Vérification déductive de programmes avec Why3

Jean-Christophe Filliâtre
CNRS

JFLA 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
```

Une autre question

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

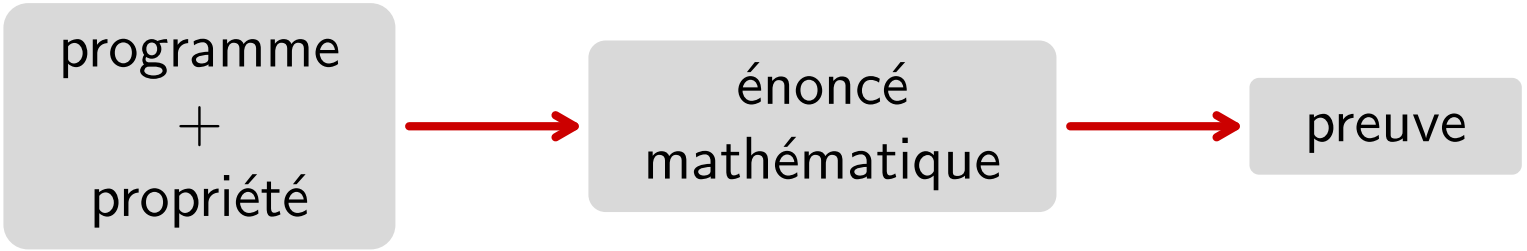
```
int i = 0, j = a.length - 1;
while (i < j)
  if (!a[i]) i++;
  else if (a[j]) j--;
  else swap(a, i++, j--);
```

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));}
```

Vérification déductive de programmes



aujourd'hui rendu possible par la révolution SMT

L'outil Why

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

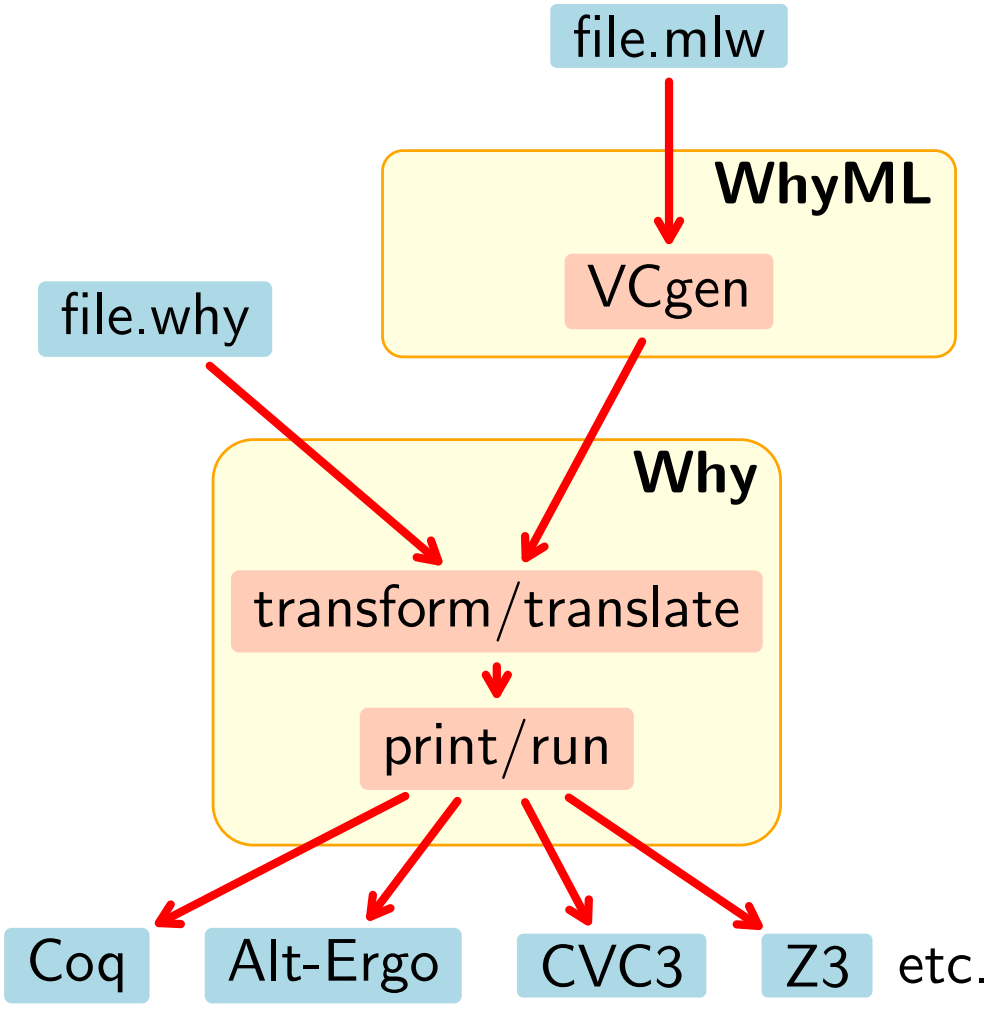
- utilisé pour la preuve
  - de programmes Java : Krakatoa (Marché Paulin Urbain)
  - de programmes C : Caduceus (Filliâtre Marché) hier
    puis greffon Jessie de Frama-C (Marché Moy) aujourd'hui
  - d'algorithmes
  - de programmes probabilistes (Barthe et al.)
  - de programmes cryptographiques (Vieira)

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

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`

Récapitulation

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

Récapitulation

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

Récapitulation

logique organisée en théories

une théorie T_1 peut être

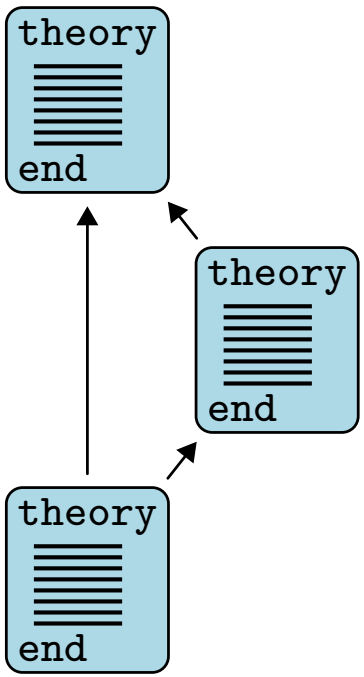
- utilisée (`use`) dans une théorie T_2
- clonée (`clone`) par une autre théorie T_2
  - les déclarations de T_1 sont copiées ou remplacées
  - les axiomes de T_1 restent des axiomes ou deviennent des lemmes/buts
  - les lemmes de T_1 deviennent des axiomes
  - les buts de T_1 sont ignorés

Sous le capot

une technologie pour parler à l'oreille des démonstrateurs

organisée autour de la notion de tâche = contexte + but

Le parcours d'une tâche

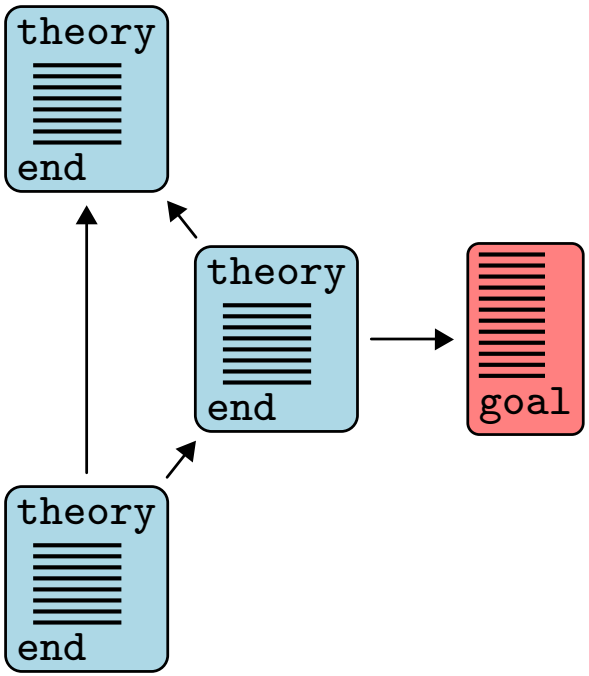


Alt-Ergo

Z3

Vampire

Le parcours d'une tâche

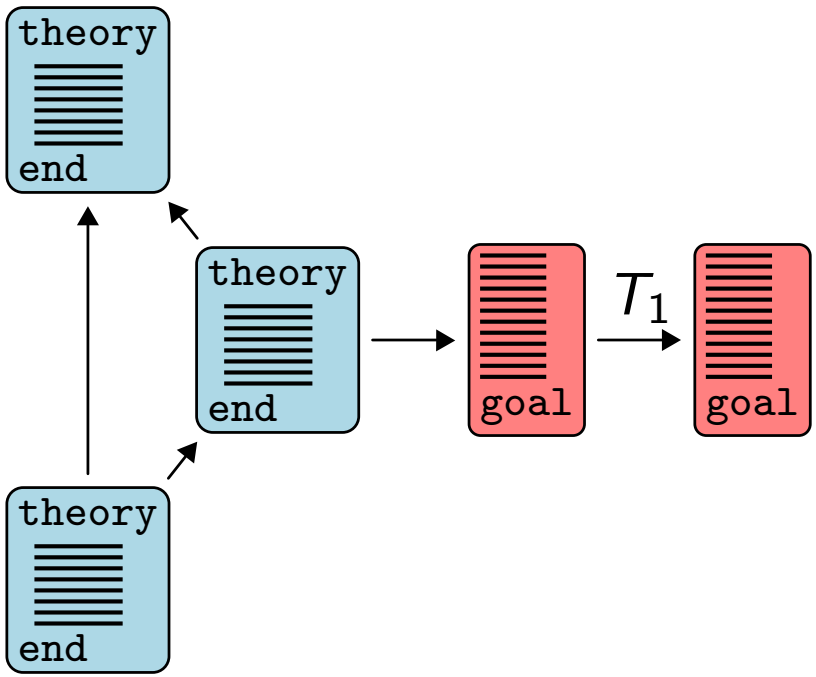


Alt-Ergo

Z3

Vampire

Le parcours d'une tâche

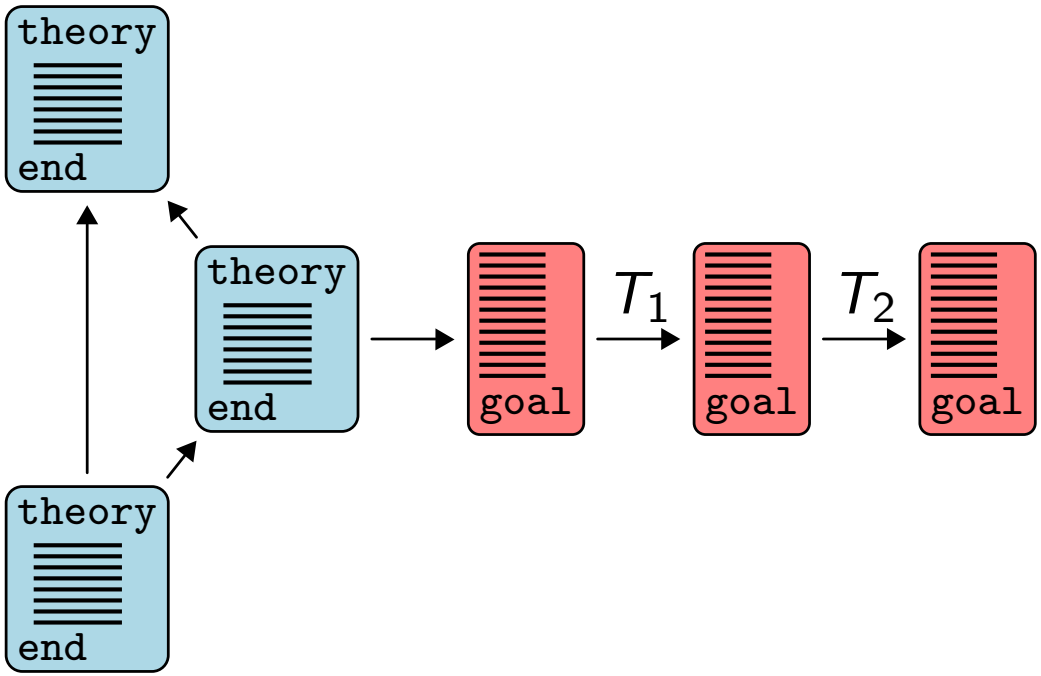


Alt-Ergo

Z3

Vampire

Le parcours d'une tâche

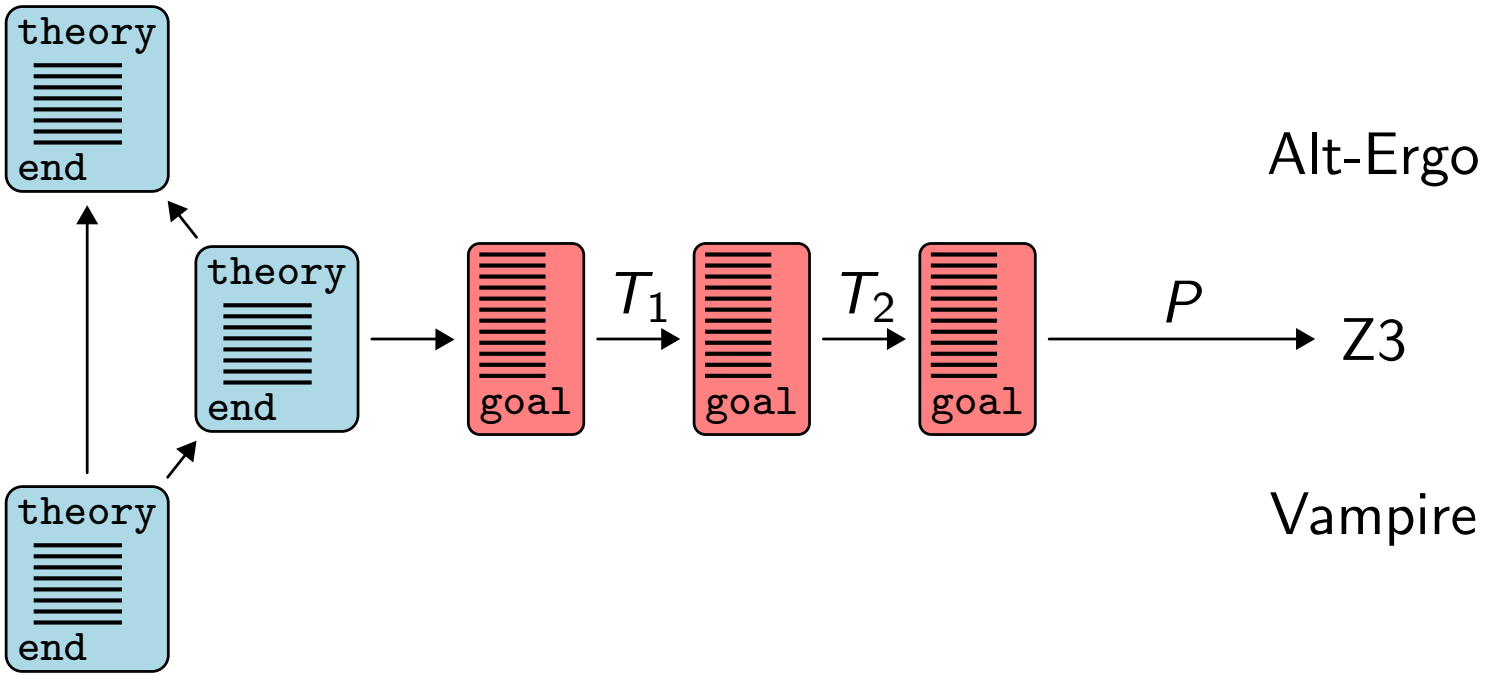


Alt-Ergo

Z3

Vampire

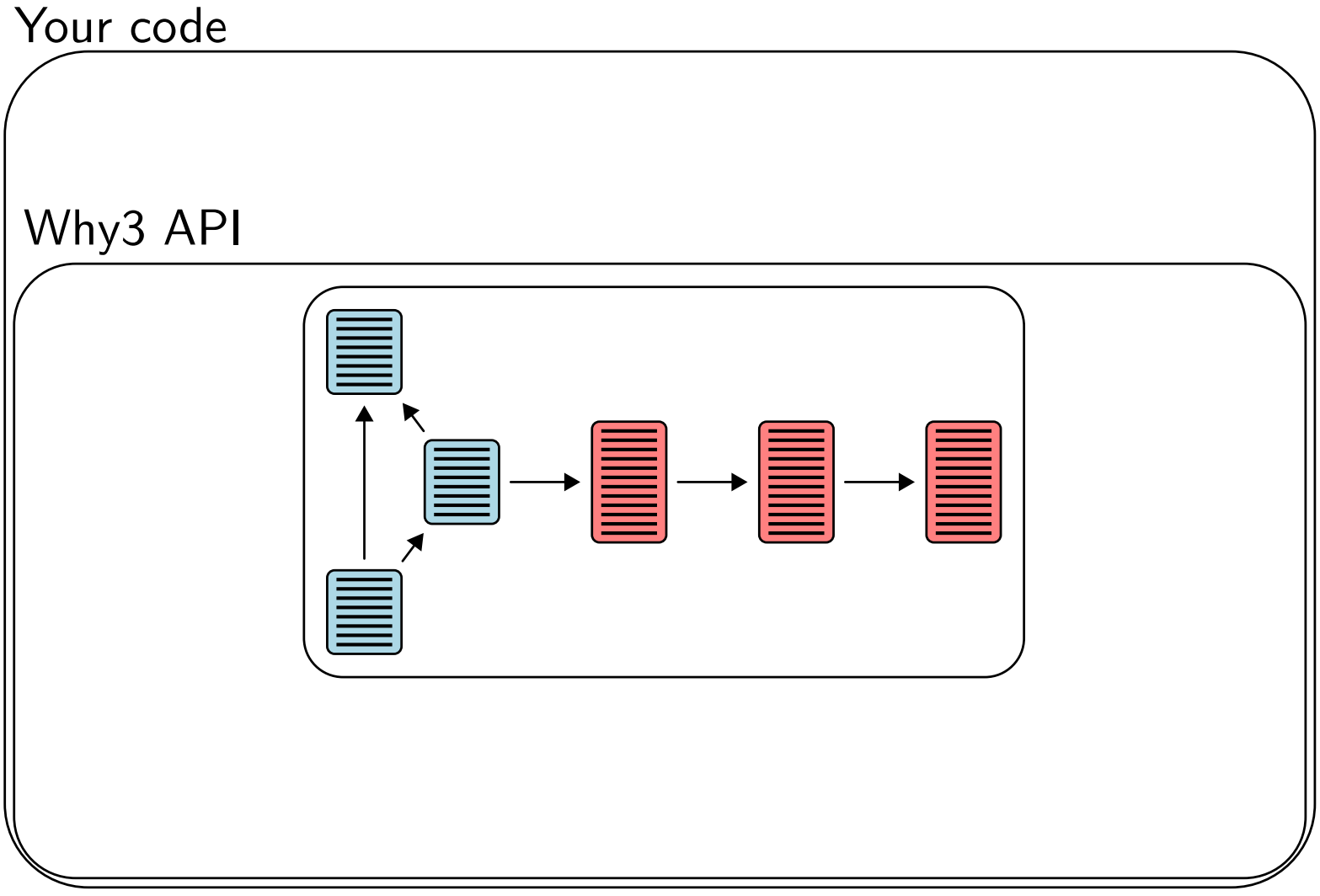
Le parcours d'une tâche

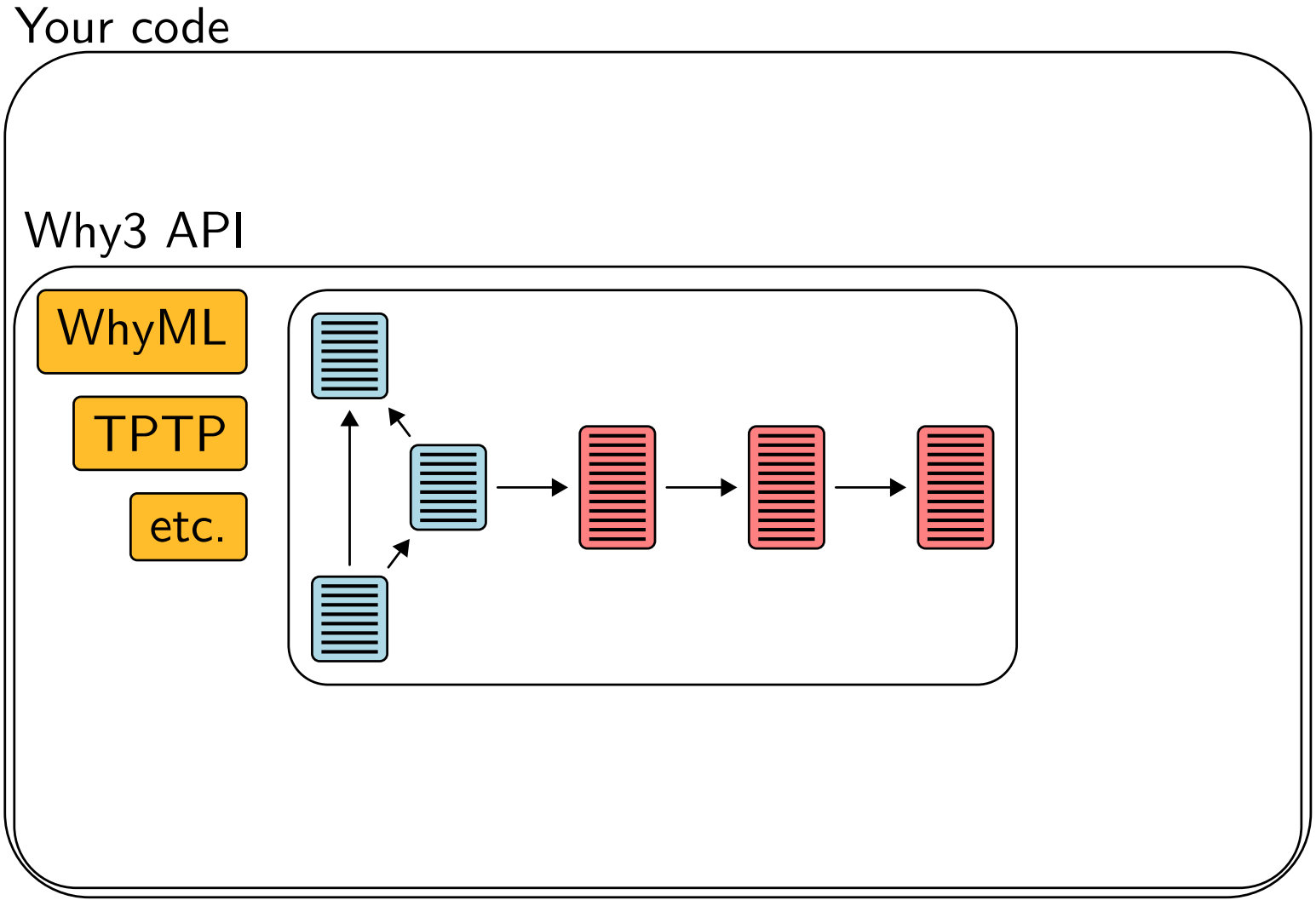


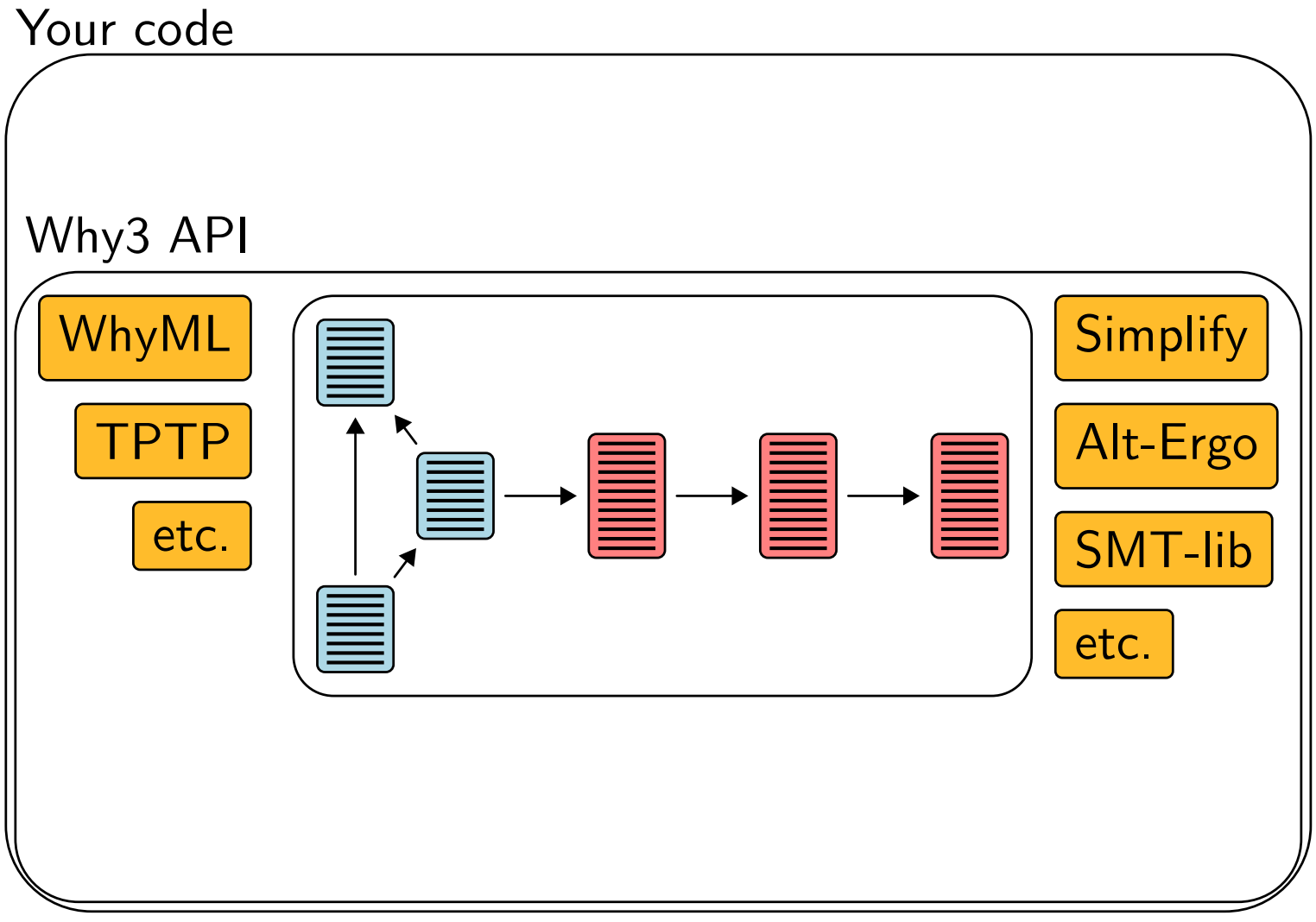
Pilote

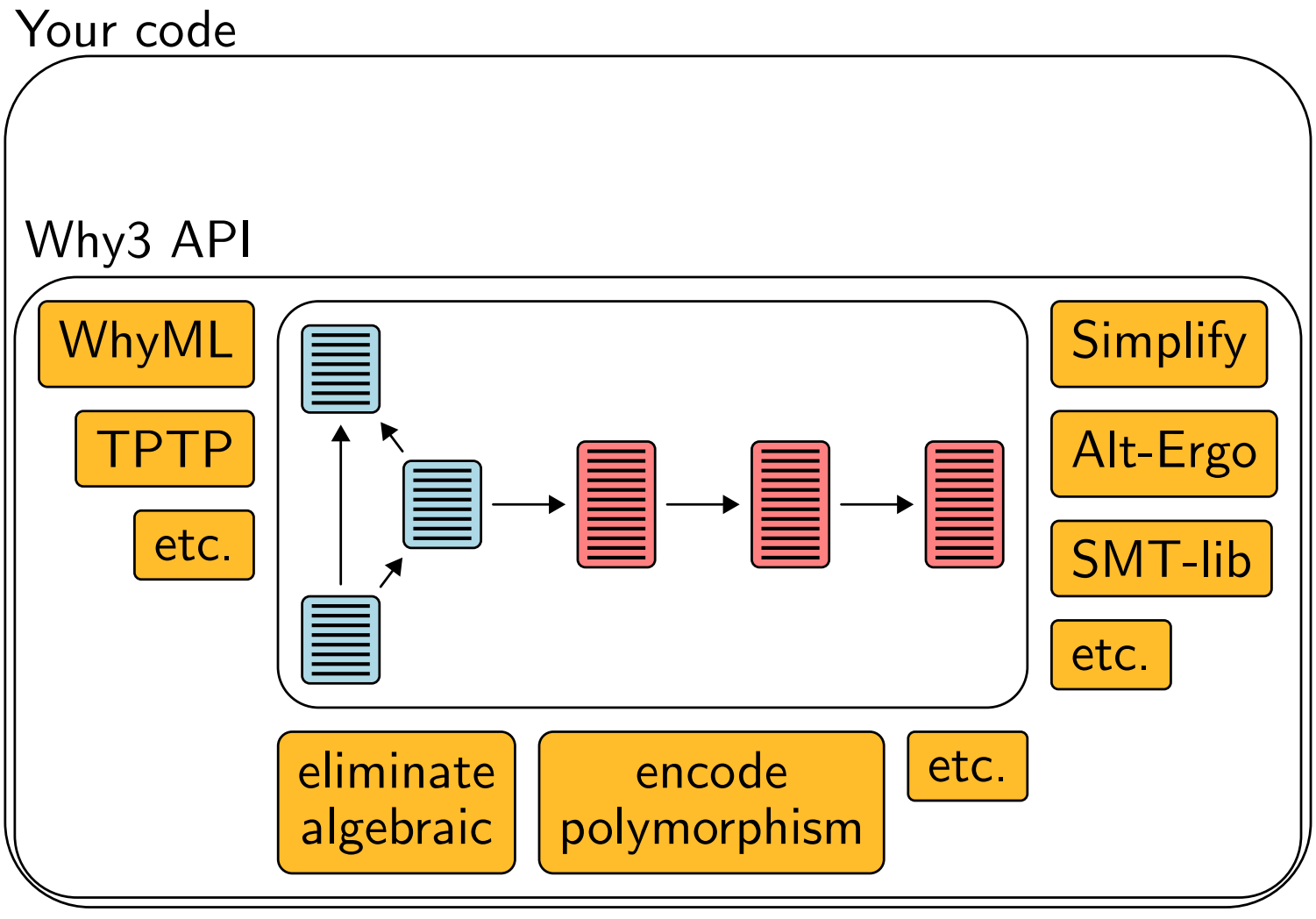
le parcours d'une tâche est piloté par un fichier

- 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 :

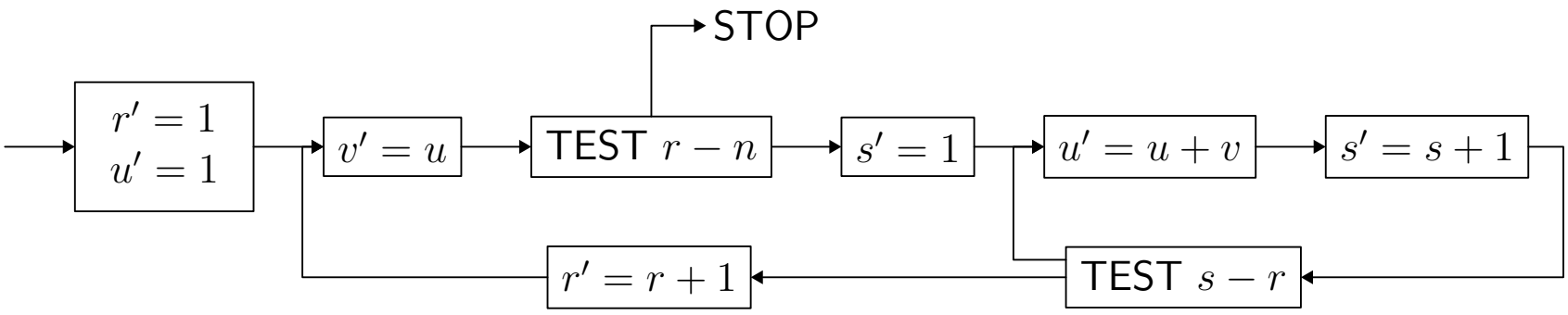
- *Why3 : Shepherd your herd of provers*. (Boogie 2011)

partie 2

Preuve de programmes

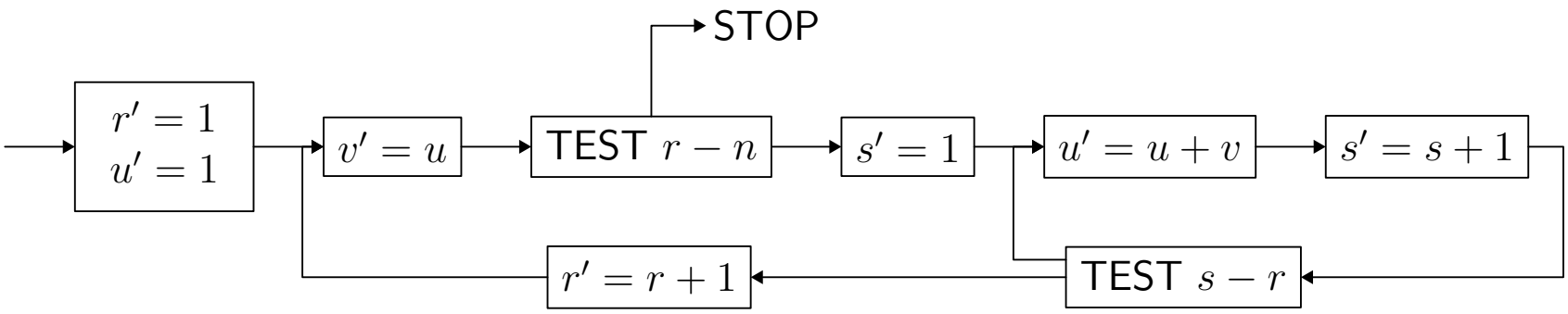
Un exemple historique

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



Un exemple historique

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)

Un autre exemple historique

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

démo (accès au code)

Un autre exemple historique

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

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)

Récapitulation

- 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

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

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

Structures de données

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

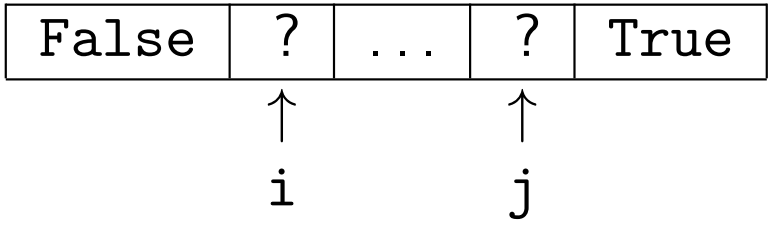
  ```
  a[e1] <- e2
  ```

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

Exemple

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
```



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

Exercice : le drapeau hollandais

```
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`

Remarque

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)

Principe

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`

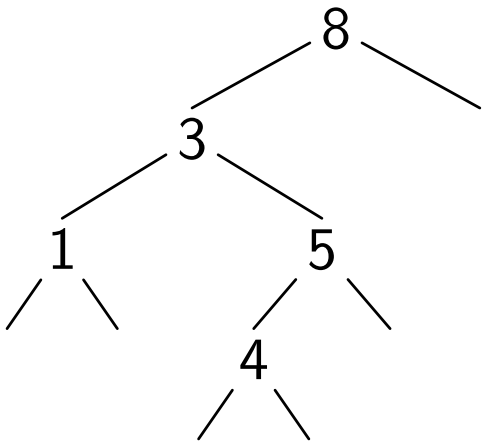
il en va de même des types algébriques en particulier

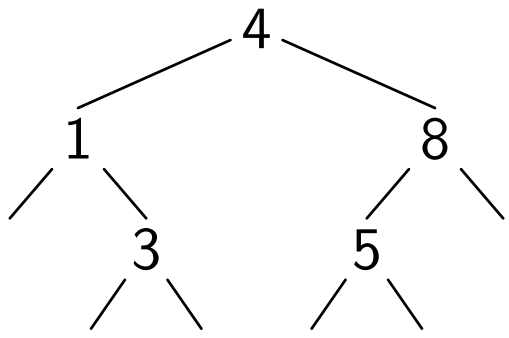
dans la bibliothèque, on trouve notamment

```
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 ?



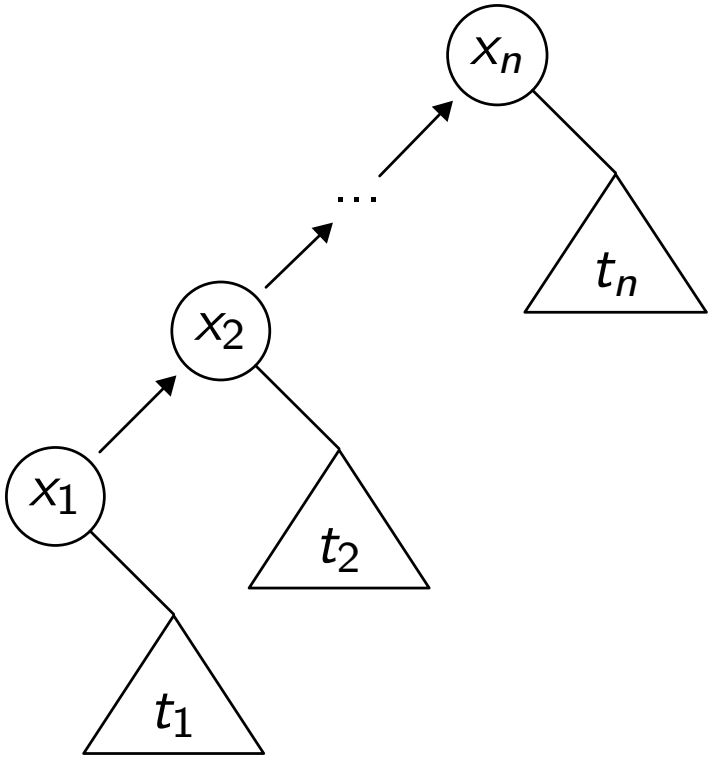


Exemple : *same fringe*

```
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 }
```

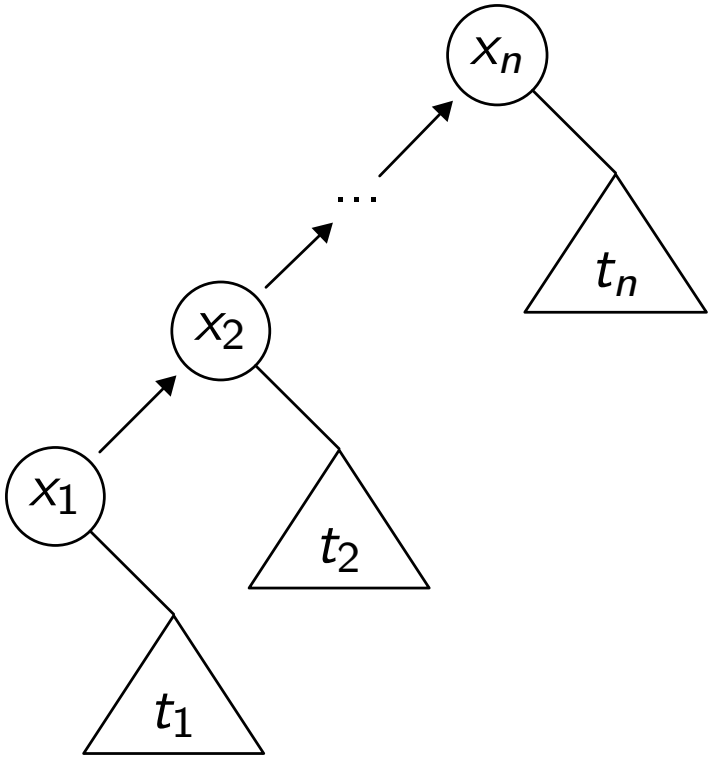
Exemple : *same fringe*

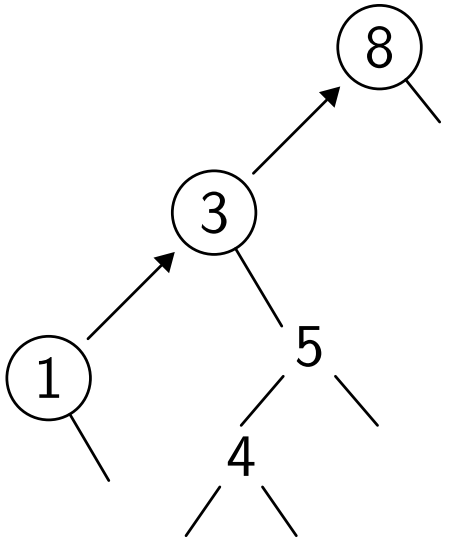
une solution : voir la branche gauche
comme une liste, de bas en haut

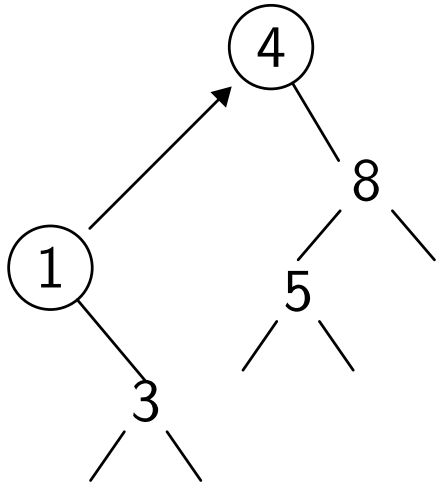


Exemple : *same fringe*

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

Retour sur les tableaux

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 |}
  ```

Retour sur les tableaux

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

Arithmétique 32 bits

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

Arithmétique 32 bits

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 }
```

Application

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

Modèle 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.

Conclusion

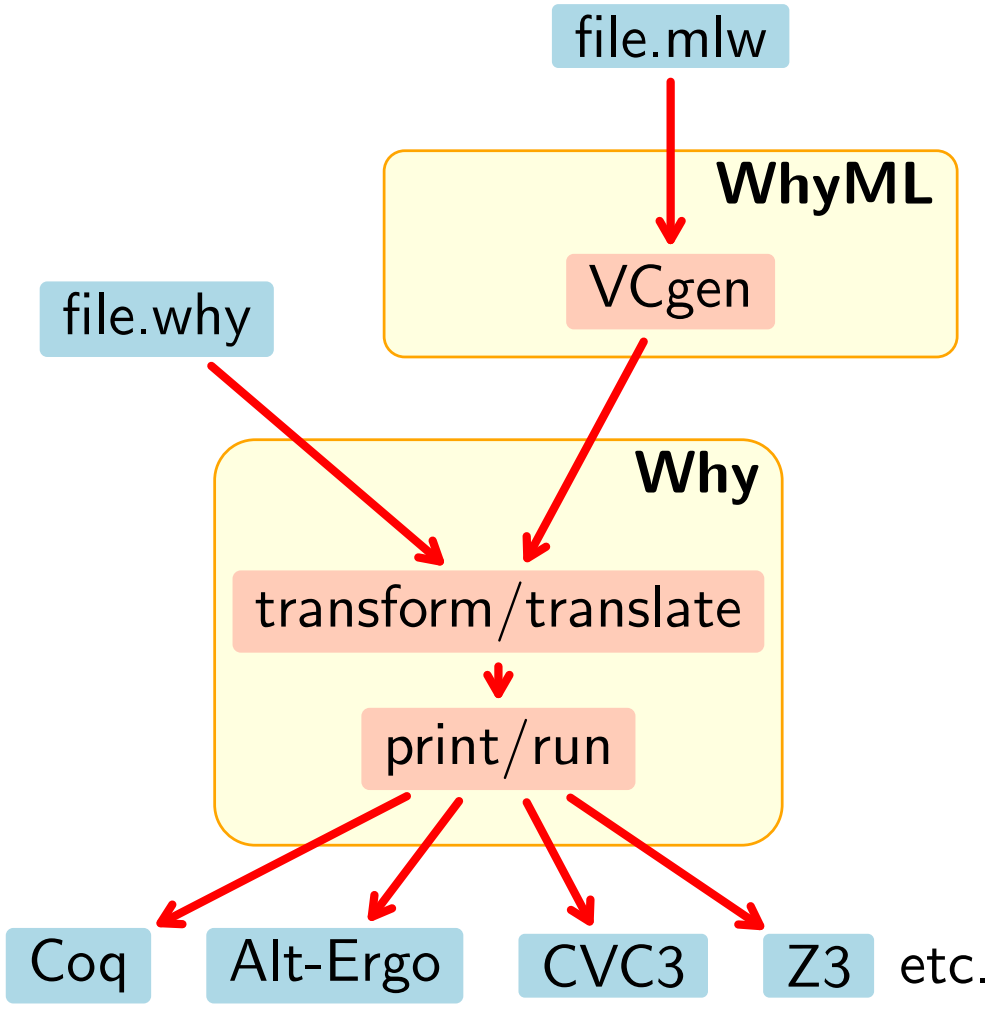
on a vu trois usages distincts de Why3

- langage logique
- preuve de programmes
- langage intermédiaire

Un langage logique

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

on encore uniquement
pour l'API OCaml
de sa logique



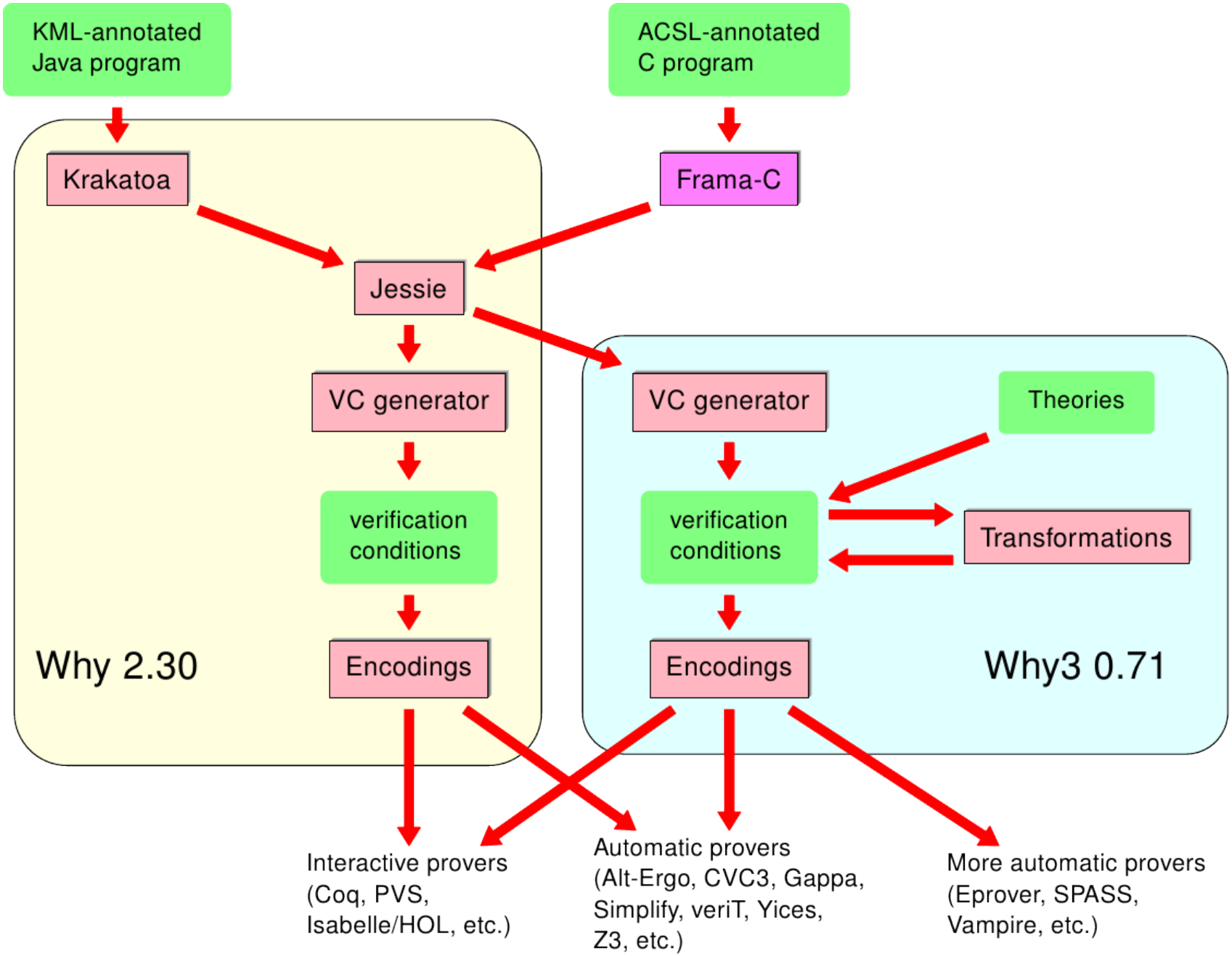
Pour la preuve de programmes ML

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