Rapport sur Iris

Un générateur de code optimisant pour x86-64

Hubert Gruniaux

Table des matières

I Introduction

Iris est un générateur de code optimisant pour x86-64. Il utilise une représentation intermédiaire en forme SSA sur laquelle il applique une succession de passes d'optimisation et de simplification. Il utilise aussi la coloration de graphes pour allouer des registres physiques.

Chose utile, Iris peut supporter une multitude de conventions d'appel pour les fonctions. Et supporte nativement, la convention C de l'ABI System 5. Ce qui permet à tout programme utilisant Iris d'appeler des fonctions écrites en C sans aucune difficulté.

Il reste de nombreuses choses à faire sur ce projet. Beaucoup de passes à implémenter, d'autres à améliorer. Simplifier le code, le rendre plus lisible, le documenter. Malheureusement, ce projet a été victime de rush (il y a des deadline) et d'une **mauvaise expérience de ma part en OCaml** (c'est la première fois que j'ai écrit de l'OCaml sur un gros projet). Le code utilisé est assez impératif dans sa globalité. C'est **assumé** mais regretté.

Ce rapport constitue une documentation interne du projet. Il explique toutes les passes implémentées dans le projet et leurs conséquences sur le code. Toutefois, il n'est pas exhaustif. L'implémentation des `switch` est ignorée, le support des structures n'est pas mentionné au même titre que le support des constantes globales (les chaînes de caractères par exemple).

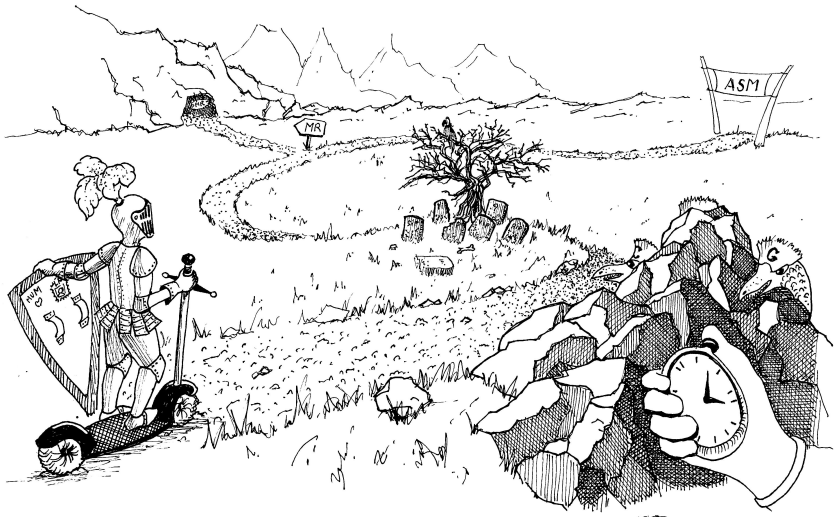
II Passes

Considérons le programme PureScript (compatible PetitPureScript) suivant :

```
module Main where

import Prelude
import Effect
import Effect.Console

f :: Int -> Int
f x =
  if x > 5 then
    let z = 4 + 2 in
    if z < 5 then
      0
    else
      0 - x
  else
    f (x + 1)

main :: Effect Unit
main = do (let y = f 5 in log (show (y + y)))
          pure unit
```

Ce programme n'a pas de but particulier sauf montrer les différentes étapes de la compilation et des optimisations du compilateur MiniPureScript et Iris. En effet, on remarquera que ce programme affiche simplement `-12` en sortie, ce qui n'a rien de particulier.

II.1 Préliminaire



On suppose que l'analyse syntaxique et le typage ont déjà été fait par le frontend. A ce niveau, le compilateur traduit l'arbre typé en la représentation intermédiaire d'Iris. Cette traduction est fait dans le fichier `CodeGen.ml` de MiniPureScript. Pour le programme ci-dessus, on obtient :

```
; f :: Int -> Int
fn _P1fi(int %1) -> int {
L1:
  %2 = 5
  %3 = sgt %1 %2
  jmpc %3 L2 L3
L2: ; preds = L1
  %4 = 4
  %5 = 2
  %6 = add %4 %5
  %7 = 5
  %8 = slt %6 %7
  jmpc %8 L4 L5
L3: ; preds = L1
  %9 = 1
  %10 = add %1 %9
  %11 = call _P1fi(%10)
  jmp L6
L6: ; preds = L3, L7
  %12 = phi [(%13, L7), (%11, L3)]
  ret %12
L4: ; preds = L2
  %14 = 0
  jmp L7
L5: ; preds = L2
  %15 = 0
  %16 = sub %15 %1
  jmp L7
L7: ; preds = L4, L5
  %13 = phi [(%14, L4), (%16, L5)]
  jmp L6
}
; main :: Effect Unit
fn _P4main() -> ptr {
L8:
  %17 = 0
```

```
    %18 = 5
    %19 = call _P1fi(%18)
    %20 = add %19 %19
    %21 = call __prt_show_int(%20)
    %22 = call __prt_log(%21)
    %23 = 0
    ret
}
```

On remarque que la fonction `f :: Int -> Int` a été renommée en `_P1fi` et `main :: Effect Unit` en `_P4main`. Le compilateur MiniPureScript, pour supporter les fonctions polymorphiques, décore le nom des fonctions (*name mangling*) pour distinguer les implémentations de la même fonction en fonction des types. L'algorithme de décoration est beaucoup inspiré de celui du C++ pour l'ABI System 5 et de Rust. Il se trouve dans le projet MiniPureScript, le fichier `NameMangler.ml`.

De même, les fonctions `show` et `log` ont été remplacées par des appels à des fonctions intégrées (implémentées dans le Runtime, en C).

La représentation intermédiare d'Iris est en forme *Static-Single Assignment* (SSA). C'est-à-dire chaque variable est assigné une seule fois (à sa définition). Dans un langage fonctionnel comme PureScript, garantir cette forme est trivial. Seul problème potentiel: les branchements avec l'expression `if`. La solution est d'introduire une pseudo-instruction magique, l'instruction φ. Cette conversion est faite par l'API `IrBuilder` d'Iris qu'utilise MiniPureScript.

Vous remarquerez que la convertion de l'arbre typé vers la IR est assez simple dans sa globalité. Les eventuelles subtilités sont gommées par l'API `IrBuilder` à tel point que le fichier `CodeGen.ml` se résume à des patterns matchings qui traduisent littéralement vers Iris.

C'est à partir de cette étape que le travail du backend commence. **On se concentrera sur `f :: Int -> Int` pour toute la suite**.

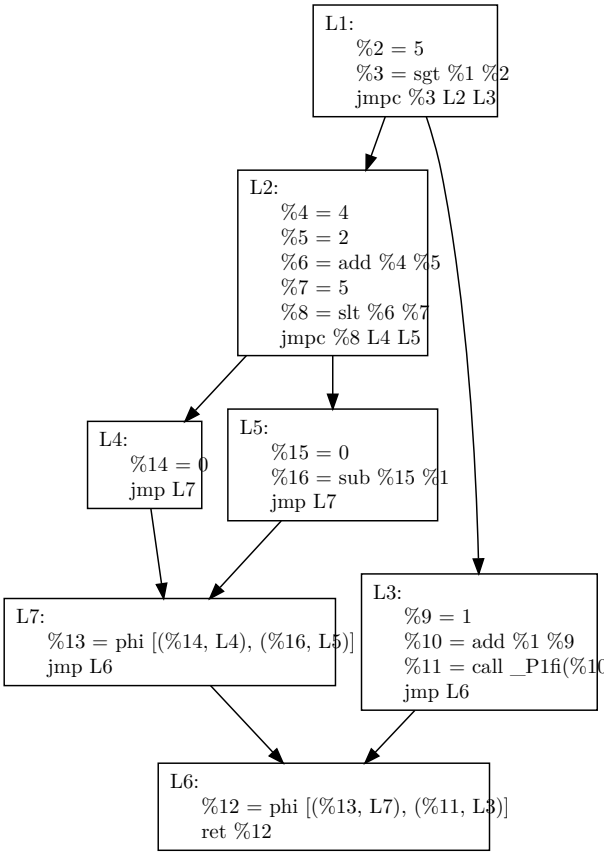


Fig. 2. – Le CFG[1] de `f`

II.1.1 Graphe des appels (`CallGraph`)

Avant d'exécuter toutes les passes, le `PassManager` commence par calculer le graphe des appels pour tout le programme. Cette information lui permet :

- D'une part, de supprimer les fonctions inutilisées à la fin (à condition qu'elles ne soient pas publiques).
- D'autre part, à exécuter les optimisations sur les fonctions dans le post-ordre du parcours du graphe. C'est-à-dire, on essaye d'optimiser les fonctions un maximum avant qu'elles ne soient appelées. Ceci permet d'améliorer l'efficacité des éventuelles passes d'*inlining* de fonction.

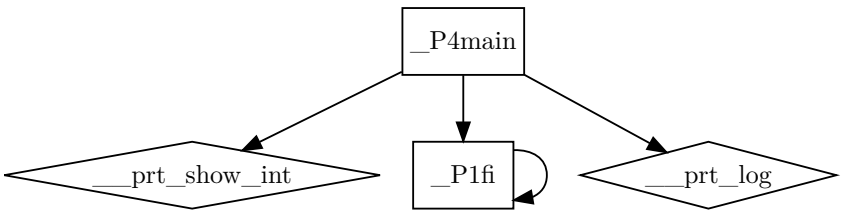


Fig. 3. – Le graphe des appels

[1]Control Flow Graph

II.2 Sur la représentation intermédiataire (`IR`)

La plupart des passes d'optimisation sont exécutées sur la représentation intermédiaire d'Iris. Les raisons sont multiples :

- Cette représentation est générique et indépendante de l'architecture cible. Les optimisations sur cette dernière sont donc universelles.
- L'IR est sous forme SSA[2], ce qui simplifie de nombreuses optimisations.
- En interne, cette représentation est moins flexible, plus typée. Il est plus difficile de faire des erreurs avec.

Iris n'implémente pas encore beaucoup de passes d'optimisation. Pour le moment, seulement les suivantes sont disponibles :

- `SimplifyCFGPass` : Supprime les basic blocks morts et fusionne certains blocks entre eux.
- `CopyPropagationPass` : Propage les valeurs des copies et des constantes autant que possible. Permet de grandement améliorer l'efficacité des passes suivantes.
- `InstCombinePass` :
  - Calcule les instructions qui peuvent être calculées à la compilation (car les opérantes sont toutes constantes).
  - Applique certaines simplifications algébriques.
  - Réécris certaines opérations (associativité, commutativité).
  - Réécris les multiplications et divisions par des shifts si possible.
- `DCEPass` (DeadCodeEliminationPass) : Élimine les valeurs inutilisées dans le code et donc le calcul n'as pas d'effet de bord.

[2]Single-Static Assignment

II.2.1 Propagation des copies (`CopyPropagationPass`)

Cette passe tente au maximum de propager les constantes et les copies de registre. Ceci permet de simplifier le code, de permettre d'autres optimisations (par exemple dans `InstcombinePass`), et d'exploiter facilement les immédiats dans le jeu d'instruction de x86-64.

Avant	Après
`fn _P1fi(int %1) -> int {`	`fn _P1fi(int %1) -> int {`
`L1:`	`L1:`
`  %2 = 5`	`  %2 = 5`
`-  %3 = sgt %1 %2`	
	`+  %3 = sgt %1 5`
`  jmpc %3 L2 L3`	`  jmpc %3 L2 L3`
`L2:`	`L2:`
`  %4 = 4`	`  %4 = 4`
`  %5 = 2`	`  %5 = 2`
`-  %6 = add %4 %5`	
	`+  %6 = add 4 2`
`  %7 = 5`	`  %7 = 5`
`-  %8 = slt %6 %7`	
	`+  %8 = slt %6 5`
`  jmpc %8 L4 L5`	`  jmpc %8 L4 L5`
`L3:`	`L3:`
`  %9 = 1`	`  %9 = 1`
`-  %10 = add %1 %9`	
	`+  %10 = add %1 1`
`  %11 = call _P1fi(%10)`	`  %11 = call _P1fi(%10)`
`  jmp L6`	`  jmp L6`
`L6:`	`L6:`
`  %12 = phi [(%13, L7), (%11, L3)]`	`  %12 = phi [(%13, L7), (%11, L3)]`
`  ret %12`	`  ret %12`
`L4:`	`L4:`
`  %14 = 0`	`  %14 = 0`
`  jmp L7`	`  jmp L7`
`L5:`	`L5:`
`  %15 = 0`	`  %15 = 0`
`-  %16 = sub %15 %1`	
	`+  %16 = sub 0 %1`
`  jmp L7`	`  jmp L7`
`L7:`	`L7:`
`  %13 = phi [(%14, L4), (%16, L5)]`	`  %13 = phi [(%14, L4), (%16, L5)]`
`  jmp L6`	`  jmp L6`
`}`	`}`

Figure 1. – Résultat de la passe

II.2.2 Optimisations locales, algébriques, etc. (`InstCombinePass`)

Cette passe effectue deux types d'optimisations :

Toute expression arithmétique ou logique dont toutes les opérandes sont des immédiats, alors l'expression peut être calculé au moment de la compilation et est être remplacé par le résultat. Par exemple, `%0 = add 1 8` peut être remplacé par `%0 = 9`. C'est ce qu'on appelle le pliage des constantes (*constant folding*).

De plus, certaines expressions algébriques peuvent être simplifié. En voici quelques unes:

- $x + 0 = x$
- $x - x = 0$
- $x \bmod 1 = 0$
- etc.

Actuellement, Iris n'exploite pas les règles d'associativités des opérateurs. Ainsi le code :

```
%1 = add %0 5
%2 = add %1 7
```

Ne sera pas simplifié.

Avant

```
fn _P1fi(int %1) -> int {
L1:
  %2 = 5
  %3 = sgt %1 5
  jmpc %3 L2 L3
L2:
  %4 = 4
  %5 = 2
-  %6 = add 4 2
   %7 = 5
   %8 = slt %6 5
   jmpc %8 L4 L5
L3:
  %9 = 1
  %10 = add %1 1
  %11 = call _P1fi(%10)
  jmp L6
L6:
  %12 = phi [(%13, L7), (%11, L3)]
  ret %12
L4:
  %14 = 0
  jmp L7
L5:
  %15 = 0
-  %16 = sub 0 %1
   jmp L7
L7:
  %13 = phi [(%14, L4), (%16, L5)]
  jmp L6
}
```

Après

```
fn _P1fi(int %1) -> int {
L1:
  %2 = 5
  %3 = sgt %1 5
  jmpc %3 L2 L3
L2:
  %4 = 4
  %5 = 2
+  %6 = 6
  %7 = 5
  %8 = slt %6 5
  jmpc %8 L4 L5
L3:
  %9 = 1
  %10 = add %1 1
  %11 = call _P1fi(%10)
  jmp L6
L6:
  %12 = phi [(%13, L7), (%11, L3)]
  ret %12
L4:
  %14 = 0
  jmp L7
L5:
  %15 = 0
+  %16 = neg %1
  jmp L7
L7:
  %13 = phi [(%14, L4), (%16, L5)]
  jmp L6
}
```

Figure 2. – Résultat de la passe

Observation — On observe que `%6` vaut désormais la constante `6`, en effet `4 + 2 = 6`. De plus, la soustraction dans `%16` a été remplacée par une négation (`0 - x = x`), ce qui permet plus tard de faciliter la sélection d'une meilleure instruction machine pour calculer cette ligne (`neg` de x86 au lieu de la soustraction).

II.2.3 Élimination du code mort (`DCEPass`)

Avant	Après
`fn _P1fi(int %1) -> int {`	`fn _P1fi(int %1) -> int {`
`L1:`	`L1:`
`-  %2 = 5`	
`  %3 = sgt %1 5`	`  %3 = sgt %1 5`
`  jmpc %3 L2 L3`	`  jmpc %3 L2 L3`
`L2:`	`L2:`
`-  %4 = 4`	
`-  %5 = 2`	
`  %6 = 6`	`  %6 = 6`
`-  %7 = 5`	
`  %8 = slt %6 5`	`  %8 = slt %6 5`
`  jmpc %8 L4 L5`	`  jmpc %8 L4 L5`
`L3:`	`L3:`
`-  %9 = 1`	
`  %10 = add %1 1`	`  %10 = add %1 1`
`  %11 = call _P1fi(%10)`	`  %11 = call _P1fi(%10)`
`  jmp L6`	`  jmp L6`
`L6:`	`L6:`
`  %12 = phi [(%13, L7), (%11, L3)]`	`  %12 = phi [(%13, L7), (%11, L3)]`
`  ret %12`	`  ret %12`
`L4:`	`L4:`
`  %14 = 0`	`  %14 = 0`
`  jmp L7`	`  jmp L7`
`L5:`	`L5:`
`-  %15 = 0`	
`  %16 = neg %1`	`  %16 = neg %1`
`  jmp L7`	`  jmp L7`
`L7:`	`L7:`
`  %13 = phi [(%14, L4), (%16, L5)]`	`  %13 = phi [(%14, L4), (%16, L5)]`
`  jmp L6`	`  jmp L6`
`}`	`}`

Figure 3. – Résultat de la passe

Observation — Les valeurs `%5`, `%7`, `%9` et `%15` ont été supprimées du code. Ceci est possible, car elles ne sont utilisées nulle part. De plus, elles n'ont pas d'effet de bord, c'est-à-dire pas

de comportement observable depuis l'extérieur. Autrement dit, elles peuvent être simplement supprimées sans conséquences. Ça simplifie le code !

La plupart des instructions sont sans effet de bord (en particulier toutes les instructions arithmétiques et logiques). Quelques cas particuliers :

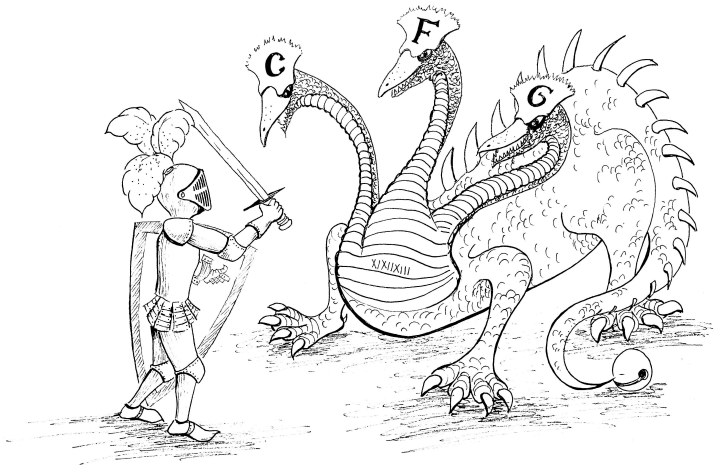
- Les appels de fonction peuvent avoir des effets de bord. La fonction appelée peut par exemple créer des fichiers, faire des appels réseau, etc.
- Les écritures en mémoire. D'autres parties du programme peuvent dépendre de ces valeurs en mémoire.

Ces passes d'optimisations sont exécutées tant que le code ne converge pas. Certaines optimisations en ouvrent d'autres. Voici par exemple le résultat après quelques itérations (le code a convergé) :

Avant	Après
`fn _P1fi(int %1) -> int {`	`fn _P1fi(int %1) -> int {`
`L1:`	`L1:`
`-  %2 = 5`	
`  %3 = sgt %1 5`	`  %3 = sgt %1 5`
`  jmpc %3 L2 L3`	`  jmpc %3 L2 L3`
`L2:`	`L2:`
`-  %4 = 4`	
`-  %5 = 2`	
`-  %6 = 6`	
`-  %7 = 5`	
`-  %8 = slt %6 5`	
`-  jmpc %8 L4 L5`	
	`+  jmp L5`
`L3:`	`L3:`
`-  %9 = 1`	
`  %10 = add %1 1`	`  %10 = add %1 1`
`  %11 = call _P1fi(%10)`	`  %11 = call _P1fi(%10)`
`  jmp L6`	`  jmp L6`
`L6:`	`L6:`
`  %12 = phi [(%13, L7), (%11, L3)]`	`  %12 = phi [(%13, L7), (%11, L3)]`
`  ret %12`	`  ret %12`
`-L4:`	`-L4:`
`+L4:`	`+L4:`
`  %14 = 0`	`  %14 = 0`
`  jmp L7`	`  jmp L7`
`L5:`	`L5:`
`-  %15 = 0`	
`  %16 = neg %1`	`  %16 = neg %1`
`  jmp L7`	`  jmp L7`
`L7:`	`L7:`
`  %13 = phi [(%14, L4), (%16, L5)]`	`  %13 = phi [(%14, L4), (%16, L5)]`
`  jmp L6`	`  jmp L6`
`}`	`}`

Figure 4. – Résultat de la passe

II.2.4 Simplification du CFG (`SimplifyCFGPass`)



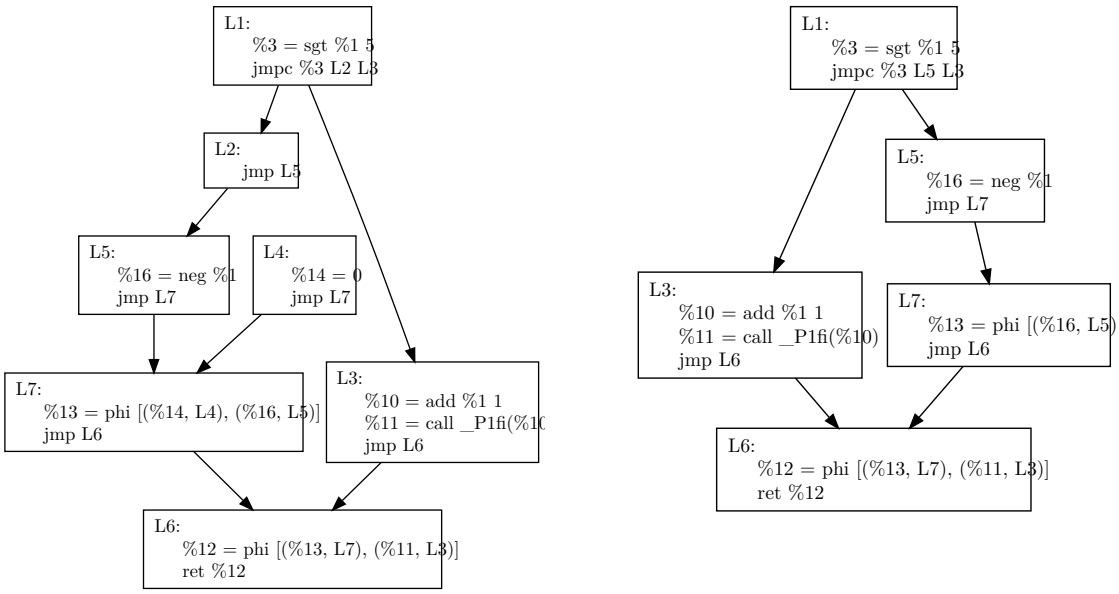


Fig. 7. – CFG avant et après simplification

Observation — Le basic block `L4` a été supprimé car il est inatteignable. En effet, il n'as pas de prédecesseur et il n'est pas le block d'entré de la fonction. De plus, le basic block `L2` a été fusionné dans le basic block `L5` (il n'as pas d'instructions sauf un jump vers `L5`).

Les conséquences sur le code IR textuel est donné ci-dessous :

Avant	Après
`fn _P1fi(int %1) -> int {`	`fn _P1fi(int %1) -> int {`
`L1:`	`L1:`
`  %3 = sgt %1 5`	`  %3 = sgt %1 5`
`-  jmpc %3 L2 L3`	
`-L2:`	`-L2:`
`-  jmp L5`	
	`+  jmpc %3 L5 L3`
`L3:`	`L3:`
`  %10 = add %1 1`	`  %10 = add %1 1`
`  %11 = call _P1fi(%10)`	`  %11 = call _P1fi(%10)`
`  jmp L6`	`  jmp L6`
`L6:`	`L6:`
`  %12 = phi [(%13, L7), (%11, L3)]`	`  %12 = phi [(%13, L7), (%11, L3)]`
`  ret %12`	`  ret %12`
`-L4:`	`-L4:`
`-  %14 = 0`	
`-  jmp L7`	
`-L5:`	`-L5:`
`+L5:`	`+L5:`
`  %16 = neg %1`	`  %16 = neg %1`
`  jmp L7`	`  jmp L7`
`-L7:`	`-L7:`
`-  %13 = phi [(%14, L4), (%16, L5)]`	
`+L7:`	`+L7:`
	`+  %13 = phi [(%16, L5)]`
`  jmp L6`	`  jmp L6`
`}`	`}`

Figure 5. – Résultat de la passe

II.2.5 Supression des instructions φ (`LowerPhiPass`)

Bien sûr, les passes précédentes sont encore une fois exécutées en boucle tant que le code ne converge pas. Après quelques itérations, on obtient le code suivant :

Avant

```
fn _P1fi(int %1) -> int {
L1:
  %3 = sgt %1 5
  jmpc %3 L5 L3
L3:
  %10 = add %1 1
  %11 = call _P1fi(%10)
-  jmp L6
-L6:
-  %12 = phi [(%13, L7), (%11, L3)]
-  ret %12
L5:
  %16 = neg %1
  jmp L7
L7:
  %13 = phi [(%16, L5)]
-  jmp L6
+L8:
}
```

Après

```
fn _P1fi(int %1) -> int {
L1:
  %3 = sgt %1 5
  jmpc %3 L5 L3
L3:
  %10 = add %1 1
  %11 = call _P1fi(%10)
-L6:
+  jmp L8
L5:
  %16 = neg %1
  jmp L7
L7:
  %13 = phi [(%16, L5)]
+  jmp L8
+L8:
+  %12 = phi [(%13, L7), (%11, L3)]
+  ret %12
}
```

Figure 6. – Récapitulatif de l'IR

Il reste alors une étape importante avant de passer à une représentation plus proche de la machine : l'élimination des nœuds φ. Ces derniers sont très utiles pour préserver la forme SSA[3], mais ils ne sont désormais plus nécessaires.

Pour supprimer un nœud φ, il suffit d'insérer des instructions `MOV` (représentées dans l'IR par `%0 = %1`) à la fin des basic blocks prédécesseurs. Évidemment, cette transformation fait perdre la forme SSA.

Cette passe, c'est aussi l'occasion de simplifier certains nœuds φ. Notamment, ceux qui ont un seul prédécesseur (e.g. `%0 = phi [( %1, L1 )]`). Ceux-là peuvent être simplifiés par une seule instruction `MOV`.

Le résultat de la passe est donné ci-dessous.

[3]Static-Single Assigment

Avant	Après
`fn _P1fi(int %1) -> int {`	`fn _P1fi(int %1) -> int {`
`L1:`	`L1:`
`  %3 = sgt %1 5`	`  %3 = sgt %1 5`
`  jmpc %3 L5 L3`	`  jmpc %3 L5 L3`
`L3:`	`L3:`
`  %10 = add %1 1`	`  %10 = add %1 1`
`  %11 = call _P1fi(%10)`	`  %11 = call _P1fi(%10)`
	`+ %12 = %11`
`  jmp L8`	`  jmp L8`
`L5:`	`L5:`
`  %16 = neg %1`	`  %16 = neg %1`
	`+ %13 = %16`
`  jmp L7`	`  jmp L7`
`L7:`	`L7:`
`- %13 = phi [(%16, L5)]`	
	`+ %12 = %13`
`  jmp L8`	`  jmp L8`
`L8:`	`L8:`
`- %12 = phi [(%13, L7), (%11, L3)]`	
`  ret %12`	`  ret %12`
`}`	`}`

Figure 7. – Résultat de la passe

Observation — La valeur `%13` est directement remplacée par `%16` car le nœud φ ne possède qu'un seul précesseur : `%16` (c'est un cas simple que l'optimisateur saisit). Dans le cas `%12`, des moves sont insérés dans les basic blocks prédecesseurs (`L7` et `L3`). On remarque deux choses importantes :

- Le code final ne contient plus de nœuds φ (le backend n'a pas à gérer ces derniers).
- Le code n'est plus en forme SSA. En effet `%12` est désormais assignée deux fois (dans `L3` et `L7`).

II.3 Sur la représentation machine (`MR`)

Toutes les bonnes choses ont une fin. Bien que la représentation intermédiare a de nombreux avantages, il faut à un moment travailler plus proche de l'architecture cible. Ainsi, le code IR[4] est converti en code MR[5]. Cette convertion est spécifique à chaque architecture, et pour x86-64, son implémentation se trouve dans le fichier `X86Instsel.ml`.

De nombreux codes travaillant sur le MR dans Iris veulent être la plus architecture-agnostique possible. Ce qui implique, que le pretty-printer n'a aucune connaissance des réels noms des registres physiques. Pour se retrouver, voici une correspondance entre les registres internes d'Iris, et leur correspondance physique pour x86-64 :

Nom dans Iris	Nom sur x86-64
`r0`	`rax`
`r1`	`rbx`
`r2`	`rcx`
`r3`	`rdx`
`r4`	`rsp`
`r5`	`rbp`
`r6`	`rsi`
`r7`	`rdi`
`r8`, …, `r15`	`r8`, …, `r15`
`r50`	`rflags`

Fig. 9. – Nom des registres physiques

Voici le résultat de la sélection des instructions pour notre fonction `f :: Int -> Int`. On maintient encore l'usage des registres virtuels, et certaines instructions sont encore de haut-niveau : les appels de fonctions, les paramètres de la fonction, les jumps conditionnels prennent deux labels, etc.

```
; f :: Int -> Int
```

[4]Intermediate Representation
[5]Machine Representation

```
fn _P1fi(int %1) {
L1:
  cmp %1, 5 ; defs = {r50}, uses = {%1}
  setg %3 ; defs = {%3}, uses = {r50}
  and %3, 255 ; defs = {r50, %3}, uses = {%3}
  cmp %3, 0 ; defs = {r50}, uses = {%3}
  jz L3, L5 ; defs = {}, uses = {r50}
L3: ; preds = L1
  mov %10, %1 ; defs = {%10}, uses = {%1}
  add %10, 1 ; defs = {r50, %10}, uses = {%10}
  call %11, _P1fi, %10 ; defs = {r0, r2, r3, r6, r7, r8, r9}, uses = {}
  mov %12, %11 ; defs = {%12}, uses = {%11}
  jmp L8 ; defs = {}, uses = {r50}
L5: ; preds = L1
  mov %16, %1 ; defs = {%16}, uses = {%1}
  neg %16 ; defs = {r50, %16}, uses = {%16}
  mov %13, %16 ; defs = {%13}, uses = {%16}
  jmp L7 ; defs = {}, uses = {r50}
L7: ; preds = L5
  mov %12, %13 ; defs = {%12}, uses = {%13}
  jmp L8 ; defs = {}, uses = {r50}
L8: ; preds = L3, L7
  mov r0, %12 ; defs = {r0}, uses = {%12}
  ret  ; defs = {}, uses = {r0, r1, r4, r5, r12, r13, r14, r15}
}
```

Observations — Les ensembles `defs` et `uses` définies pour chaque instruction spécifie quelles registres sont définies ou utilisés par quelles instructions. Ces informations sont utiles pour l'allocation des registres.

II.3.1 Implémentation des appels (`LowerCallsPass`)

Lors de la sélection des instructions, les appels de fonction sont resté de haut-niveau. Cette passe remédie à ce problème, et s'occupe d'insérer les instructions `MOV` et `PUSH` pour charger les arguments, et à récupérer la valeur de retour de l'appel.

L'implémentation actuelle de la passe est très flexible et supporte une très grande variété de convention d'appel. Il suffit d'implémenter la structure `Mr.calling_convention_info` :

```
    cc_caller_saved : Reg.set;  (** The caller-saved (volatile) registers. *)
    cc_callee_saved : Reg.set;  (** The callee-saved (non-volatile) registers. *)
    cc_args_regs : Reg.t list;
        (** A list of physical registers used to pass arguments, in order. *)
    cc_args_regs_count : int;
         (** Count of arguments that are passed by physical registers. Must be the
    length of [cc_args_regs]. *)
    cc_args_stack_ltr : bool;
        (** True if the arguments in the stack are passed from left to right; false
    otherwise. *)
    cc_return_reg : Reg.t option;
        (** The physical register where the function's return value must be stored.
    None if the return value is stored in the stack. *)
    cc_caller_cleanup : bool;
         (** True if the caller has to clean the stack (pop arguments passed in the
    stack if any),false if it is the callee. *)
```

Et la passe s'adapte.

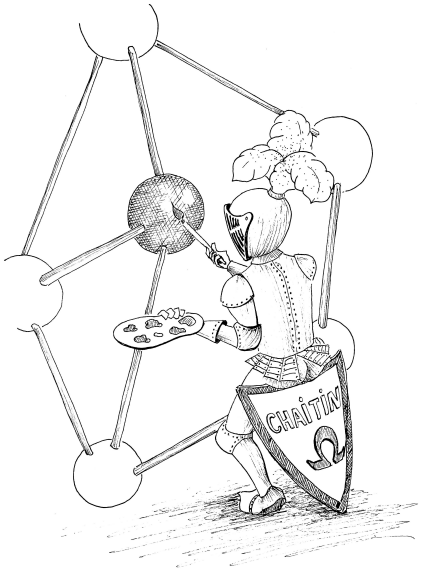
Par exemple, l'appel récursif à `f` est traduit par :

Avant	Après
	`+ mov r7, %10`
`- call %11, _P1fi, %10`	
	`+ call _P1fi`
	`+ mov %11, r0`

Figure 8. – Résultat de la passe

Observations — L'argument est passé dans `r7` (qui correspond à `rdi` sur x86-64) et la valeur de retour est récupée dans `r0` (qui correspond à `rax`). La fonction `f` suit la convention d'appel du langage C définie dans l'ABI System 5.

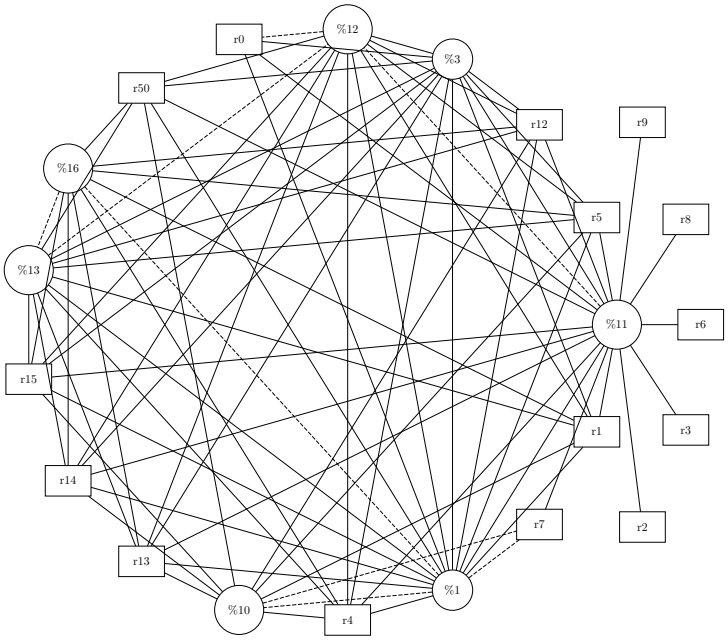
II.3.2 Allocation des registres (`RegAlloc`)



Probablement l'une des étapes les plus importantes du compilateur : l'allocation des registres. Iris supporte, dans sa représentation intermédiaire, une infinité de registres appelés registres virtuels. Malheureusement, la machine ne supporte qu'un nombre limité de registres. Il serait possible d'utiliser la pile pour simuler les registres virtuels, mais une meilleure solution est d'associer, au moins autant que possible, une partie des registres virtuels à des registres physiques. C'est l'allocation des registres.

Dans Iris, cette allocation utilise l'algorithme de Chaitin modifié par George et Appel. Il tente au maximum de *coalescer* les registres liés par des instructions `MOV`, et pour cela utilise le critère de George.

L'implémentation se trouve dans le fichier `RegAlloc.ml`. Et exploite le graphe d'interférence calculé dans `Interference.ml` et `Liveness.ml`. Ce dernier est donné ci-dessous :



RegV	Couleur
%1	r3
%3	r2
%16	r0
%13	r0
%10	r0
%11	Spilled 0
%12	r0

Fig. 11. – Graphe d'interférence

II.3.3 Débordement des registres (NaiveSpillerPass)

Après l'allocation des registres, il est possible que certains d'entre eux ne puissent être stockés dans un registre physique. Dans ce cas, ils sont *spillés* et sont stockés sur la pile, au sein de la *frame* de la fonction. Comme les processeurs ne supportent pas les lectures et écritures en mémoire comme avec les registres physiques, le code doit être modifié. En particulier, des instructions permettant de récupérer la valeur à partir de la pile et de la sauvegarder ensuite doivent être insérées. C'est le but de cette passe.

Malheureusement, pour diverses raisons (temps et bogue), l'implémentation actuelle est terriblement naïve comme le suggère le nom. Dans le cas de x86, deux registres (`r10` et `r11`) sont réservés pour implémenter le spillage.

Avant	Après
`fn _P1fi() {`	`fn _P1fi() {`
`L1:`	`L1:`
`  mov %1, r7`	`  mov %1, r7`
`  cmp %1, 5`	`  cmp %1, 5`
`  setg %3`	`  setg %3`
`  and %3, 255`	`  and %3, 255`
`  cmp %3, 0`	`  cmp %3, 0`
`  jz L3, L5`	`  jz L3, L5`
`L3:`	`L3:`
`  mov %10, %1`	`  mov %10, %1`
`  add %10, 1`	`  add %10, 1`
`  mov r7, %10`	`  mov r7, %10`
`  call _P1fi`	`  call _P1fi`
`-  mov %11, r0`	
	`+  mov STACK[0], r10`
	`+  mov r10, r0`
	`+  mov STACK[0], r10`
`  add r4, 0`	`  add r4, 0`
`  mov r4, r4`	`  mov r4, r4`
`-  mov %12, %11`	
	`+  mov r10, STACK[0]`
	`+  mov %12, r10`
`  jmp L8`	`  jmp L8`
`L5:`	`L5:`
`  mov %16, %1`	`  mov %16, %1`
`  neg %16`	`  neg %16`
`  mov %13, %16`	`  mov %13, %16`
`  jmp L7`	`  jmp L7`
`L7:`	`L7:`
`  mov %12, %13`	`  mov %12, %13`
`  jmp L8`	`  jmp L8`
`L8:`	`L8:`
`  mov r0, %12`	`  mov r0, %12`
`  ret`	`  ret`
`}`	`}`

Figure 9. – Résultat de la passe

Observations — Pour toutes utilisations et définitions du pseudo-registre `%12`, des instructions `MOV` de x86 ont été insérées (écriture et lectures sur la frame de la fonction).

II.3.4 Supression des pseudos registres (`RewriteVRegsPass`)

L'une des dernières étapes avant l'émission du code assembleur : la suppression totale des pseudo-registres. Cette passe remplace toute occurrence d'un pseudo-registre par le registre physique associé (donné par l'allocation des registres). Ainsi, après l'emetteur du code assembleur n'as plus à se soucier des pseudo-registres et peut faire son travail plus simplement.

II.3.5 Ajout du prologue et de l'épilogue (`PrologEpilogPass`)

Ultime étape avant la génération du code assembleur ! L'insertion du prologue et l'épilogue de la fonction.

Le prologue est le bout de code chargé de sauvegarder le *frame pointer* et d'allouer l'espace nécessaire sur la pile pour les variables locales (à ce moment, on sait exactement combien de variables vont vivre sur la pile). De même, le code épilogue est chargé de restaurer le *frame pointer* et de libérer la place allouée sur la pile.

On n'affichera pas la totalité du code MR à la suite de cette passe, car rien ne change à part quelques instructions qui sont insérées au début de la fonction et avant chaque instruction `RET`.

Cependant, voici quand même le code du prologue sur x86 :

```
push rbp
mov rbp, rsp
sub rsp, 8   ; alloue 8 octets sur la pile
and rsp, -16 ; pour aligner la pile sur 16 octets (convention d'appel du C)
```

et de l'épilogue :

```
mov rsp, rbp
pop rbp
ret
```

II.4 Émission du code assembleur (`X86Emit`)

Annexe A : Représentation intermédiaire (IR)

La représentation intermédiaire (IR) est spécifié dans le fichier OCaml `Ir.ml`. Différentes fonctions pour la manipuler sont fournis dans `Instruction.ml`, `Terminator.ml`, `BasicBlock.ml`, etc.

L'IR est constitué d'un contexte. C'est ce dernier qui recense toutes les fonctions (externes ou non) définies et utilisés dans le programme. Elle y maintient, en particulier, une table des symboles pour les fonctions. Les constantes globales sont aussi stocké dans le contexte.

Une fonction au sein de l'IR est simplement un Control-Flow Graph (CFG) de Basic Block (BB). Un basic block est une séquence d'instruction terminé par une instruction terminatrice (Terminator). Toutes les instructions au sein d'un basic block n'ont qu'un seul prédecesseur et qu'un seul successeur, sauf la première instruction (qui peut avoir plusieurs predecesseurs) ou la dernière, le terminateur (qui peut avoir plusieurs successeurs).

Liste des terminateurs :

- `ret` : termine la fonction et ne renvoie pas de valeur.
- `ret %value` : termine la fonction et renvoie la valeur `%value`.
- `jmp LABEL` : saute unconditionnellement vers le basic block `LABEL`.
- `jmpc %value THEN_LABEL ELSE_LABEL` : si `%value` est nulle, alors saute vers le basic block `ELSE_LABEL`, autrement saute vers le basic block `THEN_LABEL`.
- `switch %value OTHERWISE [ (%0, L0), ..., (%n, Ln) ]` : si `%value` a l'une des valeurs spécifiés par les `%i` alors saute vers le basic block `Li`, sinon saute vers le basic block `OTHERWISE`.
- `unreachable` : signale que la fin du basic block ne peut être atteinte (par exemple après un appel de fonction terminale ou une fonction ne renvoyant, tel que `abort()`).

Liste des instructions :

-

L'aventure s'arrête enfin. On quitte totalement le MR[6] et l'IR[7] pour embrasser entièrement le code assembleur ! Il est temps de générer le fichier assembleur final qui pourra être donné à votre assembleur préféré puis à votre éditeur de lien.

Cette partie est extrêmement spécifique à l'architecture cible. Ici, comme demandé dans le sujet, on se limitera à x86. De plus, la syntaxe utilisée est celle d'Intel (une directive `.intel_syntax` est émise au début du fichier pour GNU As).

L'émetteur de x86 est assez simple : il parcourt le CFG de la fonction et émet une instruction à la fois. La traduction est triviale, car la MR est déjà très proche du code machine. La seule subtilité concerne les sauts et sauts conditionnels. En particulier, l'émetteur s'arrange pour minimiser le nombre de sauts à générer (en produisant le code du basic block successeur directement après).

C. Conclusion

Malheureusement, Iris n'arrive pas à réduire le code à un simple appel de `log` avec la constante `-12`. Il est donc moins efficace que le compilateur officiel d'Ocaml, que GCC ou bien encore que Clang. Sans étonnement.

Cependant, pour de nombreux tests fournis, Iris ne produit jamais les instructions qui sont censé être testés. Le passe `InstCombine` simplifie les opérateurs arithmétiques, et donc à la fin

[6] Machine Representation

[7] Intermediate Representation

aucune instruction `add`, `div`, etc. n'est généré dans les tests `arith*.purs` (à condition d'activer les optimisations).

Quelques optimisations que pourrait faire Iris en plus :

- Elimination du code mort au niveau du MR. Mais contrairement à l'IR, le code n'est plus sous forme SSA ce qui complexifie la tache.
- Propagation des copies et des constantes au niveau du MR. De même, ne pas avoir la forme SSA complexifie le problème.
- Propagation des opérandes mémoires au niveau du MR.
- *Inlining* des fonctions au niveau de l'IR. Cela ne devrait pas être trop compliqué ; l'IR est suffisament de haut niveau et est sous forme SSA ce qui simplifie la β-réduction. La vraie difficulté, je suppose, c'est de déterminer quand *inliner* ou non.
- Exploiter l'associativité des opérateurs binaires dans `InstCombine`.
- Faire de la propagation de copie globalement sur la fonction. Actuellement, le passe le fait qu'au sein d'un basic block. Il faudrait pour ça utiliser l'arbre des dominateurs.
- Etc. Cela ne finit jamais.