INF889A
Analyse de programmes pour la sécurité logicielle

Simple and Efficient Construction of Static Single Assignment Form

---

Philippe Grégoire

2024-04-05

Université du Québec à Montréal

Le sujet choisi

- *Simple and Efficient Construction of Static Single Assignment Form*
- Par: Braun, Buchwald, Hack, Leissa, Mallon, and Zwinkau
- Publié en 2013 à la conférence *Compiler Construction*
- Présente un nouvel algorithme de construction de SSA

Les motivations

- Les RIs permettent de modéliser la sémantique d'un programme
- Permettent la réutilisation de programmes d'analyse pour différents languages
- Liées à mon sujet de mémoire sur l'analyse de programmes Java compilés
- Explorer différentes représentations intermédiaires (RIs)

Le programme

- Les représentations intermédiaires
- La forme SSA
- L'algorithme Braun et al.
- La conclusion

Les représentations intermédiaires

Les représentations intermédiaires

> *An IR is the data structure or code used internally by a compiler or virtual machine to represent source code. An IR is designed to be conducive to further processing, such as optimization and translation.*
> *Wikipedia, Intermediate Representation*

- *[...] is designed to be conducive to further processing [...]*
- Selon les langages ou les objectifs, une RI peut être préférable à une autre

Les formes de RIs

- Les RIs ont des formes qui leur confèrent des dés/avantages
  - *Three-address code* (3AC), e.g. Jimple
  - *Continuation-Passing Style* (CPS), e.g. Scheme, (historiquement) Haskell (GHC)
  - *A-Normal Form* (ANF), e.g. Core Scheme, plutôt théorique
  - *Static Single Assignment* (SSA), e.g. Java, LLVM
- Une forme de RI n'est pas une RI

En pratique - LLVM IR

- Utilisé par LLVM et un ensemble d'outils d'analyse, d'optimisation et transformation
- Une forme SSA

En pratique - LLVM IR

```
define i32 @foo(i32 %a) {
  switch i32 %a, label %def [ i32 42, label %case1 ]
case1:
    %x.1 = mul i32 %a, 2
    br label %ret
def:
    %x.2 = mul i32 %a, 3
    br label %ret
ret:
    %x.0 = phi i32 [ %x.2, %def ], [ %x.1, %case1 ]
    ret i32 %x.0
}
```

En pratique - VEX IR[1]

- Utilisée par `Valgrind`, `angr`, `codereason`, etc.
- Prétend avoir une forme quasi-SSA
- En pratique, plus près d'une forme 3AC

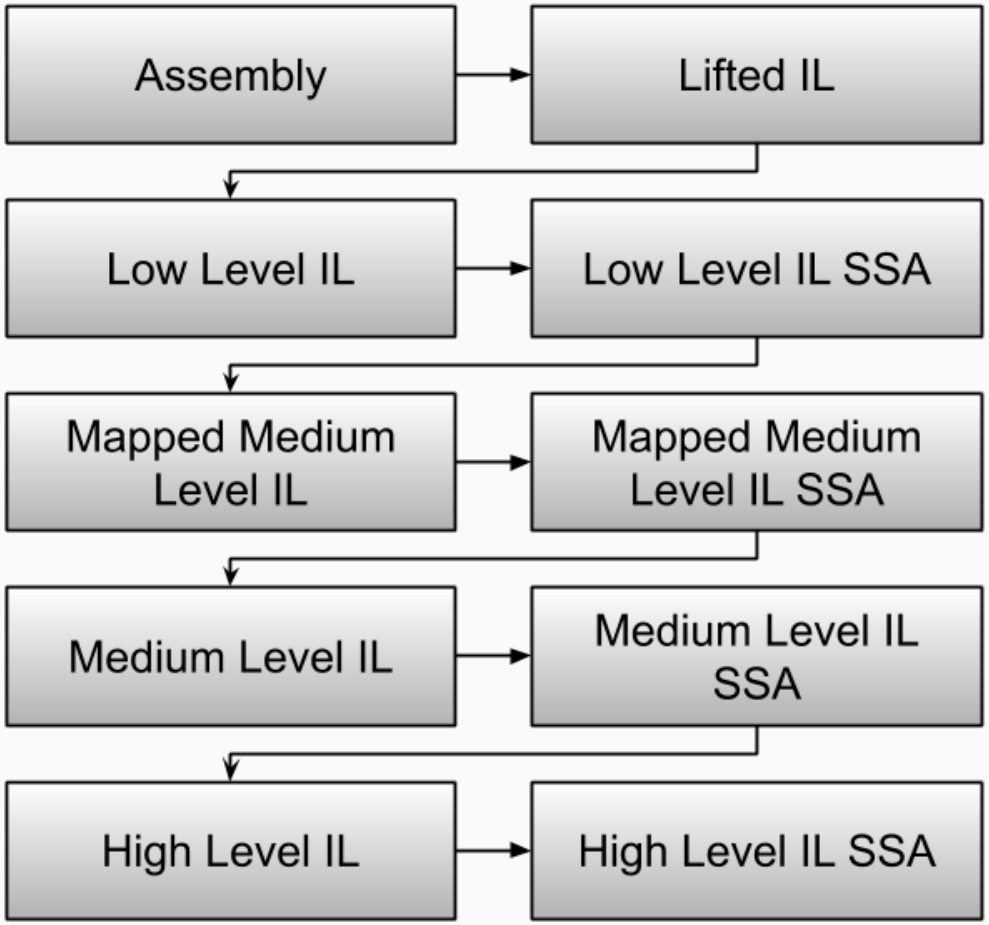
```
t0 = GET:I32(16)
t1 = 0x8:I32
t3 = Sub32(t0,t1)
PUT(16) = t3
PUT(68) = 0x59FC8:I32
```

[1] Il s'agit d'un exemple verbatim. Oui, `t2` est absent.

En pratique - BNIL

- *Binary Ninja Intermediate Language* (BNIL)
- Développé par et pour Binary Ninja
- Un outil de rétro-ingénierie de programmes compilés
- Une RI multi-niveau (*Multi-Level IL*, MLIL)
- Différentes formes: 3AC, SSA

En pratique - BNIL



En pratique - BNIL

Exemple de *Low-Level IL*

```
ebp = 0
r9 = rdx
rsi = pop
rdx = rsp {ubp_av}
rsp = rsp & 0xfffffffffffffff0
push(rax)
```

En pratique - BNIL

Exemple de *Low-Level IL SSA*

```
rbp#1 = zx.q(0)
r9#1 = rdx#0

# pop
rsi#1 = [rsp#0 {__return_addr}].q @ mem#0
rsp#1 = rsp#0 + 8

rdx#1 = rsp#1 {ubp_av}
rsp#2 = rsp#1 & 0xfffffffffffffff0

# push(rax)
[rsp#2 - 8 {stack_end}].q = rax#0 @ mem#0 -> mem#1
rsp#3 = rsp#2 - 8
```

En pratique - P-Code IR

- Ghidra *Pseudo-Code* (P-Code)
- Une forme 3AC
- L'objectif est de pouvoir modéliser n'importe quelle instruction.
- Aucune opération implicite (e.g. `OF` sur un `ADD`)

```
SHR         RAX,0x3f
$Ub7c0:4 = INT_AND 63:4, 63:4
$Ub7d0:8 = COPY RAX
RAX      = INT_RIGHT RAX, $Ub7c0
$U33e0:1 = INT_NOTEQUAL $Ub7c0, 0:4
```

En pratique - Soot

- Soot utilise trois (3) RIs: Baf, Jimple et Grimp
- En entrée: *bytecode* Java $\rightarrow$ Baf $\rightarrow$ Jimple $\rightarrow$ Grimp
- En sortie: Grimp $\rightarrow$ Baf $\rightarrow$ *bytecode* Java
- Shimple: une variante SSA de Jimple (3AC)

D'autres utilisations de RIs

- Les compilateurs Go, GCC, Java HotSpot, Swift, etc.
- Le décompilateur Boomerang
- Divers *runtime*: wasmtime, OpenJDK JVM, v8, Ruby, Python, etc.

Les flots

- Les compilateurs
  - Traduction du langage haut-niveau vers une RI
  - Analyse ou transformation de la RI
  - Traduction de la RI vers du code machine ou du *bytecode*
- Les programmes d'analyse
  - Traduction du langage machine vers une RI
  - Analyse ou transformation de la RI
  - ?

Les flots

- Les compilateurs
  - Traduction du langage haut-niveau vers une RI
  - Analyse ou transformation de la RI
  - Traduction de la RI vers du code machine ou du *bytecode*
- Les programmes d'analyse
  - Traduction du langage machine vers une RI
  - Analyse ou transformation de la RI
  - ?

On peut:

- se satisfaire des résultats de l'analyse
- traduire la RI résultante vers du code machine ou du *bytecode*
- traduire la RI résultante vers un langage haut-niveau

La forme SSA

L'origine

- *Static Single Assignment*: forme statique à assignation unique
  - Chaque variable se voit affectée une (1) seule fois
  - Les affectations subséquentes réutilisent le nom de la variable, avec un indicateur de version
- Inventée par Rosen, Wegman et Zadeck
  - En 1988, dans *Global Value Numbers and Redundant Computations*
  - Traite d'optimisation par élimination de code dupliqué
  - Passe par une RI de forme SSA pour atteindre l'objectif

Un exemple

```
int x, y;
x = 0;
x = 1;
y = x + 2;
```

```
x.1 := 0
x.2 := 1
y.1 := x.2 + 2
```

L'optimisation

Si la variable de gauche n'apparait pas à droite, on l'élimine.

Étape 1

```
x.1 := 0     # code mort
x.2 := 1
y.1 := x.2 + 2
```

Étape 2

```
x.2 := 1          # constante
y.1 := x.2 + 2 # <- propagation
```

Étape 3

```
x.2 := 1      # code mort
y.1 := 1 + 2 # somme de
              # constantes
```

Étape 4

```
y.1 := 3     # résultat
```

Jimple - À titre de comparaison

Programme Java

```
static void foo() {
  int x, y;
  x = 10;
  x = 11;
  y = x + 12;
}
```

Bytecode Java

```
 0: bipush   10
 2: istore_1
 3: bipush   11
 5: istore_1
 6: iload_1
 7: bipush   12
 9: iadd
10: istore_2
11: return
```

Jimple - À titre de comparaison

Jimple, la seconde passe de Soot, optimise déjà à l'étape 2 et 5.

À l'étape 1

```
unknown op0, op1, op2;

op0 = 10;
op0 = 11;
op1 = 12;
op2 = op0 + op1;
```

À l'étape 2

```
unknown op0;

op0 = 23;
```

À l'étape 5

```
int i0;

i0 = 23;
```

Les étapes 3 et 4 distinguent les paramètres des variables de pile, et type les variables finales, respectivement. À l'étape 2, l'optimisation a déjà commencée.

Des réflexions[2]

Selon l'objectif de notre analyse, on peut préférer une RI à une autre.

- Avons-nous besoin des types? Des détails de l'architecture?
- Est-ce que la transformation en RI a altérée l'expression originale et comment?
- Comment conserver des marqueurs comme `const`, `private` ou `static`? Sont-il importants?
- Quels sont les avantages d'une forme ou une autre?

[2] Ces réflexions s'adressent au lecteur sur le sujet en général.

Les branchements

```
void bar(int v, int c) {
  if (c != 0) {
    v = v + 1;
  }

  return v;
}
```

```
void bar(int v.0, int c.0) {
  if (c.0 != 0) {
    v.1 := v.0 + 1
  }
  // v.0 ou v.1 ?
  return ??????
}
```

- `v.0` peut ne pas être retourner
- `v.1` peut ne pas exister
- Un `else` hypothétique ne peut pas affecter `v.1 := v.0` sans créer une nouvelle affection à `v.1`

Les branchements

Pour permettre ce genre de construction, la fonction ϕ: *phi* exprime une affectation à partir de candidats. Selon l'origine de sortie, on utilise un symbole ou un autre.

```
void bar(int v, int c) {
  if (c != 0) {
    v = v + 1;
  }

  return v;
}
```

```
void bar(int v.0, int c.0) {
  if (c.0 != 0) {
    v.1 := v.0 + 1
  }
  # soit v.0, soit v.1
  return phi(v.0, v.1)
}
```

Les variations de SSA

- Problème: chaque branchement dans une boucle à le potentiel de créer plusieurs fonctions ϕ
- À quel point doit-on tolérer des ϕ inutile?
- Certaines variations de la forme SSA existent pour réduire leur nombre.
  - *Pruned SSA*, utilise une analyse de variables vivantes pour évaluer la nécéssité de ϕ
  - *Semi-pruned SSA*, idem, mais uniquement au niveau des *basic blocks*
  - *Block arguments*, les blocs deviennent paramétrés

L'algorithme de Cytron et al.

- Cytron et al. (1991), *Efficiently computing static single assignment form and the control dependence graph*
  1. Construire un *Control Flow Graph* (CFG)
  2. Déterminer les frontières de dominance
  3. Déterminer, pour chaque variables, l'emplacement des fonctions ϕ selon les frontières de dominance
  4. Renommer les variables pour satisfaire l'affectation unique
- Minimise le nombre de fonctions ϕ
- Utilise une analyse vers l'avant (*forward*)
- Utilisé par GCC[3], LLVM[4], etc.

[3] https://gcc.gnu.org/onlinedocs/gccint/SSA.html
[4] https://llvm.org/pubs/2010-08-SBLP-SSI.pdf

L'algorithme de Braun et al.

Le survol

- Critique de Cytron et al.
  - Force la génération d'un CFG
  - Doit calculer la *liveness* des variables et du code pour minimiser le nombre de ϕ
- À partir d'un arbre de syntaxe abstraite (AST)
- Produit une forme *pruned* ou minimale, sans dépendre d'autres analyses
- Utilise une analyse vers l'arrière (*backward*)
- Utilisé par wasmtime[5], Firm[6], etc.

[5] wasmtime/ssa.rs

[6] libfirm/loop.c

La numérotation des valeurs locales

On parcours le programme en ordre d'exécution et, lorsqu'un *basic block* est rencontré, ses valeurs sont numérotées.

```
a = 42                    v.1 := 42
b = a
c = a + b                 v.2 := v.1 + v.1
                          v.3 := 23
a = c + 23                v.4 := v.2 + v.3
c = a + d                 v.5 := v.4 + ?
```

- `d` n'est pas définie dans le *basic block*
- un bloc sans variable non-défini est marqué *rempli*
- on progresse aux successeurs uniquement lorsqu'un bloc est *rempli*
- s'il manque une définition, on cherche dans les valeurs globales

La numérotation des valeurs globales

```
x = ...
while (...)

{
  if (...)
  {
    x = ...
  }

}

foo(x)
```

```
x = ...
2:              ; p: 1, 9
  br 4, 11
4:              ; p: 2
  br 6, 9
6:              ; p: 4
  x = ...

9:              ; p: 4, 6
  br 2
11:             ; p: 2
  foo(x)
```

La numérotation des valeurs globales

- Le bloc 11 possède un (1) seul prédécesseur: 2
- On n'insère pas de ϕ dans le bloc 11 et on remonte

```
x.0 = ...
2:              ; p: 1, 9
  br 4, 11
4:              ; p: 2
  br 6, 9
6:              ; p: 4
  x.1 = ...
9:              ; p: 4, 6
  br 2
11:             ; p: 2
  foo(x.?)
```

```
x.0 = ...
2:              ; p: 1, 9
  br 4, 11
4:              ; p: 2
  br 6, 9
6:              ; p: 4
  x.1 = ...
9:              ; p: 4, 6
  br 2
11:             ; p: 2
  foo(x.2)      ; <--
```

La numérotation des valeurs globales

- Le bloc 2 ne définit pas x et a 2 prédécesseurs.
- On insère une ϕ sans opérandes dans le bloc.

```
x.0 = ...
2:              ; p: 1, 9

  br 4, 11
4:              ; p: 2
  br 6, 9
6:              ; p: 4
  x.1 = ...
9:              ; p: 4, 6
  br 2
11:             ; p: 2
  foo(x.2)
```

```
x.0 = ...
2:              ; p: 1, 9
  x.2 = phi(?, ?) ; <---
  br 4, 11
4:              ; p: 2
  br 6, 9
6:              ; p: 4
  x.1 = ...
9:              ; p: 4, 6
  br 2
11:             ; p: 2
  foo(x.2)
```

La numérotation des valeurs globales

- Le prédécesseur 1 a une (1) définition pour x
- On remonte les prédécesseurs de 9

```
x.0 = ...
2:              ; p: 1, 9
  x.2 = phi(x.0, ?) ; <---
  br 4, 11
4:              ; p: 2
  br 6, 9
6:              ; p: 4
  x.1 = ...
9:              ; p: 4, 6
  br 2
11:             ; p: 2
  foo(x.2)
```

La numérotation des valeurs globales

- Sans être défini, `x` existe dans le bloc 9
- 9 possède deux (2) prédécesseurs, on insère une ϕ

```
x.0 = ...
2:              ; p: 1, 9
  x.2 = phi(x.0, ?)
  br 4, 11
4:              ; p: 2
  br 6, 9
6:              ; p: 4
  x.1 = ...
9:              ; p: 4, 6
  x.3 = phi(?, ?) ; <---
  br 2
11:             ; p: 2
  foo(x.2)
```

La numérotation des valeurs globales

- Le prédécesseur 6 définit `x.1`

```
x.0 = ...
2:              ; p: 1, 9
  x.2 = phi(x.0, ?)
  br 4, 11
4:              ; p: 2
  br 6, 9
6:              ; p: 4
  x.1 = ...
9:              ; p: 4, 6
  x.3 = phi(x.1, ?) ; <---
  br 2
11:             ; p: 2
  foo(x.2)
```

La numérotation des valeurs globales

- On remonte 4 jusqu'à (revenir au) bloc 2

```
x.0 = ...
2:              ; p: 1, 9
  x.2 = phi(x.0, ?)
  br 4, 11
4:              ; p: 2 <-- unique prédécesseur
  br 6, 9
6:              ; p: 4
  x.1 = ...
9:              ; p: 4, 6
  x.3 = phi(x.1, x.2) ; <---
  br 2
11:             ; p: 2
  foo(x.2)
```

La numérotation des valeurs globales

- Les blocs sont remplis alors en revient.

```
x.0 = ...
2:              ; p: 1, 9
  x.2 = phi(x.0, x.3) ; <---
  br 4, 11
4:              ; p: 2
  br 6, 9
6:              ; p: 4
  x.1 = ...
9:              ; p: 4, 6
  x.3 = phi(x.1, x.2)
  br 2
11:             ; p: 2
  foo(x.2)
```

Les CFGs incomplets

Nous avons triché. On connaissait les prédécesseurs au moment de faire la traduction. Braun et al. construise la RI au fur et à mesure que le programme est traversé.

```
x = ...
while (...) {
  foo(x)   ; <--- x.?
  x = ...
}
```

Les CFGs incomplets

Il y a deux (2) attributs sur les blocs:

- Un bloc est **rempli** lorsque les variables sont complètes
  - Seul les blocs remplis ont des successeurs
  - Il permet aux successeurs d'y référer
- Un bloc est **scellé** lorsque tous ses prédécesseurs sont connus
  - Il permet au référant d'accéder à ses prédécesseurs

Les CFGs incomplets

```
1
2   x = ...
3   while (...)
4
5
6   {
7     foo(x)
8     x = ...
9
10  }
```

```
1   1:      ; R, S
2     x.0 = ...
```

Les CFGs incomplets

```
1
2   x = ...
3   while (...)
4
5
6   {
7     foo(x)
8     x = ...
9
10  }
```

```
1   1:      ; R, S
2     x.0 = ...
3   3:      ; p: 1, ? / R
4
5     br 6, 11
6   6:      ; p: 3
7     ...
8
9
10
11  11:     ; p: 3
12    ...
```

Le `while` implique deux (2) prédécesseurs.

Les CFGs incomplets

```
1
2   x = ...
3   while (...)
4
5
6   {
7     foo(x)
8     x = ...
9
10  }
```

```
1   1:      ; R, S
2     x.0 = ...
3   3:      ; p: 1, ? / R
4     ? = phi(?, ?) <-- 2. proxy
5     br 6, 11
6   6:      ; p: 3 / S
7     foo(?)      <-- 1. lookup
8     x.2 = ...
9     br 3
10
11  11:     ; p: 3
12    ...
```

- 6 est scellé et 3 est rempli
- On doit remplir 6 avant de continuer, mais 3 n'est pas scellé

Les CFGs incomplets

```
1
2   x = ...
3   while (...)
4
5
6   {
7     foo(x)
8     x = ...
9
10  }
```

```
1   1:      ; R, S
2     x.0 = ...
3   3:      ; p: 1, 3 / R, S <--
4     ? = phi(?, ?)
5     br 6, 11
6   6:      ; p: 3 / S
7     foo(?)
8     x.2 = ...
9     br 3
10
11  11:     ; p: 3
12    ...
```

- On complète les prédécesseurs de 3 pour le sceller.

Les CFGs incomplets

```
1
2   x = ...
3   while (...)
4
5
6   {
7     foo(x)
8     x = ...
9
10  }
```

```
1   1:      ; R, S
2     x.0 = ...
3   3:      ; p: 1, 3 / R, S
4     ? = phi(x.0, x.2) ; <--
5     br 6, 11
6   6:      ; p: 3 / S
7     foo(?)
8     x.2 = ...
9     br 3
10
11  11:     ; p: 3
12    ...
```

- On peut complèter ϕ de 3

Les CFGs incomplets

```
1
2   x = ...
3   while (...)
4
5
6   {
7     foo(x)
8     x = ...
9
10  }
```

```
1   1:      ; R, S
2     x.0 = ...
3   3:      ; p: 1, 3 / R, S
4     x.3 = phi(x.0, x.2)   <--
5     br 6, 11
6   6:      ; p: 3 / R, S
7     foo(x.3)   <--
8     x.2 = ...
9     br 3
10
11  11:     ; p: 3
12    ...
```

- ϕ a deux (2) paramètres différents
- on développe `x.3` et on propage

Plus encore

- Aucune ϕ morte
- Capacité d'éliminer les ϕ triviales (phi(x0,x0) ou phi(x0))
- Propose l'intégration d'optimisations

Une limitation?

- Quoi faire s'il existe plusieurs prédécesseurs vers un bloc?
  - `break` et `goto`

```
a:
while (...) {
  if (...) {
    break;
  }
  if (...) {
    goto a;
  }
}
```

La conclusion

- Un algorithme élégant pour construire une RI de forme SSA
- Une SSA prouvée *pruned* ou minimale
- Aussi, sinon plus, efficace que l'algorithme de Cytron et al.

Les références

SSA

- Braun et al. (2013). Simple and Efficient Construction of Static Single Assignment Form
- Braun et al. (2013). Simple and Efficient Construction of Static Single Assignment Form - Slides
- Rosen et al. (1988). Global value numbers and redundant computations
- Cytron et al. (1991). Efficiently computing static single assignment form and the control dependence graph
- CC 2013 - International Conference on Compiler Construction
- SSA Bibliography
- Awesome Program Analysis
- Static Single Assignment Book

Les références

Soot

- Soot
- Vallée-Rai et al. (1999). Soot - a Java Bytecode Optimization Framework
- Vallée-Rai et Hendren (2004). Jimple: Simplifying Java Bytecode for Analyses and Transformations

Les références

Autres

- Binary Ninja (2024). Binary Ninja Intermediate Language: Overview
- Valgrind - Writing a New Tool
- angr documentation - Intermediate Representation
- Three-Address Code IR
- P-Code Reference Manual