GRAPPA — Rapport final : Implémentation des E-Graph en OCaml

Hugo Guerrier

Février 2022

Abstract

Ce document est un rapport résumant le travail qui a été réalisé dans le cadre du projet de GRAPA, il a pour but de donner l'analyse des problématiques de ce travail et d'en résumer les solutions. En complément de ce document, vous pouvez retrouver le code du projet à cette adresse : CAML_EGG. Ce rapport suppose également que vous possédez déjà quelques notions sur les E-Graph

1 Introduction

1.1 EGG, la source d'inspiration

Lors de la première partie de l'UE GRAPA, nous avons étudié des articles de recherche portants principalement sur les langages de programmation, celui que j'ai décidé d'étudier plus en détail est un article portant sur une méthode d'optimisation à la compilation reposant sur les "E-Graph" (ou graphes d'équivalence) et la saturation d'égalités en y ajoutant un dimension d'optimisation et d'analyse statique. Vous pouvez retrouver cet article ici : egg: Fast and Extensible Equality Saturation.

1.2 Le projet de fin d'UE

Après la lecture détaillée de cet article, nous avons décidé avec mon professeur référent Monsieur PESCHANSKI que le projet final de GRAPA serait de ré-implémenter EGG dans un autre langage, cela semblait intéressant d'un point de vue pédagogique et technique. Le fait d'implémenter les algorithmes et les structures de données qu'un article décrit permet de bien saisir les problématiques et les enjeux de ceux-ci. De plus, le faire dans un autre langage que celui proposé par l'article constitue un défi supplémentaire.

Il a donc été décidé que mon projet serait d'implémenter les structures de données et les algorithmes de EGG en OCaml, un langage a dominante fonctionnelle ayant aussi des aspects impératif et objet. Ce langage a été choisi car il encourage fortement à utiliser des structures et des opérations tirés de la programmation

fonctionnelle (pattern matching, foncteurs...), tout en permettant la manipulation de mémoire et d'objets relativement facilement.

Nous allons maintenant voir et décrire toutes les structures de données et opérations de manipulation que j'ai implémentés durant ce projet.

2 La structure globale des "E-Graph" en OCaml

Afin de représenter un "E-graph", il me fallait déjà comprendre de quoi il était composé et comment le modéliser dans la mémoire. Pour ce faire, je me suis grandement inspiré de l'article dont est tiré ce projet (egg: Fast and Extensible Equality Saturation) qui propose une structure assez simple à mettre en oeuvre dans l'extrait disponible sur la Figure 1.

Definition 2.1 (Definition of an E-Graph). Given the definitions and syntax in Figure 1, an *e-graph* is a tuple (U, M, H) where:

- A union-find data structure [Tarjan 1975] U stores an equivalence relation (denoted with $\equiv_{\text{id}}$) over e-class ids.
- The *e-class map* M maps e-class ids to e-classes. All equivalent e-class ids map to the same e-class, i.e., $a \equiv_{\text{id}} b$ iff $M[a]$ is the same set as $M[b]$. An e-class id a is said to *refer to* the e-class $M[\texttt{find}(a)]$.
- The *hashcons*[3] H is a map from e-nodes to e-class ids.

Figure 1: Extrait de l'article sur EGG définissant un E-Graph

Ainsi, en suivant ce modèle de représentation, le type représentant un "E-Graph" dans "CAML_EGG" est composé comme suit :

```
type e_graph = {
    id_cpt: int;
    map: (e_class ref) EClassIdMap.t;
    hc: (e_class ref) Hashcons.t;
}
```

En sachant que "EClassIdMap" et "Hashcons" sont des modules crées à partir du foncteur "Map" de la bibliothèque standard d'OCaml. Il permettent tous les deux d'avoir des structures de données basées sur les "Map" (ou "Table") pour associer des clés uniques à des valeurs.

Vous pouvez remarquer que l'on retrouve dans ce type les "M" et "H" de la définition donnée par l'article, cependant la structure "union-find" n'y transparaît pas. En réalité, grâce à la bibliothèque UnionFind, j'ai pu simplement ajouter la structure nécessaire au travers des identifiants de "E-Class" comme vous pouvez le voir dans le type suivant :

```
type e_class_id = int UnionFind.elem
```

Un fois tous ces types crées, j'avais la possibilité de stocker toutes les informations nécessaires pour mémoriser un graphe d'équivalence, il ne me restait donc plus qu'à créer une fonction permettant d'en instancier un nouveau dans la mémoire et de retourner son adresse :

```
let create_e_graph () : e_graph ref =
    ref {id_cpt = 0 ; map = EClassIdMap.empty ; hc = Hashcons.empty}
```

Comme vous pouvez également le constater dans les types définis ci-dessus, il est fait mention plusieurs fois des types inconnus pour l'instant ("e_class") il s'agit là de structures de données utiles pour stocker les informations relatives aux "E-Graph" et nécessaires à son bon fonctionnement. Nous allons détailler par la suite ces structures ainsi leur utilité.

3 Les "E-Class"

3.1 Définition de la structure et des opérations

La structure principale dans un "E-Graph" est ce qu'on appelle une "E-Class" (ou classe d'équivalence), le graphe peut en contenir autant que nécessaire afin de représenter un programme ou un terme. Le rôle d'une classe d'équivalence est de représenter des termes équivalents les uns avec les autres, vous pouvez voir un exemple de congruence "d'E-Class" sur la Figure 2 : Dans cet exemple, deux expressions sont représentées : "2+2" et "2*2" et étant données les règles arithmétiques, ces deux expressions sont équivalentes (2+2 = 4 — 2*2 = 4), il est donc possible d'établir un lien de congruence entre elles.

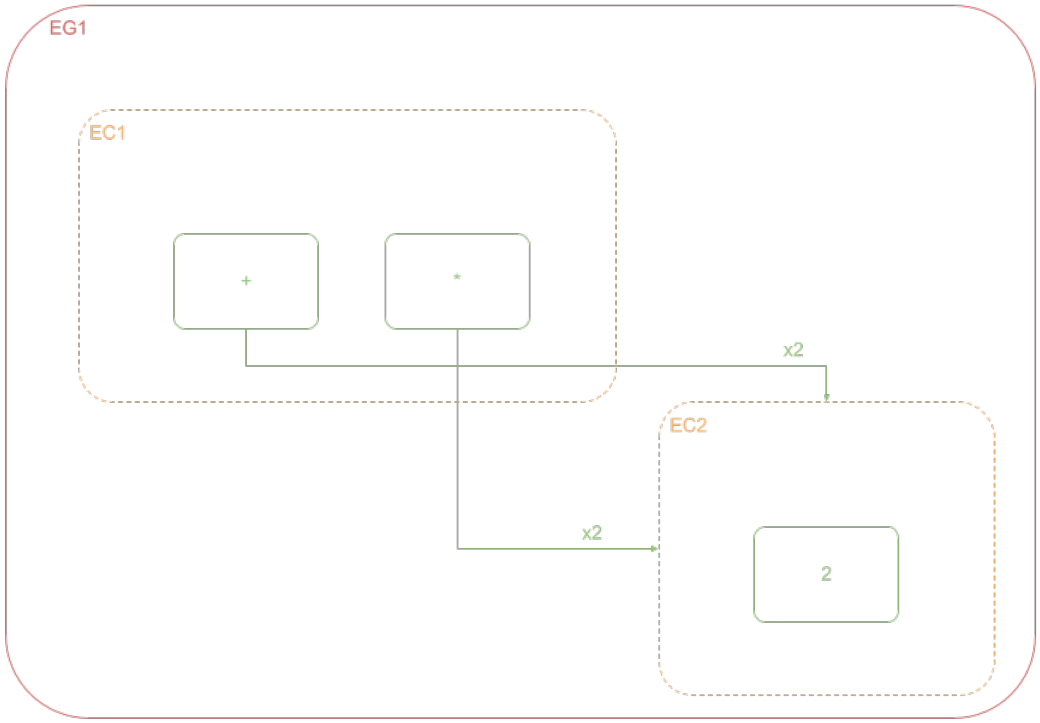

Figure 2: Exemple de "E-Graph" représentant la congruence entre "2+2" et "2*2", en orange les "E-Class"

Afin de modéliser les "E-Class", j'ai implémenté un type en OCaml permettant de retenir les informations nécessaires :

```
type e_class = {
    id: e_class_id;
    e_nodes: (e_node ref) list;
}
```

Une "E-Class" est définie par son identifiant (faisant lui même partie d'une structure union-find) et une liste de références vers des "E-Node" représentant les termes équivalents dans la classe.

Une fois ce type créé, je devais implémenter des fonctions permettant de créer et de manipuler les classes d'équivalences, selon l'article, les opérations nécessaires pour les classes d'équivalences sont :

- L'opération "merge ec1 ec2" permettant de signifier que deux classes d'équivalences *ec1* et *ec2* sont effectivement équivalentes dans le "E-Graph"
- L'opération "find ec" permettant de retrouver la "E-Class" la plus "haute" dans la structure union-find par rapport à *ec*, autrement dit à retrouver l'équivalent de *ec*

Ces opérations ont été programmées dans le fichier "lib/caml_egg.ml" du projet si vous voulez plus de détails sur leur implémentation.

3.2 Un exemple

Voici un exemple de création et de fusion de classes d'équivalences :

Dans un premier temps il faut créer un "E-Graph" et y ajouter deux "E-Class".

```
let eg1 = create_e_graph () in
let ec1 = create_e_class eg1 in
let ec2 = create_e_class eg1 in
...
```

Sur la Figure 3 vous pouvez observer l'état d'un graphe d'équivalence après l'exécution de ce code :

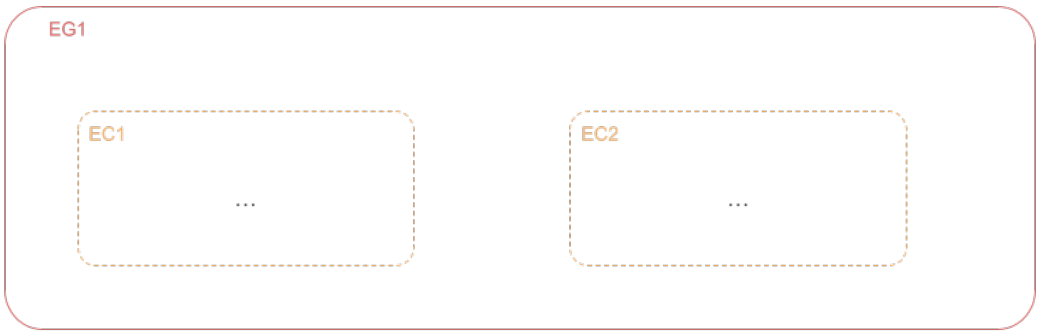

Figure 3: Création de deux "E-Class" dans un "E-Graph"

Une fois les classes crées, il est possible de donner une relation de congruence entre-elles de la manière suivante :

```
...
let _ = e_class_union !ec1.id !ec2.id in
...
```

Vous pouvez voir l'état de notre "E-graph" après la création de la congruence sur la Figure 4.

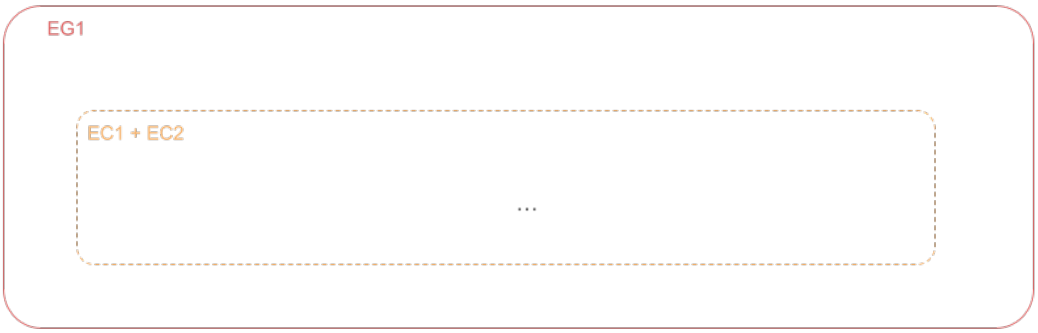

Figure 4: Établissement d'un lien de congruence entre les deux "E-Class"

Un fois cette relation donnée, il est maintenant possible de récupérer l'équivalent de *ec2* (à savoir *ec1*) via l'opération "find" :

```
e_class_canon !ec2.id (* -> !ec1.id *)
```

Comme vous pouvez le voir tout au long de l'exemple, toutes les opérations de manipulation des "E-Class" s'effectuent avec leurs identifiants, cela permet de simplifier le traitement et il est très aisé de retrouver une classe à l'aide de son identifiant dans le "M" de la structure "E-Graph".

Vous avez également pu remarquer dans la définition des "E-Class" que l'on utilisait des "E-Node", c'est un type permettant de représenter une expression dans le "E-Graph", nous allons dans la suite détailler cet élément.

4 Les "E-Node"

4.1 Définition de la structure de données et des opérations

Pour stocker les applications de fonctions dans un "E-Graph", nous avons besoin d'une autre structure que les "E-Class", c'est pour cela que nous faisons appel aux "E-Node". C'est une structure stockant simplement le nom de la "fonction" ainsi que ses arguments (qui sont représentés par des "E-Class"). Pour reprendre l'exemple précédent, dans la Figure 5, on voit qu'un "E-Node" peut être la fonction $+$ qui prend deux arguments, mais on voit aussi qu'un noeud peut représenter une constante (qui est en soit une fonction ne prenant aucun argument) comme *2*.

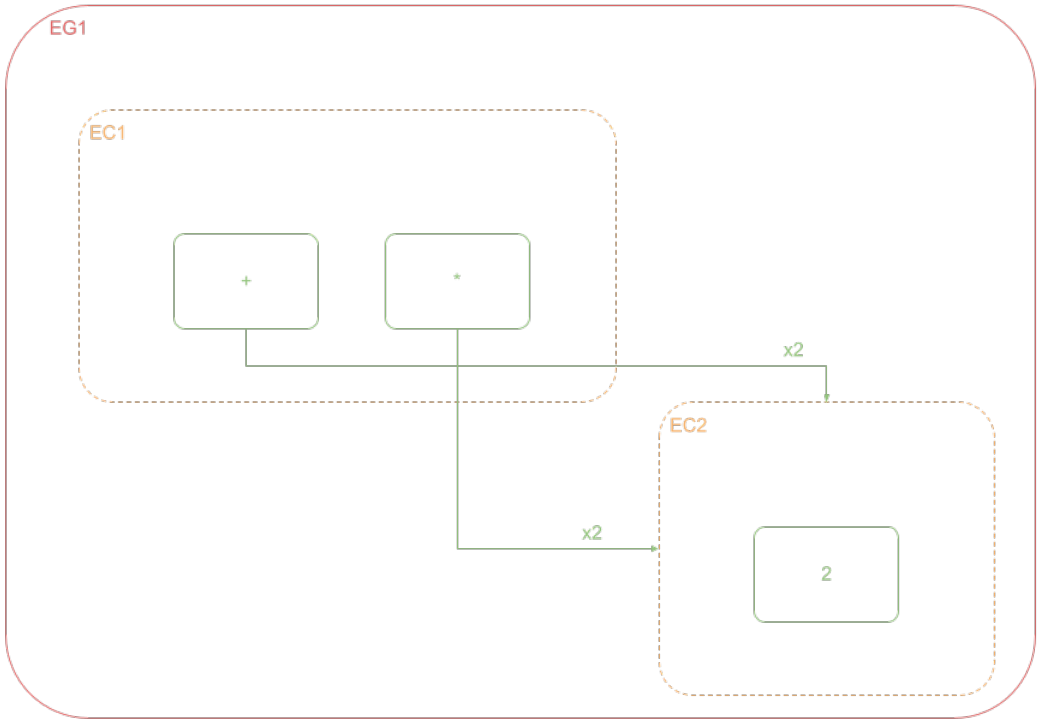

Figure 5: Exemple d'E-Graph représentant la congruence entre "2+2" et "2*2", en vert les "E-Node"

Voici donc le type définissant dans "CAML_EGG" les noeuds des graphes d'équivalences :

```
type e_node = {
    fn: string;
    children: e_class_id list;
}
```

Un "E-Node" est défini par une chaîne de caractère représentant le symbole de la fonction ainsi qu'une liste d'identifiants de "E-Class" permettant de retrouver les arguments de cette fonction.

Afin de manipuler les "E-Node", je devais ensuite créer des opérations permettant de créer et de gérer les noeuds dans un graphe. Encore une fois selon l'article de référence, il faut les opérations :

- L'opération "canon en" permettant d'obtenir la forme canonique d'un "E-Node" voulu en canonisant tous ses "arguments"
- L'opération "eq en1 en2" permettant de tester l'équivalence de deux "E-Node" basée sur la "E-Class" à laquelle ils appartiennent
- L'opération "add en ec" pour ajouter un argument à un noeud

Toutes les implémentations sont disponibles dans le fichier "lib/caml_egg.ml" pour plus de détails sur ces fonctions.

4.2 Un exemple

Voici un exemple d'utilisation de toutes les structures vus jusqu'ici pour aboutir au "E-Graph" fil rouge de ce rapport (Figure 5).

On commence par créer le "E-Graph" et les trois "E-Class" amenées à contenir les noeuds :

```
let eg = create_e_graph () in
let add_ec = create_e_class eg in
let mul_ec = create_e_class eg in
let two_ec = create_e_class eg in
...
```

Après l'exécution de ce code, on obtient un graphe comme sur la Figure 6.

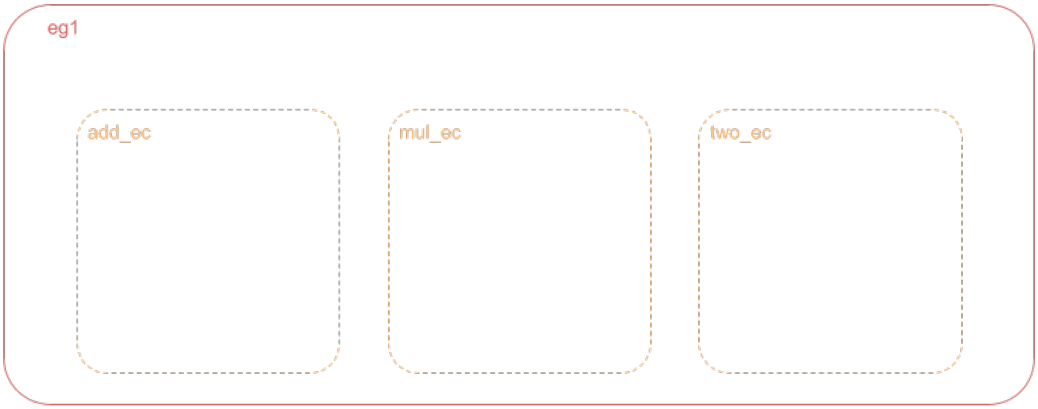

Figure 6: Exemple de la construction d'un "E-Graph" - Étape 1

Une fois les trois classes crées, il faut ajouter dans chaque classe le noeud correspondant au terme que l'on souhaite ajouter. On y arrive par le biais du code suivant :

```
...
let add_en = create_e_node eg add_ec "+" in
let mul_en = create_e_node eg mul_ec "*" in
let two_en = create_e_node eg two_ec "2" in
...
```

Vous pouvez voir l'état visuel du graphe sur la Figure 7 après l'exéctuion de cette portion de code.

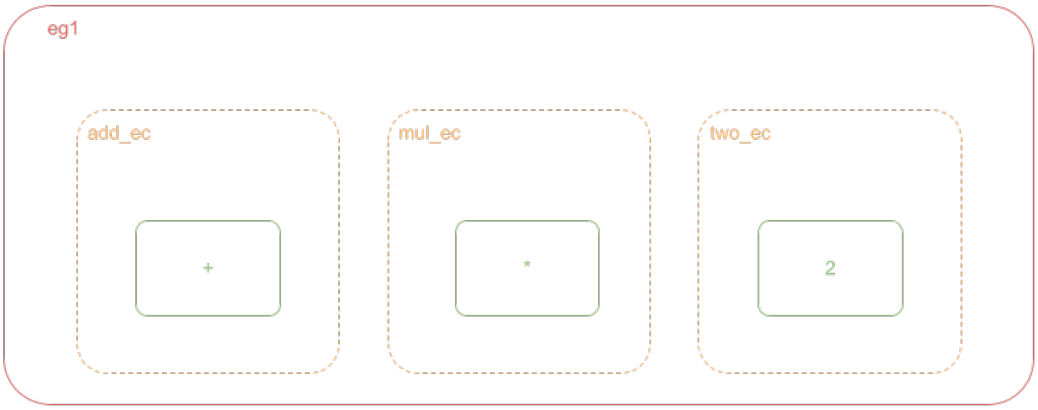

Figure 7: Exemple de la construction d'un "E-Graph" - Étape 2

Ensuite, il faut créer les lien de fonction à argument entre les classes et les noeuds au sein de notre graphe à l'aide de cette portion de code :

```
...
e_node_add_child eg add_en two_ec ;
e_node_add_child eg add_en two_ec ;
e_node_add_child eg mul_en two_ec ;
e_node_add_child eg mul_en two_ec ;
...
```

Vous pouvez observer sur la Figure 8 l'état visuel du graphe une fois les relations crées.

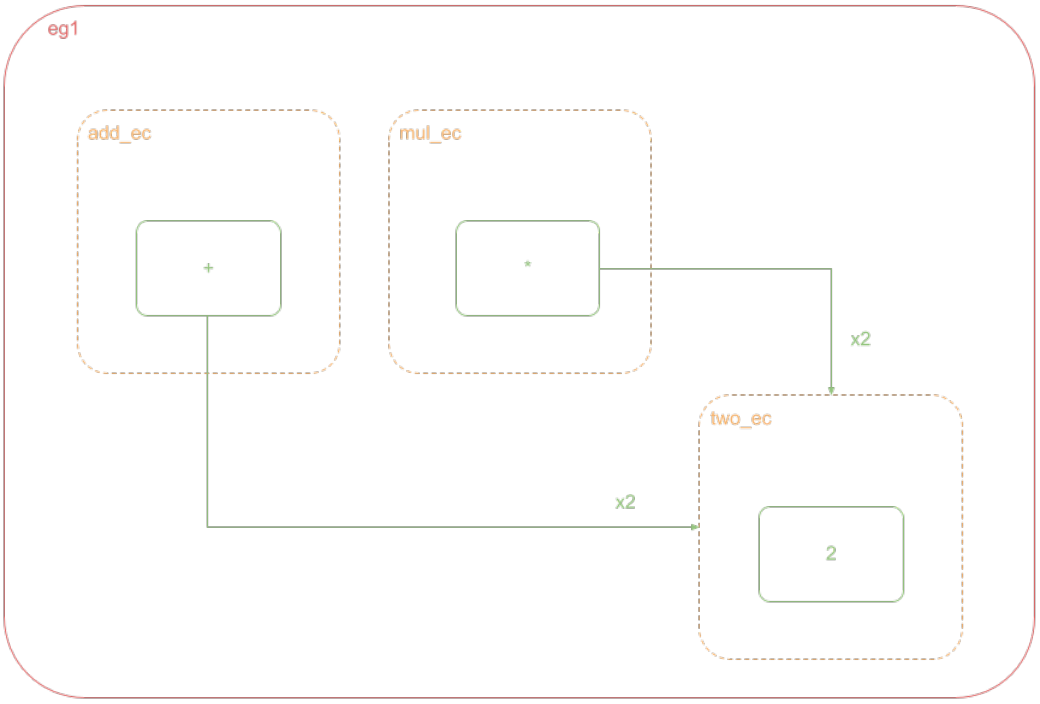

Figure 8: Exemple de la construction d'un "E-Graph" - Étape 3

Enfin, pour obtenir le graphe final, il nous faut créer la relation de congruence entre les deux classe "add" et "mul" :

```
...
let _ = e_class_union !add_ec.id !mul_ec.id in
...
```

Comme vous pouvez le voir sur la Figure 9 l'état visuel de notre graphe correspond bien à celui de notre exemple fil rouge.

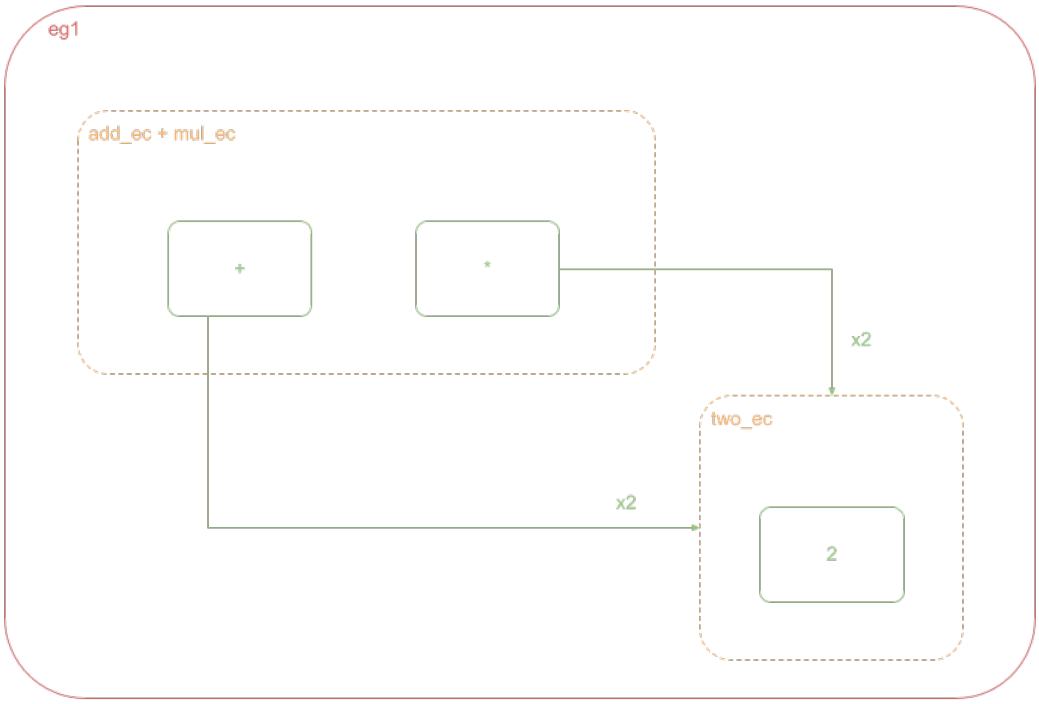

Figure 9: Exemple de la construction d'un "E-Graph" - Étape 4

Grâce à toutes les notions introduites jusqu'ici, il est désormais possible de construire n'importe quel "E-Graph" et de le modifier de manière simple.

5 Les algorithmes de "rebuilding" et de "E-Class analysis"

Dans l'article référence, sont introduits deux nouveaux concepts servants à construire, optimiser et analyser les "E-Graph" : le "rebuilding" et le "E-Class analysis". Ces deux méthodes permettent respectivement de renforcer les invariants des "E-Graph" quand souhaité et de procéder à une passe d'analyse contextuelle sur toute la structure du "E-Graph" permettant d'effectuer des optimisations non-statiques. Elle sont le coeur de l'article ainsi que sa principale contribution et leurs algorithmes ainsi que leur structures de données sont explicités dans ce dernier, cependant, par manque de temps et de connaissances je n'ai pas pu les implémenter dans mon projet. Cela aurait pourtant été très intéressant à développer, mais surtout à tester sur les même exemples que ceux utilisés dans l'article sur EGG afin de comparer avec l'implémentation en Rust, malheureusement ce ne sera pas possible au terme de ce projet.

6 Conclusion

Durant ce projet, j'ai pu définir les structures de données et les opérations nécessaires à la construction et à l'utilisation des "E-Graph" en OCaml. Ce ne fut pas une tâche aisée car les structures utilisées et le langage OCaml ne m'étaient pas du tout familiers avant ce projet, j'ai donc appris beaucoup de choses, entre autre :

- La création et la gestion de projet OCaml avec "dune
- L'utilisation des modules et des foncteurs OCaml
- L'implémentation de la structure union-find
- L'utilisation de hashcons

Néanmoins, j'aurais aimé avoir plus de temps à consacrer à ce projet et aller plus loin dans la recréation de EGG et même pouvoir comparer les deux bibliothèques à terme.

Ce projet fut une très bonne expérience car très formateur techniquement (par l'apprentissage approfondi d'OCaml et de ses modules) mais aussi académiquement (au travers de la lecture d'article et l'étude d'algorithmes).

Merci de votre lecture.