# Lustre Model Checker

Diane Gallois-Wong, Raphaël Rieu-Helft

17 décembre 2015

# Exemple de l'énoncé

```
node incr (tic: bool) returns (ok: bool);
var cpt : int;
let
cpt = (0 -> pre cpt) + if tic then 1 else 0;
ok = true -> (pre cpt) <= cpt;
tel
```

$$\Delta(n) = \begin{cases} cpt(n) = ite(n = 0, 0, cpt(n-1)) + ite(tic(n), 1, 0) \\ ok(n) = ite(n = 0, true, cpt(n-1) \leq cpt(n)) \end{cases}$$

$$P(n) = ok(n)$$

$k$-induction :

$$\Delta(0), \Delta(1), \ldots, \Delta(k) \models P(0), P(1), \ldots, P(k)$$

$$\forall n, \quad \Delta(n), \ldots, \Delta(n+k), P(n), \ldots, P(n+k-1) \models P(n+k)$$

1. De l'ast typée fournie à l'ast d'AEZ
   - Étape 1 : traduction, gestion de $\rightarrow$ et *pre*
   - Étape 2 : élimination des termes réduits à une formule
   - Étape 3 : élimination des tuples et appels de noeuds
2. Implémentation du solveur
3. Une optimisation : élimination de cas particuliers

# De l'ast typée fournie à l'ast d'AEZ

$e$ ::=

| | |
|---|---|
| $\mid c$ | constante |
| $\mid x$ | variable |
| $\mid op(e, ..., e)$ | opération |
| $\mid nd(e, ..., e)$ | appel de noeud |
| $\mid prim(e, ..., e)$ | primitive |
| $\mid e \rightarrow e$ | |
| $\mid pre(e)$ | |
| $\mid (e, ..., e)$ | tuple |

$t$ ::=

| | |
|---|---|
| $\mid c$ | constante |
| $\mid t\ op\ t$ | opération arithmétique |
| $\mid ite(f, t, t)$ | opération logique |
| $\mid \varphi(t, ..., t)$ | appel de fonction |

$f$ ::=

| | |
|---|---|
| $\mid t$ | terme |
| $\mid t\ cmp\ t$ | comparaison |
| $\mid f\ op\ f$ | opération logique |

# De l'ast typée fournie à l'ast d'AEZ

$e$ ::=

| | |
|---|---|
| $\mid c$ | constante |
| $\mid x$ | variable |
| $\mid op(e, \ldots, e)$ | opération |
| $\mid nd(e, \ldots, e)$ | appel de noeud |
| $\mid prim(e, \ldots, e)$ | primitive |
| $\mid e \rightarrow e$ | |
| $\mid pre(e)$ | |
| $\mid (e, \ldots, e)$ | tuple |

$t$ ::=

| | |
|---|---|
| $\mid c$ | constante |
| $\mid t\ op\ t$ | opération arithmétique |
| $\mid ite(f, t, t)$ | opération logique |
| $\mid \varphi(t, \ldots, t)$ | appel de fonction |
| $\mid x(n - k)$ | variable à un instant |
| $\mid f$ | formule |
| $\mid (t, \ldots, t)$ | tuple |
| $\mid nd(t, \ldots, t)$ | appel de noeud |

$f$ ::=

| | |
|---|---|
| $\mid t$ | terme |
| $\mid t\ cmp\ t$ | comparaison |
| $\mid f\ op\ f$ | opération logique |
| $\mid n = k$ | temps = constante |

# De l’ast typée fournie à l’ast d’AEZ

- Étape 1 : changement d’ast, gestion de $\rightarrow$ et *pre*.
- Étape 2 : élimination des termes réduits à une formule.
- Étape 3 : élimination des tuples et appels de noeuds.

# Étape 1 : traduction, gestion de $\rightarrow$ et *pre*

$n$ : terme global représentant le temps

Objectif : $\quad x \rightarrow \text{pre } y \quad \longrightarrow \quad \text{if } n = 0 \text{ then } x(n) \text{ else } y(n-1)$

On propage $k$, décalage par rapport à $n$ :

$$\Phi(\text{pre } e,\ k) \quad = \quad \Phi(e,\ k+1)$$

Variables :

$$\Phi(x,\ k) \quad = \quad x(n-k)$$

$\rightarrow$ :

$$\Phi(e1 \rightarrow e2,\ k) \quad = \quad \text{if } n = k \text{ then } \Phi(e1,\ k) \text{ else } \Phi(e2,\ k)$$

# Étape 2 : élimination des termes réduits à une formule

$$f \qquad \longrightarrow \qquad aux$$

où *aux* est une variable fraîche, en ajoutant la formule

$$(aux \Rightarrow f) \quad \&\& \quad (f \Rightarrow aux)$$

# Étape 3 : élimination des tuples et appels de noeuds

$$(x1, x2) = (t1, t2) \quad \longrightarrow \quad \begin{cases} x1 = t1 \\ x2 = t2 \end{cases}$$

if $b$ then $(u1, u2)$ else $(v1, v2)$ $\longrightarrow$ (if $b$ then $u1$ else $v1$, ...)

$$nd(t1, t2) \quad \longrightarrow \quad (out1, out2, out3)$$

où $out1$, $out2$, $out3$ sont des variables fraîches.
On ajoute toutes les formules compilées de $nd$ en renommant toutes les variables en variables fraîches, en particulier les sorties en $out1$, $out2$, $out3$.
Soit $in1$, $in2$ les nouveaux noms des entrées, on ajoute aussi les formules $in1 = t1$, $in2 = t2$.

# Le procédé de $k$-induction

Cas de base :

$$\Delta(0), \Delta(1), \ldots, \Delta(k) \models P(0), P(1), \ldots, P(k)$$

Cas inductif :

$$\forall n, \Delta(n), \ldots, \Delta(n+k), P(n), \ldots, P(n+k-1) \models P(n+k)$$

## Génération de code

$$\Delta(0), \Delta(1), \ldots, \Delta(k) \models P(0), P(1), \ldots, P(k)$$

$$\forall n > 0, \Delta(n), \ldots, \Delta(n+k), P(n), \ldots, P(n+k-1) \models P(n+k)$$

- On augmente k jusqu'à échouer le cas de base ou réussir l'induction
- Code généré à partir de la syntaxe abstraite AEZ compilée

# Une optimisation : élimination de cas particuliers

```
a -> pre (b -> pre (c -> pre d));
```

est compilé en la formule :

```
if n=0 then a(n) else if n=1 then b(n-1) else if n=2 then
c(n-2) else d(n-3)
```

Beaucoup de if... On voudrait pouvoir supposer $n > 2$ pour l'induction

# Une optimisation : élimination de cas particuliers

Il suffit d'ajouter des cas de base :

$$\Delta(0), \Delta(1), \ldots, \Delta(k+p) \models P(0), P(1), \ldots, P(k+p)$$

$$\forall n > p, \Delta(n), \ldots, \Delta(n+k), P(n), \ldots, P(n+k-1) \models P(n+k)$$

On peut beaucoup simplifier les formules dans l'induction, voire terminer avec un $k$ plus petit.

# Un exemple

```
node check () returns (ok: bool);
  var n0, n1:int; b:bool;
  let
    n0 = 0 -> pre n0;
    n1 = 0 -> pre n1;
    b = true -> pre (true -> false);
    ok = if b then n0=n1 else true;
  tel
```

1-induction sans l'optimisation, pas besoin d'induction avec !

# Choix de $p$

- On prend le plus grand $i$ tel que `if(n=i)` apparaît dans le code généré

$\Longrightarrow$ correspond à la plus grande profondeur de flèches dans le code source Lustre

- Pas optimal, mais on ne peut pas toujours déterminer le meilleur $p$ statiquement