
Model-checking sur des théories avec liaison de variables

David Baelde¹

Dale Miller¹ Andrew Gacek² Gopalan Nadathur² Alwen Tiu³

¹ INRIA & LIX, École Polytechnique

² Digital Technology Center and Dept of CS, University of Minnesota

³ Australian National University and NICTA

Travaux financés par les “Équipes Associées” Slimmer et par la NSF.

On s'intéresse au raisonnement sur des spécifications formelles de systèmes avec liaison de variable dans le style HOAS. Bedwyr se situe à mi-chemin dans ce programme :

Calcul : λ Prolog

→ Model-checking : Bedwyr

Raisonnement : ...

Le développement de la logique LINC fournit déjà un support théorique pour la dernière étape du programme. Bedwyr est le premier outil qui l'utilise. Il offre un gain d'expressivité intéressant par rapport à (λ) Prolog, et reste totalement automatique.

Plan :

- LINC et la méthodologie HOAS
- Bedwyr
- Quelques points d'implémentation

LINC et la méthodologie HOAS

Quelques idées directrices, plus faciles à accepter qu’à respecter :

- L’ α -équivalence doit être prise en compte dès le départ : les noms ne sont qu’un artefact de la syntaxe concrète, comme les caractères d’espace.
- Comme on est passé des *chaînes de caractères* de Church ou Gödel aux *arbres* de syntaxe abstraite, on passe maintenant aux *λ -termes* de syntaxe abstraite.
- “There is no such thing as a free variable” : tout est lié quelquepart. Quand on calcule/raisonne, on déplace les liaisons de variables, des termes (objets) aux formules, puis aux séquents.

On encode par des variables liées :

- les variables du λ -calcul ;
- les noms du π -calcul ;
- les metavariables des patterns de ML ;
- les metavariables de l’unification dans l’algorithme W...

Idée : Utiliser les lieux et le contexte d'une logique pour y représenter ceux de systèmes objets.

term (app M N) :- term M, term N.

term (abs M) :- pi x \ term x => term (M x).

C'est bien pour *calculer* mais c'est trop puissant pour *raisonner* ! Aucun espoir par exemple de spécifier correctement la bisimulation, qui comporte déjà un peu de raisonnement...

```
term x => sigma m\ sigma n\ x = app m n ; x = abs m ?
```

Il faut pouvoir inverser les clauses du programme, procéder à une étude de cas. Logiquement, on passe à la notion de définition. En pratique, l'implication :- se transforme en :=. Mais cela implique aussi d'éviter les constructions non monotones.

On peut réécrire term en explicitant le contexte :

```
term Gamma X          := mem X Gamma.
term Gamma (app M N)  := term Gamma M, term Gamma N.
term Gamma (abs M)    := pi x\ term (cons x Gamma) (M x).

% term l x =>
%   (mem x l ; sigma m\ sigma n\ x = app m n ; x = abs m).
```

Voyons ce qui ne va pas avec l'utilisation de la quantification universelle.

$\text{seq } \Gamma \text{ (and } A \ B) := \text{seq } \Gamma \ A, \text{ seq } \Gamma \ B.$

$\text{seq } \Gamma \text{ (or } A \ B) := \text{seq } \Gamma \ A; \text{ seq } \Gamma \ B.$

$\text{seq } \Gamma \text{ (imp } A \ B) := \text{seq (cons } A \ \Gamma) \ B.$

$\text{seq } \Gamma \text{ (forall } F) := \text{pi } x \backslash \text{seq } \Gamma \ (F \ x).$

$\text{seq } \Gamma \text{ (exists } F) := \text{sigma } x \backslash \text{seq } \Gamma \ (F \ x).$

[...]

On ne s'en sort pas si mal puisqu'on a :

$$\vdash \text{seq } [\Gamma] [A] \quad \text{si et seulement si} \quad \Gamma \vdash A$$

La spécification se comporte bien, mais on ne peut toujours pas raisonner dessus sans passer au niveau meta (comme Twelf avec son *meta-theorem-prover*).

Voyons cela en détail. Supposons qu'on ait la prouvabilité de :

$$x, y; p\ x\ T, p\ y\ U \vdash p\ x\ W$$

Que peut-on en déduire ? Nécessairement, $T = W$.

Pourtant on n'a pas :

$\pi\ t \setminus \pi\ u \setminus \pi\ w \setminus$

(seq nil

(forall $x \setminus$ forall $y \setminus$ ($p\ x\ t \rightarrow p\ y\ u \rightarrow p\ x\ w$)))

$\Rightarrow w = t$

En fait, on ne voulait pas dire \forall , on voulait dire... ∇ . Sans rentrer dans les détails de la logique, $(\nabla x \dots)$ se lit intuitivement

“Supposons que j'aie une variable *fraîche* x , alors...”

On peut maintenant reformuler dans LINC la sémantique one-step du π -calcul puis la (bi)simulation. Toutes les conditions auxiliaires ayant trait aux noms sont traduites dans l'agencement des trois quantificateurs de LINC.

```
% kind  n    type.    % Names
% type  a    type.    % Actions
% kind  p    type.    % Processes

% type  dn, up      n -> n -> a.

% type  z          p.
% type  in         n -> (n -> p) -> p.
% type  out        n -> n -> p -> p.
% type  plus, par  p -> p -> p.
% type  nu         (n -> p) -> p.
```

Extraits de la définition des deux jugements pour les transitions :

- $\text{one} :: p \rightarrow a \rightarrow p \rightarrow o$
pour l'input lié et l'output libre ;
- $\text{onep} :: p \rightarrow (n \rightarrow a) \rightarrow (n \rightarrow p) \rightarrow o$
pour l'output lié et l'input, où nom lié est...lié.

% bound input

$\text{onep} (\text{in } X \ v \setminus M \ v) (v \setminus \text{dn } X \ v) (v \setminus M \ v).$

% free output

$\text{one} (\text{out } X \ Y \ P) (\text{up } X \ Y) P.$

% comm

$\text{one} (\text{par } P \ Q) \ \text{tau} \ (\text{par } (M \ Y) \ T) :=$
 $\text{onep } P \ (\text{dn } X) \ (v \setminus M \ v) \ \& \ \text{one } Q \ (\text{up } X \ Y) \ T.$

% restriction

one (nu x\ P x) A (nu x\ Q x) :=

nabla x\ one (P x) A (Q x).

onep (nu x\ P x) A (y\ nu x\ Q x y) :=

nabla x\ onep (P x) A (y\ Q x y).

% open

onep (nu x\ M x) (up X) N :=

nabla y\ one (M y) (up X y) (N y).

% close

one (par P Q) tau (nu y\ par (M y) (N y)) :=

sigma X\ onep P (dn X) (y\ M y) & onep Q (up X) (y\ N y).

On peut enfin écrire la simulation :

$$\begin{aligned} \text{coinductive sim } P \ Q := & \ \text{pi } A \ \text{pi } P1 \ \text{pi } M \ \backslash \\ & \ (\text{one } P \ A \ P1 \Rightarrow \text{one } Q \ A \ Q1 \ \& \ \text{sim } P1 \ Q1), \\ & \ (\text{onep } P \ (\text{dn } X) \ M \Rightarrow \text{onep } Q \ (\text{dn } X) \ N, \\ & \ \text{pi } w \ \backslash \ \text{sim } (M \ w) \ (N \ w)), \\ & \ (\text{onep } P \ (\text{up } X) \ M \Rightarrow \text{onep } Q \ (\text{up } X) \ N, \\ & \ \text{nabla } w \ \backslash \ \text{sim } (M \ w) \ (N \ w)). \end{aligned}$$

Les problèmes de liaison et de fraîcheur sont complètement exprimés grâce aux trois quantificateurs de LINC.

$$\begin{aligned} a(x).a(y).0 & \sim a(x).vz.a(y).0 \\ a(x).vy.[x = y].P & \sim a(x).0 \quad (\vdash \forall x.\nabla y.x \neq y) \\ vx.\bar{a}(x).c(y).[x = y].P & \not\sim vx.\bar{a}(x).c(y).0 \quad (\not\vdash \nabla x.\forall y.x \neq y) \end{aligned}$$

Et maintenant, voyons comment Bedwyr peut exécuter cette spécification.

Bedwyr

Bedwyr construit des preuves dans un fragment de LINC. Vu son expressivité, on peut encore parler d'un langage de programmation (purement) logique.

$$\begin{aligned} L0 & ::= L0 \wedge L0 \quad | \quad L0 \vee L0 \quad | \quad s = t \quad | \quad p\vec{t} \\ & \quad | \quad \nabla x.L0x \quad | \quad \exists x.L0x \\ L1 & ::= L1 \wedge L1 \quad | \quad L1 \vee L1 \quad | \quad s = t \quad | \quad p\vec{t} \\ & \quad | \quad \nabla x.L1x \quad | \quad \exists x.L1x \\ & \quad | \quad \forall x.L1x \quad | \quad L0 \supset L1 \end{aligned}$$

Implicitement : conditions syntaxiques sur les définitions des atomes p .

La clé : à gauche des implications il n'y a que des connecteur *inversibles*.

La stratégie sera donc de les introduire de façon gourmande.

On cherche une substitution θ (sur les variables existentielles) telle que $\vdash (A \supset B)\theta$.

1. Calculer toutes les σ_i (sur les variables *universelles*) telles que $\vdash A\sigma_i$.
2. Trouver θ tel que pour tout i , $\vdash B\sigma_i\theta$.

Cas particulier : l'échec fini sur la formule A de niveau 0 entraîne un succès sur $A \supset \perp$.

Le moteur de Bedwyr est en fait une procédure standard de recherche de preuve, à quelques détails près :

- \forall et \supset ne sont acceptés qu'à droite ;
- on unifie des variables existentielles à droite, mais des universelles à gauche ;
- les variables existentielles sont interdites à gauche.

λ Prolog ne fait aucune étude de cas :

```
p (f a) :- true.
```

```
p (f b) :- true.
```

```
?- pi x\ p x => sigma y\ x = f y.
```

► Échoue avec λ -Prolog, passe avec Bedwyr.

D'un autre côté, Bedwyr fait *toujours* une étude de cas exhaustive :

```
nat z      := true.
```

```
nat (s X) := nat X.
```

```
?= pi x\ nat x => nat x.
```

► Succès avec λ -Prolog mais boucle infinie avec Bedwyr.

Bedwyr dispose aussi d'un dispositif expérimental de *tabling*, afin d'éviter les calculs *redondants* ou *cycliques*.

Si l'on déclare explicitement une définition comme étant **inductive** ou **coinductive**, Bedwyr mettra en mémoire les instances prouvées/réfutées de la définition, ainsi que celles que la recherche en cours a reconstruit (pour détecter les cycles).

$$\frac{\frac{?}{\vdash d\bar{x}}}{\vdots \quad \vdots \quad \vdots} \vdash d\bar{x}$$

Un cycle sur une définition inductive est un échec irrémédiable, mais c'est un succès pour une définition coinductive.

Ceci se justifie logiquement à l'aide des règles de (co)induction de LINC.

Bedwyr : quelques points d'implémentation

Les grands traits de l'implémentation, en OCaml :

- Boucle de recherche de preuve par continuation, unification destructive, annulée efficacement (par *trailing*) lors du *backtracking*...
- Comme dans de nombreux outils, l'unification est restreinte aux *higher-order patterns*. On a ainsi l'existence d'un unificateur le plus général. On utilise une implémentation SML de Linnell & Nadathur, traduite et adaptée.
- Bedwyr ne dispose pas (vraiment) d'un *type-checker*. Cela serait bien pratique pour l'utilisateur, mais ne pose pas de problème pour l'implémentation.

Dès lors qu'on a des termes d'ordre supérieur, une implémentation naïve de ∇ ne pose pas de problème. On peut le voir comme une simple λ -abstraction sur les formules, sous laquelle on pourrait raisonner. On le voit dans les équivalences suivantes, ce “connecteur” a un petit rôle :

$$\nabla a.(P a \wedge Q a) \equiv (\nabla a.P a) \wedge (\nabla a.Q a)$$

$$\nabla a.\forall x.F a x \equiv \forall h.\nabla a.F a (h a)$$

$$\nabla a.(u a = v a) \equiv (\lambda a.u a) = (\lambda a.v a)$$

Il suffit de garder trace des variables génériques introduites, pour les appliquer aux variables existentielles et universelles, et abstraire les égalités.

On retrouve nombre de techniques classiques dans nos modules de manipulation des (λ -)termes :

- Indices de de Bruijn pour les variables liées ;
- Normalisation paresseuse par substitutions explicites (*suspension calculus* de Nadathur) ;
- Les variables libres sont annotées par un *niveau* qui indique la position de leur introduction dans le préfixe de quantifications \forall/\exists . C'est plus compact et aisé à manipuler dans l'algorithme d'unification.

Ce qu'on a ajouté/adapté :

- L'algorithme d'unification est paramétré par la donnée des types de variables *instanciables* et *constantes*. Selon qu'on travaille à gauche ou à droite, on change ainsi le comportement de l'algorithme plutôt que les termes.
- On ajoute une annotation de *niveau local* donnant la position de l'introduction d'une variable dans les quantifications ∇ . C'est plus compact, et permet d'étendre naturellement l'algorithme d'unification, pour supporter quelques problèmes supplémentaires. Par exemple :

$$\nabla y. \exists a. \forall x. a x = a x$$

qui deviendrait

$$\exists a. \forall x. (\lambda y. (A y) (x y)) = (\lambda y. (A y) (x y))$$

L'unification à gauche pose problème (ceci est indépendant de ∇).

- L'unification à gauche peut faire sortir des *higher-order patterns*.
- En plus, notre représentation peut le cacher !

% La représentation par niveaux cache un motif illégal.

$\pi x \setminus \pi y \setminus \sigma Z \setminus x = y \Rightarrow Z = x$

implémentation

version skolemisée

$$x^0 = y^1 \supset Z^1 = x^0 \quad \sim \quad x = y \supset Zxy = x$$

$$Z^1 = x^0 \quad \not\sim \quad Zxx = x$$

% Encore pire, les niveaux sont faussés.

$\pi x \setminus \sigma Y \setminus \pi z \setminus x = f z \Rightarrow Y = f z$

implémentation

version skolemisée

$$x^0 = fz^1 \supset Y^0 = fz^1 \quad \sim \quad x = fz \supset Yx = fz$$

$$Y^0 = fz^1 \quad \not\sim \quad Y(fz) = fz$$

Le code source de Bedwyr ainsi que de nombreux exemples sont disponibles. La librairie de représentation et d'unification des termes est assez re-utilisable, utilisée actuellement dans plusieurs projets.

`http://slimmer.gforge.inria.fr/bedwyr`

Pour la suite :

- Dans Bedwyr : tabling, optimisations, ...
- Et surtout, au delà de Bedwyr :
 - On a déjà des prototypes interactifs implémentant (des variantes de) LINC, mais il reste aussi de nombreuses questions...
 - Étude des limites de LINC, différents formalismes autour de ∇ . Je cherche à combler certaines faiblesses de LINC, tout en préservant sa simplicité, et l'orthogonalité entre ∇ et le reste de la logique.
 - Étude de la structure des preuves par (co)induction. Notamment : extension de la focalisation au points fixes en logique linéaire ; extension du format autorisé pour les points fixes ; résultats d'expressivité.

- *The Bedwyr system for model checking over syntactic expressions.*
Baelde, Gacek, Miller, Nadathur & Tiu. CADE 2007.
- *A Proof Theory for Generic Judgements.*
Dale Miller & Alwen Tiu. TOCL, 2005.
- *A Logical Framework for Reasoning about Logical Specifications.*
Alwen Tiu. PhD, 2004.
- *Model checking for pi-calculus using proof search.*
Alwen Tiu. CONCUR 2005.
- *A Semi-Functional Implementation of a Higher-Order Logic Programming Language.* Elliot & Pfenning, 1990.
- *Least and greatest fixed points in linear logic.*
David Baelde & Dale Miller. LPAR 2007.