HAL archives-ouvertes.fr

Méthodes et outils pour la spécification et la preuve de propriétés difficiles de programmes séquentiels (Documents de soutenance)

Martin Clochard

▶ **To cite this version:**

Martin Clochard. Méthodes et outils pour la spécification et la preuve de propriétés difficiles de programmes séquentiels (Documents de soutenance). 2019. hal-02047458

HAL Id: hal-02047458

https://hal.inria.fr/hal-02047458

Submitted on 24 Feb 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Méthodes et outils pour la spécification et la preuve de propriétés difficiles de programmes séquentiels

Martin Clochard

LRI, Université Paris-Sud, CNRS &
Inria, Université Paris-Saclay

30 mars 2018

Vérification de programmes :

= démontrer qu'un programme se comporte comme prévu

- programmes critiques : centrales nucléaires, avions,
  voitures autonomes...
- conséquences d'un bug potentiellement désastreuses

Méthodes de vérification :

- test formel
- interprétation abstraite
- vérification de modèles
- vérification déductive

Vérification déductive de programmes :

- preuve de bon comportement effectuée via un ensemble de règles de déduction
- exemple : logiques de programmes
  - logique de Hoare
  - logique de séparation
- exemple : génération de conditions de vérification
  - calcul de plus faible pré-condition
  - exécution symbolique

La preuve de programme peut demander des efforts conséquents

- micro-noyau seL4 (11 personnes-années)
- compilateur C CompCert (6 personnes-années)

Objectif de cette thèse : réduire l'effort de vérification

- inventer de nouvelles méthodes pour la vérification déductive
- focus : programmes séquentiels, propriétés difficiles

Expériences menées via l'outil de preuve de programme Why3

- outil basé sur le calcul de plus faible pré-condition
- repose sur des démonstrateurs automatiques de théorèmes

Contribution centrale de ma thèse :

- méthode de preuve « par débogage »
- applications : programmes dérécursifiés, compilateurs
- un fondement théorique sur des jeux transfinis

1. Preuve par « débogage »
2. Aller plus loin : technique de preuve d'un compilateur
3. Extension aux comportements infinis
4. Résumé des contributions & perspectives

Fonction 91 de McCarthy [1970]

$$f_{91}(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ f_{91}(f_{91}(n + 11)) & \text{sinon} \end{cases}$$

Fonction 91 de McCarthy [1970]

$$f_{91}(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ f_{91}(f_{91}(n+11)) & \text{sinon} \end{cases}$$

Solution unique :

$$f_{91}(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ 91 & \text{sinon} \end{cases}$$

Preuve : récurrence sur $101 - n$

Fonction 91 de McCarthy [1970]

$$f_{91}(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ f_{91}(f_{91}(n + 11)) & \text{sinon} \end{cases}$$

Solution unique :

$$f_{91}(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ 91 & \text{sinon} \end{cases}$$

Why3 APPROVED

Preuve : récurrence sur $101 - n$

Vérifions cette version itérative (calcule $f_{91}(n) = f_{91}^{(e)}(r)$)

```
e = 1; r = n;
while(1) {
  if(r > 100) {
    r = r - 10;
    e = e - 1;
    if(e == 0) return r;
  } else {
    r = r + 11;
    e = e + 1;
  }
}
```

Vérifions cette version itérative (calcule $f_{91}(n) = f_{91}^{(e)}(r)$)

```
e = 1; r = n;
while(1) {
  if(r > 100) {
    r = r - 10;
    e = e - 1;
    if(e == 0) return r;
  } else {
    r = r + 11;
    e = e + 1;
  }
}
```

Spécification :

- termine
- le résultat est

$$f_{91}(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ 91 & \text{sinon} \end{cases}$$

Comment prouver ce résultat ?

Vérifions cette version itérative (calcule $f_{91}(n) = f_{91}^{(e)}(r)$)

```
e = 1; r = n;
while(1) {
  if(r > 100) {
    r = r - 10;
    e = e - 1;
    if(e == 0) return r;
  } else {
    r = r + 11;
    e = e + 1;
  }
}
```

Spécification :

- termine
- le résultat est

$$f_{91}(n) = \begin{cases} n - 10 & \text{si } n > 100 \\ 91 & \text{sinon} \end{cases}$$

Comment prouver ce résultat ?

Idée : utiliser la structure d'une version récursive

Combinaison de programmes inspirée par le débogage

Le programme récursif contrôle l'exécution du programme itératif

```
e = 1; r = n;
while(1) {
  if(r > 100) {
    r = r - 10;
    e = e - 1;
    if(e == 0) return r;
  } else {
    r = r + 11;
    e = e + 1;
  }
}
```

```
let rec f91 r =
  if r > 100
  then r - 10
  else f91 (f91 (r + 11))
in
f91 n
```

Le programme récursif contrôle l'exécution du programme itératif

L : point d'arrêt, CONT : exécution jusqu'au point d'arrêt

```
e = 1; r = n;
while(1) {
L:if(r > 100) {
    r = r - 10;
    e = e - 1;
    if(e == 0) return r;
  } else {
    r = r + 11;
    e = e + 1;
  }
}
```

```
CONT();
let rec f91 r =
  if r > 100
  then (CONT(); r-10)
  else (CONT(); f91(f91(r+11)))
in
f91 n
```

Le programme récursif contrôle l'exécution du programme itératif

L : point d'arrêt, CONT : exécution jusqu'au point d'arrêt

```
e = 1; r = n;
while(1) {
L:if(r > 100) {
    r = r - 10;
    e = e - 1;
    if(e == 0) return r;
  } else {
    r = r + 11;
    e = e + 1;
  }
}
```

```
CONT();
let rec f91() =
  if r > 100
  then CONT()
  else (CONT(); f91(); f91())
in
f91()
```

Le programme récursif contrôle l'exécution du programme itératif

L : point d'arrêt, CONT : exécution jusqu'au point d'arrêt

```
e = 1; r = n;
while(1) {
L:if(r > 100) {
     r = r - 10;
     e = e - 1;
     if(e == 0) return r;
  } else {
     r = r + 11;
     e = e + 1;
  }
}
```

```
CONT();
let rec f91() =
  requires { e > 0 ∧ pc = L }
  ensures  { r = f91(old r) }
  ensures  { e = old e - 1 }
  ensures  { pc = if e ≠ 0
                 then L else End }
  variant { 101 - r }
  if r > 100 then CONT()
  else (CONT(); f91(); f91())
in
f91()
```

Le programme récursif contrôle l'exécution du programme itératif

L : point d'arrêt, CONT : exécution jusqu'au point d'arrêt

```
e = 1; r = n;
while(1) {
L:if(r > 100) {
      r = r - 10;
      e = e - 1;
      if(e == 0) return r;
   } else {
      r = r + 11;
      e = e + 1;
   }
}
```

```
CONT();
let rec f91() =
  requires { e > 0 ∧ pc = L }
  ensures  { r = f91(old r) }
  ensures  { e = old e - 1 }
  ensures  { pc = if e ≠ 0
                  then L else End }
  variant { 101 - r }
  if r > 100 then CONT()
  else (CONT(); f91(); f91())
in
f91()
```

```
val CONT()
  requires { pc = Begin ∨ pc = L }
  ensures  {
    (old pc = Begin → r = n ∧ e = 1 ∧ pc = L) ∧
    (old pc = L → if old r > 100
      then r = old r - 10 ∧ e = old e - 1
        ∧ (if e = 0 then pc = End else pc = L)
      else r = old r + 11 ∧ e = old e + 1 ∧ pc = L)
  }
```

Modélise l'exécution du programme itératif entre deux points d'arrêt

- peut être mécaniquement générée par exécution symbolique
- nécessite ≥ 1 point d'arrêt par cycle du flot de contrôle

Quelle est la méthode sous-jacente ?

- Langage étudié $\mathcal{L}$
  - cible de la vérification (C, ML, assembleur, etc.)
  - sémantique opérationnelle connue
- Langage de vérification $\mathcal{W}$
  - jusqu'ici, aussi bien Why3 que Dafny, Viper, Boogie, etc.
  - variable globale `now` : état de l'exécution de $\mathcal{L}$ (ici `pc,r,e,n`)
  - fonction `CONT` : avance l'exécution
  - `now` uniquement modifié à travers `CONT`

Quelle est la méthode sous-jacente ?

- Langage étudié $\mathcal{L}$
  - cible de la vérification (C, ML, assembleur, etc.)
  - sémantique opérationnelle connue
- Langage de vérification $\mathcal{W}$
  - jusqu'ici, aussi bien Why3 que Dafny, Viper, Boogie, etc.
  - variable globale `now` : état de l'exécution de $\mathcal{L}$ (ici `pc,r,e,n`)
  - fonction `CONT` : avance l'exécution
  - `now` uniquement modifié à travers `CONT`

Propriété de transfert

Toute propriété au sujet de `now` établie dans $\mathcal{W}$
implique la même propriété pour l'exécution dans $\mathcal{L}$.

Application : programmes dérécursifiés

J'ai vérifié plusieurs exemples en Why3 via cette méthode :

- deux algorithmes de marquage des sommets atteignables :
  - algorithme de Schorr-Waite (graphe mémoire)
  - second problème de la compétition VerifyThis 2016 (arbres binaires avec pointeurs parents)
- la suppression dans un arbre binaire de recherche
  - troisième problème de la compétition VerifyThis 2012

Cela donne des preuves plus simples :

- annotations plus simples (pas de pile explicite)
- plus facile pour les démonstrateurs automatiques

1 Preuve par « débogage »

2 Aller plus loin : technique de preuve d'un compilateur

3 Extension aux comportements infinis

4 Résumé des contributions & perspectives

Des idées proches sont applicables à la preuve d'un compilateur

Cadre :

- langage source $\mathcal{S}$
- langage cible $\mathcal{C}$
- sémantiques à petits pas pour $\mathcal{S}$ et $\mathcal{C}$
- spécification : tout comportement du programme compilé est un comportement du programme source (simulation en arrière)

Hypothèse simplificatrice : comportements finis (pour l'instant)

Simulation en arrière

Énoncé formel : pour n'importe quels

- états initiaux corrélés x_i, y_i pour les langages $\mathcal{S}$, $\mathcal{C}$
- trace d'exécution de $\mathcal{C}$ depuis y_i vers l'état final y_f

$$\begin{tikzcd}
x_i \arrow[d, phantom, "\wr"] & \\
y_i \arrow[r, "\mathcal{C}"'] & y_f
\end{tikzcd}$$

Simulation en arrière

Énoncé formel : pour n'importe quels

- états initiaux corrélés x_i, y_i pour les langages $\mathcal{S}$, $\mathcal{C}$
- trace d'exécution de $\mathcal{C}$ depuis y_i vers l'état final y_f

il existe

- un état final x_f corrélé à y_f
- une trace d'exécution de $\mathcal{S}$ depuis x_i vers x_f

$$\begin{tikzcd}
x_i \arrow[r, dashed, "\mathcal{S}"'] \arrow[d, phantom, "\wr"] & x_f \arrow[d, phantom, "\wr"] \\
y_i \arrow[r, "\mathcal{C}"'] & y_f
\end{tikzcd}$$

Preuve directe : opérateur « et » paresseux

Source : b_1 && b_2

Compilé (pour une machine à pile) :

```
S: c1      ; empile le résultat booléen
   jz E    ; saute à E si 0, laisse sur la pile
   pop     ; dépile
   c2      ; empile le résultat booléen
E: nop
```

Source : b_1 `&&` b_2

Compilé (pour une machine à pile) :

```
S: c1      ; empile le résultat booléen
   jz E    ; saute à E si 0, laisse sur la pile
   pop     ; dépile
   c2      ; empile le résultat booléen
E: nop
```

Cas b_1 vrai :

$$\begin{array}{ccc} x_1 & & \\ \wr & & \\ y_1 & \longrightarrow & y_4 \end{array}$$

Source : b_1 `&&` b_2

Compilé (pour une machine à pile) :

```
S: c1      ; empile le résultat booléen
   jz E    ; saute à E si 0, laisse sur la pile
   pop     ; dépile
   c2      ; empile le résultat booléen
E: nop
```

Cas b_1 vrai :

$$
\begin{tikzcd}
x_1 \arrow[r, dashed, "b_1"] \arrow[d, phantom, "\wr"] & x_2 \arrow[d, phantom, "\wr"] & & \\
y_1 \arrow[r, "c_1"] & y_2 \arrow[rr] & & y_4
\end{tikzcd}
$$

Preuve directe : opérateur « et » paresseux

Source : b_1 `&&` b_2

Compilé (pour une machine à pile) :

`S:` c_1 `; empile le résultat booléen`
`   jz E   ; saute à E si 0, laisse sur la pile`
`   pop    ; dépile`
`   ` c_2 `; empile le résultat booléen`
`E: nop`

Cas b_1 vrai :

$$\begin{tikzcd}
x_1 \arrow[r, dashed, "b_1"] \arrow[d, phantom, "\wr"] & x_2 \arrow[r, dashed, "\texttt{\&\&}"] \arrow[d, phantom, "\wr"] & x_3 \arrow[d, phantom, "\wr"] & \\
y_1 \arrow[r, "c_1"] & y_2 \arrow[r, "\texttt{jz E + pop}"] & y_3 \arrow[r] & y_4
\end{tikzcd}$$

Preuve directe : opérateur « et » paresseux

Source : b_1 `&&` b_2

Compilé (pour une machine à pile) :

S: c_1 `      ; empile le résultat booléen`
`   jz E   ; saute à E si 0, laisse sur la pile`
`   pop    ; dépile`
`   `c_2 `   ; empile le résultat booléen`
`E: nop`

Cas b_1 vrai :

$$\begin{tikzcd}
x_1 \arrow[r, dashed, "b_1"] \arrow[d, phantom, "\wr"] & x_2 \arrow[r, dashed, "\&\&"] \arrow[d, phantom, "\wr"] & x_3 \arrow[rr, dashed, "b_2"] \arrow[d, phantom, "\wr"] & & x_4 \arrow[d, phantom, "\wr"] \\
y_1 \arrow[r, "c_1"] & y_2 \arrow[r, "\texttt{jz E + pop}"] & y_3 \arrow[r, "c_2"] & y_{3.5} \arrow[r] & y_4
\end{tikzcd}$$

Preuve directe : opérateur « et » paresseux

Source : b_1 `&&` b_2

Compilé (pour une machine à pile) :

S: c_1 ; empile le résultat booléen
    `jz E`   ; saute à E si 0, laisse sur la pile
    `pop`   ; dépile
    c_2 ; empile le résultat booléen
E: `nop`

Cas b_1 vrai :

$$\begin{tikzcd}
x_1 \arrow[r, dashed, "b_1"] \arrow[d, phantom, "\wr"] & x_2 \arrow[r, dashed, "\texttt{\&\&}"] \arrow[d, phantom, "\wr"] & x_3 \arrow[rr, dashed, "b_2"] \arrow[d, phantom, "\wr"] & & x_4 \arrow[dl, phantom, "\wr"] \arrow[d, phantom, "\wr"] \\
y_1 \arrow[r, "c_1"] & y_2 \arrow[r, "\texttt{jz E + pop}"] & y_3 \arrow[r, "c_2"] & y_{3.5} \arrow[r, "\texttt{nop}"] & y_4
\end{tikzcd}$$

Une preuve directe demande des efforts conséquents

- hors de portée des démonstrateurs automatiques actuels
- il faut expliciter les états intermédiaires manuellement
- en particulier, il faut traiter les différents cas manuellement

Problèmes similaires à ceux de la logique de Hoare brute

Idée 1 : construire un témoin de simulation

Idée 2 : utiliser un programme de $\mathcal{W}$ comme témoin

- deux variables globales : $\texttt{now}_{\mathcal{S}}$, $\texttt{now}_{\mathcal{C}}$
- deux procédures d'exécution : $\texttt{STEP}_{\mathcal{S}}$, $\texttt{STEP}_{\mathcal{C}}$
- $\texttt{now}_{\mathcal{S}}$, $\texttt{now}_{\mathcal{C}}$ seulement modifiées par $\texttt{STEP}_{\mathcal{S}}$, $\texttt{STEP}_{\mathcal{C}}$

Propriété de transfert

Toute propriété au sujet de $\texttt{now}_{\mathcal{S}}$ et $\texttt{now}_{\mathcal{C}}$ prouvée dans $\mathcal{W}$ implique la même propriété pour l'exécution de $\mathcal{S}$ et $\mathcal{C}$.

Le compilateur produit le témoin

Spécification du compilateur : contrat pour le témoin

```
let rec compile_expr (e:expr) : (c:asm, ghost π:𝒲)
  requires { well_formed e }
  ensures  { {related_pre e c}π{related_post e c} }
= ...
```

`related_pre e c` : états initiaux corrélés pour *e* and *c*

`related_post e c` : états finaux corrélés pour *e* and *c*

Remarquons que

- les programmes de $\mathcal{W}$ deviennent des objets de première classe
- il faut garantir la propriété de transfert de $\texttt{now}_{\mathcal{S}/\mathcal{C}}$ vers $\mathcal{S}/\mathcal{C}$

$\mathcal{W}$ doit être un langage dédié interne

- défini dans l'outil de vérification du compilateur
- sémantique de $\mathcal{W}$: plus faibles pré-conditions $WP(\pi, Q)$
- propriété de transfert pour $\mathcal{W}$: à démontrer

Pour n'importe quels

- programme $\pi \in \mathcal{W}$
- ensemble de paires d'états $Q \subseteq State_{\mathcal{S}} \times State_{\mathcal{C}}$
- $(x_1, y_1) \in WP(\pi, Q)$
- trace d'exécution finie maximale T de $\mathcal{C}$ depuis y_1 vers y_f

$$x_1$$

$$(x_1, y_1) \in WP(\pi, Q)$$

$$y_1 \longrightarrow y_f \not\rightarrow$$

Propriété de transfert (simulation arrière)

Pour n'importe quels

- programme $\pi \in \mathcal{W}$
- ensemble de paires d'états $Q \subseteq State_{\mathcal{S}} \times State_{\mathcal{C}}$
- $(x_1, y_1) \in WP(\pi, Q)$
- trace d'exécution finie maximale T de $\mathcal{C}$ depuis y_1 vers y_f

il existe

- $(x_2, y_2) \in Q$ où y_2 apparaît dans T
- une trace d'exécution de $\mathcal{S}$ depuis x_1 vers x_2

$$x_1 \dashrightarrow {\color{red} x_2}$$

$$(x_1, y_1) \in WP(\pi, Q) \qquad {\color{red} (x_2, y_2) \in Q}$$

$$y_1 \longrightarrow {\color{red} y_2} \rightarrow y_f \not\rightarrow$$

$b \in \mathcal{S}$:	$c \in \mathcal{C}$:	
		{ related_pre b c }
b_1 && b_2	S : c_1 jz E pop c_2 E : nop	
		{ related_post b c }

$b \in \mathcal{S}$:	$c \in \mathcal{C}$:	
		`{ now`$_\mathcal{S}$ $\sim$ `now`$_\mathcal{C}$ $\wedge$ `now`$_\mathcal{C}$`.pc = S` $\wedge$ `now`$_\mathcal{S}$`.evalFocus = "`b_1 `&&` b_2`" }`

b_1 && b_2	`S:` c_1
	`jz E`
	`pop`
	c_2
	`E: nop`	

		`{ now`$_\mathcal{S}$ $\sim$ `now`$_\mathcal{C}$ $\wedge$ `now`$_\mathcal{C}$`.pc = E+1`
		$\wedge$ `fullyEvaluated(now`$_\mathcal{S}$`.evalFocus)`
		$\wedge$ `(old now`$_\mathcal{S}$`).context = now`$_\mathcal{S}$`.context`
		$\wedge$ `let v = valueOf(now`$_\mathcal{S}$`.evalFocus) in`
		`now`$_\mathcal{C}$`.stack = [v] + old now`$_\mathcal{C}$`.stack }`

$b \in \mathcal{S}$:	$c \in \mathcal{C}$:	
		{ $\mathtt{now}_{\mathcal{S}} \sim \mathtt{now}_{\mathcal{C}} \wedge \mathtt{now}_{\mathcal{C}}\mathtt{.pc = S}$ $\wedge\ \mathtt{now}_{\mathcal{S}}\mathtt{.evalFocus = "}b_1 \mathbin{\&\&} b_2\mathtt{"}$ }
		$\mathtt{STEP}_{\mathcal{S}}\mathtt{();}$ (* $\mathtt{now}_{\mathcal{S}}\mathtt{.evalFocus = "}b_1\mathtt{"}$ *)

$b_1 \mathbin{\&\&} b_2$	`S:` c_1
	 `jz E`
	 `pop`
	 c_2
	 `E: nop`	

		{ $\mathtt{now}_{\mathcal{S}} \sim \mathtt{now}_{\mathcal{C}} \wedge \mathtt{now}_{\mathcal{C}}\mathtt{.pc = E+1}$ $\wedge\ \mathtt{fullyEvaluated(now}_{\mathcal{S}}\mathtt{.evalFocus)}$ $\wedge\ \mathtt{(old\ now}_{\mathcal{S}}\mathtt{).context = now}_{\mathcal{S}}\mathtt{.context}$ $\wedge\ \mathtt{let\ v = valueOf(now}_{\mathcal{S}}\mathtt{.evalFocus)\ in}$ $\mathtt{now}_{\mathcal{C}}\mathtt{.stack = [v] + old\ now}_{\mathcal{C}}\mathtt{.stack}$ }

$b \in \mathcal{S}$:	$c \in \mathcal{C}$:	
		`{` $now_{\mathcal{S}} \sim now_{\mathcal{C}} \wedge now_{\mathcal{C}}$`.pc = S` $\wedge$ $now_{\mathcal{S}}$`.evalFocus = "`b_1 `&&` b_2`"` `}`
	`S :` c_1	$\mathrm{STEP}_{\mathcal{S}}$`();` `(*` $now_{\mathcal{S}}$`.evalFocus = "`b_1`" *)`
		π_1`();` `(* témoin "`b_1`"/ "`c_1`" *)`
	`jz E`	
b_1 && b_2	`pop`	
	c_2	
	`E : nop`	
		`{` $now_{\mathcal{S}} \sim now_{\mathcal{C}} \wedge now_{\mathcal{C}}$`.pc = E+1` $\wedge$ `fullyEvaluated(`$now_{\mathcal{S}}$`.evalFocus)` $\wedge$ `(old` $now_{\mathcal{S}}$`).context =` $now_{\mathcal{S}}$`.context` $\wedge$ `let v = valueOf(`$now_{\mathcal{S}}$`.evalFocus) in` $now_{\mathcal{C}}$`.stack = [v] + old` $now_{\mathcal{C}}$`.stack }`

$b \in \mathcal{S}$:	$c \in \mathcal{C}$:	
		$\{\ now_{\mathcal{S}} \sim now_{\mathcal{C}} \wedge now_{\mathcal{C}}.pc = S$ $\wedge\ now_{\mathcal{S}}.evalFocus = "b_1\ \&\&\ b_2"\ \}$
		$STEP_{\mathcal{S}}();$ (* $now_{\mathcal{S}}.evalFocus = "b_1"$ *)
	S : c_1	$\pi_1();$ (* témoin "b_1"/ "c_1" *)
	jz E	$STEP_{\mathcal{C}}();$ (* jz E *)
b_1 && b_2	pop	
	c_2	
	E : nop	
		$\{\ now_{\mathcal{S}} \sim now_{\mathcal{C}} \wedge now_{\mathcal{C}}.pc = E+1$ $\wedge\ fullyEvaluated(now_{\mathcal{S}}.evalFocus)$ $\wedge\ (\mathbf{old}\ now_{\mathcal{S}}).context = now_{\mathcal{S}}.context$ $\wedge\ \mathbf{let}\ v = valueOf(now_{\mathcal{S}}.evalFocus)\ \mathbf{in}$ $now_{\mathcal{C}}.stack = [v] + \mathbf{old}\ now_{\mathcal{C}}.stack\ \}$

$b \in \mathcal{S}$:	$c \in \mathcal{C}$:	
		{ $\mathtt{now}_{\mathcal{S}} \sim \mathtt{now}_{\mathcal{C}} \wedge \mathtt{now}_{\mathcal{C}}.\mathtt{pc} = \mathtt{S}$ $\wedge\ \mathtt{now}_{\mathcal{S}}.\mathtt{evalFocus} =$ "b_1 && b_2" }
		$\mathtt{STEP}_{\mathcal{S}}()$; (* $\mathtt{now}_{\mathcal{S}}.\mathtt{evalFocus} =$ "b_1" *)
	S : c_1	$\pi_1()$; (* témoin "b_1"/ "c_1" *)
	jz E	$\mathtt{STEP}_{\mathcal{C}}()$; (* jz E *)
b_1 && b_2	pop	if($\mathtt{now}_{\mathcal{C}}.\mathtt{pc}$ = E)
	c_2	
	E : nop	
		{ $\mathtt{now}_{\mathcal{S}} \sim \mathtt{now}_{\mathcal{C}} \wedge \mathtt{now}_{\mathcal{C}}.\mathtt{pc} = \mathtt{E+1}$ $\wedge$ fullyEvaluated($\mathtt{now}_{\mathcal{S}}$.evalFocus) $\wedge$ (old $\mathtt{now}_{\mathcal{S}}$).context = $\mathtt{now}_{\mathcal{S}}$.context $\wedge$ let v = valueOf($\mathtt{now}_{\mathcal{S}}$.evalFocus) in $\mathtt{now}_{\mathcal{C}}$.stack = [v] + old $\mathtt{now}_{\mathcal{C}}$.stack }

$b \in \mathcal{S}$:	$c \in \mathcal{C}$:	
		{ $\texttt{now}_{\mathcal{S}} \sim \texttt{now}_{\mathcal{C}} \wedge \texttt{now}_{\mathcal{C}}\texttt{.pc = S}$ $\wedge\ \texttt{now}_{\mathcal{S}}\texttt{.evalFocus = "}b_1\ \texttt{\&\&}\ b_2\texttt{"}$ }
		$\texttt{STEP}_{\mathcal{S}}\texttt{();}$ (* $\texttt{now}_{\mathcal{S}}\texttt{.evalFocus = "}b_1\texttt{"}$ *)
	S : c_1	$\pi_1\texttt{();}$ (* témoin "b_1"/ "c_1" *)
	jz E	$\texttt{STEP}_{\mathcal{C}}\texttt{();}$ (* jz E *)
b_1 && b_2	pop	**if**($\texttt{now}_{\mathcal{C}}\texttt{.pc = E}$)
	c_2	**then** $\texttt{STEP}_{\mathcal{S}}\texttt{()}$
	E : nop	
		{ $\texttt{now}_{\mathcal{S}} \sim \texttt{now}_{\mathcal{C}} \wedge \texttt{now}_{\mathcal{C}}\texttt{.pc = E+1}$ $\wedge\ \texttt{fullyEvaluated(now}_{\mathcal{S}}\texttt{.evalFocus)}$ $\wedge\ \texttt{(old now}_{\mathcal{S}}\texttt{).context = now}_{\mathcal{S}}\texttt{.context}$ $\wedge\ \texttt{let v = valueOf(now}_{\mathcal{S}}\texttt{.evalFocus) in}$ $\texttt{now}_{\mathcal{C}}\texttt{.stack = [v] + old now}_{\mathcal{C}}\texttt{.stack}$ }

$b \in \mathcal{S}$:	$c \in \mathcal{C}$:	
b_1 && b_2	S : c_1 jz E pop c_2 E : nop	see below

```
{ now_S ~ now_C ∧ now_C.pc = S
  ∧ now_S.evalFocus = "b1 && b2" }

STEP_S();        (* now_S.evalFocus = "b1" *)
π1();            (* témoin "b1"/ "c1" *)
STEP_C();        (* jz E *)
if(now_C.pc = E)
then STEP_S()
else (STEP_C(); STEP_S(); π2());


{ now_S ~ now_C ∧ now_C.pc = E+1
  ∧ fullyEvaluated(now_S.evalFocus)
  ∧ (old now_S).context = now_S.context
  ∧ let v = valueOf(now_S.evalFocus) in
     now_C.stack = [v] + old now_C.stack }
```

$b \in \mathcal{S}$:	$c \in \mathcal{C}$:	
		$\{$ `now`$_\mathcal{S}$ $\sim$ `now`$_\mathcal{C}$ $\wedge$ `now`$_\mathcal{C}$`.pc = S` $\wedge$ `now`$_\mathcal{S}$`.evalFocus = "`b_1 `&&` b_2`"` $\}$
	`S :` c_1	`STEP`$_\mathcal{S}$`();` `(* now`$_\mathcal{S}$`.evalFocus = "`b_1`" *)`
	`jz E`	π_1`();` `(* témoin "`b_1`"/ "`c_1`" *)`
b_1 && b_2	`pop`	`STEP`$_\mathcal{C}$`();` `(* jz E *)`
	c_2	`if(now`$_\mathcal{C}$`.pc = E)` `then STEP`$_\mathcal{S}$`()` `else (STEP`$_\mathcal{C}$`(); STEP`$_\mathcal{S}$`();` π_2`());`
	`E : nop`	`STEP`$_\mathcal{C}$`();` `(* nop *)`
		$\{$ `now`$_\mathcal{S}$ $\sim$ `now`$_\mathcal{C}$ $\wedge$ `now`$_\mathcal{C}$`.pc = E+1` $\wedge$ `fullyEvaluated(now`$_\mathcal{S}$`.evalFocus)` $\wedge$ `(old now`$_\mathcal{S}$`).context = now`$_\mathcal{S}$`.context` $\wedge$ `let v = valueOf(now`$_\mathcal{S}$`.evalFocus) in now`$_\mathcal{C}$`.stack = [v] + old now`$_\mathcal{C}$`.stack` $\}$

$b \in \mathcal{S}$:

b_1 && b_2

$c \in \mathcal{C}$:

```
S :  c1
     jz  E
     pop
     c2
E :  nop
```

$\{\ \mathtt{now}_{\mathcal{S}} \sim \mathtt{now}_{\mathcal{C}} \wedge \mathtt{now}_{\mathcal{C}}.\mathtt{pc} = \mathtt{S}$
$\quad \wedge\ \mathtt{now}_{\mathcal{S}}.\mathtt{evalFocus} = \text{"}b_1 \ \&\& \ b_2\text{"}\ \}$

$\mathtt{STEP}_{\mathcal{S}}();$ $\qquad$ (* $\mathtt{now}_{\mathcal{S}}.\mathtt{evalFocus} = \text{"}b_1\text{"}$ *)
$\pi_1();$ $\qquad$ (* témoin "b_1"/ "c_1" *)
$\mathtt{STEP}_{\mathcal{C}}();$ $\qquad$ (* jz E *)
$\mathtt{if}(\mathtt{now}_{\mathcal{C}}.\mathtt{pc} = \mathtt{E})$
$\mathtt{then}\ \mathtt{STEP}_{\mathcal{S}}()$
$\mathtt{else}\ (\mathtt{STEP}_{\mathcal{C}}();\ \mathtt{STEP}_{\mathcal{S}}();\ \pi_2());$
$\mathtt{STEP}_{\mathcal{C}}();$ $\qquad$ (* nop *)

$\{\ \mathtt{now}_{\mathcal{S}} \sim \mathtt{now}_{\mathcal{C}} \wedge \mathtt{now}_{\mathcal{C}}.\mathtt{pc} = \mathtt{E+1}$
$\quad \wedge\ \mathtt{fullyEvaluated}(\mathtt{now}_{\mathcal{S}}.\mathtt{evalFocus})$
$\quad \wedge\ (\mathtt{old}\ \mathtt{now}_{\mathcal{S}}).\mathtt{context} = \mathtt{now}_{\mathcal{S}}.\mathtt{context}$
$\quad \wedge\ \mathtt{let}\ \mathtt{v} = \mathtt{valueOf}(\mathtt{now}_{\mathcal{S}}.\mathtt{evalFocus})\ \mathtt{in}$
$\qquad \mathtt{now}_{\mathcal{C}}.\mathtt{stack} = [\mathtt{v}] + \mathtt{old}\ \mathtt{now}_{\mathcal{C}}.\mathtt{stack}\ \}$

Gains :

- Preuve des règles de compilation facilitée par le calcul de WP
  - preuves à la portée des démonstrateurs automatiques

- Travail laissé à l'utilisateur :
  - démontrer la propriété de transfert pour $\mathcal{W}$
  - définir des spécifications pour les témoins de simulation
  - définir les témoins de simulation ($\pi \in \mathcal{W}$)

Nous avons vérifié en Why3 un petit compilateur
[Clochard, Gondelman, JFLA 2015]

- traduit un langage `While` vers une machine à pile
- preuve basée sur une version préliminaire des idées présentées

1. Preuve par « débogage »
2. Aller plus loin : technique de preuve d'un compilateur
3. Extension aux comportements infinis
4. Résumé des contributions & perspectives

Comment étendre la preuve aux comportements infinis ?

- transférer les traces d'exécution
- mêmes effets observables (entrées/sorties)

Pour que la même approche fonctionne, il faut :

- étendre $\mathcal{W}$ avec des constructions qui gèrent la divergence
- pouvoir spécifier le comportement des exécutions infinies

Étape 1 : utiliser des bornes supérieures comme limites

- état «après» un nombre infini d'étapes $= \sup_{n \in \mathbb{N}} \texttt{now}_n$
- les étapes doivent respecter un ordre strict de progression
- plus général : état = historique, limite = historique infini
- plus précis : état + historique de ce qui est observable

Étape 1 : utiliser des bornes supérieures comme limites

- état «après» un nombre infini d'étapes $= \sup_{n\in\mathbb{N}} \mathtt{now}_n$
- les étapes doivent respecter un ordre strict de progression
- plus général : état = historique, limite = historique infini
- plus précis : état + historique de ce qui est observable

Étape 2 : permettre de rattraper la non-terminaison

- la non-terminaison est vue comme une exception
- on ajoute un gestionnaire aux constructions `while`/`rec`
- l'état est la borne supérieure des itérations/de la pile d'appel

Remplacer le variant par une valeur de progression

- augmente strictement à chaque étape
- restreint les comportements infinis possibles

```
while B do
  invariant { I }
  progress  { V, ≺ }
  s1

done
```

Remplacer le variant par une valeur de progression

- augmente strictement à chaque étape
- restreint les comportements infinis possibles

Gestionnaire de divergence : paramétré par une séquence d'états

- séquence strictement croissante pour la valeur de progression
- les états de la séquence satisfont l'invariant de boucle
- à la fin du gestionnaire, sort de la boucle

while B do
  invariant { I }
  progress { V, $\prec$ }
  s_1
at_infinity_and_beyond$(\texttt{now}_n)_{n \in \mathbb{N}} \rightarrow s_2$
done

Plus faible pré-condition pour la post-condition Q et $\mathtt{now}$:

- $\mathtt{I}(\mathtt{now})$
- $\forall \mathtt{now}_0.\ \mathtt{B}(\mathtt{now}_0) \land \mathtt{I}(\mathtt{now}_0) \Rightarrow$
  $\mathtt{now}_0 \in \mathit{WP}(s_1, \{\mathtt{now}_1 \mid \mathtt{I}(\mathtt{now}_1) \land \mathtt{V}(\mathtt{now}_0) \prec \mathtt{V}(\mathtt{now}_1)\})$
- $\forall \mathtt{now}_0.\ \neg\mathtt{B}(\mathtt{now}_0) \land \mathtt{I}(\mathtt{now}_0) \Rightarrow \mathtt{now}_0 \in Q$
- $\forall (\mathtt{now}_n)_{n\in\mathbb{N}}.$
  $((\forall nm \in \mathbb{N}.\ n < m \rightarrow \mathtt{V}(\mathtt{now}_n) \prec \mathtt{V}(\mathtt{now}_m))$
  $\land\ (\forall n \in \mathbb{N}.\ \mathtt{I}(\mathtt{now}_n))$
  $\land\ \mathtt{now}_0 = \mathtt{now})$
  $\Rightarrow (\sup_{n\in\mathbb{N}} \mathtt{now}_n) \in \mathit{WP}(s_2, Q)$

Nous utilisons la même idée pour la construction récursive

- séquence = pile d'appel infinie
- le gestionnaire doit terminer le premier appel récursif ($\texttt{now}_0$)

$$
\begin{array}{l}
\texttt{let rec f ()} \\
\quad \texttt{requires \{ P \}} \\
\quad \texttt{ensures \{ Q \}} \\
\quad \texttt{progress \{ V, } \prec \texttt{ \}} \\
\texttt{= } s_1 \\
\texttt{at\_infinity\_and\_beyond}(\texttt{now}_n)_{n\in\mathbb{N}} \rightarrow s_2 \\
\texttt{in ...}
\end{array}
$$

Résultat souhaité :

Propriété de transfert

Toute propriété au sujet de `now` établie dans $\mathcal{W}$ implique
la même propriété pour l'exécution dans le(s) langage(s) d'étude.

Difficulté : prouver cette propriété pour `while`/`rec` étendus

Formalisation effectuée dans le cadre des jeux :

- l'état du jeu correspond à `now`
- les transitions suivent une structure $\exists - \forall$:
  - joueur $\exists$: correspond aux choix contrôlés par $\mathcal{W}$
    - = arguments de `STEP`
    - exemple : preuve d'accessibilité
  - joueur $\forall$: correspond au non-déterminisme non contrôlé
    - = résultats possibles de `STEP`
    - exemple : entrées du programme
- les états du jeu sont ordonnés, les transitions sont croissantes
- cas limites : les parties infinies reprennent à la borne supérieure

Théorème de transfert

Pour tous

- jeu $\mathbb{G}$
- programme π de $\mathcal{W}_{\mathbb{G}}$
- Q sous-ensemble du domaine de $\mathbb{G}$
- $x \in WP(\pi, Q)$

il existe une stratégie gagnante avec mémoire pour le joueur $\exists$ qui depuis x atteint un élément de Q.

Existence d'une stratégie gagnante $\approx$ triplet de Hoare

Propriété de transfert $\approx$ correction du calcul de WP

Généralisation à un contexte :

- variables locales (correspondent aux paramètres auxiliaires)
- procédures locales : pour traiter la récursion

Théorème auxiliaire : simulation entre jeux

- donne la construction itérative comme cas particulier

Pour le cas de la construction récursive :

- enrichissement du jeu pour simuler les appels récursifs
- appels récursifs déroulés quand exploités
- repose de manière critique sur l'alternance arbitraire $\exists/\forall$

J'ai mécanisé une variante de ce théorème en Why3

- logique de Hoare plutôt que plus faible pré-conditions
- sur papier : passage au calcul de WP

	manuscrit (nb pages)	Why3 (loc)	Conditions de vérification
jeux	13	1641	915
simulation	16	2259	1737
logique de Hoare	15	1273	1136
lien avec sémantique petits pas	8	776	518
total	**52**	**5949**	**4306**

Application principale : preuve de compilateur

Dans la thèse (sur papier) : éléments pour prouver
un compilateur de taille significative

- source : langage `While` étendu avec des fonctions récursives
  et des pointeurs de fonctions
- cible : machine à pile
- établit la simulation avant et arrière
- correspondance des comportements infinis
- en particulier, correspondance des traces entrées/sorties
- pas besoin de corrélation à l'échelle de la pile

1. Preuve par « débogage »
2. Aller plus loin : technique de preuve d'un compilateur
3. Extension aux comportements infinis
4. Résumé des contributions & perspectives

Logique de programme basée sur les jeux

- comportements infinis, non-déterminisme
- application : preuves de correction/simulation

Mécanisme léger de preuve déclarative [Clochard, JFLA 2017]

- basé sur des indicateurs de coupures (`by`/`so`)
- pierre angulaire de la mécanisation des jeux

Preuves via calcul

- générer les conditions de vérification pour le langage dédié $\mathcal{W}$
- preuve par réflexion : multiplication matrices de Strassen [Clochard, Gondelman & Pereira, JAR 2018]

Méthode pour une classe particulière de débordements entiers [Clochard, Filliâtre & Paskevich, VSTTE 2015]

- absence de débordement avant 2^{64} étapes de calcul

Mise en pratique :

- implémenter la «preuve par débogage» dans un outil
  - principalement automatiser la génération de `CONT`
- mécaniser la preuve de compilateur effectuée sur papier

Problèmes ouverts :

- nécessité de la mémoire pour les stratégies ?
- extension du théorème de transfert pour les jeux ?
  - pour l'instant limité aux programmes du premier ordre dans $\mathcal{W}$
  - motif problématique d'ordre supérieur :
    `let rec f g = ... (f (fun x → ... (f ...)))`

Remarque : $\mathcal{S}$ et/ou $\mathcal{C}$ peuvent être non-déterministes

- décisions non-déterministes potentiellement mal corrélées
- de même pour les exécutions

Solution : contrôler les décisions non-déterministes pour $\mathcal{S}$

- garder $\mathtt{STEP}_{\mathcal{C}}$ tel quel
- récupérer ces décisions à travers $\mathtt{now}_{\mathcal{C}}$
- passer ces décisions comme un argument à $\mathtt{STEP}_{\mathcal{S}}$

Le rôle du non-déterminisme change le type de résultat que l'on peut obtenir

Autres contributions : débordements arithmétiques

Méthode pour une classe particulière de débordements entiers
[Clochard, Filliâtre & Paskevich, VSTTE 2015]

- taille collection, rangs union-find, hauteurs AVL ...
- idée : pas assez de temps pour qu'il y ait un débordement
- les entiers augmentent d'au plus 1 par unité de temps
- entiers 64 bits, unité de temps 1 ns : $\approx$ 584 ans

Garanti par typage :

```
type peano = private { v: int }
val zero
val succ (x: peano) : peano
```

```
n = 0;
while(n ≥ 0) { n = n + read_paren(); }
```

- `read_paren()` : lit une parenthèse d'un flux d'entrée, infini et non-déterministe
- '(' est 1, ')' est -1

Spécification :

```
n = 0;
while(n ≥ 0) { n = n + read_paren(); }
```

- `read_paren()` : lit une parenthèse d'un flux d'entrée,
  infini et non-déterministe
- '(' est 1, ')' est -1

Spécification : soit le programme

- termine en lisant w), avec w bien parenthésé

```
n = 0;
while(n ≥ 0) { n = n + read_paren(); }
```

- `read_paren()` : lit une parenthèse d'un flux d'entrée,
  infini et non-déterministe
- '(' est 1, ')' est -1

Spécification : soit le programme

- termine en lisant $w)$, avec w bien parenthésé
- diverge en lisant $w_0(w_1(\dots w_n(\dots$

```
n = 0;
while(n ≥ 0) { n = n + read_paren(); }
```

- `read_paren()` : lit une parenthèse d'un flux d'entrée,
  infini et non-déterministe
- '(' est 1, ')' est -1

Spécification : soit le programme

- termine en lisant $w)$, avec w bien parenthésé
- diverge en lisant $w_0(w_1(\dots w_n(\dots$
- diverge en lisant $w_0(w_1(\dots w_n((w_{n+1})(w_{n+2}) \dots (w_{n+m}) \dots$

```
n = 0;
while(n ≥ 0) { n = n + read_paren(); }
```

- Historique de lecture conservé dans l'état
- Établir la correction partielle : facile par «débogage»

```
n = 0;
while(n ≥ 0) { L: n ← n + read_paren(); }
```

- Historique de lecture conservé dans l'état
- Établir la correction partielle : facile par «débogage»

```
CONT();
let rec parse ()

= while true do

    CONT(); if last(now.read) = ')' then break;
    parse()
  done
in parse ()
```

```
n = 0;
while(n ≥ 0) { L: n ← n + read_paren(); }
```

- Historique de lecture conservé dans l'état
- Établir la correction partielle : facile par «débogage»

```
CONT();
let rec parse ()
  ensures { ∃w∈Dyck. now.read = old now.read + w + ')' }
= while true do
    invariant { ∃w∈Dyck. now.read = old now.read + w }
    CONT(); if last(now.read) = ')' then break;
    parse()
  done
in parse ()
```

```
n = 0;
while(n ≥ 0) { L: n ← n + read_paren(); }
```

- Historique de lecture conservé dans l'état
- Établir la correction partielle : facile par «débogage»
- Correspondance entre les cas de divergence et `while`/`rec`

```
CONT();
let rec parse ()
  ensures { ∃w∈Dyck. now.read = old now.read + w + ')' }
= while true do
    invariant { ∃w∈Dyck. now.read = old now.read + w }
    CONT(); if last(now.read) = ')' then break;
    parse()
  done
in parse ()
```

```
CONT();
let rec parse ()
  ensures { ∃w∈Dyck. now.read = old now.read + w + ')' }
= while true do
    invariant { ∃w∈Dyck. now.read = old now.read + w }
    CONT(); if last(now.read) = ')' then break;
    parse()
  done
in parse ()
```

Cas :

- termine en lisant $w)$, avec w bien parenthésé
- diverge en lisant $w_0(w_1(\ldots w_n(\ldots$
- diverge en lisant $w_0(w_1(\ldots w_n((w_{n+1})(w_{n+2})\ldots(w_{n+m})\ldots$

```
CONT();
let rec parse ()
  ensures { ∃w∈Dyck. now.read = old now.read + w + ')' }
= while true do
    invariant { ∃w∈Dyck. now.read = old now.read + w }
    CONT(); if last(now.read) = ')' then break;
    parse()
  done
in parse ()
```

Valeurs ordonnées de progression :

- récursion : différence now/old now = séquence de mots $w($, avec w bien parenthésé
- boucle : différence now/old now = séquence de mots (w), avec w bien parenthésé

```
CONT();
let rec parse ()
  ensures { (∃w∈Dyck. now.read = old now.read + w + ')')
    ∨ ∃(w_m)_{m∈ℕ}. now.read = old now.read + w_0(w_1(...w_n(...
      ∨ ∃n. ... + w_0(w_1(...(w_n((w_{n+1})(w_{n+2})...(w_{n+m})... }
  progress { now.read, ≺_rec }
= while true do
    invariant { ∃w∈Dyck. now.read = old now.read + w }
    progress { now.read, ≺_iter }
    CONT(); if last(now.read) = ')' then break;
    parse (); if infinite(now.read) then break;
  at_infinity_and_beyond (now_n)_{n∈ℕ} → ()
  done
at_infinity_and_beyond (now_n)_{n∈ℕ} → ()
in parse ()
```