Cours de « concepts avancés de compilation »

Projet de compilation

Auteur : F. Védrine

Ce document décrit les étapes précises du projet concernant à la mise sous forme SSA ou « Static Single Assignement ». Il est organisé en 7 étapes distinctes. Chaque étape donne lieu à un démonstrateur que vous pouvez lancer sur les fichiers `essai.c` et `essai2.c`. Les fichiers source sont sur le Wiki du cours.
Les différentes étapes sont :

1. La définition des sources au début du projet
2. La construction de l'arbre de domination
3. La construction de la frontière de domination
4. La propagation de l'insertion des fonctions Φ
5. La propagation de l'insertion des fonctions Φ sur la frontière de domination des labels
6. L'insertion des fonctions Φ
7. Le renommage des expressions

Les étapes 2, 4, 5, 7 font intervenir une `WorkList` se propageant sur le graphe de contrôle. Pour l'étape 2, la propagation se fait par point fixe. Pour les étapes 4 et 7, la propagation s'effectue qu'en une et une seule passe sur les expressions. Pour l'étape 5, la propagation se fait par point fixe, mais sur les labels uniquement.
Les étapes 3 et 6 sont déclinées sur tous les labels.

La définition des sources au début du projet

Les sources sont constituées de 6 fichiers, à savoir les fichiers `SimpleC.lex`, `SimpleC.yy`, `SyntaxTree.h`, `SyntaxTree.cpp`, `Algorithms.h`, `Makefile`. On se référera à la notice explicative pour une description de leur contenu.

Le résultat de l'affichage après `make` et ./mycomp essai.c sur le fichier essai.c suivant :

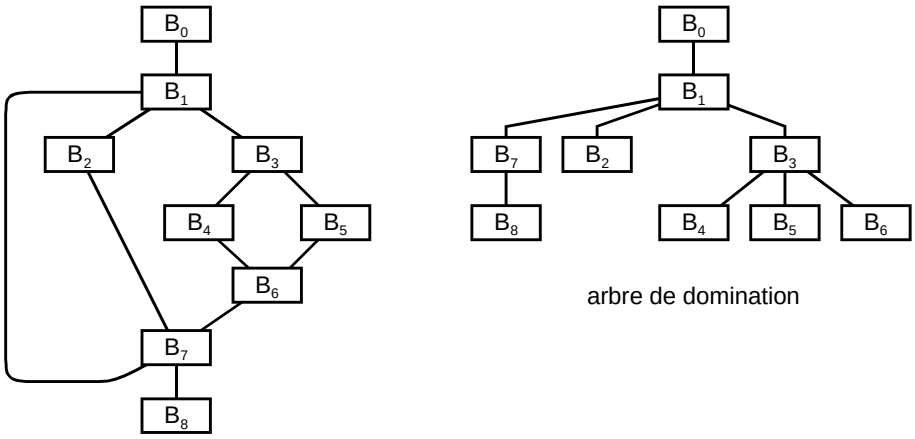
```
int f(int x) {
   return x+2;
}

int main(int argc, char** argv) {
  int x;
  int y;
  x = argc;
  y = 2;
  x = x * x;
  do {
     x = x+2;
  } while (x < 100);

  if (x > y) {
    int  k;
    k = 8;
    x = 4 + x * 2 - 1;
  }
  else {
    x = x + 1 - f(y);
  };
  return x + 3;
}
```

est

```
…
function f
0   {
1     return ([parameter 0: x] + 2);

function main
0   {0x584ec0 x := 0    0x585070 y := 1
1     [local 0: x] = [parameter 0: argc];
2     [local 1: y] = 2;
3     [local 0: x] = ([local 0: x] * [local 0: x]);
4     goto label 0x585338
5     label 585338
6     {
7       [local 0: x] = ([local 0: x] + 2);
8     }
9     if (([local 0: x] < 100))
10    then
11    goto loop 585338
12    else
13    if (([local 0: x] > [local 1: y]))
14    then
15    {0x585680 k := 0
16      [local 2: k] = 8;
17      [local 0: x] = ((4 + ([local 0: x] * 2)) - 1);
18    }
25    goto label 0x585b80
19    else
20    {
21      [local 0: x] = (([local 0: x] + 1) - f([local 1: y]));
22    }
23    goto label 0x585b80
24    label 585b80
26    return ([local 0: x] + 3);
```

La construction de l'arbre de domination

La construction de l'arbre de domination consiste à partir d'un graphe dirigé (DAG) à construire l'arbre ci-dessous.



La construction est progressive et s'effectue par plus grand point fixe. Au départ l'ensemble des dominateurs d'un point est tous les autres points. Cet ensemble diminue progressivement, jusqu'au résultat qui ne contient qu'un fil de dominateurs constitué de son dominateur immédiat, du dominateur immédiat de son dominateur immédiat, …, jusqu'au point initial de la fonction. L'arbre peut être complètement déterminé à partir d'un stockage initial minimal, constitué de nœuds contenant leur profondeur dans l'arbre de domination et ayant un lien vers leur dominateur immédiat. Sachant que pour les nœuds sauf les labels, le dominateur immédiat est l'unique précédent, nous nous contentons de stocker la hauteur de domination au niveau des nœuds et le dominateur immédiat au niveau des labels. La construction par plus grand point fixe implique que cette hauteur ne peut que diminuer au cours du temps.

Mise à jour des structures de données

1. Rajoutez le champ **`int`** `m_dominationHeight` au niveau de la classe `VirtualInstruction` dans le fichier `SyntaxTree.h:631`. Ce champ est la profondeur de l'instruction dans l'arbre de domination.

```
class VirtualInstruction {
  …
 private:
  …
  int m_registrationIndex;
  int m_dominationHeight;

 public:
  VirtualInstruction() : m_type(TUndefined), m_next(NULL), m_previous(NULL), m_registrationIndex(-1),
                         m_dominationHeight(0), mark(NULL) {}
  virtual ~VirtualInstruction() {}
  …
  virtual bool propagateOnUnmarked(VirtualTask& task, WorkList& continuations, Reusability& reuse) const
    {  …
    }
  void setDominationHeight(int height) { m_dominationHeight = height; }
  const int& getDominationHeight() const { return m_dominationHeight; }

  …
};
```

2. Rajoutez un champ `VirtualInstruction* m_dominator` au niveau de la classe `LabelInstruction` dans le fichier `SyntaxTree.h:799`. Ce champ est à afficher pour l'utilisateur

```
class LabelInstruction : public VirtualInstruction {

 private:
  GotoInstruction* m_goto;
  VirtualInstruction* m_dominator;

 public:
  LabelInstruction() : m_goto(NULL), m_dominator(NULL) { setType(TLabel); }

  virtual int countPreviouses() const { … }
  VirtualInstruction* getSDominator() const { return m_dominator; }
  void setGotoFrom(GotoInstruction& gotoPoint) { assert(!m_goto); m_goto = &gotoPoint; }
  virtual void print(std::ostream& out) const
    {  out << "label " << this;
       if (m_dominator) {
          out << '\t' << "dominated by " << m_dominator->getRegistrationIndex() << ' ';
          m_dominator->print(out);
       }
       else
          out << '\n';
    }
};
```

Définition des tâches

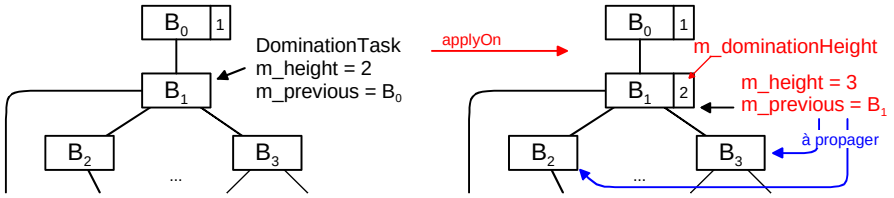
3. En vous inspirant du code de `PrintTask` et de `PrintAgenda` dans `Algorithms.h:54`, écrivez environ 40 lignes de code correspondant à la classe `DominationTask` (tâche se propageant en avant et calculant les dominateurs immédiats ainsi que la hauteur des instructions dans l'arbre de domination) et à la classe `DominationAgenda`.

```
enum TypeTask { TTUndefined, TTPrint, TTDomination };
class PrintTask { … };
class PrintAgenda { … };
class DominationTask : public VirtualTask {
 public:
  int m_height;
  VirtualInstruction* m_previous;

 public:
  DominationTask(const VirtualInstruction& instruction) : m_height(1), m_previous(NULL)
    {  setInstruction(instruction); }
  DominationTask(const DominationTask& source)
    :  VirtualTask(source), m_height(source.m_height) , m_previous(NULL) {}
```

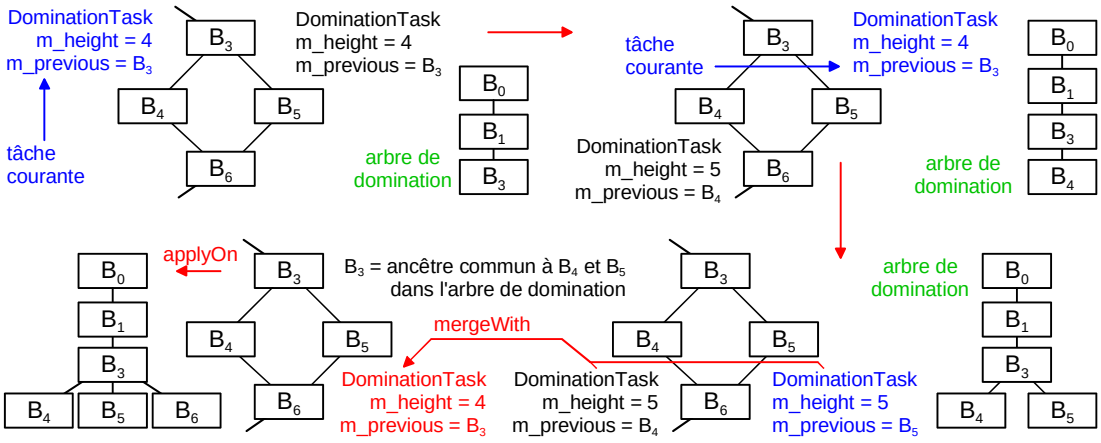
```
  void setHeight(int newHeight) { assert(newHeight < m_height); m_height = newHeight; }
  VirtualInstruction* findDominatorWith(VirtualInstruction& instruction) const;

  virtual void applyOn(VirtualInstruction& instruction, WorkList& continuations)
    {  instruction.setDominationHeight(…);
       m_previous = …;
       m_height = …;
    }
  virtual VirtualTask* clone() const { return new DominationTask(*this); }
  virtual int getType() const { return TTDomination; }
  virtual bool mergeWith(VirtualTask& vtSource)
    {  assert(dynamic_cast<const DominationTask*>(&vtSource));
       DominationTask& source = (DominationTask&) vtSource;
       m_previous = …;
       m_height = …;
       return true;
    }
};

class DominationAgenda : public WorkList {
  public:
  DominationAgenda(const Function& function)
    {  addNewAsFirst(new DominationTask(function.getFirstInstruction())); }
  virtual void markInstructionWith(VirtualInstruction& instruction, VirtualTask& task) {}
};
```

La méthode `DominationTask::findDominatorWith` sera implantée plus tard. Elle détermine l'ancêtre commun dans l'arbre de domination (déjà calculé) entre deux instructions à savoir `DominationTask::m_previous` et `instruction`.

La méthode `DominationTask::applyOn` applique la tâche de domination pour la préparer à sa propagation. Elle est appelée par le traitement des tâches de domination pour toutes les instructions : voir `VirtualInstruction::handle` dans `SyntaxTree.cpp:99`. L'implantation se contente alors de définir la profondeur de domination de l'instruction `VirtualInstruction::m_dominationHeight` avec `m_height` et de préparer notre tâche de domination pour qu'elle se propage sur l'instruction suivante.



La méthode `DominationTask::mergeWith` fusionne deux tâches dans l'agenda. Elle est appelée par la méthode `WorkList::addNewSorted` dans `SyntaxTree.cpp:67`, elle-même appelée par `WorkList::execute` dans `SyntaxTree.cpp:47`, suite au traitement d'une propagation `GotoInstruction::propagateOnUnmarked` sur les `goto` avant les labels (fichier `SyntaxTree.h:751`). B3, B4, B5.



La méthode `DominationAgenda::markInstructionsWith` indique juste que nous ne devons pas faire de marquage à ce niveau. Le marquage est finalement effectué dans le champ `LabelInstruction::m_dominator`. Lorsque ce champ est stable, l'algorithmique stoppe la propagation : ce point est traité par `LabelInstruction::handle` pour les tâches de domination.

Implantation de l'algorithmique locale

4. Écrivez la méthode `LabelInstruction::handle` traitant spécifiquement des tâches de domination, ce qui correspond à 15 lignes de code. Rajoutez la déclaration de la méthode dans `SyntaxTree.h:805`.

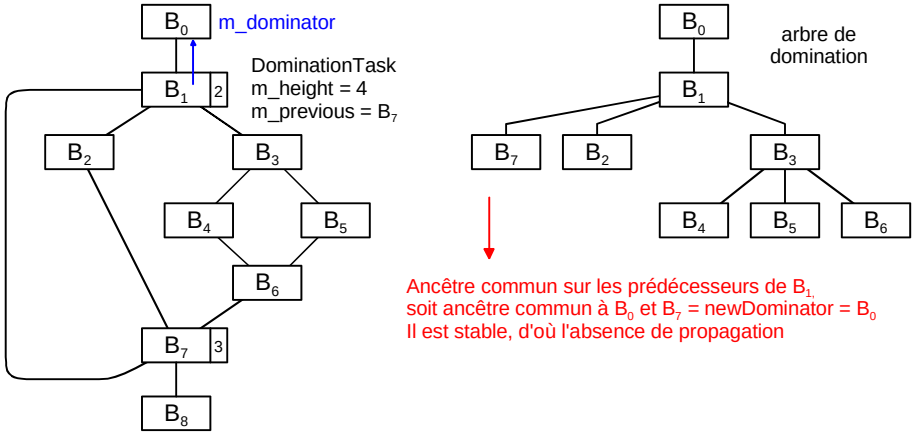
```
class LabelInstruction : public VirtualInstruction {
  …
  virtual int countPreviouses() const { return …; }
  virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
  VirtualInstruction* getSDominator() const { return m_dominator; }
  …
};
```

Implantez la méthode dans `SyntaxTree.cpp:101`.

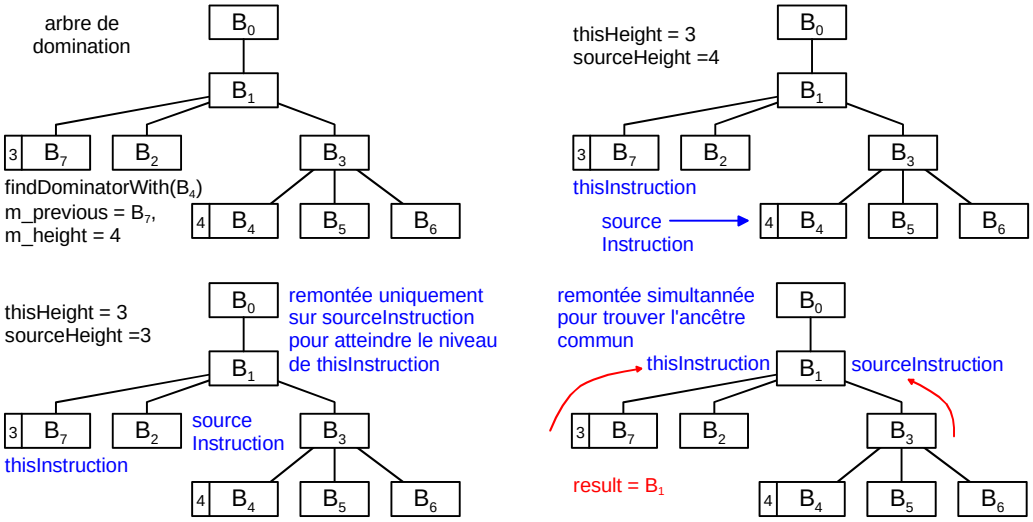
```
void
LabelInstruction::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  int type = vtTask.getType();
  if (type == TTDomination) { 1
    assert(dynamic_cast<const DominationTask*>(&vtTask));
    DominationTask& task = (DominationTask&) vtTask;
    if (!m_dominator) 2
      m_dominator = …;
    else { 3
      VirtualInstruction* newDominator = …; 4
      if (newDominator == m_dominator) 5
        return;
 6    task.setHeight(…);
      m_dominator = …;
    };
  };
  VirtualInstruction::handle(vtTask, continuations, reuse); 7
}
```

1 permet d'effectuer un traitement particulier pour les tâches de domination.
2 traite la première fois où nous arrivons sur un label = comme une instruction
3 traite le cas où nous revenons sur un label après être passé une première fois.
4 calcule l'ancêtre commun dans l'arbre de domination entre l'ancien dominateur stocké et le précédent de notre label provenant de `task`. Le problème est similaire à la méthode `DominationTask::mergeWith`, si ce n'est que la fusion a lieu dans le temps.

5 indique si nous avons stabilité dans le calcul du dominateur. Si oui, le plus grand point fixe est atteint et nous ne propageons pas la tâche sur le successeur du label en court-circuitant `VirtualInstruction::handle`.
6 fait en sorte que la mise à jour de la tâche par la méthode `DominationTask::applyOn` (appelée par `VirtualInstruction::handle`) en vue de sa propagation soit correcte.
7 propage la tâche non stable (puisque nous ne sommes pas dans le cas 5) sur le suivant du label.



5. Complétez l'algorithmique en implantant la méthode `findDominatorWith` pour la classe `DominationTask` dans le fichier `Algorithm.cpp`. Il faut créer ce dernier et le placer dans le `Makefile`. Cette méthode trouve le dominateur commun à deux instructions (`m_previous` et l'instruction en paramètre) en synchronisant la recherche sur une hauteur commune `VirtualInstruction::m_dominationHeight`). Noter que le dominateur d'une instruction standard (hors label) est son unique prédécesseur `VirtualInstruction::getSPreviousInstruction()` et que le dominateur d'un label est disponible par `LabelInstruction::getSDominator()`.




```
// Fichier Algorithms.cpp
#include "Algorithms.h"

VirtualInstruction*
DominationTask::findDominatorWith(VirtualInstruction& source) const {
  int thisHeight = m_height-1;
  VirtualInstruction *thisInstruction = m_previous, *sourceInstruction = &source;
  int sourceHeight = sourceInstruction->getDominationHeight();
  while (thisHeight > sourceHeight) {
    assert(thisInstruction);
    if (thisInstruction->countPreviouses() == 1)
      thisInstruction = …;
```

```
    else {
      assert(dynamic_cast<const LabelInstruction*>(thisInstruction));
      thisInstruction = …;
    };
    thisHeight = …;
  };
  while (sourceHeight > thisHeight) {
    assert(sourceInstruction);
    if (sourceInstruction->countPreviouses() == 1)
      sourceInstruction = …;
    else {
      assert(dynamic_cast<const LabelInstruction*>(sourceInstruction));
      sourceInstruction = …;
    };
    sourceHeight = …;
  };
  while (thisInstruction != sourceInstruction) {
    assert(thisInstruction && sourceInstruction);
    if (thisInstruction->countPreviouses() == 1)
      thisInstruction = …;
    else {
      assert(dynamic_cast<const LabelInstruction*>(thisInstruction));
      thisInstruction = …;
    };
    if (sourceInstruction->countPreviouses() == 1)
      sourceInstruction = …;
    else {
      assert(dynamic_cast<const LabelInstruction*>(sourceInstruction));
      sourceInstruction = …;
    };
  };
  return thisInstruction;
}
```

```
# Fichier Makefile
CXX = g++
CXXFLAGS = -Wall -Winline -fmessage-length=0 -ggdb -fno-inline

all : my_comp.exe

my_comp.exe : SyntaxTree.o SimpleC_gram.o SimpleC_lex.o  Algorithms.o
	$(CXX) -o my_comp.exe $(CXXFLAGS) SyntaxTree.o SimpleC_gram.o SimpleC_lex.o Algorithms.o -lfl

SyntaxTree.o : SyntaxTree.cpp SyntaxTree.h Algorithms.h
	$(CXX) -c -o SyntaxTree.o $(CXXFLAGS) SyntaxTree.cpp

Algorithms.o : Algorithms.cpp SyntaxTree.h Algorithms.h
	$(CXX) -c -o Algorithms.o $(CXXFLAGS) Algorithms.cpp

SimpleC_gram.o : SimpleC_gram.cpp SyntaxTree.h
	$(CXX) -c -o SimpleC_gram.o $(CXXFLAGS) SimpleC_gram.cpp

SimpleC_lex.o : SimpleC_lex.cpp SyntaxTree.h
	$(CXX) -c -o SimpleC_lex.o $(CXXFLAGS) SimpleC_lex.cpp

SimpleC_gram.cpp : SimpleC.yy
	bison --debug -o SimpleC_gram.cpp SimpleC.yy

SimpleC_lex.cpp : SimpleC.lex
	flex -oSimpleC_lex.cpp SimpleC.lex
```

Assemblage

6. Déclarez la méthode `computeDominators` dans la classe `Program` (fichier `SyntaxTree.h:1032`).

```
void
class Program {
  …
  void printWithWorkList(std::ostream& out) const;
  void computeDominators();
  class ParseContext {
    …
  };
};
```

Implantez cette méthode dans le fichier `SyntaxTree.cpp:152`.

```
void
Program::printWithWorkList(std::ostream& osOut) const {
  …
}

void
Program::computeDominators() {
  for (std::set<Function>::const_iterator functionIter = m_functions.begin();
       functionIter != m_functions.end(); ++functionIter) {
    DominationAgenda agenda(*functionIter);
    agenda.execute();
  };
}
```

7. Rajouter dans `main()` (fichier `SyntaxTree.cpp:177`) le code suivant

```
int main( int argc, char** argv ) {
  …
  std::cout << std::endl;
  pProgram.printWithWorkList(std::cout);
  std::cout << std::endl;
  program.computeDominators();
  program.printWithWorkList(std::cout);
  std::cout << std::endl;
  return 0;
}
```

Le résultat sur `essai.c` est la sortie suivante :

```
function f
0  {
1    return ([parameter 0: x] + 2);

function main
0  {0x594ed0 x := 0    0x595080 y := 1
1    [local 0: x] = [parameter 0: argc];
2    [local 1: y] = 2;
3    [local 0: x] = ([local 0: x] * [local 0: x]);
4    goto label 0x595360
5    label 595360      dominated by 4 goto label 0x595360
6    {
7      [local 0: x] = ([local 0: x] + 2);
8    }
9    if (([local 0: x] < 100))
10   then
11   goto loop 595360
12   else
13   if (([local 0: x] > [local 1: y]))
14   then
15   {0x5956e0 k := 0
16     [local 2: k] = 8;
17     [local 0: x] = ((4 + ([local 0: x] * 2)) - 1);
18   }
25   goto label 0x595c18
19   else
20   {
21     [local 0: x] = (([local 0: x] + 1) - f([local 1: y]));
22   }
23   goto label 0x595c18
24   label 595c18      dominated by 13 if (([local 0: x] > [local 1: y]))
26   return ([local 0: x] + 3);
```

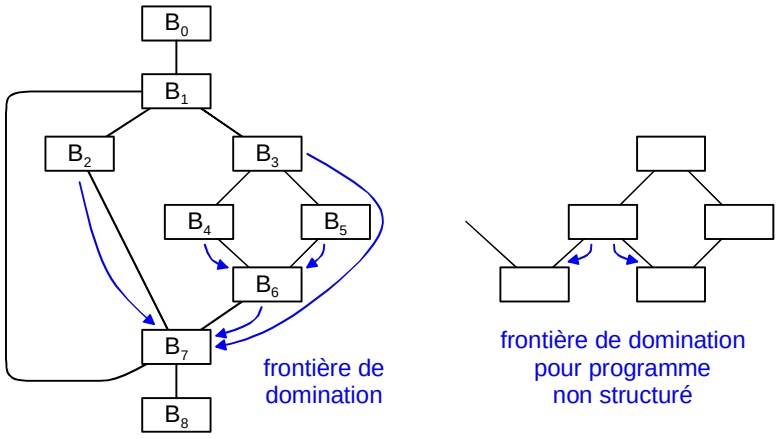
Le résultat sur `essai2.c` est la sortie suivante :

```
function main
0  {0x594cd8 x := 0    0x594e98 y := 1
1    [local 0: x] = [parameter 0: argc];
2    [local 0: x] = (([local 0: x] * [local 0: x]) - (2 * [local 0: x]));
3    [local 1: y] = (4 + [parameter 0: argc]);
4    goto label 0x595248
5    label 595248      dominated by 4 goto label 0x595248
6    {
7      if (([local 0: x] > 2))
8      then
9      {
10       [local 0: x] = ([local 0: x] - 1);
11     }
30     goto label 0x595b70
12     else
```

```
13    {
14      if (((2 * [local 0: x]) > [local 1: y]))
15      then
16      {
17        [local 1: y] = 2;
18      }
25      goto label 0x595a28
19      else
20      {
21        [local 0: x] = ([local 0: x] - 1);
22      }
23      goto label 0x595a28
24      label 595a28  dominated by 14 if (((2 * [local 0: x]) > [local 1: y]))
26      [local 0: x] = ([local 0: x] - 1);
27    }
28    goto label 0x595b70
29    label 595b70    dominated by 7 if (([local 0: x] > 2))
31    [local 0: x] = ([local 0: x] - 1);
32  }
33  if (([local 0: x] > 0))
34  then
35  goto loop 595248
36  else
37  return 0;
```

La construction de la frontière de domination

La frontière de domination d'une instruction A est l'ensemble des nœuds que A ne domine pas, mais dont A domine un des prédécesseurs. Ces nœuds sont nécessairement des labels et la frontière de domination a essentiellement un intérêt pour les instructions situées sur la branche `then` ou `else` d'un if, ainsi que pour les labels.



Mise à jour des structures de données

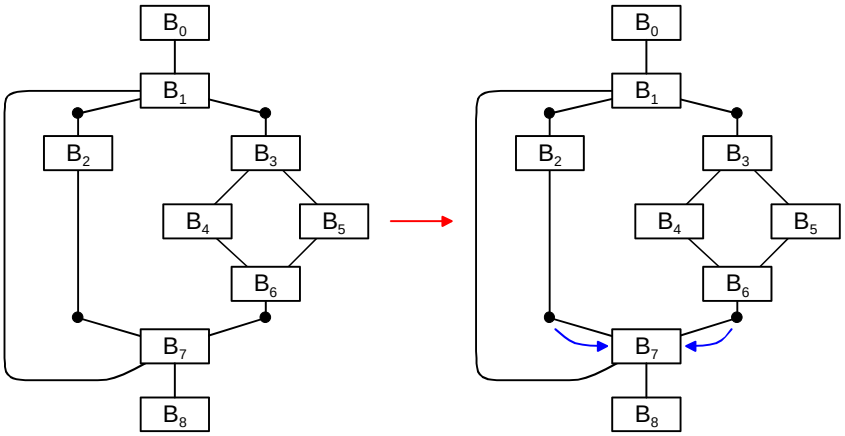
Rajoutez le champ `std::vector<GotoInstruction*> m_dominationFrontier` au niveau de la classe `GotoInstruction` et de la classe `LabelInstruction` dans le fichier `SyntaxTree.h:739` et `SyntaxTree.h:809`. Les labels qui sont dans la frontière de domination sont alors les suivants des `goto`s enregistrés dans ces champs. Nous avons besoin des `goto`s précédents les labels pour déterminer l'origine lors de l'insertion des fonctions Φ.

```
class GotoInstruction : public VirtualInstruction {
  public:
   typedef std::vector<GotoInstruction*> DominationFrontier;
   …
  private:
   Context m_context;
   DominationFrontier m_dominationFrontier;

  public:
   …
   virtual bool propagateOnUnmarked(VirtualTask& task, WorkList& continuations, Reusability& reuse) const
     { … }
   void addDominationFrontier(GotoInstruction& gotoInstruction);
   DominationFrontier& getDominationFrontier() { return m_dominationFrontier; }
```

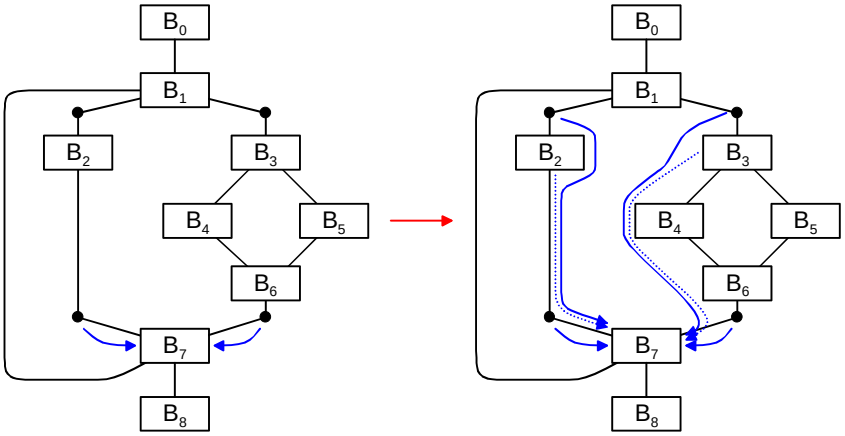
```
  const DominationFrontier& getDominationFrontier() const { return m_dominationFrontier;}
  virtual void print(std::ostream& out) const
    {  if (m_context == CLoop)
         …
       else if (…)
         …
       else
         out << "goto " << std::hex << (int) getSNextInstruction() << std::dec;
       if (!m_dominationFrontier.empty()) {
         out << "\tdomination frontier = ";
         for (DominationFrontier::const_iterator iter = m_dominationFrontier.begin();
              iter != m_dominationFrontier.end(); ++iter)
           out << ((*iter)->getSNextInstruction())->getRegistrationIndex();
       };
       out << '\n';
    }
};

class LabelInstruction : public VirtualInstruction {
 public:
  typedef std::vector<GotoInstruction*> DominationFrontier;
 private:
  GotoInstruction* m_goto;
  VirtualInstruction* m_dominator;
  DominationFrontier m_dominationFrontier;

 public:
  …
  virtual void print(std::ostream& out) const
    {  …
       if (m_dominator) { … }
       else out << '\n';
       if (!m_dominationFrontier.empty()) {
         out << "\tdomination frontier of label = ";
         for (std::vector<GotoInstruction*>::const_iterator iter = m_dominationFrontier.begin();
              iter != m_dominationFrontier.end(); ++iter)
           out << (*iter)->getSNextInstruction()->getRegistrationIndex();
         out << '\n';
       };
    }
  void addDominationFrontier(GotoInstruction& gotoInstruction)
    { m_dominationFrontier.push_back(&gotoInstruction); }
  DominationFrontier& getDominationFrontier() { return m_dominationFrontier; }
  const DominationFrontier& getDominationFrontier() const { return m_dominationFrontier;}
  friend class Function;
};

inline void
GotoInstruction::connectToLabel(LabelInstruction& liInstruction) {  … }

inline void
GotoInstruction::addDominationFrontier(GotoInstruction& gotoInstruction)
  {  m_dominationFrontier.push_back(&gotoInstruction); }
```

Calcul de la frontière locale de domination

1. Rajoutez la méthode `setDominationFrontier` à la classe `Function`, méthode qui calcule la frontière de domination (fichier `SyntaxTree.h:992` et `SyntaxTree.cpp:141`). Cette méthode parcourt tous les labels de la fonction et pour chaque label `label` :
   a) Les précédents (`label.getSPreviousInstruction()` et `label.m_goto`) du label mettent `label` dans leur frontière de domination si et seulement si ces précédents ne dominent pas `label`.



b) Pour chaque `VirtualInstruction` dont `label` se trouve être dans sa frontière de domination (le `VirtualInstruction` est un précédent du label comme indiqué au point 1, ou un dominateur d'un précédent du label comme indiqué au point 2), on place `label` dans la frontière de domination de son dominateur immédiat, si et seulement si ce dominateur immédiat ne domine pas `label`. Noter que la frontière de domination n'est stockée (traits bleus pleins) qu'au niveau des `GotoInstruction` en destination de `IfInstruction` (en dehors des `GotoInstruction` avant `label`), les autres instructions se contentant de propager l'information.



Ces deux points sont repris dans le code suivant (à compléter) et l'algorithmique est expliquée par la suite. La méthode `Function::setDominationFrontier` est déclarée dans `SyntaxTree.h:995`.

```
class Function {
  private:
   …
  public:
   …
   void addNewInstructionAfter(VirtualInstruction* newInstr, VirtualInstruction& prev)
     { … }
   void setDominationFrontier();
   void addFirstInstruction(VirtualInstruction* pviNewInstruction)
     { … }
   …
};
```

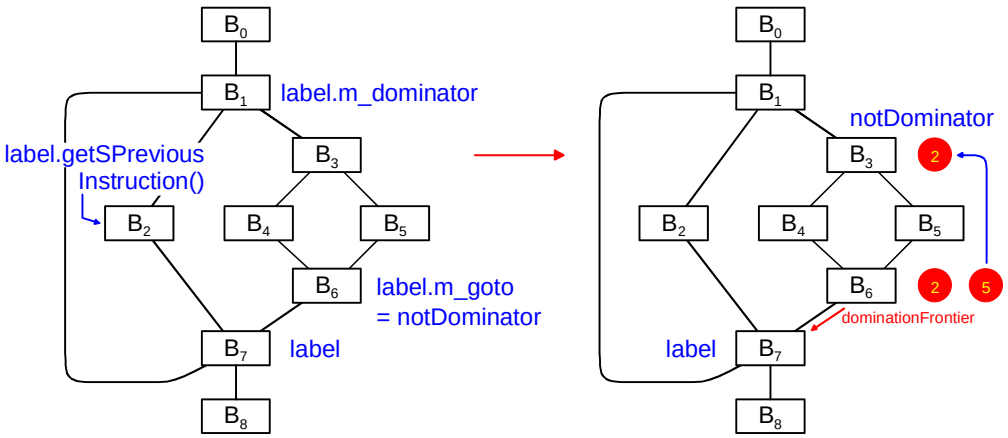
La méthode `Function::setDominationFrontier` est implantée dans `SyntaxTree.cpp:141`.

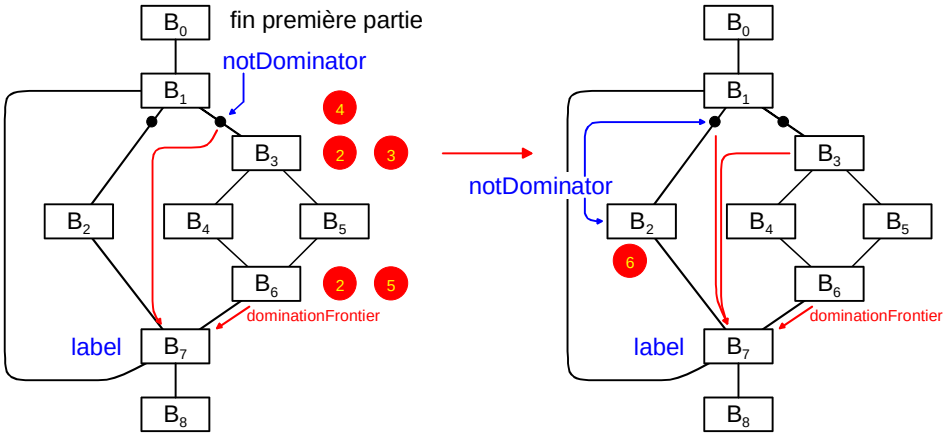
```
void
Function::setDominationFrontier() {
   for (std::vector<VirtualInstruction*>::const_iterator iter=m_instructions.begin(); iter != m_instructions.end();++iter) {
      if ((*iter)->type() == VirtualInstruction::TLabel) {
         assert(dynamic_cast<const LabelInstruction*>(*iter));
```

```
      LabelInstruction& label = *((LabelInstruction*) *iter); 1
      if (label.m_goto != NULL) {
        VirtualInstruction* notDominator = label.m_goto;
        while (label.m_dominator != notDominator) { 2
          while (notDominator->type() != VirtualInstruction::TLabel
              && notDominator->getSPreviousInstruction()
              && notDominator->getSPreviousInstruction()->type() != VirtualInstruction::TIf)
            notDominator = …; 3
          if (notDominator->getSPreviousInstruction()
              && (notDominator->getSPreviousInstruction()->type()  == VirtualInstruction::TIf)) { 4
            assert(dynamic_cast<const GotoInstruction*>(notDominator));
            ((GotoInstruction&) *notDominator).addDominationFrontier(…);
            notDominator = …;
          };
          if (notDominator->type() == VirtualInstruction::TLabel) { 5
            assert(dynamic_cast<const LabelInstruction*>(notDominator));
            LabelInstruction& labelInstruction = (LabelInstruction&) *notDominator;
            labelInstruction.addDominationFrontier(…);
            notDominator = …;
          };
        };
      };
      if (label.getSPreviousInstruction() != NULL) { 6
        VirtualInstruction* notDominator = label.getSPreviousInstruction();
        assert(dynamic_cast<const GotoInstruction*>(notDominator));
        GotoInstruction* origin = (GotoInstruction*) notDominator;
        while (label.m_dominator != notDominator) {
          while (…)
            notDominator = …;
          if (notDominator->getSPreviousInstruction() && notDominator->getSPreviousInstruction()->type() == …) {
            …
          };
          if (notDominator->type() == …) {
            …
          };
        };
      };
    };
  };
}
```

1 choisit le label et construit la frontière de domination en partant des précédents du label.

2 tant que le précédent et ses dominateurs ne dominent pas le label, notre label est dans leur frontière de domination.





3 traite le cas des instructions standards en remontant sur l'unique précédent qui correspond au dominateur immédiat.

4 traite le cas des then ou else après un if, pour lesquels il est nécessaire de stocker `label` dans la frontière de domination.

5 traite le cas des frontière de domination sur les labels, en remontant sur le dominateur immédiat du label.

6 tant que le précédent et ses dominateurs ne dominent pas le label, `label` est dans leur frontière de domination.

2. Terminez l'algorithmique en rajoutant la méthode `computeDominationFrontiers` à la classe `Program` (fichier `SyntaxTree.h:1063` et `SyntaxTree.cpp:215, 246`).

```
class Program {
  private:
   …
  public:
   …
   void computeDominators();
   void computeDominationFrontiers();
   class ParseContext { … };
};

…
void
Program::computeDominators() { … }

void
Program::computeDominationFrontiers() {
   for (std::set<Function>::iterator functionIter = m_functions.begin();
        functionIter != m_functions.end(); ++functionIter)
     const_cast<Function&>(*functionIter).setDominationFrontier();
}
…

int main(int argc, char** argv ) {
   …
   program.computeDominators();
   program.printWithWorkList(std::cout);
   std::cout << std::endl;
   program.computeDominationFrontiers();
   program.printWithWorkList(std::cout);
   std::cout << std::endl;
   return 0;
}
```

Le résultat sur `essai.c` est la sortie suivante :

```
function f
0   {
1     return ([parameter 0: x] + 2);

function main
0   {0xa045178 x := 0   0xa045120 y := 1
1     [local 0: x] = [parameter 0: argc];
2     [local 1: y] = 2;
```

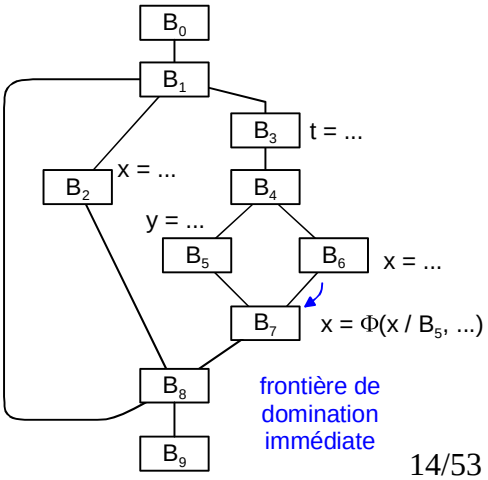
```
3    [local 0: x] = ([local 0: x] * [local 0: x]);
4    goto label 0xa045508
5    label a045508     dominated by 4 goto label 0xa045508
      domination frontier of label = 5
6    {
7      [local 0: x] = ([local 0: x] + 2);
8    }
9    if (([local 0: x] < 100))
10   then      domination frontier = 5
11   goto loop a045508
12   else
13   if (([local 0: x] > [local 1: y]))
14   then      domination frontier = 24
15   {0xa0457f0 k := 0
16     [local 2: k] = 8;
17     [local 0: x] = ((4 + ([local 0: x] * 2)) - 1);
18   }
25   goto label 0xa045e20
19   else      domination frontier = 24
20   {
21     [local 0: x] = (([local 0: x] + 1) - f([local 1: y]));
22   }
23   goto label 0xa045e20
24   label a045e20     dominated by 13 if (([local 0: x] > [local 1: y]))
26   return ([local 0: x] + 3);
```

Le résultat sur `essai2.c` est la sortie suivante :

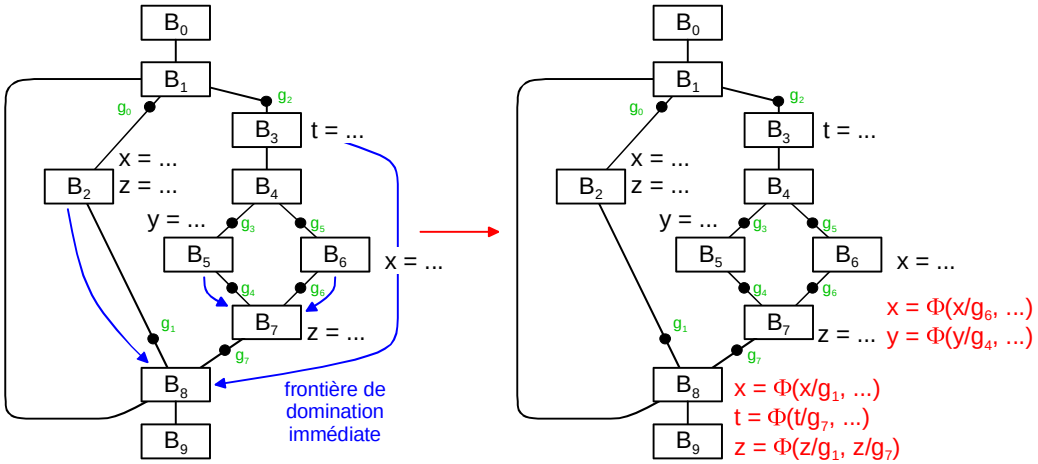
```
function main
0   {0x734be0 x := 0    0x734da0 y := 1
1    [local 0: x] = [parameter 0: argc];
2    [local 0: x] = (([local 0: x] * [local 0: x]) - (2 * [local 0: x]));
3    [local 1: y] = (4 + [parameter 0: argc]);
4    goto label 0x735158
5    label 735158      dominated by 4 goto label 0x735158
      domination frontier of label = 5
6    {
7      if (([local 0: x] > 2))
8      then    domination frontier = 29
9      {
10       [local 0: x] = ([local 0: x] - 1);
11     }
30     goto label 0x735ad8
12     else    domination frontier = 29
13     {
14       if (((2 * [local 0: x]) > [local 1: y]))
15       then  domination frontier = 24
16       {
17         [local 1: y] = 2;
18       }
25       goto label 0x735970
19       else  domination frontier = 24
20       {
21         [local 0: x] = ([local 0: x] - 1);
22       }
23       goto label 0x735970
24       label 735970  dominated by 14 if (((2 * [local 0: x]) > [local 1: y]))
      domination frontier of label = 29
26       [local 0: x] = ([local 0: x] - 1);
27     }
28     goto label 0x735ad8
29     label 735ad8    dominated by 7 if (([local 0: x] > 2))
      domination frontier of label = 5
31     [local 0: x] = ([local 0: x] - 1);
32   }
33   if (([local 0: x] > 0))
34   then      domination frontier = 5
35   goto loop 735158
36   else
37   return 0;
```

La propagation de l'insertion locale des fonctions Φ

L'insertion des fonctions Φ s'effectue en premier lieu sur la frontière de domination des blocs concernés.



Cette passe se contente de propager les fonctions Φ sur les frontières de domination immédiate : lorsqu'une variable est modifiée dans un bloc d'instructions en séquence, on ajoute alors nécessairement une fonction Φ sur la frontière de domination immédiate à cette séquence.
L'algorithmique ne parcourt les instructions qu'une et une seule fois et le résultat est stocké dans le marquage de la classe `LabelInstruction` (voir le champ `VirtualInstruction::mark`). Le résultat n'est donc pas visible à l'impression, mais l'est après l'insertion effective des fonctions Φ.
Le résultat de cette phase est dessiné sur le schéma suivant :



Les points de suspension sur ce schéma seront complétés par la propagation de l'insertion des fonctions Φ à la section suivante. Notez que la variable y est indiquée être modifiée au niveau du bloc B_7 et une fonction Φ, concernant y est alors insérée. Néanmoins, le fait que y soit indiquée comme étant modifiée n'est pas présent au niveau du bloc B_8. Cette insertion sera effective dans la section suivante par la propagation de l'insertion des fonctions Φ. Finalement, la variable z étant modifiée sur les deux branches, la fonction Φ est complète pour cette variable au niveau du bloc B_8.
Le résultat de cette passe ne s'affiche pas. Il est juste répertorié au niveau du champ `VirtualInstruction::mark` des labels `LabelInstruction`. Ce champ est alors de type `PhiInsertionTask::LabelResult` et contient en particulier une fonction d'association, qui à une variable modifiée localement (de type `VirtualExpression`), associe le `GotoInstruction*` précédant le label, par lequel vient la variable modifiée localement.
`std::map<VirtualExpression*,std::pair<GotoInstruction*,GotoInstruction*>,PhiInsertionTask::IsBefore>`
est la structure de données C++ dédiée, sachant que pour trier les variables et éviter les doublons (notamment pour la variable z au niveau du bloc B_8), nous avons besoin d'un ordre total entre les variables. Cet ordre est alors simplement fourni par l'emplacement d'enregistrement dans la portée. Ainsi pour l'exemple considéré, le résultat est

B_7.mark = x → goto$_6$, **NULL**
y → goto$_4$, **NULL**

B_8.mark = x → goto$_1$, **NULL**
t → goto$_7$, **NULL**
z → goto$_1$, goto$_7$

Définition des tâches

1. Définissez dans le fichier `Algorithms.h:6, 90`, les tâches d'insertions des fonctions Φ. Toute l'algorithmique se trouve au niveau du traitement des instructions (voir les dérivés de la méthode `VirtualInstruction::handle`). Les tâches propageant l'insertion des fonctions Φ recensent à partir d'une instruction avec frontière de domination (label, `then` ou `else`), jusqu'à sa frontière de domination effective les variables qui ont été modifiées (voir le champ

`PhiInsertionTask::m_modified`). Pour ne pas gérer des variables qui ont été modifiées plusieurs fois, les variables modifiées sont recensées dans un ensemble trié dont l'ordre `IsBefore` est simplement la position de la variable dans la pile locale. Même si ce n'est pas le cas sur l'implantation, nous considérons que les variables susceptibles d'être modifiées par la SSA, sont les variables locales, mais également les paramètres de fonction, ainsi que les variables globales. L'ordre `IsBefore` considèrera alors les variables globales comme étant plus petites que les paramètres, eux-même plus petits (situés en avant dans la pile locale) que les variables locales.

```
enum TypeTask { TTUndefined, TTPrint, TTDomination, TTPhiInsertion };
class PrintTask { … };
class PrintAgenda { … };
class DominationTask : public VirtualTask { … };
class DominationAgenda : public WorkList { … };

class PhiInsertionTask : public VirtualTask {
 public:
  class IsBefore { 1
   public:
    bool operator()(VirtualExpression* fst, VirtualExpression* snd) const;
  };

  public:
   std::vector<GotoInstruction*>* m_dominationFrontier; 2
   Scope m_scope; 3
   std::set<VirtualExpression*, IsBefore> m_modified; 4
   bool m_isLValue; 5

   typedef std::set<VirtualExpression*, IsBefore> ModifiedVariables;
   class LabelResult { 6
    private:
     typedef std::map<VirtualExpression*, std::pair<GotoInstruction*, GotoInstruction*>, PhiInsertionTask::IsBefore>
        Map;
     Map m_map; 7
     bool m_hasMark; 8
     Scope m_scope; 9
     ModifiedVariables m_variablesToAdd; 10

    public:
     LabelResult() : m_hasMark(false) {}

     void setMark() { m_hasMark = true; }
     void setScope(const Scope& scope) { m_scope = scope; }
     LocalVariableExpression getAfterScopeLocalVariable() const 11
       { return LocalVariableExpression(std::string(), m_scope.count(), m_scope); }
     Map& map() { return m_map; }
     bool hasMark() const { return m_hasMark; }
     const ModifiedVariables& variablesToAdd() const { return m_variablesToAdd; }
     ModifiedVariables& variablesToAdd() { return m_variablesToAdd; }
     typedef Map::iterator iterator;
   };

  public:
   PhiInsertionTask(const Function& function)
     : m_dominationFrontier(NULL), m_scope(function.globalScope()), m_isLValue(false)
     { setInstruction(function.getFirstInstruction()); }
   PhiInsertionTask(GotoInstruction& gotoInstruction, const PhiInsertionTask& source)
     : m_dominationFrontier(&gotoInstruction.getDominationFrontier()),
       m_scope(source.m_scope), m_isLValue(false)
     { setInstruction(gotoInstruction); }
   PhiInsertionTask(const PhiInsertionTask& source)
     : VirtualTask(source), m_scope(source.m_scope), m_modified(source.m_modified), m_isLValue(false) {}

   virtual VirtualTask* clone() const { return new PhiInsertionTask(*this); }
   virtual int getType() const { return TTPhiInsertion; }
};

class PhiInsertionAgenda : public WorkList { 12
 private:
```

```
  std::vector<LabelInstruction*> m_labels;  13

  public:
   PhiInsertionAgenda(const Function& function) { addNewAsFirst(new PhiInsertionTask(function)); }
   virtual ~PhiInsertionAgenda()
     {  for (std::vector<LabelInstruction*>::iterator labelIter = m_labels.begin();
             labelIter != m_labels.end(); ++labelIter) {
           if ((*labelIter)->mark) {
             delete (PhiInsertionTask::LabelResult*) (*labelIter)->mark;
             (*labelIter)->mark = NULL;
           };
        };
        m_labels.clear();
     }

   const std::vector<LabelInstruction*>& labels() const { return m_labels; }
   virtual void markInstructionWith(VirtualInstruction& instruction, VirtualTask& vtTask)
     {  if (instruction.type() == VirtualInstruction::TLabel) {  14
          assert(dynamic_cast<const PhiInsertionTask*>(&vtTask));
          const PhiInsertionTask& task = (const PhiInsertionTask&) vtTask;
          if (instruction.mark == NULL) {
            assert(dynamic_cast<const LabelInstruction*>(&instruction));
            instruction.mark = new PhiInsertionTask::LabelResult();
          };
          PhiInsertionTask::LabelResult& result = *((PhiInsertionTask::LabelResult*) instruction.mark);
          if (!result.hasMark()) {
            m_labels.push_back((LabelInstruction*) &instruction);
            result.setMark();
            result.setScope(task.m_scope);
          };
        };
     }
};
```

1 définit l'ordre (l'opération <) entre les variables globales, les paramètres de fonction, les variables locales. Le but de cet ordre est de garantir l'unicité d'enregistrement d'une variable : il est à fournir à la bibliothèque standard `std::map`. L'implantation de cet ordre se trouve dans le fichier `Algorithms.cpp:47`.

```
bool
PhiInsertionTask::IsBefore::operator()(VirtualExpression* fst, VirtualExpression* snd) const {
  if (fst->type() == snd->type()) {
    if (fst->type() == VirtualExpression::TLocalVariable) {
      assert(dynamic_cast<const LocalVariableExpression*>(fst)
          && dynamic_cast<const LocalVariableExpression*>(snd));
      return ((const LocalVariableExpression&) *fst).getGlobalIndex()
        < ((const LocalVariableExpression&) *snd).getGlobalIndex();
    };
    if (fst->type() == VirtualExpression::TParameter) {
      assert(dynamic_cast<const ParameterExpression*>(fst)
          && dynamic_cast<const ParameterExpression*>(snd));
      return ((const ParameterExpression&) *fst).getIndex()
        < ((const ParameterExpression&) *snd).getIndex();
    };
    assert(dynamic_cast<const GlobalVariableExpression*>(fst)
        && dynamic_cast<const GlobalVariableExpression*>(snd));
    return ((const GlobalVariableExpression&) *fst).getIndex()
      < ((const GlobalVariableExpression&) *snd).getIndex();
  };
  return fst->type() > snd->type();
}
```

Notez qu'étant donné que seules les variables locales sont réellement traitées pour l'insertion des fonctions Φ, il est possible de restreindre l'implantation à

```
return ((const LocalVariableExpression&) *fst).getGlobalIndex()
   < ((const LocalVariableExpression&) *snd).getGlobalIndex();
```

2 correspond à la frontière de domination immédiate de notre bloc d'instruction. Le bloc d'instruction en question est le bloc attaché à l'instruction courante `PhiInsertionTask::getInstruction()`. Lorsque notre tâche passe sur un label, un `then` ou un `else`, cette frontière de domination est mise à jour. Ce champ sert pour

éviter d'insérer les variables modifiées par le bloc B_0 au niveau du bloc B_1, parce que B_1, même s'il est atteignable par B_0 n'est pas dans la frontière de domination de ce dernier.

3 correspond à la portée de notre instruction courante `PhiInsertionTask::getInstruction()`. La portée sert à savoir si une variable existe toujours, au moment où la tâche arrive sur la frontière de domination. C'est notamment le cas pour l'exemple `essai.c` où la variable k, déclarée et modifiée dans un bloc `then` d'un `if then else`, ne doit pas être insérée en tant que fonction Φ au niveau de la frontière de domination, puisqu'étant sorti du bloc `then`, la variable k n'existe plus.

4 représente l'ensemble des variables modifiées par les instructions du bloc d'instructions. Cet ensemble grossit peu à peu lorsque la tâche passe sur les instructions (voire l'interprétation des dérivés de `VirtualInstruction::handle`, comme `ExpressionInstruction::handle`). Il se vide lorsque l'instruction courante atteint la frontière de domination du bloc d'instruction. Ainsi lorsque notre `PhiInsertionTask` est en B_6, notre ensemble inclut la variable `x`. Notre ensemble se vide de ses variables sur le label B_7, et reprend du même coup les variables comme t, stockée sur ce label dans B_7`.mark->m_variablesToAdd` où B_7`.mark` est de type `PhiInsertionTask::LabelResult`. Puis sur cette même instruction B_7, notre ensemble inclut la variable z. Ces variables t et z sont alors transformées en fonctions Φ au niveau du label B_8.

5 représente un booléen. Lorsque ce booléen est à `true`, cela signifie que notre tâche se propage sur la partie gauche d'une affectation, ce qui active le fait d'insérer les variables dans le champ `m_modified`. Ce booléen est mis à `true` lors du passage de notre tâche au niveau des `AssignExpression` sur la sous-expression gauche. Il est ensuite replacé à `false` avant que notre tâche ne passe sur la sous-expression droite.

6 représente le marquage laissé au niveau des frontières de dominations, qui se trouvent être des labels. Ainsi `LabelInstruction::mark` est implicitement de type `LabelResult*`, jusqu'à l'insertion des fonctions Φ. Ce marquage contient les champs suivants :

- 7 indique pour chaque variable modifiée et dont notre label se trouve être à la frontière de domination immédiate de cette modification, le `goto` précédent notre label et par lequel arrive cette modification.
- 8 indique si une tâche `PhiInsertionTask` est effectivement passée sur le label, et ce afin d'éviter d'en propager de nouvelles qui n'apporteront pas plus d'informations. Les `LabelResult` sont en effet créés en avance au niveau des frontières de dominations (pour maintenir le champ `m_variablesToAdd`) ; ainsi leur existence n'est en aucun cas garante d'un passage effective d'une `PhiInsertionTask`. C'est pour cette raison qu'est introduit `m_hasMark`.
- 9 donne la portée du label. Ce champ est mis en place lors du passage de la première tâche `PhiInsertionTask`. Il ne change pas quelles que soient les chemins pour arriver au label.
- 10 sert à stocker les variables modifiées `PhiInsertionTask::m_modified`, qui ont du être temporairement abandonnées pour cause de rentrée dans la branche `then` ou `else` d'une instruction `IfInstruction`. Lors du passage de la `PhiInsertionTask` sur `goto`$_2$, seule t est présente dans `PhiInsertionTask::sveModified`. t se retrouve alors dans le `mvVariablesToAdd` de la frontière de domination de `goto`$_2$, soit B_7. Lorsque la `PhiInsertionTask` rencontre effectivement B_7, `m_variablesToAdd` est transféré dans l'autre sens, dans `PhiInsertionTask::m_modified`.

11 donne le rang de la première variable non accessible par la portée du label. Ce rang est compatible avec l'ordre `IsBefore` et sert à éliminer brutalement des champs comme `PhiInsertionTask::m_modified`, toutes les variables non viables après le label.

12 définit l'agenda qui sert à propager les `PhiInsertionTask` sur tout le graphe de contrôle. Cet agenda contient :

13 répertorie tous les labels, dans un double but. Le premier but est d'assurer que la propagation de l'insertion des fonctions Φ sur la frontière de domination des labels s'effectue efficacement (voir la section suivante). Le second but est de pouvoir nettoyer rapidement le marquage au niveau des labels, ce qui est symbolisé par le destructeur de la classe `PhiInsertionAgenda`.

14 définit l'algorithmique de la méthode `markInstructionWith`. Cette dernière se contente de maintenir à jour le champ `m_labels` correspondant aux labels répertoriés, ainsi que la portée et le marquage des labels.

Implantation de l'algorithmique locale

2. Implantez la méthode de traitement locale pour les variables locales, si ces variables locales sont dans la partie gauche d'une affectation. Cette méthode doit être déclarée dans le fichier `SyntaxTree.h:355`.

```
class LocalVariableExpression : public VirtualExpression {
 private:
  …
 public:
  LocalVariableExpression(const std::string& …, int …, Scope …) : … { … }
  virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
  virtual void print(std::ostream& out) const { … }
  …
};
```

L'implantation de la méthode `LocalVariableExpression::handle` se trouve au niveau du fichier `SyntaxTree.cpp:78` et prend 30 lignes de code. Elle insère notre variable parmi les variables modifiées du bloc d'instructions si nous sommes dans la partie gauche d'une affectation (`task.m_isLValue` positionné sur `true`). Dans ce cas, notre méthode insère une fonction Φ pour notre variable au niveau de chaque frontière de domination. Cette fonction Φ sera complétée lorsque `task` arrivera effectivement sur la frontière de domination.

```
void
LocalVariableExpression::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  int type = vtTask.getType();
  if (type == TTPhiInsertion) {
    assert(dynamic_cast<const PhiInsertionTask*>(&vtTask));
    PhiInsertionTask& task = (PhiInsertionTask&) vtTask;
    if (task.m_isLValue) {
      if (task.m_modified.find(…) == task.m_modified.end())
        task.m_modified.insert(…);

      if (task.m_dominationFrontier) {
        for (std::vector<GotoInstruction*>::const_iterator labelIter = task.m_dominationFrontier->begin();
             labelIter != task.m_dominationFrontier->end(); ++labelIter) {
          LabelInstruction& label = (LabelInstruction&) *((*labelIter)->getSNextInstruction());
          if (!label.mark)
            label.mark = new PhiInsertionTask::LabelResult();
          PhiInsertionTask::LabelResult& result = *((PhiInsertionTask::LabelResult*) label.mark);

          if (result.map().find(this) == result.map().end()) {
            std::pair<VirtualExpression*, std::pair<GotoInstruction*, GotoInstruction*> > insert;
            insert.first = …;
            insert.second.first = NULL;
            insert.second.second = NULL;
            result.map().insert(insert);
          };
        };
      };
```

```
    };
  };
}
```

3. Implantez la méthode de traitement locale pour les affectations en propageant le traitement sur la partie gauche de l'affectation `AssignExpression`. Cette méthode doit être déclarée dans le fichier `SyntaxTree.h:601`.

```
class AssignExpression : public VirtualExpression {
 private:
  …
 public:
  …
  AssignExpression& setRValue(VirtualExpression* pveRValue) { … }
  virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
  virtual void print(std::ostream& osOut) const { … }
  virtual std::auto_ptr<VirtualType> newType(Function* pfFunction) const { … }
};
```

L'implantation de la méthode `AssignExpression::handle` se trouve au niveau du fichier `SyntaxTree.cpp:109` et prend 5 lignes de code.

```
void
AssignExpression::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  int type = vtTask.getType();
  if (type == TTPhiInsertion && m_lvalue.get()) {
    assert(dynamic_cast<const PhiInsertionTask*>(&vtTask));
    PhiInsertionTask& task = (PhiInsertionTask&) vtTask;
    task.m_isLValue = …;
    m_lvalue->handle(task, continuations, reuse);
    task.m_isLValue = …;
  };
}
```

4. Implantez la méthode de traitement locale pour les instructions-affectations en propageant le traitement sur l'affectation. Cette méthode doit être déclarée dans le fichier `SyntaxTree.h:703`.

```
class ExpressionInstruction : public VirtualInstruction {
 private:
  std::auto_ptr<VirtualExpression> apveExpression;

 public:
  ExpressionInstruction() { setType(TExpression); }
  virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
  ExpressionInstruction& setExpression(VirtualExpression* pveExpression) { … }
  virtual void print(std::ostream& osOut) const { … }
};
```

L'implantation de la méthode `ExpressionInstruction::handle` se trouve au niveau du fichier `SyntaxTree.cpp:144` et prend 3 lignes de code.

```
void
ExpressionInstruction::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  int type = vtTask.getType();
  if (type == TTPhiInsertion) {
    if (m_expression.get())
      m_expression->handle(vtTask, continuations, reuse);
  };
  VirtualInstruction::handle(vtTask, continuations, reuse);
}
```

5. Implantez la méthode de traitement local pour les gotos en destination de `IfInstruction`. Cette méthode doit être déclarée dans le fichier `SyntaxTree.h:753`.

```
class GotoInstruction : public VirtualInstruction {
 public:
  …
 private:
  …
 public:
  …
  void connectToLabel(LabelInstruction& liInstruction);
  virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
  virtual bool propagateOnUnmarked(VirtualTask& task, WorkList& continuations,
          Reusability& reuse) const;
```

```
  ...
};
```

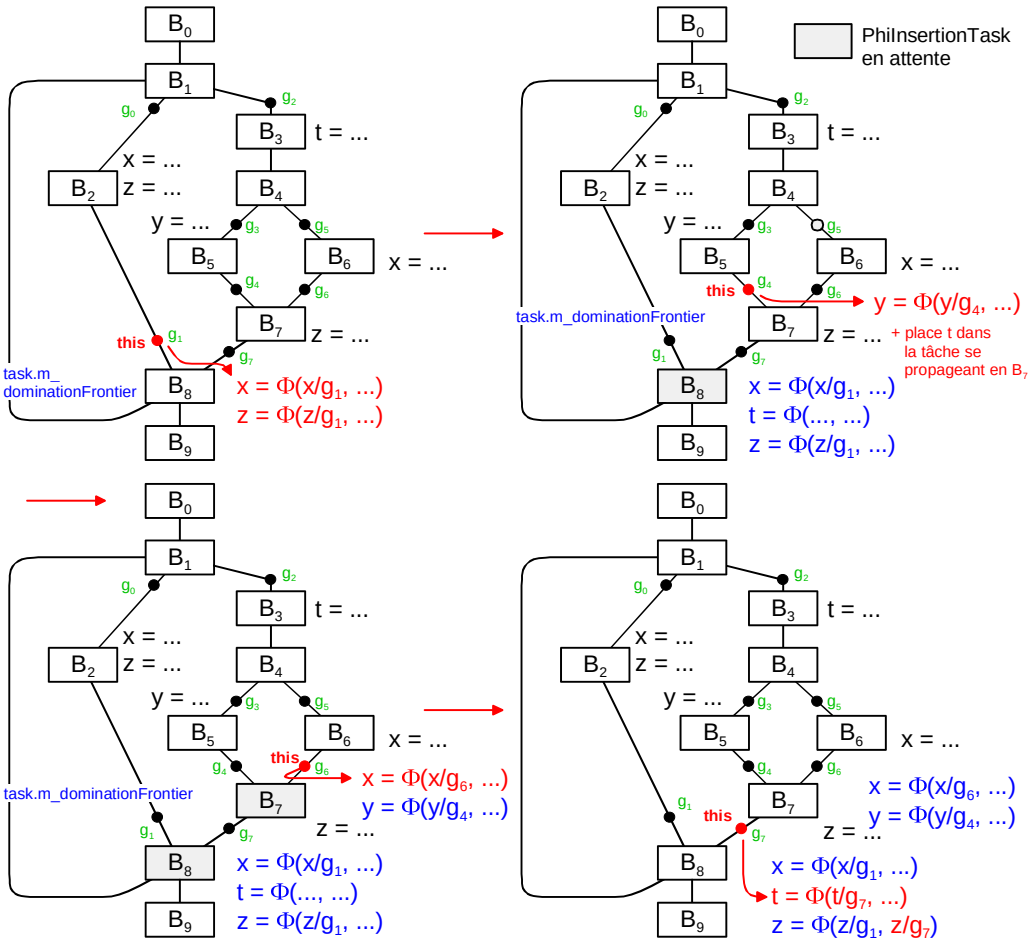
L'implantation de la méthode `GotoInstruction::handle` se trouve au niveau du fichier `SyntaxTree.cpp:154` et prend 20 lignes de code. Les 3 lignes principales sont là pour mettre à jour la frontière de domination au niveau de `task`. Les lignes secondaires traitent le problème des `then` et `else`. Ils doivent laisser temporairement les variables modifiées pour les reprendre lorsque `task` atteint leur frontière de domination. Le stockage temporaire s'effectue naturellement au niveau de chaque label de la frontière de domination du `then` et du `else` dans le champ `label.mark->m_variablesToAdd`.

```
void
GotoInstruction::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  int type = vtTask.getType();
  if ((type == TTPhiInsertion)
      && ((m_context == CAfterIfThen) || (m_context == CAfterIfElse))) {
    assert(dynamic_cast<const PhiInsertionTask*>(&vtTask));
    PhiInsertionTask& task = (PhiInsertionTask&) vtTask;
    task.m_dominationFrontier = …;
    if (getSPreviousInstruction() && getSPreviousInstruction()->type() == TIf) {
      for (std::vector<GotoInstruction*>::const_iterator labelIter = task.m_dominationFrontier->begin();
           labelIter != task.m_dominationFrontier->end(); ++labelIter) {
        LabelInstruction& label = (LabelInstruction&) *(*labelIter)->getSNextInstruction();
        if (!label.mark)
          label.mark = new PhiInsertionTask::LabelResult();
        ((PhiInsertionTask::LabelResult*) label.mark)->variablesToAdd().insert(….begin(), ….end());
      };
    };
  };
  VirtualInstruction::handle(vtTask, continuations, reuse);
}
```

6. Implantez la méthode de propagation pour les gotos en origine de `LabelInstruction`. L'implantation de la méthode `GotoInstruction::propagateOnUnmarked` est transférée de `SyntaxTree.h:754` et se trouve dorénavant au niveau du fichier `SyntaxTree.cpp:176` et prend 50 lignes de code. Elle a pour objectif d'insérer toutes les variables modifiées par la séquence courante d'instructions en tant que fonctions Φ dans le marquage du `LabelInstruction` de type `PhiInsertionTask::LabelResult`. La propagation sur le label suivant n'a lieu que pour les labels non marqués par le passage d'une tâche (voir la méthode `PhiInsertionTask::LabelResult::hasMark`). Noter que cette méthode effectue l'essentiel pour l'insertion des fonctions Φ, ce qui permet à la méthode `PhiInsertionTask::mergeWith` de conserver son implantation triviale à savoir oublier une des deux tâches arrivant simultanément sur le label. C'est possible parce que les méthodes `PhiInsertionAgenda::markInstructionWith` et `LabelInstruction::handle` ne prennent pas en compte les champs propres à la tâche (comme `PhiInsertionTask::m_modified`), mais les champs structurels (du graphe de contrôle, comme `PhiInsertionTask::m_scope`) que la tâche infère.

```
bool
GotoInstruction::propagateOnUnmarked(VirtualTask& vtTask, WorkList& continuations,
   Reusability& reuse) const {
  if (m_context >= CLoop && (vtTask.getType() == TTPhiInsertion)) {
    if (getSNextInstruction() && getSNextInstruction()->type() == TLabel) {
      assert(dynamic_cast<const LabelInstruction*>(getSNextInstruction()));
      LabelInstruction& label = (LabelInstruction&) *getSNextInstruction();
      assert(dynamic_cast<const PhiInsertionTask*>(&vtTask));
      PhiInsertionTask& task = (PhiInsertionTask&) vtTask;
      if (!label.mark)
        label.mark = new PhiInsertionTask::LabelResult();
      bool hasFoundFrontier = false;
      if (task.m_dominationFrontier) {
        for (std::vector<GotoInstruction*>::const_iterator labelIter = task.m_dominationFrontier->begin();
             labelIter != task.m_dominationFrontier->end(); ++labelIter) {
          if ((*labelIter)->getSNextInstruction() == &label) { 1
            PhiInsertionTask::LabelResult& result = *((PhiInsertionTask::LabelResult*) label.mark);
            LocalVariableExpression afterLast(std::string(), task.m_scope.count(), task.m_scope);
            for (PhiInsertionTask::ModifiedVariables::iterator iter = task.m_modified.begin();
```

```
                iter != task.m_modified.end(); ++iter) {
              if (PhiInsertionTask::IsBefore().operator()(*iter, &afterLast)) { 2
                PhiInsertionTask::LabelResult::iterator found = result.map().find(*iter);
                if (found != result.map().end()) { 3
                  if (found->second.first == NULL)
                    found->second.first = …;
                  else if (found->second.second == NULL)
                    found->second.second = …;
                  else { assert(false); }
                };
              };
            };
            hasFoundFrontier = true;
          };
        };
      };
4   if (!((PhiInsertionTask::LabelResult*) label.mark)->hasMark()) {
        task.clearInstruction();
        task.setInstruction(*getSNextInstruction());
        reuse.setReuse();
      };
      if (hasFoundFrontier)
        task.m_modified = ((PhiInsertionTask::LabelResult*) label.mark)->…;
      else if (label.mark) {
        PhiInsertionTask::LabelResult& result = *(PhiInsertionTask::LabelResult*) label.mark;
        task.m_modified.insert(result.….begin(), result.….end());
      };
      return true;
    };
  }
  if ((m_context >= CLoop) || !m_context)
    reuse.setSorted();
  return VirtualInstruction::propagateOnUnmarked(vtTask, continuations, reuse);
}
```



1 se prépare à insérer les fonctions Φ au niveau du label suivant notre `goto`, si et seulement si ce label fait partie de la frontière de domination de notre bloc

d'instructions. Dans ce cas, le booléen `hasFoundFrontier` est placé à **`true`**, ce qui donnera l'ordre à 4 de vider les éléments l'ensemble des éléments modifiés. Ce test est nécessaire pour traiter le cas où nous venons de B_0 pour aller en B_1 ; dans ce cas il ne faut pas insérer de fonctions Φ en B_1 pour les variables modifiées par B_0.

2 regarde pour chaque variable modifiée présente dans `task.m_modified`. Le test indique que « si cette variable est encore dans la portée », alors il faudra placer au niveau du label suivant une fonction Φ concernant cette variable.

3 indique que si la fonction Φ n'est pas déjà insérée, alors il est nécessaire d'en créer une pour notre variable considérée. Dans tous les cas, il faut indiquer que l'origine de la fonction Φ est notre `goto`.

4 indique que la propagation de notre tâche d'insertion se poursuit sur le label si et seulement si le label n'est pas marqué. Il met également à jour l'ensemble des variables modifiées. Ainsi pour une propagation sur bloc B_7, il est nécessaire de relâcher les variables x et y et de reprendre la variable t, qui avait été stockée au moment de prendre une des branches $goto_3$ ou $goto_5$ dans les variables à rajouter du `LabelResult` de la frontière de domination de ces `goto`s, à savoir B_7.

7. Implantez la méthode de traitement local pour les labels lors de l'unique passage des tâches d'insertion des fonctions Φ. L'implantation de la méthode `LabelInstruction::handle` se trouve au niveau du fichier `SyntaxTree.cpp:245`.

```
void
LabelInstruction::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  int type = vtTask.getType();
  if (type == TTDomination) {
    …
  }
  else if (type == TTPhiInsertion) {
    assert(dynamic_cast<const PhiInsertionTask*>(&vtTask));
    ((PhiInsertionTask&) vtTask).m_dominationFrontier = …;
  };
  VirtualInstruction::handle(vtTask, continuations, reuse);
}
```

8. Implantez la méthode de traitement locale pour les entrées de blocs. L'implantation de la méthode `EnterBlockInstruction::handle` est sur `SyntaxTree.cpp:252`.

```
void
EnterBlockInstruction::handle(VirtualTask& virtualTask, WorkList& continuations, Reusability& reuse) {
  VirtualInstruction::handle(virtualTask, continuations, reuse);
  int type = virtualTask.getType();
  if (type == TTPrint) {
    …
  }
  else if (type == TTPhiInsertion) {
    assert(dynamic_cast<const PhiInsertionTask*>(&virtualTask));
    ((PhiInsertionTask&) virtualTask).m_scope = …;
  };
}
```

9. Implantez la méthode de traitement locale pour les sorties de blocs. L'implantation de la méthode `ExitBlockInstruction::handle` se trouve au niveau du fichier `SyntaxTree.cpp:267`, notamment pour supprimer de `task.m_modified` toutes les variables modifiées qui ne sont plus dans la portée courante (supérieures ou égales à `last` pour l'ordre `PhiInsertionTask::IsBefore`).

```
void
ExitBlockInstruction::handle(VirtualTask& virtualTask, WorkList& continuations, Reusability& reuse) {
  int type = virtualTask.getType();
  if (type == TTPrint) {
    …
  }
  else if (type == TTPhiInsertion) {
    assert(dynamic_cast<const PhiInsertionTask*>(&virtualTask));
    PhiInsertionTask& task = (PhiInsertionTask&) virtualTask;
    LocalVariableExpression lastExpr(std::string(), 0, task.m_scope);
```

```
    task.m_scope.…();
    PhiInsertionTask::ModifiedVariables::iterator last = task.m_modified.upper_bound(&lastExpr);
    task.m_modified.erase(…, …);
  };
  VirtualInstruction::handle(virtualTask, continuations, reuse);
}
```

Assemblage

10. Déclarez la méthode `insertPhiFunctions` dans la classe `Program` (fichier `SyntaxTree.h:1061`).

```
class Program {
  …
  void printWithWorkList(std::ostream& osOut) const;
  void computeDominators();
  void computeDominationFrontiers();
  void insertPhiFunctions();
  class ParseContext { … };
}
```

Implantez dette méthode dans le fichier `SyntaxTree.cpp:368`.

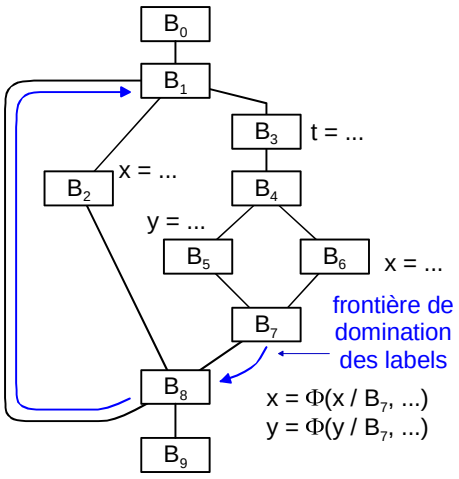
```
void
Program::computeDominationFrontiers(){
  …
}

void
Program::insertPhiFunctions() {
  for (std::set<Function>::const_iterator functionIter = m_functions.begin();
       functionIter != m_functions.end(); ++functionIter) {
    PhiInsertionAgenda phiInsertionAgenda(*functionIter);
    phiInsertionAgenda.execute();
  };
}
```

11. Rajouter dans `main()` (fichier `SyntaxTree.cpp:399`) le code suivant

```
int main(int argc, char** argv ) {
  …
  program.computeDominationFrontiers();
  program.printWithWorkList(std::cout);
  std::cout << std::endl;
  program.insertPhiFunctions();
  program.printWithWorkList(std::cout);
  std::cout << std::endl;
  return 0;
}
```

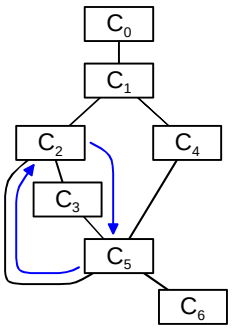
La propagation de l'insertion des fonctions Φ sur la frontière de domination des labels

L'insertion des fonctions Φ est correcte à ce point pour les frontières de domination immédiates. Néanmoins il est nécessaire de continuer la propagation des frontières de domination. Ainsi une fonction Φ est bien insérée pour la variable y en B_7 sur la figure ci-contre. Il reste alors à propager cette insertion en B_8 qui se trouve dans la frontière de domination de B_7, et également en B_1 qui se trouve dans la frontière de domination de B_8. Cette propagation se poursuit jusqu'à atteindre un point fixe (aucune nouvelle variable n'est à insérer), point fixe assez rapide à atteindre, puisque la propagation n'a lieu que sur les labels `LabelInstruction` référencés

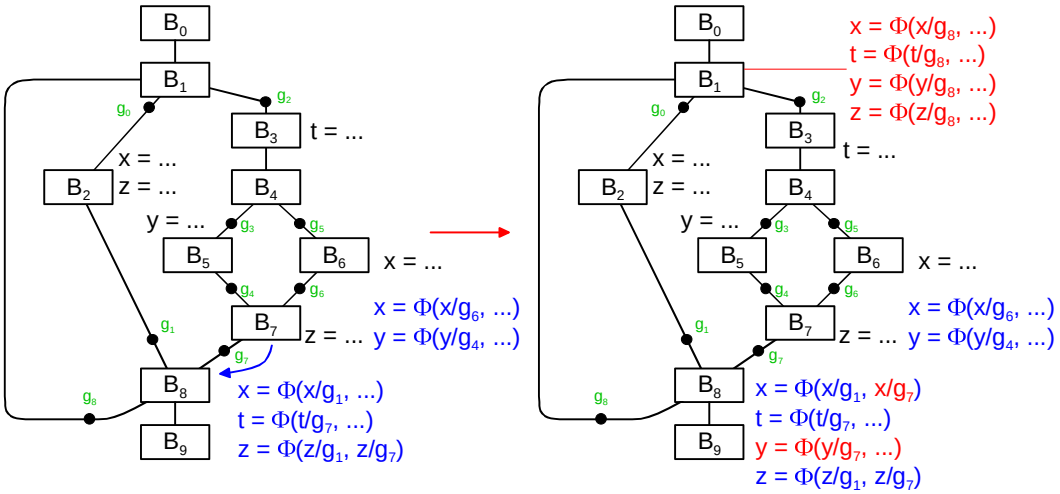


par l'agenda `PhiInsertionAgenda` de la section précédente.

C'est bien un point fixe, car sur l'exemple ci-contre, C_5 est dans la frontière de domination de C_2 et C_2 est dans la frontière de domination de C_5.
Le résultat est une complétion du marquage des labels ((PhiInsertionTask::LabelResult*) LabelInstruction::mark)->map(). Il n'est donc toujours pas visible à l'impression, mais l'est après l'insertion effective des fonctions Φ à la section suivante.



Le résultat de cette phase est dessiné en rouge sur le schéma suivant :



Définition des tâches

1. Définissez dans le fichier `Algorithms.h:6, 179`, les tâches d'insertions des fonctions Φ sur les frontières de domination des labels. L'essentiel de l'algorithmique se trouve dans l'implantation des méthodes `LabelPhiFrontierAgenda::propagate` et `LabelPhiFrontierAgenda::propagateOn`. L'implantation des deux méthodes est relativement similaire et c'est leur contexte d'appel qui les différentie. La méthode `LabelPhiFrontierAgenda::propagate` est appelée sur tous les labels d'une fonction, alors que la méthode `LabelPhiFrontierAgenda::propagateOn` est appelée par propagation successives sur les labels. Les tâches propageant l'insertion des fonctions Φ sur les frontières de domination des labels partent avec un ensemble de variables modifiées `LabelPhiFrontierTask::m_modified` à insérer sur la frontière de domination. Chaque variable modifiée est insérée au niveau des frontières de domination, si elle n'y est pas déjà, ce qui repropage de nouvelles tâches par `LabelPhiFrontierAgenda::propagateOn`, jusqu'à ce que toutes les variables soient dans les frontières de domination des labels.

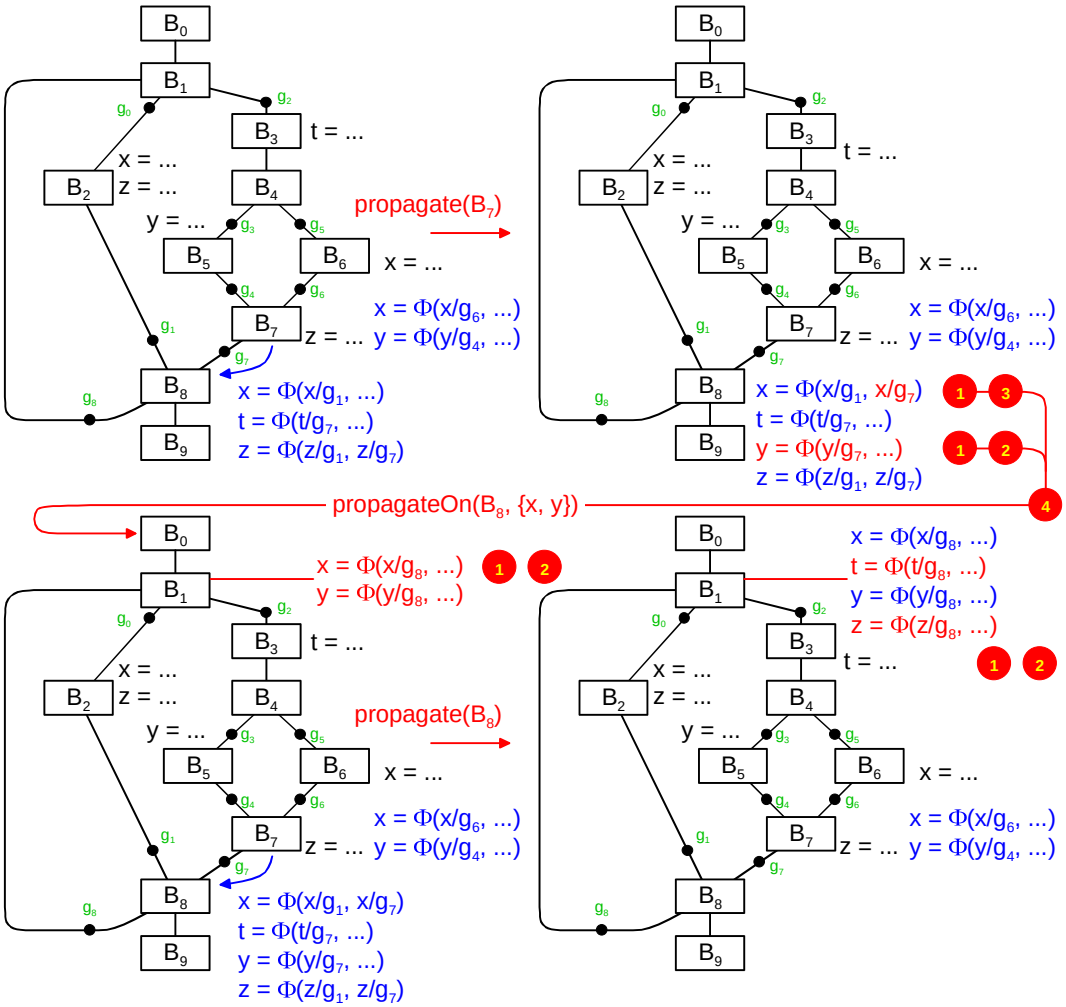
```
enum TypeTask { TTUndefined, …, TTPhiInsertion, TTLabelPhiFrontier };
class PrintTask { … };
class PrintAgenda { … };
class DominationTask : public VirtualTask { … };
class DominationAgenda : public WorkList { … };
class PhiInsertionTask : public VirtualTask { … };
class PhiInsertionAgenda : public WorkList { … };

class LabelPhiFrontierTask : public VirtualTask {
 public:
  typedef PhiInsertionTask::IsBefore IsBefore;
  std::set<VirtualExpression*, IsBefore> m_modified;

 public:
  LabelPhiFrontierTask(const LabelInstruction& label, std::set<VirtualExpression*, IsBefore>& modified)
```

```
    {  setInstruction(label); m_modified.swap(modified); }
  LabelPhiFrontierTask(const LabelPhiFrontierTask& source) : VirtualTask(source) {}

  virtual VirtualTask* clone() const { return new LabelPhiFrontierTask(*this); }
  virtual int getType() const { return TTLabelPhiFrontier; }
};

class LabelPhiFrontierAgenda : public WorkList {
 public:
  LabelPhiFrontierAgenda() {}

  typedef PhiInsertionTask::IsBefore IsBefore;
  typedef PhiInsertionTask::LabelResult LabelResult;
  void propagate(const LabelInstruction& label);
  void propagateOn(const LabelInstruction& label, const std::set<VirtualExpression*, IsBefore>& originModified);
};
```

2. Implantez dans le fichier `Algorithms.cpp:69`, les méthodes `LabelPhiFrontierAgenda::propagate` (appelée par la méthode `Program::insertPhiFunctions`) et `LabelPhiFrontierAgenda::propagateOn` (appelée par la méthode `LabelInstruction::handle`). Ces deux méthodes ont le même objectif qui est d'insérer les fonctions Φ propagées sur les frontières de dominations des labels. Pour la méthode `propagate`, les variables modifiées à l'origine de l'insertion des fonctions Φ sur les frontières des labels proviennent des fonctions Φ déjà présentes sur les labels. Pour la méthode `propagateOn`, les variables modifiées à l'origine de l'insertion des fonctions Φ sur les frontières des labels proviennent des fonctions Φ additionnelles générées par notre algorithmique et passées dans l'argument `originModified`.

```
void
LabelPhiFrontierAgenda::propagate(const LabelInstruction& label) {
  assert(label.mark);
  LabelResult& result = *(LabelResult*) label.mark;
  for (std::vector<GotoInstruction*>::const_iterator frontierIter = label.getDominationFrontier().begin();
       frontierIter != label.getDominationFrontier().end(); ++frontierIter) {
    std::set<VirtualExpression*, IsBefore> modified;
    LabelResult& receiver = *(LabelResult*) (*frontierIter)->getSNextInstruction()->mark;
    for (LabelResult::iterator exprIter = result.map().begin(); exprIter != result.map().end(); ++exprIter) {
      LocalVariableExpression afterScope = receiver.getAfterScopeLocalVariable();
      if (IsBefore().operator()(exprIter->first, &afterScope)) { 1
        LabelResult::iterator found = receiver.map().find(exprIter->first);
        if (found == receiver.map().end()) { 2
          std::pair<VirtualExpression*, std::pair<GotoInstruction*, GotoInstruction*> > insert;
          insert.first = …;
          insert.second.first = …;
          insert.second.second = NULL;
          found = receiver.map().insert(found, insert);
          if (modified.find(…) == modified.end())
            modified.insert(…);
        }
        else if ((found->second.first != …) && (found->second.second != …)) { 3
          if (found->second.first == NULL)
            found->second.first = …;
          else if (found->second.second == NULL)
            found->second.second = …;
          else
            {  assert(false); }
        };
      };
    };
    if (!modified.empty()) 4
      addNewAsFirst(new LabelPhiFrontierTask(
        *(LabelInstruction*) (*frontierIter)->getSNextInstruction(), modified));
  };
}

void
LabelPhiFrontierAgenda::propagateOn(const LabelInstruction& label,
    const std::set<VirtualExpression*, IsBefore>& originModified) {
  for (std::vector<GotoInstruction*>::const_iterator frontierIter = label.getDominationFrontier().begin();
       frontierIter != label.getDominationFrontier().end(); ++frontierIter) {
```

```
    std::set<VirtualExpression*, IsBefore> modified;
    LabelResult& receiver = *(LabelResult*) (*frontierIter)->getSNextInstruction()->mark;
    for (std::set<VirtualExpression*, IsBefore>::iterator exprIter = originModified.begin();
         exprIter != originModified.end(); ++exprIter) {
      LocalVariableExpression afterScope = receiver.getAfterScopeLocalVariable();
      if (IsBefore().operator()(*exprIter, &afterScope)) { 1
        LabelResult::iterator found = receiver.map().find(*exprIter);
        if (found == receiver.map().end()) { 2
          std::pair<VirtualExpression*, std::pair<GotoInstruction*, GotoInstruction*> > insert;
          insert.first = …;
          insert.second.first = …;
          insert.second.second = NULL;
          found = receiver.map().insert(found, insert);
          if (modified.find(*exprIter) == modified.end())
            modified.insert(…);
        }
        else if ((found->second.first != …) && (found->second.second != …)) { 3
          if (found->second.first == NULL)
            found->second.first = …;
          else if (found->second.second == NULL)
            found->second.second = …;
          else
            { assert(false); }
        };
      };
    };
    if (!modified.empty()) 4
      addNewAsFirst(new LabelPhiFrontierTask(*(LabelInstruction*)
            (*frontierIter)->getSNextInstruction(), modified));
  };
}
```



1 teste si la variable modifiée existe encore au niveau de la frontière de domination. Si c'est le cas, soit cette variable est déjà insérée sur le label de la frontière de

domination avec le `GotoInstruction` de la frontière considérée (on ne fait rien pour ne pas propager de nouvelle tâche), soit cette variable est déjà insérée sur le label de la frontière de domination avec un autre `GotoInstruction` (on applique alors le cas 3), soit cette variable n'est pas présente sur le label de la frontière de domination (on applique alors le cas 2).

2 insère une nouvelle fonction Φ sur le label de la frontière de domination. Comme cette nouvelle variable n'était pas déjà présente, elle doit être propagée explicitement par la méthode `propagateOn` et c'est pour cette raison qu'on l'insère parmi les variables `modified`.

3 met à jour la fonction Φ existante associée à la variable `*exprIter` sur le label `receiver` en lui indiquant une nouvelle provenance de modification à savoir le `GotoInstruction` de la frontière de domination. Étant donné que la fonction Φ existait déjà, une propagation de cette dernière a déjà eu lieu, soit par la méthode `propagate`, soit par la méthode `propagateOn`.

4 regarde si au moins une nouvelle fonction Φ a été insérée au niveau du label. Si tel est le cas, alors il faut propager spécifiquement toutes les nouvelles fonctions Φ insérées de frontières de domination en frontière de domination en appelant la méthode `propagateOn` par l'intermédiaire de la méthode `LabelInstruction::handle`.

3. Implantez dans le fichier `SyntaxTree.cpp:249` comment se propage les fonctions Φ sur la frontière de domination des labels. Étant donné que la propagation est effectuée par l'intermédiaire de la méthode `LabelInstruction::handle` (puisque les labels sont les seules instructions autorisées pour cette tâche), il faut éviter d'appeler l'implantation générique `VirtualInstruction::handle` qui propage sur les suivants.

```
void
LabelInstruction::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  int type = vtTask.getType();
  if (tpe == TTDomination) { … }
  else if (type == TTPhiInsertion) { … }
  else if (type == TTLabelPhiFrontier) {
    assert(dynamic_cast<const LabelPhiFrontierTask*>(&vtTask)
      && dynamic_cast<const LabelPhiFrontierAgenda*>(&continuations));
    … // appel à propagateOn
    return;
  };
  VirtualInstruction::handle(vtTask, continuations, reuse);
}
```

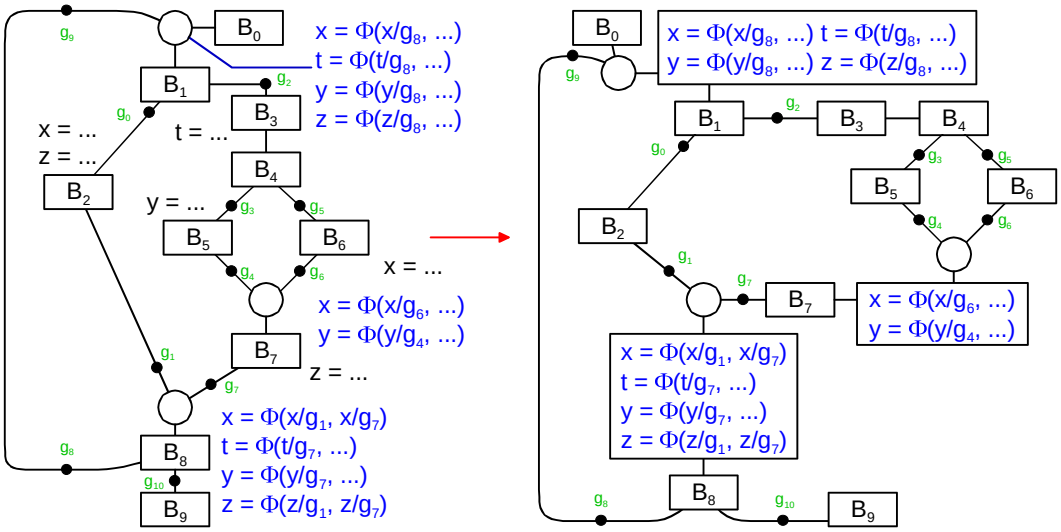
4. Rajoutez dans la fonction `Program::insertPhiFunctions` du fichier `SyntaxTree.cpp:381` la propagation de l'insertion des fonctions Φ sur les frontières de domination.

```
void
Program::insertPhiFunctions() {
  for (std::set<Function>::const_iterator functionIter = m_functions.begin();
      functionIter != m_functions.end(); ++functionIter) {
    PhiInsertionAgenda phiInsertionAgenda(*functionIter);
    phiInsertionAgenda.execute();
    for (std::vector<LabelInstruction*>::const_iterator labelIter = phiInsertionAgenda.labels().begin();
        labelIter != phiInsertionAgenda.labels().end(); ++labelIter) {
      LabelPhiFrontierAgenda labelPhiFrontierAgenda;
      … // appel à propagate
      labelPhiFrontierAgenda.execute();
    };
  };
}
```

L'insertion effective des fonctions Φ

L'insertion effective des fonctions Φ consiste simplement à transformer les fonctions Φ de `((PhiInsertionTask::LabelResult*) LabelInstruction::mark)->map()` en

instructions effectives. Ces instructions sont situées juste après les labels. Ce sont des `ExpressionInstruction` constituées d'une `AssignExpression` dont la partie gauche est la variable de la fonction Φ et dont la partie droite est une nouvelle expression, à savoir une `PhiExpression`.



Mise à jour des structures de données

1. Les expressions Φ ou `PhiExpression` font référence à des variables (`LocalVariableExpression`, `GlobalVariableExpression`, `ParameterExpression`), qu'il est nécessaire de dupliquer pour contrôler la mémoire. Rajoutez un constructeur de recopie et une méthode virtuelle **`virtual`** `VirtualExpression* clone()` **`const`** aux classes `VirtualExpression`, `LocalVariableExpression`, `ParameterExpression`, `GlobalVariableExpression` dans le fichier `SyntaxTree.h:298, 356, 378, 396`.

```
class VirtualExpression {
 public:
  enum Type { …};
  …
 public:
  VirtualExpression() : m_type(TUndefined) {}
  VirtualExpression(const VirtualExpression& source) : m_type(source.m_type) {}

  virtual ~VirtualExpression() {}
  virtual VirtualExpression* clone() const { assert(false); return NULL; }
  Type type() const { return tType; }
  …
};

class LocalVariableExpression : public VirtualExpression {
 private:
  …
 public:
  LocalVariableExpression(const std::string& name, int localIndex, Scope scope) : … { … }
  LocalVariableExpression(const LocalVariableExpression& source)
     :  VirtualExpression(source), m_scope(source.m_scope),
        m_name(source.m_name), m_localIndex(source.m_localIndex) {}

  virtual void handle(VirtualTask& vtTask, WorkList& wcContinuations, Reusability& rReuse);
  virtual VirtualExpression* clone() const { return new LocalVariableExpression(*this); }
  virtual void print(std::ostream& osOut) const { … }
  …
};

class ParameterExpression : public VirtualExpression {
 private:
  …
 public:
  ParameterExpression(const std::string& name, int localIndex) : … { … }
  ParameterExpression(const ParameterExpression& source)
```

```
    : VirtualExpression(source), m_name(source.m_name), m_localIndex(source.m_localIndex) {}

  virtual VirtualExpression* clone() const { return new ParameterExpression(*this); }
  int getIndex() const { … }
  …
};

class GlobalVariableExpression : public VirtualExpression {
 private:
  …
 public:
  GlobalVariableExpression(const std::string& name, int localIndex) : … { … }
  GlobalVariableExpression(const GlobalVariableExpression& source)
    : VirtualExpression(source), m_name(source.m_name), m_localIndex(source.m_localIndex) {}

  virtual VirtualExpression* clone() const { return new GlobalVariableExpression(*this); }
  const int& getIndex() const { … }
  …
};
```

2. Rajoutez une méthode `VirtualInstruction::disconnectNext` dans le fichier `SyntaxTree.h:694`. Cette méthode déconnecte notre instruction de son instruction suivante. Le résultat est l'instruction suivante déconnectée. Cette méthode est utilisée pour insérer les instructions correspondant aux fonctions Φ qui commencent par déconnecter les labels de leurs suivants avant de reconnecter par la méthode `VirtualInstruction::connectTo` le label à l'instruction fonction Φ et l'instruction fonction Φ à l'instruction qui suivait le label avant la déconnexion.

   Complétez cette définition par la méthode `void Function::insertNewInstructionAfter( VirtualInstruction* newInstruction, VirtualInstruction& previous)` dans le fichier `SyntaxTree.h:1012`.

```
class VirtualInstruction {
 public:
  …
  void connectTo(VirtualInstruction& next)
    { m_next = &next; next.m_previous = this; }
  VirtualInstruction* disconnectNext()
    { VirtualInstruction* result = m_next;
      if (m_next) {
        assert(m_next->m_previous == this);
        m_next->m_previous = …;
        m_next = …;
      };
      return result;
    }
  virtual void print(std::ostream& out) const = 0;
  …
};
…
class Function {
 private:
  …
 public:
  …
  void addNewInstructionAfter(VirtualInstruction* newInstruction, VirtualInstruction& previous) { … }
  void insertNewInstructionAfter(VirtualInstruction* newInstruction, VirtualInstruction& previous)
    { newInstruction->setRegistrationIndex(m_instructions.size());
      m_instructions.push_back(newInstruction);
      VirtualInstruction* next = previous.…();
      previous.connectTo(…);
      if (next)
        newInstruction->connectTo(…);
    }
  void setDominationFrontier();
  …
};
```

Définition des expressions Φ

3. Les expressions Φ ou `PhiExpression` sont définies comme des affectations d'une variable à elle-même, mais de provenance différente. Après la phase de renommage, les variables affectées seront différentes des variables à droite de l'affectation. Elles sont ainsi représentées par x = Φ(x/goto$_1$, x/goto$_2$) ou par x = Φ(x/goto$_1$, …). Après la phase de renommage, elles sont représentées par `x3 = Φ(x1/goto1, x2/goto2)` ou par `x3 = Φ(x1/goto1, x2/NULL)`.

   Rajoutez la classe `PhiExpression` dans le fichier `SyntaxTree.h:631` en permettant de reconnaître les expressions Φ à partir de la classe `VirtualExpression` du fichier `SyntaxTree.h:281`. Implantez la méthode `PhiExpression::print` dans le fichier `SyntaxTree.cpp:137`.

```
class VirtualExpression {
  public:
   enum Type
     {  TUndefined, TConstant, TChar, TString, TLocalVariable, TParameter, TGlobalVariable,
        TComparison, TUnaryOperator, TBinaryOperator, TDereference, TCast, TFunctionCall,
        TAssign, TPhi
     };
   typedef WorkList::Reusability Reusability;
   …
};
…
class AssignExpression : public VirtualExpression {
   …
};

class GotoInstruction;
class PhiExpression : public VirtualExpression {
  private:
   std::auto_ptr<VirtualExpression> m_fst;
   GotoInstruction* m_fstFrom;
   std::auto_ptr<VirtualExpression> m_snd;
   GotoInstruction* m_sndFrom;

  public:
   PhiExpression() : m_fstFrom(NULL), m_sndFrom(NULL) { setType(TPhi); }

   PhiExpression& addReference(VirtualExpression* expression, GotoInstruction& gotoInstruction)
     {  if (m_fst.get()) {
          assert(!m_snd.get());
          m_snd.reset(expression);
          m_sndFrom = &gotoInstruction;
        }
        else {
          m_fst.reset(expression);
          m_fstFrom = &gotoInstruction;
        };
        return *this;
     }

   virtual void print(std::ostream& out) const;
   virtual std::auto_ptr<VirtualType> newType(Function* function) const
     {  return m_fst.get() ? m_fst->newType(function) : m_snd->newType(function); }
};

// fichier SyntaxTree.cpp:137
void
PhiExpression::print(std::ostream& out) const {
   out << "phi(" ;
   if (m_fst.get()) {
     m_fst->print(out);
     if (m_fstFrom)
       out << ' ' << m_fstFrom->getRegistrationIndex();
   }
   else
     out << "no-expr";
   out << ", ";
   if (m_snd.get()) {
     m_snd->print(out);
```

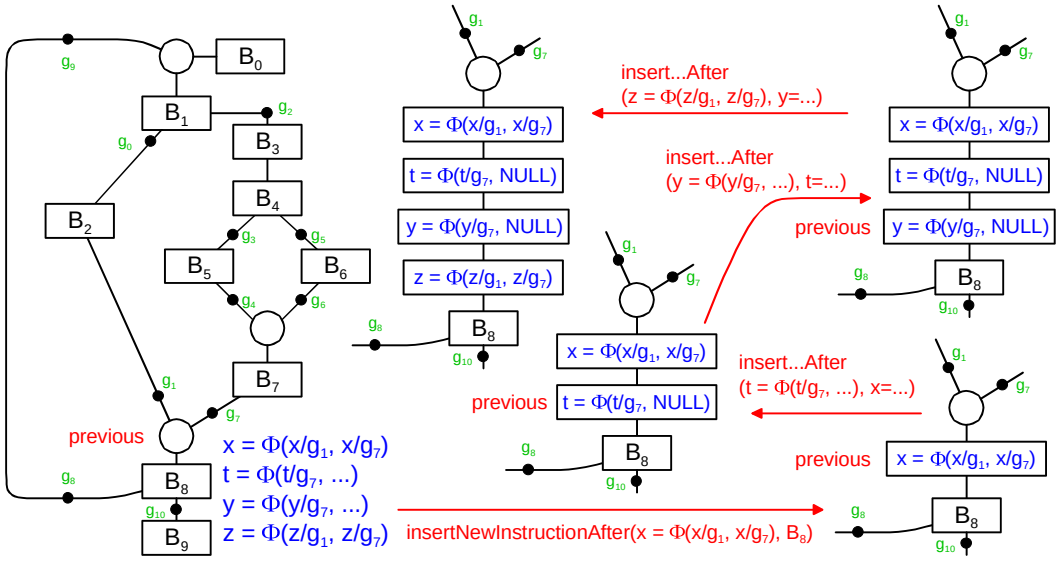
```
    if (m_sndFrom)
      out << ' ' << m_sndFrom->getRegistrationIndex();
  }
  else
    out << "no-expr";
  out << ')';
}
```

4. Rajoutez une méthode publique d'accès aux champs `AssignExpression::m_lvalue` et `AssignExpression::m_rvalue` dans le fichier `SyntaxTree.h:615`. Cette méthode permet de savoir si une `ExpressionInstruction` est une fonction Φ. Implantez alors la méthode `ExpressionInstruction::isPhi` dans le fichier `SyntaxTree.h:757`.

```
class AssignExpression : public VirtualExpression {
 private:
  std::auto_ptr<VirtualExpression> m_lvalue;
  std::auto_ptr<VirtualExpression> m_rvalue;

 public:
  …
  AssignExpression& setLValue(VirtualExpression* lvalue) { m_lvalue.reset(lvalue); return *this; }
  AssignExpression& setRValue(VirtualExpression* rvalue) { m_rvalue.reset(rvalue); return *this; }
  const VirtualExpression& getLValue() const { return *m_lvalue; }
  const VirtualExpression& getRValue() const { return *m_rvalue; }

  virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
  …
};
…
class ExpressionInstruction : public VirtualInstruction {
 private:
  …
 public:
  ExpressionInstruction() { setType(TExpression); }
  bool isPhi() const
    {  return (…) && (…); }

  virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
  …
};
```

Implantation de l'algorithmique

5. Déclarez dans le fichier `SyntaxTree.h:1056` et implantez dans le fichier `SyntaxTree.cpp:369` la méthode `void Function::insertPhiFunctions(const std::vector<LabelInstruction*>& labels)`. Cette méthode parcourt tous les labels passés en arguments et définit l'ensemble des fonctions Φ à insérer comme étant `((PhiInsertionTask::LabelResult*) label.mark)->map()`. Pour chaque variable et sa fonction Φ dans cet ensemble, nous définissons une instruction expression `ExpressionInstruction` correspondant à la fonction Φ et nous insérons cette instruction après l'instruction courante. L'instruction courante pour la première insertion est le label, et pour les insertions suivantes l'instruction précédemment insérée.




```
void
Function::insertPhiFunctions(const std::vector<LabelInstruction*>& labels) {
  for (std::vector<LabelInstruction*>::const_iterator iter = labels.begin(); iter != labels.end(); ++iter) {
    LabelInstruction& label = **iter;
    if (label.mark != NULL) {
      PhiInsertionTask::LabelResult* result = (PhiInsertionTask::LabelResult*) label.mark;
      VirtualInstruction* previous = &label;
      for (PhiInsertionTask::LabelResult::iterator labelIter = result->map().begin();
           labelIter != result->map().end(); ++labelIter) {
        if (labelIter->second.first || labelIter->second.second) {
          ExpressionInstruction* assign = new ExpressionInstruction();
          insertNewInstructionAfter(…, …);
          previous = …;
          AssignExpression* assignExpression = new AssignExpression();
          assign->setExpression(assignExpression);
          VirtualExpression* lvalue = labelIter->first->clone();
          assignExpression->setLValue(lvalue);
          PhiExpression* phi = new PhiExpression();
          assignExpression->setRValue(phi);
          if (labelIter->second.first)
            phi->addReference(labelIter->first->clone(), *labelIter->second.first);
          if (labelIter->second.second)
            phi->addReference(labelIter->first->clone(), *labelIter->second.second);
        };
      };
      delete result;
      label.mark = NULL;
    };
  };
}
```

6. Appelez la méthode `Function::insertPhiFunctions` à partir de la méthode existante `Program::insertPhiFunctions` dans le fichier `SyntaxTree.cpp:439`.

```
void
Program::insertPhiFunctions() {
  for (std::set<Function>::const_iterator functionIter = m_functions.begin();
       functionIter != m_functions.end(); ++functionIter) {
    PhiInsertionAgenda phiInsertionAgenda(*functionIter);
    phiInsertionAgenda.execute();
    for (std::vector<LabelInstruction*>::const_iterator labelIter = phiInsertionAgenda.labels().begin();
         labelIter != phiInsertionAgenda.labels().end(); ++labelIter) {
      …
    };
    const_cast<Function&>(*functionIter).insertPhiFunctions(…);
  };
}
```

Vérifiez que le résultat sur `essai.c` est la sortie suivante :

```
function f
0   {
1     return ([parameter 0: x] + 2);

function main
```

```
0   {0x5f4ed0 x := 0    0x5f5080 y := 1
1     [local 0: x] = [parameter 0: argc];
2     [local 1: y] = 2;
3     [local 0: x] = ([local 0: x] * [local 0: x]);
4     goto label 0x5f5368
5     label 5f5368      dominated by 4 goto label 0x5f5368
       domination frontier of label = 5
28    [local 0: x] = phi([local 0: x] 11, no-expr);
6     {
7       [local 0: x] = ([local 0: x] + 2);
8     }
9     if (([local 0: x] < 100))
10    then      domination frontier = 5
11    goto loop 5f5368
12    else
13    if (([local 0: x] > [local 1: y]))
14    then      domination frontier = 24
15    {0x5f5718 k := 0
16      [local 0: k] = 8;
17      [local 0: x] = ((4 + ([local 0: x] * 2)) - 1);
18    }
25    goto label 0x5f5c60
19    else      domination frontier = 24
20    {
21      [local 0: x] = (([local 0: x] + 1) - f([local 1: y]));
22    }
23    goto label 0x5f5c60
24    label 5f5c60      dominated by 13 if (([local 0: x] > [local 1: y]))
29    [local 0: x] = phi([local 0: x] 25, [local 0: x] 23);
26    return ([local 0: x] + 3);
```

Le résultat sur `essai2.c` est la sortie suivante :

```
function main
0   {0x5f4cd8 x := 0    0x5f4e98 y := 1
1     [local 0: x] = [parameter 0: argc];
2     [local 0: x] = (([local 0: x] * [local 0: x]) - (2 * [local 0: x]));
3     [local 1: y] = (4 + [parameter 0: argc]);
4     goto label 0x5f5250
5     label 5f5250      dominated by 4 goto label 0x5f5250
       domination frontier of label = 5
39    [local 0: x] = phi([local 0: x] 35, no-expr);
40    [local 1: y] = phi([local 1: y] 35, no-expr);
6     {
7       if (([local 0: x] > 2))
8       then    domination frontier = 29
9       {
10        [local 0: x] = ([local 0: x] - 1);
11      }
30      goto label 0x5f5bd0
12      else    domination frontier = 29
13      {
14        if (((2 * [local 0: x]) > [local 1: y]))
15        then  domination frontier = 24
16        {
17          [local 1: y] = 2;
18        }
25        goto label 0x5f5a68
19        else  domination frontier = 24
20        {
21          [local 0: x] = ([local 0: x] - 1);
22        }
23        goto label 0x5f5a68
24        label 5f5a68  dominated by 14 if (((2 * [local 0: x]) > [local 1: y]))
       domination frontier of label = 29
43        [local 0: x] = phi([local 0: x] 23, no-expr);
44        [local 1: y] = phi([local 1: y] 25, [local 1: y] 23);
26        [local 0: x] = ([local 0: x] - 1);
27      }
28      goto label 0x5f5bd0
29      label 5f5bd0    dominated by 7 if (([local 0: x] > 2))
       domination frontier of label = 5
41      [local 0: x] = phi([local 0: x] 30, [local 0: x] 28);
42      [local 1: y] = phi([local 1: y] 28, no-expr);
31      [local 0: x] = ([local 0: x] - 1);
32    }
33    if (([local 0: x] > 0))
34    then      domination frontier = 5
35    goto loop 5f5250
```

```
36    else
37    return 0;
```

Le résultat sur `essai3.c` est la sortie suivante :

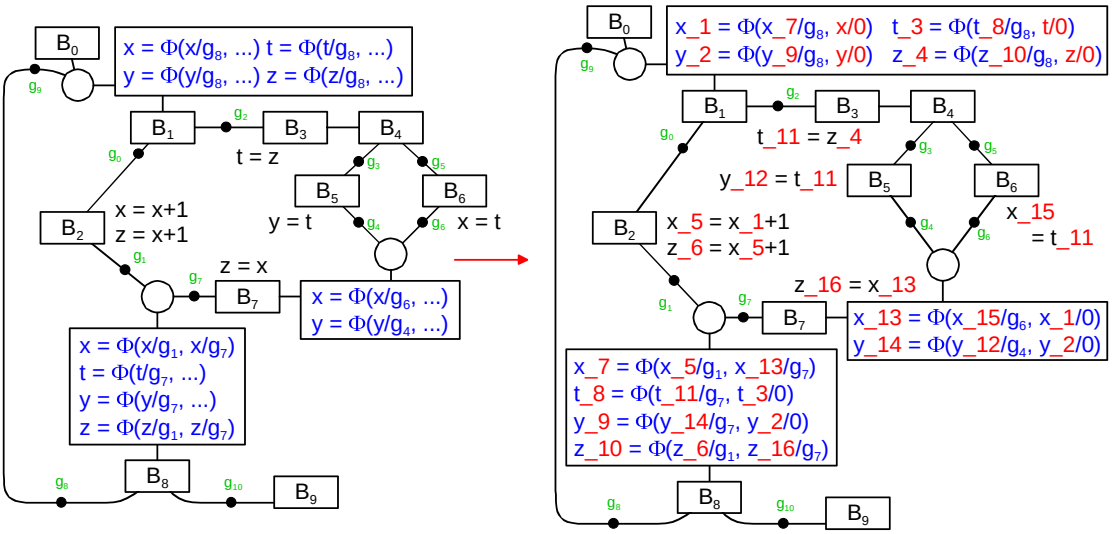
```
function main
0   {0x5f4cd8 x := 0    0x5f4e98 y := 1
1     [local 0: x] = [parameter 0: argc];
2     [local 0: x] = (([local 0: x] * [local 0: x]) - (2 * [local 0: x]));
3     [local 1: y] = (4 + [parameter 0: argc]);
4     goto label 0x5f5250
5     label 5f5250      dominated by 4 goto label 0x5f5250
       domination frontier of label = 5
39    [local 0: x] = phi([local 0: x] 35, no-expr);
40    [local 1: y] = phi([local 1: y] 35, no-expr);
6     {
7       if (([local 0: x] > 2))
8       then    domination frontier = 29
9       {
10        [local 0: x] = ([local 0: x] - 1);
11      }
30      goto label 0x5f5ba0
12      else    domination frontier = 29
13      {
14        [local 0: x] = ([local 0: x] - 1);
15        if (((2 * [local 0: x]) > [local 1: y]))
16        then  domination frontier = 25
17        {
18          [local 1: y] = 2;
19        }
26        goto label 0x5f5ae0
20        else  domination frontier = 25
21        {
22          [local 1: y] = 3;
23        }
24        goto label 0x5f5ae0
25        label 5f5ae0  dominated by 15 if (((2 * [local 0: x]) > [local 1: y]))
        domination frontier of label = 29
43        [local 1: y] = phi([local 1: y] 26, [local 1: y] 24);
27      }
28      goto label 0x5f5ba0
29      label 5f5ba0    dominated by 7 if (([local 0: x] > 2))
        domination frontier of label = 5
41      [local 0: x] = phi([local 0: x] 30, [local 0: x] 28);
42      [local 1: y] = phi([local 1: y] 28, no-expr);
31      [local 0: x] = ([local 0: x] - 1);
32    }
33    if (([local 0: x] > 0))
34    then      domination frontier = 5
35    goto loop 5f5250
36    else
37    return 0;
```

Le renommage des expressions

Une fois les fonctions Φ insérées, tout est défini pour le renommage final. Ce renommage donne à chaque affectation de variable un nouveau nom de variable, sauf si cette variable n'était pas déjà définie. Le renommage est effectif lorsqu'à chaque déclaration de variable locale définie dans la portée, est associée une unique instruction de définition.

Les tâches de renommage parcourent le graphe de contrôle et contiennent à chaque instant une fonction (`std::map`) qui à une variable à renommer associe la nouvelle variable. Le renommage est organisé en pile, c'est-à-dire que chaque branche `then` ou `else` introduit un nouveau niveau alors que les `goto`s vers un label dépilent un ou plusieurs niveaux. La recherche d'une variable qui peut être à renommer part du niveau courant et remonte de niveau en niveau. Si aucun renommage n'est trouvé, alors la variable n'est pas changée. Si un renommage est trouvé, alors c'est le premier renommage (niveau le plus haut) qui sert au remplacement de la variable. Notez que pour les fonctions Φ, le cas `x = Φ(x/g, …)` remplace le … par le renommage de la variable x, recherché à partir du niveau supérieur au niveau courant (dominateur du label). Ainsi sur l'exemple ci-dessous, `y = Φ(y/g7, …)` est remplacé par `y_15 = Φ(y_11/g7, y_2/NULL)` avec `y_15` une nouvelle variable, copie de la

variable `y`, `y_11` correspondant au renommage de `y` à partir du renommage courant, `y_2` correspondant au renommage de `y` à partir du renommage supérieur au renommage courant. L'implantation courante introduit une pile locale pour chaque variable renommée (ensemble de variables renommées organisées en pile). Cette implantation s'oppose à une pile d'ensemble de variables renommées. Chacune des deux possibilités d'implantation a ses avantages et ses inconvénients.



Mise à jour des structures de données

1. Mettez à jour les classes `SymbolTable` et `Scope` dans le fichier `SyntaxTree.h:94, 99, 118, 161`. Cette mise à jour consiste à associer à chaque déclaration de variable une unique instruction de définition qui est généralement sous la forme d'une affectation. La méthode `SymbolTable::setSizeDefinitions` associe à chaque déclaration locale de variable une unique définition initialement définie à `NULL`. Elle permet à la méthode `SymbolTable::setSSADefinition` de fonctionner correctement en lui permettant d'accéder à une zone mémoire réservée `avtUniqueDefinitions[…]`. Les méthodes `SymbolTable::…Definition` et `Scope::…Definition` sont utilisées pour mettre à jour les déclarations de variables introduites par le programme. La méthode `SymbolTable::addSSADeclaration` appelée par l'intermédiaire de `Scope::addSSADeclaration` ajoute la déclaration d'une nouvelle variable comme `x_13`. Cette méthode est implantée sur la question suivante.

```
class Scope;
class Function;
class VirtualInstruction;
class SymbolTable {
  private:
   std::map<std::string, int> m_localIndexes;
   std::vector<VirtualType*> m_types;
   std::vector<VirtualInstruction*> m_uniqueDefinitions;
   std::shared_ptr<SymbolTable> m_parent;

  public:
   …
   const VirtualType& getType(int uIndex) const { return *avtTypes[uIndex]; }

   void setSizeDefinitions()
     {  m_uniqueDefinitions.reserve(m_types.size());
        for (std::vector<VirtualType*>::const_iterator iter = m_types.begin();
             iter != m_types.end(); ++iter)
          m_uniqueDefinitions.push_back(NULL);
     }
   int addSSADeclaration(std::string& name, int localIndex, int& ident);
   bool hasSSADefinition(int localIndex) const
     {  return m_uniqueDefinitions[localIndex] != NULL; }
```

```
  void setSSADefinition(int localIndex, VirtualInstruction& instruction)
    {  m_uniqueDefinitions[localIndex] = &instruction; }

  friend class Scope;
  …
};

class Scope {
 private:
  …
 public:
  …
  void addDeclaration(const std::string& name, VirtualType* type) { … }
  void setSizeDefinitions() { m_last->setSizeDefinitions(); }
  int addSSADeclaration(std::string& name, int localIndex, int& ident)
    {  return m_last->addSSADeclaration(name, localIndex, ident); }
  bool hasSSADefinition(int localIndex) const
    {  return m_last->hasSSADefinition(localIndex); }
  void setSSADefinition(int localIndex, VirtualInstruction& instruction)
    {  m_last->setSSADefinition(localIndex, instruction); }
  FindResult find(const std::string& name, int& local, Scope& scope) const;
  …
};
```

2. Implantez la méthode `SymbolTable::addSSADeclaration` dans le fichier `SyntaxTree.cpp:8, 12`. Cette méthode définit, pour une variable déclarée qui s'appelait « name » = `xyz` et dont le type est `m_types[localIndex]`, une nouvelle variable (identifiée par son indice local retourné par la fonction et le changement de nom `name`) avec un nouveau nom `xyz_36` où `36` est un nombre unique fourni par `index` (il est incrémenté pour pouvoir fournir un nouveau nombre unique). La définition de cette nouvelle variable déclarée sera ensuite définie par l'intermédiaire de la méthode `SymbolTable::setSSADefinition`.

```
…
extern Program::ParseContext ppcParseContext;
extern FILE *yyin;

#include <sstream>

int yyparse ();

int
SymbolTable::addSSADeclaration(std::string& name, int localIndex, int& ident) {
  std::stringstream out;
  out << name << '_' << ident;
  ++ident;
  m_localIndexes.insert(std::pair<std::string, int>(name = out.str(), m_types.size()));
  m_types.push_back(m_types[localIndex]);
  m_uniqueDefinitions.push_back(NULL);
  return m_types.size()-1;
}
```

3. Rajoutez un accès en lecture à la classe `LocalVariableExpression::m_scope` dans le fichier `SyntaxTree.h:388`.

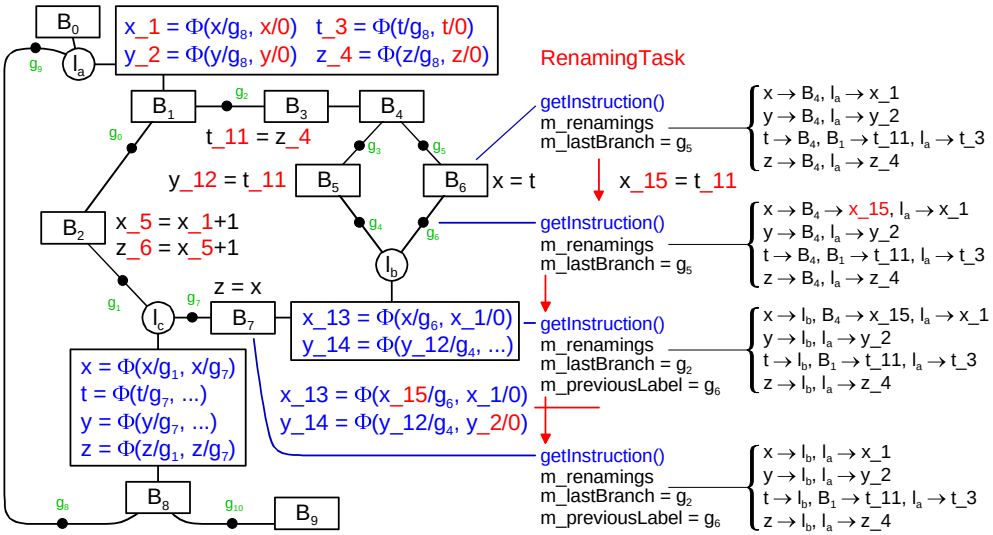
```
class LocalVariableExpression : public VirtualExpression {
 private:
  …
 public:
  …
  int getGlobalIndex() const { return m_scope.getFunctionIndex(localIndex); }
  Scope& scope() { return m_scope; }
  virtual std::auto_ptr<VirtualType> newType(Function* function) const { … }
};
```

Définition des tâches

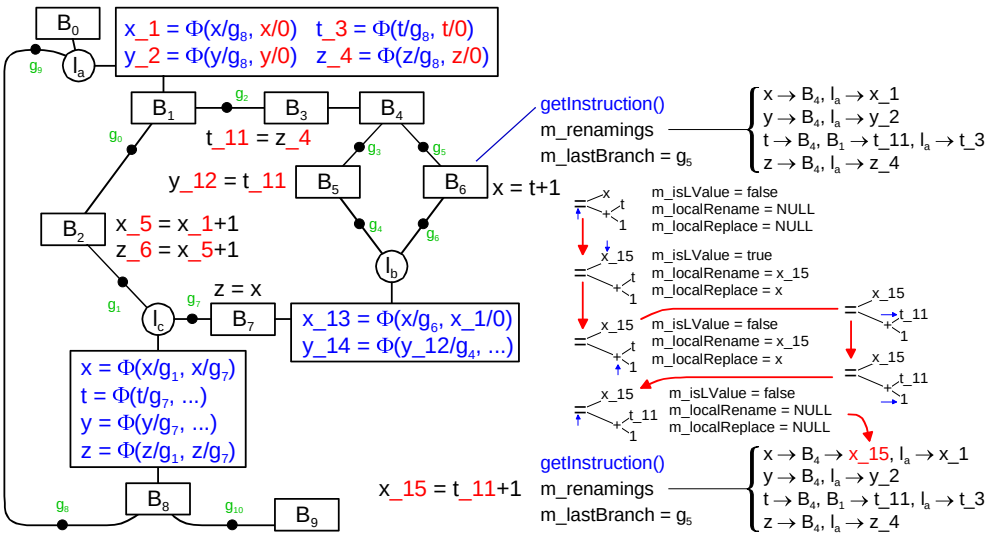
4. Définissez la classe `RenamingTask` dans le fichier `Algorithm.h:6, 203`. Cette classe correspond aux tâches de renommage qui parcourent l'intégralité d'un graphe de contrôle. Chaque instruction du graphe de contrôle n'est parcourue qu'une et une seule fois, sauf les fonctions Φ (`ExpressionInstruction` tel que la méthode `isPhi()`

retourne `true`) et les labels qui sont parcourus deux fois (pour chaque `goto` précédent le label). Cette phase de renommage est une phase assez lourde qu'il est intéressant de ne pas stabiliser par point fixe, mais par simple parcourt sur chaque instruction. Noter une caractéristique importante des tâches de renommage : elles ne se fusionnent pas entre elles (la méthode `mergeWith` retourne **`false`**), car les fonctions Φ doivent être mises à jour à partir des deux `goto`s. Par contre une des deux tâches meure naturellement à la fin de la propagation sur les fonctions Φ alors que l'autre tâche continue naturellement.

Le contenu d'une tâche de renommage au niveau instruction est explicité sur le schéma suivant :



Le contenu d'une tâche de renommage au niveau expression est défini par :



…

```
#include "SyntaxTree.h"

enum TypeTask { TTUndefined, TTPrint, TTDomination, TTPhiInsertion, TTLabelPhiFrontier, TTRenaming };

…
class LabelPhiFrontierAgenda : public WorkList {
  …
} ;

/* Tâche de renommage */

class RenamingAgenda;
class RenamingTask : public VirtualTask {
  public:
```

```
class VariableRenaming { 1
  private:
   VirtualExpression* m_toReplace; 2
   LocalVariableExpression* m_newValue; 3
   Function* m_function; 4
   VirtualInstruction* m_lastDominator; 5
   std::shared_ptr<VariableRenaming> m_parent; 6

  public:
   VariableRenaming(VirtualExpression& locate, const Function& function)
     : m_toReplace(&locate), m_newValue(NULL), m_function(&const_cast<Function&>(function)) {}
   VariableRenaming(VirtualExpression* toReplace, LocalVariableExpression* newValue,
        const Function& function, VirtualInstruction* lastDominator)
     : m_toReplace(toReplace), m_newValue(newValue), m_function(&const_cast<Function&>(function)),
       m_lastDominator(lastDominator) {}
   class Transfert {};
   VariableRenaming(const VariableRenaming& source, Transfert)
     : m_toReplace(NULL), m_newValue(source.m_newValue),
       m_function(source.m_function), m_lastDominator(source.m_lastDominator)
     { const_cast<VariableRenaming&>(source).m_newValue = NULL; }
   VariableRenaming(const VariableRenaming& source)
     : m_toReplace(source.m_toReplace), m_newValue(source.m_newValue),
       m_function(source.m_function), m_lastDominator(source.m_lastDominator),
       m_parent(source.m_parent) {}
   VariableRenaming& operator=(const VariableRenaming& source)
     { m_toReplace = source.m_toReplace;
       m_newValue = source.m_newValue;
       m_function = source.m_function;
       m_lastDominator = source.m_lastDominator;
       m_parent = source.m_parent;
       return *this;
     }

   void cloneExpressions()
     { m_toReplace = m_toReplace->clone();
       if (m_newValue)
         m_newValue = (LocalVariableExpression*) m_newValue->clone();
     }
   Comparison compare(const VariableRenaming& source) const;
   bool operator==(const VariableRenaming& source) const { return compare(source) == CEqual; }
   bool operator<(const VariableRenaming& source) const { return compare(source) == CLess; }
   bool operator>(const VariableRenaming& source) const { return compare(source) == CGreater; }
   bool operator<=(const VariableRenaming& source) const
     { Comparison result = compare(source); return result == CLess || result == CEqual; }
   bool operator>=(const VariableRenaming& source) const
     { Comparison result = compare(source); return result == CGreater || result == CEqual; }
   bool operator!=(const VariableRenaming& source) const
     { Comparison result = compare(source); return result == CGreater || result == CLess; }
   void free()
     { if (m_toReplace) {
         delete m_toReplace;
         m_toReplace = NULL;
       };
       if (m_newValue) {
         delete m_newValue;
         m_newValue = NULL;
       };
     }
   LocalVariableExpression& getNewValue() const
     { return m_newValue ? *m_newValue : *m_parent->m_newValue; }
   LocalVariableExpression* getSDominatorNewValue() const
     { return m_parent.get() ? m_parent->m_newValue : NULL; }
   void setNewValue(LocalVariableExpression* newValue, VirtualInstruction* lastDominator)
     { m_newValue = newValue; m_lastDominator = lastDominator; }
   bool setDominator(VirtualInstruction& dominator, GotoInstruction* previousLabel=NULL);
   friend class RenamingTask;
};

public:
 std::set<VariableRenaming> m_renamings; 7
 bool m_isLValue; 8 // attribut hérité
 LocalVariableExpression* m_localRename; 9 // attribut synthétisé
 LocalVariableExpression* m_localReplace; 10 // attribut synthétisé
 const Function& m_function; 11
```

```
  GotoInstruction* m_previousLabel; 12
  GotoInstruction* m_lastBranch; 13
  bool m_isOnlyPhi; 14

 private:
  static std::set<VariableRenaming>* cloneRenamings(const std::set<VariableRenaming>& source)
    {  std::set<VariableRenaming>* result = new std::set<VariableRenaming>(source);
       for (std::set<VariableRenaming>::const_iterator iter = result->begin(); iter != result->end(); ++iter)
          const_cast<VariableRenaming&>(*iter).cloneExpressions();
       return result;
    }
  static void freeRenamings(std::set<VariableRenaming>& source)
    {  for (std::set<VariableRenaming>::const_iterator iter = source.begin(); iter != source.end(); ++iter)
          const_cast<VariableRenaming&>(*iter).free();
       source.clear();
    }

 public:
  RenamingTask(const Function& function)
    :  m_isLValue(false), m_localRename(NULL), m_localReplace(NULL),
       m_function(function), m_previousLabel(NULL), m_lastBranch(NULL), m_isOnlyPhi(false)
    {  setInstruction(function.getFirstInstruction()); }
  RenamingTask(const RenamingTask& source)
    :  VirtualTask(source), m_renamings(source.m_renamings), m_isLValue(false),
       m_localRename(NULL), m_localReplace(NULL), m_function(source.m_function),
       m_previousLabel(source.m_previousLabel), m_lastBranch(source.m_lastBranch),
       m_isOnlyPhi(source.m_isOnlyPhi)
    {  for (std::set<VariableRenaming>::const_iterator iter = m_renamings.begin(); iter != m_renamings.end(); ++iter)
          const_cast<VariableRenaming&>(*iter).cloneExpressions();
    }

  virtual VirtualTask* clone() const { return new RenamingTask(*this); }
  virtual int getType() const { return TTRenaming; }
  virtual bool mergeWith(VirtualTask& source) { ((RenamingTask&) source).m_isOnlyPhi = true; return false; }
  friend class RenamingAgenda;
};
```

1 est la classe qui permet de renommer la variable `m_toReplace`. Les objets de cette classe sont destinés à être intégrés dans un ensemble afin d'autoriser le renommage de plusieurs variables pour une instruction. Comme pour la variable t dans l'exemple, il existe un renommage organisé en pile : $t \rightarrow B_4$, $B_1 \rightarrow t\_11$, $l_a \rightarrow t\_3$. Il signifie que après B_4, la variable t n'a pas été renommée, après B_1 t est à renommer à t_11 et après l_a, mais avant B_1, t est à renommer en t_3. Ainsi lorsqu'on rencontre la fin du bloc B_1, à savoir l_c, il est possible de reconstituer le renommage $t \rightarrow l_c$, $l_a \rightarrow t\_3$, indiquant que dorénavant t doit être renommé en t_3. Noter que ce renommage n'est valable que pour les arguments des fonctions Φ qui n'ont pas été pourvu par une branche.

2 est la variable à renommer.

3 est la nouvelle variable qui remplace `m_toReplace`. Si ce champ est encore `NULL`, alors la nouvelle variable est `m_parent->m_toReplace`. Si `m_parent` n'est pas défini et `m_newValue = NULL`, alors la variable ne doit pas être remplacée. Noter que ce dernier cas ne devrait pas se produire.

4 est le nom de la fonction et sert pour l'implantation de la méthode `compare`.

5 est l'instruction `IfInstruction` ou le label `LabelInstruction` qui domine le renommage courant. Ce champ sert pour savoir combien d'éléments de la pile de remplacement supprimer lorsqu'un label est rencontré. Il est en particulier utilisé par la méthode `setDominator`.

6 est l'élément précédent dans la pile de remplacement. Cet élément est maintenu à jour par la méthode `setDominator`. C'est un `shared_ptr` pour éviter de dupliquer la pile entière pour chaque variable lorsqu'une `IfInstruction` est rencontré et que la tâche de renommage est propagée sur la branche `then` et sur la branche `else`.

7 est l'ensemble des renommages, trié par variable selon l'ordre `VariableRenaming::compare`. Cet ordre est un ordre total et le même que

`PhiInsertionTask::IsBefore` permettant d'identifier rapidement chaque variable selon leur ordre dans la pile de déclaration `Scope`.

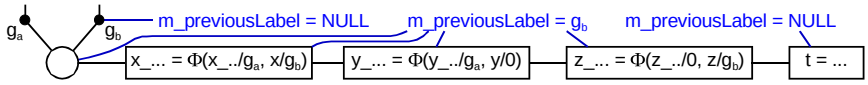
8 est un booléen activé uniquement lorsque la tâche de renommage descend dans les expressions à gauche des affectations. C'est un booléen qui descend jusqu'au niveau `LocalVariableExpression`, pour que les variables `m_localRename` et `m_localReplace` puissent remonter au niveau `ExpressionInstruction` afin de modifier le renommage à la suite de l'instruction.

9 est la variable supportant l'affectation après qu'une tâche de renommage soit descendue dans la partie gauche d'une affectation. Cette variable affectée remonte au niveau `ExpressionInstruction` qui décide comment mettre à jour le contenu de la tâche de renommage.

10 est l'ancienne variable qui était affectée avant que la tâche de renommage ne soit passée sur la partie gauche d'une affectation. Cette variable reste à NULL, lorsqu'aucun renommage n'a lieu sur la partie gauche de la définition. Cette variable affectée remonte au niveau `ExpressionInstruction` qui décide comment mettre à jour le contenu de la tâche de renommage.

11 est un champ initialisé à la construction qui, une fois fourni à chaque `VariableRenaming`, permet l'implantation de la méthode `VariableRenaming::compare`.

12 est le goto précédant le label. Il est mis en place lorsqu'une tâche de renommage passe sur un goto avant un label et il est effacé après le passage sur la dernière fonction Φ.



13 est le dernier `then` ou `else` rencontré. Cette branche sert à savoir combien de renommages dépiler lorsqu'un label est rencontré. Ce champ sert à définir à `VariableRenaming::m_lastDominator` pour tous les renommages de variable. Les variables à dépiler sont alors celles qui ont un rang de domination supérieur à celui de `VariableRenaming::m_lastDominator`.

14 est défini comme `m_previousLabel` (il est `true` lorsque `m_previousLabel` est défini). Lorsqu'il est mis à `true`, ce booléen signifie que la tâche ne doit pas se propager au-delà de la dernière fonction Φ.

5. Définissez la classe `RenamingAgenda` dans le fichier `Algorithm.h:317`. Cet agenda est construit avec une tâche initiale de renommage, qui se propage ensuite sur tout le graphe de contrôle par l'intermédiaire de la méthode `execute`. L'agenda marque les labels afin d'éviter que les instructions soient parcourues plusieurs fois. Ce marquage est à effacer lorsque la propagation des tâches de renommage est complètement terminée.

```
class RenamingTask : public VirtualTask {
  …
};

class RenamingAgenda : public WorkList {
 public:
  const Function* m_function;
  std::list<LabelInstruction*> m_markedInstruction;
  int m_ident;

 public:
  RenamingAgenda(const Function& function) : m_function(&function), m_ident(1)
    {  addNewAsFirst(new RenamingTask(function)); }
  ~RenamingAgenda()
    {  for (std::list<LabelInstruction*>::iterator iter = m_markedInstruction.begin();
            iter != m_markedInstruction.end(); ++iter)
          (*iter)->mark = NULL;
       m_markedInstruction.clear();
```

```
    }

  virtual void markInstructionWith(VirtualInstruction& instruction, VirtualTask& task)
    {  if ((instruction.type() == VirtualInstruction::TLabel) && !instruction.mark) {
         m_markedInstruction.push_back((LabelInstruction*) &instruction);
         instruction.mark = (void*) 1;
       };
    }
};
```

6. Implantez la méthode `RenamingTask::VariableRenaming::compare` dans le fichier `Algorithm.cpp:144`. L'implantation de cette méthode est similaire à celle de la méthode `PhiInsertionTask::IsBefore`. Elle se contente de trier le renommage de variable par rapport à l'emplacement de la variable dans la pile `Scope`.

```
void
LabelPhiFrontierAgenda::propagateOn(const LabelInstruction& liLabel,
    const std::set<VirtualExpression*, IsBefore>& sveModifiedSource) {
  …
}

Comparison
RenamingTask::VariableRenaming::compare(const VariableRenaming& source) const {
  assert(m_function == source.m_function);
  if (m_toReplace->type() == source.m_toReplace->type()) {
    if (m_toReplace->type() == VirtualExpression::TLocalVariable) {
      assert(dynamic_cast<const LocalVariableExpression*>(m_toReplace)
          && dynamic_cast<const LocalVariableExpression*>(source.m_toReplace));
      int thisIndex = ((const LocalVariableExpression&) *m_toReplace).getFunctionIndex(*m_function),
        sourceIndex = ((const LocalVariableExpression&) *source.m_toReplace).getFunctionIndex(*m_function);
      return (thisIndex < sourceIndex) ? CLess : ((thisIndex > sourceIndex) ? CGreater : CEqual);
    };
    if (m_toReplace->type() == VirtualExpression::TParameter) {
      assert(dynamic_cast<const ParameterExpression*>(m_toReplace)
          && dynamic_cast<const ParameterExpression*>(source.m_toReplace));
      int thisIndex = ((const ParameterExpression&) *m_toReplace).getIndex(),
        sourceIndex = ((const ParameterExpression&) *source.m_toReplace).getIndex();
      return (thisIndex < sourceIndex) ? CLess : ((thisIndex > sourceIndex) ? CGreater : CEqual);
    };
    assert(dynamic_cast<const GlobalVariableExpression*>(m_toReplace)
        && dynamic_cast<const GlobalVariableExpression*>(source.m_toReplace));
    int thisIndex = ((const GlobalVariableExpression&) *m_toReplace).getIndex(),
      sourceIndex = ((const GlobalVariableExpression&) *source.m_toReplace).getIndex();
    return (thisIndex < sourceIndex) ? CLess : ((thisIndex > sourceIndex) ? CGreater : CEqual);
  };
  return (m_toReplace->type() < source.m_toReplace->type()) ? CGreater : CLess;
}
```

7. Implantez la méthode `RenamingTask::VariableRenaming::setDominator` dans le fichier `Algorithm.cpp:171`. Cette méthode met à jour le champ `VariableRenaming::m_lastDominator` avec `dominator` correspondant au dominateur du label. Cette méthode peut augmenter (d'un renommage) ou diminuer la pile de renommage. Ainsi pour une pile de renommage de $t \rightarrow B_4$, $B_1 \rightarrow t\_11$, $l_a \rightarrow t\_3$, l'appel à la méthode `setDominator(l_c)` lorsque la $t \rightarrow l_c$, $l_a \rightarrow t\_3$ dépile $B_1 \rightarrow t\_11$ et remplace pas B_4 par l_c. Finalement cette méthode retourne le fait que le renommage de notre variable ait encore un effet sur la suite. Noter que pour éviter de supprimer le renommage lié à l'insertion des fonctions Φ, nous testons si (`m_lastDominator != previousLabel`), ce qui indique une fonction Φ nouvellement insérée et nous permet de conserver le renommage.

```
Comparison
RenamingTask::VariableRenaming::compare(const VariableRenaming& source) const {
  …
}

bool
RenamingTask::VariableRenaming::setDominator(VirtualInstruction& dominator,GotoInstruction* previousLabel) {
  assert(m_lastDominator || !m_parent.get());
  if (m_lastDominator && m_lastDominator->getDominationHeight() >= dominator.getDominationHeight()) {
    if (m_newValue && m_lastDominator != previousLabel) {
      delete m_newValue;
      m_newValue = NULL;
```

```
    };
    m_lastDominator = &dominator;
    while (m_parent.get() && m_parent->m_lastDominator
        && m_parent->m_lastDominator->getDominationHeight() >= dominator.getDominationHeight())
      m_parent = m_parent->m_parent;
  }
  else if (m_newValue) {
    std::shared_ptr<VariableRenaming> newParent(new VariableRenaming(*this, Transfert()));
    newParent->m_parent = m_parent;
    m_parent = newParent;
  };
  m_lastDominator = &dominator;
  return m_newValue || m_parent.get();
}
```

Implantation de l'algorithmique

8. Rajoutez la déclaration de la méthode `virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse)` pour toutes les expressions, notamment celles qui peuvent avoir des accès en écriture (comme pour l'insertion des fonctions Φ), mais également des accès en lecture à des variables locales. Ces expressions sont directement touchées par la phase de renommage. Des déclarations ont lieu dans le fichier `SyntaxTree.h:467, 497, 517, 550, 574, 599, 630, 687`.

```
class ComparisonExpression : public VirtualExpression {
 public:
  …
  ComparisonExpression& setSnd(VirtualExpression* snd) { … }

  virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
  virtual void print(std::ostream& out) const { … }
  …
};

class UnaryOperatorExpression : public VirtualExpression {
 public:
  …
  UnaryOperatorExpression& setSubExpression(VirtualExpression* subExpression) { … }

  virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
  virtual void print(std::ostream& out) const { … }
  …
};

class BinaryOperatorExpression : public VirtualExpression {
 public:
  …
  BinaryOperatorExpression& setSnd(VirtualExpression* snd) { … }

  virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
  virtual void print(std::ostream& out) const { … }
  …
};

class DereferenceExpression : public VirtualExpression {
 private:
  …
 public:
  …
  DereferenceExpression& setSubExpression(VirtualExpression* subExpression) { … }

  virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
  virtual void print(std::ostream& out) const { … }
  …
};

class ReferenceExpression : public VirtualExpression {
 private:
  …
 public:
  …
  ReferenceExpression& setSubExpression(VirtualExpression* subExpression) { … }
```

```
    virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
    virtual void print(std::ostream& out) const { … }
    …
};

class CastExpression : public VirtualExpression {
  private:
    …
  public:
    …
    CastExpression& setType(VirtualType* type) { … }

    virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
    virtual void print(std::ostream& out) const { … }
    …
};

class FunctionCallExpression : public VirtualExpression {
  private:
    …
  public:
    …
    virtual ~FunctionCallExpression() { … }

    virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
    FunctionCallExpression& addArgument(VirtualExpression* argument) { … }
    …
};

class AssignExpression : public VirtualExpression {
    …
} ;

class GotoInstruction;
class PhiExpression : public VirtualExpression {
  private:
    …
  public:
    …
    PhiExpression& addReference(VirtualExpression* expression, GotoInstruction& gotoInstruction) { … }

    virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
    virtual void print(std::ostream& out) const;
    …
};
```

9. Déclarez la méthode `IfInstruction::handle` et la méthode `ReturnInstruction::handle` dans le fichier `SyntaxTree.h:809, 947`.

```
class GotoInstruction;
class IfInstruction : public VirtualInstruction {
  private:
    …
  public:
    IfInstruction() { setType(TIf); }

    virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
    virtual int countNexts() const { … }
    …
};
…
class ReturnInstruction : public VirtualInstruction {
  private:
    …
  public:
    ReturnInstruction() { … }

    virtual void handle(VirtualTask& task, WorkList& continuations, Reusability& reuse);
    ReturnInstruction& setResult(VirtualExpression* expression) { … }
    …
};
```

10. Implantez les méthodes `BinaryOperatorExpression::handle`, `DereferenceExpression::handle`, `ReferenceExpression::handle`, `CastExpression::handle`, `FunctionCallExpression::handle`, `ReturnInstruction::handle` dans le fichier `SyntaxTree.cpp:122, 363`. Ces

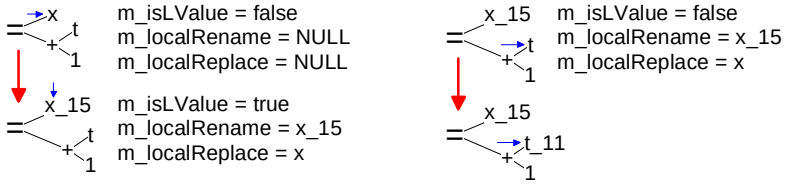
méthodes propagent simplement la méthode `handle` sur les expressions qu'elles contiennent pour remplacer les variables à remplacer.

```
void
LocalVariableExpression::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  …
}
…

void
ComparisonExpression::handle(VirtualTask& task, WorkList& continuations, Reusability& reuse) {
  int type = task.getType();
  if (type == TTRenaming) {
    if (m_fst.get())
      m_fst->handle(task, continuations, reuse);
    if (m_snd.get())
      m_snd->handle(task, continuations, reuse);
  };
}

void
UnaryOperatorExpression::handle(VirtualTask& task, WorkList& continuations, Reusability& reuse) {
  int type = task.getType();
  if (type == TTRenaming) {
    if (m_subExpression.get())
      m_subExpression->handle(task, continuations, reuse);
  };
}

void
BinaryOperatorExpression::handle(VirtualTask& task, WorkList& continuations, Reusability& reuse) {
  int type = task.getType();
  if (type == TTRenaming) {
    if (m_fst.get())
      m_fst->handle(task, continuations, reuse);
    if (m_snd.get())
      m_snd->handle(task, continuations, reuse);
  };
}

void
DereferenceExpression::handle(VirtualTask& task, WorkList& continuations, Reusability& reuse) {
  int type = task.getType();
  if (type == TTRenaming) {
    if (m_subExpression.get())
      m_subExpression->handle(task, continuations, reuse);
  };
}

void
ReferenceExpression::handle(VirtualTask& task, WorkList& continuations, Reusability& reuse) {
  int type = task.getType();
  if (type == TTRenaming) {
    if (m_subExpression.get())
      m_subExpression->handle(task, continuations, reuse);
  };
}

void
CastExpression::handle(VirtualTask& task, WorkList& continuations, Reusability& reuse) {
  int type = task.getType();
  if (type == TTRenaming) {
    if (m_subExpression.get())
      m_subExpression->handle(task, continuations, reuse);
  };
}

void
FunctionCallExpression::handle(VirtualTask& task, WorkList& continuations, Reusability& reuse) {
  int type = task.getType();
  if (type == TTRenaming) {
    for (std::vector<VirtualExpression*>::const_iterator iter = m_arguments.begin();
         iter != m_arguments.end(); ++iter) {
      if (*iter)
        (*iter)->handle(task, continuations, reuse);
    };
  };
}
```

```
...

void
LabelInstruction::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  ...
}

void
ReturnInstruction::handle(VirtualTask& task, WorkList& continuations, Reusability& reuse) {
  int type = task.getType();
  if ((type == TTRenaming) && m_result.get())
    m_result->handle(task, continuations, reuse);
  VirtualInstruction::handle(task, continuations, reuse);
}
```

11. Implantez la méthode `LocalVariableExpression::handle` pour les tâches de renommage dans le fichier `SyntaxTree.cpp:120`. Cette méthode traite les différents cas :
    - remplacement de variable à gauche de l'affectation d'une fonction Φ.
    - remplacement de variable à gauche d'une affectation normale. Il est alors nécessaire de créer une nouvelle variable lorsque la déclaration de l'ancienne variable a déjà une définition.
    - remplacement de variable dans une expression. Il faut alors rechercher si un renommage existe pour la variable considérée et si le renommage existe, effectuer le renommage de la variable.

    Pour l'implantation on peut se référer à la description des champs de la classe `RenamingTask` et `VariableRenaming` en question 4.




```
void
LocalVariableExpression::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  int type = vtTask.getType();
  if (type == TTPhiInsertion) {
    ...
  }
  else if (type == TTRenaming) {
    assert(dynamic_cast<const RenamingTask*>(&vtTask) && dynamic_cast<const RenamingAgenda*>(&continuations));
    RenamingTask& task = (RenamingTask&) vtTask;
    if (task.m_isLValue) {
      if (!task.m_isOnlyPhi && m_scope.hasSSADefinition(m_localIndex)) {
        RenamingAgenda& remaingContinuations = (RenamingAgenda&) continuations;
        task.m_localReplace = new LocalVariableExpression(*this);
        m_localIndex = m_scope.addSSADeclaration(m_name, m_localIndex, remaingContinuations.m_ident);
      };
      task.m_localRename = ...;
    }
    else { // !rtTask.m_isLValue
      RenamingTask::VariableRenaming locate(*this, task.m_function);
      std::set<RenamingTask::VariableRenaming>::iterator found = task.m_renamings.find(locate);
      if (found != task.m_renamings.end()) {
        m_localIndex = found->getNewValue()....;
        m_name = found->getNewValue()....;
      };
    };
  };
}
```

12. Implantez la méthode `PhiExpression::handle` dans le fichier `SyntaxTree.cpp:261`. Cette méthode met à jour la partie gauche d'une fonction Φ en traitant tous les cas (recherche du renommage au niveau du goto d'où l'on vient ou recherche du renommage au niveau du dominateur du goto d'où l'on vient). Noter qu'en pratique $x = \Phi(x/g_a, x/0)$, rechercher le renommage du second x au niveau du goto g_b ou au niveau de son dominateur revient au même puisque le fait d'arriver par

g_b signifie que la variable x n'a pas été modifié depuis le dominateur du label (propriété de l'insertion des fonctions Φ sur la frontière de domination).

```
…
void
PhiExpression::print(std::ostream& osOut) const {
  …
}

void
PhiExpression::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  int type = vtTask.getType();
  if (type == TTRenaming) {
    assert(dynamic_cast<const RenamingTask*>(&vtTask));
    RenamingTask& task = (RenamingTask&) vtTask;
    assert(m_fst.get());
    if (!m_snd.get())
      m_snd.reset(m_fst->clone());
    if (task.m_previousLabel == m_fstFrom)
      m_fst->handle(task, continuations, reuse);
    else if (!m_sndFrom) {
      m_sndFrom = task.m_previousLabel;
      std::set<RenamingTask::VariableRenaming>::const_iterator found = task.m_renamings
        .find(RenamingTask::VariableRenaming(*m_fst, task.m_function));
      LocalVariableExpression* newValue = NULL;
      if (found != task.m_renamings.end())
        newValue = found->getSDominatorNewValue();
      if (newValue)
        m_snd.reset(newValue->clone());
    }
    else {
      assert(task.m_previousLabel == m_sndFrom);
      m_snd->handle(vtTask, continuations, reuse);
    };
  };
}
```

13. Implantez la méthode `AssignExpression::handle` dans le fichier `SyntaxTree.cpp:222`. Cette méthode se contente de gérer le champ `RenamingTask::m_isLValue`.

```
void
FunctionCallExpression::handle(VirtualTask& task, WorkList& continuations, Reusability& reuse) {
  …
};

void
AssignExpression::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  int type = vtTask.getType();
  if (type == TTPhiInsertion && m_lvalue.get()) {
    assert(dynamic_cast<const PhiInsertionTask*>(&vtTask));
    PhiInsertionTask& task = (PhiInsertionTask&) vtTask;
    task.m_isLValue = …;
    m_lvalue->handle(vtTask, continuations, reuse);
    task.m_isLValue = …;
  }
  else if (type == TTRenaming) {
    assert(dynamic_cast<const RenamingTask*>(&vtTask));
    RenamingTask& task = (RenamingTask&) vtTask;
    task.m_isLValue = …;
    m_lvalue->handle(task, continuations, reuse);
    task.m_isLValue = …;
    m_rvalue->handle(task, continuations, reuse);
  };
}
```

14. Implantez la méthode `ExpressionInstruction::handle` dans le fichier `SyntaxTree.cpp:311`. Cette méthode gère les champs `RenamingTask::m_localReplace` et `RenamingTask::m_localRename` alors qu'ils remontent des sous-expressions de l'instruction. Lorsque `RenamingTask::m_localRename` est défini, cela signifie qu'une affectation a eu lieu. Si, de plus `RenamingTask::m_localReplace` est défini, alors un nouveau nom de

variable a été généré comme résultat de l'affectation. Ce nouveau nom doit alors mettre à jour l'ensemble des renommages dans notre méthode.
La méthode `ExpressionInstruction::handle` gère également la propagation de la tâche de renommage sur les fonctions Φ juste après les labels. On sait qu'on est dans cette phase de propagation lorsque `task.m_previousLabel` est défini. À la fin de cette propagation sur les fonctions Φ et si `task.m_isOnlyPhi` est activé, alors la propagation s'arrête. Noter que `task.m_isOnlyPhi` est activé si et seulement si le label a été marqué et qu'une tâche de propagation a déjà eu lieu. Finalement, le traitement qui aurait du être géré au niveau du label doit également être géré par notre méthode sur la dernière fonction Φ. Nous avons du retarder cette gestion car les fonctions Φ sont traitées avec le renommage du goto de leur provenance. Le traitement au niveau du label est simplement la mise à jour du dominateur de tous les renommages `VariableRenaming` en appelant la méthode `VariableRenaming::setDominator`.

```
void
ExpressionInstruction::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  int type = vtTask.getType();
  if (type == TTPhiInsertion) {
    …
  }
  else if (type == TTRenaming) {
    if (m_expression.get())
      m_expression->handle(vtTask, continuations, reuse);
    assert(dynamic_cast<RenamingTask*>(&vtTask));
    RenamingTask& task = (RenamingTask&) vtTask;
    if (task.m_localRename) {
      if (task.m_localReplace) {
        std::set<RenamingTask::VariableRenaming>::iterator found = task.m_renamings
          .find(RenamingTask::VariableRenaming(*task.m_localReplace, task.m_function));
        if (found != task.m_renamings.end()) {
          const_cast<RenamingTask::VariableRenaming&>(*found)
            .setNewValue(new LocalVariableExpression(*task.…),
              task.m_previousLabel ? task.m_previousLabel : (task.m_lastBranch
                ? task.m_lastBranch->getSPreviousInstruction() : NULL));
          delete task.…;
          task.… = NULL;
        }
        else {
          assert(!task.m_lastBranch ||
              dynamic_cast<IfInstruction*>(task.m_lastBranch->getSPreviousInstruction()));
          task.m_renamings.insert(RenamingTask::VariableRenaming(
            task.…, new LocalVariableExpression(*task.…),
            task.m_function, task.m_previousLabel
              ? task.m_previousLabel : (task.m_lastBranch
              ? task.m_lastBranch->getSPreviousInstruction()
              : NULL)));
          task.m_localReplace = NULL;
        };
      };
      task.m_localRename->scope().setSSADefinition(task.m_localRename->getLocalScope(), …);
      task.… = NULL;
    };
    if (task.m_previousLabel) {
      bool isLast = true;
      if (getSNextInstruction() && getSNextInstruction()->type() == TExpression) {
        assert(dynamic_cast<const ExpressionInstruction*>(getSNextInstruction()));
        if (((const ExpressionInstruction&) *getSNextInstruction()).isPhi()) {
          isLast = false;
          if (task.m_isOnlyPhi)
            VirtualInstruction::handle(task, continuations, reuse);
        };
      };
      if (isLast) {
        assert(dynamic_cast<const LabelInstruction*>(task.m_previousLabel->getSNextInstruction()));
        VirtualInstruction* dominator = ((LabelInstruction*)
          task.m_previousLabel->getSNextInstruction())->getSDominator();
        assert(dominator);
        std::set<RenamingTask::VariableRenaming>::iterator iter = task.m_renamings.begin();
        while (iter != task.m_renamings.end()) {
          if (!const_cast<RenamingTask::VariableRenaming&>(*iter)
```

```
                .setDominator(*dominator, task.m_previousLabel)) {
              std::set<RenamingTask::VariableRenaming>::iterator nextIter(iter);
              ++nextIter;
              const RenamingTask::VariableRenaming* next = (nextIter != task.m_renamings.end())
                ? &(*nextIter) : NULL;
              task.m_renamings.erase(iter);
              if (next)
                iter = task.m_renamings.find(*next);
              else
                iter = task.m_renamings.end();
            }
            else
              ++iter;
          };
          task.m_previousLabel = NULL;
        };
        if (task.m_isOnlyPhi)
          return;
      };
    };
    VirtualInstruction::handle(vtTask, continuations, reuse);
}
```

15. Implantez la méthode `IfInstruction::handle` dans le fichier `SyntaxTree.cpp:384`. Cette méthode met à jour sa condition. Elle empile également un nouveau dominateur par l'intermédiaire de la méthode `RenamingTask::VariableRenaming::setDominator`, et ce pour chaque renommage avant que la tâche `task` ne soit dupliquée sur les deux branches du **`if`**.

```
void
ExpressionInstruction::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  …
}

void
IfInstruction::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  int type = vtTask.getType();
  if (type == TTRenaming) {
    if (m_expression.get())
      m_expression->handle(vtTask, continuations, reuse);
    assert(dynamic_cast<const RenamingTask*>(&vtTask));
    RenamingTask& task = (RenamingTask&) vtTask;
    std::set<RenamingTask::VariableRenaming>::iterator iter = task.m_renamings.begin();
    while (iter != task.m_renamings.end()) {
      if (!const_cast<RenamingTask::VariableRenaming&>(*iter).setDominator(*this)) {
        std::set<RenamingTask::VariableRenaming>::iterator nextIter(iter);
        ++nextIter;
        const RenamingTask::VariableRenaming* next = (nextIter != task.m_renamings.end())
          ? &(*nextIter) : NULL;
        task.m_renamings.erase(iter);
        if (next)
          iter = task.m_renamings.find(*next);
        else
          iter = task.m_renamings.end();
      }
      else
        ++iter;
    };
  };
  VirtualInstruction::handle(vtTask, continuations, reuse);
}
```

16. Implantez la méthode `GotoInstruction::handle` dans le fichier `SyntaxTree.cpp:431`. Cette méthode effectue un traitement spécifique pour les branches du if et pour les gotos avant les labels. Pour les branches du if, la tâche de renommage est mise à jour, notamment à travers `RenamingTask::m_lastBranch`, indiquant la dernière branche rencontrée. Pour les gotos avant les labels, notre méthode gère le champ `RenamingTask::m_isOnlyPhi` en le plaçant à **`true`** (pour ne propager la tâche de renommage que sur les fonctions Φ à la suite du label) si une propagation de la tâche de renommage a déjà eu lieu par un autre label.

```
void
GotoInstruction::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
```

```
  int type = vtTask.getType();
  if ((type == TTPhiInsertion) && ((m_context == CAfterIfThen) || (m_context == CAfterIfElse))) {
    …
  }
  else if (type == TTRenaming) {
    assert(dynamic_cast<const RenamingTask*>(&vtTask));
    RenamingTask& task = (RenamingTask&) vtTask;
    if ((m_context == CAfterIfThen) || (m_context == CAfterIfElse))
      task.m_lastBranch = …;
    else if (m_context >= CLoop) {
      task.m_previousLabel = …;
      if (getSNextInstruction()->mark) {
        task.m_isOnlyPhi = true;
        if (getSNextInstruction()) {
          task.clearInstruction();
          task.setInstruction(*getSNextInstruction());
          reuse.setReuse();
        };
        return;
      };
    };
  };
  VirtualInstruction::handle(vtTask, continuations, reuse);
}
```

17. Implantez la méthode `LabelInstruction::handle` dans le fichier `SyntaxTree.cpp:532`. Cette méthode gère la propagation de la tâche de renommage. Il est à noter que s'il existe des fonctions Φ après les labels, cette propagation est effectuée par les `ExpressionInstruction` à la suite du label qui correspondent à des fonctions Φ. À la fin de cette propagation sur les fonctions Φ et si `task.m_isOnlyPhi` est activé, alors la propagation s'arrête. Noter que `task.m_isOnlyPhi` est activé si et seulement si le label a été marqué et qu'une tâche de propagation a déjà eu lieu. Si aucune fonction Φ n'est présente, le traitement au niveau du label est simplement la mise à jour du dominateur de tous les renommages `VariableRenaming` en appelant la méthode `VariableRenaming::setDominator`.

```
void
LabelInstruction::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  int type = vtTask.getType();
  if (type == TTDomination) {
    …
  }
  else if (type == TTPhiInsertion) {
    …
  }
  else if (type == TTLabelPhiFrontier) {
    …
  }
  else if (type == TTRenaming) {
    assert(dynamic_cast<const RenamingTask*>(&vtTask));
    RenamingTask& task = (RenamingTask&) vtTask;
    if (task.m_previousLabel) {
      bool isLast = true;
      if (getSNextInstruction() && getSNextInstruction()->type() == TExpression) {
        assert(dynamic_cast<const ExpressionInstruction*>(getSNextInstruction()));
        if (((const ExpressionInstruction&) *getSNextInstruction()).isPhi()) {
          isLast = false;
          if (task.m_isOnlyPhi)
            VirtualInstruction::handle(vtTask, continuations, reuse);
        };
      }
      if (isLast) {
        assert(m_dominator);
        std::set<RenamingTask::VariableRenaming>::iterator iter = task.m_renamings.begin();
        while (iter != task.m_renamings.end()) {
          if (!const_cast<RenamingTask::VariableRenaming&>(*iter).setDominator(*m_dominator)) {
            std::set<RenamingTask::VariableRenaming>::iterator nextIter(iter);
            ++nextIter;
            const RenamingTask::VariableRenaming* next = (nextIter != task.m_renamings.end())
              ? &(*nextIter) : NULL;
            task.m_renamings.erase(iter);
            if (next)
              iter = task.m_renamings.find(*next);
            else
```

```
                    iter = task.m_renamings.end();
                }
                else
                  ++iter;
              };
              task.m_previousLabel = NULL;
            };
            if (task.m_isOnlyPhi)
              return;
          };
        };
        VirtualInstruction::handle(vtTask, continuations, reuse);
      }
```

18. Implantez la méthode `EnterBlockInstruction::handle` dans le fichier `SyntaxTree.cpp:593`. Cette méthode se contente d'allouer la place nécessaire pour que les définitions des variables puissent être associées à leurs déclarations.

```
void
EnterBlockInstruction::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  VirtualInstruction::handle(vtTask, continuations, reuse);
  int type = vtTask.getType();
  if (type == TTPrint) {
    …
  }
  else if (type == TTPhiInsertion) {
    …
  }
  else if (type == TTRenaming)
    m_scope.setSizeDefinitions();
}
```

Assemblage

19. Déclarez (ficher `SyntaxTree.h:1162`) et implantez (ficher `SyntaxTree.cpp:745`) la méthode `Program::renameSSA`. Cette méthode se contente d'appliquer l'algorithme de renommage sur chaque fonction du programme.

```
class Program {
 private:
  …
 public:
  …
  void insertPhiFunctions();
  void renameSSA();
  class ParseContext {
    …
  };
  friend class ParseContext;
};

// Fichier SyntaxTree.cpp
void
Program::insertPhiFunctions() {
  …
}

void
Program::renameSSA() {
  for (std::set<Function>::const_iterator functionIter = m_functions.begin();
       functionIter != m_functions.end(); ++functionIter) {
    RenamingAgenda renamingAgenda(*functionIter);
    renamingAgenda.execute();
  };
}
```

20. Appelez la méthode `Program::renameSSA` à partir de la fonction `main` (fichier `SyntaxTree.cpp:779`).

```
int main(int argc, char** argv ) {
  …
  program.printWithWorkList(std::cout);
  std::cout << std::endl;
  program.renameSSA();
  program.printWithWorkList(std::cout);
```

```
  std::cout << std::endl;
  return 0;
}
```

Vérifiez que le résultat sur `essai.c` est la sortie suivante :

```
function f
0   {
1     return ([parameter 0: x] + 2);

function main
0   {0x614ef0 x := 0    0x614ef0 x_1 := 2       0x614ef0 x_2 := 3       0x614ef0 x_3 := 4       0x614ef0 x_4 := 5   0x614ef0 x_5 := 6
          0x614ef0 x_6 := 7       0x614ff0 y := 1
1     [local 0: x] = [parameter 0: argc];
2     [local 1: y] = 2;
3     [local 2: x_1] = ([local 0: x] * [local 0: x]);
4     goto label 0x6153a0
5     label 6153a0      dominated by 4 goto label 0x6153a0
       domination frontier of label = 5
28    [local 3: x_2] = phi([local 4: x_3] 11, [local 2: x_1] 4);
6     {
7       [local 4: x_3] = ([local 3: x_2] + 2);
8     }
9     if (([local 4: x_3] < 100))
10    then      domination frontier = 5
11    goto loop 6153a0
12    else
13    if (([local 4: x_3] > [local 1: y]))
14    then      domination frontier = 24
15    {0x615760 k := 0
16      [local 0: k] = 8;
17      [local 5: x_4] = ((4 + ([local 4: x_3] * 2)) - 1);
18    }
25    goto label 0x615cc0
19    else      domination frontier = 24
20    {
21      [local 6: x_5] = (([local 4: x_3] + 1) - f([local 1: y]));
22    }
23    goto label 0x615cc0
24    label 615cc0      dominated by 13 if (([local 4: x_3] > [local 1: y]))
29    [local 7: x_6] = phi([local 5: x_4] 25, [local 6: x_5] 23);
26    return ([local 7: x_6] + 3);
```

Vérifiez que le résultat sur `essai2.c` est la sortie suivante :

```
function main
0   {0x614ce8 x := 0    0x614ce8 x_1 := 2       0x614ce8 x_10 := 11     0x614ce8 x_12 := 13     0x614ce8 x_2 := 3   0x614ce8
          x_4 := 5       0x614ce8 x_6 := 7       0x614ce8 x_7 := 8       0x614ce8 x_9 := 10  0x614df8 y := 1 0x614df8 y_11 := 12
          0x614df8 y_3 := 4       0x614df8 y_5 := 6       0x614df8 y_8 := 9
1     [local 0: x] = [parameter 0: argc];
2     [local 2: x_1] = (([local 0: x] * [local 0: x]) - (2 * [local 0: x]));
3     [local 1: y] = (4 + [parameter 0: argc]);
4     goto label 0x615278
5     label 615278      dominated by 4 goto label 0x615278
       domination frontier of label = 5
39    [local 3: x_2] = phi([local 13: x_12] 35, [local 0: x] 4);
40    [local 4: y_3] = phi([local 12: y_11] 35, [local 1: y] 4);
6     {
7       if (([local 3: x_2] > 2))
8       then    domination frontier = 29
9       {
10        [local 5: x_4] = ([local 3: x_2] - 1);
11      }
30      goto label 0x615c30
12      else    domination frontier = 29
13      {
14        if (((2 * [local 3: x_2]) > [local 4: y_3]))
15        then  domination frontier = 24
16        {
17          [local 6: y_5] = 2;
18        }
25        goto label 0x615ad8
19        else  domination frontier = 24
20        {
21          [local 7: x_6] = ([local 3: x_2] - 1);
22        }
23        goto label 0x615ad8
24        label 615ad8  dominated by 14 if (((2 * [local 3: x_2]) > [local 4: y_3]))
       domination frontier of label = 29
```

```
43      [local 8: x_7] = phi([local 7: x_6] 23, [local 3: x_2] 25);
44      [local 9: y_8] = phi([local 6: y_5] 25, [local 4: y_3] 23);
26      [local 10: x_9] = ([local 8: x_7] - 1);
27    }
28    goto label 0x615c30
29    label 615c30    dominated by 7 if (([local 3: x_2] > 2))
      domination frontier of label = 5
41    [local 11: x_10] = phi([local 5: x_4] 30, [local 10: x_9] 28);
42    [local 12: y_11] = phi([local 9: y_8] 28, [local 4: y_3] 30);
31    [local 13: x_12] = ([local 11: x_10] - 1);
32  }
33  if (([local 13: x_12] > 0))
34  then      domination frontier = 5
35  goto loop 615278
36  else
37  return 0;
```

Vérifiez que le résultat sur `essai3.c` est la sortie suivante :

```
function main
0  {0x614ce8 x := 0    0x614ce8 x_1 := 2      0x614ce8 x_11 := 12     0x614ce8 x_2 := 3      0x614ce8 x_4 := 5   0x614ce8 x_5
        := 6        0x614ce8 x_9 := 10       0x614df8 y := 1 0x614df8 y_10 := 11 0x614df8 y_3 := 4          0x614df8 y_6 := 7
        0x614df8 y_7 := 8      0x614df8 y_8 := 9
1    [local 0: x] = [parameter 0: argc];
2    [local 2: x_1] = (([local 0: x] * [local 0: x]) - (2 * [local 0: x]));
3    [local 1: y] = (4 + [parameter 0: argc]);
4    goto label 0x615278
5    label 615278      dominated by 4 goto label 0x615278
      domination frontier of label = 5
39   [local 3: x_2] = phi([local 12: x_11] 35, [local 2: x_1] 4);
40   [local 4: y_3] = phi([local 11: y_10] 35, [local 1: y] 4);
6    {
7     if (([local 3: x_2] > 2))
8     then    domination frontier = 29
9     {
10      [local 5: x_4] = ([local 3: x_2] - 1);
11    }
30    goto label 0x615c18
12    else    domination frontier = 29
13    {
14      [local 6: x_5] = ([local 3: x_2] - 1);
15      if (((2 * [local 6: x_5]) > [local 4: y_3]))
16      then  domination frontier = 25
17      {
18        [local 7: y_6] = 2;
19      }
26      goto label 0x615b58
20      else  domination frontier = 25
21      {
22        [local 8: y_7] = 3;
23      }
24      goto label 0x615b58
25      label 615b58  dominated by 15 if (((2 * [local 6: x_5]) > [local 4: y_3]))
      domination frontier of label = 29
43      [local 9: y_8] = phi([local 7: y_6] 26, [local 8: y_7] 24);
27    }
28    goto label 0x615c18
29    label 615c18    dominated by 7 if (([local 3: x_2] > 2))
      domination frontier of label = 5
41    [local 10: x_9] = phi([local 5: x_4] 30, [local 6: x_5] 28);
42    [local 11: y_10] = phi([local 9: y_8] 28, [local 4: y_3] 30);
31    [local 12: x_11] = ([local 10: x_9] - 1);
32  }
33  if (([local 12: x_11] > 0))
34  then      domination frontier = 5
35  goto loop 615278
36  else
37  return 0;
```