Vérification déductive de programmes avec Why3

Jean-Christophe Filliâtre
CNRS

EJCP 2012

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

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

quand est-ce que f renvoie 91 ? termine-t-elle toujours ?

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

quand est-ce que f renvoie 91 ? termine-t-elle toujours ?
est-ce équivalent au programme suivant ?

```
e ← 1
while e > 0 do
   if n > 100 then
      n ← n − 10
      e ← e − 1
   else
      n ← n + 11
      e ← e + 1
return n
```

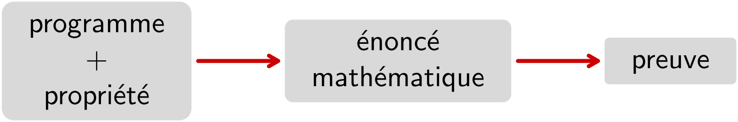
ce code Java trie-t-il bien un tableau de booléens ?

```
int i = 0, j = a.length - 1;
while (i < j)
  if (!a[i]) i++;
  else if (a[j]) j--;
  else swap(a, i++, j--);
```

Plus difficile

ce programme C est-il correct ?

```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,(
c+d)/2));return f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0<<q),0,0));}
```



aujourd'hui rendu possible par la révolution SMT

L'outil Why

- développé depuis une dizaine d'années
  dans l'équipe ProVal (LRI / INRIA)

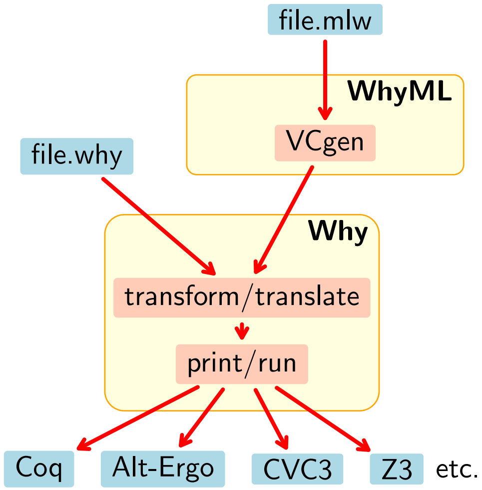
- utilisé pour la preuve
  - de programmes Java : Krakatoa (Marché Paulin Urbain)
  - de programmes C : Caduceus (Filliâtre Marché) hier
    puis greffon Jessie de Frama-C (Marché Moy) aujourd'hui
  - d'algorithmes
  - de programmes probabilistes (Barthe et al.)
  - de programmes cryptographiques (Vieira)

nouvelle version, développée depuis février 2010

auteurs : F. Bobot, JCF, C. Marché, G. Melquiond, A. Paskevich

http://why3.lri.fr/

Schématiquement



partie 1

Logique

démo

Récapitulation

logique de Why3 = logique du premier ordre polymorphe, avec

- types algébriques (mutuellement) récursifs
- symboles de fonctions/prédicats (mutuellement) récursifs
- prédicats (mutuellement) inductifs
- let-in, match-with, if-then-else

la logique du premier ordre simplement typée (*many-sorted first-order logic*) est standard ; voir par exemple Extensions of first-order logic de Manzano (CUP, 1996)

comment définir la logique du premier ordre polymorphe ?

Definition (signature)

un triplet $\Sigma = (S, F, P)$ où

- S ensemble de sortes
- F ensemble de symboles de fonctions $\phi : s_1 \times \cdots \times s_n \to s$, avec $n \in \mathbb{N}$ et $s_1, \ldots, s_n, s \in S$
- P ensemble de symboles de prédicats $\pi : s_1 \times \cdots \times s_n$, avec $n \in \mathbb{N}$ et $s_1, \ldots, s_n \in S$

exemple

```
type nat
function zero : nat
function succ nat : nat
predicate is_zero nat
```

Definition (terme, formule)

$$
\begin{array}{rcll}
t & ::= & x_s & s \in S \\
 & | & \phi(t, \ldots, t) & \phi \in F
\end{array}
$$

$$
\begin{array}{rcll}
f & ::= & \pi(t, \ldots, t) & \pi \in P \\
 & | & \mathsf{true} \mid t = t \mid f \vee f \mid \neg f & \\
 & | & \exists x_s.\, f & s \in S
\end{array}
$$

on définit la notion de terme bien typé (resp. formule bien formée)

sucre syntaxique : false, $f_1 \wedge f_2$, $f_1 \Rightarrow f_2$, $f_1 \Leftrightarrow f_2$, $\forall x_s.\, f$

Definition (modèle)

Un modèle est un triplet de familles

$$M = ((M_s)_{s \in S}, (I_\phi)_{\phi \in F}, (I_\pi)_{\pi \in P})$$

tel que

1. pour tout $s \in S$, M_s ests un ensemble non vide (universe)
2. pour tout $\phi : s_1 \times \cdots \times s_n \to s \in F$, I_ϕ est une fonction de $M_{s_1} \times \cdots \times M_{s_n}$ vers M_s
3. pour tout $\pi : s_1 \times \cdots \times s_n \in P$, I_π est une fonction de $M_{s_1} \times \cdots \times M_{s_n}$ vers $\{\top, \bot\}$

Definition (environnement)

Soit M un modèle. Un environnement E pour M est une fonction partielle des variables vers les univers telle que, si x_s est dans le domaine de E, alors $E(x_s) \in M_s$

Definition (interprétation)

Soient M un modèle, E un environnement pour M, t un terme de type s. L'interprétation de t, notée $M_E(t)$, est l'élément de M_s défini par

$$\begin{array}{rcl} M_E(x_s) & = & E(x_s) \\ M_E(\phi(t_1, \ldots, t_n)) & = & I_\phi(M_E(t_1), \ldots, M_E(t_n)). \end{array}$$

Definition (valeur de vérité)

M un modèle, E un environnement pour M. La valeur de vérité de f, notée $\mathsf{T}_E(f)$, est l'élément de $\{\top, \bot\}$ défini par

$$
\begin{array}{rcl}
\mathsf{T}_E(\pi(t_1, \ldots, t_n)) & = & I_\pi(M_E(t_1), \ldots, M_E(t_n)) \\
\mathsf{T}_E(\mathrm{true}) & = & \top \\
\mathsf{T}_E(t_1 = t_2) & = & \top \text{ ssi } M_E(t_1) = M_E(t_2) \\
\mathsf{T}_E(f_1 \vee f_2) & = & \top \text{ ssi } \mathsf{T}_E(f_1) = \top \text{ or } \mathsf{T}_E(f_2) = \top \\
\mathsf{T}_E(\neg f_1) & = & \top \text{ ssi } \mathsf{T}_E(f_1) = \bot \\
\mathsf{T}_E(\exists x_s.\, f_1) & = & \top \text{ ssi il existe } m \in M_s \text{ such that} \\
 & & \mathsf{T}_{E + x_s \mapsto m}(f_1) = \top.
\end{array}
$$

la valeur de vérité d'une formule close f ne dépend pas de E ; on l'écrit $\mathsf{T}(f)$

Definition (validité)

Soit f une formule close. On dit que f est vraie dans le modèle M si $\mathsf{T}(f) = \top$. On dit que f est valide si elle est vraie dans tous les modèles.

Definition (théorème)

Soit Γ un ensemble de formules closes. On dit que f est une conséquence logique, ou théorème, de Γ, noté $\Gamma \models f$, si f est vraie dans tous les modèles pour lesquels toutes les formules de Γ sont vraies.

note : l'ensemble Γ n'est pas nécessairement fini ajoutons maintenant la notion de polymorphisme

on va se ramener au cas de la logique du premier ordre simplement typée

Definition (sorte polymorphe)

Une sorte polymorphe est une paire $s : n$ avec s un symbole de sorte et $n \in \mathbb{N}$ son arité. (Une sorte d'arité 0 est dite monomorphe.)

Definition (type polymorphe)

Soit S un ensemble de sortes polymorphes. Un type polymorphe τ est défini ainsi

$$\begin{array}{rcll} \tau & ::= & \alpha & \text{variable de type} \\ & | & s(\tau, \ldots, \tau) & s \in S \end{array}$$

on définit la notion de type bien formé

un type τ sans variable de type est dit monomorphe

Definition (signature)

Une signature est une triplet (S, F, P) avec

- S un ensemble de sortes polymorphes
- F un ensemble de symboles de fonctions

  $$\phi[\alpha_1, \ldots, \alpha_k] : \tau_1 \times \cdots \times \tau_n \to \tau$$

  avec $k, n \in \mathbb{N}$
- P un ensemble de symboles de prédicats

  $$\pi[\alpha_1, \ldots, \alpha_k] : \tau_1 \times \cdots \times \tau_n$$

  avec $k, n \in \mathbb{N}$

un symbole est dit polymorphe (resp. monomorphe) si $k > 0$ (resp. $k = 0$)

termes et formules sont construits à partir de variables et d'instances de symboles

Definition (terme)

$$
\begin{array}{rcll}
t & ::= & x_\tau & \\
 & | & \phi[\tau, \ldots, \tau](t, \ldots, t) & \phi \in F
\end{array}
$$

Definition (formule)

$$
\begin{array}{rcll}
f & ::= & \pi[\tau, \ldots, \tau](t, \ldots, t) & \pi \in P \\
 & | & \mathsf{true} \mid t = t \mid f \vee f \mid \neg f & \\
 & | & \exists x_\tau .\, f &
\end{array}
$$

Definition

Soit $\Sigma = (S, F, P)$ une signature polymorphe. On définit la signature $\mathrm{mono}(\Sigma) = (S_1, F_1, P_1)$ ainsi :

- S_1 est l'ensemble des types monomorphes construits à partir de Σ
- F_1 est l'ensemble des instances monomorphes de F, c'est-à-dire tous les $\phi[\tau_1, \ldots, \tau_k]$ avec τ_i monomorphe
- P_1 est l'ensemble des instances monomorphes de P, c'est-à-dire tous les $\pi[\tau_1, \ldots, \tau_k]$ avec τ_i monomorphe

Definition

Soit f une formule. On note mono(f) l'ensemble de toutes les instances monomorphes de f, c'est-à-dire de toutes les formules

$$f[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n]$$

où les α_i sont les variables de types de f, et τ_i des types monomorphes.

Definition (validité)

Soit Γ un ensemble de formules closes (possiblement polymorphes) et f une formule close et monomorphe. On dit que f est un théorème de Γ si et seulement si

$$\mathrm{mono}(\Gamma) \models f$$

dans la logique simplement typée, pour la signature $\mathrm{mono}(\Sigma)$.

Difficulté

l'ensemble mono(Γ) est rapidement infini

on ne peut pas décider quel sous-ensemble de mono(Γ) sera nécessaire à la preuve de f

$\Rightarrow$ il faut traduire Γ dans la logique simplement typée par un nombre fini d'axiomes

en savoir plus :

- *Expressing Polymorphic Types in a Many-Sorted Language*. (FroCos 2011)

Récapitulation

- déclaration de type
  - abstrait : `type t`
  - alias : `type t = list int`
  - algébrique : `type list 'a = Nil | Cons 'a (list 'a)`

- déclaration de fonction / prédicat
  - non interprété : `function f int : int`
  - défini : `predicate non_empty (l: list 'a) = l <> Nil`

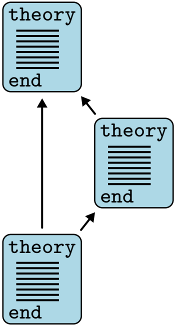
- déclaration de prédicat inductif
  - `inductive trans t t = ...`

- axiome / lemme / but
  - `goal G: forall x: int. x >= 0 -> x*x >= 0`

logique organisée en théories

une théorie T_1 peut être

- utilisée (`use`) dans une théorie T_2
- clonée (`clone`) par une autre théorie T_2

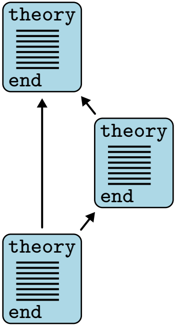


logique organisée en théories

une théorie T_1 peut être

- utilisée (`use`) dans une théorie T_2
  - les symboles de T_1 sont partagés
  - les axiomes de T_1 restent des axiomes
  - les lemmes de T_1 deviennent des axiomes
  - les buts de T_1 sont ignorés

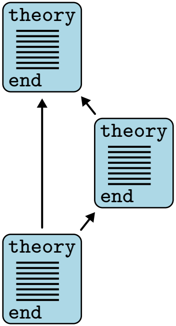
- clonée (`clone`) par une autre théorie T_2



logique organisée en théories

une théorie T_1 peut être

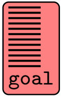
- utilisée (`use`) dans une théorie T_2
- clonée (`clone`) par une autre théorie T_2
  - les déclarations de T_1 sont copiées ou remplacées
  - les axiomes de T_1 restent des axiomes ou deviennent des lemmes/buts
  - les lemmes de T_1 deviennent des axiomes
  - les buts de T_1 sont ignorés

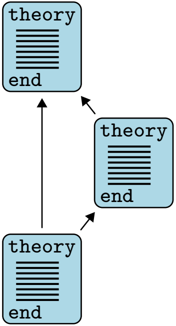


Sous le capot

une technologie pour parler à l'oreille des démonstrateurs

organisée autour de la notion de tâche = contexte + but

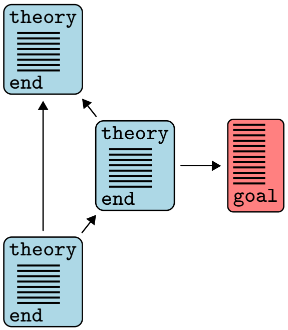




Alt-Ergo

Z3

Vampire

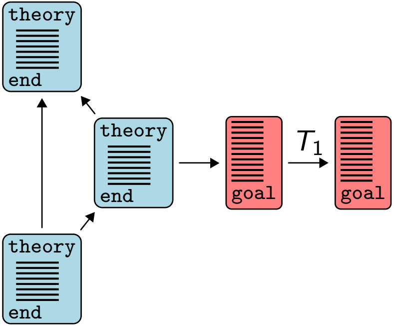


Alt-Ergo

Z3

Vampire

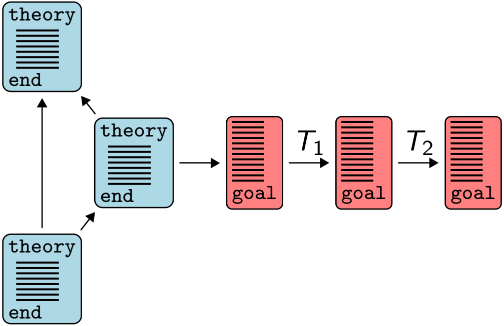
Le parcours d'une tâche



Alt-Ergo

Z3

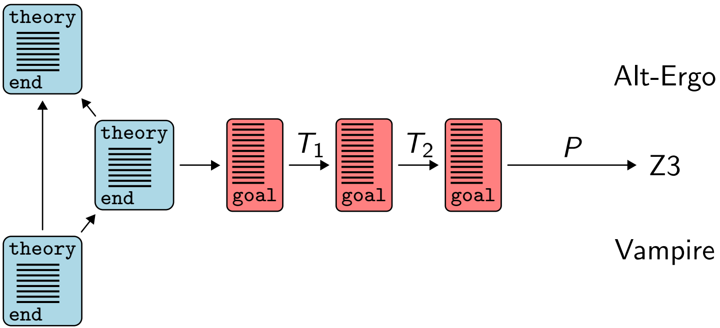
Vampire



Alt-Ergo

Z3

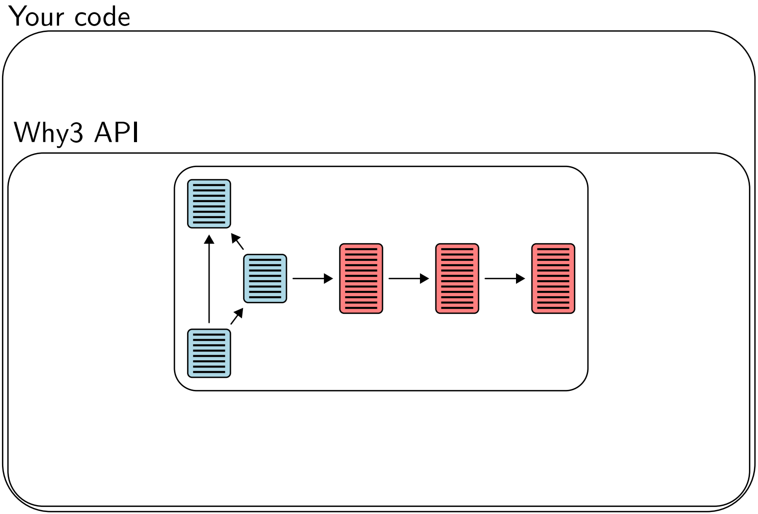
Vampire

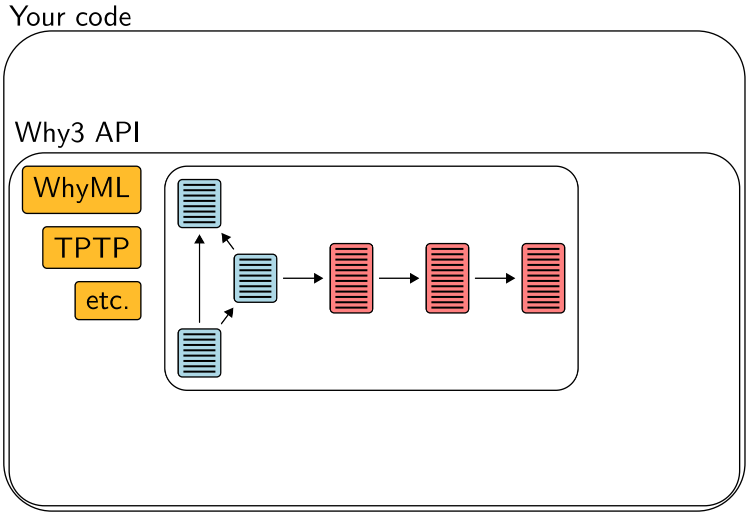

theory
end
theory
end
theory
end
goal
T_1
goal
T_2
goal
P
Alt-Ergo
Z3
Vampire

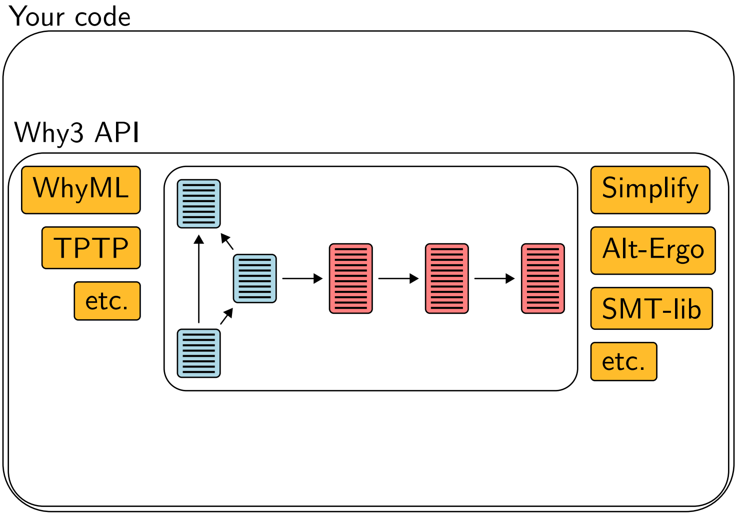
Pilote

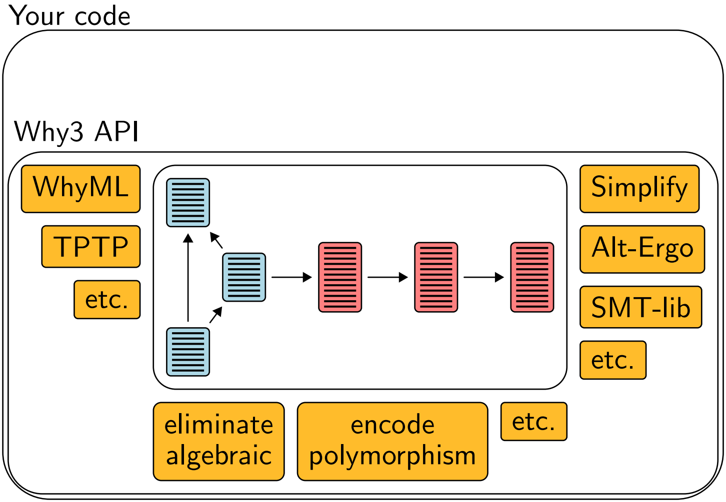
le parcours d'une tâche est piloté par un fichier

- transformations à appliquer
- format de sortie
  - syntaxe de sortie
  - symboles / axiomes prédéfinis
- diagnostique des messages du démonstrateur


Your code
Why3 API


Your code
Why3 API
WhyML
TPTP
etc.


Your code
Why3 API
WhyML
TPTP
etc.
Simplify
Alt-Ergo
SMT-lib
etc.


Your code
Why3 API
WhyML
TPTP
etc.
Simplify
Alt-Ergo
SMT-lib
etc.
eliminate algebraic
encode polymorphism
etc.

Récapitulation

- nombreux démonstrateurs supportés
  - Coq, SMT, TPTP, Gappa
- système extensible par l'utilisateur
  - syntaxe d'entrée
  - transformations
  - syntaxe de sortie
- efficace
  - le résultat des transformations est mémoïsé

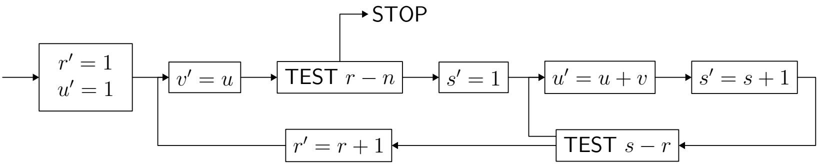
en savoir plus :

- *Why3 : Shepherd your herd of provers.* (Boogie 2011)

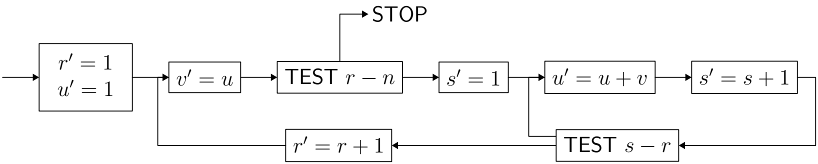
partie 2

Preuve de programmes

A. M. Turing. Checking a Large Routine. 1949.



A. M. Turing. Checking a Large Routine. 1949.



$u \leftarrow 1$
for $r = 0$ to $n - 1$ do
    $v \leftarrow u$
    for $s = 1$ to r do
        $u \leftarrow u + v$

démo (accès au code)

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

démo (accès au code)

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

démo (accès au code)

$e \leftarrow 1$
while $e > 0$ do
  if $n > 100$ then
    $n \leftarrow n - 10$
    $e \leftarrow e - 1$
  else
    $n \leftarrow n + 11$
    $e \leftarrow e + 1$
return n

démo (accès au code)

Remarque

comme on l'a vu avec la fonction 91, prouver la terminaison peut nécessiter de prouver également des propriétés fonctionnelles

un autre exemple :

- l'algorithme du lièvre et de la tortue de Floyd

- pré/postcondition

```
let f x = { P } ... { Q }
```

- invariant de boucle

```
while  ...  do invariant { I } ... done

for i = ... do invariant { I(i) } ... done
```

la terminaison d'une boucle (resp. fonction récursive) est garantie par un variant

$$\texttt{variant } \{t\} \texttt{ with } R$$

- R est une relation d'ordre bien fondée
- t décroît pour R à chaque tour de boucle (resp. chaque appel récursif)

par défaut, t est de type `int` et R la relation

$$y \prec x \stackrel{\text{def}}{=} y < x \land 0 \leq x$$

la correction d'un programme est traduite en une formule logique grâce à un calcul de plus faibles préconditions on se donne

- une expression de programme e
- une postcondition f
- un ensemble $\vec{q}$ de postconditions exceptionnelles $E_1 \rightarrow f_1, \ldots, E_n \rightarrow f_n$

on définit alors la plus faible précondition

$$\mathrm{wp}(e, f, \vec{q})$$

par récurrence sur la structure de e

Propriété

si $\text{wp}(e, f, \vec{q})$ est valide alors

- les assertions de e sont vérifiées
- si l'évaluation de e termine sur une valeur v alors on a $f[\mathit{result} \leftarrow v]$
- si l'évaluation de e lève l'exception E_i alors on a f_i

Plus faible précondition : exemples

exemple : let-in

$$\mathrm{wp}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, f, \vec{q}) \stackrel{\mathrm{def}}{=} \mathrm{wp}(e_1, \mathrm{wp}(e_2, f, \vec{q})[x \leftarrow \mathit{result}], \vec{q}).$$

cas particulier

$$\mathrm{wp}(e_1;\ e_2, f, \vec{q}) \stackrel{\mathrm{def}}{=} \mathrm{wp}(e_1, \mathrm{wp}(e_2, f, \vec{q}), \vec{q}).$$

autre exemple : assert

$$\mathrm{wp}(\mathsf{assert}\ f_1, f_2, \_) \stackrel{\mathrm{def}}{=} f_1 \wedge f_2.$$

voir les notes de cours

jusqu'à présent, on s'est limité aux entiers

considérons maintenant des structures plus complexes

- tableaux
- types algébriques

la bibliothèque de Why3 fournit des tableaux

```
use import module array.Array
```

c'est-à-dire

- un type polymorphe

```
array 'a
```

- une opération d'accès, notée

```
a[e]
```

- une opération d'affectation, notée

```
a[e1] <- e2
```

- des opérations `create`, `append`, `sub`, `copy`, etc.

trier un tableau de booléens, avec l'algorithme suivant

```
let two_way_sort (a: array bool) =
  let i = ref 0 in
  let j = ref (length a - 1) in
  while !i < !j do
    if not a[!i] then
      incr i
    else if a[!j] then
      decr j
    else begin
      let tmp = a[!i] in
      a[!i] <- a[!j];
      a[!j] <- tmp;
      incr i;
      decr j
    end
  done
```

False	?	...	?	True
	↑		↑	
	i		j	

démo (accès au code)

Exercice : le drapeau hollandais

un tableau contient des valeurs de trois sortes

```
type color = Blue | White | Red
```

il s'agit de le trier, de manière à avoir au final

... Blue ...	... White ...	... Red ...

```
let dutch_flag (a:array color) (n:int) =
  let b = ref 0 in
  let i = ref 0 in
  let r = ref n in
  while !i < !r do
     match a[!i] with
     | Blue ->
         swap a !b !i;
         incr b;
         incr i
     | White ->
         incr i
     | Red ->
         decr r;
         swap a !r !i
     end
  done
```

exercice : `exo_flag.mlw`

tout comme prouver la terminaison, chercher à montrer la bonne exécution (par ex. pas d'accès en dehors des bornes) peut être arbitrairement compliqué

un exemple :

- calcul des N premiers nombres premiers de Knuth (TAOCP) une idée centrale de la logique de Hoare :

> *tous les types et symboles logiques*
> *peuvent être utilisés dans les programmes*

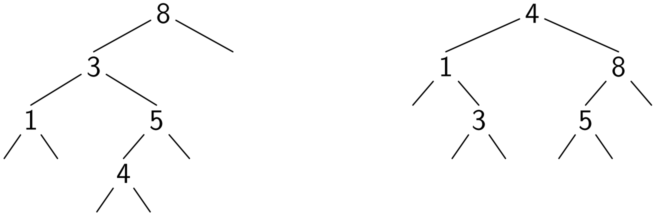
note : on l'a déjà fait avec le type `int` il en va de même des types algébriques en particulier

dans la bibliothèque, on trouve notamment

```
type bool = True | False                    (dans bool.Bool)
type option 'a = None | Some 'a         (dans option.Option)
type list 'a = Nil | Cons 'a (list 'a) (dans list.List)
```

Exemple : *same fringe*

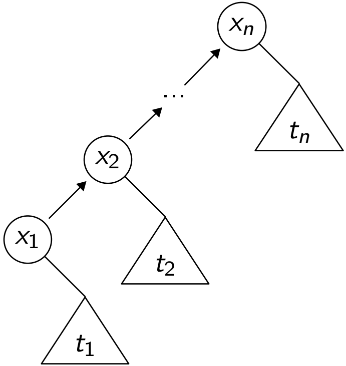
étant donnés deux arbres binaires,
présentent-ils les mêmes éléments dans un parcours infixe ?



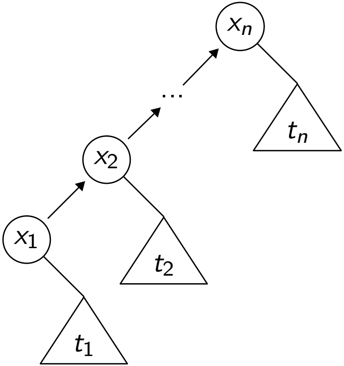
Exemple : *same fringe*

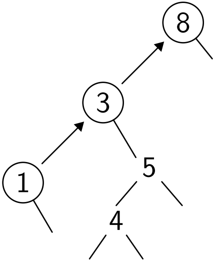
```
type elt

type tree =
  | Empty
  | Node tree elt tree

function elements (t: tree) : list elt = match t with
  | Empty -> Nil
  | Node l x r -> elements l ++ Cons x (elements r)
end

let same_fringe t1 t2 =
  { }
  ...
  { result=True <-> elements t1 = elements t2 }
```

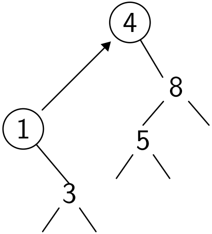
une solution : voir la branche gauche
comme une liste, de bas en haut



une solution : voir la branche gauche
comme une liste, de bas en haut







démo (accès au code)

Exercice : parcours infixe

```
type elt
type tree = Null | Node tree elt tree
```

parcours infixe de `t`, pour stocker ses éléments dans le tableau `a`

```
let rec fill (t: tree) (a: array elt) (start: int) : int =
  match t with
  | Null ->
      start
  | Node l x r ->
      let res = fill l a start in
      if res <> length a then begin
        a[res] <- x;
        fill r a (res + 1)
      end else
        res
   end
```

exercice : `exo_fill.mlw`

partie 3

Modélisation

dans la bibliothèque, on trouve

```
type array 'a model {| length: int; mutable elts: map int 'a |}
```

deux significations différentes :

- dans les programmes, un type abstrait :

  ```
  type array 'a
  ```

- dans la logique, un enregistrement immuable :

  ```
  type array 'a = {| length: int; elts: map int 'a |}
  ```

on ne peut pas coder d'opérations sur le type `array 'a`
(il est abstrait) mais on peut les déclarer

exemples :

```
val ([]) (a: array 'a) (i: int) :
  { 0 <= i < length a }
  'a
  reads a
  { result = Map.get a.elts i }

val ([]<-) (a: array 'a) (i: int) (v: 'a) :
  { 0 <= i < length a }
  unit
  writes a
  { a.elts = Map.set (old a.elts) i v }
```

on peut modéliser de la même manière de nombreuses structures de données, qu'elles soient codées ou non

exemples : piles, files, files de priorité, graphes, etc.

Exemple : tables de hachage

```
type t 'a 'b

val create: int -> t 'a 'b

val clear: t 'a 'b -> unit

val add:  t 'a 'b -> 'a -> 'b -> unit

exception Not_found

val find: t 'a 'b -> 'a -> 'b
```

démo (accès au code)

Limitation

il est également possible de coder les tables de hachage
(cf le code dans le transparent précédent)

```
type t 'a 'b = array (list ('a, 'b))
...
```

cependant, il n'est pas (encore) possible de vérifier que ce code est conforme au modèle précédent

Remarque

l'idée de modélisation n'est pas limitée aux structures impératives

exemple : une file réalisée avec deux listes

```
type queue 'a = {| front: list 'a; lenf: int;
                   rear : list 'a; lenr: int; |}
```

peut être modélisée par une seule liste

```
function sequence (q: queue 'a) : list 'a =
  q.front ++ reverse q.rear
```

Exemple : arithmétique 32 bits

modélisons l'arithmétique 32 bits signée

deux possibilités :

- prouver l'absence de débordement arithmétique
- modéliser fidèlement l'arithmétique de la machine

une contrainte :
ne pas perdre les capacités arithmétiques des démonstrateurs

on introduit un nouveau type pour les entiers 32 bits

```
type int32
```

sa valeur est donnée par

```
function toint int32 : int
```

dans les annotations, on n'utilise que le type `int`

une expression x : `int32` apparaît donc sous la forme `toint` x

on définit la plage des entiers 32 bits

```
function min_int: int = -2147483648
function max_int: int =  2147483647
```

quand on les utilise...

```
axiom int32_domain:
  forall x: int32. min_int <= toint x <= max_int
```

... et quand on les construit

```
val ofint (x:int) :
  { min_int <= x <= max_int }
  int32
  { toint result = x }
```

considérons la recherche dichotomique dans un tableau trié (*binary search*)

montrons l'absence de débordement arithmétique

démo

on a trouvé un bug

le calcul

```
let m = (l + u) / 2 in
```

peut provoquer un débordement arithmétique
(par exemple avec un tableau de 2 milliards d'éléments)

on peut corriger ainsi

```
let m = l + (u - l) / 2 in
```

Principe

la seconde idée centrale de la logique de Hoare

> *on peut identifier statiquement les différents emplacements mémoire ; c'est l'absence d'alias*

en particulier, les emplacements mémoire ne sont pas des valeurs de première classe dans la logique

pour traiter des programmes avec alias,
il faut modéliser la mémoire

exemple : un modèle pour des programmes C
avec des pointeurs de type `int*`

```
type pointer

val memory: ref (map pointer int)
```

une expression C

```
*p
```

devient l'expression Why3

```
!memory[p]
```

il existe des modèles plus subtiles
comme le modèle Burstall / Bornat, dit *component-as-array*

chaque champ de structure devient un tableau

le type C

```
struct List {
  int          head;
  struct List *next;
};
```

est modélisé par

```
type pointer
val head: ref (map pointer int)
val next: ref (map pointer pointer)
```

partie 4

Conclusion

Non traité dans ce cours

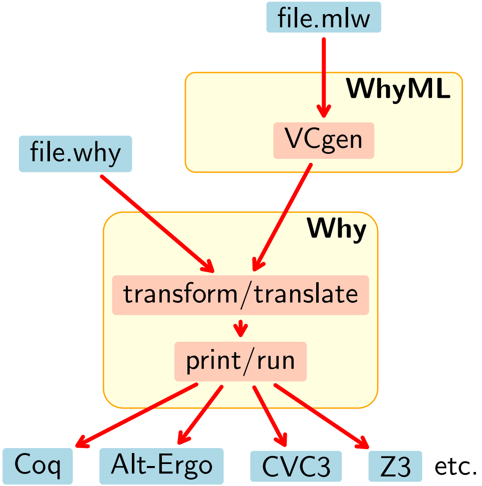
- comment sont exclus les alias
- comment sont calculées les obligations de preuve
- comment les formules sont envoyées aux démonstrateurs
- comment modéliser l'arithmétique flottante
- etc.

on a vu trois usages distincts de Why3

- langage logique
- preuve de programmes
- langage intermédiaire

on peut utiliser Why3 uniquement comme un langage unique pour parler aux démonstrateurs

on encore uniquement pour l'API OCaml de sa logique



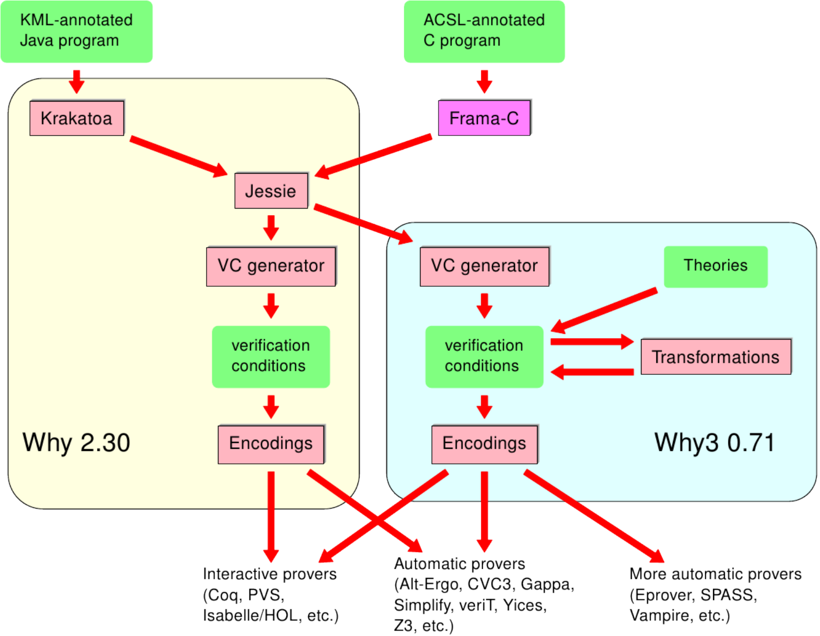
Pour la preuve de programmes ML

51 preuves de programmes avec Why3 sur

`http://proval.lri.fr/gallery/`

note : il y aura (bientôt) une extraction de code OCaml

Comme langage intermédiaire



merci