SMT : satisfaction modulo théories

Gérard Berry

Collège de France
Chaire Algorithmes, machines et langages
gerard.berry@college-de-france.fr

Cours 4, 23/03/2016

Et séminaire de Sylvain Conchon (LRI Orsay)

COLLÈGE
DE FRANCE
1530

Objectif, questions centrales et applications

- Vérifier ou infirmer des formules mêlant logique et théories particulières
  - Surtout des théories décidables utiles en informatique
  - Quelquefois des théories indécidables, mais avec des algorithmes utilisables en pratique (ex. : quantification universelle).
- Trois grandes questions
  - Quelles théories traiter ? Avec quels algorithmes ?
  - Comment écrire des formules agréables pour l'utilisateur ?
  - Comment combiner les théories entre elles et avec la logique?
- Beaucoup d'applications
  - Vérification de circuits et programmes, génération de tests logique, arithmétique, structure de données , tableaux, pointeurs, appels de fonctions, etc.
  - Programmation par contraintes
  - Problèmes d'optimisation
  - Simulation, bioinformatique, etc.

Exemples de théories décidables et utiles

- Egalité avec fonctions non interprétées
  - un bon niveau d'abstraction dès que les détails des fonctions ne sont pas essentiels (ex. vérification de pipelines ou d'optimisations)
  - satisfiabilité : NP-complet, mais bien utilisable
- Arithmétique réelle
  - programmation linéaire : $(3x_1 + 2x_2 \leq 5) \wedge (2x_1 - 2x_2 = 0)$
    très grand nombre d'applications, grosse industrie !
    satisfiabilité : simplexe général, 2^n en théorie, mais bon en pratique
    ellipsoïdes, polynomial, pas encore meilleur en pratique
- Arithmétique entière
  - ILP : Integer Linear Programming (coefficients et solutions)
    satisfiabilité : branch-and-bound, NP-complet
    applications : scheduling, optimisation, etc.
  - inéquations aux différences : $(x - y \leq 3) \wedge (y - x) \leq 2 \wedge (x - z) \leq -6$
    satisfiabilité : polynomiale
    application majeure : automates temporisés, cf cours 5 du 30/03/2016

Exemples de théories décidables et utiles

- Bitvecteurs
  - importants pour la programmation système, l'optimisation de compilateurs et la vérification de circuits
  - se ramène à SAT (*bit blasting*), donc NP-complet

  mais expression bien plus agréable que dans SAT direct
- Manipulation de tableaux
  - mélange de non-interprété et d'arithmétique entière
  - satisfiabilité : dépend des manipulations arithmétiques des indices
- Logiques de pointeurs (non traitée dans ce cours)
  - logique de séparation : gestion du tas, programmes parallèles
- Combinaisons de théories décidables
  - en général indécidable !
  - mais des cas où ça marche....

Les principaux solveurs SMT

- Alt-Ergo : (Inria Saclay, LRI Orsay, Ocaml Pro)
  - conçu pour la preuve de programmes (Why 3)
  - cf. séminaire de Sylvain Conchon ce jour !
- CVC4 : New York University, Iowa University
  - solveur généraliste, applications variées (langages, model checking, génération de tests, etc)
- Simplify : le premier du domaine : Greg Nelson (DEC → ∗)
- Yices : SRI Menlo Park
  - solveur généraliste, en liaison avec l'assistant PVS
- Z3 : Microsoft Research Redmond
  - solveur généraliste open source
- MathSat (arithmétique, Trento), VeriT (LORIA Nancy), Boolector (Bitvecteurs, Linz), Prover SL (Stålmark), etc.

Un format standard (SMT-LIB), une compétition (SMT-COMP)

Agenda

1. Principe général de SMT
2. Egalité avec fonctions non interprétées
3. Arithmétiques entières et réelles
4. Bitvecteurs
5. Tableaux
6. Combinaisons de théories
7. Conclusion

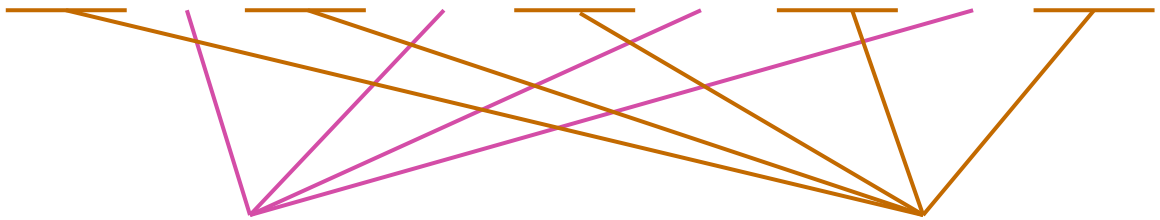
Agenda

1. Principe général de SMT
2. Egalité avec fonctions non interprétées
3. Arithmétiques entières et réelles
4. Bitvecteurs
5. Tableaux
6. Allocation mémoire et pointeurs
7. Combinaisons de théories
8. Conclusion

Principe général de SMT

- Travailler avec des formules mélangeant logique et théories

$$((a = 1) \vee (a = 2)) \wedge (a \geq 3) \wedge ((b \leq 2) \vee (b \geq 3))$$



logique arithmétique

$$(f(a) = 1) \vee (a - 3 = 2)) \wedge (g(a) \geq 3) \wedge ((B[0] \leq 2) \vee (B[1] \geq 3))$$

logique + arithmétique + fonctions + tableaux

- Satisfiabilité : il existe un modèle, i.e. une valeur des inconnues dans les théories qui rend la formule vraie
- Validité : la formule est vraie pour tout modèle
  $\Leftrightarrow$ sa négation n'est pas satisfiable

Méthode 1 : DNF + traitement par cas

- Mise de la formule logique en DNF (disjunctive normal form)
- Résolution des cubes par des solveurs dédiés

$$((a = 1) \vee (a = 2)) \wedge (a \geq 3) \wedge ((b \leq 2) \vee (b \geq 3))$$

DNF

1. $((a = 1) \wedge (a \geq 3) \wedge (b \leq 2))$
2. $\vee ((a = 2) \wedge (a \geq 3) \wedge (b \leq 2)$
3. $\vee ((a = 1) \wedge (a \geq 3) \wedge (b \geq 3))$
4. $\vee ((a = 2) \wedge (a \geq 3) \wedge (b \geq 3))$

4 appels du solveur arithmétique Arith :
clauses 1. et ~~3.~~ insatisfiables à cause de $(a = 1) \wedge (a \geq 3)$
clauses 2. et ~~4.~~ insatisfiables à cause de $(a = 2) \wedge (a \geq 3)$

Efficacité ⇒ apprentissage et mémorisation dans le solveur !

Méthode 2 (la bonne) : interagir avec SAT

$$((a = 1) \vee (a = 2)) \wedge (a \geq 3) \wedge ((b \leq 2) \vee (b \geq 3))$$

$$(x_1 \vee x_2) \wedge x_3 \wedge (x_4 \vee x_5)$$

$$x_1 : (a = 1),\ x_2 : (a = 2),\ x_3 : (a \geq 3),\ x_4 : (b \leq 2),\ x_5 : (b \geq 3)$$

1. SAT → x_1, x_3, x_4 (convention électrique : vrai, faux)
   → $(a = 1) \wedge (a \geq 3) \wedge (b \leq 2)$ → Arith → *Unsat*

2. SAT → x_2, x_3, x_4
   → $(a = 2) \wedge (a \geq 3) \wedge (b \leq 2)$ → Arith → *Unsat*

~~3. SAT → x_1, x_3, x_5~~
~~→ $(a = 1) \wedge (a \geq 3) \wedge (b \geq 3)$ → Arith → *Unsat*~~

Coopération bidirectionnelle SAT ⟷ Arith

$$((a = 1) \vee (a = 2)) \wedge (a \geq 3) \wedge ((b \leq 2) \vee (b \geq 3))$$

$$(x_1 \vee x_2) \wedge x_3 \wedge (x_4 \vee x_5)$$

$$x_1 : (a = 1),\ x_2 : (a = 2),\ x_3 : (a \geq 3),\ x_4 : (b \leq 2),\ x_5 : (b \geq 3)$$

1. SAT → x_1, x_3, x_4

   → $x_1 : (a = 1) \wedge x_3 : (a \geq 3) \wedge x_4 : (b \leq 2)$ → Arith → $(\overline{x}_1 \vee \overline{x}_3)$

analyse du conflit par Arith pour retour à SAT

Coopération bidirectionnelle SAT ⟷ Arith

$$((a = 1) \vee (a = 2)) \wedge (a \geq 3) \wedge ((b \leq 2) \vee (b \geq 3))$$

$$(x_1 \vee x_2) \wedge x_3 \wedge (x_4 \vee x_5) \wedge (\overline{x}_1 \vee \overline{x}_3)$$

$$x_1 : (a = 1),\ x_2 : (a = 2),\ x_3 : (a \geq 3),\ x_4 : (b \leq 2),\ x_5 : (b \geq 3)$$

1. SAT → x_1, x_3, x_4

   → $x_1 : (a = 1) \wedge x_3 : (a \geq 3) \wedge x_4 : (b \leq 2)$ → Arith → $(\overline{x}_1 \vee \overline{x}_3)$

2. SAT → x_2, x_3, x_4

   → $x_2 : (a = 2) \wedge x_3 : (a \geq 3) \wedge x_4 : (b \leq 2)$ → Arith → $(\overline{x}_2 \vee \overline{x}_3)$

Coopération bidirectionnelle SAT ⟷ Arith

$$((a=1)\vee(a=2))\wedge(a\geq 3)\wedge((b\leq 2)\vee(b\geq 3))$$

$$(x_1\vee x_2)\wedge x_3\wedge(x_4\vee x_5)\wedge(\overline{x}_1\vee\overline{x}_3)\wedge(\overline{x}_2\vee\overline{x}_3)$$

$$x_1:(a=1),\ x_2:(a=2),\ x_3:(a\geq 3),\ x_4:(b\leq 2),\ x_5:(b\geq 3)$$

1. SAT → x_1, x_3, x_4
   → $x_1:(a=1)\wedge x_3:(a\geq 3)\wedge x_4:(b\leq 2)$ → Arith → $(\overline{x}_1\vee\overline{x}_3)$

2. SAT → x_2, x_3, x_4
   → $x_2:(a=2)\wedge x_3:(a\geq 3)\wedge x_4:(b\leq 2)$ → Arith → $(\overline{x}_2\vee\overline{x}_3)$

3. SAT → *Unsat*

Agenda

1. Principe général de SMT
2. Egalité avec fonctions non interprétées
3. Arithmétiques entières et réelles
4. Bitvecteurs
5. Tableaux
6. Combinaisons de théories
7. Conclusion

Fonctions non-interprétées

- Calcul minimal des fonctions
  - vérification de propriétés abstraites d'expressions
  - optimisation de pipelines (HW) et de code généré (SW)
  - records, ensembles, etc.

Exemple : pour x, y, z entiers et f fonction entière,
la formule suivante peut-elle être vraie ?

$$(x = y) \wedge (x \times (f(y) + f(x)) = t) \wedge (y \times (f(x) + f(x)) \neq t)$$

Non, car l'égalité extensionnelle s'écrit

$$x = y \Rightarrow f(x) = f(y)$$

Donc $(x = y) \wedge (x \times (f(y) + f(x)) = t) \Rightarrow (y \times (f(x) + f(x)) = t)$
et la formule initiale est fausse

De plus, les entiers, + et × n'ont rien à voir là dedans :

$$(x = y) \wedge (h(x,(g(f(y),f(z)))) = t) \Rightarrow h(y,(g(f(x),f(z)))) = t$$

Egalité avec fonctions non interprétées

- Formules booléennes dont les atomes sont des expressions comportant des égalités sur des variables, des constantes et des appels de fonctions, mais sans sémantique spécifiée

  formule $F = (x = y) \wedge (y \neq z) \wedge ((f(x) = a) \vee (g(y,z) \neq f(b))$

  axiome (arité 2) : $(x = x') \wedge (y = y') \Rightarrow f(x,y) = f(x',y')$

1. F est satisfiable s'il existe un modèle des variables, constantes et fonctions qui la rend vraie

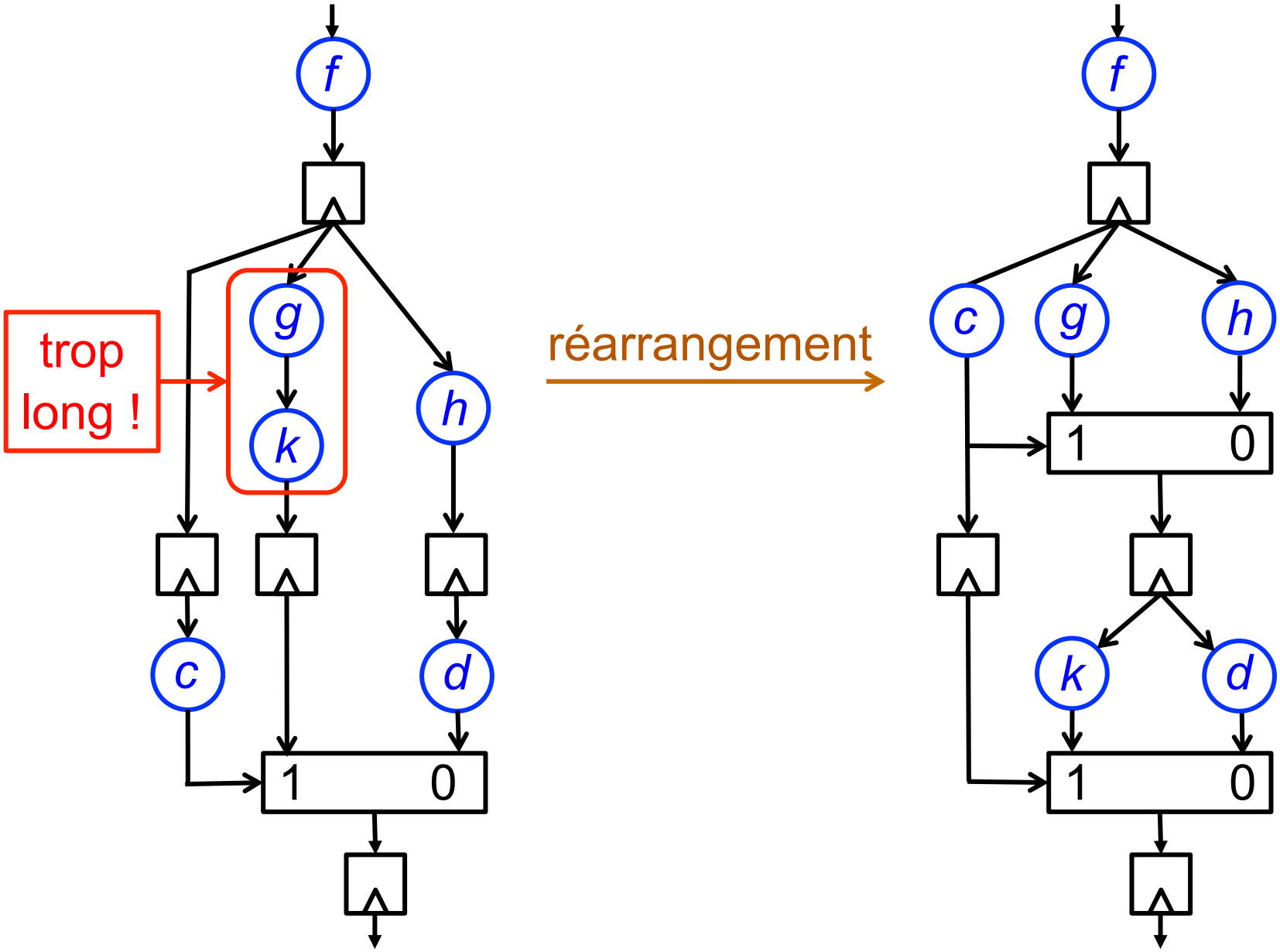
   satisfiable : $(x = y) \wedge (y \neq z) \wedge (f(x,y) \neq f(x,z))$

   non-satisfiable : ~~$(x = y) \wedge (y = z) \wedge (f(x,y) \neq f(x,z))$~~

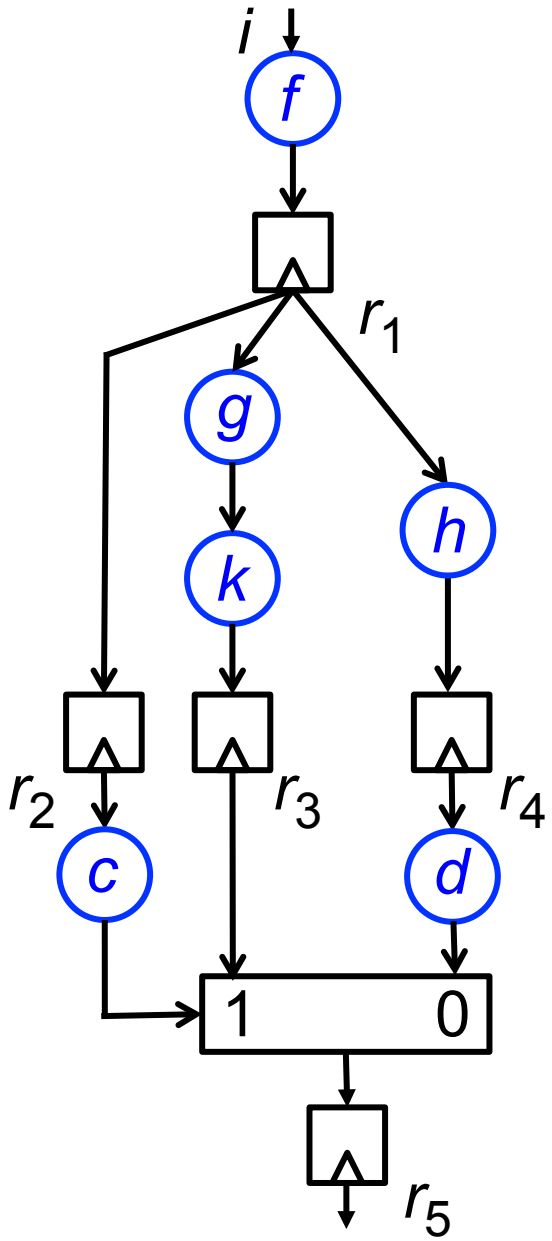
2. F est valide si elle est vraie pour tout modèle

   valide : $(x = x') \wedge (y = y') \Rightarrow f(x,y) = f(x',y')$

Exemple : optimisation de pipeline HW

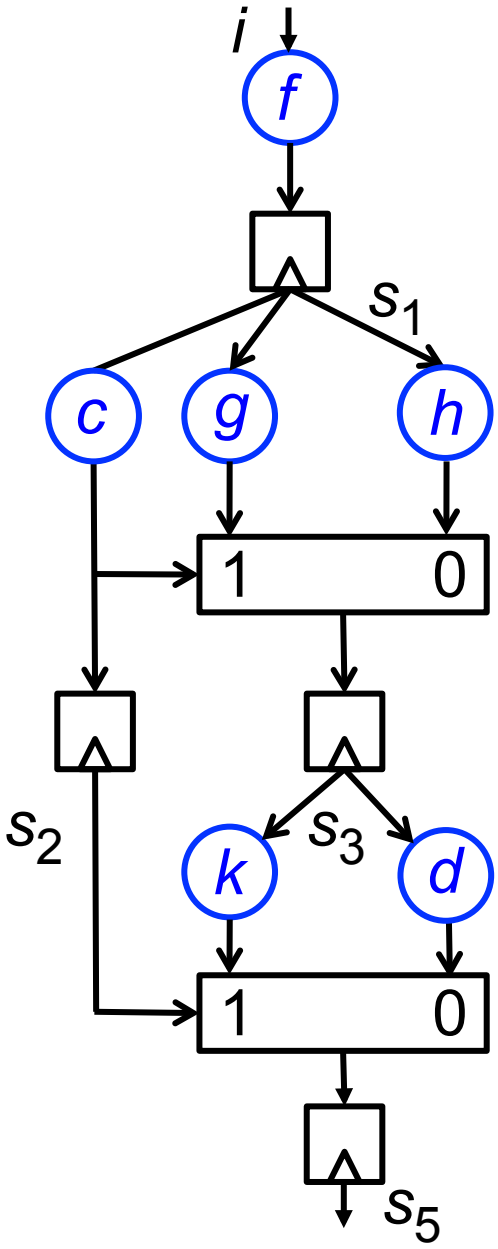


Equations du circuit original



$$
\begin{aligned}
& r_1 = f(i) \\
\wedge\ & r_2 = r_1 \\
\wedge\ & r_3 = k(g(r_1)) \\
\wedge\ & r_4 = h(r_1) \\
\wedge\ & r_5 = mux(c(r_2), r_3, d(r_4))
\end{aligned}
$$

Equations du circuit optimisé



$$
\begin{aligned}
& s_1 = f(i) \\
\wedge\ & s_2 = c(s_1) \\
\wedge\ & s_3 = mux(c(s_1), g(s_1), h(s_1)) \\
\wedge\ & s_5 = mux(s_2, k(s_3), d(s_3))
\end{aligned}
$$

Formule à infirmer pour valider le retiming

$$
\begin{aligned}
(\quad & (r_1 = f(i)) \wedge (r_2 = r_1) \wedge (r_3 = k(g(r_1))) \\
& \wedge (r_4 = h(r_1)) \wedge (r_5 = mux(c(r_2), r_3, d(r_4))) \\
& \wedge (s_1 = f(i)) \wedge (s_2 = c(s_1)) \\
& \wedge (s_3 = mux(c(s_1), g(s_1), h(s_1))) \\
& \wedge (s_5 = mux(s_2, k(s_3), d(s_3))) \\
& \wedge \boxed{(r_5 \neq s_5)}
\end{aligned}
$$

Mais cette formule est satisfiable !
Il faut rajouter des éléments de comportement pour *c* et *mux*

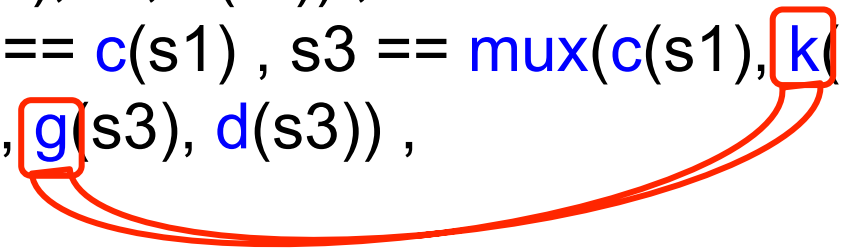
Vérification en Z3 (MSR, version Python)

```
solve ( r1 == f(i) , r2 == r1 , r3 == k(g(r1)) , r4 == h(r1) ,
        r5 == mux(c(r2), r3, d(r4)) ,
        s1 == f(i) , s2 == c(s1) , s3 == mux(c(s1), g(s1), h(s1)) ,
        s5 == mux(s2, k(s3), d(s3)) ,
        s5 != r5 ,
        T != F ,
        (c(r2) == T or c(r2) == F) , (c(s1) == T or c(s1) == F)
        mux(T, r3, d(r4)) == r3 ,
        mux(F, r3, d(r4)) == d(r4) ,
        mux(T, g(s1), h(s1)) == g(s1) ,
        mux(F, g(s1), h(s1)) == h(s1),
        mux(T, k(s3), d(s3)) == k(s3) ,
        mux(F, k(s3), d(s3)) == d(s3) )
```

no solution

Que se passe-t-il en cas d'erreur ?

solve (r1 == f(i) , r2 == r1 , r3 == k(g(r1)) , r4 == h(r1) ,
r5 == mux(c(r2), r3, d(r4)) ,
s1 == f(i) , s2 == c(s1) , s3 == mux(c(s1), k(s1), h(s1)) ,
s5 == mux(s2, g(s3), d(s3)) ,
s5 != r5 ,



T != F ,

(c(r2) == T or c(r2) == F) , (c(s1) == T or c(s1) == F)

mux(T, r3, d(r4)) == r3 ,
mux(F, r3, d(r4)) == d(r4) ,
mux(T, g(s1), h(s1)) == g(s1) ,
mux(F, g(s1), h(s1)) == h(s1),
mux(T, k(s3), d(s3)) == k(s3) ,
mux(F, k(s3), d(s3)) == d(s3))

Le solveur produit un contre-exemple

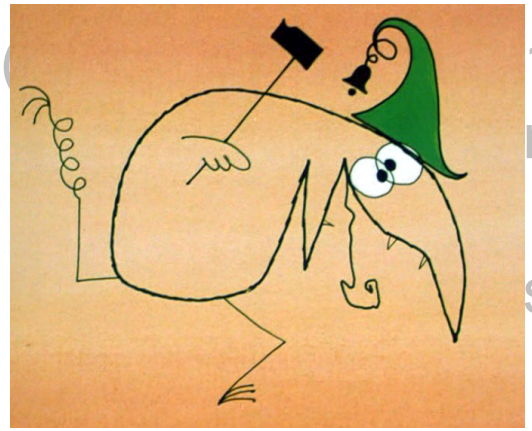
i = 2,
T = 1,
F = 4,
s5 = 10,
s3 = 7,
s2 = 1,
s1 = 3,
r5 = 12,
r4 = 6,
r3 = 12,
r2 = 3,
r1 = 3,
d = [7 -> 9, 6 -> 13, else -> 9],
k = [3 -> 5, 11 -> 12, 7 -> 14, else -> 5],

mux = [(1, 5, 6) -> 7,
(1, 8, 9) -> 10,
(1, 12, 13) -> 12,
(4, 12, 13) -> 13,
(1, 11, 6) -> 11,
(4, 11, 6) -> 6,
(1, 14, 9) -> 14,
(4, 14, 9) -> 9,
else -> 7],
f = [2 -> 3, else -> 3],
h = [3 -> 6, else -> 6],
c = [3 -> True, else -> True],
g = [7 -> 8, 3 -> 11, else -> 8]

Si vous en voulez un autre, ajoutez (automatiquement) la contrainte « et pas ces valeurs là » !

Pourquoi faire simple si on peut faire compliqué ?

solve (



1 , r3 == k(g(r1)) , r4 == h(r1) ,
r3, d(r4)) ,
c(s1) , s3 == mux(c(s1), g(s1), h(s1)) ,
s3), d(s3)) ,

T != F ,

mais pourquoi pas typer c ?

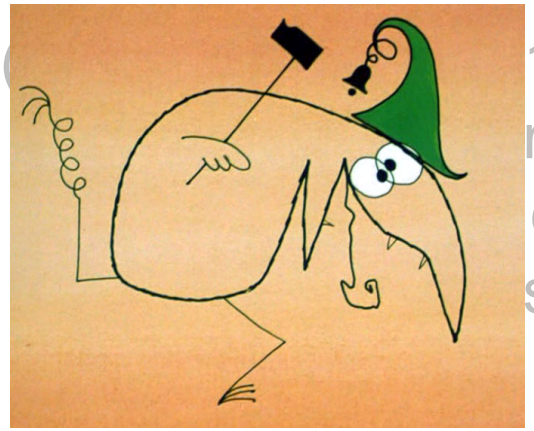
(c(r2) == T or c(r2) == F) , (c(s1) == T or c(s1) == F)

mux(T, r3, d(r4)) == r3 ,
mux(F, r3, d(r4)) == d(r4) ,
mux(T, g(s1), h(s1)) == g(s1) ,
mux(F, g(s1), h(s1)) == h(s1),
mux(T, k(s3), d(s3)) == k(s3) ,
mux(F, k(s3), d(s3)) == d(s3))

possible dans
tous les solveurs

Pourquoi faire simple si on peut faire compliqué ?

solve (



1 , r3 == k(g(r1)) , r4 == h(r1) ,
r3, d(r4)) ,
c(s1) , s3 == mux(c(s1), g(s1), h(s1)) ,
s3), d(s3)) ,

T != F ,

forall y, z. mux(T, y, z)) == y && mux(F, y, z) == z

F)

~~mux(T, r3, d(r4)) == r3 ,~~
~~mux(F, r3, d(r4)) == d(r4) ,~~
~~mux(T, g(s1), h(s1)) == g(s1) ,~~
~~mux(F, g(s1), h(s1)) == h(s1),~~
~~mux(T, k(s3), d(s3)) == k(s3) ,~~
~~mux(F, k(s3), d(s3)) == d(s3))~~

Et pourquoi pas
quantifier universellement
les arguments 2,3 de mux ?

parce que la logique
devient indécidable !

Possible avec des heuristiques délicates, cf. séminaire

Se débarrasser des constantes et fonctions

$$F = (x = y) \wedge (y \neq z) \wedge ((f(x) = a) \vee (g(y,z) \neq f(b)))$$

1. Introduire une variable par constante, et des clauses exprimant qu'elles sont toutes différentes :

$$F_1 = (x = y) \wedge (y \neq z) \wedge ((f(x) = a) \vee (g(y,z) \neq f(b)) \wedge (a \neq b)$$

Se débarrasser des constantes et fonctions

$$F = (x = y) \wedge (y \neq z) \wedge ((f(x) = a) \vee (g(y,z) \neq f(b)))$$

1. Introduire une variable par constante, et des clauses exprimant qu'elles sont toutes différentes :

$$F_1 = (x = y) \wedge (y \neq z) \wedge ((f(x) = a) \vee (g(y,z) \neq f(b)) \wedge (a \neq b)$$

2. Introduire une variable par appel de fonction, et des clauses exprimant que deux appels de la même fonction sont égaux si les arguments le sont

$$F_2 = (x = y) \wedge (y \neq z) \wedge (f_1 = a) \vee (g(y,z) \neq f_2) \wedge (a \neq b)$$
$$\wedge ((x = b) \Rightarrow (f_1 = f_2))$$

Se débarrasser des constantes et fonctions

$$F = (x = y) \wedge (y \neq z) \wedge ((f(x) = a) \vee (g(y,z) \neq f(b))$$

1. Introduire une variable par constante, et des clauses exprimant qu'elles sont toutes différentes :

$$F_1 = (x = y) \wedge (y \neq z) \wedge ((f(x) = a) \vee (g(y,z) \neq f(b)) \wedge (a \neq b)$$

2. Introduire une variable par appel de fonction, et des clauses exprimant que deux appels de la même fonction sont égaux si les arguments le sont

$$F_2 = (x = y) \wedge (y \neq z) \wedge (f_1 = a) \vee (g(y,z) \neq f_2) \wedge (a \neq b)$$
$$\wedge ((x = b) \Rightarrow (f_1 = f_2)$$

$$F_3 = (x = y) \wedge (y \neq z) \wedge (f_1 = a) \vee (g_1 \neq f_2) \wedge (a \neq b)$$
$$\wedge ((x = b) \Rightarrow (f_1 = f_2))$$

Exemple papier-crayon

Calculer $g(g(x))$ itérativement par $y = g(x); z = g(y)$.

A montrer automatiquement :

$$y = \underset{1}{g}(x) \wedge z = \underset{2}{g}(y) \wedge t = \underset{3}{g}(\underset{4}{g}(x)) \Rightarrow z = t$$

Transformation de la prémisse

$$y = g_1 \wedge z = g_2 \wedge t = g_3$$
$$\wedge (x = y \Rightarrow g_1 = g_2)$$
$$\wedge (x = g_4 \Rightarrow g_1 = g_3)$$
$$\wedge (x = x \Rightarrow g_1 = g_4)$$
$$\wedge (y = g_4 \Rightarrow g_2 = g_3)$$
$$\wedge (y = x \Rightarrow g_2 = g_4)$$

Exemple papier-crayon

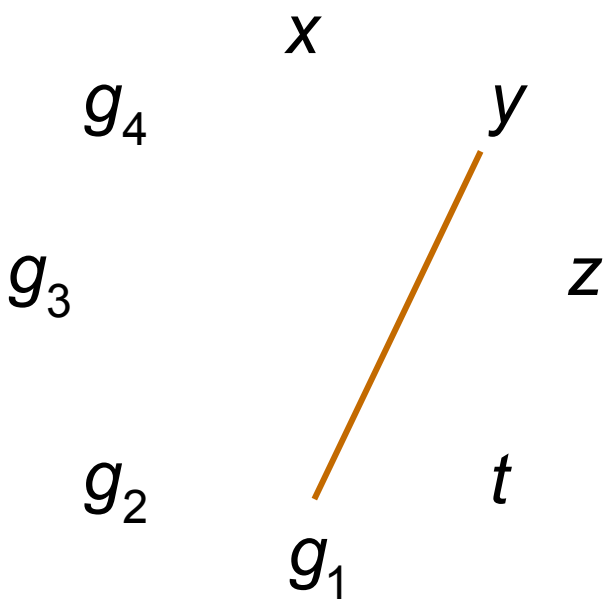
Calculer $g(g(x))$ itérativement par $y = g(x); z = g(y)$.

A montrer automatiquement :

$$y = \underset{1}{g}(x) \wedge z = \underset{2}{g}(y) \wedge t = \underset{3}{g}(\underset{4}{g}(x)) \Rightarrow z = t$$

Transformation de la prémisse

$$y = g_1 \wedge z = g_2 \wedge t = g_3$$
$$\wedge (x = y \Rightarrow g_1 = g_2)$$
$$\wedge (x = g_4 \Rightarrow g_1 = g_3)$$
$$\wedge (x = x \Rightarrow g_1 = g_4)$$
$$\wedge (y = g_4 \Rightarrow g_2 = g_3)$$
$$\wedge (y = x \Rightarrow g_2 = g_4)$$



Exemple papier-crayon

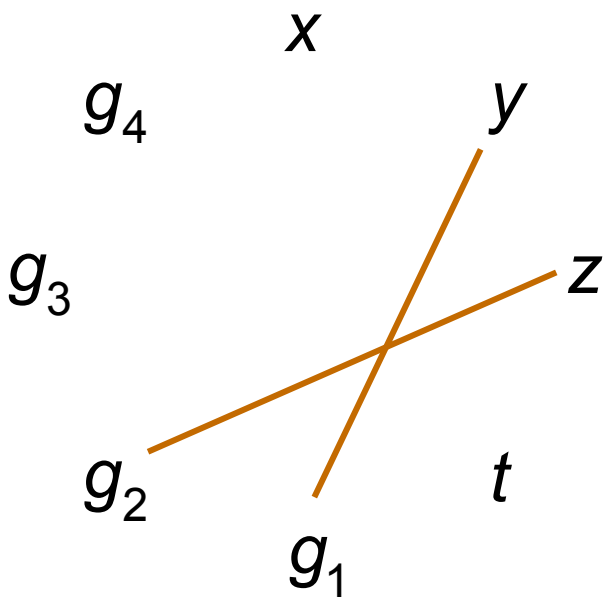
Calculer $g(g(x))$ itérativement par $y = g(x); z = g(y)$.

A montrer automatiquement :

$$y = \underset{1}{g}(x) \wedge z = \underset{2}{g}(y) \wedge t = \underset{3}{g}(\underset{4}{g}(x)) \Rightarrow z = t$$

Transformation de la prémisse

$$y = g_1 \wedge \boxed{z = g_2} \wedge t = g_3$$
$$\wedge (x = y \Rightarrow g_1 = g_2)$$
$$\wedge (x = g_4 \Rightarrow g_1 = g_3)$$
$$\wedge (x = x \Rightarrow g_1 = g_4)$$
$$\wedge (y = g_4 \Rightarrow g_2 = g_3)$$
$$\wedge (y = x \Rightarrow g_2 = g_4)$$



Exemple papier-crayon

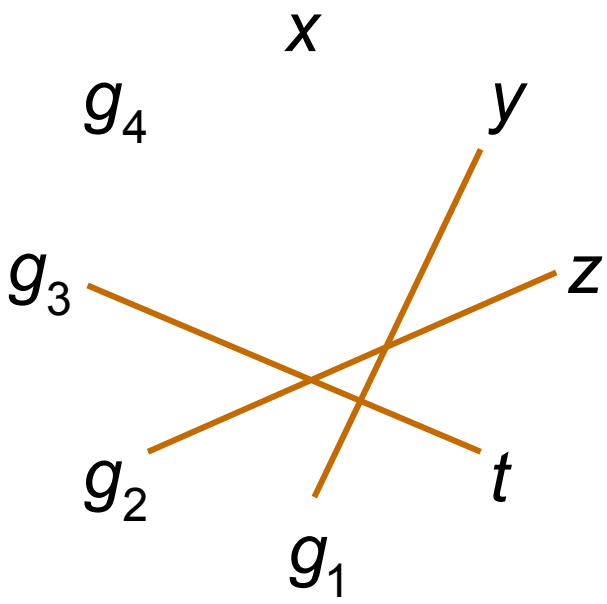
Calculer $g(g(x))$ itérativement par $y = g(x); z = g(y)$.

A montrer automatiquement :

$$y = \underset{1}{g}(x) \wedge z = \underset{2}{g}(y) \wedge t = \underset{3}{g}(\underset{4}{g}(x)) \Rightarrow z = t$$

Transformation de la prémisse

$$y = g_1 \wedge z = g_2 \wedge t = g_3$$
$$\wedge (x = y \Rightarrow g_1 = g_2)$$
$$\wedge (x = g_4 \Rightarrow g_1 = g_3)$$
$$\wedge (x = x \Rightarrow g_1 = g_4)$$
$$\wedge (y = g_4 \Rightarrow g_2 = g_3)$$
$$\wedge (y = x \Rightarrow g_2 = g_4)$$



Exemple papier-crayon

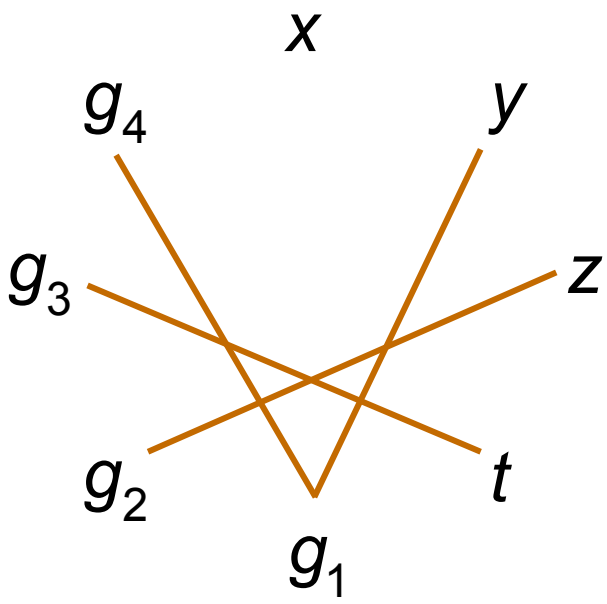
Calculer $g(g(x))$ itérativement par $y = g(x); z = g(y)$.

A montrer automatiquement :

$$y = \underset{1}{g}(x) \wedge z = \underset{2}{g}(y) \wedge t = \underset{3}{g}(\underset{4}{g}(x)) \Rightarrow z = t$$

Transformation de la prémisse

$$y = g_1 \wedge z = g_2 \wedge t = g_3$$
$$\wedge (x = y \Rightarrow g_1 = g_2)$$
$$\wedge (x = g_4 \Rightarrow g_1 = g_3)$$
$$\wedge (\boxed{x = x} \Rightarrow \boxed{g_1 = g_4})$$
$$\wedge (y = g_4 \Rightarrow g_2 = g_3)$$
$$\wedge (y = x \Rightarrow g_2 = g_4)$$



Exemple papier-crayon

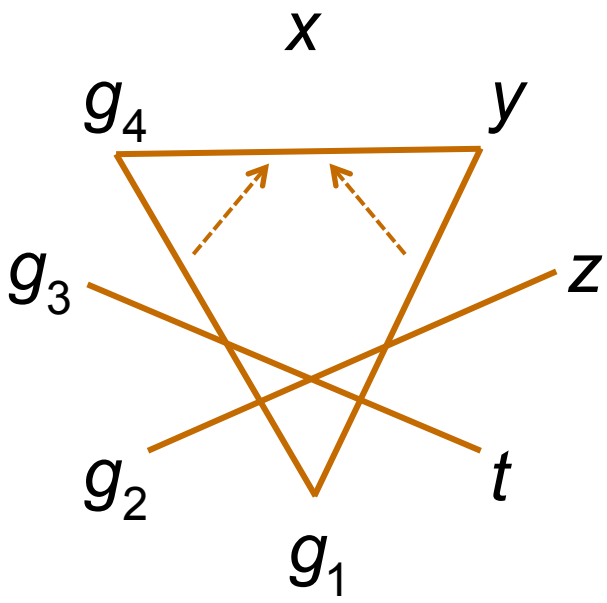
Calculer $g(g(x))$ itérativement par $y = g(x); z = g(y)$.

A montrer automatiquement :

$$y = \underset{1}{g}(x) \wedge z = \underset{2}{g}(y) \wedge t = \underset{3}{g}(\underset{4}{g}(x)) \Rightarrow z = t$$

Transformation de la prémisse

$$y = g_1 \wedge z = g_2 \wedge t = g_3$$
$$\wedge (x = y \Rightarrow g_1 = g_2)$$
$$\wedge (x = g_4 \Rightarrow g_1 = g_3)$$
$$\wedge (x = x \Rightarrow g_1 = g_4)$$
$$\wedge (y = g_4 \Rightarrow g_2 = g_3)$$
$$\wedge (y = x \Rightarrow g_2 = g_4)$$



Exemple papier-crayon

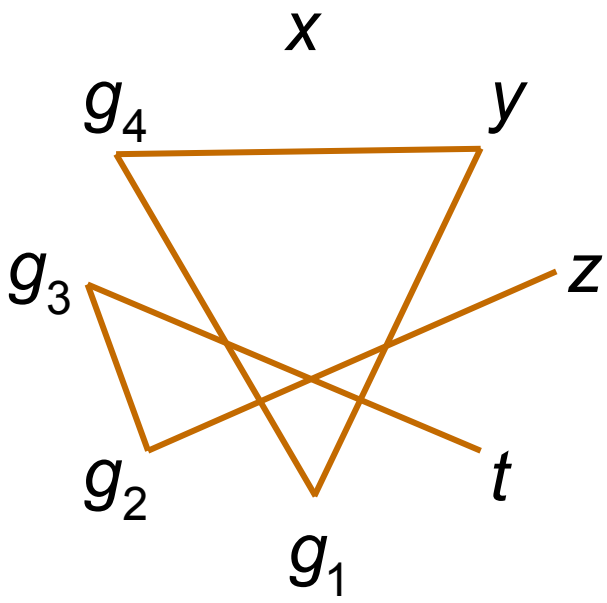
Calculer $g(g(x))$ itérativement par $y = g(x); z = g(y)$.

A montrer automatiquement :

$$y = \underset{1}{g}(x) \wedge z = \underset{2}{g}(y) \wedge t = \underset{3}{g}(\underset{4}{g}(x)) \Rightarrow z = t$$

Transformation de la prémisse

$$y = g_1 \wedge z = g_2 \wedge t = g_3$$
$$\wedge (x = y \Rightarrow g_1 = g_2)$$
$$\wedge (x = g_4 \Rightarrow g_1 = g_3)$$
$$\wedge (x = x \Rightarrow g_1 = g_4)$$
$$\wedge (y = g_4 \Rightarrow g_2 = g_3)$$
$$\wedge (y = x \Rightarrow g_2 = g_4)$$



Exemple papier-crayon

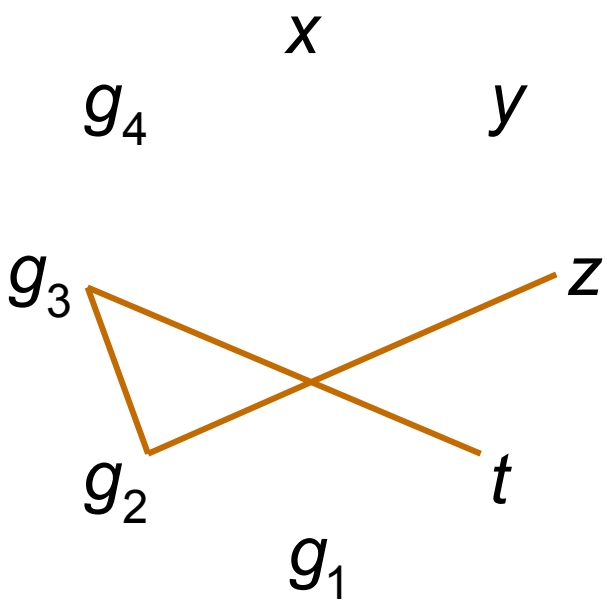
Calculer $g(g(x))$ itérativement par $y = g(x); z = g(y)$.

A montrer automatiquement :

$$y = \underset{1}{g}(x) \wedge z = \underset{2}{g}(y) \wedge t = \underset{3}{g}(\underset{4}{g}(x)) \Rightarrow z = t$$

Transformation de la prémisse

$$y = g_1 \wedge z = g_2 \wedge t = g_3$$
$$\wedge(x = y \Rightarrow g_1 = g_2)$$
$$\wedge(x = g_4 \Rightarrow g_1 = g_3)$$
$$\wedge(x = x \Rightarrow g_1 = g_4)$$
$$\wedge(y = g_4 \Rightarrow g_2 = g_3)$$
$$\wedge(y = x \Rightarrow g_2 = g_4)$$



Exemple papier-crayon

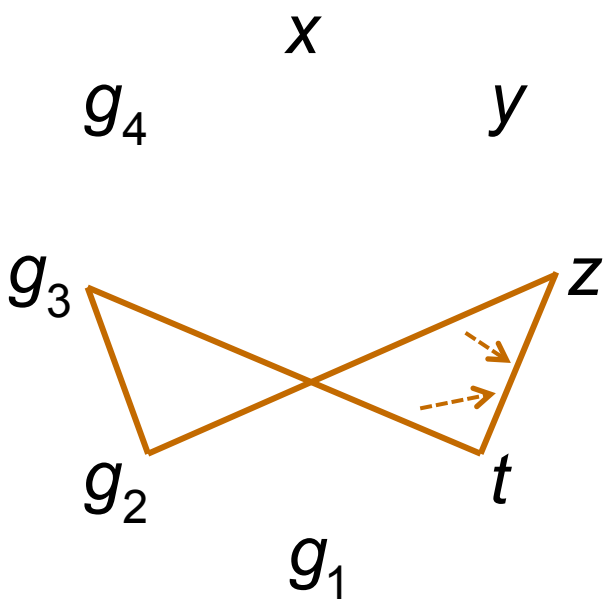
Calculer $g(g(x))$ itérativement par $y = g(x); z = g(y)$.

A montrer automatiquement :

$$y = \underset{1}{g}(x) \wedge z = \underset{2}{g}(y) \wedge t = \underset{3}{g}(\underset{4}{g}(x)) \Rightarrow \boxed{z = t}$$

Transformation de la prémisse

$$y = g_1 \wedge z = g_2 \wedge t = g_3$$
$$\wedge (x = y \Rightarrow g_1 = g_2)$$
$$\wedge (x = g_4 \Rightarrow g_1 = g_3)$$
$$\wedge (x = x \Rightarrow g_1 = g_4)$$
$$\wedge (y = g_4 \Rightarrow g_2 = g_3)$$
$$\wedge (y = x \Rightarrow g_2 = g_4)$$



Fatigant pour nous, mais pas pour un ordinateur !

Différence au lieu d'égalité

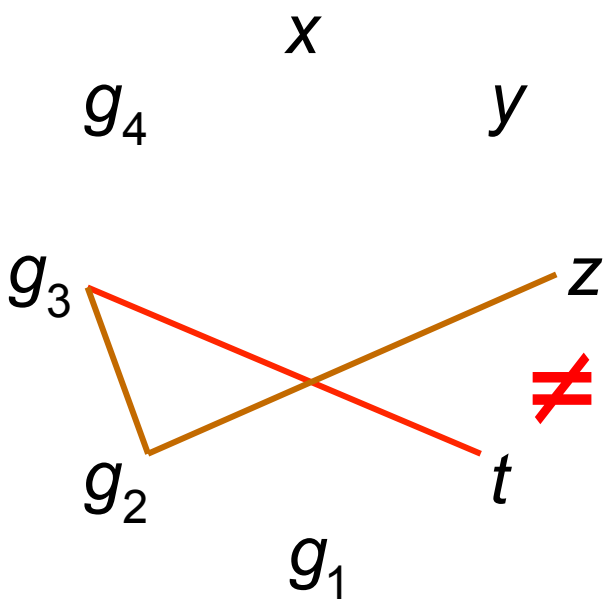
Calculer $g(g(x))$ itérativement par $y = g(x); z = g(y)$.

A montrer automatiquement :

$$y = \underset{1}{g}(x) \wedge z = \underset{2}{g}(y) \wedge t \neq \underset{3}{g}(\underset{4}{g}(x)) \Rightarrow \boxed{z \neq t}$$

Transformation de la prémisse

$$y = g_1 \wedge z = g_2 \wedge t \neq g_3$$
$$\wedge (x = y \Rightarrow g_1 = g_2)$$
$$\wedge (x = g_4 \Rightarrow g_1 = g_3)$$
$$\wedge (x = x \Rightarrow g_1 = g_4)$$
$$\wedge (y = g_4 \Rightarrow g_2 = g_3)$$
$$\wedge (y = x \Rightarrow g_2 = g_4)$$



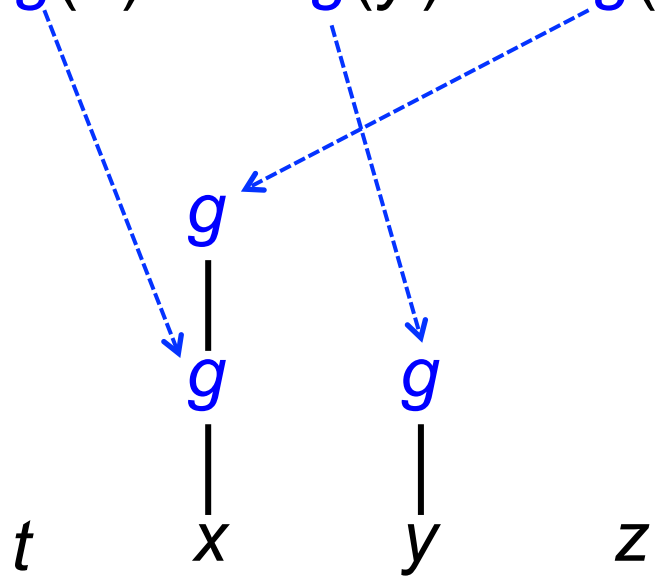
Chemin avec une seule différence !

Bien plus efficace avec partage !

Calculer $g(g(x))$ itérativement par $y = g(x), z = g(y)$.

A montrer automatiquement :

$$y = g(x) \wedge z = g(y) \wedge t = g(g(x)) \Rightarrow z = t$$

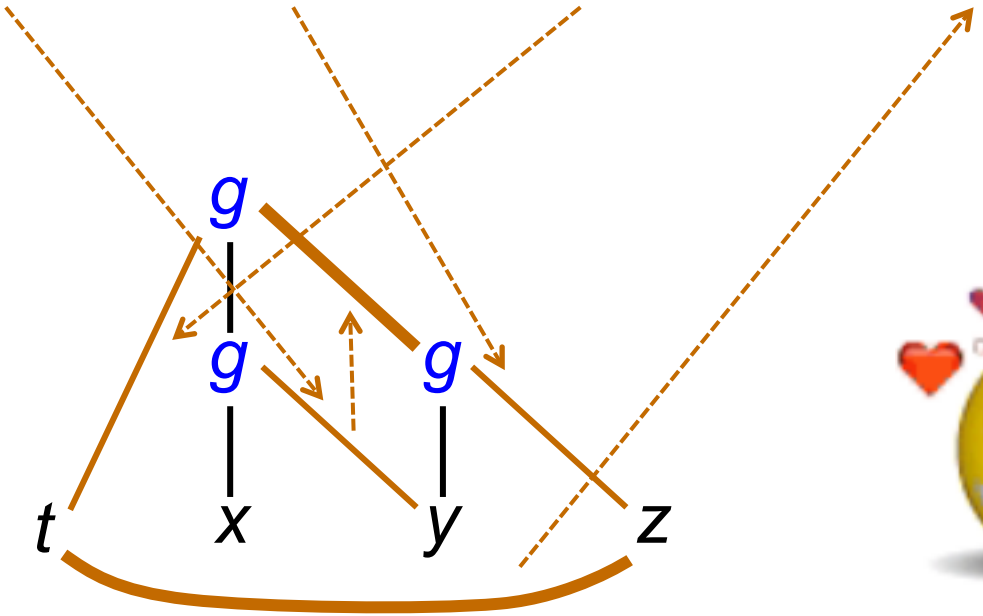


Bien plus efficace avec partage !

Calculer $g(g(x))$ itérativement par $y = g(x), z = g(y)$.

A montrer automatiquement :

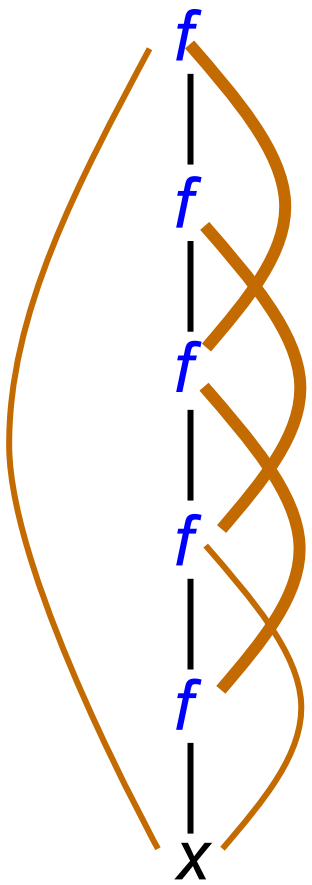
$$y = g(x) \wedge z = g(y) \wedge t = g(g(x)) \Rightarrow z = t$$



Autre exemple avec partage

Maths: $\forall f.\ f^2(x) = x \wedge f^5(x) = x \Rightarrow f(x) = x$

Non-interprété : $(f_1(f_2(x)) = x) \wedge (f_3(f_4(f_5(f_6(f_7(x))))) = x) \Rightarrow f(x) = x$

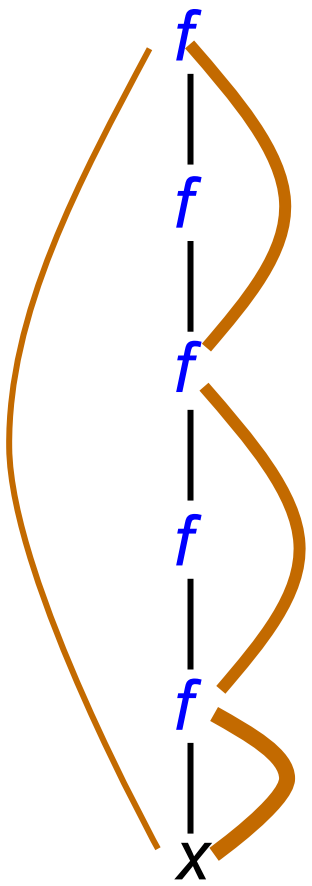


(7×6) / 2 = 21
équations induites !

Autre exemple avec partage

Maths: $\forall f.\ f^2(x) = x \wedge f^5(x) = x \Rightarrow f(x) = x$

Non-interprété : $(f_1(f_2(x)) = x) \wedge (f_3(f_4(f_5(f_6(f_7(x))))) = x) \Rightarrow f(x) = x$



(7×6) / 2 = 21
équations induites !

Théorie AC = associatif commutatif

- Il est fréquent d'utiliser des opérations associatives (A) ou associatives-commutatives (AC):
  - arithmétique : +, × AC
  - mots : concaténation A
  - multi-ensembles : union AC
  - ensembles : union AC + idempotence (A U A = A)
  - etc.
- Comme extension directe du non-interprété, les solveurs utilisent des algorithmes de filtrage (unification) ou AC dans ces cas, :

$$f \text{ A}: \quad (x = y) \Rightarrow f(f(x,y),z) = f(x,f(y,z))$$

$$f \text{ AC}: (x = y) \Rightarrow f(f(x,y),z) = f(y,f(z,x))$$

Agenda

1. Principe général de SMT
2. Egalité avec fonctions non interprétées
3. Arithmétiques entières et réelles
4. Bitvecteurs
5. Tableaux
6. Conclusion

Arithmétiques entières ou réelles

- L'arithmétique entière est indécidable (Gödel) → fragments
- Comparaisons : $x \lhd\rhd y$, avec $\lhd\rhd \in \{ <, \leq, \geq, >, =, \neq \}$
  - Arithmétique linéaire : $2x-3y+4z \leq 5$
  - Logique de la différence : $x-y \lhd\rhd k$, cf cours du 30/03
  - UTVPI ?? : $\pm x \pm y \lhd\rhd k$
  - Arithmétique polynomiale : $2xy-3yz^2+4z^3 \leq 5$

LP : Linear Programming sur les réels

- Systèmes d'inéquations linéaires réelles

$$\begin{aligned} 3x_1 + 2x_2 &\le 5x_3 + 2 \\ x_1 - 4x_2 &= 0 \end{aligned} \quad \Longrightarrow \quad \begin{aligned} 3x_1 + 2x_2 - 5x_3 - s_1 &= 0 \\ x_1 - 4x_2 &= 0 \\ s_1 &\le 2 \end{aligned}$$

- En général, présenté comme problème d'optimisation par rapport à un critère de coût, mais se généralis en problème de satisfiabilité

- De nombreux algorithmes et logiciels (libres ou industriels)
  - Fourier-Motzkin
  - Simplexe : pire cas exponentiel, mais efficace en pratique
    méthode du pivot, calculs matriciels
  - Ellipsoïdes (Khachiyan, 1979) : polynomial, grosse surprise !

ILP : Integer Linear Programming

- Résoudre des inéquations linéaires à coefficients entiers et solution entières
  - NP-complet, et plutôt cher en pratique
  - très nombreuses applications : algorithmes combinatoires variés, scheduling, optimisation de code, etc.
- Plusieurs méthodes :
  - Branch-and-Bound, Omega test, recuit simulé, etc.
- Principe du Branch-and-Bound
  - résoudre le problème sur les réels, par exemple par le simplexe
  - si la solution est entière, succès
  - sinon, au moins une variable est fractionnaire, par exemple $x = 0.75$
  - alors, contruire deux sous-problèmes avec $x \leq 0$ et $x \geq 1$
  - itérer (astucieusement) jusqu'à trouver une solution entière
  - s'il en existe une, elle sera trouvée, sinon *Unsat*

Inéquations aux différences (Difference Logic)

Formules booléennes dont les atomes sont de la forme

$x_i - x_j \leq c$

où c est une constante rationnelle

Notes :

$x - y \geq c \Leftrightarrow y - x \leq c$

$x - y = c \Leftrightarrow x - y \leq c \wedge y - x \leq c$

$x - y < c \Leftrightarrow x - y \leq c \wedge \neg(x - y = c)$

$x \leq c \Leftrightarrow x - x_0 \leq c$ où x_0 est une variable spéciale valant 0

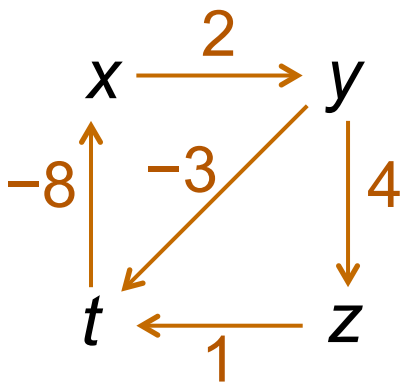
Fondamental pour la vérification d'automates temporisés, cf cours 4 du 30/03/2016 et séminaire de Kim Larsen

Inéquations aux différence : algorithme

- Satisfiabilité polynomiale
- Non satisfiable ⟺ existence d'un cycle de poids négatif

→ vieil algorithme de Bellman-Ford en recherche opérationnelle

$x - y \leq 2$
$y - z \leq 4$
$y - t \leq 7$
$z - t \leq 1$
$t - x \leq -8$

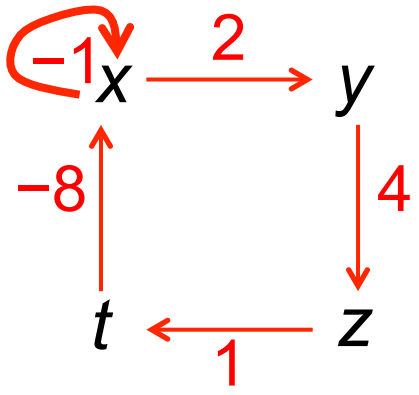


Inéquations aux différence : algorithme

- Satisfiabilité polynomiale
- Non satisfiable ⟺ existence d'un cycle de poids négatif

→ algorithme de plus courts chemins (recherche opérationnelle)

$x - y \le 2$
$y - z \le 4$
$y - t \le -3$
$z - t \le 1$
$t - x \le -8$



$x - y \le 2$
$y - z \le 4$
$z - t \le 1$
$t - x \le -8$

$x - y + y - z + z - t + t - x \le 2 + 4 + 1 - 8$

~~$0 \le -1$~~

Agenda

1. Principe général de SMT
2. Egalité avec fonctions non interprétées
3. Tableaux
4. Arithmétiques entières et réelles
5. Bitvecteurs
6. Allocation mémoire et pointeurs
7. Combinaisons de théories
8. Conclusion

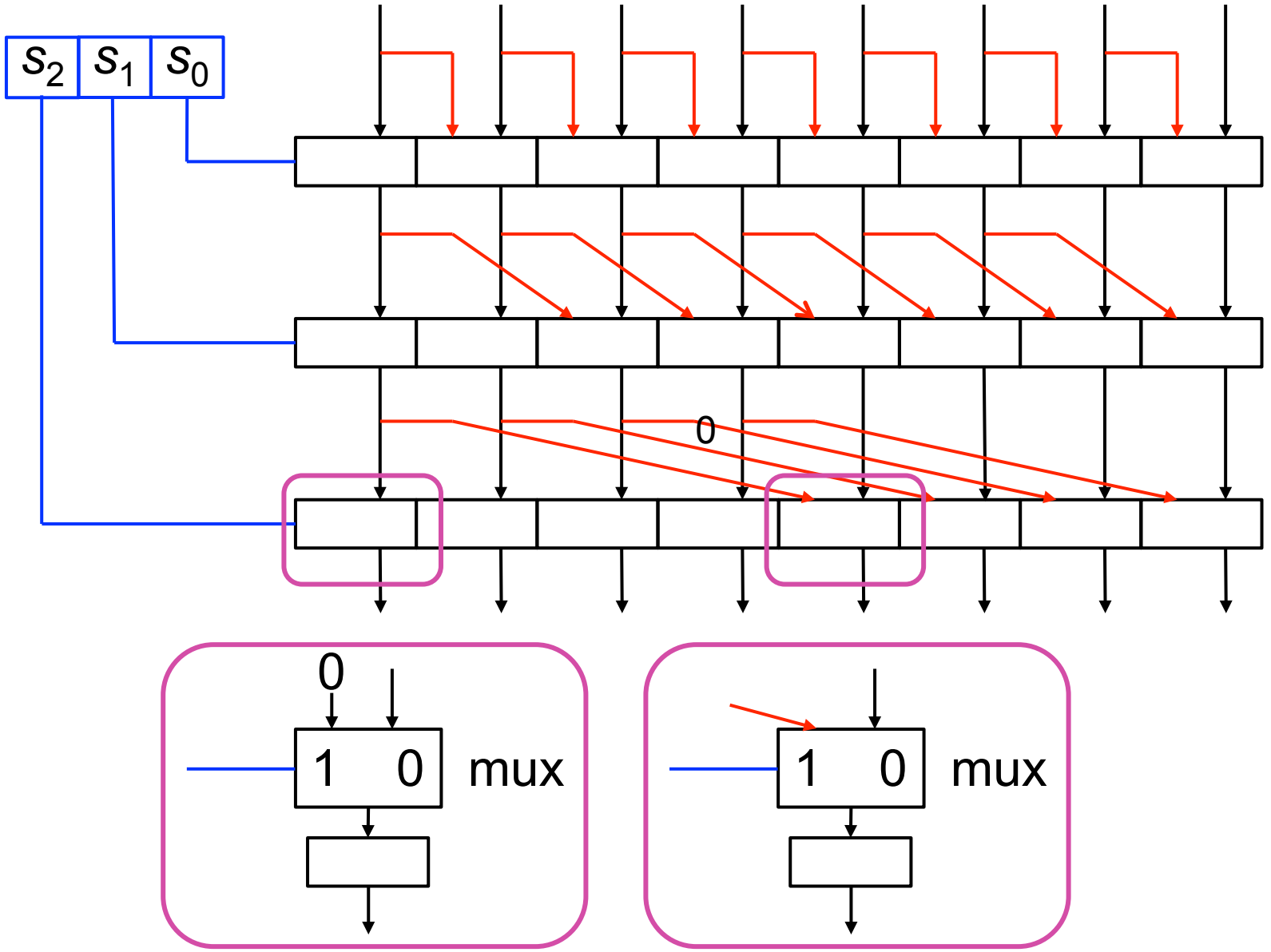
Bitvecteurs

- Les bitvecteurs sont essentiels pour l'implémentation efficace des algorithmes, soit manuelle, soit de préférence automatique par compilation optimisée
- Mais leur usage est délicat et leurs bugs très sournois !
- Plusieurs types de codages et d'opérations :
  - entiers bornés, arithmétique modulaire signée ou non signée : +, −, ∗, /, % (modulo ??), =, ≠, <, ≤, >, ≥, extension de signe
  - codage d'ensembles par leur fonctions caractéristiques
    intersection : x&y, union : x|y, complémentaire : !x,
    union exclusive : xˆy
  - décalages signés ou non signés : x>>y, x<<y, rotations, etc.
    multiplexeur : x?y,z
  - nombres flottants et leur arithmétique

Vérification des bitvecteurs par Bit Blasting

- Idée : transformer la vérification en problème SAT
- Opérations ensemblistes : directement par ∧, ∨,¬
- Opérations arithmétiques : implémentation par circuits booléens
  les plus simples possibles !
  - addition, soustraction : additionneur / soustracteur de l'école, efficace
  - multiplication : par décalages et additions successives, cher ! :
  - division générale :
  - égalité : bit à bit ; différence ≠ → ¬=
  - comparaisons : de gauche à droite
- Décalages, rotations :
  - barrel shifter : composer des décalages de 1,2,4,8,16... bits
    en envoyant les bits de la décomposition binaire du nombre
    de décalages / rotations dans des tableaux de multiplexeurs

Right Barrel Shifter 8 bits



Exemples de calcul sur bitvecteurs

- Un entier n est-il une puissance de 2 ?

```
unsigned n;
int IsPowerOf2 (x) {
    return n & −n == 0;
}
```

- Calculs compliqués, à effectuer sans test ni boucle
  - un test est une rupture dans le flot des instructions avec les ralentissements d'accès mémoire et de cache associés
  - Une boucle est également une rupture dans le flot d'instructions
  - comment faire le même calcul sans boucle, avec une simple séquence d'instructions machines disponibles ?

Henry S. Warren Jr., *Hacker's Delight*

Calcul de la taille d'un ensemble

- Nombre de 1 dans un mot (= taille de l'ensemble qu'il code)

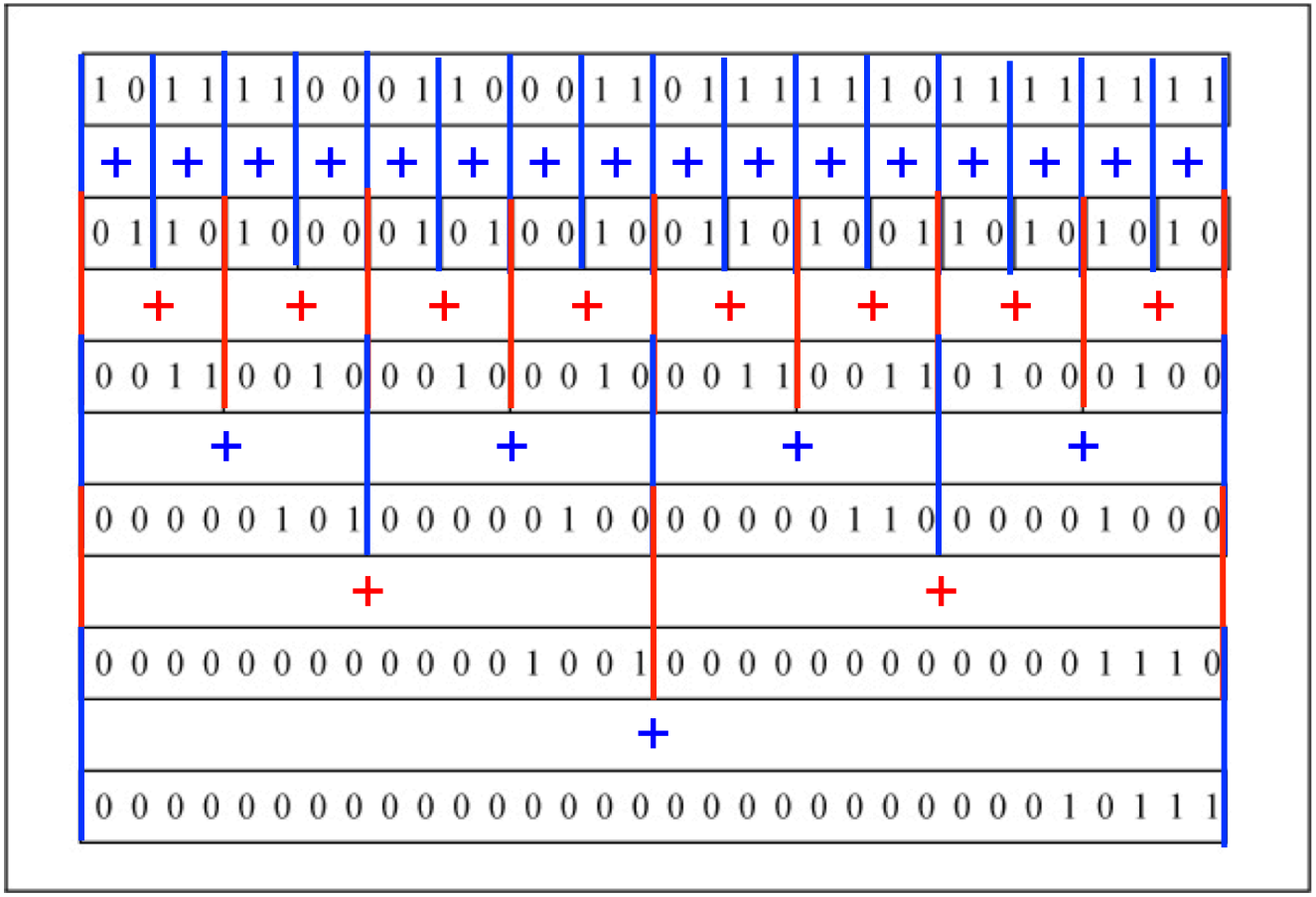


FIGURE 5–1. Counting 1-bits, "divide and conquer" strategy.

source *Hacker's delight*

Exemples de calcul sur bitvecteurs

- Un programme C sans boucle, 21 instructions machine !

```
int NumberOf1 (unsigned x) {
    x = x – ((x>>1) & 0x55555555) ;
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333) ;
    x = (x + (x >> 4)) & 0x0F0F0F0F ;
    x = x + (x >> 8) ;
    x = x + (x >> 16) ;
    return x & 0000003F ;
}
```

- A comparer au dessin de la page précédente !

Distribution de Max sur 2 ensembles

$$Max(X,Y) = \{\max(x,y) \mid x \in X, y \in Y\}$$
$$= \{z \in X \cup Y \mid z \geq \max(\min(X), \min(Y))\}$$

Codage en bitvecteurs :

$$Max(X,Y) = (X \mid Y) \& (X \mid -X) \& (Y \mid -Y)$$

X = 0b00101010 = {1,3,5}

Y = 0b01100100 = {2,5,6}

Max(X,Y) = 0b0110110 = {2,3,5,6}

X = 0b00101010

$-X$ = 0b11010110

$X \mid -X$ = 0b11111110

$= \{z \mid z \geq \min(X)\}$

$X \mid Y$ = 0b01101110 $= \{z \mid X \cup Y\}$

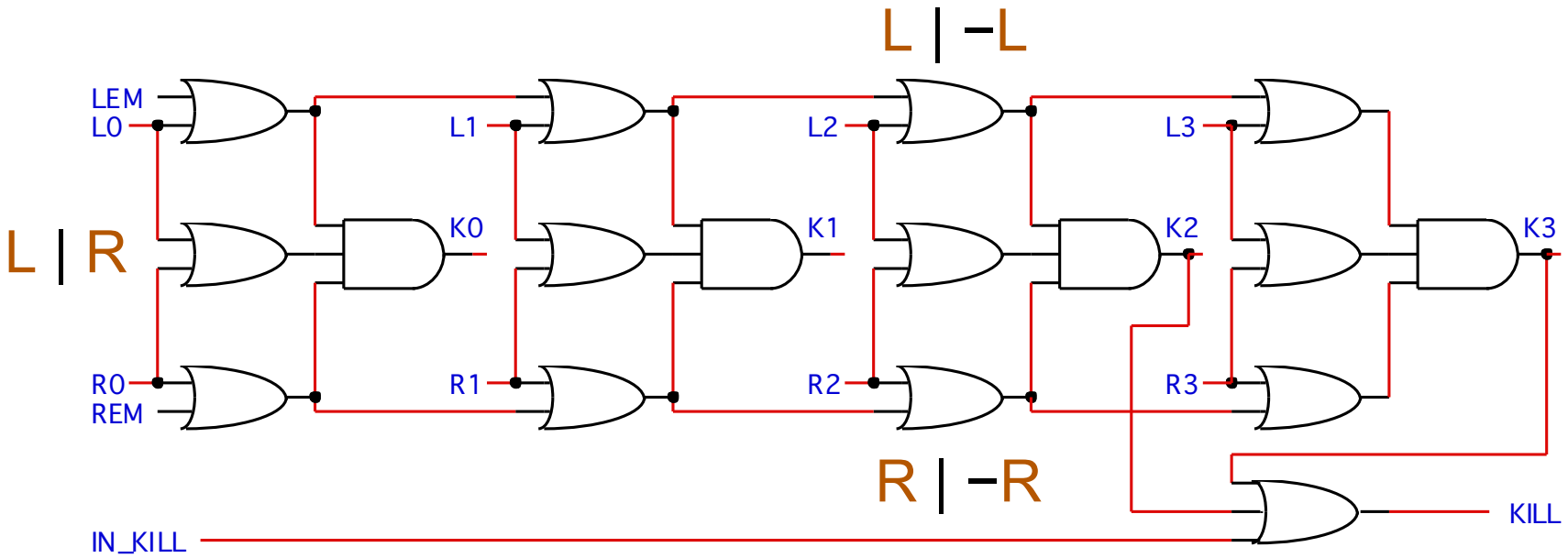
& $X \mid -X$ = 0b11111110 $= \{z \mid z \geq \min(Y)\}$

& $Y \mid -Y$ = 0b11111100 $= \{z \mid z \geq \min(Y)\}$

Max(X,Y) = 0b01101100 = {2,3,5,6}

Preuve :
Why3
+ Alt-Ergo
+ CVC4

Esterel : Synchroniseur du parallèle (Gonthier) (cf cours 4 du 23 avril 2013)



$$K = Max(L, R)$$

Agenda

1. Principe général de SMT
2. Egalité avec fonctions non interprétées
3. Arithmétiques entières et réelles
4. Bitvecteurs
5. Tableaux
6. Quantificateurs
7. Conclusion

Théorie axiomatique des tableaux (Mc Carthy)

Deux opérations avec a tableau et i index entier

$read(a, i) \rightarrow v$

$write(a, i, v)$

Deux axiomes :

1. Intuitivement : $a[i] := v \Rightarrow a[i] = v$

   Algébriquement : $read(write(a, i, v), i) = v$

2. Intuitivement, après $a[i] := v$, *si* $j \neq i$ la valeur de $a[j]$ est inchangée

   Algébriquement : $\forall a,i,j,v.\ i \neq j \Rightarrow read(write(a, i, v), j) = read(a, j)$

fonctions non interprétées
+ arithmétique
= indécidable

→ restrictions sur les calculs d'indices

Exemple de preuve sur tableaux

```
for (int n := 0; n<100; n++) {
```

$\forall i.\ i < n \Rightarrow a[i] = 0$

```
  a[n] := 0 ;
```

$\forall i.\ i \leq n \Rightarrow a[i] = 0$

```
}
```

$n = 99 \wedge \forall i,\ i \leq n \Rightarrow a[i] = 0$

Règle de l'écriture de tableaux :

remplacer l'instruction a[n] := 0 par

$a'[n] = 0 \wedge \forall j \neq n.\ a'[j] = a[j]$

Preuve par contradiction

$\forall i.\ i < n \Rightarrow a[i] = 0$

a[n] := 0 ;

$\forall i.\ i \leq n \Rightarrow a[i] = 0$

réfuter

$\forall i.\ i < n \Rightarrow a[i] = 0$

a[n] := 0 ;

$\exists i.\ i \leq n \wedge a[i] \neq 0$

Règle de l'écriture de tableau

$$\begin{aligned} & \forall i.\ i < n \Rightarrow a[i] = 0 \\ \wedge\ & a'[n] = 0 \\ \wedge\ & \forall j.\ j \neq n \Rightarrow a'[j] = a[j] \\ \wedge\ & \exists i.\ i \leq n \wedge a'[i] \neq 0 \end{aligned}$$

Instanciation de l'existentiel (Skolémisation)

$$\begin{aligned} & \forall i.\ i < n \Rightarrow a[i] = 0 \\ \wedge\ & a'[n] = 0 \\ \wedge\ & \forall j.\ j \neq n \Rightarrow a'[j] = a[j] \\ \wedge\ & k \leq n \wedge a'[k] \neq 0 \end{aligned}$$

Instanciation de l'existentiel

$$\begin{aligned} & \forall i.\ i < n \Rightarrow a[i] = 0 \\ \wedge\ & a'[n] = 0 \\ \wedge\ & \forall j.\ j \neq n \Rightarrow a'[j] = a[j] \\ \wedge\ & k \leq n \wedge a'[k] \neq 0 \end{aligned}$$

remplacement de l'universel par les index libres n et k

$$\begin{aligned} & (\cancel{n < n \Rightarrow a[n] = 0}) \wedge (k < n \Rightarrow a[k] = 0) \\ \wedge\ & a'[n] = 0 \\ \wedge\ & (\cancel{n \neq n \Rightarrow a'[n] = a[n]}) \\ \wedge\ & (k \neq n \Rightarrow a'[k] = a[k]) \\ \wedge\ & k \leq n \wedge a'[k] \neq 0 \end{aligned}$$

simplification des trivialités

$$\begin{aligned} & (k < n \Rightarrow a[k] = 0) \\ \wedge\ & a'[n] = 0 \\ \wedge\ & k \neq n \Rightarrow a'[k] = a[k] \\ \wedge\ & k \leq n \wedge a'[k] \neq 0 \end{aligned}$$

Preuve par contradiction

Passage aux fonctions non interprétées puis cas arithmétiques

$$(k < n \Rightarrow A(k) = 0) \wedge A'[n] = 0 \wedge (k \neq n \Rightarrow A'(k) = A(k)) \wedge k \leq n \wedge A'(k) \neq 0$$

1. Cas $k < n$:

~~$A(k) = 0 \wedge (A'(k) = A(k)) \wedge A'(k) \neq 0$~~

2. Cas $k = n$:

~~$A'(n) = 0 \wedge A'(n) \neq 0$~~

3. Cas $k > n$:

~~$k \leq n$~~

... On est vraiment content de faire faire tout ça à la machine !

Agenda

1. Principe général de SMT
2. Egalité avec fonctions non interprétées
3. Arithmétiques entières et réelles
4. Bitvecteurs
5. Tableaux
6. Combinaisons de théories
7. Conclusion

Combiner les théories : Nelson-Oppen

- Formule combinant librement plusieurs théories :

$$(f(x_1,0) \geq x_3) \wedge (f(x_2,0) \leq x_3) \wedge (x_1 \geq x_2) \wedge (x_2 \geq x_1) \wedge (x_3 - f(x_1,0) \geq 1)$$

- Purification : introduction de variables pour séparer les théories (ces variables peuvent être partagées)

$$(f_1 \geq x_3) \wedge (f_2 \leq x_3) \wedge (x_1 \geq x_2) \wedge (x_2 \geq x_1) \wedge (x_3 - f_1 \geq 1) \wedge (c_0 = 0)$$

arithmétique

$$\wedge (f_1 \Leftrightarrow f(x_1,c_0)) \wedge (f_2 \Leftrightarrow f(x_2,c_0))$$

non-interprété

partage de variable

Mélanger les théories : Nelson-Oppen

$$(f_1 \geq x_3) \wedge (f_2 \leq x_3) \wedge (x_1 \geq x_2) \wedge (x_2 \geq x_1) \wedge (x_3 - f_1 \geq 1) \wedge (c_0 = 0)$$
$$\wedge (f_1 = f(x_1, c_0)) \wedge (f_2 = f(x_2, c_0))$$

1. Arithmétique → $x_1 = x_2$

$$(f_1 \geq x_3) \wedge (f_2 \leq x_3) \wedge (x_1 \geq x_2) \wedge (x_2 \geq x_1) \wedge (x_3 - f_1 \geq 1) \wedge (c_0 = 0)$$
$$\wedge (f_1 = f(x_1, c_0)) \wedge (f_2 = f(x_2, c_0)) \wedge (x_1 = x_2)$$

2. Non-interprété : $f_1 = f_2$

$$(f_1 \geq x_3) \wedge (f_2 \leq x_3) \wedge (x_1 \geq x_2) \wedge (x_2 \geq x_1) \wedge (x_3 - f_1 \geq 1) \wedge (c_0 = 0)$$
$$\wedge (f_1 = f(x_1, c_0)) \wedge (f_2 = f(x_2, c_0)) \wedge (x_1 = x_2) \wedge (f_1 = f_2)$$

3. Arithmétique → $f_1 = x_3$ → *Unsat*

$$(f_1 \geq x_3) \wedge (f_2 \leq x_3) \wedge (x_1 \geq x_2) \wedge (x_2 \geq x_1) \wedge (x_3 - f_1 \geq 1) \wedge (c_0 = 0)$$
$$\wedge (f_1 = f(x_1, c_0)) \wedge (f_2 = f(x_2, c_0)) \wedge (x_1 = x_2) \wedge (f_1 = f_2) \wedge (f_1 = x_3)$$

Résumé de la méthode (provisoire)

1. Si la formule F fait intervenir un ensemble $\{T_i\}$ de théories, la purifier pour construire une conjonction $\wedge_i F_i$ où chaque F_i rassemble les clauses dans T_i et les égalités entre variables.

2. Ensuite itérer ainsi :
   - 2.1. si chaque F_i est satisfiable dans sa théorie T_i, alors F est satisfiable.
   - 2.2. sinon, déduire de n'importe quelle F_i de nouvelles égalités de variables, les ajouter à F et recommencer

Mais la méthode est-elle correcte ?

- Oui pour les théories infinitaires et convexes
  - infinitaire : admet des modèles infinis
  - convexe (constructive) : si $F \Rightarrow \bigvee_i F_i$ alors $\exists i.\, F \Rightarrow F_i$

- Exemples et contre-exemples
  - arithmétiques réelles : infinitaires et convexes
  - bitvecteurs : ni infinitaire ni convexe
  - arithmétiques entières : infinitaires mais non convexes

OUI : $(x \geq 0) \wedge (x \leq 1) \Rightarrow (x = 0) \vee (x = 1)$

NON : $(x \geq 0) \wedge (x \leq 1) \Rightarrow (x = 0)$

NON : $(x \geq 0) \wedge (x \leq 1) \Rightarrow (x = 1)$

Traitement de théories non-convexes

- Exploiter ou engendrer des disjonctions d'égalités et raisonner par cas

$$\text{si } F \Rightarrow (x = 1) \vee (x = 2),$$

$$\text{il suffit de vérifier } (F \wedge (x = 1)) \wedge (F \wedge (x = 2))$$

Beaucoup d'autres méthodes, voir le séminaire de Sylvain Conchon ce jour sur ALT-ERGO

Conclusion

- SMT est nettement plus technique et compliqué que SAT, mais indispensable dans beaucoup d'applications
- Les algorithmes des diverses théories sont très différent les uns des autres
- Pourtant apparemment anodine, la théorie des fonctions non-interprétés est très importante
- La combinaison de théories pose des problèmes difficiles, et n'a pas toujours été correcte
- Les quantificateurs demandent des heuristiques subtiles

Mais heureusement, tout cela est implémenté
dans de nombreux systèmes, dont plusieurs en *open source*
(académiques et industriels)

Bibliographie

Leonardo De Moura et Nikolaj Björner

Satisfiability Modulo Theories: Introduction and Applications

Comm. ACM vol. 54 no. 9, pp. 69-77, sept. 2011

Daniel Kroening et Ofer Strichman

Decision Procedures – An Algorithmic Point of View

Springer, 2008