Implémentation d'une structure de données pour l'allocateur de registres du compilateur *Cranelift* en *Rust*

Valentin Millet, Nabil Lamrabet

Mai 2020

Mots-clés Compilation, CraneLift, Rust, WebAssembly, allocateur de registres, BitSet.

1 Introduction

Ce POM [1] a été proposé et encadré par Matthieu Moy et Laure Gonnord (CASH, LIP/ENS Lyon, Université Claude Bernard Lyon 1).

La Bytecode Alliance [2] est une organisation composée de Mozilla, Fastly, Intel et Red Hat. Elle a pour objectif de poser les fondations d'un écosystème permettant d'exécuter de façon sécurisée et performante des applications sans tenir compte de leur plateforme.

Son travail porte actuellement sur *WebAssembly* : un format de bytecode [3] pensé pour le Web qui a également pour but d'être universel. Il s'agit d'un standard récent venant compléter JavaScript [4] et dont le but est d'apporter des performances proches d'applications natives pour des applications Web (en offrant une meilleure portabilité) avec de solides garanties au niveau de la sécurité.

Cranelift est un compilateur pour *WebAssembly* (*JiT* [5] et *AoT* [6]) dont le focus est porté sur le temps de compilation. Il a pour objectif de prendre n'importe quel langage pour le transformer en représentation intermédiaire [7] puis en en bytecode exécutable. Au début du projet il avait pour but d'être intégré à *Firefox* mais n'était pas encore celui utilisé pour *WebAssembly* (ce qui est désormais le cas).

Le projet proposé a pour contexte une collaboration avec Benjamin Bouvier, un ingénieur compilation chez Mozilla travaillant sur ce projet. Ce dernier avait au préalable identifié des contributions accessibles à notre niveau (rapports de bugs, nouvelles fonctionnalités, amélioration de performances des algorithmes). Les objectifs ont donc été les suivants :

— Participer au développement d'un logiciel open-source important.
— Apprendre un langage moderne (*Rust* [8]) et comprendre les nouveaux concepts qu'il apporte.
— Comprendre les enjeux de la compilation.

1. Projet d'Orientation en Master, Unité d'enseignement dispensée à l'Université Lyon 1, dans le cadre du Master 1 d'Informatique
2. https ://bytecodealliance.org/
3. Code intermédiaire executable uniquement via une machine virtuelle.
4. https ://developer.mozilla.org/en-US/docs/Web/JavaScript
5. Just in time : seul les portions de code nécessaires au moment de l'exécution sont compilées
6. Ahead of time : le code est entièrement compilé avant l'exécution du programme
7. https ://en.wikipedia.org/wiki/Intermediate_representation
8. https ://www.rust-lang.org/

1.1 Cranelift

Un compilateur est un logiciel dont le but est de transformer un programme écrit par un humain (en langage C par exemple) en du code exécutable par une machine.

Le compilateur *Cranelift* est écrit dans le langage *Rust*, un langage conçu et développé par Mozilla depuis moins d'une dizaine d'années, ce langage est très innovant par son système de borrow checker[9] qui permet d'obtenir les performances d'un langage comme le C et la sûreté des accès mémoire habituellement réservée aux langages de plus haut niveau.

Les contributions de ce projet sont :

— Résolution d'un bug et compréhension du mécanisme d'ajout de fonctionnalités dans un projet open-source.

— Implémentation d'une structure de donnée *BitSet* pour accélérer le temps passé dans l'allocateur de registres de *Cranelift*.

Initialement, nous devions implémenter des passes d'optimisations simples ("constantfolding", "constant propagation", "jump threading" et "tail duplication") mais Benjamin Bouvier nous a proposé de rejoindre l'allocateur de registres expérimental sur lequel il travaillait. Cet allocateur vient récemment de devenir celui utilisé par *Cranelift*.

2 État de l'art de l'allocation de registres

Il y a différentes étapes lors de la compilation [1] d'un programme, on ne s'intéresse pas aux étapes qui précédent la construction d'une représentation intermédiaire (analyse lexical, syntaxique, sémantique, typage).

Cranelift dispose d'un format de représentation intermédiaire appelé clif. Une représentation intermédiaire est un code comportant des fonctions et des blocks. Chaque fonction comporte un certain nombre de block. Un block est une suite d'instructions, chaque block comporte des points d'entré de sortie.

Voici une fonction en C function qui calcule la moyenne d'un tableau de floats[10] :

```
float average(const float *array, size_t count) {
    double sum = 0;
    for (size_t i = 0; i < count; i++)
        sum += array[i];
    return sum / count;
}
```

Voici la même fonction compilée au format de représentation intermédiaire de *Cranelift* :

```
function %average(i32, i32) -> f32 system_v {
    ss0 = explicit_slot 8          ; Stack slot for ``sum``.

block1(v0: i32, v1: i32):
    v2 = f64const 0x0.0
    stack_store v2, ss0
    brz v1, block5                 ; Handle count == 0.
    jump block2

block2:
    v3 = iconst.i32 0
    jump block3(v3)
```

9. https ://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html
10. https ://cranelift.readthedocs.io/en/stable/ir.html

```
block3(v4: i32):
    v5 = imul_imm v4, 4
    v6 = iadd v0, v5
    v7 = load.f32 v6              ; array[i]
    v8 = fpromote.f64 v7
    v9 = stack_load.f64 ss0
    v10 = fadd v8, v9
    stack_store v10, ss0
    v11 = iadd_imm v4, 1
    v12 = icmp ult v11, v1
    brnz v12, block3(v11)           ; Loop backedge.
    jump block4

block4:
    v13 = stack_load.f64 ss0
    v14 = fcvt_from_uint.f64 v1
    v15 = fdiv v13, v14
    v16 = fdemote.f32 v15
    return v16

block5:
    v100 = f32const +NaN
    return v100
}
```

Par exemple on remarque que le point de sortie du block1 est relié au point d'entré du block2 ("jump block2"). Avec ces liaisons on peut construire un graphe orienté qui trace tous les chemins qu'un programme peut prendre lors de son exécution, il s'agit du graphe de flot de contrôle.

On se sert de la représentation intermédiaire et du graphe de flot de contrôle [2] pour allouer nos registres.

On s'intéresse à l'allocation de registres car elle représente une bonne partie du temps de compilation. D'après nos benchmarks, elle représente au moins 30% du temps total de compilation.

```
Running `/home/na/wasmtime/target/release/clif-util wasm --target arm64
-vdT /home/na/cranelift-new-backend-tests/bz2/bz2.wasm`
Handling: "/home/na/cranelift-new-backend-tests/bz2/bz2.wasm"
Translating... ok
Compiling... ======== ========  ==================================
   Total     Self  Pass
-------- --------  ----------------------------------
   0.016    0.000  Translate WASM module
   0.016    0.016  Translate WASM function
   0.043    0.040  Verify Cranelift IR
   0.000    0.000  Verify CPU flags
   0.853    0.052  Compilation passes
   0.002    0.002  Control flow graph
   0.002    0.002  Dominator tree
   0.000    0.000  Remove unreachable blocks
   0.757    0.757  Register allocation
   0.001    0.001  Binary machine code emission
======== ========  ==================================
```

La phase d'allocation de registres est donc une phase d'optimisation importante pour le compilateur et il existe un grand nombre de stratégies [3] pour allouer les variables d'un programmes.

Un registre est un emplacement pouvant stocker des valeurs (variables, instructions). Il s'agit de l'emplacement le plus rapide car il est directement présent sur le CPU [11], c'est en revanche le moins disponible en nombre. Lorsqu'on souhaite avoir un code performant, on souhaite allouer le plus de variables possibles dans ces registres or ce n'est pas toujours possible et dans ces cas certaines variables doivent être allouées en mémoire (ce qui est une opération bien plus lente et qu'on souhaite éviter).

Il existe deux types de registres : un registre réel et un registre virtuel.

Un registre réel correspond à ce qu'on vient de décrire : il représente le registre physique utilisé dans le CPU. Un registre virtuel est un registre dont on ne donne pas de correspondance physique (registre physique ou mémoire).

Nous représentons un registre par le type *Reg*. Il s'agit simplement d'un entier sur 32 bits.

```
pub struct Reg {
  reg: u32,
}
```

De ce fait cet entier est composé de deux façons différentes.

Pour les deux types, le bit de poids fort détermine s'il s'agit d'un registre réel (0) ou virtuel (1). Les 3 bits qui suivent déterminent la classe de registre.

Pour le registre réel, nous avons 12 bits inutilisés, 8 bits pour l'encodage et 8 bits pour l'index. Pour le registre virtuel les 28 bits restant sont utilisés pour l'index.

```
Real Reg:      0  rc:3  uu:12  enc:8  index:8
Virtual Reg:   1  rc:3                index:28
```

On ne s'intéresse pas à la classe des registres ni à l'encodage des registres réels. On s'intéresse uniquement au type du registre et à son index car ces deux informations, ensembles, forment un identifiant unique pour le registre.

Il peut y avoir 2^8 (256) registres réels mais il n'y a pas à notre connaissance des architectures supportant plus 64 registres. On peut avoir 2^{28} (268 435 456) registres virtuels.

3 Implémentation d'une structure de données pour l'allocateur de registres

Lorsqu'on souhaite allouer nos registres, on souhaite connaître la durée de vie des registres, il s'agit de la phase de liveness analysis. Cette phase utilise massivement la structure de données "ensemble". L'objectif de cette structure est de déterminer quels éléments appartiennent à cet ensemble. Ici elle intervient principalement pour savoir quels registres sont vivants à l'entré d'un block donné (ensemble livein) et quels registres sont vivants en sortie dudit block (ensemble liveout).

Le prototype de la fonction qui effectue ces calculs est la suivante :

```
#[inline(never)]
pub fn calc_livein_and_liveout<F: Function>(
    func: &F,
    def_sets_per_block: &TypedIxVec<BlockIx, Set<Reg>>,
    use_sets_per_block: &TypedIxVec<BlockIx, Set<Reg>>,
    cfg_info: &CFGInfo,
    univ: &RealRegUniverse,
) -> (
    TypedIxVec<BlockIx, Set<Reg>>,
    TypedIxVec<BlockIx, Set<Reg>>,
) {
```

11. central processing unit, en français unité centrale de traitement ou processeur.

On l'utilise également pour calculer d'autres ensembles (on peut noter "def_sets_per_block" et "use_sets_per_block").

L'allocateur utilise une structure de données appelée *Set*.

```
pub struct Set<T> {
  set: FxHashSet<T>, // FxHashSet a pour avantage d'être déterministe
}
```

On stocke dans une *HashMap* les registres appartenant à cet ensemble.

Nous allons apporter des améliorations à cette structure dans le but de diminuer le temps passé dans l'allocation de registres.

Comme on vient de le voir, l'implémentation actuelle du *Set* utilise simplement une *HashMap*. On stocke chaque registre appartenant à l'ensemble. Si l'ensemble comporte 64 registres, on utilise 256 octets. On peut améliorer l'utilisation de la mémoire avec une structure appelée *BitSet*. Étant donnée que l'appartenance est une information binaire, nous n'avons besoin que d'un bit par registre. Au lieu d'utiliser 32 bits par registre on peut utiliser un seul bit. La plupart du temps, on n'utilisera pas plus de 64 bits.

Avec cette structure nous souhaitons manipuler des ensembles de registres (faire des intersections, unions, différences). Ces manipulations sont parallélisable grâce à l'arithmétique booléenne qui est directement supportée par le processeur.

Des implémentations de *BitSet* sont déjà disponibles sur crates.io [12] (l'hébergeur principal de paquets de *Cargo* [13], le gestionnaire de paquets du langage *Rust*) mais nous ne pouvons nous reposer sur des dépendances extérieures car il s'agit d'une pièce logicielle critique qui intégrera d'une part le logiciel *Firefox* et d'autre part dans le futur le compilateur *Rust* en tant que *backend* pour obtenir des temps de compilation plus rapides.

3.1 Première solution et analyse

Dans un premier temps nous avions voulu écrire une structure générique *BitSet*. L'objectif de la généricité est de pouvoir utiliser la structure avec des types de données différents. On souhaite principalement l'utiliser pour les registres mais potentiellement pour d'autres données. Il est compliqué de faire ainsi car les registres dépendent de leur type et de leur index. En C++ il existe la spécialisation de génériques qui permet d'écrire un code générique et des morceaux spécifiques pour des exceptions. Cela n'est actuellement pas encore possible en *Rust*. Il faut donc créer 2 structures similaires mais différentes. Enfin, nous nous sommes rendu compte qu'il est plus simple d'écrire une structure spécialisée pour les registres et de la généraliser par la suite.

```
pub struct RegBitSet {
    bits: Vec<u64>,
}
```

Dans notre nouvelle structure de données, nous avons besoin de deux informations. L'index du registre et l'appartenance du registre à l'ensemble à laquelle on l'associe.

Voici à quoi ressemble l'insertion d'un registre :

```
    const BLOCK_SIZE: usize = 64;

    fn insert(&mut self, item: Reg) {
        if !self.contains(item) {
            let bits_index = item.get_index() / BLOCK_SIZE;
            let offset = item.get_index() % BLOCK_SIZE;
```

12. https ://crates.io/
13. https ://doc.rust-lang.org/cargo/

```
            if bits_index >= self.bits.len() {
                self.bits.resize(bits_index + 1, 0);
            }
            self.bits[bits_index] |= 1 << offset;
        }
    }
```

Les principes sont les mêmes pour retirer un registre ou vérifier son appartenance (on utilise les opérateurs de manipulation de bit que nous fournit le langage).

3.2 Améliorations et corrections de notre structure

Jusqu'ici tout va bien, si on jette un coup d'œil aux signatures de nos autres méthodes, elles retournent des booléens, entiers ou *RegBitSet* :

```
pub trait RegSet {
    fn empty() -> Self;
    fn unit(item: Reg) -> Self;
    fn two(item1: Reg, item2: Reg) -> Self;
    fn card(&self) -> usize;
    fn insert(&mut self, item: Reg);
    fn delete(&mut self, item: Reg);
    fn is_empty(&self) -> bool;
    fn contains(&self, item: Reg) -> bool;
    fn intersect(&mut self, other: &Self);
    fn union(&mut self, other: &Self);
    fn remove(&mut self, other: &Self);
    fn intersects(&self, other: &Self) -> bool;
    fn is_subset_of(&self, other: &Self) -> bool;
    fn equals(&self, other: &Self) -> bool;
}
```

Mais nous avions oublié l'itérateur qui retourne une référence sur un *Reg*. Malheureusement l'allocateur de registres fait une utilisation intensive des itérateurs, on ne peut donc simplement changer son comportement.

L'idéal serait de pouvoir construire le *Reg* qu'on souhaite retourner à la volée mais étant donné qu'on ne garde que l'index d'un registre il nous manque des informations.

Cela signifie qu'on doit stocker ces registres.

```
pub struct RegBitSet {
    bits: Vec<u64>,
    data: Vec<Reg>,
}
```

```
    fn insert(&mut self, item: Reg) {
        if !self.contains(item) {
            let bits_index = item.get_index() / BLOCK_SIZE;
            let offset = item.get_index() % BLOCK_SIZE;

            if bits_index >= self.bits.len() {
                self.bits.resize(bits_index + 1, 0);
            }
            self.bits[bits_index] |= 1 << offset;
```

```
            self.data.push(item);
        }
    }
```

Nous avons également commis une autre une erreur en pensant que les indexes des registres suffisaient pour former un identifiant unique. Par exemple, le registre r1 (registre réel 1 encodé comme 0x80000001) et le registre v1 (registre virtuel encodé comme : 0x00000001) ont le même index 1.

Cela nous a demandé de modifier notre structure afin de séparer les registres réels et virtuels. Pour cela, lorsqu'on manipule des registres virtuels on utilise un décalage égal au nombre de registres réels.

```
const BLOCK_SIZE: usize = 64;
const NUMBER_REAL_REG: usize = 64;

fn insert(&mut self, item: Reg) {
    if !self.contains(item) {
        let reg_index = RegBitSet::get_reg_index(item);
        let bits_index = reg_index / BLOCK_SIZE;
        let offset = reg_index % BLOCK_SIZE;

        if bits_index >= self.bits.len() {
            self.bits.resize(bits_index + 1, 0);
        }
        self.bits[bits_index] |= 1 << offset;

        if reg_index >= self.data.len() {
            self.data.resize(reg_index + 1, None);
        }
        self.data[reg_index] = item;
    }
}

fn get_reg_index(item: Reg) -> usize {
    if item.is_real() {
        item.get_index()
    } else {
        NUMBER_REAL_REG + item.get_index()
    }
}
```

3.3 Résultats

Il reste pas mal d'optimisations possibles et nous en avons reperées quelques unes avec le logiciel de profiling *Callgrind*. On peut dès à présent faire quelques benchmarks pour aprécier les améliorations (ici bz2) :

```
/home/na/wasmtime/target/release/clif-util wasm --target arm64
-vdT /home/na/cranelift-new-backend-tests/bz2/bz2.wasm
```

On passe de 0.757 ms dans l'allocation de registres à 0.572 ms. Soit un gain de 24,4%.

4 Analyse de nos contributions : mieux contribuer à un projet open source

4.1 Organisation

Le projet s'est déroulé du mois de janvier jusqu'au mois de juin. Nous étions encadré par Laure Gonnord et Matthieu Moy. Benjamin Bouvier nous accompagnait sur le plan technique à distance.

La communication s'est faite par mail et via la messagerie instantanée de Mozilla (Matrix/Riot) pour du temps réel. Nous avions des réunions une fois par semaine (en visioconférence avec les personnes ne pouvant se rendre sur place ou pendant le confinement).

Les moyens mis en place sont très efficaces mais non pleinement exploités. Nous utilisions souvent la messagerie instantanée pour poser des questions sur des points techniques. Les emails étaient utilisés pour communiquer avec nos encadrants sur les questions organisationnelles. En revanche nous communiquions entre nous par des moyens privés, laissant nos encadrant à l'écart de la possibilité d'apporter tout soutien. Nous avons travaillé sur des tâches différentes en parallèle, il aurait été optimal d'alterner nos tâches régulièrement tout en se conseillant mutuellement.

4.2 Apprendre et s'améliorer

Le choix de ce projet est poussé par l'envie de nous améliorer dans la contribution aux projets open-sources. En effet nous avions auparavant déjà réalisé quelques contributions pour Mozilla. L'un des objectifs est donc d'appliquer les bonnes pratiques de code et d'intégration dans les outils.

4.3 Mieux comprendre l'écosystème de *Rust*

Ce projet est également motivé par l'apprentissage du langage *Rust*. Ce langage a la particularité de demander plus de temps d'apprentissage et plus de pratique.

Sur certains points il apporte des concepts intéressants. Le gestionnaire de dépendance *Cargo* apporte différents outils. Nous avons eu l'occasion d'utiliser "cargo clippy" qui apporte des indicateurs au niveau du code comme les variables inutilisées, les portions de codes qui pourraient être plus idiomatiques, des astuces etc. Nous avons "cargo fmt" qui formate les fichiers du projet.

On utilise "cargo test" pour lancer les tests, "cargo check" si l'on souhaite rechercher les erreurs sans compiler le projet. Nous avons exploité qu'une petite partie de ce dont *Cargo* est capable mais ça nous permet déjà de comprendre sa puissance.

4.4 Découvrir les acteurs principaux

Contribuer à un projet open-source nécessite d'entrer en contact avec les principaux acteurs dudit projet, dans notre cas M. Bouvier. En effet, découvrir une base de code existante n'est pas simple, on saisit donc l'avantage d'avoir un contact travaillant sur le projet pour nous orienter et répondre à nos questions.

4.5 Méthodes d'analyse d'un bug

Pour nous échauffer nous avons travaillé sur un bug simple. Lorsqu'un bug est découvert une issue est crée sur GitHub. Une issue est un message qui initie une discussion concernant une modification à apporter au projet.

Dans notre cas c'était le suivant :

```
cranelift-reader should have an option to error on pathological cases #951

bjorn3:
The following function will attempt to allocate 477777777 ebbs.

function %a(){
```

```
ebb477777777:
}
```

Pour résoudre ce problème on doit effectuer plusieurs étapes. La première consiste à reproduire l'erreur.

On créer un fichier au format .clif et on met le code dedans puis on exécute :

```
./clif-util compile bug.clif --target arm64
```

On s'assure qu'on a bien le comportement attendu (le programme ne s'arrête pas ou finit par renvoyer une erreur). On fait quelques tests pour s'assurer qu'il s'agit bien d'un comportement anormal (par exemple allouer un nombre faibles d'ebbs).

L'étape suivante consiste à écrire un test qui échoue si ce bug se produit. Nous allons simplement considérer qu'on devrait pas avoir plus d'un certain nombre d'ebbs par fonction :

```
    #[test]
    fn ebbs_number() {
        let ParseError {
            location,
            message,
            is_warning,
        } = Parser::new(
            "function %a() {
                block100001:",
        )
        .parse_function(None)
        .unwrap_err();

        assert_eq!(location.line_number, 2);
        assert_eq!(message, "too many blocks");
        assert!(!is_warning);
    }
```

On s'assure bien que ce test échoue. Maintenant il faut modifier le code pour que le test passe. Donc quelque part on fait les modifications nécessaires :

```
static MAX_BLOCKS_IN_A_FUNCTION: u32 = 100000;

// ...

if block_num.as_u32() > MAX_BLOCKS_IN_A_FUNCTION {
    return Err(self.error("too many blocks"));
}
// ...
```

On créé une pull request (figure 1). Il s'agit d'une demande d'intégration de nos modification au projet. Dans cette pull request on documente nos décisions en apportant un maximum d'explications. Notre commit doit être atomique car il existe une commande (git blame) qui nous permet de lire le message de commit pour chaque ligne de code du projet. C'est désagréable d'avoir message faisant référence à un autre commit que l'utilisateur ne peut pas avoir directement sous les yeux. On conseille donc d'apporter un maximum d'information dans le message du commit (ne pas utiliser git commit -m).

FIGURE 1 – *Message de commit d'une pull request sur GitHub.*

Il se peut qu'on ait des reviews. Il s'agit de (précieux) commentaires d'autres contributeurs. Dans un cas idéal, on effectue des modifications, on demande des reviews, on en reçoit, on modifie et on répète ce processus jusqu'à avoir une pull request acceptée par la communauté.

À ce moment là on peut vouloir réécrire l'historique des commits afin de laisser uniquement l'essentiel. Pour cela on utilise l'outil git squash (il permet de fusionner les commits, les renommer, les réécrire etc). C'est un outil très utile pour fournir un travail propre.

4.6 Repérage de bugs

Lors de l'implémentation de notre structure de données nous devions remplacer l'ancienne. À ce moment là nous avons découvert qu'elle avait des bugs.

Une des raisons vient du fait que nous avons écrit notre base de code, et une fois satisfait de ce code nous écrivions des tests unitaires.

Une meilleure pratique aurait été d'adopter la méthodologie test driven development. Pour chaque méthode, on écrit tous nos tests en on s'assure qu'ils échouent. Ensuite on écrit le code nécessaire pour faire passer ces tests. On peut ensuite faire de refactoring mais on ne fait pas plusieurs étapes en même temps.

Pour trouver des bugs, on peut utiliser des outils de coverage. Ces outils nous permettent de trouver facilement les portions de code non couvertes par des tests unitaires. Nous avons essayé quelques outils, l'expérience n'a pas été concluante faute de perseverance et peut-être de documentation (on rappelle que l'écosystème de *Rust* est plutôt jeune par rapport à ses concurrents).

On peut également faire appel à un debugger. On conseille tout de même d'essayer les méthodes cités precedemment avant car la recherche de bugs avec ce type d'outil devient fastidieuse en travaillant sur de grosses bases de code.

Nous avons développé une méthodologie pour bien utiliser cet outil. D'abord on récupère une version de notre code où le bug n'a pas encore été introduit. On cherche ensuite un commit postérieur où le bug est introduit (on peut utiliser git bisect pour cela).

On execute les 2 instances de notre programme en parallèle jusqu'à trouver des différence de comportement inattendu (dans notre cas, notre structure de donnée devait faire la même chose que celle qu'elle remplaçait c'était donc plus simple). À partir de là on affine les recherches, on regarde les différentes méthodes impliquées, on essaie de comprendre comment les données sont calculées en espérant trouver le bug en question.

Une fois que c'est fait, on isole et reproduit le bug dans un test unitaire. Ce test doit échouer. Enfin, on fait les modifications nécessaires pour faire passer ce test.

4.7 Autres difficultés

Il est nécessaire de souligner que Valentin a rencontré des problèmes de santé cette année. Le projet ne s'est donc pas déroulé comme nous le pensions. Il s'agit d'une contrainte qu'on peut rencontrer dans la vie ou dans n'importe quel projet et nous aurions dû, rétrospectivement, mieux la prendre en compte. Son travail n'est pas visible mais pourtant réel. Sa santé ne lui permettant pas d'adopter un rythme régulier, il a dû avancer à sa propre vitesse. Ainsi il a souvent aidé et accompagné Nabil quand il le fallait en apportant de précieuses réflexions lorsque le projet n'avançait pas. De son côté il n'a cessé d'apprendre (en faisant des tests, en apprenant le langage *Rust*).

La période de confinement a peut-être eu un impacte psychologique pour certains mais sur le plan organisationnel nous n'avons pas eu à aménager de changements.

5 Conclusion

Nous avons profité de ce projet d'orientation pour approfondir nos connaissances en compilation et comprendre les enjeux autour d'un nouveau compilateur (*Cranelift*) comportant des éléments expérimentaux. Nous y avons découvert un langage modern et élégant : *Rust*. Ce langage nous apporte des garanties au niveau de la sécurité de la mémoire avec des performances proches du C. Il demande en revanche de comprendre des concepts tels que les lifetimes ou le borrow checker. Enfin, nous avons découvert de nombreuses bonnes pratique de développement qui nous seront utiles à l'avenir.

Références

[1] Laure Gonnord. https ://compil-lyon.gitlabpages.inria.fr/compil-lyon/mif081920_lyon1/mif08_all_slides4p.pdf. Jan 2020.

[2] Fernando Pereira. Register allocation, https ://homepages.dcc.ufmg.br/ fernando/classes/dcc888/ementa/slides/registerallocation.pdf.

[3] Fernando Pereira. A Survey on Register Allocation, http ://compilers.cs.ucla.edu/fernando/publications/drafts/survey.pdf. Oct 2008.