Vers un développement formel non incrémental

Thi-Kim-Dung Pham[1,3], Catherine Dubois[2], Nicole Levy[1]

1. Conservatoire National des Arts et Métiers, lab. Cedric, Paris

2. ENSIIE, lab. Samovar, Évry

3. University of Engineering and Technology, Vietnam National University, Hanoi

Résumé

Modularité, généricité, héritage sont des mécanismes qui facilitent le développement et la vérification formelle de logiciels corrects par construction en permettant de réutiliser des spécifications, du code et/ou des preuves. Cependant les lignes de produits exploitent d'autres techniques de réutilisation ou de modification graduelle. Les méthodes formelles permettant la production de code correct par construction (en B ou FoCaLiZe par exemple) ne sont pas bien adaptées à la variabilité telle qu'elle apparaît dans les lignes de produits. Nous proposons d'approcher ce problème par la définition d'un langage formel, GFML, proche de la variabilité mise en œuvre dans les lignes de produits permettant de spécifier, implanter et prouver. Ce langage est compilé vers un formalisme existant, ici FoCaLiZe. Cet article illustre par l'exemple une des constructions offertes par GFML ainsi que sa traduction en FoCaLiZe.

1 Introduction

Nous aimerions construire des programmes corrects par construction, c'est-à-dire qui répondent au cahier des charges, sont cohérents et complets. Il est effectivement possible de le faire à l'aide de certaines méthodes qui utilisent des langages formels tels que B [1] ou FoCaLiZe [6]. Il faut alors définir les propriétés que le programme doit vérifier et utiliser des outils de preuve pour prouver que l'on a bien suivi toutes les étapes de la méthode.

Un des principaux problèmes de ces méthodes réside dans la difficulté à les appliquer. On pourrait espérer que les méthodes de réutilisation facilitent le développement. Il existe plusieurs manières de réutiliser un programme. Par exemple, les méthodes orientées objets qui proposent de compléter ou de modifier, par héritage, une classe d'objets. Le langage FoCaLiZe possède l'héritage et permet ainsi de réutiliser par enrichissement.

Mais ce n'est pas suffisant. On aimerait réutiliser une fonctionnalité d'un programme mais pour l'utiliser dans un contexte différent. Il existe des mécanismes permettant de décrire les propriétés d'un contexte, indispensables pour un programme, sous la forme de paramètres de ce programme. En FoCaLiZe, les composants s'appellent des espèces (*species*) et peuvent être paramétrés par des espèces représentant des objets ou des propriétés du contexte requis.

Mais ce n'est pas encore suffisant. L'utilisation de la paramétrisation se révèle assez difficile et restrictive.

En fait, pour réutiliser un programme (ou une partie d'un programme), il faut savoir comment il a été construit et connaître les décisions prises lors de son développement.

Le problème qui se pose est l'expression des savoirs-faire et leur réutilisation. La réalité, c'est qu'il est bien difficile de décrire ou de formaliser ce savoir-faire. Mais c'est un des avantages des approches lignes de produits [3] qui permettent de décrire la succession des décisions à prendre et d'associer à chaque décision une solution concrète sous la forme d'un morceau de code (*asset*). En général, les décisions prises sont constructives : il s'agit d'ajouter une information, une fonctionnalité, une propriété, etc. Dans ce cas, la composition des morceaux peut être définie simplement.

Mais qu'en est-il lorsque la décision prise à une étape est de modifier l'étape précédente et non seulement de rajouter quelque chose ? Comment formaliser des méthodes de réutilisation non incrémentales ?

Il existe des approches telle que celle de Thomas Thüm [12, 11] qui utilise les lignes de produits pour développer des programmes Java selon une approche par contrats à l'aide de pré et post-conditions. Il utilise le langage JML pour écrire les contrats et les traduit ensuite en Coq pour les vérifier. C'est de là qu'est venue l'idée d'introduire un langage dédié pour décrire la construction des modifications à apporter au code.

Notre proposition est de faciliter de telles étapes de développement, permettant de modifier une étape précédente, c'est-à-dire de modifier le code préalablement défini, et prouvé. L'idée est d'introduire un langage d'expression de ces modifications et un compilateur réalisant la modification à opérer et générant le code. Le langage que nous proposons s'appelle GFML. Couplée à une approche lignes de produits, notre démarche permet de décrire des savoirs-faire qui ne sont pas purement incrémentaux mais peuvent passer par des étapes de remise en cause de développements préalables, tout en bénéficiant de l'approche de développement de code correct par construction.

Nous avons choisi d'illustrer notre démarche dans le cadre proposé par FoCaLiZe. En effet, il permet de définir à la fois spécifications, code et preuves, donc tous les artefacts associés à une caractéristique d'une ligne de produits. D'autre part, FoCaLiZe admet pour seul mécanisme de raffinement, l'enrichissement par héritage, ce qui laisse beaucoup de liberté.

La suite de cet article est structurée de la manière suivante. Dans un premier temps nous allons étudier les constructions existantes en FoCaLiZe qui permettent de suivre un développement formel incrémental. Nous signalerons aussi les limites de ce modèle. Nous montrons à l'aide d'un exemple de compte bancaire un exemple de modification souhaitée, à savoir le renforcement d'une fonctionnalité tout en conservant au maximum les preuves déjà faites. Cette modification peut être exprimée en FoCaLiZe à l'aide de l'utilisation conjointe de l'héritage et de la paramétrisation. Nous présentons ensuite l'expression de cette modification dans notre langage GFML et sa traduction en FoCaLiZe. Un avantage de cette approche est le fait qu'elle s'adapte très bien aux lignes de produits puisque la composition des morceaux associés à chaque étape entraîne une suite de modifications et non seulement des enrichissements. L'approche générale est présentée dans [7], nous détaillons ici, sur un exemple, une des constructions offertes par GFML et sa compilation en FoCaLiZe.

2 Le modèle FoCaLiZe

L'environnement FoCaLiZe (anciennement FoCal) [6, 5] permet le développement de programmes certifiés pas à pas, de la spécification à l'implantation. Cet environnement propose un langage également nommé FoCaLiZe et des outils d'analyse de code,

de preuve (Zenon) et des compilateurs vers Ocaml, Coq et Dedukti [4]. Le langage FoCaLiZe permet d'écrire des propriétés en logique du premier ordre, des signatures et des fonctions dans un style fonctionnel et enfin des preuves dans un style déclaratif. Ce langage offre également des mécanismes inspirés par la programmation orientée objets, comme l'héritage, la liaison retardée et la redéfinition afin de faciliter la modularisation et la réutilisation. FoCaLiZe permet d'hériter de spécifications, de code mais aussi de preuves [8]. En présence d'une redéfinition, un calcul de dépendances permet de déterminer quelles sont les preuves qui sont encore valides et celles qui sont invalidées par la redéfinition et doivent donc être refaites. Un mécanisme de paramétrisation vient compléter l'ensemble.

Les constructions de FoCaLiZe [2] permettent d'ajouter à une espèce, une signature, une définition, une propriété, une preuve, de définir une représentation concrète pour les objets manipulés, de redéfinir une fonction. Dans ce dernier cas, le calcul des dépendances fait par le compilateur détermine si les preuves déjà effectuées concernant la fonction redéfinie doivent être refaites ou non. Ces primitives assurent une construction incrémentale des spécifications, des programmes et des preuves. Dans le modèle actuel de FoCaLiZe, par héritage, les spécifications d'une fonction redéfinie sont héritées, elles peuvent être complétées par de nouvelles propriétés mais elles ne peuvent pas être *redéfinies* (en restreignant par exemple la précondition comme dans la section suivante), voire supprimées comme on le désirerait par exemple pour développer les fonctionnalités du logiciel en mode dégradé. La redéfinition mise en œuvre dans FoCaLiZe impose également de conserver l'arité d'une fonction, il n'y a donc pas de possibilité de surcharge. Enfin, la représentation d'un type (mot-clé `representation` dans les exemples de la section 3), une fois définie dans un composant C, ne peut être redéfinie dans les composants qui héritent de C. Ceci conduit à un choix tardif de cette représentation. Mais dans le cadre d'un développement non incrémental, on peut être amené à modifier la représentation choisie, par exemple en ajoutant des attributs à la manière des langages orientés objets. Une étude similaire a été proposée dans le cadre de la programmation orientée *feature* et de la programmation par contrats par Thüm et al. dans [10]. Relativement à la classification proposée dans cet article, le modèle actuel de FoCaLiZe, en cas d'héritage, met en œuvre le raffinement de spécifications par sous-typage. Avec l'exemple ci-dessous, nous nous rapprochons du raffinement dit explicite et du raffinement par surcharge de contrats.

3 Raffinement d'une propriété

Nous utilisons dans la suite l'exemple (inspiré de [12]) de la ligne de produits concernant la gestion de comptes bancaires représentée graphiquement (par son diagramme de caractéristiques ou *features*) à la figure 1. La feature racine, BankAccount (ou BA), définit le fonctionnement de base d'un compte : calculer le solde d'un compte, débiter et créditer un compte. Le débit n'est autorisé que si le compte reste créditeur d'une certaine somme (ici `over`). A ces fonctionnalités élémentaires peuvent être ajoutées, de manière optionnelle, les fonctionnalités dénotées par les features DailyLimit (DL), LowLimit (LL) et Currency. DL introduit une limitation pour les retraits. Dans la suite, seuls BA et DL sont utilisés.

A chaque feature de l'arbre est associé un fichier FoCaLiZe contenant des spécifications, du code et des preuves. Les artefacts associés au feature racine, BA, définissent le noyau minimal. Le fichier FoCaLiZe associé contient trois espèces données ci-dessous, `BA_spec0`, `BA_spec` et `BA_impl`, les deux premières contenant la spécification et

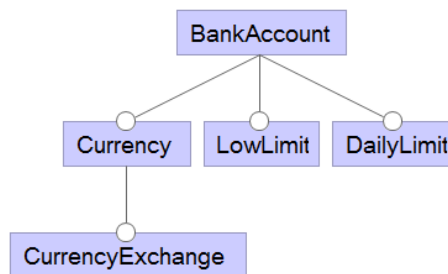

FIGURE 1 – Ligne de produits des comptes bancaires

la dernière l'implantation des fonctions ainsi que les preuves des propriétés énoncées dans les autres espèces. La séparation de la spécification en deux espèces facilitera la définition des autres features.

```
species BA_spec0 =
signature update : Self -> int -> Self;
signature balance : Self -> int;
signature over : int;
property ba_over : all  b: Self,  balance(b) >= over;
end ;;

species BA_spec =
inherit BA_spec0;
property ba_update : all b:Self, all a: int,
  balance(b) + a >= over ->
    balance(update(b, a)) = balance(b) + a;
end ;;

species  BA_impl =
inherit BA_spec;
representation = int;
let balance(b) = b;
let update(b,a) = if b + a >= over then b + a else b;
proof of ba_update = by definition of update, balance ;
proof of ba_over = assumed ;
end ;;
```

Dans BA_spec0 sont introduites les signatures de deux fonctions, balance et update et une constante over. La fonction balance permet de consulter le solde du compte passé en argument, update permet de créditer/débiter le compte passé en argument, le résultat est le nouveau compte. La propriété/spécification ba_over est une propriété invariante, elle indique que tout compte doit avoir un solde supérieur ou égal à over. Cette propriété doit être vérifiée dans tout produit dérivé de cette ligne de produits. La deuxième espèce hérite des signatures et des propriétés de la première espèce et ajoute une nouvelle propriété spécifiant la mise à jour d'un compte, en particulier les conditions de succès de cette opération. Enfin, l'espèce BA_impl réalise la spécification. Elle précise la représentation des comptes bancaires. Ici un compte est représentée par son solde, soit un entier. Et donc l'accès au solde d'un compte (la fonction balance) est l'identité. La preuve de la propriété ba_update énoncée précédemment est faite très simplement par le prouveur automatique Zenon.

Il suffit de lui indiquer d'utiliser les définitions des fonctions `update` et `balance`. La preuve de la propriété `ba_over` est quant à elle admise. En fait, elle ne peut pas être faite directement mais elle doit être remplacée par la preuve que cette propriété est préservée par chacune des fonctions (dont le résultat a le type `Self`) de l'espèce [9].

La caractéristique DL permet de définir les fonctionnalités des comptes bancaires avec une limite de retrait. Ainsi on ajoute une nouvelle constante `limit_withdraw`. Cette extension demande une redéfinition de la fonction `update`, ce que permet le langage FoCaLiZe. En cas de retrait (argument négatif), la fonction va vérifier que l'on est bien en deçà de la limite de retrait. Cependant la propriété `ba_update` n'est plus vraie dans le contexte de DL. Nous avons ici une propriété plus restreinte. Nous définissons deux propriétés par renforcement des pré-conditions de `ba_update`, ce qui revient à spécifier séparément les actions créditer et débiter. Nous appelons ce schéma, un raffinement de propriété : il se limite à la redéfinition d'une propriété en ajoutant une prémisse. Du fait de cette modification de comportement, on peut définir la spécification de `DL` en héritant de `BA_spec0` mais on ne peut hériter de `BA_spec` car la propriété `ba_update` doit être adaptée. C'est la raison pour laquelle nous avions scindée en deux espèces la spécification de `BA`.

L'idée est de réutiliser le code existant et les preuves existantes. Le `super` tel qu'on le trouve en Java (ou `original` de la programmation par aspects) n'existe pas en FoCaLiZe, il va donc falloir le simuler en utilisant héritage et paramétrisation.

La caractéristique DL est donc définie elle aussi en trois espèces : la première introduit la limite de retrait et hérite des spécifications réutilisables de BA, i.e. l'espèce `BA_spec0`, la seconde définit les propriétés obtenues par raffinement et enfin la dernière contient les implémentations des fonctions et les preuves des propriétés.

```
species DL_spec0 =
inherit BA_spec;
signature limit_withdraw : nat;
end ;;

species DL_spec =
inherit DL_spec0;
property ba_update_ref1 : all b:Self, all a: int,
  a >= 0 -> balance(b) + a >= over ->
    balance(update(b, a)) = balance(b) + a;

property ba_update_ref2 : all b:Self, all a: int,
  a < 0 -> (0 - a) <= limit_withdraw ->
    balance(b) + a >= over ->
      balance(update(b, a)) = balance(b) + a;
end ;;

species DL_impl (M is BA_impl) =
inherit DL_spec;
representation = M;
let balance(b) = M!balance (b) ;
let over = M!over ;

proof of ba_over =
  by definition of balance, over property M!ba_over ;
let update(b,a) =
  if a < 0 then
```

```
    if (0 - a) <= limit_withdraw then M!update (b, a)  else b
  else M!update(b, a);

proof of ba_update_ref1 =
  by definition of update, balance, over
     property M!ba_update int_ge_le_eq  ;

proof of ba_update_ref2 =
  by definition of update, balance, over
     property M!ba_update ;
end ;;
```

L'espèce `DL_impl` a été écrite de manière à réutiliser autant que possible le code et les preuves déjà faites dans le cadre de la caractéristique BA.

4 De GFML à FoCaLiZe

Dans l'approche que nous proposons, l'idée est de décrire en GFML les modifications à réaliser, c'est à dire les seules informations qui différencient DL de BA. Le module GFML, décrit ci-dessous, permet de générer automatiquement les espèces de DL sont générées. Ce module GFML est compilé à l'aide du traducteur que nous développons actuellement et qui produit automatiquement les espèces `DL_spec0`, `DL_spec` et `DL_impl`. On remarquera que les preuves sont faites dans le format accepté par FoCaLiZe (cette partie est en fait recopiée textuellement par le compilateur GFML vers FoCaLiZe). On peut également noter que la syntaxe de GFML est largement inspirée de celle de FoCaLiZe.

```
fmodule DL from BA

signature limit_withdraw : nat;

property ba_update_ref1 refines BA!ba_update
extends premise (a >= 0);
property ba_update_ref2 refines BA!ba_update
extends a < 0 ∧ -a <= limit_withdraw;

let update(b,a) =
if a < 0
  then if -a <= limit_withdraw then BA!update (b, a) else b
  else BA!update(b, a);

proof of ba_update_ref1 =
{* focalizeproof =  by definition of update, balance, over
     property M!ba_update, int_ge_le_eq  ; *}
proof of ba_update_ref2 =
{* focalizeproof =  by definition of update, balance, over
     property M!ba_update ; *}
end ;;
```

La caractéristique racine BA est également décrite à l'aide d'un module GFML, détaillé ci-dessous, qui rassemble les signatures, définitions et preuves. La compilation de ce module en FoCaLiZe produit exactement les trois espèces `BA_spec0`, `BA_spec` et `BA_s`.

```
fmodule BA

signature update : Self -> int -> Self;
signature balance : Self -> int;
signature over : int;
invariant property ba_over : all  b: Self,  balance(b) >= over;
property ba_update : all b:Self, all a: int,
  balance(b) + a >= over ->
    balance(update(b, a)) = balance(b) + a;

representation = int;

let balance(b) = b;
let update(b,a) = if b + a >= over then b + a else b;

proof of ba_update =
  {* focalizeproof = by definition of update, balance ;*}
proof of ba_over = assumed ;
end ;;
```

5 Conclusion

Dans cet article, nous avons présenté une approche de la construction non incrémentale de logiciels à l'aide d'un langage formel, GFML, proche de la variabilité mise en œuvre dans les lignes de produits et permettant de spécifier, implanter et prouver les étapes de développement par ajout ou modification. Ce langage est compilé vers un formalisme existant, ici FoCaLiZe. Nous avons illustré notre approche par l'exemple en ne présentant qu'une des constructions offertes par GFML et avons explicité sa traduction en FoCaLiZe.

GFML est un langage formel dont la sémantique est donnée par sa traduction en FoCaLiZe. Le compilateur de GFML vers FoCaLiZe est en cours de réalisation, en même temps que la définition de différents opérateurs de développement liés à l'approche lignes de produits. En plus de l'exemple donné d'ajout d'une pré-condition, citons l'ajout ou la modification d'un paramètre d'une fonction qui va avoir un impact partout où est appelée cette fonction. Les modifications calculées ont un impact sur les preuves déjà réalisées. L'opérateur décrit le travail à réaliser pour obtenir une nouvelle version prouvée du logiciel.

Références

[1] J. Abrial. On constructing large software systems. In J. van Leeuwen, editor, *Algorithms, Software, Architecture - Information Processing '92, Volume 1, Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain*, volume A-12 of *IFIP Transactions*, pages 103–112. North-Holland, 1992.

[2] S. Boulmé. *Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel*. Thèse de doctorat, Université Paris 6, 2000.

[3] P. Clements and L. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001.

[4] Dedukti. http://dedukti.gforge.inria.fr/.

[5] C. Dubois, T. Hardin, and V. Donzeau-Gouge. Building certified components within FOCAL. In H. Loidl, editor, *Revised Selected Papers from the Fifth Symposium on Trends in Functional Programming, TFP 2004*, volume 5 of *Trends in Functional Programming*, pages 33–48, 2006.

[6] FoCaLiZe. http://focalize.inria.fr/.

[7] T.-K.-Z. Pham, C. Dubois, and N. Levy. Towards correct-by-construction product variants of a software product line : Gfml, a formal language for feature modules. In Proceedings 6th Workshop on *Formal Methods and Analysis in SPL Engineering,* London, UK, 11 April 2015, volume 182 of *EPTCS*, pages 44–55, 2015.

[8] V. Prevosto and D. Doligez. Algorithms and proofs inheritance in the FOC language. *J. Autom. Reasoning*, 29(3-4) :337–363, 2002.

[9] R. Rioboo. Invariants for the focal language. *Ann. Math. Artif. Intell.*, 56(3-4) :273–296, 2009.

[10] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, and G. Saake. Applying design by contract to feature-oriented programming. In *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012*, volume 7212 of *Lecture Notes in Computer Science*, pages 255–269. Springer, 2012.

[11] T. Thüm. Product-line verification with feature-oriented contracts. In M. Pezzè and M. Harman, editors, *Int'l Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 374–377. ACM, 2013.

[12] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof composition for deductive verification of software product lines. In *Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST'11)*, pages 270–277. IEEE Computer Society, 2011.