Génération de code fonctionnel certifié à partir de spécifications inductives dans l'environnement **Focalize**

David Delahaye [*] — **Catherine Dubois** [**] — **Pierre-Nicolas Tollitte** [**]

[*] *CEDRIC/CNAM,*
292 rue Saint-Martin 75141 Paris Cedex 03
David.Delahaye@cnam.fr

[**] *CEDRIC/ENSIIE,*
1 square de la résistance 91025 Évry
dubois@ensiie.fr, tollitte@ensiie.fr

RÉSUMÉ. *Nous proposons une méthode de génération de code fonctionnel à partir de spécifications inductives dans le cadre de l'atelier Focalize. Cette méthode consiste en une analyse préalable de cohérence de mode, qui vérifie si un calcul est possible par rapport aux entrées/sorties sélectionnées, puis en la génération de code proprement dite. Le code produit est certifié en ce sens qu'il est systématiquement accompagné d'une preuve de correction également générée en Focalize.*

ABSTRACT. *We propose a method of generation of functional code from inductive specifications in the framework of the Focalize environment. This method consists of a preliminary analysis of mode consistency, which verifies if a computation is possible with respect to the selected inputs/outputs, and the code generation itself. The obtained code is certified in the sense that it is systematically produced with a proof of correctness, which is also generated in Focalize.*

MOTS-CLÉS : *Génération de code, Spécifications inductives, Code fonctionnel, Preuves de correction, Focalize.*

KEYWORDS: *Code Generation, Inductive Specifications, Functional Code, Correctness Proofs, Focalize.*

1. Introduction

Écrire des spécifications et en extraire automatiquement des programmes n'est pas une idée nouvelle. Les outils Lex et Yacc en sont un bel exemple de mise en œuvre et permettent à de nombreux développeurs de produire rapidement et sûrement (même si il n'existe pas, à notre connaissance, de preuve formelle concernant la correction de ces outils) des analyseurs lexico-syntaxiques à partir de grammaires. Toujours dans le domaine des langages de programmation, d'autres travaux de cette nature ont concerné la génération d'interprètes, de typeurs, de compilateurs à partir de spécifications sémantiques, comme par exemple [19, 11, 14]. Dans un cadre plus général, d'autres chercheurs se sont intéressés au même problème. Les systèmes Coq [18, 17] ou Isabelle [6] proposent des mécanismes généraux d'extraction de code fonctionnel à partir de spécifications et/ou de preuves. Arnould et al. [2] extraient des programmes OCaml à partir de spécifications CASL, Delahaye et al. [13] extraient des programmes OCaml à partir de définitions inductives de prédicats et complètent ainsi l'extraction de l'assistant à la preuve Coq, de la même manière Berghofer et Nipkow [8] proposent d'extraire des programmes ML de spécifications du système Isabelle (ce travail a été refondu dernièrement [7] de manière à produire une spécification équationnelle). Les objectifs de cette génération de code à partir de spécifications concernent généralement le prototypage à des fins d'animation ou de test des spécifications mais aussi à des fins opérationnelles de génération de code en phase avec les spécifications, c'est-à-dire correct par rapport aux spécifications. Concernant ce dernier point, il est à noter qu'une même spécification a l'avantage de pouvoir contenir plusieurs comportements calculatoires, qu'il est possible de sélectionner au moment de la génération de code.

C'est dans cette lignée de travaux que se place le travail présenté ici. Notre objectif est de définir un mécanisme et un outil d'extraction de programmes certifiés à partir de spécifications de propriétés au sein de l'atelier de développement Focalize [24]. Par programmes certifiés, nous sous-entendons des programmes accompagnés de la preuve formelle (vérifiée par Coq) que le programme extrait véri-

fie les spécifications initiales. À la différence de ce qui est proposé dans [13], les programmes extraits le sont dans le langage de Focalize et non directement en OCaml. Plusieurs fonctions peuvent être extraites à partir de la spécification initiale. De plus, cette dernière est formée de propriétés écrites dans le langage de Focalize, donc des formules logiques écrites dans un langage proche du langage des prédicats. Ce travail n'a pas pour but d'introduire dans Focalize la notion de prédicats définis inductivement, mais d'utiliser le langage de spécifications implanté actuellement. Notons également que ce travail est l'occasion d'étudier les liens entre héritage, redéfinition et extraction de fonctions, en particulier l'impact de l'ajout d'une nouvelle propriété dans un composant fils en termes d'extraction de code.

Une des contributions importantes de ce travail est la génération automatique des preuves formelles et mécanisées de correction des fonctions extraites. Dans les travaux précédents de deux des auteurs, la correction est abordée via la correction de l'outil d'extraction. En effet dans [13], la fonction d'extraction est formalisée et sa correction est démontrée à l'aide d'une preuve manuelle. Dans [2], Arnould et al. proposent un mécanisme de synthèse de programmes OCaml à partir de spécifications algébriques CASL. Ils proposent une vérification automatique de la terminaison et de la convergence quand cette vérification est simple et génèrent des obligations de preuve à destination de l'utilisateur dans le cas contraire. Dans les travaux autour de l'extraction de programmes à partir de prédicats définis inductivement dans le cadre d'Isabelle, [8] n'aborde pas formellement la correction et dans [7], la correction de l'approche repose sur la correction du générateur de code [16]. En effet dans ces derniers travaux, le programme extrait est d'abord produit sous la forme d'un ensemble d'équations dont le générateur produit du code fonctionnel. Le générateur voit les programmes générés comme l'implantation d'un système de réécriture : toute séquence de réécriture effectuée par le programme extrait peut être simulée dans la logique, propriété qui assure la correction partielle du générateur.

L'article suit le plan suivant. La section 2 présente brièvement l'environnement Focalize et le langage sous-jacent pour écrire les

spécifications, les implantations et les preuves. Nous présentons ensuite dans la section 3 de manière informelle le mécanisme d'extraction introduit dans cet article. Les sections 4 et 5 sont dédiées à la présentation de la génération de code et de la génération des preuves de correction. Enfin, la section 6 termine l'article en présentant les perspectives de ce travail.

2. L'environnement Focalize

2.1. *Qu'est-ce que* Focalize *?*

Pour comprendre les motivations et surtout les fondements de l'environnement Focalize [15, 3, 24], nous devons remonter au milieu des années 90 avec les discussions informelles qui ont eu lieu au sein du groupe de travail BiP (avec, en particulier, T. Hardin, V. Viguié Donzeau-Gouge et J.-R. Abrial). Avec des experts à la fois en Coq [23] et en B [1], ce groupe a pensé à un langage avec des spécifications plus structurées que les formalisations « plates » faites en Coq et avec une notion de développement incrémental dans l'idée du raffinement de B. De plus, il était important que ce nouveau langage soit fortement typé, probablement d'une manière légèrement moins puissante qu'en Coq, mais plus élaborée que la théorie des ensembles utilisée par B.

Le projet Focalize (initialement projet Foc, puis Focal) a débuté en 1997 sous l'impulsion de T. Hardin et R. Rioboo avec, en particulier, la thèse de S. Boulmé [12]. Dans cette thèse, un nouveau langage a été conçu dans lequel il est possible de construire des applications pas à pas, partant de spécifications abstraites, appelées espèces, vers des implantations concrètes, appelées collections. Ces différentes structures sont combinées en utilisant l'héritage et la paramétrisation, inspirés de la programmation orientée objet. De plus, chacune de ces structures est équipée d'un type support, procurant typiquement un aspect spécification algébrique. Par la suite, V. Prevosto [20] a développé un compilateur pour ce langage, capable de produire du code OCaml [22] pour l'exécution, du code Coq pour la certification, mais aussi du code pour la documentation. Ce compilateur a récemment été réécrit par F. Pessaux dans le cadre

du projet SSURF [25]. Enfin, D. Doligez a également fourni un outil de preuve automatique au premier ordre, appelé Zenon [10], qui aide l'utilisateur à réaliser ses preuves en Focalize au moyen d'un langage de preuve déclaratif. Cet outil de preuve automatique peut produire des preuves Coq, qui peuvent être réinsérées dans les spécifications Coq générées par le compilateur Focalize et vérifiées intégralement par Coq.

2.2. *Spécification : espèce*

La première notion principale du langage Focalize est la structure d'*espèce*, qui correspond au niveau le plus élevé d'abstraction dans une spécification. Une espèce peut être globalement vue comme une liste d'attributs de trois sortes :

– le type support, appelé *représentation*, qui est le type des entités qui sont manipulées par les fonctions de l'espèce ; la représentation peut être soit abstraite soit concrète ;

– les fonctions, qui dénotent les opérations permises sur les entités de la représentation ; les fonctions peuvent être soit des *définitions* (quand un corps est fourni), soit des *déclarations* (quand seul le type est donné) ;

– les propriétés, qui doivent être vérifiées par toute implantation de l'espèce ; les propriétés peuvent être soit simplement des *propriétés* (quand seul l'énoncé est donné), soit des *théorèmes* (quand la preuve est également fournie).

La syntaxe d'une espèce est la suivante :

```
species <name> =
  representation [= <type>];        (* représentation *)
  signature <name> : <type>;        (* déclaration *)
  let <name> = <body>;              (* définition *)
  property <name> : <prop>;         (* propriété *)
  theorem <name> : <prop>           (* théorème *)
  proof = <proof>;
end;;
```

où <*name*> est un nom, <*type*> une expression de type (principalement core-ML sans polymorphisme mais avec les types concrets de données), <*body*> un corps de fonction (principalement core-ML

avec conditionnelle, filtrage et récursion), <*prop*> une proposition (du premier ordre) et <*proof*> une preuve (exprimée au moyen d'un langage de preuve déclaratif). Dans le langage des types, l'expression spécifique « Self » fait référence au type de la représentation et peut être utilisée partout sauf lorsque l'on définit une représentation concrète. De plus, les fonctions ou propriétés d'espèces ou de collections peuvent être référencées en utilisant le préfixe « ! », tandis les fonctions ou propriétés top-level peuvent être utilisées avec le préfixe « # ».

Comme cela a été précisé, les espèces peuvent être combinées en utilisant l'héritage (possiblement multiple), qui fonctionne comme attendu. Il est possible de définir des fonctions qui étaient précédemment seulement déclarées ou de prouver des propriétés qui n'avait pas de preuves. Il est aussi possible de redéfinir des fonctions précédemment définies ou de reprouver des propriétés déjà prouvées. Cependant, la représentation ne peut pas être redéfinie et les fonctions ainsi que les propriétés doivent garder leurs types et propositions respectifs tout au long du chemin d'héritage. Une autre façon de combiner les espèces consiste à utiliser la paramétrisation. Les espèces peuvent être paramétrées soit par des collections (voir la section 2.3) soit par des entités provenant de collections. Si le paramètre est une collection, l'espèce paramétrée a seulement accès à l'interface de cette collection, c'est-à-dire seulement sa représentation abstraite, ses déclarations et ses propriétés. Ces deux mécanismes de combinaison d'espèces complètent la syntaxe précédente comme suit :

```
species <name> (<name> is <name>[(<pars>)],
                <name> in <name>, ...) =
  inherit <name>, <name> (<pars>), ...;
end;;
```

où <*pars*> est une liste de <*name*>, qui dénote les noms utilisés comme paramètres effectifs. Quand le paramètre est une déclaration de paramètre de collection, le mot-clé « is » est utilisé. Quand c'est une déclaration de paramètre d'entité, le mot-clé « in » est utilisé.

2.3. *Implantation : collection*

L'autre notion majeure du langage Focalize est la structure de *collection*, qui correspond à l'implantation d'une spécification. Une collection implante une espèce de manière à ce que chaque attribut devienne concret : la représentation doit être concrète, les fonctions doivent être définies et les propriétés doivent être prouvées. Si l'espèce implantée est paramétrée, la collection doit aussi fournir des implantations pour ces paramètres : soit une collection s'il s'agit d'un paramètre de collection, soit une entité s'il s'agit d'un paramètre d'entité. De plus, une collection est vue (par les autres espèces et collections) à travers son interface correspondante ; en particulier, la représentation est un type abstrait de données et seules les définitions de la collection peuvent manipuler les entités de ce type. Enfin, une collection est un objet terminal, qui ne peut pas être étendu ou raffiné par héritage. La syntaxe d'une collection est la suivante :

```
collection <name> = implement <name> (<pars>) end;;
```

2.4. *Certification :* Zenon

La certification d'une spécification Focalize est assurée par la possibilité de prouver les propriétés. Pour ce faire, un outil de preuve automatique au premier ordre (basé sur la méthode des tableaux), appelé Zenon [10], aide l'utilisateur à construire ses preuves. Le langage permettant d'interagir avec Zenon est un langage de preuve déclaratif. Nous ne décrirons pas ce langage dans cet article, puisque nous ne l'utiliserons pas ; en revanche, le lecteur intéressé peut se référer à [24] pour avoir une description complète de ce langage. Pour faire ses preuves, il est également possible de donner directement une preuve Coq dans le langage de Focalize. Cette option peut s'avérer utile lorsqu'une preuve fait appel à certains aspects logiques que Zenon ne sait pas gérer ; ce sera le cas en section 5, où les preuves décrites doivent être faites par induction. Les preuves Coq sont directement insérées dans le fichier de spécification Coq généré par le compilateur. Elles sont introduites comme suit :

```
theorem <name> : <prop>
proof =
  coq proof
  definition of <name>, ...
  {* <coq> *};
```

où <*name*> est le nom d'une définition utilisée dans la preuve Coq, et <*coq*> une preuve Coq.

3. Présentation informelle

L'extraction, c'est à dire la génération du code, est réalisée au sein d'une espèce Focalize non pas à partir d'un prédicat défini inductivement mais à partir d'une spécification composée de différentes propriétés ayant toutes une conclusion portant sur un même prédicat dont seule la signature est connue. Celles-ci concernent des données de types définis inductivement (encore appelés dans la suite types concrets ou types sommes) par la donnée des constructeurs de valeurs. Les propriétés sont nommées et figurent dans l'espèce courante ou dans une des espèces parentes. Chaque propriété peut comporter une ou plusieurs prémisses et porte sur un prédicat dont la signature est connue (sans aucune définition associée).

Par exemple, la spécification suivante peut être utilisée : elle introduit le prédicat *add* par sa signature, elle utilise le type *nat* défini par ses deux constructeurs *Zero* et *Succ*. Le prédicat *add* est spécifiée à ce stade par les deux propriétés *addZ* et *addS*.

```
type nat = | Zero | Succ (nat);;

species AddSpecif =
  signature add : nat → nat → nat → bool;
  property addZ : all n : nat, add (n, Zero, n);
  property addS : all n m p : nat, add (n, m, p) →
    add (n, Succ (m), Succ (p));
end;;
```

De cette spécification, on peut extraire plusieurs contenus algorithmiques, c'est-à-dire plusieurs fonctions : $add_{12}(n, m)$ qui calcule l'addition de deux entiers naturels, $add_{23}(m, p)$ qui calcule la soustraction de deux entiers naturels et $add_{123}(n, m, p)$ qui teste si $n = m + p$. Pour choisir la fonction à produire, on utilise un mode

qui sélectionne la liste des arguments de la signature qui seront des entrées pour la fonction générée. Un mode apparaît comme la liste des indices des arguments : par exemple le mode $(1, 2)$ signifie que l'on veut considérer le premier et le deuxième argument du prédicat comme des entrées et les autres comme des sorties. Un mode est dit total si tous les arguments sont des entrées ; il est dit partiel si au moins un des arguments est une sortie.

L'extraction est demandée par l'utilisateur en utilisant la nouvelle commande « extract », qui a été ajoutée à Focalize. De manière à séparer spécification et implantation, fût-elle générée automatiquement, l'utilisateur doit définir une nouvelle espèce qui hérite des spécifications. Ainsi, c'est cette sous-espèce qui contiendra la ou les commandes d'extraction. Par exemple, l'espèce *AddImpl* ci-dessous contient deux commandes d'extraction qui synthétisent automatiquement respectivement le code du prédicat et de l'addition de deux entiers :

```
species AddImpl =
  inherits AddSpecif;
  extract add all from (addZ addS) (struct 2);
  extract add12 = add (1, 2) from (addZ addS) (struct 2);
end;;
```

La commande « extract » déclenche l'extraction et peut être vue comme une définition lorsque l'extraction réussit. Cela signifie en particulier que la première commande d'extraction de l'espèce *AddImpl* permet de définir le prédicat *add* (qui n'était que jusque là déclaré), tandis que la seconde commande extraction permet de définir une nouvelle fonction *add_12*. L'utilisateur peut alors utiliser ces fonctions générées comme toute autre fonction définie de Focalize dans d'autres spécifications ou implantations dans la suite de son développement.

Les propriétés qui serviront à l'extraction peuvent être réparties dans différentes espèces, contrairement à un prédicat défini inductivement en Coq où les clauses sont données toutes ensemble. Néanmoins à l'extraction tout se passe comme si on avait donné une définition inductive dont les clauses seraient les propriétés spécifiées à l'extraction.

En Focalize, les fonctions doivent terminer et une preuve de terminaison est attendue. Elle est fournie automatiquement dans le cas de récursion primitive structurelle, mais l'utilisateur doit toutefois préciser l'argument sur lequel la récursion est effectuée, au moyen du mot-clé « struct » suivi de la position de l'argument. Dans les autres cas (récursion bien fondée), la preuve de terminaison est laissée à l'utilisateur sous la forme d'obligations de preuve (qui consistent à montrer que les arguments de l'appel récursif décroissent selon un ordre bien fondé donné). Cette construction est définie par Bartlett dans [5], elle est partiellement implantée dans le nouveau compilateur Focalize. La commande d'extraction proposée dans cet article se limite pour le moment à la récursion structurelle, ce qui permet d'avoir une génération de code et de preuves de correction entièrement automatique.

La commande d'extraction fournit également la preuve de correction de la fonction produite par rapport aux propriétés initiales. Par exemple, dans le cas précédent de la première commande d'extraction de l'espèce *AddImpl*, le théorème de correction consiste à montrer que les propriétés *addZ* et *addS* sont vérifiées en associant à *add* l'implantation générée. Pour la cas de l'extraction en mode $(1, 2)$, il s'agit de vérifier que *add_12 (n, Zero)* est bien égal (au sens de l'égalité structurelle) à n (conformité par rapport à la propriété *addZ*) et que si *add_12 (n, m)* est égal à p, alors *add_12 (n, Succ (m))* est bien égal à *Succ (p)* (conformité par rapport à la propriété *addS*). Dans ce dernier cas, l'utilisateur a la possibilité de nommer ces propriétés de correction dans la commande d'extraction (en tant qu'arguments optionnels). Les théorèmes de correction peuvent eux aussi, tout comme la fonction générée, être utilisés dans la démonstration d'autres propriétés.

Une extraction peut échouer, ce qui entraîne un arrêt de la compilation. Elle peut échouer car le mode ne permet pas d'extraire un code fonctionnel (le flot des données ne permet pas de calculer un résultat), ou car la fonction n'est pas déterministe syntaxiquement, c'est-à-dire que les clauses de définition extraites des propriétés se recouvrent. Ainsi, L'addition ne peut être extraite en mode $(1, 3)$ car *(n, n)* et *(n, Succ (p))* s'unifient. L'extraction en mode (2) est

également impossible car on n'a aucun moyen de calculer m à partir de n et p. Nous rejetons peut-être des définitions à tort mais les accepter demanderait de vérifier la convergence des clauses de définition, ce qui nous amènerait à générer de nouvelles obligations de preuve comme dans [2]. Nous restons ici dans l'optique de fournir un seul résultat et non la liste des résultats possibles comme dans [14] ou [8].

De plus, nous ne considérerons par la suite que les modes partiels avec exactement une seule sortie (les modes partiels à plusieurs sorties pouvant se ramener à des modes partiels à une sortie par curryfication partielle).

4. Génération de code

Comme dit dans la section 2, l'environnement Focalize possède un compilateur produisant du code vers les langages OCaml et Coq. En revanche, aucune fonctionnalité de Focalize ne permettait jusqu'à présent de synthétiser du code directement à partir de spécifications. Dans cette section, nous allons voir comment il est possible de générer du code à partir d'une relation spécifiée dans une espèce Focalize. Par la suite, dans la section 5, nous verrons comment certifier ce code généré, c'est-à-dire montrer qu'il vérifie les propriétés définissant la relation considérée pour l'extraction.

4.1. *Forme des propriétés*

Dans la section 3, nous avons vu qu'en Focalize, une relation est définie par une signature (un nom et un type) et un ensemble de propriétés sur cette signature. Cette signature et ces propriétés doivent avoir la forme suivante :

```
signature f : τ1 → ... → τp → bool;
property prop : all xi : τi,
  f1 (a11, ..., a1p1) → ... → fn (an1, ..., anpn) →
  f (a1, ..., ap);
```

Dans la suite on appelle *prémisses* les parties de la propriété à gauche d'une implication et *conclusion* le prédicat tout à droite.

Les x_i sont des variables associées à des types inductifs τ_i, les f_n sont des fonctions pour lesquelles on a donné un mode d'extraction, et où les a_{jk} peuvent contenir des constructeurs de types inductifs, les variables x_i et des appels de fonctions . Les appels de fonctions ne sont cependant pas tolérées dans les sorties (selon le mode) des prémisses. Par exemple pour l'extraction de f de type $nat \rightarrow nat \rightarrow nat \rightarrow bool$ en mode $(1, 2)$, on ne peut pas utiliser une propriété définie comme suit :

```
all n, m, p : nat, f (n, m, g (p)) → f (n, Succ (m), Succ (p))
```

car *g (p)* est une sortie pour la fonction générée en mode $(1, 2)$.

4.2. *Mode et analyse de cohérence de mode*

Le principe de l'extraction et les algorithmes associés suivent la proposition de Delahaye et al. dans [13], ils ont été adaptés pour le contexte de Focalize.

La notion de mode, initialement introduite dans le cadre de la programmation Prolog, rend explicite le flot de données d'un calcul. Le mode précise les entrées du calcul, les arguments restants sont les sorties du calcul (rangées dans un n-uplet si celles-ci sont multiples). La première étape de notre génération de code, comme dans [7], consiste à vérifier que ce mode est cohérent avec les propriétés dont on extrait le code. L'analyse de cohérence de mode a pour but de vérifier que l'on peut toujours obtenir une sortie à partir des entrées, c'est-à-dire que l'évaluation des prémisses dans le mode considéré peut en effet fournir le ou les résultats attendus. L'analyse de mode peut, si besoin est, procéder à la permutation des prémisses. Elle est réalisée propriété par propriété en analysant les variables utilisées dans chaque prémisse et dans la conclusion et elle dépend bien sûr du mode choisi pour l'extraction. Si l'analyse échoue, la génération sera refusée.

Nous ne détaillons pas ici l'analyse de mode formalisée dans [13], nous l'illustrons sur l'exemple de l'addition en mode $(1, 2)$. On se donne deux fonctions *in* et *out* qui déterminent si les variables utilisées dans une propriété sont des entrées connues ou bien des élé-

ments à calculer pour la fonction que l'on va générer. Elles prennent en argument une prémisse ou la conclusion d'une propriété et le mode avec lequel est effectuée l'analyse. Il s'agit de vérifier que le mode $(1,2)$ est cohérent pour chacune des deux propriétés constituant la spécification, *addZ* et *addS*. Dans la suite p_c désigne la conclusion de la propriété p et p_i désigne la $i^{\text{ème}}$ prémisse de la propriété p.

On a, pour la propriété *addZ* :

– Soit E_0 l'ensemble des variables connues initialement. Ces dernières se déduisent à partir des arguments connus, information donnée par le mode. Ainsi, on obtient : $E_0 = in(addZ_c,(1,2)) = in(add(n, Zero, n),(1,2)) = \{n\}$.

– Le mode est cohérent pour la propriété si les sorties sont calculables à partir des entrées connues, soit ici les éléments de E_0. $out(addZ_c,(1,2)) = out(add(n, Zero, n),(1,2)) = \{n\} \subseteq E_0$, le mode est donc cohérent pour la propriété *addZ*.

et pour la propriété *addS* :

– $E_0 = in(addS_c,(1,2)) =$
$in(add(n, Succ(m), Succ(p)),(1,2)) = \{n, m\}$.

– On s'intéresse maintenant à la prémisse de la propriété. Tout d'abord on vérifie que l'appel de la prémisse, ou plus exactement de la future fonction extraite en mode $(1,2)$, peut se déclencher, c'est-à-dire que les variables de ses arguments sont connues : $in(addS_1,(1,2)) = in(add(n, m, p),(1,2)) = \{n, m\} \subseteq E_0$.
Soit E_1 l'ensemble des variables connues après le calcul extrait de cette prémisse. C'est l'ensemble des variables déjà connues auxquelles on ajoute celles calculées par la prémisse, c'est-à-dire les sorties de la prémisse. On a donc : $E_1 = out(addS_1,(1,2)) = out(add(n, m, p),(1,2)) \cup E_0 = \{p\} \cup E_0 = \{n, m, p\}$.

– Reste à vérifier que la sortie attendue est calculable à partir des variables connues : $out(addS_c,(1,2)) = out(add(n, Succ(m), Succ(p)),(1,2)) = \{p\} \subseteq E_1$. Par conséquent le mode $(1,2)$ est cohérent pour la propriété *addS*.

Après l'analyse de cohérence de mode, une seconde vérification est opérée, celle-ci concerne l'ensemble des conclusions des propriétés. Comme dit précédemment, notre approche consiste à extraire une fonction qui, à un n-uplet d'arguments, associe un résultat unique (et non l'ensemble des résultats possibles). Nous cherchons donc à rejeter les spécifications non déterministes. Une solution peu coûteuse est de vérifier que les arguments des conclusions des propriétés selon le mode choisi ne se recouvrent pas, ou plus techniquement que ces arguments ne sont pas unifiables.

4.3. *Compilation*

L'algorithme de génération exposé dans [13] permet d'extraire une fonction en OCaml à partir d'une définition inductive d'un prédicat. Comme nous l'avons dit précédemment, nous ne partons pas d'un prédicat défini inductivement mais d'un ensemble de propriétés ayant toutes une conclusion portant sur un même prédicat dont seule la signature est connue. C'est la première différence avec [13], cependant les contraintes syntaxiques portant sur les définitions inductives incluent les nôtres. Notons qu'en aucun cas nous ne reconstituons dans le code Coq produit par compilation une définition inductive. La seconde différence importante entre ce travail et l'approche présentée dans [13] concerne le langage cible : dans [13] la génération de code se fait vers OCaml, la fonction extraite peut donc éventuellement boucler. En revanche ici, nous extrayons du code fonctionnel Focalize qui sera par la suite traduit en OCaml et Coq par le procédé habituel de compilation de Focalize. L'environnement Focalize, pour des raisons de cohérence, n'accepte que les fonctions qui terminent, il est donc nécessaire de prouver la terminaison des fonctions Focalize, ce qui bien sûr peut être contourné par l'utilisation de la tactique de preuve « assumed » qui permet d'admettre une propriété. Nous adaptons donc l'algorithme de traduction de [13] pour prendre en compte ces différences. Dans ce travail, nous ne considérons que les cas où la fonction extraite, si elle est récursive, suit un schéma de récursion structurelle. Dans ce cas, nous fournissons automatiquement la preuve de terminaison, à partir des informations données par l'utilisateur dans la commande

d'extraction (position de l'argument de récursion structurelle indiquée par « struct »). Dans les autres cas de récursion, l'utilisateur doit donc fournir cette preuve de terminaison s'il veut produire une collection.

Par conséquent, dans notre contexte, c'est-à-dire celui de la récursion structurelle, nous ne préoccupons pas dans la génération de code de la preuve de terminaison.

Bien que les fonctions générées n'apparaissent pas sous leur forme textuelle dans le code source mais uniquement dans l'arbre de syntaxe abstraite correspondant, nous montrons dans la suite leur forme décompilée.

Ainsi, la fonction générée suivra la syntaxe suivante :

```
let rec f (x1 in τ1, ..., xn in τn) (struct xi) : τ =
  (* corps de f *);
```

où les τ_i, avec $i = 1 \ldots n$, sont les types des arguments et τ le type du résultat.

L'indication concernant l'argument de décroissance est extraite des arguments de la commande d'extraction. Cette fonction est compilée par le compilateur Focalize en un « let rec » de OCaml et un « Fixpoint » de Coq.

Le schéma de compilation utilisé dans [13] utilise des gardes dans les clauses de filtrage. Le langage Focalize n'offre pas cette possibilité. Néanmoins dans notre cadre, grâce à la contrainte de non recouvrement des conclusions des propriétés, et donc des motifs du filtrage de la fonction générée, ces gardes peuvent se traduire à l'aide de constructions « if then/else ».

La traduction est adaptée de celle présentée et formalisée dans [13]. Celle-ci distingue deux schémas, selon que le mode est total ou partiel. Nous l'illustrons sur l'exemple de *add* avec le mode $(1, 2)$.

La traduction part de la liste des propriétés (composée ici de *addZ* et *addS*), de la position de l'argument décroissant (2 dans notre cas), et de l'environnement de modes $\mathcal{M}$ qui associe aux prédicats concernés par l'extraction, leur mode respectif et le nom de la

fonction extraite associée (réduit à celui qui associe à *add* le mode $(1,2)$ et le nom *add12*).

Puisque *add* apparaît dans les prémisses des propriétés, la fonction générée sera récursive. Donc la fonction aura la forme :

```
let rec add12 (p1, p2) (struct p2) =
  match (p1, p2) with
  (* traduction des propriétés *);
```

Chaque propriété donnera lieu à une clause de filtrage, le motif est constitué des termes correspondant aux entrées désignées par le mode du prédicat, donc ici les termes 1 et 2. Lorsque les termes ne sont pas linéaires, les variables sont renommées de manière à ce que les termes deviennent linéaires et les contraintes liant les variables apparaissent en tant que condition d'une expression conditionnelle dans la partie droite de la clause (ceci est possible car les motifs ne se recouvrent pas). Dans le cas de l'extraction en mode $(1,2)$ les termes sont linéaires, mais ce n'est pas le cas pour l'extraction en mode total (voir la traduction ci-dessous).

Les propriétés peuvent être traitées dans n'importe quel ordre (puisqu'elles ne se recouvrent pas). Ici on traite d'abord *addZ* puis *addS*. Le squelette devient alors :

```
let rec add12 (p1, p2) (struct p2) =
  match (p1, p2) with
  | (n, Zero) → (* traduction de la prémisse 1 *)
  | (n, Succ (m)) → (* traduction de la prémisse 2 *)
```

Les parties droites sont constituées des traductions des prémisses et du résultat attendu (terme en position de sortie), c'est-à-dire schématiquement :

```
match f1 (t11, ..., t1n1) with
| out1 →
  (match f2 (t21, ..., t2n2) with
   | out2 →
     (... → (* résultat de la propriété *) ...))
```

où f_k est le nom de la fonction générée pour le prédicat de la prémisse k, $t_{k1} \dots t_{kn_k}$ les termes en entrée dans la prémisse k, et out_k le terme de sortie dans la prémisse k (voir la traduction ci-après lorsque la prémisse n'a pas de terme de sortie, c'est-à-dire lorsque son mode est total).

Le fragment manquant de la prémisse 1 est immédiat, en effet la propriété n'a pas de prémisse, il suffit donc de traduire que le résultat est le terme en position sortie dans la conclusion de la propriété, ici *n*. Pour la prémisse 2, on suit le schéma donné plus haut avec une seule prémisse récursive.

On obtient donc le code final :

```
let rec add12 (p1, p2) (struct p2) =
  match (p1, p2) with
  | (n, Zero) → n
  | (n, Succ (m)) →
    (match add12 (n, m) with
     | p → (Succ (p)));
```

Pour le mode total, on obtient le code ci-dessous. On remarquera que le premier motif n'étant pas linéaire, il est traduit à l'aide d'un « if then/else » ; nous avons vu précédemment que c'est possible puisque les conclusions des propriétés définissant la relation *add* ne se recouvrent pas en mode $(1, 2)$.

```
let rec add (p1, p2, p3) (struct p2) =
  match (p1, p2, p3) with
  | (n, Zero, n0) → if (n0 = n) then true else false
  | (n, Succ (m), Succ (p)) →
    if add (n, m, p) then true else false
  | _ → false;
```

Il est à noter que dans [13], les filtrages sont systématiquement complétés par un cas par défaut (renvoyant *false* dans le cas d'un mode total et une exception dans le cas d'un mode partiel). Il n'est pas possible de faire de même dans le cadre de l'environnement Focalize, puisque certains des langages cibles de son compilateur, notamment Coq, imposent des contraintes fortes sur le filtrage. En particulier, il n'est permis ni d'écrire un filtrage non exhaustif, ni de compléter un filtrage exhaustif. Cela signifie que la synthèse de code vue précédemment doit s'appuyer sur une analyse de l'exhaustivité du filtrage afin de déterminer si un cas par défaut doit être ajouté ou non.

4.4. *Implantation*

Comme dit dans la section 2, le compilateur de l'environnement Focalize est un compilateur de haut niveau, dans la mesure où il produit du code pour des langages de haut niveau, à savoir OCaml et Coq. Dans ce schéma de compilation, le fichier source est traduit vers un arbre de syntaxe abstraite (AST), qui est ensuite typé avant d'être utilisé par deux traducteurs indépendants produisant du code OCaml et du pseudo-Coq. Le pseudo-Coq contient des trous pour les preuves, qui doivent être comblés par Zenon, l'outil de preuve automatique de Focal. La figure 1 montre comment le mécanisme de synthèse de code décrit précédemment s'intègre au schéma de compilation de Focalize. Le code de la fonction extraite est directement produit sous forme d'un AST avant la phase de typage. Cet AST est ensuite intégré à l'AST global de la spécification : il est substitué à la commande d'extraction. Ainsi, la génération de code peut être réalisée en une seule passe de compilation, sans que le compilateur n'ait à savoir typer de nouvelles constructions.

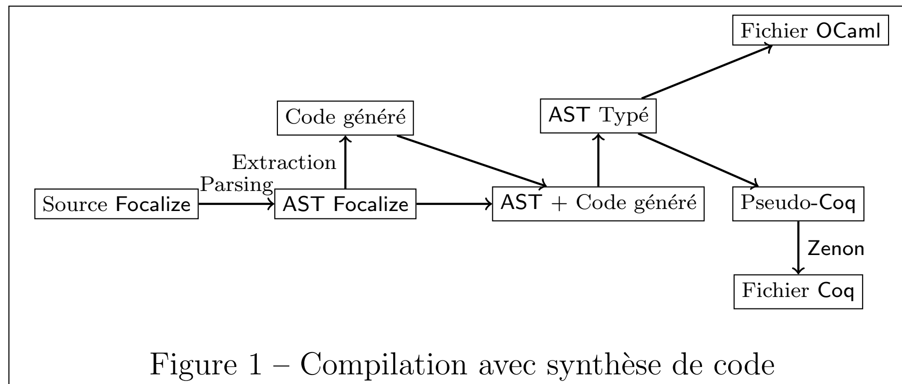

Figure 1 – Compilation avec synthèse de code

5. Génération des preuves de correction

Le mécanisme d'extraction que nous proposons dans cet article permet non seulement de synthétiser du code fonctionnel à partir d'un ensemble de propriétés, mais aussi de produire des preuves de correction du code extrait (à savoir que le code extrait vérifie bien les propriétés initiales ou une forme dérivée de celles-ci, selon

le mode). Ces preuves de correction permettent d'étendre le cadre d'utilisation du code généré aux applications qui nécessitent un niveau de sûreté important. Même s'il s'agit seulement d'animer une spécification et d'utiliser le code extrait comme oracle pour valider des tests, ces preuves de correction permettent de garantir un certain niveau de confiance dans l'oracle. Dans cette section, nous allons décrire, au moyen d'exemples, comment ces preuves de correction sont générées, en distinguant notamment deux cas selon que le mode est total ou partiel.

5.1. *Preuves en mode total*

Lorsque l'extraction a lieu en mode total, le code extrait est utilisé pour définir la fonction représentant la relation (voir section 3). Ainsi, pour montrer la correction de ce code extrait, il suffit simplement de montrer les propriétés définissant la relation. Par exemple, si nous reprenons le cas de la relation *add* introduite dans l'espèce *AddSpecif* (voir section 3), et en particulier la commande d'extraction en mode total (mot-clé « all ») invoquée dans la sous-espèce *AddImpl*, cette dernière commande génère les preuves des propriétés *AddZ* et *AddS* comme suit :

```
theorem addZ : all n in nat, add (n, Zero, n);
proof = coq proof
  {* intro n; simpl; generalize (basics.beq_refl nat__t n);
     case (basics._equal_ nat__t n n); auto. *};

theorem addS : all n m p in nat, add (n, m, p) →
  add (n, Succ (m), Succ (p));
proof = coq proof
  {* intros n m p; simpl; case (add n m p); auto. *};
```

Il est tout d'abord à noter que d'un point de vue conceptuel, ces preuves sont générées automatiquement par le compilateur et que l'utilisateur ne les voit donc pas (le code précédent est donc une version décompilée, qui nous permet d'expliciter la génération de ces preuves de correction). On peut également remarquer que ces preuves sont directement des preuves Coq, qui sont insérées telles quelles dans le fichier de spécification Coq généré par le compilateur, et qui n'utilisent donc pas l'outil de preuve automatique Zenon. La

raison essentielle réside dans le fait que ces preuves se font par induction et que Zenon ne sait pas encore gérer les preuves par induction.

Pour comprendre comment ces preuves sont générées, il est nécessaire d'introduire le contexte dans lequel elles sont censées s'insérer. Ce contexte est le fichier de spécification Coq généré par le compilateur. Ainsi, dans ce fichier, la fonction *add* extraite (voir section 4) est traduite par la fonction Coq suivante :

```
Fixpoint add (a1 a2 a3 : (nat__t)%type) {struct a2} :
  basics.bool__t :=
  match (a1, a2, a3) with
  | (n, Zero, n0) ⇒
    if (basics._equal_ _ n0 n) then true else false
  | (n, Succ m, Succ p) ⇒ if (add n m p) then true else false
  | _ ⇒ false
  end.
```

où *nat__t* et *basics.bool__t* représentent respectivement les types *nat* et *bool* de Focalize (les noms de leurs constructeurs sont inchangés), tandis que *basics._equal_* correspond à l'égalité de Focalize.

Dans ce contexte, les théorèmes de correction sont introduits comme suit (les preuves Coq proviennent de la version compilée de la spécification Focalize, mais n'ont de sens que dans ce contexte) :

```
Theorem addZ : forall n : nat__t, Is_true (add n Zero n).
Proof.
  intro n; simpl; generalize (basics.beq_refl nat__t n);
  case (basics._equal_ nat__t n n); auto.
Save.

Theorem addS : forall n m p : nat__t,
  Is_true (add n m p) → Is_true (add n (Succ m) (Succ p)).
Proof.
  intros n m p; simpl; case (add n m p); auto.
Save.
```

où *Is_true* est une fonction prédéfinie de Coq permettant d'injecter les booléens dans l'ensemble des propositions et *basics.beq_refl* le théorème de réflexivité de *basics._equal_*.

On peut remarquer que ces deux preuves de correction obéissent au même schéma de preuve, qui consiste à exécuter la fonction *add*, puis à destructurer les « if then/else » dans les branches du filtrage

(le « generalize » dans la preuve de *addZ* ne sert qu'à introduire la réflexivité de l'égalité et pourrait être supprimé si le théorème correspondant était ajouté à la base des théorèmes de la tactique « auto »).

5.2. *Preuves en mode partiel*

Lors d'une extraction en mode partiel, le code extrait est une nouvelle fonction ajoutée à l'espèce contenant la commande d'extraction (voir section 3). Ainsi, la correction de ce code extrait est assuré par de nouvelles propriétés que nous devons générer en nous aidant des propriétés considérées pour l'extraction. En reprenant l'exemple de la relation *add* de l'espèce *AddSpecif* (voir section 3), et en considérant une extraction en mode $(1,2)$ (fonction *add12*), les preuves de correction générées sont les suivantes :

```
theorem add12_addZ : all n : nat, add (n, Zero, add12 (n, Zero))
proof = coq proof {* intro n; simpl; apply addZ. *};

theorem add12_addS : all n m p : nat, add (n, m, add12 (n, m)) →
  add (n, Succ (m), add12 (n, Succ (m)))
proof = coq proof
  {* intros n m p H; simpl; apply addS; auto. *};
```

De même que précédemment, ces preuves de correction n'ont de sens que dans le contexte du fichier Coq généré par le compilateur. Dans ce fichier, la fonction *add12* extraite (voir section 4) est traduite comme suit :

```
Fixpoint add12 (a1 a2 : (nat__t)%type) {struct a2} : nat__t :=
  match (a1, a2) with
  | (n, Zero) ⇒ n
  | (n, Succ m) ⇒
    match (add12 n m) with
    | p ⇒ (Succ p)
    end
  end.
```

Dans ce contexte, les théorèmes de correction sont les suivants (comme pour le mode total, les preuves Coq proviennent de la version compilée de la spécification Focalize et sont insérées telles quelles dans ce nouveau contexte) :

```
Theorem add12_addZ : forall n : nat__t,
  Is_true (add n (Zero) (add12 n Zero)).
Proof.
  intro n; simpl; apply addZ.
Save.

Theorem add12_addS : forall n m p : nat__t,
  Is_true (add n m (add12 n m)) ->
  Is_true (add n (Succ m) (add12 n (Succ m))).
Proof.
  intros n m p H; simpl; apply addS; auto.
Save.
```

De même que pour les preuves de correction pour le mode total, ces preuves suivent un même schéma de preuve, qui consiste à exécuter la fonction *add12*, puis à appliquer la propriété correspondante (parmi celles considérées dans la commande d'extraction).

5.3. *Implantation*

Afin de générer les preuves de correction, le schéma de compilation de l'environnement Focalize est modifié comme le montre la figure 2. Comme cela a été précisé auparavant, nous n'utilisons pas le langage de preuve qui permet d'interagir avec Zenon (puisque les preuves de correction générées sont des preuves par induction et que Zenon ne gère pas encore l'induction), mais directement le langage de preuve de Coq. Ces preuves Coq sont directement introduites dans l'AST de Focalize sous la forme d'une chaîne de caractères et ensuite transmises telles quelles à Coq (la passe obligatoire par Zenon n'a aucun effet puisqu'aucune directive ne lui est transmise dans le pseudo-Coq intermédiaire).

6. Conclusion

Dans cet article, nous avons proposé une méthode de génération de code fonctionnel à partir de spécifications dans le cadre de l'environnement Focalize. Par rapport à [13] qui présentait une méthode d'extraction dans le cadre du calcul des constructions inductives (avec une implantation dans le système Coq), l'originalité de ce travail réside essentiellement dans le fait que le code extrait est gé-

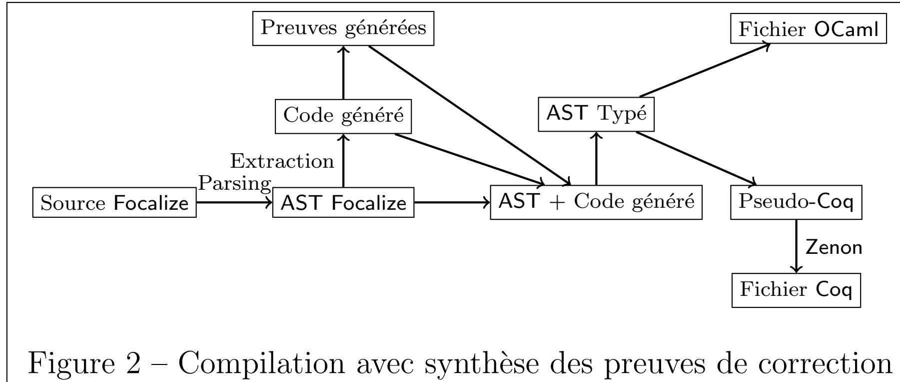

Figure 2 – Compilation avec synthèse des preuves de correction

néré dans le même langage de spécification (il ne s'agit pas d'un langage cible d'extraction particulier comme dans [13]). Ainsi, ceci permet non seulement d'animer les spécifications au sein du même formalisme, mais aussi et surtout de certifier le code généré (ce qui n'est pas possible lorsque le langage cible d'extraction est distinct du langage de spécification). En revanche, le langage de spécification impose un certain nombre de contraintes et la mise en œuvre d'une procédure d'extraction dans ce langage est plus complexe. En particulier, la propriété de normalisation forte du langage de spécification impose une garantie de terminaison des fonctions extraites. Dans cet article, nous avons également montré qu'en se limitant à une récursion structurelle, la génération du code extrait et des théorèmes de correction est entièrement automatique (modulo l'indication par l'utilisateur de l'argument de récursion). Enfin, il est à noter que ce mécanisme de génération de code fonctionnel certifié a été implanté et intégré à la dernière version du compilateur de Focalize.

Ce travail offre plusieurs perspectives. Parmi celles-ci, il y a en particulier l'extension du mécanisme d'extraction à la récursion bien fondée (par opposition à la récursion structurelle). Cette extension devrait s'avérer assez directe dans la mesure où le formalisme nécessaire sous-jacent a déjà fait l'objet de travaux [5], et a notamment été développé dans le compilateur de Focalize. Ce formalisme repose principalement sur un mécanisme du système Coq qui fournit une aide à l'utilisateur dans l'écriture de fonctions récursives géné-

rales [4]. Outre une fonction, ce mécanisme fournit par ailleurs un schéma d'induction fonctionnel, que nous devrions pouvoir utiliser pour générer automatiquement les théorèmes de correction.

Une autre perspective de travail, plus conceptuelle, consiste à comprendre comment ce mécanisme d'extraction s'intègre à l'environnement orienté objet offert par Focalize et comment il interagit en particulier avec l'héritage. En effet, contrairement à un système comme Coq par exemple, la définition du prédicat dont nous souhaitons extraire du code, n'est pas close et peut être étendue par héritage. Cela ajoute une certaine flexibilité au langage de spécification (dans d'autres systèmes, notamment Coq, où toute définition inductive est systématiquement clôturée, ce mécanisme peut être difficile à mettre en œuvre ; voir [9] en particulier), mais pose également la question de savoir si l'ajout d'une nouvelle propriété par héritage doit invalider ou non une définition obtenue par extraction, ainsi que ses théorèmes de correction. En effet, si l'extraction s'est faite en mode total, il semble effectivement raisonnable d'invalider la définition, puisqu'il est peu probable que l'ancienne définition vérifie la nouvelle clause définissante. En revanche, avec un mode partiel d'extraction, la fonction extraite est une nouvelle définition ajoutée à l'espèce contenant la commande d'extraction et rien n'oblige a priori cette fonction à vérifier cette nouvelle clause ; dans ce cas, l'invalidation de la définition extraite n'est donc pas forcément la solution la plus appropriée. De toutes les façons, en cas d'invalidation, il est à noter que nous serons confrontés à des problématiques similaires à celles abordées dans [21] (avec une ampleur plus importante puisqu'une invalidation concernera non plus seulement les preuves, mais également les définitions), où il s'agira de comprendre quel est le meilleur moment pour réaliser une extraction et de proposer pour ce faire de nouveaux motifs de conception.

Références

[1] J.-R. Abrial. *The B Book, Assigning Programs to Meanings.* Cambridge University Press, Cambridge (UK), 1996. ISBN 0521496195.

[2] A. Arnould, L. Fuchs, M. Aiguier, and T. Brunet. Automatic Generation of Functional Programs from CASL Specifications. In *International Conference on Software Engineering Advances (ICSEA)*, page 34, Papeete (Tahiti, French Polynesia), Oct. 2006. IEEE CS Press.

[3] P. Ayrault, M. Carlier, D. Delahaye, C. Dubois, D. Doligez, L. Habib, T. Hardin, M. Jaume, C. Morisset, F. Pessaux, R. Rioboo, and P. Weis. Trusted Software within Focal. In *Computer & Electronics Security Applications Rendez-Vous (C&ESAR)*, pages 142–158, Rennes (France), Dec. 2008.

[4] G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and Reasoning About Recursive Functions : A Practical Tool for the Coq Proof Assistant. In *Functional and Logic Programming (FLOPS)*, volume 3945 of *LNCS*, pages 114–129, Fuji Susono (Japan), Apr. 2006. Springer.

[5] W. Bartlett. Compilation des fonctions récursives dans l'atelier Focal. Mémoire d'ingénieur, ENSIIE, 2007. À paraître sous la forme d'un rapport CEDRIC.

[6] S. Berghofer. Program Extraction in Simply-Typed Higher Order Logic. In *Types for Proofs and Programs (TYPES)*, volume 2646 of *LNCS*, pages 21–38, Berg en Dal (Netherlands), Apr. 2002. Springer.

[7] S. Berghofer, L. Bulwahn, and F. Haftmann. Turning Inductive into Equational Specifications. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, pages 131–146, Munich (Germany), Aug. 2009. Springer.

[8] S. Berghofer and T. Nipkow. Executing Higher Order Logic. In *TYPES*, volume 2277 of *LNCS*, pages 24–40, Durham (UK), Dec. 2000. Springer.

[9] O. Boite. Proof Reuse with Extended Inductive Types. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3223 of *LNCS*, pages 50–65, Park City (Utah, USA), Sept. 2004. Springer.

[10] R. Bonichon, D. Delahaye, and D. Doligez. Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs.

In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 4790 of *LNCS/LNAI*, pages 151–165, Yerevan (Armenia), Oct. 2007. Springer.

[11] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur : The System. In *Practical Software Development Environments (PSDE)*, volume 24(2) of *SIGPLAN Notices*, pages 14–24, Boston (MA, USA), Nov. 1988. ACM Press.

[12] S. Boulmé. *Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel*. PhD thesis, Université Pierre et Marie Curie (Paris 6), Dec. 2000.

[13] D. Delahaye, C. Dubois, and J.-F. Étienne. Extracting Purely Functional Contents from Logical Inductive Types. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 4732 of *LNCS*, pages 70–85, Kaiserslautern (Germany), Sept. 2007. Springer.

[14] C. Dubois and R. Gayraud. Compilation de la sémantique naturelle vers ML. In *Journées Francophones des Langages Applicatifs (JFLA)*, Morzine-Avoriaz (France), Feb. 1999.

[15] C. Dubois, T. Hardin, and V. Viguié Donzeau-Gouge. Building Certified Components within Focal. In *Trends in Functional Programming (TFP)*, volume 5, pages 33–48, Munich (Germany), Nov. 2004. Intellect (Bristol, UK).

[16] F. Haftmann and T. Nipkow. A Code Generator Framework for Isabelle/HOL. Technical Report 364/07, Department of Computer Science, University of Kaiserslautern, Sept. 2007.

[17] P. Letouzey. A New Extraction for Coq. In *TYPES*, volume 2646 of *LNCS*, pages 200–219, Berg en Dal (The Netherlands), Apr. 2002. Springer.

[18] C. Paulin-Mohring and B. Werner. Synthesis of ML Programs in the System Coq. *Journal of Symbolic Computation (JSC)*, 15(5/6) :607–640, May 1993.

[19] M. Pettersson. A Compiler for Natural Semantics. In *Compiler Construction (CC)*, volume 1060 of *LNCS*, pages 177–191, Linköping (Sweden), Apr. 1996. Springer.

[20] V. Prevosto. *Conception et implantation du langage Foc pour le développement de logiciels certifiés.* PhD thesis, Université Pierre et Marie Curie (Paris 6), Sept. 2003.

[21] V. Prevosto and M. Jaume. Making Proofs in a Hierarchy of Mathematical Structures. In *Calculemus*, pages 89–100, Roma (Italy), Sept. 2003. LIP6.

[22] The Caml Development Team. *Objective Caml, version 3.11.1.* INRIA, June 2000. http://caml.inria.fr/.

[23] The Coq Development Team. *Coq, version 8.2.* INRIA, Feb. 2009. http://coq.inria.fr/.

[24] The Focalize Development Team. *Focalize, version 0.1.0.* CNAM, INRIA, and LIP6, May 2009. http://focalize.inria.fr/.

[25] The SSURF Project, 2007. http://www-spi.lip6.fr/~jaume/ssurf.html.