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 :

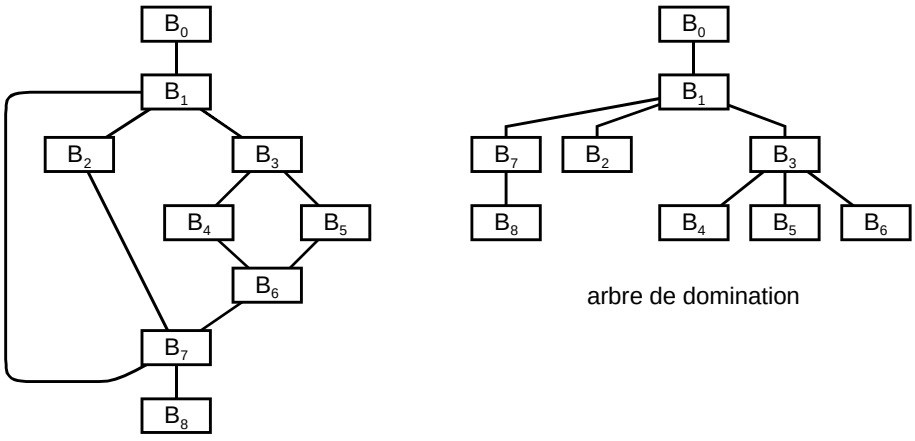
```
int f(int x) {
   return x+2;
}

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

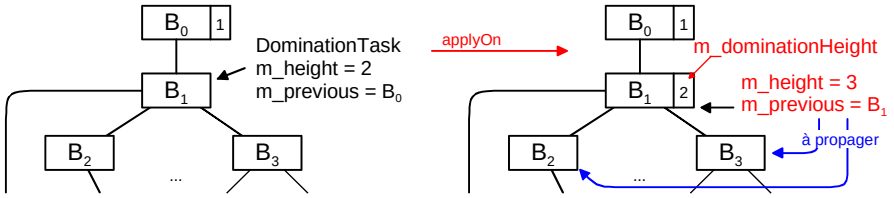
3. En vous inspirant du code de `PrintTask` et de `PrintAgenda` dans `Algorithms.h:54`, écrivez environ 40 lignes de code correspondant à la classe `DominationTask` (tâche se propageant en avant et calculant les dominateurs immédiats ainsi que la hauteur des instructions dans l'arbre de domination) et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

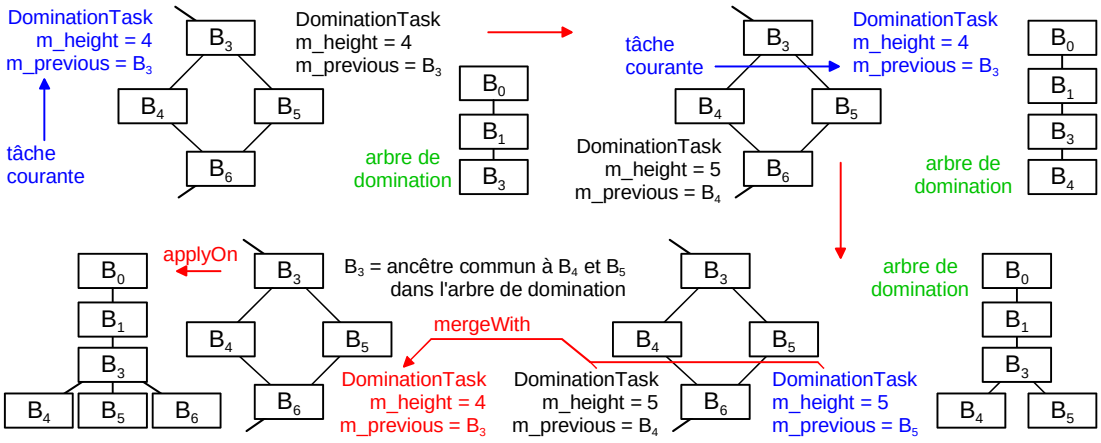
```
  void setHeight(int newHeight) { assert(newHeight < m_height); m_height = newHeight; }
  VirtualInstruction* findDominatorWith(VirtualInstruction& instruction) const;

  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`.

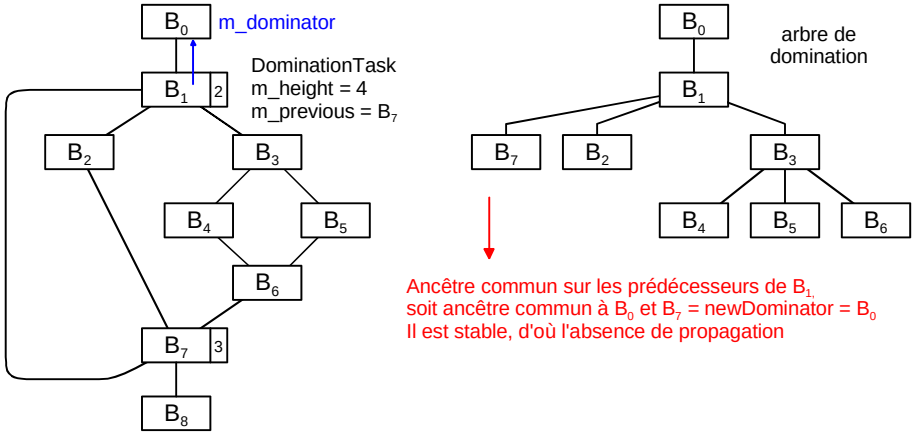
```
void
LabelInstruction::handle(VirtualTask& vtTask, WorkList& continuations, Reusability& reuse) {
  int type = vtTask.getType();
  if (type == TTDomination) { 1
    assert(dynamic_cast<const DominationTask*>(&vtTask));
    DominationTask& task = (DominationTask&) vtTask;
    if (!m_dominator) 2
      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.

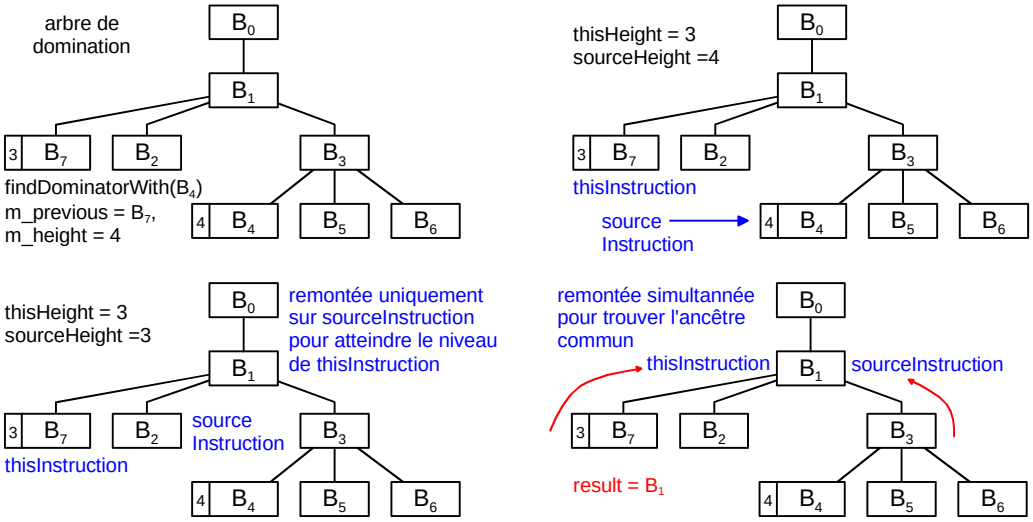
5 indique si nous avons stabilité dans le calcul du dominateur. Si oui, le plus grand point fixe est atteint et nous ne propageons pas la tâche sur le successeur du label en court-circuitant `VirtualInstruction::handle`.

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);
```

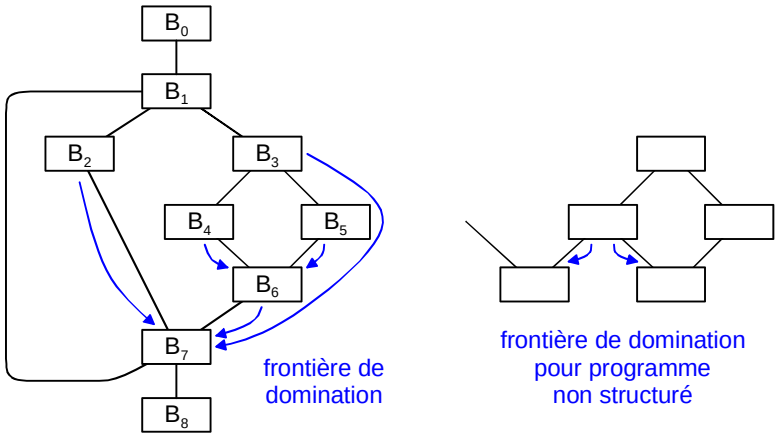
Le résultat sur `essai2.c` est la sortie suivante :

```
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

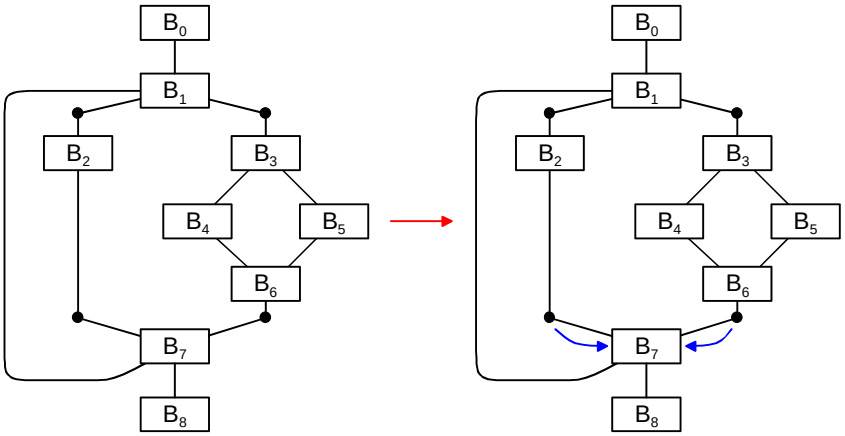
Rajoutez le champ `std::vector<GotoInstruction*> m_dominationFrontier` au niveau de la classe `GotoInstruction` et de la classe `LabelInstruction` dans le fichier `SyntaxTree.h:739` et `SyntaxTree.h:809`. Les labels qui sont dans la frontière de domination sont alors les suivants des `goto`s enregistrés 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; }
```

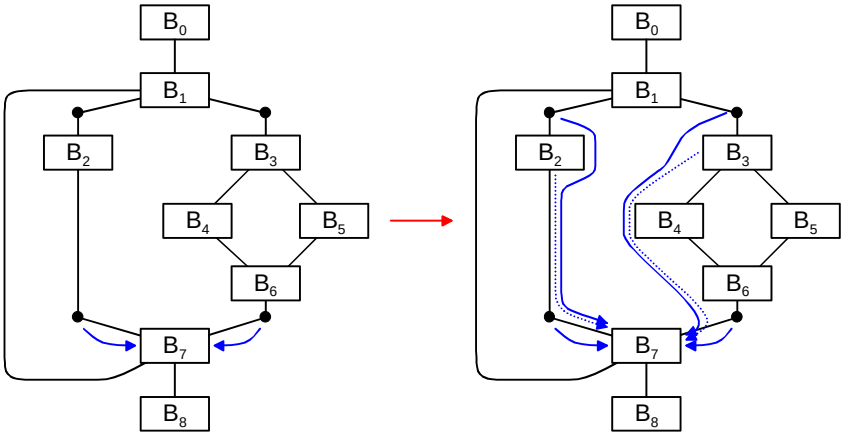
```
  const DominationFrontier& getDominationFrontier() const { return m_dominationFrontier;}
  virtual void print(std::ostream& out) const
    {  if (m_context == CLoop)
         …
       else if (…)
         …
       else
         out << "goto " << std::hex << (int) getSNextInstruction() << std::dec;
       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)
     { … }
   …
};
```

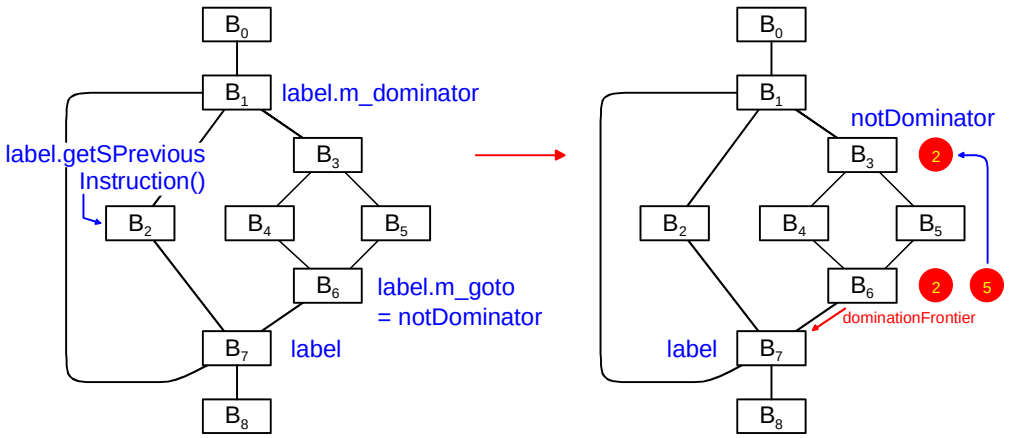
La méthode `Function::setDominationFrontier` est implantée dans `SyntaxTree.cpp:141`.

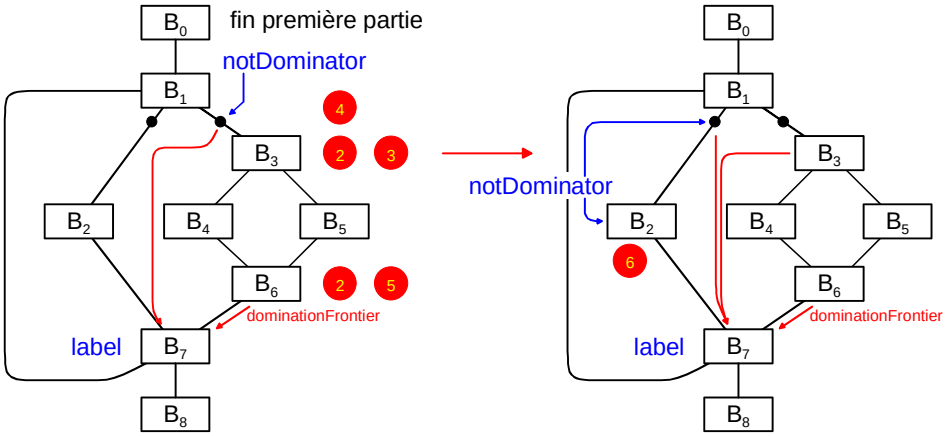
```
void
Function::setDominationFrontier() {
   for (std::vector<VirtualInstruction*>::const_iterator iter=m_instructions.begin(); iter != m_instructions.end();++iter) {
      if ((*iter)->type() == VirtualInstruction::TLabel) {
         assert(dynamic_cast<const LabelInstruction*>(*iter));
```

```
      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;
```