c = b

d = b + b

e = d

d = b

f = e

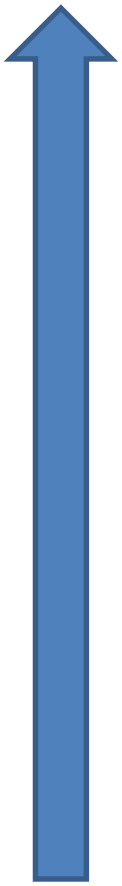
Les variables vivantes

- L'analyse correspondant à l'élimination du code mort repose sur la notion de variable vivante
- Une variable est dite vivante à un endroit du programme si sa valeur est lue avant d'être réécrite
- L'élimination de code mort repose sur la collecte des variables vivantes et la suppression des affectations concernant les variables mortes
- Analyse s'effectuant depuis la fin du bloc vers le début de ce dernier

Calcul des variables vivantes

- Initialement seul un sous ensemble des variables sont connues comme étant vivantes
  - Ex : valeur de retour pour une fonction…
- Lorsque qu'une instruction du type a = b op c est rencontrée
  - Avant cette instruction a n'est pas vivante puisque sa valeur va être réécrite
  - Avant cette instruction b et c sont vivantes car leur valeur est lue

Variables vivantes



{b}

a = b

{b}

c = b

{b}

d = b + b

{b,d}

e = d

{b,e}

d = b

{b,d,e}

f = e

{b, d}

Elimination de code mort

{b}

~~a = b~~

{b}

~~c = b~~

{b}

d = b + b

{b,d}

e = d

{b,e}

d = b

{b,d,e}

~~f = e~~

{b, d}

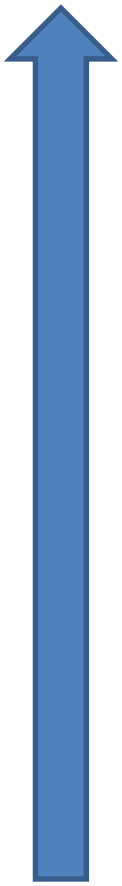
Elimination de code mort

d = b + b

e = d

d = b

Variables vivantes (2)



{b}

d = b + b

{b,d}

e = d

{b}

d = b

{b, d}

Elimination de code mort (2)

{b}

d = b + b

{b,d}

~~e = d~~

{b}

d = b

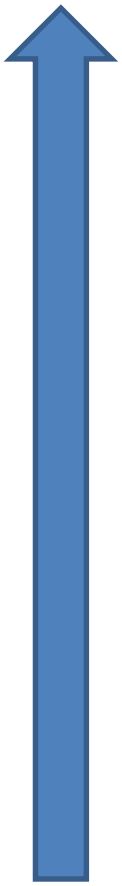
{b, d}

Elimination de code mort (2)

d = b + b

d = b

Variables vivantes (3)



{b}

d = b + b

{b}

d = b

{b, d}

Elimination de code mort (3)

{b}

~~d = b + b~~

{b}

d = b

{b, d}

Elimination de code mort (3)

d = b