Stage : Synthèse de fonctions C à partir de définitions axiomatiques

Jeremy Damour[1], Catherine Dubois[2], and Julien Signoles[3]

[1]Université Paris Cité
[2]ENSIIE, Samovar équipe Méthodes
[3]Université Paris-Saclay, CEA, List

22 mars 2022 - 26 août 2022

1 Synthèse

Contexte général Un code est correct, s'il ne contient pas d'erreur à l'exécution et respecte bien les spécifications qui lui sont associées. Une des solutions existantes pour vérifier qu'un code est correcte, vis-à-vis de ces spécifications, peut être la vérification à l'exécution (*rutime assertion checking*). Les spécifications peuvent être écrites dans un langage formel, qui offrent la possibilité d'écrire des définitions non explicite de fonction. Ces définitions peuvent être axiomatique ou inductive et ne sont pas directement exécutables.
Les outils de vérification à l'éxécution, sachant que ces définitions ne sont pas directement exécutables, utilisent souvent un langage ne bénéficiant pas de ces constructions. Rendre ces définitions exécutables permettrait donc à un outil vérification à l'exécution de pouvoir vérifié des spécifications avec des définitions non explicites.

Problème étudié Au CEA LIST, le Laboratoire de Sureté Logicielle (LSL) développe l'outil Frama-C étant une plateforme logicielle facilitant le développement d'outils d'analyse de programme C. Frama-C offre la possibilité d'annoter un programme C avec des spécifications formelles écrites dans le langage ACSL. Le greffon E-ACSL de Frama-C permet de faire de la vérification à l'exécution. Pour ce faire, E-ACSL génère à partir d'un code C annoté par des spécifications dites exécutables un nouveau code C vérifiant ces spécifications. Le langage (aussi appelé) E-ACSL contient donc la sous-classe des spécifications exécutables du langage de spécification ACSL. Le but de mon stage est de rendre exécutables des définitions non explicites. Permettant au greffon E-ACSL d'élargir les constructions qu'il traite du langage ACSL et de mieux se combiner avec d'autres analyses.
Des outils permettant d'extraire du code exécutable à partir de relations inductives et utilisant des types inductifs existent. Mon approche permet quant à elle permet de rendre exécutable des définitions exécutables à partir des définitions inductives mais aussi axiomatique proposées par le langage de spécification ACSL en utilisant des types primitifs C et les prédicats mémoire ACSL.

Contribution proposée ACSL propose trois types de définitions non explicites : les prédicats inductifs, les prédicats et les fonctions axiomatisées. Le but est d'extraire une définition exécutable de ces définitions non explicites. Les définitions non explicites du langage ACSL ont une très grande expressivité, il faut donc dans un premier temps définir la sous-classe de ces définitions qu'il est possible de traduire. Pour ne pas avoir plusieurs traductions, les prédicats et fonction axiomatisée sont transformés en prédicat inductif. Une normalisation sur les inductifs est appliquée, pour faciliter l'extraction. L'extraction peut alors être effectuée et produit une fonction ou un prédicat logique E-ACSL, permettant de bénéficier des différentes constructions déjà présentes dans E-ACSL lors de sa traduction vers du code C.

Arguments en faveur de sa validité L'extraction produit des définitions qu'un développeur aurait écrites manuellement en regardant la spécification. Cette extraction se fonde essentiellement sur celle de P-N.Tollite [17] qui fait une extraction des définitions inductive de Coq. Il a fait une preuve de correction partielle son extraction, qui conforte en partie la correction de l'implémentation. Par ailleurs, la solution a été implantée en dans le code du greffon E-ACSL de Frama-C écrit en OCaml. Ce qui m'a permis d'inspecter manuellement la sortie de l'extraction sur plusieurs exemples et ajouter ces exemples à la banque de tests d'E-ACSL. Ces tests permettent notamment de vérifié l'absence d'incorrection du code C généré par E-ACSL, grâce au greffon EVA étant l'analyseur de valeur par interprétation abstraite de Frama-C. La traduction finale dépend donc de la correction de la traduction des fonctions logiques de E-ACSL vers du code C. Cette traduction est limitée par certaines constructions du langage non présent. Comme la gestion des erreurs dans les annotations ACSL, ce qui limite dans la traduction des fonctions qui ne peuvent pas effectuer de retour en arrière.

Bilan et perspectives Les travaux précédents nécessitaient d'utiliser des types sommes, pour effectuer une traduction. Mon extraction permet d'extraire des définitions pures (potentiellement récursives) en utilisant des opérateurs usuels comme : des conditions ternaires, des relations de comparaison, des conjonctions, des disjonctions ainsi que l'introduction de variables. Elle est donc facilement transposable à d'autres langages de programmation.
Rendre exécutable des définitions non explicites permet d'élargir le langage E-ACSL et permet au greffon d'être mieux combiné à d'autres greffons de Frama-C. Néanmoins, il reste encore des points à traiter pour que cette traduction devienne plus générale, notamment ajouter la gestion de certains opérateurs utilisable dans ces définitions non explicite. Mais aussi ajouter une phase d'optimisation sur les définitions explicites traduites par E-ACSL afin de ne pas retester certaines conditions plusieurs fois. Dans le cas de l'extraction de fonction, la fonction généré peut produire une erreur si la définition axiomatique exprimée n'est pas complète (des axiomes logiques manquent pour définir tous ses comportements). Cette vérification de complétude est faite à l'exécution, mais il serait possible dans certains cas de signifier une définition est incomplète pendant la génération du code. La formalisation de cet algorithme n'a pas été réalisé du fait de sa taille et de sa complexité. Il serait intéressant de le faire à l'aide de Coq et d'extraire le générateur de code prouvé.

2 Introduction

Au CEA-LIST, le Laboratoire de Sureté Logicielle (LSL) développe l'outil Frama-C [2], une plate-forme logicielle facilitant le développement d'outils d'analyses de programmes C. Les programmes C analysés par Frama-C peuvent être annotés par des spécifications formelles écrites dans un langage appelé ACSL [4]. Pour valider ces spécifications, Frama-C offre différentes techniques de vérification pouvant collaborer les unes avec les autres. Une de ces techniques a notamment pour but de traduire une sous-classe des annotations ACSL dites exécutables en instructions C, pour les intégrer au programme analysé. Cette transformation, programmée dans Frama-C dans un greffon appelé (aussi) E-ACSL permet d'obtenir un nouveau programme C dont la correction vis-à-vis de sa spécification est vérifiée dynamiquement. Cette technique de vérification offerte par E-ACSL est appelée la vérification à l'exécution (*runtime assertion checking*) [9].
Le greffon E-ACSL souhaite inclure à son langage les définitions inductives ainsi que les fonctions et les prédicats axiomatiques pouvant être défini dans ACSL. Ces définitions non explicites ont leur comportement défini par un ensemble de prédicat logique. Ajouter cette traduction des définitions non explicite permet à E-ACSL d'élargir les constructions qu'il supporte [16] et lui permet de mieux se combiner à d'autres greffons de Frama-C. Par exemple, il peut être mieux combiné au greffon WP [3] faisant de la preuve déductive [12] de programme C avec des spécifications écrites en ACSL, qui arrive plus facilement à effectuer des preuves avec des définitions axiomatiques ou inductives que des définition explicite récursives.

État de l'art

Différents outils de vérification à l'exécution existent comme E-ACSL pour le language C, il existe Ortac [11] pour OCaml, JML [8] pour Java... Ces outils utilisent un langage de spécification formelle, souvent portant le même nom. La plupart de ces outils ne peuvent pas rendre exécutables des spécifications utilisant des définitions non explicites, car ces définitions ne sont pas directement exécutables et donc ne les supportes pas. Certains assistants de preuve comme Isabelle/HOL permettent de définir des des types de données inductifs ainsi que des relations inductives et d'y extraire du code fonctionnel [6]. Dans l'assistant de preuve Coq, P-N.Tollitte [17] a développé un outil permettant d'extraire du code OCaml exécutable à partir de relation inductive. Il a aussi développé un outil similaire pour générer du code fonctionnel, à partir de spécification dans un environnement Focalise [10]. Récemment, quickChick un générateur de test Coq utilise une méthode similaire pour rendre exécutable des relations inductives et générer des tests [14]. Des travaux similaires dans Coq son toujours en cours, notamment L. Dubois de Prisque avec l'aide de P.Vial, V.Blot et C.Keller au LMF travaille actuellement à décider automatiquement des prédicats inductifs dans le cadre du projet SMTCoq [1] utilisant des prouveurs automatiques classiques.
La méthode d'extraction que je propose s'inspire essentiellement de celle de P-N.Tollitte utilisant un mode d'extraction (Section 4.1). Mais diffère totalement sur l'utilisation des types (n'étant pas des types inductifs, mais des types C), ainsi sur les langages d'entrée et de sortie.

3 Présentation du processus d'extraction

Langage source ACSL propose essentiellement deux constructions de définition non explicite : les prédicats inductifs et les axiomatiques.
Un prédicat inductif est défini par un nom, une liste d'argument ayant chacun un type et un ensemble de constructeur étant des prédicats logiques à respecter. Extraire une définition inductive correspond à extraire un prédicat vérifiant ses constructeurs (prédicat logique spécifiant le comportement de l'inductif).
Un axiomatique permet de définir un ensemble d'axiome à respecter. On peut y ajouter des déclarations de fonction ou prédicat logique et définir leurs comportements avec des axiomes. Un axiomatique a donc un nom et contient un ensemble de déclarations de fonction et de prédicat ainsi qu'un ensemble d'axiome. Les fonction et prédicat axiomatique seront transformés en prédicat inductif et par la suite et sont extraits en prédicat ou en fonction logique.

Langage destination Le langage de destination n'est pas directement C dans mon cas, mais les fonctions et les prédicats logiques d'E-ACSL. Cela garantit la pureté des définitions générées et permet de bénéficier de toutes constructions qu'E-ACSL traduit vers du code C. Comme la phase de typage des termes et des prédicats logiques [5, 13] permettant de produire du code C efficace. Ce qui lui permet notamment d'utilisé les types logiques ACSL qui définissent : integer représentant les entiers relatifs $\mathbb{Z}$, real représentant les nombres réels $\mathbb{R}$ ou encore bool représentant les booléens $\mathbb{B}$. Ou encore des fonctions intégrées au langage comme $\backslash min$, $\backslash max$, $\backslash sum$...

Exemple de plusieurs définitions du PGCD La figure 1 montre de multiple définition possible d'une même fonction en ACSL. Dans cet exemple, on y définie un prédicat vérifiant le Plus Grand Commun Diviseur (PGCD) entre deux nombres n et m et son résultat r. Ce prédicat peut être défini de manière non explicite, par un prédicat inductif (Figure 1b) ou par un prédicat axiomatisé (Figure 1a). L'extraction du prédicat explicite associé est un prédicat logique ACSL (Figure 1c). Pour définir la fonction de PGCD de manière non explicite ACSL offre la possibilité de définir des fonctions axiomatisées (Figure 1a). L'extraction de la fonction de PGCD explicite génère une fonction logique ACSL (Figure 1d).

```
axiomatic axiomatic_gcd {
    predicate is_gcd(integer n, integer m, integer r);
    logic integer gcd(integer n, integer m);

// axiome definissant le comportement du predicat is_gcd
    case pred_gcd_0:
    \forall integer n1; is_gcd(n1, 0, n1);

    case pred_gcd_n:
    \forall integer n2, m2, r2;
        m2 != 0 ==> is_gcd(m2, n2 % m2, r2) ==>
        is_gcd(n2, m2, r2);

// axiome definissant le comportement de la fonction gcd
    case fun_gcd_n:
    \forall integer n1; gcd(n1, 0) == n1;

    case fun_gcd_n:
        \forall integer n2, m2, r2;
        m2 != 0 ==> gcd(n2, m2) == gcd(m2, n2 % m2);
}
```

(a) Axiomatique définisant et un prédicat de vérfication du PGCD et la fonction PGCD

```
inductive is_gcd(integer n, integer m, integer r) {
    case gcd_0:
    \forall integer n1; is_gcd(n1, 0, n1);

    case gcd_n:
    \forall integer n2, m2, r2;
        m2 != 0 ==> is_gcd(m2, n2 % m2, r2) ==>
        is_gcd(n2, m2, r2);
}
```

(b) Inductif vérifiant le PGCD

```
predicate is_gcd(integer n, integer m, integer r) =
    m == 0 && r == n ||
    m != 0 && is_gcd(m, n % m, r);
```

(c) Prédicat extrait vérifiant le PGCD

```
logic integer gcd(integer n, integer m) =
    m == 0 ? n
        : m != 0 ? pgcd(m, n % m)
            : ( "␣Error␣incomplete␣" : 0);
```

(d) Fonction extrait vérifiant le PGCD

FIGURE 1 – Multiple définition du PGCD

Les définitions axiomatiques de ACSL ont une forme très générale, par exemple, elle n'impose pas la positivité comme en Coq. Pour ne pas traiter toute la syntaxe de ACSL dans ces définitions, je commence donc par définir la syntaxe des inductifs et des axiomatiques que je peux traduire. Pour cela, je définis le langage Mini-Inductive comme un sous-ensemble des inductifs ACSL et Mini-Axiomatic comme un sous-ensemble des axiomatiques ACSL que l'on peut extraire.

Pour ne pas avoir à définir trois algorithmes d'extraction, je fais une traduction des définitions axiomatiques (prédicats et fonctions) vers les inductifs et j'utilise un mode d'extraction. Le mode dit complet correspond à la génération de prédicats logiques et le mode dit incomplet à la génération de fonctions logiques.

Pour extraire facilement un inductif, une phase de normalisation de l'inductif suivant le mode utilisé est nécessaire. Après avoir normalisé l'inductif, il devient simple d'extraire une définition exécutable en suivant les différents constructeurs de l'inductif.

La figure 2 ci-dessous représente tout le processus d'extraction des définitions axiomatiques. On peut y retrouvé dans les rectangles pleins les constructions pouvant être extraite du langage ACSL. Les autres rectangles correspondent aux sous ensembles syntaxiques que j'ai définis. La bulle *normalization* est l'algorithme de normalisation appliqué sur les définitions inductives à extraire. Enfin, les flèches pleines représentent la traduction vers un prédicat explicite. Tandis que celle en pointiller représente la traduction des fonctions explicite.

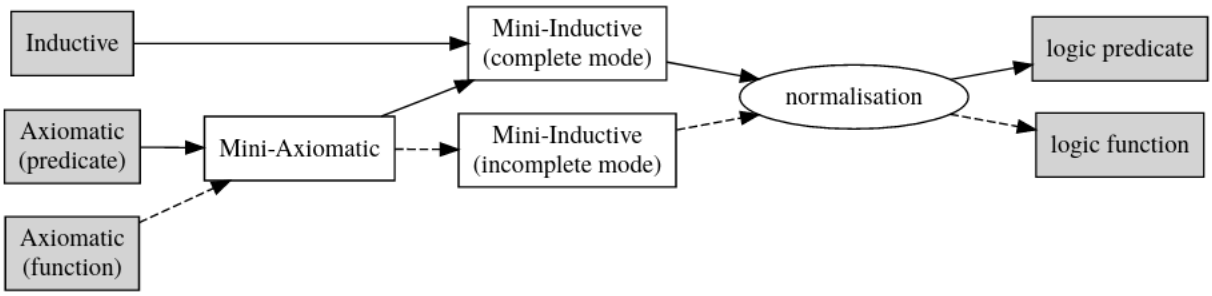

FIGURE 2 – Processus d'extraction

4 Langages d'étude

4.1 Inductif

ACSL offre la possibilité de définir des prédicats inductifs, permettant de définir inductivement le comportement d'une fonction. Ces définitions permettent de vérifier qu'une fonction a le bon comportement. L'extraction d'un prédicat inductif produit alors un prédicat logique que E-ACSL qui sera retranscrite en C par E-ACSL. Un inductif ACSL dans sa forme générale est constitué d'un nom, d'un ensemble de constructeurs et une liste de paramètres. Un constructeur a un nom et un prédicat le définissant. On appelle ici conclusion d'un constructeur le dernier prédicat à respecter et prémisse ce qui se trouve avant.
Par exemple, dans le constructeur gcd_0, la conclusion est : $gcd(n2, m2, r2)$ et la prémisse ce qu'il se trouve avant.

Définition d'un langage inductif Mini-Inductive Je commence donc par définir le langage Mini-Inductive comme un sous ensemble des inductif ACSL et servant de base pour la suite. Les inductifs ont donc :

- Comme conclusion d'un constructeur une application inductive de l'inductif auquel il appartient.
- La quantification des variables peut se faire uniquement par l'opérateur universel `\forall` sous forme prénexe.
- La prémisse d'un constructeur utilise des prédicats simples, séparés entre eux par des implications ou des `\let` (servant à introduire des variables locales).
- Les prédicats contenus dans la prémisse contiennent que des comparaisons entre des termes, des applications de prédicat ou des applications inductives.
- Un terme contenu dans un `\let` ne contient pas de prédicat, il ne contient donc pas d'application inductive, d'application de prédicat ou de comparaison.
- Les prédicats inductifs ne peuvent pas s'utiliser entre eux.
- Les types utilisables sont tous les C ainsi que les types logiques de ACSL. Et les types logiques ACSL étant les nombre relatif défini par le type integer, les nombres réels défini par real et les booléens par bool.

La définition syntaxique de ce langage Mini-Inductive se trouve dans l'annexe A.

Définition On peut définir formellement un prédicat inductif comme un triplet $(name_{ind}, \tau, \Gamma)$ contenant son nom, son type τ et son ensemble Γ des constructeurs. Le type τ est nécessairement un type de la forme $\tau_1 \to \cdots \to \tau_n \to Prop$ dans lequel le τ_i le types de ses arguments. Les constructeurs γ_i contenus dans Γ sont des triplets $(name_{ctor}, B, Q, C)$ dans lequel se trouve le nom du constructeur, l'ensemble de variables universelles B introduit au début du prédicat associé au constructeur (chaque variable est un couple contenant le nom de la variable et son type), la prémisse Q du constructeur (se terminant toujours par `\true`) et la conclusion C étant l'application de l'inductif en cours de définition.

Exemple La définition de l'inductif vérifiant le PGCD de la Figure 1b se traduit donc par cette définition :

$$
\begin{aligned}
& Inductive_{is\_gcd} = (\text{"is\_gcd"}, \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z} \to Prop, \{\gamma_1, \gamma_2\}) \\
\text{avec } & \gamma_1 = (\text{"gcd\_0"}, B_1, \texttt{\textbackslash true}, \gcd(n1, 0, n1)) \\
& B_1 = \{(n1, \mathbb{Z})\} \\
\text{et } & \gamma_2 = (\text{"gcd\_S"}, B_2, Q_2, \gcd(n2, m2, r2)) \\
& B_2 = \{(n2, \mathbb{Z}), (m2, \mathbb{Z}), (r2, \mathbb{Z})\} \\
& Q_2 = \gcd(n2, n2\%m2, r2) \Longrightarrow \texttt{\textbackslash true}
\end{aligned}
$$

Les inductifs sont centraux dans l'extraction et vont nous permettre à la fois d'extraire des prédicats, mais aussi des fonctions logiques. Pour permettre l'extraction de ces deux définitions, on définit un mode d'extraction.

Modes d'extraction

Un mode d'extraction permet de définir les arguments d'entrée et de sortie d'une application inductive, ce qui est essentiel pour pouvoir déterminer si l'extraction concerne une fonction ou un prédicat. L'utilisation de mode d'extraction est inspiré de l'extraction proposée par P-N.Tollite. Un mode d'extraction est défini comme un ensemble d'indices, correspondant aux indices des arguments d'une définition inductive et indique donc les arguments d'entrée et de sortie de la fonction.
Un mode sera dit *complet* s'il comprend tous les indices de la définition inductive (donc des indices allant de 1 à n) et *incomplet* si le dernier indice n'est pas présent dans l'ensemble, donc ayant les indices : $\{1 \cdots n-1\}$.

On commence par définir le complémentaire d'un mode m noté $\overline{m}$:

$$\overline{mode} \triangleq \{1, \cdots, n\} \setminus mode$$

On peut définir la fonction *in*, rendant un n-upplet des termes en entrée de l'application inductive suivant un mode d'extraction.

$$\text{in}(\{i_1, \cdots, i_m\}, ind(t_1, \cdots, t_n)) \triangleq (t_{i_1}, \cdots, t_{i_m})$$

Et une fonction *out* retourne, \texttt{true} si tous les arguments sont des termes d'entrée, un terme unique si un seul argument est celui de sortie, sinon il renvoie un n-upplet des arguments de sortie.

$$\text{out}(mode, ind(t_1, \cdots, t_n)) \triangleq \begin{cases} \texttt{true} & \text{si } n = |mode| \quad (1) \\ t_i & \text{si } \exists! i, 1 \leq i \leq n \wedge i \notin mode \quad (2) \\ in(\overline{mode}, ind(t_1, \cdots, t_n)) & \text{sinon} \quad (3) \end{cases}$$

On peut définir les fonctions *in_vars* retournant les variables d'entrée et *our_vars* celle retournant les variables sorties d'une application inductive suivant un mode. Ces fonctions utilisent la fonction *vars* retournant l'ensemble des variables libres d'un terme de la façon suivante :

$$\text{in\_vars}(mode, ind(t_1, \cdots, t_n)) \triangleq \bigcup_{i \in mode} \text{vars}(t_i)$$
$$\text{out\_vars}(mode, ind(t_1, \cdots, t_n)) \triangleq \bigcup_{i \in \overline{mode}} \text{vars}(t_i)$$

Plus tard, nous utiliserons uniquement le mode complet pour l'extraction vers des prédicats (dans le cas des inductifs et des prédicats axiomatiques) et le mode incomplet pour l'extraction des fonctions (En pratique les fonctions axiomatiques). La fonction *out* ne produira donc jamais de n-upplet (cas 3) en pratique.

Exemple Voici des exemples de variables d'entrée et de sortie, dans l'application inductive de *is_gcd* (Figure 1b) contenue dans la prémisse de *gcd_S* en utilisant différents modes

d'extractions.

$$\begin{aligned}\text{invars}(\{1,2,3\}, \text{is\_gcd}(m2, n2\%m2, r2)) &= \{n2, m2, r2\}\\ \text{outvars}(\{1,2,3\}, \text{is\_gcd}(m2, n2\%m2, r2)) &= \emptyset\end{aligned}$$

$$\begin{aligned}\text{invars}(\{1,2\}, \text{is\_gcd}(m2, n2\%m2, r2)) &= \{n2, m2\}\\ \text{outvars}(\{1,2\}, \text{is\_gcd}(m2, n2\%m2, r2)) &= \{n2, m2\}\end{aligned}$$

Définir un mode d'extraction permet de fixer les arguments d'entrée et de sortie d'une application inductive. Ainsi utiliser un mode complet consiste donc à extraire un prédicat vérifiant l'application inductive (car tous les arguments sont des arguments d'entrée), tant dit qu'utiliser un mode incomplet consiste à pouvoir calculer un terme en position de sortie d'une conclusion d'un constructeur. Pour effectuer une extraction, il est nécessaire de voir si celui-ci est cohérent vis-à-vis des variables qu'il utilise.

Cohérence d'un inductif

La cohérence d'un inductif en suivant un mode d'extraction permet de vérifier si un inductif est exécutable vis-à-vis de ces constructeurs. Le procédé de vérification de la cohérence s'effectue sur les constructeurs d'un inductif en utilisant un mode d'extraction. Pour plus de lisibilité, on introduit la fonction $\mathcal{M}$ renvoyant le mode associé à un inductif.
Si tous les constructeurs sont cohérents, alors l'inductif est lui-même cohérent. On a défini la fonction $is\_consistent$ faisant cette vérification, utilisant la fonction $is\_consistent_\gamma$ (définie par la suite) vérifiant qu'un constructeur est bien cohérent.

$$\begin{aligned}&\text{is\_consistent} : Inductif.t \to \mathbb{B}\\ &\text{is\_consistent}((name_{ind}, \tau, \Gamma)) \triangleq \quad \forall\ \gamma_i \in \Gamma, \text{is\_consistent}_\gamma(\gamma_i)\end{aligned}$$

Vérifier la cohérence d'un constructeur, consiste à construire un ensemble de variable connue $venv$ et à vérifier que chaque élément de la prémisse à ses variables contenus dans $venv$. À l'initialisation, les variables contenus dans $venv$ sont les variables globales V_{glob} ainsi que les variables d'entrée de la conclusion. Une fois la prémisse vérifiée, il faut vérifié que le terme de sortie puisse bien être construit avec les variables initialisées par les prémices, ainsi qu'avec les variables d'initialisation. On vérifie donc que les variables de sortie de la conclusion sont bien incluses dans $venv$.
Pour faire cette vérification, on définit la fonction $is\_consistent_\gamma$, utilisant la fonction $consistent\_venv_Q$ (défini par la suite) vérifiant la cohérence de la prémisse et nouveaux ensemble $venv$ à vérifé.

$$\begin{aligned}&is\_consistent_\gamma : Constructor.t \to \mathbb{B}\\ &\text{is\_consistent}_\gamma((name_{ctor}, B, Q, C)) \triangleq \begin{cases} vars_{out} \subseteq venv & \text{si } venv_{opt} = Some(venv)\\ \bot & \text{sinon } venv_{opt} = None\end{cases}\\ \text{avec}\quad &venv_{opt} = \text{consistent\_venv}_\text{Q}(V_{glob} \cup vars_{in}, Q)\\ &vars_{in} = in\_vars(\mathcal{M}(ind), ind(t_1, \cdots, t_n))\\ &vars_{out} = out\_vars(\mathcal{M}(ind), ind(t_1, \cdots, t_n))\end{aligned}$$

Vérifié la cohérence de la prémisse consiste à vérifier récursivement les éléments de la prémisse et d'ajouter des variables dans *venv* si l'élément vérifié en crée de nouvelles variables. Si la prémisse est incohérente, la valeur de retour de la fonction $consistent\_venv_Q$ est signifiée par le renvoie de la valeur $None$. Pour chaque élément de la prémisse, il faut :

- Vérifier pour une application inductive en suivant son mode d'extraction que ses variables d'entrée sont dans *venv* et ajouter à *venv* les variables de sortie.
- Dans le cas d'une application de prédicat ou d'une comparaison, il faut vérifié que toutes ses variables libres sont dans *venv*.
- Dans le cas de l'utilisation d'un `\let` introduisant une variable locale, il faut vérifier que toutes les variables du terme à calculer sont dans *venv* et ajouter à *venv* la variable créé par `\let`.

$$consistent\_venv_Q : VarSet.t \to Predicat.t \to VarSet.t\ option$$

$$\text{consistent\_venv}_\text{Q}(venv, ind(t_1, \cdots, t_n) \Longrightarrow P) \triangleq \begin{cases} \text{consistent\_venv}_\text{Q}(venv \cup vars_{out}, P) & \text{si } vars_{in} \subseteq venv \\ None & \text{sinon} \end{cases}$$

$$\text{avec} \quad \begin{array}{lcl} vars_{in} & = & in\_vars(\mathcal{M}(ind), ind(t_1, \cdots, t_n)) \\ vars_{out} & = & out\_vars(\mathcal{M}(ind), ind(t_1, \cdots, t_n)) \end{array}$$

$$\text{consistent\_venv}_\text{Q}(venv, p \Longrightarrow P) \triangleq \begin{cases} \text{consistent\_venv}_\text{Q}(venv, P) & \text{si } vars(p) \subseteq venv \\ None & \text{sinon} \end{cases}$$

$$\text{consistent\_venv}_\text{Q}(venv, \texttt{\textbackslash let}\ v = t;\ P) \triangleq \begin{cases} \text{consistent\_venv}_\text{Q}(\{v\} \cup venv, P) & \text{si } vars(t) \subseteq venv \\ None & \text{sinon} \end{cases}$$

$$\text{consistent\_venv}_\text{Q}(venv, \texttt{\textbackslash true}) \triangleq Some(venv)$$

Exemple Certains constructeurs peuvent devenir incohérent en suivant des modes différent. C'est le cas du prédicat inductif *sign* défini dans la Figure 3 ci-dessous, vérifiant le signe d'un nombre réel. Ce prédicat inductif contient le constructeur *is_pos* qui est cohérent en suivant un mode complet et incohérent dans un mode incomplet. En effet, si on vérifie la cohérence de ce constructeur en suivant un mode complet, on commence par initialiser *venv* avec l'ensemble des variables d'entrée de la conclusion, correspondant à l'ensemble $\{n2, r2\}$. Le premier élément de la prémisse, $x2 > 0.0$ est bien vérifié, car $x2$ est contenue dans *venv*, on peut passé au prochain prédicat à vérifier. Le second prédicat de la prémisse $r2 == POS$, nécessite la variable $r2$ soit bien contenue dans *venv*, ce qui est le cas. La prémisse vérifie donc la cohérence, la fonction $consistent\_venv_Q$ renvoie donc le même *venv* qu'à l'initialisation, car aucune variable n'a été introduite. Il ne reste plus qu'a vérifié que les variables de sortie de la conclusion sont incluses dans *venv*, ce qui est le cas, car l'ensemble des variables de sortie de la conclusion est vide. Le constructeur *is_pos* est donc bien cohérent en suivant un mode complet. À l'inverse, en utilisant un mode incomplet, ce constructeur devient incohérent, car à l'initialisation l'ensemble *venv* ne contient pas $r2$, car celui-ci est position de sortie. La vérification du prédicat $r2 == POS$, n'est alors pas vérifié, car il nécessite $r2$ pour être vérifié. Ce constructeur est donc incohérent suivant ce mode d'extraction.

```
#define POS 1
#define NEG -1
#define ZERO 0

/*@
inductive sign(real x, integer r) {
    case is_zero:
    sign(0., ZERO);

    case is_positve:
    \forall real x2, integer r2;
        x2 > 0. ==> r2 == POS; sign(x2, r2);

    case is_negative:
    \forall real x3;
        x3 < 0. ==> sign(x3, NEG);
}
*/
```

FIGURE 3 – Inductif vérifiant le signe d'un nombre réel

La cohérence d'un inductif est un bon outil pour construire une normalisation des inductifs permettant de transformer facilement un inductif en définition explicite.

Normalisation des inductifs

La normalisation des inductifs permet de rendre un inductif simple à extraire suivant un mode d'extraction. La normalisation conserve la cohérence, voir dans certain cas rend cohérent des constructeurs et échoue s'il est impossible de rendre cohérent le constructeur. Elle permet de déterminer facilement quelles variables seront utilisé comme argument pour une définition explicite et rend l'extraction de définition récursive simple.
La normalisation se fait sur chaque constructeur et consiste à :

1. Délinéariser les variables des applications inductives en suivant le mode d'extraction.
2. Introduire des égalités pour tous les termes qui ne sont pas des variables uniques en suivant le mode d'extraction.
3. Transformé des égalités en `\let` si une variable manque pour rendre cohérent certaines prémisses.
4. Renommer toutes les variables d'entrée des conclusions similairement.

Introduction des termes Dans l'application inductive, si un de ses arguments n'est pas une variable simple, ce terme est remplacé pour une nouvelle variable et une égalité est ajoutée.
Si l'application inductive est la conclusion, les arguments en position d'entrée sont normalisés (les arguments en position de sortie ne sont pas concernés). On crée donc une égalité avec

l'argument concerné en tête des prémices et la nouvelle variable fraiche crée. Cela permet au argument d'entrée de la conclusion d'être uniquement des variables, qui pourront ensuite être utilisé comme arguments de la définition explicite.
Dans le cas d'une application inductive contenue dans une prémisse, la normalisation se fait qu'en suivant les arguments en position de sortie et la condition est insérée à la suite de l'application inductive. Ces applications inductives contenues dans les prémisses signifient que la définition extraite sera récursive. Ce cas est utilisable qu'avec l'utilisation d'un mode incomplet, l'argument de sortie normalisé sera donc la variable contenant le résultat de l'appel récursif. Avoir cette variable permet de traduire facilement cette application inductive en appelle à la fonction explicite récursive.

La figure 4 ci-dessous montre la normalisation introduisant des termes du constructeur *gcd_0* dans un mode complet. On peut voir que le second argument de la conclusion étant 0, a été remplacé par la variable fraiche $m1$ et l'égalité $m1 == 0$ a été ajouté en tête de la prémisse.

```
1 inductive is_gcd(integer n, integer m, integer r) {
2     case gcd_0:
3     \forall integer n1, m1;
4         m1 == 0 ==> gcd(n1, m1, n1);
5     ...
6 }
```

FIGURE 4 – Normalisation introduisant un terme du constructeur gcd_0

Dé-linéarisation des variables Délinéarisée les variables permet de rendre tous les arguments d'entrée de la conclusion en variables uniques. Pour cela, on crée une nouvelle variable fraiche et ajoute une égalité en tête de prémisse.
Cette dé-linéarisation dans le cas d'une application inductive présente dans une prémisse n'est pas nécessaires, car nous utilisons des modes ayant au maximum un argument de sortie. Cette dé-linéarisation des variables de la conclusion permettant de rendre unique chaque variable d'entrée de la conclusion rend simple la détermination des des arguments de la définition explicite.
La figure ci 5 montre la dé-linéarisation de $n1$ sur le constructeur *gcd_0* en utilisant un mode complet.

```
1 inductive is_gcd(integer n, integer m, integer r) {
2     case gcd_0:
3     \forall integer n1;
4         r1 == n1 ==> gcd(n1, 0, r1);
5     ...
6 }
```

FIGURE 5 – Normalisation délinéarisant les variables du constructeur gcd_0 en utilisant un mode complet

Transformation des égalités

Certaines égalités présentes dans une prémisse servent à vérifier une égalité mais d'autre exprime le souhait de fixer une variable. Ces égalités fixant des variables doivent être remplacées par un `\let`. Pour identifier ces égalités, on se base sur la cohérence du constructeur. Ainsi les variables devant êtres fixés se traduisent dans la cohérence par une variable manquante dans *venv*. Si cette variable est à gauche ou à droite de l'égalité, il est alors simple de la fixer avec un `\let`.

La figure 6 montre une transformation d'une égalité en utilisant un mode incomplet, rendant cohérent ce constructeur. À l'initialisation *venv* contiendrait que la variable $n1$, par la suite, lors de la vérification de l'égalité entre $r1$ et $n1$ une erreur d'incohérence serait relevé, car la variable $r1$ n'est pas dans *venv*. En regardant plus attentivement ce constructeur et l'extraction en mode incomplet, on se rend compte que le constructeur doit déterminer le terme $r1$ à partir de sa prémisse, pour l'utiliser dans le terme de sortie de la conclusion. En changeant l'égalité par un `\let` il n'y a plus de problème de cohérence.

```
1  inductive is_gcd(integer n, integer m, integer r) {
2      case gcd_0:
3      \forall integer n1;
4          r1 == n1 ==> gcd(n1, 0, r1);
5  // Dans un mode incomplet {1,2}
6  // venv = {n1}, "r1 == n1" est incoherent car: {r1,n1} ⊈ {n1}
7
8      ...
9  }
10 inductive is_gcd(integer n, integer m, integer r) {
11     case gcd_0:
12     \forall integer n1;
13         \let r1 = n1; gcd(n1, 0, r1);
14 // Avec la normalisation "\let r1 = n1;"
15 // deviens coherent car: {n1} ⊆ {n1}
16
17     ...
18 }
```

FIGURE 6 – Changement d'une égalité en `\let` dans un mode incomplet

Renommage des variables Les dernières étapes de la normalisation consiste à renommer toutes les variables du constructeur, en créant une table de renommage constitué des variables unique d'entrée de la conclusion. Et des variables données avec le type de l'inductif à la même position. Cela permet d'avoir les mêmes noms de variables dans chaque constructeur, et ainsi pouvoir extraire une définition avec ces mêmes variables en argument.

La figure 7 est la normalisation complète de l'inductif *is_gcd* (Figure 1b) en utilisant un mode complet. On y voit les normalisations précédentes ainsi que le renommage des différentes variables des constructeurs en suivant le nom des arguments des constructeurs.

```
inductive is_gcd(integer n, integer m, integer r) {
    case gcd_0:
    \forall integer n, m, r;
        m == 0 ==> n == r ==> gcd(n, m, r);

    case gcd_S:
     \forall integer n, m, r;
        m != 0 ==> gcd(m, n % m, r) ==> gcd(n, m, r);
}
```

FIGURE 7 – Inductif PGCD totalement normalisé (utilisant un mode complet)

Les inductifs ainsi normalisé suivent la cohérence du mode utilisé et permet une extraction simple d'une définition explicite. En effet, l'induction normalisée bénéficie d'une conclusion avec des variables d'entrée assimilables aux arguments d'une définition explicite. Il bénéficie aussi dans le cas des fonctions récursives se traduisant par application inductive dans les prémisses d'un argument de sortie étant une variable unique pouvant stocké le résultat de l'appel récursif.

4.2 Axiomatiques

Le langage d'annotation ACSL, contient des définitions axiomatiques, pouvant définir des fonctions et des prédicats axiomatisés. Il est possible d'y extraire de ces définitions des fonction et prédicat logique ACSL. Les axiomatiques de ACSL comme ses inductifs ont une très grande expressivité sur les axiomes qu'il est possible de définir. On fixe donc des règles sur les axiomes choisis pour l'extraction. Pour être généralisé la méthode, les définitions axiomatiques sont transformé en inductif et extrait par la suite, en suivant un mode complet pour les prédicats et incomplet pour les fonctions axiomatisé.

Un axiomatique ACSL dans sa forme général utilise un nom, un ensemble d'axiome logique et à un ensemble déclaration de fonction et de prédicat logique. Les axiomes peuvent servir à définir le comportement de ces fonctions et ces prédicats.

Exemple L'exemple d'extraction d'un prédicat axiomatisé est déjà visible sur la Figure 1 montrant l'extraction du prédicat *is_gcd* utilisant les axiomes *pred_gcd_0* et *pred_gcd_n* pour définir le comportement du prédicat *is_gcd*. Pour générer le prédicat logique *is_gcd* de la Figure 1c, on extrait d'abord l'inductif du meme nom (Figure 1b). Puis en utilisant un mode complet extrait le prédicat logique associé (Figure 1c).

La figure 8 montre l'extraction de la fonction logique ACSL factoriel (Figure 8c) en utilisant l'axiomatique *factorial_axiomatic* (Figure 8a). Pour cela, on extrait l'inductif *factorial_ind* (Figure 8b), puis utilise un mode incomplet pour effectuer l'extraction.

```
axiomatic factorial_axiomatic {
    logic integer factorial(integer n);

    axiom factorial_1:
    \forall integer n;
        n <= 1 ==> factorial(n) == 1;

    axiom factorial_n:
    \forall integer n;
        n > 1 ==> factorial(n - 1) * n == factorial(n);
}
```

(a) Axiomatique définissant la fonction factorielle

```
inductive factorial_ind(integer n, integer r) {
    case factorial_1:
    \forall integer n;
        n <= 1 ==> factorial_ind(n, 1);

    case factorial_n:
    \forall integer n, tmp1;
        n > 1 ==> factorial_ind(n - 1, tmp1) ==>
        factorial_ind(n, tmp1 * n);
}
```

(b) Inductif extrait de l'axiomatique pour la fonction factorielle

```
logic integer factorial_logic(integer n) =
    n <= 1 ? 1 :
        (n > 1 ? \let tmp1 = factorial_logic(n - 1);
            tmp1 * n  : ( "Error incomplete": 0));
```

(c) Fonction logique généré par factoriel

FIGURE 8 – Extraction la fonction axiomatisé Factoriel

Définition d'un langage axiomatique Mini-axiomatic Les axiomes des axiomatiques ACSL ayant une forme très générale tout comme les prédicats inductifs. On définit donc le langage Mini-Axiomatic étant un sous ensemble des axiomatiques ACSL que l'on peut extraire. Les axiomatiques de ce langage étant transformé dans le langage Mini-Inductive sont donc similaires. Le langage Mini-Axiomatic utilise :

- Une quantification des variables dans les axiomes que par l'opérateur universel `\forall` sous forme prénexe.
- Des axiomes ayant des prédicats qui sont séparés par des implications ou des `\let`.
- Des prédicats contenus dans la prémisse des axiomes ne contiennent que des comparaisons entre des termes, des applications de prédicat.
- Un terme contenu dans un `\let` ne contient pas de prédicat.

— Les mêmes types que Mini-Inductive.

La définition syntaxique du langage Mini-Axiomatic se trouve dans l'annexe B.

Définition Un axiomatique est donc défini formellement par un quatre-uplet $(name_{axiomatic}, P, F, \Phi)$ contenant : le nom de l'axiomatique, l'ensemble des déclarations de prédicats P, l'ensemble des déclarations de fonctions logiques F ainsi que l'ensemble de ses axiomes Φ. Chaque axiome ϕ_i contenu dans Φ est un quatre-uplet $(name_{axiom}, B, Q, C)$ similaire au constructeur d'inductif et contient le nom de l'axiome $name_{axiom}$, l'ensemble de variables universelles introduit B par `\forall` au début du prédicat associé à l'axiome. Ainsi que la prémisse Q du constructeur se terminant toujours par `\true` et une conclusion C étant une application de prédicat axiomatisé ou une égalité entre une fonction axiomatisée et un terme quelconque.

Exemple La structure utilisée pour la définition axiomatique $factorial\_axiomatic$ (Figure 8a) est la suivante :

$$\text{Axiomatic}_{\text{factorial}} = (\text{"factorial\_axiomatic"}, \emptyset, \{F_{fact}\}, \{\phi_1, \phi_2\})$$

$$\begin{array}{rrlrrl}
 & F_{fact} & = & factorial : \mathbb{Z} \to \mathbb{Z} & & \\
 & \phi_1 & = & (\text{"factorial_1"}, B_1, Q_1, C_1) & \phi_2 & = & (\text{"factorial_n"}, B_2, Q_2, C_2) \\
\text{avec} & B_1 & = & \{(n, \mathbb{Z})\} & B_2 & = & \{(n, \mathbb{Z})\} \\
 & Q_1 & = & n <= 1 \Longrightarrow \texttt{\textbackslash true} & Q_2 & = & n > 1 \Longrightarrow \texttt{\textbackslash true} \\
 & C_1 & = & factorial(n) == 1 & C_2 & = & factorial(n-1) * n == factorial(n)
\end{array}$$

Sélection des axiomes Pour extraire un prédicat ou une fonction axiomatisée en prédicat inductif, il faut identifier les axiomes associés à la définition. Dans le cas des prédicats axiomatisé, les axiomes sélectionnés sont les axiomes ayant pour conclusion le prédicat axiomatisé. Pour les fonctions axiomatiques, les axiomes sélectionnés ont pour conclusion une égalité entre l'application de la fonction axiomatisée et un terme quelconque.

Exemple La Figure 1a contient deux déclarations, la première déclaration est le prédicat axiomatisé *is_gcd* vérifiant le PGCD et la seconde est la fonction axiomatisée *gcd* calculant le PGCD. Il faut donc pouvoir déterminer les axiomes associés à chaque déclaration afin de pouvoir construire l'inductif associé. Pour cela, on sélectionne les axiomes préfixés "pred_" pour le prédicat *is_gcd* et ceux préfixés par "fun_" pour la fonction *gcd*. Ces axiomes seront par la suite transformés en inductif.

Construction d'un inductif

Après avoir sélectionné les axiomes en rapport avec le prédicat ou de la fonction axiomatisé, on peut construire un inductif. Pour cela, il suffit d'extraire le type de l'inductif à construire et tranformé les axiomes précédemment sélectionnés en constructeur.

Construction à partir d'un prédicat axiomatisé Dans le cas d'un prédicat, cette construction est faite facilement, car le type reste le même et les constructeurs générés ont la même forme que les axiomes sélectionnés. La transformation d'un axiome vers un constructeur

n'a qu'à changer les applications du prédicat en cours d'extraction par l'application inductive en cours d'extraction.

Exemple À partir du prédicat *is_gcd* de l'axiomatique *axiomatic_gcd* de la Figure 1a, on peut créer l'inductif de la Figure 1b, en transformant l'axiome *pred_gcd_0* en constructeur *gcd_0* et l'axiome *pred_gcd_n* en constructeur *gcd_n*. Le constructeur n'a pas changé par rapport à l'axiome, car il conserve la même forme, seuls les application du prédicat axiomatisé ont été transformés en application inductives.

Construction à partir d'une fonction axiomatisée Pour une fonction axiomatisée, il est nécessaire d'ajouter le type de retour de la fonction axiomatisé en dernier paramètre de l'inductif généré et transformer les différents axiomes en remplaçant les applications de fonction. La transformation des axiomes en constructeur va donc transformer les égalités utilisant la fonction à extraire en application inductive, mais aussi dans les autres cas contenant un terme étant l'application de la fonction, changer ce terme en une nouvelle variable et introduire une application inductive ayant comme dernier argument la nouvelle variable.

Pour cela, on commence par introduire la fonction *reduce_term* qui change dans un terme toutes les applications de la fonction extraite par une variable fraiche et retourne un prédicat contenant toutes les applications inductives nécessaires avant d'utiliser ce terme. Ces applications inductives ont pour argument : les différents arguments des fonctions appliqués ainsi que la variable crée en dernier argument (étant sa valeur de retour). La fonction *reduce_term* utilise donc la déclaration de la fonction axiomatique à extraire, la déclaration de l'inductif extrait ainsi que le terme à changer et retourne un triplet contenant le nouveau terme à utiliser, les nouvelles variables fraiches crée (qu'il faudra quantifiés dans le constructeur) ainsi qu'un prédicat contenant des applications inductives séparées par des implications (ce prédicat vaut `\true` si aucune modification du terme est nécessaire).

$$reduce\_term : Decl.t \to IndApp.t \to Term.t \to Terme.t \times VarSet.t \times Predicate.t$$

exemple Voici l'utilisation de *reduce_term* sur l'application d'une fonction axiomatisée f prenant deux entiers et renvoyant un entier.

$$\text{reduce\_term}(f, ind_f, 1 + f(2, f(3, 4))) = (1 + b, \{a, b\}, ind_f(3, 4, a) \Longrightarrow ind_f(2, a, b))$$

À partir de la fonction *reduce_term*, on peut construire une fonction *reduce_pred*, qui change des égalités entre l'application de la fonction à extraire et un terme quelconque en une application inductive précédé des prédicats généré par *reduce_term* s'ils sont nécessaires. Pour les autres prédicats, *reduce_pred* remplace par des variables fraiches les applications de fonction et retourne le prédicat précédé des applications inductives nécessaires.

$$\text{reduce\_pred} : Decl.t \to IndApp.t \to Predicate.t \to VarSet.t \times Predicate.t$$

exemple Voici l'utilisation de *reduce_pred* sur une égalité contenant la fonction f à extraire ainsi que sur une condition d'infériorité utilisant la fonction f à extraire.

$$\text{reduce\_term}(f, ind_f, f(2, 4) == 6) = (\emptyset, ind_f(2, 4, 6))$$
$$\text{reduce\_term}(f, ind_f, f(2, 4) < 6) = (\{a\}, ind_f(2, 4, a) \Longrightarrow a < 6)$$

En utilisant ces fonctions, il est maintenant simple de changer un axiome en constructeur d'un inductif. Il suffit pour cela de remplacer tous les `\let` par des égalités (ces égalités se retransformeront en `\let` dans la normalisation des inductifs). Puis d'appliquer *reduce_pred* sur tous les éléments de la prémisse pour les remplacer les éventuelles appelle à la fonction et enfin changer la conclusion étant une égalité en inductif, en utilisant la fonction *reduce_pred*. À chaque utilisation de *reduce_pred* il faut bien veiller à ajouter les nouvelles variables crée dans les variables universelles utilisées dans le constructeur.

En utilisant cette méthode, on arrive à extraire à partir de l'axiomatique définissant la fonction factorielle (Figure 8a) en l'inductif associé (Figure 8b).

4.3 Extraction du code

Il est maintenant simple d'extraire le code pour toutes les définitions inductives normalisées. En effet, grâce à la cohérence, on sait que chaque variable est bien définie dans chacun des constructeurs. L'ordre des constructeurs sera conservé par l'extraction et des prémisses (ce qui permet d'ordonner les calculs lors de la définition non explicite). Pour extraire une fonction ou un prédicat logique, il faut donc : extraire un terme de chaque constructeur un à un et passer le terme extrait au constructeur précédent. Le dernier constructeur utilise le terme $\bot_T$ (T étant le type du terme à extraire), représentant le terme utilisé si aucun constructeur n'arrive à conclure. Ce terme est donc dans un mode complet $\bot_{\mathbb{B}}$, et dans un mode incomplet $\bot_{\tau_n}$ avec τ_n le type du dernier argument. On peut définir la fonction $compile_{ind}$, faisant l'extraction de l'inductif en déterminant $\bot_T$ suivant le mode d'extraction et utilisant la fonction $compile_\Gamma$, en utilisant la fonction $compile_\gamma$. La fonction $compile_\gamma$ transforme le constructeur en terme exécutable, en utilisant le constructeur, ainsi que le terme généré par le constructeur suivant.

$$
\begin{aligned}
\text{compile}_{\text{ind}}((name_ind, \tau_1 \to \cdots \to \tau_n, \Gamma)) \triangleq\ & compile_\Gamma(Gamma, \bot_T) \\
\text{avec}\quad \bot_T = &\begin{cases} \bot_{\mathbb{B}} & \text{si } \{1, \cdots n\} = \mathcal{M}(ind) \\ \bot_{\tau_n}\ 0 & \text{sinon} \end{cases} \\
compile_\Gamma([\Gamma_1; \cdots;\ \Gamma_n], \bot_T) \triangleq\ & \text{compile}_\gamma(\Gamma_1, \text{compile}_\gamma(\Gamma_2, \cdots \text{compile}_\gamma(\Gamma_n, \bot_T) \cdots))
\end{aligned}
$$

Transformation d'un constructeur Pour transformer un constructeur en terme exécutable, il faut générer un terme vérifiant toute la prémisse du constructeur et retourne le terme de sortie de la conclusion si les conditions de la prémisse sont respectés, sinon il faut passer au constructeur suivant.

Génération de prédicat

Pour générer un prédicat, on utilise un mode d'extraction complet (tous les arguments sont des variables d'entrée). Si aucun constructeur n'arrive à conclure, le prédicat généré devra retourné `\false` qui doit donc être notre terme $\bot_{\mathbb{B}}$. Le prédicat logique généré est alors une disjonction de cas représentant toutes les prémisses transformées des constructeurs. Pour transformer une prémisse, il faut remplacer toutes les applications inductives par une application du prédicat généré et faire une conjonction de tous les prédicats la constituant.

En utilisant cette transformation, on obtient donc bien à partir l'inductif *is_gcd* de la

Figure 1b le prédicat explicite logique du même nom de la Figure 1c. Ce prédicat est une disjonction de cas des prémices après la normalisation des différents constructeurs. L'extraction de chaque prémisse produit bien une conjonction de chaque prédicat la constituant.

Génération de fonction

Dans le cas d'une fonction utilisant un mode d'extraction incomplet (le dernier argument est le terme de retour). Si aucun constructeur n'arrive à conclure, la définition est dite incomplète et aucun constructeur ne peut retourner terme, il faut donc le signaler par une erreur. Le terme $\bot_T$ est donc l'erreur d'incomplétude de la fonction à lever. Pour transformer un constructeur, il est nécessaire de changer les applications inductives de la prémisse par un `\let` introduisant la variable contenue dans l'argument de sortie de l'application inductive et ayant pour terme l'application de la fonction en cour génération appliqué aux arguments d'entrées de l'application inductive. Pour générer le terme à exécuter suivant un constructeur, il faut commencer par transformées la prémisse du constructeur en changeant, les implications en conjonction et possiblement généré une condition ternaire si un `\let` est rencontré. Cette condition ternaire utilise les conditions précédentes, le `\let` suivie de la suite de l'extraction de la prémisse, et enfin le terme de l'extraction généré par l'extraction du prochain constructeur. À la fin de l'extraction de la prémisse, on peut ajouter le terme de sortie de la conclusion en utilisant une condition ternaire si la prémisse se terminait par une implication, sinon on peut directement ajouter ce terme.

Dans l'exemple de la fonction factorielle définie axiomatiquement (Figure 8a) puis extrait en inductif (Figure 8b). L'extraction produit bien la fonction $factorial\_logic$ (Figure 8c). On reconnait dans la première condition ternaire $n <= 1\ ?\ 1\ :\ \cdots$ le premier constructeur $factioral\_1$. La seconde condition ternaire $n > 1\ ?$ `\let` $tmp1 = factorial\_logic(n - 1);\ tmp1 * n\ :\ \cdots$ est extraite du second constructeur $factioral\_n$. Enfin, le dernier terme est une erreur signifiant l'incomplétude du constructeur. Ce terme ne sera jamais exécuté avec les conditions ternaires lors de l'exécution, cela montre que cette définition axiomatisée de la fonction factorielle est bien complète.

Ainsi, on peut extraire différentes formes de définition non explicite contenue dans ACSL vers des définitions explicites que E-ACSL pourra alors transformé en code C exécutable.

5 Conclusion et travaux futurs

Cette extraction permet donc de produire du code exécutable à partir d'un sous-ensemble des définitions inductives et axiomatiques de ACSL et étend les constructions d'E-ACSL en y ajoutant possibilité de vérifier certaine ces définitions. Cette méthode de traduction génère des définitions explicites pures et potentiellement récursives en utilisant des opérateurs usuels (comme des conditions ternaires, des relations, des conjonctions, des disjonctions ainsi que l'introduction de nouvelles variables), ce qui la rend facilement transposable à d'autres langages de programmations.
Cette traduction a été implantée dans le code du greffon E-ACSL écrit en OCaml et contient environ 1300 lignes codes. Elle a été testée manuellement sur un ensemble d'exemples. Ces exemples ont par la suite été rajoutés dans la banque de tests d'intégration du projet Frama-C, qui vérifient notamment l'absence d'incorrection du code C produit par E-ACSL, en faisant de

l'analyse de valeur par interprétation abstraite [15], grâce au greffon EVA [7] de Frama-C.
La traduction allant vers les définitions explicites logiques de E-ACSL bénéficie de toutes les constructions déjà présentes dans E-ACSL mais aussi de ses limitations et celles du noyau de Frama-C. Par exemple, le noyau de Frama-C ne permet pas de lancer et récupéré des erreurs dans les fonctions logiques, ce qui limite la traduction des fonctions explicites qui ne peuvent pas effecteur de retour en arrière. C'est pour cela que l'on change de constructeur uniquement lorsqu'une condition n'est pas vérifiée et qu'on ne récupère pas l'erreur pour effectuer un retour en arrière. Certains types ne sont pour le moment pas utilisable dans les fonctions logiques E-ACSL, c'est le cas les structures C qui aujourd'hui ne sont pas supportés dans la traduction des fonctions logiques, rendant impossible la traduction avec ses types.
Les langages, Mini-Inductive et Mini-Axiomatic proposés peuvent être élargis sur certains opérateurs. On peut y ajouter facilement l'opérateur de conjonction dans des constructeurs d'inductif ou des axiomes d'axiomatiques en les remplaçant directement par des implications, ou encore ajouter l'opérateur de disjonction en créant un nouveau constructeur.
Il est aussi possible de rendre cohérent d'autre de constructeur, pendant la phase de normalisation des égalités (Section 4.1) de résoudre des équations si l'égalité n'est pas cohérente. Par exemple, si on a l'égalité $x + 2 == y + 1$ est rencontré pendant la phase de normalisation et qu'en analysant la cohérence, y devrait être introduit pour rendre le constructeur cohérent, alors il serait possible de remplacer cette égalité par $\texttt{\textbackslash let}\ y = x+1;$. Cette résolution pourrait encore évoluer et permettre de déterminer plusieurs variables en suivant des contraintes étant les conditions précédentes de la prémisse.
La traduction ne permet pas d'utilisé plusieurs relations inductives différentes dans un même inductif. Pour rendre cela possible, il faudrait déterminer le mode à utiliser pour conserver la cohérence du constructeur et y extraire les différentes définitions explicites associées.
La fonction extraite d'une définition axiomatique peut échouer si cette définition est incomplète, cela arrive si des axiomes logiques manquent pour y définir tous ses comportements. Il serait possible de confirmer dans certain cas une génération d'une fonction complète en comparant les conditions des différents axiomes utilisés lors de l'extraction de la fonction logique.
L'extraction ne fait aucune optimisation sur les conditions devant être généré, par conséquent plusieurs même condition ou sa négation qui est testée. Il serait idéal pour E-ACSL de bénéficier d'une phase d'optimisation des définitions logiques lors de la génération de code C.
L'algorithme de traduction est long et complexe, mais il serait de le formaliser à l'aide de Coq et d'y extraire un code de traduction prouvé.

Remerciements

Je remercie mes encadrants Catherine Dubois et Julien Signoles pour les connaissances, leur patience, la liberté qu'ils m'ont accordée ainsi que la confiance qu'ils m'ont témoignée.

Je remercie Thibaut Benjamin pour ses corrections d'erreurs dans le code d'E-ACSL et de l'aide qu'il m'a apportée lors de l'intégration de mon extraction.

Je remercie l'ENSIIE et CEA List de leur accueil.

Je remercie les autres stagiaires Quelen, Félicien, Aurélien et les étudiant de l'ENSIIE, leur soutien et les bons moments partagés.

Ainsi que ma mère étant toujours d'un grand soutien.

Références

[1] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry et Benjamin Werner. « A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses ». In : *CPP*. 2011.

[2] Patrick Baudin, François Bobot, David Buhler, Loic Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles et Nicky Williams. « The Dogged Pursuit of Bug-Free C Programs : The Frama-C Software Analysis Platform ». In : *Commun. ACM* 64.8 (2021), p. 56-68. issn : 0001-0782. doi : 10.1145/3470569.

[3] Patrick Baudin, François Bobot, Loic Correnson, Zaynah Dargaye et Allan Blanchard. « WP Plug-in Manual ». In : *CPP*. 2011.

[4] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy et Virgile Prevostó. « ACSL : ANSI/ISO C Specification Language ». In : url : https://frama-c.com/download/acsl.pdf.

[5] Thibaut Benjamin, Félix Ridoux et Julien Signoles. « Formalisation d'un vérificateur efficace d'assertions arithmétiques à l'exécution ». In : *33èmes Journées Francophones des Langages Applicatifs*. Sous la dir. de Chantal Keller et Timothy Bourke. Saint-Médard-d'Excideuil, France, 2022. url : https://hal.inria.fr/hal-03626779.

[6] Stefan Berghofer et Tobias Nipkow. « Executing Higher Order Logic ». In : *Types for Proofs and Programs*. Sous la dir. de Paul Callaghan, Zhaohui Luo, James McKinna, Robert Pollack et Robert Pollack. Berlin, Heidelberg : Springer Berlin Heidelberg, 2002, p. 24-40.

[7] David Bühler. « Structuring an Abstract Interpreter through Value and State Abstractions :EVA, an Evolved Value Analysis for Frama-C ». Theses. Université de Rennes 1, 2017. url : https://hal.archives-ouvertes.fr/tel-01664726.

[8] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino et Erik Poll. *An overview of JML tools and applications*. 2005. doi : 10.1007/s10009-004-0167-4.

[9] Lori A. Clarke et David S. Rosenblum. « A Historical Perspective on Runtime Assertion Checking in Software Development ». In : *SIGSOFT Softw. Eng. Notes* 31.3 (2006), p. 25-37. issn : 0163-5948. doi : 10.1145/1127878.1127900.

[10] David Delahaye, Catherine Dubois et Pierre-Nicolas Tollitte. « Génération de code fonctionnel certifi a partir de spécifications inductives dans l'environnement Focalize ». In : (2010).

[11] Jean-Christophe Filliâtre et Clément Pascutto. « Ortac : Runtime Assertion Checking for OCaml ». In : *RV'21 - 21st International Conference on Runtime Verification*. Los Angeles, CA, United States, 2021. url : https://hal.inria.fr/hal-03252901.

[12] Reiner Hähnle et Marieke Huisman. « Deductive Software Verification : From Pen-and-Paper Proofs to Industrial Tools ». In : *Computing and Software Science*. 2019.

[13] Nikolai Kosmatov, Fonenantsoa Maurica et Julien Signoles. « Efficient Runtime Assertion Checking for Properties over Mathematical Numbers ». In : *Runtime Verification : 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6–9, 2020, Proceedings*. Los Angeles, CA, USA : Springer-Verlag, 2020, p. 310-322. ISBN : 978-3-030-60507-0. DOI : 10.1007/978-3-030-60508-7_17.

[14] Zoe Paraskevopoulou, Aaron Eline et Leonidas Lampropoulos. « Computing Correctly with Inductive Relations ». In : *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA : Association for Computing Machinery, 2022, p. 966-980. DOI : 10.1145/3519939.3523707.

[15] Xavier Rival et Kwangkeun Yi. *Introduction to Static Analysis : An Abstract Interpretation*. 2020.

[16] Julien Signoleś. « Implementation in Frama-C Plug-in E-ACSL ». In : URL : http://frama-c.com/download/e-acsl/e-acsl-implementation.pdf.

[17] Pierre-Nicolas Tollitte. « Extraction de code fonctionnel certifié à partir de spécifications inductives ». Thèse de doctorat. Conservatoire National des Arts et Métiers, 2013. URL : https://tel.archives-ouvertes.fr/tel-00968607.

A Grammaire de Mini-Inductive

Voici la grammaire de Mini-Inductive.

inductive-def	::=	**inductive** ind-name parameters$^?$	
		{ constuctor* **}**	
ind-name	::=	id	
parameters	::=	**(** parameter (**,** parameter)* **)**	
parameter	::=	type-expr id	
type-expr	::=	logic-type-expr | C-type-name	
logic-type-expr	::=	**integer** | **real** | **boolean**	
constuctor	::=	**case** constructor-name **:**	
		(**forall** binders **;**)$^?$	*univ. quantification*
		ind-pred	
constructor-name	::=	id	
binders	::=	binder (**,** binder)*	
binder	::=	type-expr variable-ident	
		(**,** variable-ident)*	
variable-ident	::=	id	
ind-pred	::=	pred **==>** ind-pred	
	|	**let** variable-ident **=** term **;** ind-pred	*local. binders*
	|	conclusion **;**	
pred	::=	term relop term	*comparison*
	|	id **(** term (**,** term)* **)**	*predicate application*
	|	ind-app	
relop	::=	**<** |**<=** | **==** | **>=** | **>** | **!=**	
ind-app	::=	ind-name **(** term (**,** term)* **)**	*inductive application*
term	::=	id	*variables*
	|	literal	*literal constant*
	|	unary-op term	
	|	term bin-op term	
	|	id (term (**,** term)*)	*function application*
	|	**(** term **)**	*parenthese*
unary-op	::=	**+** | **-**	*unary plus and minus*
bin-op	::=	**+** | **-** | ***** | **/** | **%**	
conclusion	::=	ind-app	

B Grammaire de Mini-Axiomatic

Voici la grammaire du langage Mini-Axiomatic. En bleu se trouve des définitions déjà données dans la grammaire de Mini-Inductive (Annexe A).

axiomatic-decl	::=	**axiomatic** ax-name **{** logic-decl* **}**	
logic-decl	::=	logic-predicate-decl	*ax. predicate declaration*
	|	logic-function-decl	*ax. function declaration*
	|	axiom-def	*axiom definition*
logic-predicate-decl	::=	**predicate** pred-name parameters	
predicate-name	::=	id	
logic-function-decl	::=	**logic** type-exp function-name parameters	
function-name	::=	id	
axiom-def	::=	**axiom** axiom-name **:**	
		(**\forall** binders **;**)$^?$	*univ. quantification*
		axiom-pred	
axiom-name	::=	id	
axiom-pred	::=	pred **==>** axiom-pred	
	|	**\let** variable-ident **=** term **;** axiom-pred	*local. binders*
	|	conclusion **;**	
pred	::=	axiom-predicate-app	*ax. predicate application*
	|	axiom-function-rel	*ax. function relation*
	|	term relop term	*comparison*
	|	id **(** term (**,** term)* **)**	*simple predicate application*
axiom-predicate-app	::=	predicate-name **(** term (**,** term)* **)**	
axiom-function-rel	::=	term **==** axiom-function-app	
	|	axiom-function-app **==** term	
axiom-function-app	::=	function-name **(** term (**,** term)* **)**	
conclusion	::=	axiom-predicate-rel | axiom-function-app	