Analyse thread-modulaire

construction d'abstractions relationnelles d'interférences

Raphaël Monat

Stage effectué dans l'équipe ANTIQUE, ENS Ulm
Sous la direction d'Antoine Miné

1. Motivations
2. Interprétation abstraite
3. Analyse de programmes parallèles
4. Contribution
5. Résultats
6. Conclusion

1. Motivations
2. Interprétation abstraite
3. Analyse de programmes parallèles
4. Contribution
5. Résultats
6. Conclusion

Motivations

FIGURE 1 – Crash de la première Ariane V en 1996

Motivations

- Tester ne suffit pas sur de gros projets
- Avec de l'analyse statique, possibilité de certifier un logiciel
- Très recherché en avionique et aérospatial

1 Motivations

2 Interprétation abstraite
- Introduction à l'interprétation abstraite
- Domaines numériques relationnels

3 Analyse de programmes parallèles

4 Contribution

5 Résultats

6 Conclusion

- Analyse totalement automatique.
- Calcul de surapproximations pour garantir la complétude.

Exemple

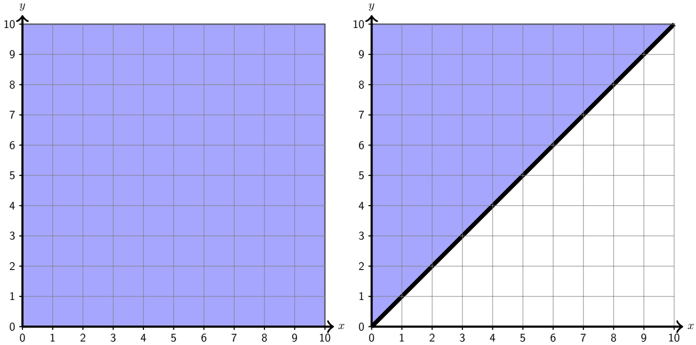
- Communications entre le monde concret et le monde abstrait.
- Notion de domaine numérique

$\{1, 2, 5\} \rightarrow + \rightarrow \mathbb{N}$
$\{1, 2, 5\} \rightarrow [1, 5] \rightarrow \{1, 2, 3, 4, 5\}$

Les domaines relationnels (tels que les polyèdres) permettent d'inférer des relations du type $\sum \alpha_i X_i \leq \beta$ entre variables :

```
assume 0 <= x <= 10;
assume 0 <= y <= 10;
if (x > y) then
  y = x;
endif
```

Les domaines relationnels (tels que les polyèdres) permettent d'inférer des relations du type $\sum \alpha_i X_i \leq \beta$ entre variables :



(a) A la fin de la ligne 2 (b) A la fin de la ligne 5

FIGURE 2 – Polyèdres lors du calcul de $y = \max(x, y)$

Il existe aussi des domaines faiblement relationnels tels que les octogones.

1 Motivations

2 Interprétation abstraite

3 Analyse de programmes parallèles
- Modèle d'exécution
- Notion d'interférences
- Analyse par "séquentialisation" de l'exécution
- Analyse modulaire

4 Contribution

5 Résultats

6 Conclusion

- les évaluations d'expressions booléennes sont atomiques
- les assignations d'expressions arithmétiques sont aussi atomiques

On parle de mémoire à cohérence séquentielle.

```
if not (x == 0) then
  y = y / x;
```

||

```
x = 0;
```

```
x = 0;
if not (x == 0) then
y = y / x;
```

```
if not (x == 0) then
  x = 0;
  y = y / x;
```

```
if not (x == 0) then
  y = y / x;
  x = 0;
```

Définition

Une interférence est une modification d'une variable par un thread, pouvant avoir des conséquences sur l'exécution d'autres threads.

Exemple

L'assignation $x = 0$ dans l'exemple ci-dessus.

- On séquentialise l'exécution, et on analyse toutes les combinaisons
- Facile à faire avec un analyseur séquentiel
- Mais coûteux

- On se base sur une approche dite de Rely-Guarantee [1], développée par Jones
- Rely-guarantee : $R, G \vdash \{P\}$ *statement* $\{Q\}$
- Cette méthode a ensuite été adaptée à l'interprétation abstraite [2]

1. Cliff B Jones. "Development methods for computer programs including a notion of interference". Thèse de doct. Oxford University Computing Laboratory, juin 1981.
2. Antoine Miné. "Static analysis of run-time errors in embedded real-time parallel C programs". In : *Logical Methods in Computer Science (LMCS)* (mar. 2012).

En interprétation abstraite, cela veut dire que l'on itère le processus suivant :

1. Trouver les interférences créées par chaque thread
2. Analyser les threads en prenant en compte les interférences

Jusqu'à atteindre un résultat d'analyse stable.

Exemple

```
var flag:int;
initial flag == 1;
```

```
thread t1:
begin
while(true) do
  while (flag != 1) do done;
  flag = 2;
done;
end
```

```
thread t2:
begin
while(true) do
  while (flag != 2) do done;
  flag = 1;
done;
end
```

Exemple

```
var flag:int;
initial flag == 1;
```

```
thread t1:
begin
while(true) do
  while (flag != 1) do
    done;
  flag = 2;
done;
end
```

```
thread t2:
begin
while(true) do
  while (flag != 2) do
    done;
  flag = 1;
done;
end
```

flag : $1 \rightsquigarrow 2$	$\perp$

Exemple

```
var flag:int;
initial flag == 1;
```

```
thread t1:
begin
while(true) do
  while (flag != 1) do
    done;
  flag = 2;
done;
end
```

```
thread t2:
begin
while(true) do
  while (flag != 2) do
    done;
  flag = 1;
done;
end
```

flag : $1 \rightsquigarrow 2$	$\perp$
flag : $1 \rightsquigarrow 2$	flag : $2 \rightsquigarrow 1$

1. Motivations
2. Interprétation abstraite
3. Analyse de programmes parallèles
4. Contribution
   - Analyse relationnelle
   - Implémentation de BATMAN
   - Comparaison avec CONCURINTERPROC
5. Résultats
6. Conclusion

```
while (true) do
  if (x < y) then
    x = x + 1;
  endif;
done;
```

||

```
while (true) do
  if (y < 10) then
    y = y + 1;
  endif;
done;
```

- Des analyses non relationnelles avaient déjà été proposées [3]
- Des combinaison de domaines permettaient d'avoir certains résultats [4]
- Comment formaliser les relations avec les domaines numériques existants ?
- $x : 1 \rightsquigarrow 2$, transformé en $x = 1, x' = 2$
- $x' > x$
- Avantage de cette méthode : elle est bien plus générale

3. Antoine Miné. "Static analysis of run-time errors in embedded real-time parallel C programs". In : *Logical Methods in Computer Science (LMCS)* (mar. 2012).

4. Antoine Miné. "Relational thread-modular static value analysis by abstract interpretation". In : *Proc. of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'14)*. Lecture Notes in Computer Science (LNCS). San Diego, California, USA : Springer, jan. 2014.

BAsic **T**hread-**M**odular **AN**alyzer

- En OCaml
- Utilise Apron
- De l'ordre de 1700 lignes de code
- Langage jouet

CONCURINTERPROC

- Développé par Bertrand Jeannet[5] (INRIA Grenoble)
- Séquentialise le programme
- Sait aussi analyser les procédures

5. Bertrand JEANNET. "Relational interprocedural verification of concurrent programs". In : *Software & Systems Modeling* (2013).

1 Motivations

2 Interprétation abstraite

3 Analyse de programmes parallèles

4 Contribution

5 Résultats
- Exemple
- Exclusion mutuelle
- Passage à l'échelle

6 Conclusion

```
var x:int,y:int;
initial x == 0 and y == 0;
```

```
thread t1:
begin
  while (true) do
    if (y < 10) then
      y = y + 1;
    endif;
  done;
end
```

```
thread t2:
begin
  while (true) do
    if (x < y) then
      x = x + 1;
    endif;
  done;
end
```

```
var x:int,y:int;
initial x == 0 and y == 0;
```

```
thread t1:
begin
  while (true) do
    if (y < 10) then
      y = y + 1;
    endif;
  done;
end
```

```
thread t2:
begin
  while (true) do
    if (x < y) then
      x = x + 1;
    endif;
  done;
end
```

$y' = y + 1,\ x = 0,\ x' = 0,$ $0 \leq y \leq 9$	$\perp$

```
var x:int,y:int;
initial x == 0 and y == 0;
```

```
thread t1:
begin
  while (true) do
    if (y < 10) then
      y = y + 1;
    endif;
  done;
end
```

```
thread t2:
begin
  while (true) do
    if (x < y) then
      x = x + 1;
    endif;
  done;
end
```

$y' = y + 1,\ x = 0,\ x' = 0,$ $0 \le y \le 9$	$\perp$
$y' = y + 1,\ x = 0,\ x' = 0,$ $0 \le y \le 9$	$x' = x + 1,\ x \ge 0,\ y' = y,$ $x + 1 \le y$

```
var x:int ,y:int ;
initial x == 0 and y == 0;
```

```
thread t1:
begin
  while (true) do
    if (y < 10) then
      y = y + 1;
    endif;
  done;
end
```

```
thread t2:
begin
  while (true) do
    if (x < y) then
      x = x + 1;
    endif;
  done;
end
```

$y' = y + 1,\ x = 0,\ x' = 0,$ $0 \leq y \leq 9$	$\perp$
$y' = y + 1,\ x = 0,\ x' = 0,$ $0 \leq y \leq 9$	$x' = x + 1,\ x \geq 0,\ y' = y,$ $x + 1 \leq y$
$y' = y + 1,\ x' = x,\ x \leq y$ $0 \leq y \leq 9$	$x' = x + 1,\ x \geq 0,\ y' = y,$ $x + 1 \leq y$

- Impossible de prouver des exclusions mutuelles classiques
- Des algorithmes assez similaires sont correctement analysés :

```
var flag:int;
initial flag == 1;
```

```
thread t1:
begin
while(true) do
  while (flag != 1) do
    done;
  flag = 2;
done;
end
```

```
thread t2:
begin
while(true) do
  while (flag != 2) do
    done;
  flag = 1;
done;
end
```

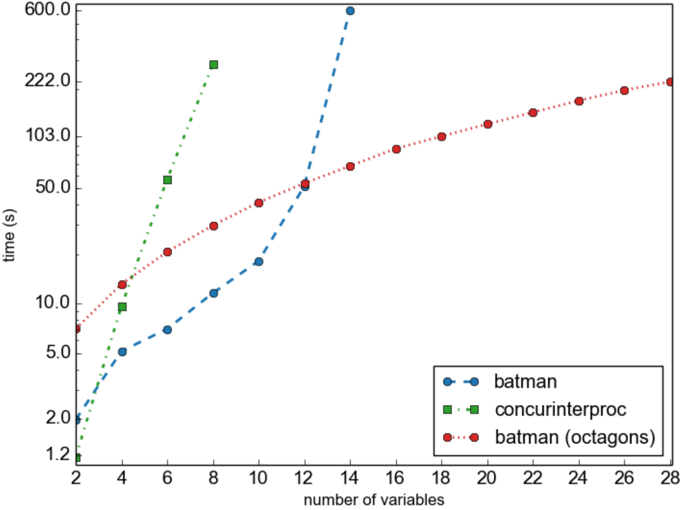
Deux types de passage à l'échelle

- par rapport au nombre de variables ✗

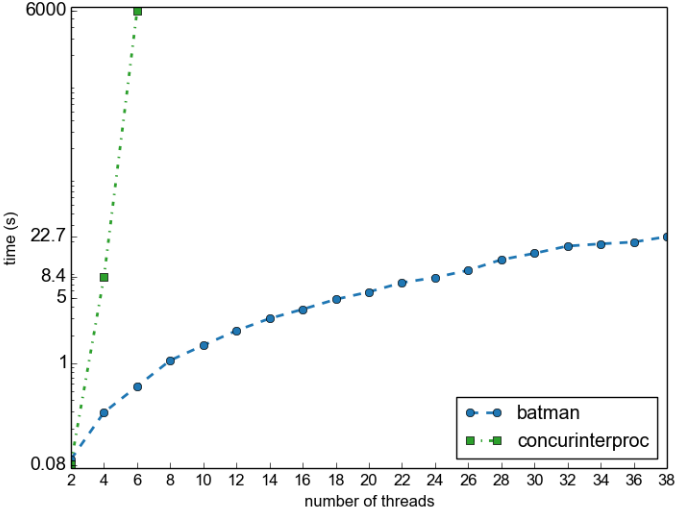
Deux types de passage à l'échelle

- par rapport au nombre de variables ✗
- par rapport au nombre de threads ✓

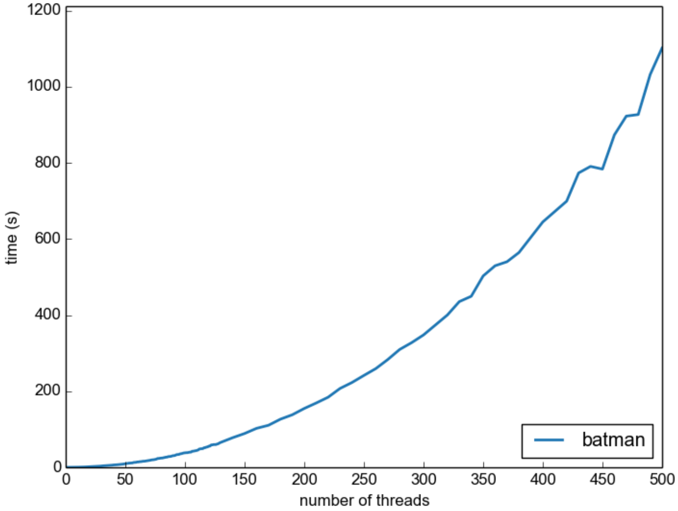
Par rapport au nombre de variables :



Par rapport au nombre de threads :



Par rapport au nombre de threads :



1. Motivations
2. Interprétation abstraite
3. Analyse de programmes parallèles
4. Contribution
5. Résultats
6. Conclusion

Conclusion

- Développement d'une sémantique dénotationnelle décrivant une analyse relationnelle et modulaire
- Développement d'un prototype pour pouvoir tester les performances de cette méthode
- Tests de performance : la méthode modulaire est très satisfaisante

7. Questions
   - Concrete semantics
   - Abstract semantics
   - “Clock” example

- Threads : $\mathcal{T}$
- Program labels : $\mathcal{L}$
- Variables : $\mathcal{V}$
- Control points : $\mathcal{C} = \mathcal{T} \to \mathcal{L}$
- Environments : $\mathcal{E} = \mathcal{V} \to \mathbb{Z}$
- Memory $\mathcal{M} = \mathcal{C} \times \mathcal{E}$
- Interference $\mathcal{I} \subseteq \mathcal{M}^2$

$$\mathbb{S}[\![\textit{stat}]\!]_t : \mathcal{M} \times \mathcal{I} \to \mathcal{M} \times \mathcal{I}$$

$$\mathbb{S}[\![^{l_1} X \leftarrow e^{l_2}]\!]_t(R, I) =$$

$$\mathbb{S}[\![\textit{stat}]\!]_t : \mathcal{M} \times \mathcal{I} \to \mathcal{M} \times \mathcal{I}$$

$$\mathbb{S}[\![^{l_1} X \leftarrow e^{l_2}]\!]_t(R, I) =$$

$$\text{let } I_1 = \{(c, \rho), (c[t \mapsto l_2], \rho[x \mapsto v]) \mid (c, \rho) \in R, v \in \mathbb{E}[\![e]\!]_t \rho\} \text{ in}$$

$$\mathbb{S}[\![\textit{stat}]\!]_t : \mathcal{M} \times \mathcal{I} \to \mathcal{M} \times \mathcal{I}$$

$$\mathbb{S}[\![^{l_1} X \leftarrow e^{l_2}]\!]_t(R, I) =$$
$$\text{let } I_1 = \{(c,\rho), (c[t \mapsto l_2], \rho[x \mapsto v]) \mid (c,\rho) \in R, v \in \mathbb{E}[\![e]\!]_t \rho\} \text{ in}$$
$$\text{let } R_1 = \{(c', \rho') \mid \exists (c,\rho), ((c,\rho),(c,\rho')) \in I_1\} \text{ in}$$

$$\mathbb{S}[\![\textit{stat}]\!]_t : \mathcal{M} \times \mathcal{I} \rightarrow \mathcal{M} \times \mathcal{I}$$

$$
\begin{aligned}
&\mathbb{S}[\![^{l_1} X \leftarrow e^{l_2}]\!]_t(R, I) = \\
&\text{let } I_1 = \{(c,\rho),(c[t \mapsto l_2], \rho[x \mapsto v]) \mid (c,\rho) \in R, v \in \mathbb{E}[\![e]\!]_t \rho\} \text{ in} \\
&\text{let } R_1 = \{(c',\rho') \mid \exists (c,\rho), ((c,\rho),(c,\rho')) \in I_1\} \text{ in} \\
&\text{let } R_2 = \operatorname{lfp} \lambda S.\ R_1 \cup \{(c',\rho') \mid (c,\rho) \in S, \\
&\qquad ((c,\rho),(c',\rho')) \in I, c'(t) = c(t)\} \text{ in}
\end{aligned}
$$

$$\mathbb{S}[\![\textit{stat}]\!]_t : \mathcal{M} \times \mathcal{I} \to \mathcal{M} \times \mathcal{I}$$

$$\begin{aligned}
&\mathbb{S}[\![^{l_1} X \leftarrow e^{l_2}]\!]_t(R, I) = \\
&\text{let } I_1 = \{(c,\rho),(c[t \mapsto l_2], \rho[x \mapsto v]) \mid (c,\rho) \in R, v \in \mathbb{E}[\![e]\!]_t\rho\} \text{ in} \\
&\text{let } R_1 = \{(c',\rho') \mid \exists (c,\rho), ((c,\rho),(c,\rho')) \in I_1\} \text{ in} \\
&\text{let } R_2 = \operatorname{lfp} \lambda S.\ R_1 \cup \{(c',\rho') \mid (c,\rho) \in S, \\
&\qquad ((c,\rho),(c',\rho')) \in I, c'(t) = c(t)\} \text{ in} \\
&\qquad R_2, I \cup I_1
\end{aligned}$$

$$\mathbb{B}[\![\textit{boolean expression}]\!]_t : \mathcal{M} \times \mathcal{I} \to \mathcal{M} \times \mathcal{I}$$

$\mathbb{B}[\![b]\!]_t(R, I) =$

$$\mathbb{B}[\![\textit{boolean expression}]\!]_t : \mathcal{M} \times \mathcal{I} \to \mathcal{M} \times \mathcal{I}$$

$$\mathbb{B}[\![b]\!]_t(R, I) =$$
$$\textsf{let } R_1 = \{(c, \rho) \in R \subseteq \mathcal{C} \times \mathcal{E} \mid \textsf{ true } \in \mathbb{E}[\![b]\!]_t \rho\} \textsf{ in}$$

$$\mathbb{B}[\![\textit{boolean expression}]\!]_t : \mathcal{M} \times \mathcal{I} \rightarrow \mathcal{M} \times \mathcal{I}$$

$$\begin{aligned}
&\mathbb{B}[\![b]\!]_t(R, I) = \\
&\qquad \textsf{let } R_1 = \{(c, \rho) \in R \subseteq \mathcal{C} \times \mathcal{E} \mid \textsf{ true } \in \mathbb{E}[\![b]\!]_t \rho\} \textsf{ in} \\
&\qquad \textsf{let } R_2 = \mathrm{lfp}\, \lambda S.\ R_1 \cup \{(c', \rho') \mid (c, \rho) \in S, \\
&\qquad\qquad ((c, \rho), (c', \rho')) \in I, c'(t) = c(t)\} \textsf{ in}
\end{aligned}$$

$$\mathbb{B}[\![\textit{boolean expression}]\!]_t : \mathcal{M} \times \mathcal{I} \to \mathcal{M} \times \mathcal{I}$$

$$\begin{aligned}
&\mathbb{B}[\![b]\!]_t(R, I) = \\
&\qquad \textsf{let } R_1 = \{(c, \rho) \in R \subseteq \mathcal{C} \times \mathcal{E} \mid \textsf{ true } \in \mathbb{E}[\![b]\!]_t \rho\} \textsf{ in} \\
&\qquad \textsf{let } R_2 = \operatorname{lfp} \lambda S.\ R_1 \cup \{(c', \rho') \mid (c, \rho) \in S, \\
&\qquad\qquad\qquad ((c, \rho), (c', \rho')) \in I, c'(t) = c(t)\} \textsf{ in} \\
&\qquad\qquad\qquad R_2, I
\end{aligned}$$

- Forget control points of other threads

- Forget control points of other threads

$$\alpha_R : \begin{cases} \mathcal{P}(\mathcal{M}) & \longrightarrow & \mathcal{L} \to \mathcal{P}(\mathcal{E}) \\ X & \longmapsto & \lambda L.\{e \mid (c, e) \in X \ \wedge c(t) = L\} \end{cases}$$

- Forget control points of other threads

$$\alpha_R : \begin{cases} \mathcal{P}(\mathcal{M}) & \longrightarrow & \mathcal{L} \to \mathcal{P}(\mathcal{E}) \\ X & \longmapsto & \lambda L.\{e \mid (c,e) \in X \ \wedge c(t) = L\} \end{cases}$$

- For each thread, get the transitions from a memory state to another one :

- Forget control points of other threads

$$\alpha_R : \begin{cases} \mathcal{P}(\mathcal{M}) & \longrightarrow & \mathcal{L} \to \mathcal{P}(\mathcal{E}) \\ X & \longmapsto & \lambda L.\{e \mid (c,e) \in X \ \wedge c(t) = L\} \end{cases}$$

- For each thread, get the transitions from a memory state to another one :

$$\alpha_I : \begin{cases} \mathcal{P}(\mathcal{I}) & \longrightarrow & \mathcal{T} \to \mathcal{P}(\mathcal{E}^2) \\ X & \longmapsto & \lambda t.\{(b,e) \in \mathcal{E}^2 \mid \exists (c_b, c_e) \in \mathcal{C}^2, \\ & & ((c_b, b), (c_e, e)) \in X, c_b(t) \neq c_e(t) \wedge \\ & & \forall t' \in \mathcal{T} \setminus \{t\}, c_b(t) = c_e(t)\} \end{cases}$$

Loss of precision :

```
var x:int;
initial x == 0;
```

```
thread t1:
begin
  x = x + 1;
end
```

```
thread t2:
begin
  x = x + 1;
end
```

Loss of precision :

```
var x:int;
initial x == 0;
```

```
thread t1:
begin
  x = x + 1;
end
```

```
thread t2:
begin
  x = x + 1;
end
```

$$x \geq 1$$

- Concretization into the memory domain

$$\gamma_{\mathcal{E}} : (\mathcal{L} \to \mathcal{D}^{\#}) \longrightarrow (\mathcal{L} \to \mathcal{P}(\mathcal{E}))$$

- Concretization into interferences, using a relational domain

$$\gamma_{\mathcal{ITF}} : (\mathcal{T} \to \mathcal{I}^{\#}) \longrightarrow (\mathcal{T} \to \mathcal{P}(\mathcal{E}^2))$$

$$\text{assign} : \mathcal{D}^\# \times \mathcal{P}(\mathcal{V} \times \mathcal{A}) \to \mathcal{D}^\#$$
$$\text{add} : \mathcal{D}^\# \times \mathcal{P}(\Sigma) \to \mathcal{D}^\#$$
$$\text{rename} : \mathcal{D}^\# \times \mathcal{P}(\Sigma^2) \to \mathcal{D}^\#$$
$$\text{delete} : \mathcal{D}^\# \times \mathcal{P}(\Sigma) \to \mathcal{D}^\#$$

- $\mathcal{D}_n^\#$ is the set of domains on $\mathcal{D}^\#$ with n variables

- $\mathcal{D}_n^\#$ is the set of domains on $\mathcal{D}^\#$ with n variables
- $\Delta_{2n}^\# \subseteq \mathcal{D}_{2n}^\#$ must satisfy the following property :
  $\exists X \subseteq \mathsf{Vars}(\Delta_{2n}^\#), \forall x \in X, x' \in \mathsf{Vars}(\Delta_{2n}^\#)$

- $\mathcal{D}_n^\#$ is the set of domains on $\mathcal{D}^\#$ with n variables
- $\Delta_{2n}^\# \subseteq \mathcal{D}_{2n}^\#$ must satisfy the following property :
  $\exists X \subseteq \mathsf{Vars}(\Delta_{2n}^\#), \forall x \in X, x' \in \mathsf{Vars}(\Delta_{2n}^\#)$
- $\mathsf{Var} : \mathcal{D}^\# \to \mathcal{P}(\Sigma)$ associates to each domain its variables

- $\mathcal{D}_n^{\#}$ is the set of domains on $\mathcal{D}^{\#}$ with n variables
- $\Delta_{2n}^{\#} \subseteq \mathcal{D}_{2n}^{\#}$ must satisfy the following property :
  $\exists X \subseteq \mathsf{Vars}(\Delta_{2n}^{\#}), \forall x \in X, x' \in \mathsf{Vars}(\Delta_{2n}^{\#})$
- $\mathsf{Var} : \mathcal{D}^{\#} \to \mathcal{P}(\Sigma)$ associates to each domain its variables
- $n = |\mathcal{V}|$

$$\text{extend} : \left\{ \begin{array}{lcl} \mathcal{D}_n^\# & \longrightarrow & \Delta_{2n}^\# \\ R^\# & \longmapsto & \textsf{let } R_1^\# = \textsf{add}(R^\#, \{x' \mid x \in \textsf{Vars}(R^\#)\}) \textsf{ in} \end{array} \right.$$

$$\text{extend} : \begin{cases} \mathcal{D}_n^\# & \longrightarrow & \Delta_{2n}^\# \\ R^\# & \longmapsto & \textsf{let } R_1^\# = \textsf{add}(R^\#, \{x' \mid x \in \textsf{Vars}(R^\#)\}) \textsf{ in} \end{cases}$$

$$\text{img} : \begin{cases} \Delta_{2n}^\# & \longrightarrow & \mathcal{D}_n^\# \\ R^\# & \longmapsto & \textsf{let } X = \{x \in \textsf{Vars}(R^\#) \mid x' \in \textsf{Vars}(R^\#)\} \textsf{ in} \\ & & \textsf{let } R_1^\# = \textsf{delete}(R^\#, \{x \mid x \in X \textsf{ in} \\ & & \textsf{rename}(R_2^\#, \{(x, x') \mid x \in X\}) \end{cases}$$

$$\text{extend} : \begin{cases} \mathcal{D}_n^\# & \longrightarrow & \Delta_{2n}^\# \\ R^\# & \longmapsto & \textcolor{blue}{\text{let }} R_1^\# = \mathsf{add}(R^\#, \{x' \mid x \in \mathsf{Vars}(R^\#)\}) \textcolor{blue}{\text{ in}} \end{cases}$$

$$\text{img} : \begin{cases} \Delta_{2n}^\# & \longrightarrow & \mathcal{D}_n^\# \\ R^\# & \longmapsto & \textcolor{blue}{\text{let }} X = \{x \in \mathsf{Vars}(R^\#) \mid x' \in \mathsf{Vars}(R^\#)\} \textcolor{blue}{\text{ in}} \\ & & \textcolor{blue}{\text{let }} R_1^\# = \mathsf{delete}(R^\#, \{x \mid x \in X \textcolor{blue}{\text{ in}} \\ & & \mathsf{rename}(R_2^\#, \{(x, x') \mid x \in X\}) \end{cases}$$

$$\text{apply} : \begin{cases} \mathcal{D}_n^\# \times \mathcal{I}^\# & \longrightarrow & \mathcal{D}_n^\# \\ R^\#, I^\# & \longmapsto & \textcolor{blue}{\text{let }} R_1^\# = \mathsf{extend}(R^\#) \textcolor{blue}{\text{ in}} \\ & & \textcolor{blue}{\text{let }} R_2^\# = R_1^\# \cap^\# I^\# \textcolor{blue}{\text{ in}} \\ & & \mathsf{img}(R_2^\#) \end{cases}$$

$$
\begin{aligned}
&\mathbb{S}_{\mathbb{A}}[\![{}^{l_1} X \leftarrow e^{l_2}]\!]_t(R^\#, I^\#) = \\
&\qquad \textsf{let } I_g^\# \quad = \bigcup_{t' \in \mathcal{T}\setminus\{t\}}^{\#} \{I^\#(t')\} \textsf{ in} \\
&\qquad \textsf{let } I_l^\# \quad = \textsf{extend}(R^\#) \textsf{ in} \\
&\qquad \textsf{let } I_l^\# \quad = \textsf{assign}(I_l^\#, \{(X', e)\}) \textsf{ in} \\
&\qquad \textsf{let } R_1^\# \quad = \textsf{img}(I_l^\#) \textsf{ in} \\
&\qquad \textsf{let } R_2^\# \quad = \lim \lambda Y^\#.\ Y^\# \triangledown (R_1^\# \cup^\# \textsf{apply}(Y^\#, I_g^\#)) \textsf{ in} \\
&\qquad\qquad R^\#[l_2 \mapsto R_2^\#], I^\#[t \mapsto I^\#(t) \cup^\# I_l^\#]
\end{aligned}
$$

```
var z:int, h:int, c:int, t:int, l:int, r:int;

initial z == 0 and h == 0 and c == 0 and t == 0 and l == 0;
```

```
thread t1:
begin
 while (z < 1000) do
  z = z + 1;
  if (h < 100) then
   h = h + 1;
  endif;
 done;
end
```

```
thread t2:
begin
 while (z < 1000) do
  z = z + 1;
  c = h;
 done;
end
```

```
thread t3:
begin
 while (z < 1000) do
  if ([0, 1] == 0) then
   t = 0;
  else
   t = t + c - l;
  endif;
   l = c;
 done;
end
```

$$0 \leq t \leq l \leq c \leq h \leq 100$$

,

$$z \geq 1000$$