- Il faut parler de gated-SSA, des points communs & différences – soit dans la partie définition soit dans ouvertures à la fin du chapitre.
- Regrouper 1.2&1.3 / 1.4&1.5 avec éventuellement des paragraphes.
- Je voudrais voir des pseudo-code pour les algos au moins pour la destruction

CHAPTER 1

Psi-SSA Form

F. de Ferrière

Progress: 50%

Review in progress

1.1 Overview

In the SSA representation, each definition of a variable is given a unique name, and new pseudo definitions are introduced on ϕ instructions to merge values coming from different control-flow paths. ~~In this representation,~~ Each definition is an unconditional definition, and the value of a variable is the value of the expression on the unique assignment to this variable. This essential property of the SSA representation does not any longer hold when definitions may be conditionally executed. When the definition for a variable is a predicated operation, the value of the variable will or will not be modified depending on the value of a guard register. As a result, the value of the variable after the predicated operation is either the value of the expression on the assignment if the predicate is true, or the value the variable had before this operation if the predicate is false. We need a way to express these conditional definitions ~~whilst~~ keeping the static single assignment property.

~~ΨSSA~~ difficile à comprendre sans contexte mettre un exemple plutôt.

1.2 Definition + Construction

1 page

The use of predicated operations allows to remove control-flow instructions and have instead straight line code. The compiler can perform such a transforma-

Le but de cette section : illustrer, donner la sémantique de ψ-SSA & souligner différence entre φ & ψ ⇒ pas de dépendance de contrôle — Ne pas s'alourdir ni sur la construction, ni sur if-conversion — ref chapitre

tion, which is called an if-conversion optimization [?, ?]. A simple example of if-conversion is given in figure 1.1. We use the notation p? <exp> to say that <exp> is executed only if the predicate p is TRUE. + notation $\bar{p}$?

```
if(p)
   a1= op1;        p? a1= op1;
else
   a2= op2;        p̄? a2= op2;
x = Phi(a1,a2)     x = Psi(p?a1,p̄?a2)
```

if(p) a=op1 else a=op2

p? a=op1
$\bar{p}$? a=op2

Fig. 1.1 ψ-SSA representation

The SSA representation introduces ϕ operations at control-flow merge points. Each argument of a ϕ operation flows from a different incoming edge.

The ψ-SSA representation adds ψ operations. ψ operations are for predicated definitions what ϕ operations are for definitions on different control-flow edges. A ψ operation merges values that are defined under different predicates, and defines a single variable to represent these different values. A ψ operation is equivalent to a ϕ operation on which all the incoming edges would have been merged into a single execution path. Each argument of a ψ operation is now defined on a different predicate.

Ne pas insister sur if-conversion. Ça soulève trop de questions.

Bof

In figure 1.1, variables a and b were originally the same variable. On the left-hand side, the SSA construction renamed the two definitions of this unique variable into two different names, and introduced a new variable x defined by a ϕ operation to merge the two values coming from the different control-flow paths. On the right-hand side, an if-conversion algorithm transformed this code to remove the control-flow edges. It introduced predicated operations for the definitions of the variables a and b and turned the ϕ operation into a ψ operation. Each argument of the ψ operation is defined by a predicated operation. The intersection of the domain of the two predicates is empty and the value of the ψ operation is given by one or the other of its arguments, depending on the value of the predicate.

a = …
if(p) b = …
x = φ(a,b)

Dans cet exemple les prédicats sont disjoints et la sémantique de ψ est si p vrai alors a = a1 sinon a = a2. De fait la sémantique d'un phi autorise des prédicats non disjoints et ? de manière générale.

The ψ operations can also represent cases where variables are defined on predicates that are computed from independent conditions. This is illustrated in figure 1.2, where the predicates p and q are independent. During the SSA construction a unique variable was renamed into the variables a, b and c and the variables x and y were introduced to merge values coming from different control-flow paths. In the non-predicated code, there is a control-dependency between x and c, which means the definition of c must be executed after the value for x has been computed. In the predicated form of this example, there are no longer any control dependencies between the definitions of a, b and c. A compiler transformation can now freely move these definitions independently of each other, which may allow more optimizations to be performed on this code. However, the semantics of the original code requires that the definition of c occurs after the definitions of a and b. The order of the arguments in a ψ operation gives information on the original order of the definitions. We take the convention that the order of the arguments in a ψ operation is, from

Ainsi

a1 = op1
p? a2 = op2
a = ψ(a1, p?a2)

souligner abut d'écriture quand prédicat = vrai

≡ a1 = op1
if(p)
 a2 = op2
a = φ(a1, a2)

$$y = \psi(p_1?y_1, p_2?y_2, \ldots, p_n?y_n) \equiv \begin{array}{l} p_1?\ y = y_1 \\ p_2?\ y = y_2 \\ \vdots \\ p_n?\ y = y_n \end{array}$$

Ainsi l'ordre des opérandes d'un phi est important.

à la différence importante que pas contrôle-dépendance en ψ-SSA left to right, equal to the original order of their definitions, from top to bottom, in the control-flow dominance tree of the program in a non-SSA representation. This information is needed to maintain the correct semantics of the code during transformations of the ψ-SSA representation and to revert the code back to a non ψ-SSA representation.

Un Ψ a-t-il une garde? (oui à cause projection)

```
if(p)
   a = 1;            p? a = 1;
else
   b = -1;           p̄? b = -1;
x = Phi(a,b)            x = Psi(p?a, p̄?b)
if(q)
   c = 0;            q? c = 0;
y = Phi(x,c)            y = Psi(p?a, p̄?b, q?c)
```

Fig. 1.2 ψ-SSA with non-disjoint predicates

⚠ Rajouter que les définitions des arguments d'un ψ dominent le ψ. Souligner les équivalences d'écriture $\psi(a_1, p?a_2) \equiv \psi(\bar{p}?a_1, p?a_2)$

~~1.3 Construction~~

~~0.5 page~~ Non SSA predicated code → Ψ-SSA

Solutions ? ~~- On place d'abord les ψ & φ avec un ψ après chaque def prédiquée x=ψ(x, p?x) puis on renomme~~ ~~- on place les ψ à la volée au moment du renommage.~~

The construction of the ψ-SSA representation is a small modification on the standard algorithm to built an SSA representation.

Only the SSA renaming part of the algorithm needs to be modified. During the SSA renaming phase, basic blocks are processed in their dominance order, and operations in each basic block are scanned from top to bottom. On an operation, for each predicated definition of a variable, a new ψ instruction must be inserted just after the operation. For the definition of a variable x under predicate p, the ψ operation will take the form ~~x = Psi(p̄?x_1, p?x)~~, where x_1 is the current renaming of x before the definition, and p is the predicate used on the definition of x. Once this instruction is inserted, the normal renaming of the operation proceeds, renaming x into a new name x_2. When the renaming of the operation is completed, the algorithm continues on the next instruction, which will be a ψ operation if there was a predicated definition. The first argument of the ψ operation is already renamed and thus is not modified. The second argument is just renamed into the current renaming for x which is x_2. On the definition of the ψ operation, the variable x is given a new name x_3 which becomes the renaming for further references to the x variable.

ψ operations can also be introduced in an SSA representation by applying an if-conversion transformation, such as the one that is described in ??. Local transformations on control-flow patterns can also require to replace ϕ operations by ψ operations. ?

x renamed to x_1 up to here
$p?\ x = \dots$ | $p?\ x_2 = \dots$
$x_3 = \psi(x_1, p?x_2)$

$p?\ x = op$

$p?\ x = op$
$x = \psi(x_1, p?x)$

$p?\ x_2 = op$
$x_2 = \psi(x_1, p?x_2)$

$p?\ x_2 = o$
$x_3 = \psi(x_1, p?x_2)$

2 solutions :
- On place φ & ψ avec renommage local qd place un ψ. Puis renommage normal :
  $p?x = op \Rightarrow p?\ x' = op$, $x = \psi(x, p?op)$
- On place les ψ à la volée au moment du renommage (plus difficile à expliquer)

1.4 SSA algorithms + Psi-SSA algorithms.

~~1 page~~

With this definition of the ψ-SSA representation, conditional definitions on predicated code are now replaced by unconditional definitions on ψ operations. Usual algorithms that perform optimizations or transformations on the SSA representation can now be easily adapted to the ψ-SSA representation, without compromising the efficiency of the transformations performed. Actually, within the ψ-SSA representation, predicated definitions behave exactly the same as non predicated ones for optimizations on the SSA representation. Only the ψ operations have to be treated in a specific way. As an example, the constant propagation algorithm described in [?] can be easily adapted to the ψ-SSA representation. In this algorithm, the only modification is that ψ operations have to be handled with the same rules as the ϕ operations. Other algorithms such as dead code elimination [?], global value numbering [?], partial redundancy elimination [?], and induction variable analysis [?] are examples of algorithm that can easily be adapted to this representation.

mettre références chapitres du livre plutôt.

~~1.5 Psi-SSA algorithms~~

~~3 page~~

In addition to standard algorithms that can be applied to ψ operations and predicated code, a number of additional transformations can be performed on the ψ operations : ψ-inlining, ψ-reduction and ψ-projection.

lempl.

ψ-inlining will recursively replace in a ψ operation an argument that is defined on another ψ operation by the arguments of this other ψ operation. Example.

ψ-reduction will remove from a ψ operation an argument whose value will always be overridden by arguments on its right in the argument list, because the domain of the predicate associated with this argument is included in the union of the domains of the predicates associated with the arguments on its right. Example.

ψ-projection will create from a ψ operation a new ψ operations ~~for uses in operations guarded by different predicates. Each new ψ operation is created~~ as the projection on a given predicate of the original ψ operation. In this new ψ operation, arguments whose associated predicate has a domain that is disjoint with the domain of the predicate on which the projection is performed actually contribute no value to the ψ operation and thus are removed. Example pourquoi

ψ-projection can be performed when the variable is used further by a predicated op. The projection is done on the predicate of this use operation.

The ψ-SSA representation can also be used on a partially predicated architecture, where only a subset of the instruction set supports a predicate operand. The only impact of partial predication on the ψ-SSA representation is that when a ψ operation is created as a replacement for a ϕ operation, during if-conversion for example, some of its arguments may be defined by operations that cannot be predicated. In this case, the only constraint is that these non-predicated arguments can be safely speculated,

ref if-conversion chapter.

par exemple peut pas spéculer une division possiblement par 0.

which means executed under some conditions on which they would not have been executed otherwise. Although these definitions are speculated, their values are only meaningful under a given predicate that must be kept in the ψ operation.

```
if(p)
   a = ADD i, 1;                a = ADD i, 1;
else
   b = ADD i, 2;                b = ADD i, 2;
x = Phi(a, b)                   x = Psi(p?a, p̄?b)
```

a) before if – conversion b) Psi operation

Fig. 1.3 Psi-SSA for partial predication

on an architecture where...

Figure 1.3 shows an example where some code with control-flow edges was transformed into a linear sequence of instructions. ~~In this example, the~~ ADD operation cannot be predicated. The information represented in the ϕ operation by the control-flow edges is now present in the ψ operation by means of predicates.

However, even if the ADD operation can be predicated, it can be profitable to perform a predicate promotion optimization to reduce the number of computed predicate registers. Using the representation in figure 1.3 b), there can be one predicate associated with the definition of a variable, and there will be one predicate associated with the use of the variable in a ψ operation. The predicate associated with an argument in a ψ operation can be promoted, without changing the semantics of the ψ operation. By predicate promotion, we mean that a predicate can be replaced by a predicate with a larger predicate domain. This promotion must obey the two following conditions so that the semantics of the ψ operation after the transformation is valid and unchanged.

\emph.

laquelle? Exemple.

- Condition 0 : $p'_i \supseteq p_i$
- **Condition 1** For an argument in a ψ operation, the domain of the predicate used on the definition of this argument must contain the domain of the new predicate associated with this argument.
  ~~for the instructions~~ In other words, for :
  p? x = ...
     y = Psi(..., q?x, ...)
  ~~then~~ we must have :
     $q \subseteq p$
- **Condition 2** For an argument in a ψ operation, the domain of the new predicate associated with it can be extended up to include the domains of the predicates associated with arguments in the ψ operation that were defined after the definition for this argument in the original program.

Ca doit être une contrainte sur la sémantique d'un ψ ie dans partie 1.

On comprend rien

à moins que l'on s'interdise d'étendre le domaine du ψ (pour lequel ça reste très flou jusqu'à présent)

Je ne suis pas d'accord. Il faut que : $(p_i' - p_i) \cap \bigcup_{k=1..i-1} p_k \subset \bigcup_{k=i+1}^{n} p_k$

for an instruction

$$y = \mathrm{Psi}(p_1?x_1, p_2?x_2, \ldots, p_i?x_i, \ldots, p_n?x_n)$$

transformed to

$$y = \mathrm{Psi}(p_1?x_1, p_2?x_2, \ldots, p_i'?x_i, \ldots, p_n?x_n)$$

then

$$p_i' \subseteq \bigcup_{k=1}^{n} p_k$$

Ça implique que pour avoir condition 1 tu t'autorise à augmenter le prédicat de la définition. Tu dois en parler à ce moment là.

This ψ-predicate promotion transformation allows to reduce the number of predicates that need to be computed, and to reduce the dependencies between predicate computations and conditional operations. In fact, the first argument of a ψ operation can usually be promoted under the TRUE predicate, provided that speculation can be applied. Also, when disjoint conditions are computed, one of them can be promoted to include the other conditions, usually reducing the dependency height of the predicated expressions. The ψ-predicate promotion transformation can be applied during an if-conversion algorithm for example. A side effect of this transformation is that it may increase the number of copy instructions to be generated during the out of ψ-SSA phase, because of more live-range interference between arguments in a ψ operation, as will be explained ~~next~~ ~~later~~ later in Section...

??

Argh! Parce que en plus tu peux changer l'ordre des arguments d'un ψ qd ils overlappent pas! C'est une autre transfo ou ça fait partie de la promotion?

1.6 ~~Out of~~ Psi-SSA ⟷ Destruction

~~3 page~~

The ~~out of~~ SSA phase reverts an SSA representation into a non-SSA representation. This phase must be adapted to the ψ-SSA representation. The algorithm we present here is derived from the ~~out of~~ SSA algorithm ~~from Sreedhar et al. [?]~~.

This algorithm uses ψ congruence classes to create a conventional ψ-SSA representation. ~~We define the conventional ψ-SSA (ψ-CSSA) form in a similar way to the Sreedhar definition of the conventional SSA (CSSA) form. The congruence relation is extended to the ψ operations.~~ Two variables x and y are in ~~a ψ congruence relation~~ the same ψSSA web if they are referenced in the same ϕ or ψ function, or if there exists a variable z such that x is in a ψ congruence relation with z and y is in a ψ congruence relation with z. ~~Then we define a ψ congruence class as the transitive closure of the ψ congruence relation.~~ The property of the ψ-CSSA form is that the renaming into a single variable of all variables that belong to the same ψ congruence class, and the removal of the ψ and ϕ operations, results in a program with the same semantics as the original program. ok

- SSA webs plutôt
- Extend the notion of SSA web initially defined for φ to φ + ψ operations to derive the notion of conventional ψ-SSA. (ψ-CSSA)

Now, ~~look at~~ consider figure 1.4 to ~~examine~~ illustrate the transformations that must be performed to convert a program from a ψ-SSA form into a program in ψ-CSSA form.

citer chapitre plutôt.

Looking at the first example, the dominance order of the definitions for the variables a and b differs from their order from left to right in the ψ operation. Such code may appear after a code motion algorithm has moved the definitions for a and b relatively to each other. ~~We said that the semantics of a ψ operation is dependent on the order of its arguments, and that the order of the arguments in a ψ operation~~

Et promotion?

An ψSSA-web is a non empty minimal set of variables such that if two variables are referenced in the same φ or ψ function then they are in the same ψSSA web.

```
p? b = ...              p? b = ...              p? b = ...
   a = ...                 a = ...                 x = ...
                        p? c = b                p? x = b
   x = Psi(1?a, p?b)       x = Psi(1?a, p?c)
```

(a) **Psi – SSA form** (b) **Psi – CSSA form** (c) **non – SSA form**

```
   a = ...                 a = ...                 x = ...
                           d = a                   y = x
p? b = ...              p? b = ...              p? x = ...
q? c = ...              q? c = ...              q? y = ...
   x = Psi(1?a, p?b)       x = Psi(1?a, p?b)
   y = Psi(1?a, q?c)       y = Psi(1?d, q?c)
```

(d) **Psi – SSA form** (e) **Psi – CSSA form** (f) **non – SSA form**

Fig. 1.4 ψ-SSA ~~and~~ ψ-CSSA forms, and non SSA forme after destruction.

Fig 1.4(a) ~~is the order of their definitions in the dominance tree in the original program~~. In ~~this exam~~ple the renaming of the variables a, b and x into a single variable ~~will not~~ preserve the semantics of the original program. The order in ~~which~~ the definitions of the variables a, b and x occur must be corrected. ~~This~~ is done through the introduction of the variable c that is defined as a copy of the variable b, and is inserted after the definition of a. Now, the renaming of the variables a, c and x into a single variable will result in the correct semantics. predicated

Fig 1.4 (d) In the second example, the renaming of the variables a, b, c, x and y into a single variable will not give the correct semantics. In fact, the value of a used in the second ψ operation would be overridden by the definition of b before the definition of the variable c. Such code will occur after copy folding has been applied on a ψ-SSA representation. We see that the value of a has to be preserved before the definition of b, resulting in the code given for the ψ-CSSA representation. Now, the variables a, b and x can be renamed into a single variable, and the variables d, c and y will be renamed in another variable, resulting in a program in a non-SSA form with the correct ~~semantics~~ behavior.

We will now present an algorithm that will transform a program from a ψ-SSA form into its ψ-CSSA form. This algorithm is made of three parts.

- **ψ-normalize** This part will put all ψ operations in what we call a *normalized* form.
- **ψ-congruence** This part will grow ψ-~~congruence classes~~ web from ψ operations, and will introduce repair code where needed.
- **ϕ-congruence** This part will extend the ψ-congruence classes with ϕ operations. This part is very similar ~~to the Sreedhar algorithm~~.

We detail now the implementation of each of these three parts.

Sreedhar → chapitre destruction
Congruence → web.

1.6.1 *Psi-normalize*

We define the notion of *normalized-ψ*. When ψ operations are created during the construction of the ψ-SSA representation, they are naturally built in their normalized form. The normalized form of a ψ operation has two characteristics:

- The predicate associated with each argument in a normalized-ψ operation is equal to the predicate used on the unique definition of this argument.
- The order of the arguments in a normalized-ψ operation is, from left to right, equal to the order of their definitions, from top to bottom, in the control-flow dominance tree.

When transformations are applied to the ψ-SSA representation, predicated definitions may be moved relatively to each others. Operation speculation and copy folding may enlarge the domain of the predicate used on the definition of a variable. These transformations may cause some ψ operations to be in a non-normalized form.

PSI-normalize implementation.

~~A dominator tree must be available for the control-flow graph to lookup the dominance relation between basic blocks. The dominance relation between two operations in a same basic block will be given by their relative positions in the basic block.~~

Exemple .

Each ψ operation is processed independently. An analysis of the ψ operations in a top down traversal of the dominator tree reduces the amount of repair code that is inserted during this pass. We only detail the algorithm for such a traversal.

For a ψ operation, the argument list is processed from left to right. For each argument arg_i, the predicate associated with this argument in the ψ operation and the predicate used on the definition of this argument are compared. If they are not equal, a new variable is introduced and is initialized just below the definition for arg_i with a copy of arg_i. This definition is predicated with the predicate associated with arg_i in the ψ operation. Then, arg_i is replaced by this new variable in the ψ operation.

Que ce passe-t-il si le prédicat du ψ est plus grand que le prédicat de l'opération?

Then, we consider the dominance order of the definition for arg_i, with the definition of the next argument in the ψ argument list, arg_{i+1}. When arg_{i+1} is defined on a ψ operation, we recursively look for the definition of the first argument of this ψ operation, until a non-ψ operation is found. Now, if the definition we found for arg_{i+1} dominates the definition for arg_i, repair code is needed. A new variable is created for this repair. This variable is initialized with a copy of arg_{i+1}, guarded by the predicate associated with this argument in the ψ operation. This copy operation is inserted at the lowest point, either after the definition of arg_i or arg_{i+1}[1]. Then, arg_{i+1} is replaced in the ψ operation by this new variable.

Pourquoi? On a besoin de qchose de + fort que ?

[1] When arg_{i+1} is defined by a ψ operation, its definition may appear after the definition for arg_i, although the non-ψ definition for arg_{i+1} appears before the definition for arg_i.

C'est normalized ça ?

b_1 domine a.

$a_1 =$
~~b_2~~ $=$
$a_2 =$
$b_2 =$
$a = \psi(a_1, a_2)$
$b = \psi(b_1, b_2)$
$x = \psi(a, b)$.

The algorithm continues with the argument arg_{i+1}, until all arguments of the ψ operation are processed. When all arguments are processed, the ψ is in its normalized form. When all ψ operations are processed, the function will contain only normalized-ψ operations.

The top-down traversal of the dominator tree will ensure that when a variable in a ψ operation is defined by another ψ operation, this ψ operation has already been analyzed and put in its normalized form. Thus the definition of its first variable already dominates the definitions for the other arguments of the ψ operation.

On ne comprend pas quelle propriété on veut ici.

1.6.2 Psi-congruence

In this pass, we repair the ψ operations when variables cannot be put into the same congruence class, because their live ranges interfere. In the same way as ~~Sreedhar~~ gives a definition of the liveness on the ϕ operation, we first give a definition for the liveness on ψ operations. With this definition of liveness, an interference graph is built.

C'est pas la liveness, mais la sémantique du ϕ & ψ ⇒ use pas endroit ~~exact~~ textuel du ϕ ou ψ ⇒ liveness non textuelle.

Liveness and interferences in Psi-SSA.

We have already seen that in some cases, repair code is needed so that the arguments and definition of a ψ operation can be renamed into a single name. We first give a definition of the liveness on ψ operations such that these cases can be easily and accurately detected by observing that live-ranges for variables in a ψ operation overlap.

Consider the code in figure 1.5. The ψ operation has been replaced by ~~explicit~~ `select` operations on each predicated definition. ~~In this example, there is no relation between predicates p and q.~~ Each of these `select` operations makes an explicit use of the variable immediately to its left in the argument list of the original ψ operation. We can see that a renaming of the variables a, b, c and x into a single representative name will still compute the same value for the variable x. Note that this transformation can only be performed on normalized ψ operations, since the definition of an argument must be dominated by the definition of the argument immediately to its left in the argument list of the ψ operation. Using this equivalent representation for the ψ operation, we now give a definition of the liveness for the ψ operations.

Argh! La sémantique dépend de comment on va remplacer le ψ → select (full predicat).

A-t-on besoin de parler de select ici. La sémantique a été donnée avec de la full-prédication

Definition *We say that the point of use of an argument in a normalized ψ operation occurs at the point of definition of the argument immediately to its right in the argument list of the ψ operation. For the last argument of the ψ operation, the point of use occurs at the ψ operation itself.*

Faire une Figure avec live-range, les uses

Given this definition of liveness on ψ operations, and using the definition of liveness for ϕ operations given ~~by Sreedhar~~, a traditional liveness analysis can be run.

En fait le select correspond à l'Equivalent de la copie parallèle dans Sreedhar ? Sauf que c'est pas parallèle (si j'ai 2 ψ qui utilisent b : $a_1 =$ $a_2 =$ $b =$ select...)

$\psi(a_1, b)$
$\psi(a_2, b)$

```
    a = op1                 a = op1
p? b = op2                  b = p ? op2 : a
q? c = op3                  c = q ? op3 : b
    x = Psi(1?a, p?b, q?c)  x = c

a) Psi-SSA form             b) select form
```

Fig. 1.5 ψ and select operations equivalence

Then an interference graph can be built to collect the interferences between variables involved in ψ or ϕ operations.

Argh! Mais le graphe va changer qd on fait les réparations!

Repairing interferences on ψ operations.

We now present an algorithm that creates congruence classes with ψ operations such that there are no interference between two variables in the same congruence class.

First, the congruence classes are initialized such that each variable in the ψ-SSA representation belongs to its own congruence class. Then, ψ operations are processed one at a time, in no specific order. Two arguments of a ψ operation interfere if at least one variable from the congruence class of the first argument interferes with at least one variable from the congruence class of the second argument. When there is an interference, the two ψ arguments are marked as needing a repair. When all pairs of arguments of the ψ operation are analyzed, repair code is inserted. For each argument in the ψ operation that needs a repair, a new variable is introduced. This new variable is initialized with a predicated copy of the argument's variable. The copy operation is inserted just below the definition of the argument's variable, predicated with the predicate associated with the argument in the ψ operation.

pourquoi?

Once a ψ operation has been processed, the interference graph must be updated, so that other ψ operations are correctly handled. Interferences for the newly introduced variables must be added to the interference graph. Conservatively, we can say that each new variable interferes with all the variables that the original variable interfered with, except those variables that are now in its congruence class. Also, conservatively, we can say that the original variable interferes with the new variable in order to avoid a merge of a later ψ or ϕ operation of the two congruence classes these two variables belong to. The conservative update of the interference graph may increase the number of copies generated during the conversion to the ψ-CSSA form.

Ca me semble insuffisant. cf Contre exemple

```
a=            a=
b=            b=
              b'=
c=            c=
              c'=
ψ(a,b)        ψ(a,b)
ψ(a,b,c)      ψ(a,b',c')
d=            d=
ψ(b,d)        ψ(b,d)
```

De fait heureusement a & b n'interfèrent pas car au niveau de l'intersection du live-range le domaine de prédicats est disjoint...

Consider the code in figure 1.6 to see how this algorithm works. The definition of liveness on the ψ operation will create a live-range for variable a that extends down to the definition of b, but not further down. Thus, the variable a does not interfere with the variables b, c or x. The live-range for variable b extends down to its use in the definition of variable d. This live-range creates an interference with the variables c and x. Thus variables b, c and x cannot be put into the same congruence class. These variables are renamed respectively into variables ~~e~~, ~~f~~ and ~~g~~ and initialized b', c', x' with predicated copies. These copies are inserted respectively after the definitions for b, c and x. Variables a, e, f and g can now be put into the same congruence class, and will be renamed later into a unique representative name.

```
p? a = ...                   p? a = ...
q? b = ...                   q? b = ...
                             q? e = b
r? c = ...                   r? c = ...
                             r? f = c
   x = Psi(p?a, q?b, r?c)    g = Psi(p?a, q?e, r?f)
                                x = g
s? d = b + 1                 s? d = b + 1
```

Rajouter les live-range sur la figure.

Fig. 1.6 Elimination of ψ live-interference

1.6.3 Phi-congruence

When all ψ operations are processed, the congruence classes built from ψ operations are extended to include the variables in ϕ operations. In this part, the algorithm ~~from~~ ~~Sreedhar~~ presented in chapter ... is used, with a few modifications.

The first modification is that the congruence classes must not be initialized at the beginning of this process. They have already been initialized at the beginning of the ψ-congruence step, and were extended during the processing of ψ operations. These congruence classes will be extended now with ϕ operations during this step.

The second modification is that the live-analysis run for this part must also take into account the special liveness rule on the ψ operations. The reason for this is that for any two variables in the same congruence class, any interference, either on a ψ or on a ϕ operation, will not preserve the correct semantics if the variables are renamed into a representative name. ?

All other parts of the ~~out-of-SSA algorithm~~ ~~from Sreedhar~~ are unchanged, and in particular, any of the three algorithms described for the conversion into a CSSA form can be used. ?

We have described a complete algorithm to convert a ψ-SSA representation into a ψ-CSSA representation. The final step to convert the code into a non-SSA form is a simple renaming of all the variables in the same ~~congruence class~~ web into a representative name. The ψ and ϕ operations are then removed.

We now present some improvements that can be added so as to reduce the number of copies inserted by this algorithm.

1.6.4 *Improvements to the out of Psi-SSA algorithm*

Non-normalized ψ operations with disjoint predicates.

When two arguments in a ψ operation do not have their definitions correctly ordered, the ψ operation is not normalized. We presented an algorithm to restore the normalized property by adding a new predicated definition of a new variable. However, if we know that the predicate domains of the two arguments are actually disjoint, the semantics of the ψ operation is independent on their relative order. So, instead of adding repair code, these two arguments can simply be reordered in the ψ operation itself, to restore the normalized property.

L'égalité des prédicats n'est pas une contrainte forte non plus ie le prédicat dans le ψ peut être changé si ça ne change pas la sémantique et permet de normaliser –

Interference with disjoint predicates.

When the live-ranges of two variables overlap, an interference is added for these two variables in the interference graph. If the definitions for these variables are predicated definitions, their live-ranges are only valid under a specific predicate domain. These domains are the domains of the predicates used on the definitions of the variables. Then, if these domains are disjoint, then although the live-range overlap, they are on disjoint conditions and thus they do not create an interference in the interference graph. Removing this interference from the interference graph will avoid the need to add repair code when live-ranges on disjoint predicates overlap.

De fait c'est plus précis : c'est le prédicat de l'utilisation qui définit le domaine de la partie basse du live range :

a = ...   a sur true

g? b=a   sur g?

Repair interference on the left argument only.

When an interference is detected between two arguments in a ψ operation, only the argument on the left actually needs a repair. The reason is that, since the ψ operations are normalized, the definition of an argument is always dominated by the definition of an argument on its left. Thus adding a copy for the argument on the right will not remove the interference. However, the copy must now be put just before the definition of the next argument in the ψ operation, or just before the ψ operation if this is the last argument.

? pourquoi pas juste après la définition?

Interference with the result of a ψ operation.

When the live-range for an argument of a ψ operation overlaps with the live-range of the variable defined by the ψ operation, this interference can be ignored. Actually, there are two cases to consider:

- If the argument is not the last one in the ψ operation, and its live-range overlaps with the live-range of the definition of the ψ operation, then this live-range also overlaps with the live-range of the last argument. Thus this interference will already be detected and repaired.

- If the argument is the last one of the ψ operation, then the value of the ψ operation is the value of this last argument, and this argument and the definition will be renamed into the same variable out of the SSA representation. Thus, there is no need to introduce a copy here.

~~This citation is to make chapters without citations build without error. Please ignore it: [?].~~

Rajouter une section références, further readings etc.