UNIVERSIDADE BEIRA INTERIOR

Verificação Deductiva de Programas em Why3

Simão Melo de Sousa

LISP LABORATORY OF COMPUTER SCIENCE, SYSTEMS AND PARALLELISM

(re)lease REliable And SEcure Computation Group

QuiVVer

Introdução

esta aula: um exercício sobre uma visão idealizada do que **poderia** ser a programação.

uma definição sintética de Programação:

> *Atividade que trata da conceção de métodos para resolver problemas usando computadores, e da sua respetiva implementação.*

dicotomia fértil entre a programação como uma atividade **técnica** e a programação como uma atividade **científica**

exploramos aqui propositadamente a segunda perspetiva

Programming is one of the branches of applied mathematics
(Edsger W. Dijkstra - How do we tell truths that might hurt? (1975))

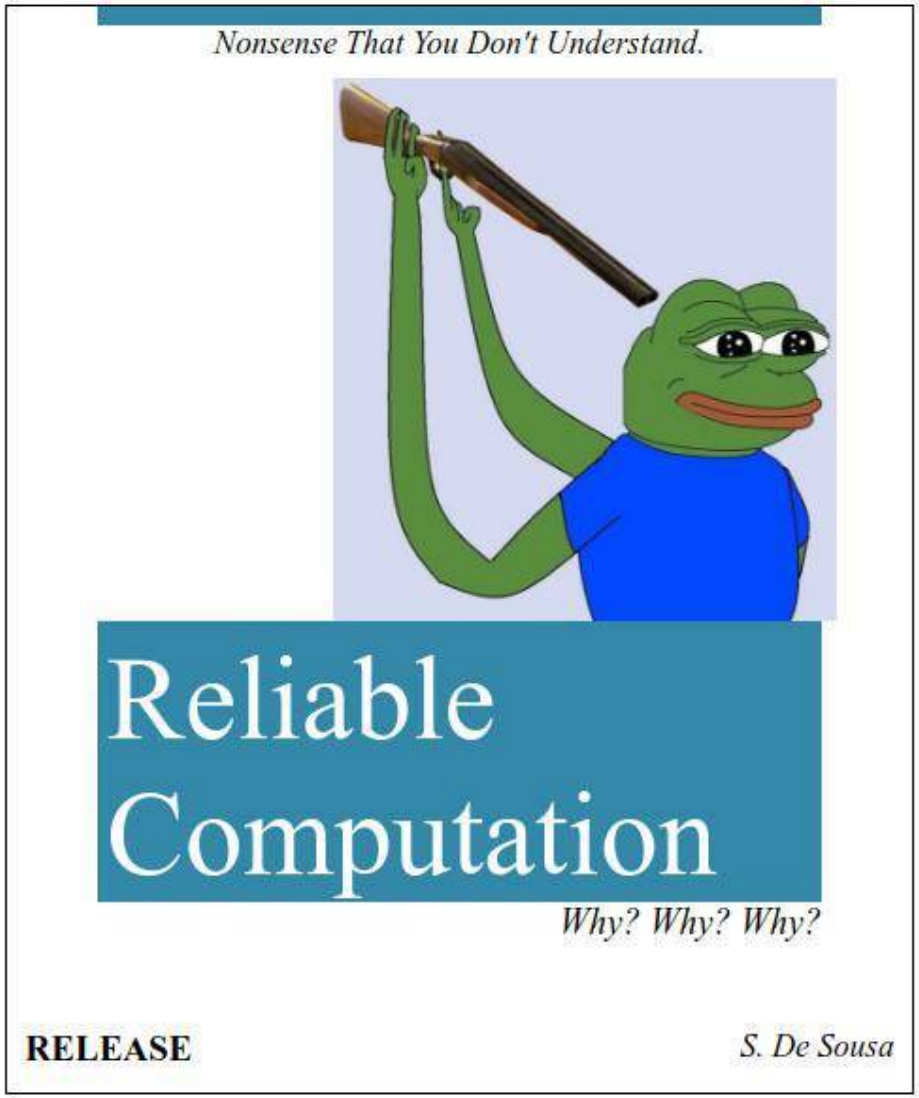
etimologia: **Informatics** vs. **Computer Science**
a informática como ciência da **informação** e ciência da **computação**.

competência/cultura transversal:

- o ensino da programação, dos algoritmos, não pode ser esvaziada e separada dos seus fundamentos, da sua cultura, do seu rigor
- as ciências de base, os fundamentos =
  - pontos de vista pertinentes,
  - ferramentas,
  - técnicas e
  - soluções sólidas

  aos problemas que o programador encontra na sua atividade diária

o dilema do estudante programador



(cortesia dos meus estudantes de Computação Fiável)

etimologia: **Informatics** vs. **Computer Science**
a informática como ciência da **informação** e ciência da **computação**.

mensagem aos alunos assustados:
- o exercício e a prática resultam
- dominar a essência duma solução programática torna-os **de-facto** melhores programadores
- Mas ... mas... mas... e as probabilidades, geometrias, análises, álgebras, lógicas?...

...enquanto se odeia, ainda se gosta - Alphonse Karr, sobre as relações longas e tumultuosas

o dilema do estudante programador

etimologia: **Informatics** vs. **Computer Science**
a informática como ciência da **informação** e ciência da **computação**.

perspetiva pessoal: é frequente o exercício da programação *fundamentada* resultar no "momento Eureka"

Bruce Schneier - Prefácio de "Secrets and Lies. Digital Security in a Networked World" em 2000

> *A colleague once told me that the world was full of bad security systems designed by people who read "Applied Cryptography".*
>
> *(...)*
>
> *Readers believed that cryptography was a kind of magic security dust that they could sprinkle over their software and make it secure.*

Sofia Rodrigues, Jornal Público - 30 de Setembro de 2004.

*(...) A partir da mesma base de dados do ministério que contém todos os docentes por colocar, a ATX Software criou um novo algoritmo, ou seja, uma solução informática, "**pensado na íntegra durante seis dias e baseado em princípios matemáticos muito sólidos** ", afirmou ontem o engenheiro informático e autor da solução, Luís Andrade, durante uma conferência de imprensa, em Lisboa.(...)*

Edsger W. Dijkstra - The Humble Programmer (1972)

If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.

Bem, mas.... **Donald Knuth**

... But software is hard!

- man in the middle (processo social interpretativo de especificação, implementação, validação, etc.)
- ditadura do *time-to-market*
- formação, educação
- software é uma peça complexa e evolutiva

Xavier Leroy na sua lição no Collège de France em 2008

Un peu de programmation éloigne de la logique mathématique; beaucoup de programmation y ramène.

Programar $\equiv$ Demonstrar

três conceitos fundamentais:

- **O-QUÊ**. O que pretendemos resolver?
- **COMO**. Tendo o computador em mente como maquina produtora do resultado, como operacionalmente, é construida a resolução para obter a solução?
- **PORQUÊ**. O que permite afirmar que o algoritmo/código (**COMO**) faz precisamente o que é pretendido (**O-QUÊ**)?

especificação formal, refinamento, cálculo de programas

$$\text{O-QUÊ} \xrightarrow{\qquad\text{PORQUÊ}\qquad} \text{COMO}$$

teoria de tipos, isomorfismo de Curry Howard, prova assistida

$$\text{O-QUÊ} = \text{COMO} = \text{PORQUÊ}$$

Lógica de Hoare, Cálculo de pré-condições mais fracas,
Design-by-contract $\rightarrow$ **verificação deductiva de programas**

$$\frac{\text{O-QUÊ}}{\text{COMO}}\ \text{PORQUÊ}$$

Jean-Christophe Filliâtre - arte, ciência e engenho

"Deductive program verification is the **art** of turning the correctness of a program into a mathematical statement and then the **science** of proving it", ... **using a computer**.

verificação dedutiva, numa imagem

o que é um programa?

o conjunto de:

- **o que** calculamos
- **como** o calcular
- a razão **porque** funciona

o código é somente um dos aspectos!

a documentação, o caderno de encargo, comentários, papers, etc. são componentes igualmente

a nossa proposta, com alguma ousadia, é ver um programa como o conjunto formado pelo *Quê*, *Como* e *Porquê*

algoritmo vs. código - *o mapa e o território*

uma pequena digressão: podemos entender o algoritmo como parte do "*como*", como o código?

em cartografia, na física, nas artes, em sociologia etc. o debate é aceso

será o mundo dos jornais televisivos o mundo real?
será o mundo conceptualizado pela física o mundo real?

todos nós conhecemos a tela do Magritte: "Ceci n'est pas une pipe"

Jorge Luis Borges em "On Exactitude in Science" (1946) conta como os cartógrafos do Império realizaram um mapa 1-1 do território e como esse se tornou inutilizável logo inútil e em consequência abandonado... tornando-se aos poucos habitado e logo... território

mas o mundo da programação, da informática tem a sua especificidade: as leis fundamentais do mundo onde evoluí o programa são moldadas à medida!

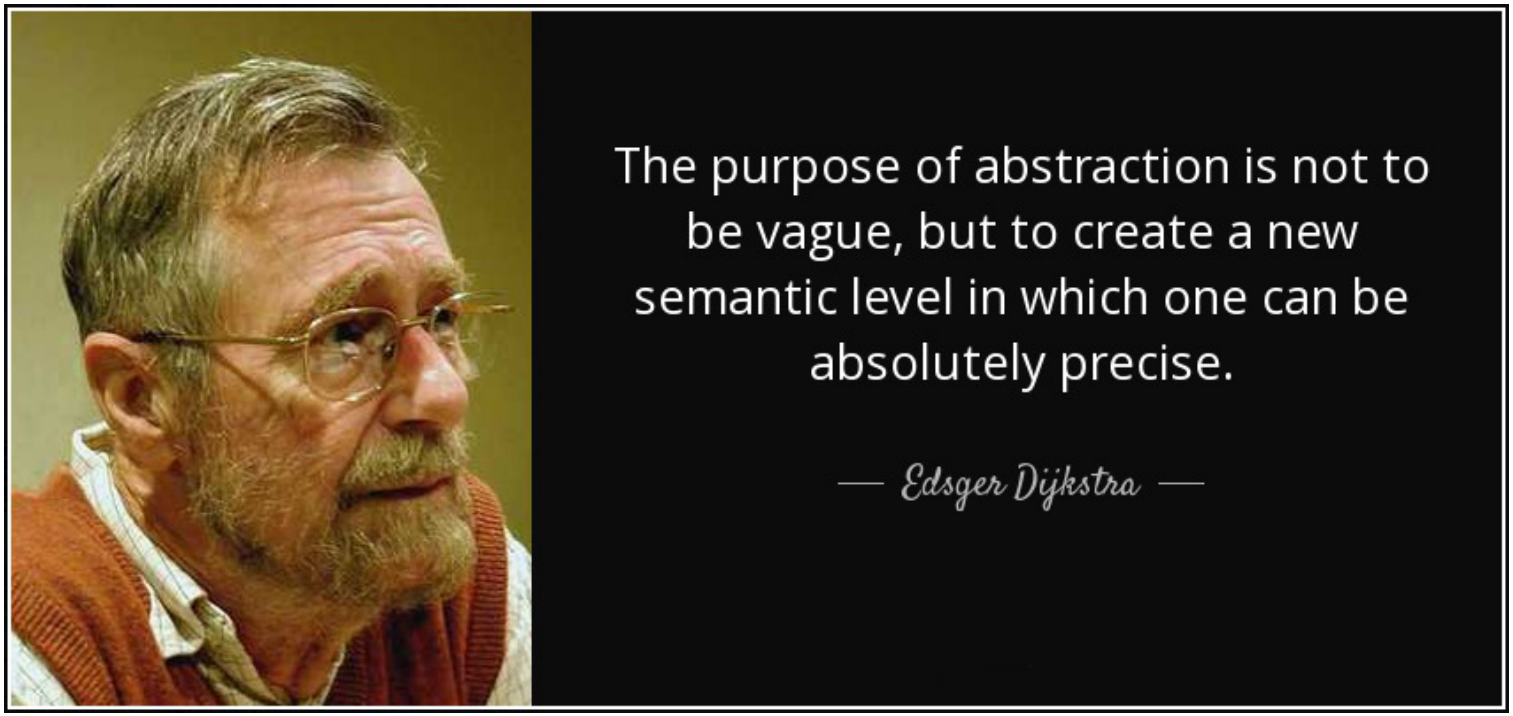
Mais, o próprio código pode ser considerado como uma abstração, logo o *mapa* de outro *território*...(e.g. uma base de dados de automóveis é uma abstração de uma parque automóvel)

o mapa precede o território

J. Baudillard, em "Simulacros e simulação" (1991) "O território já não precede o mapa, nem lhe sobrevive. É agora o mapa que precede o território – precessão dos simulacros – é ele que engendra os territórios cujos fragmentos apodrecem sobre a extensão do mapa".

Michel Houellebecq em "o mapa e o território" (2010): Período Michelin - a personagem expõe como obra de arte fotografias de mapas Michelin com a legenda "o mapa tem mais interesse do que o território"

pragmaticamente: em programação, a exploração tanto do mapa como do território tem a sua utilidade, é preciso perceber e conhecer bem as diferenças para melhor conjugá-las



veremos em particular como o Why3 ataca este problema

Verificação deductiva em Why3

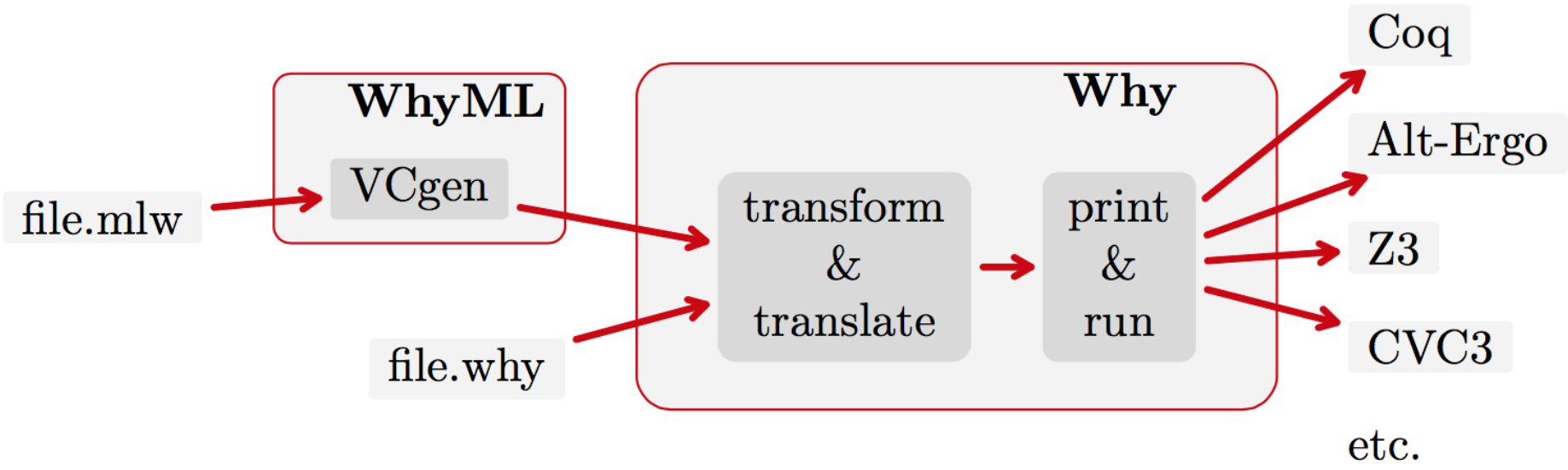
parte significativa do material a seguir apresentado foi retirado de duas formações Why3 ministradas pelo **Jean-Christophe Filliâtre** (link)

- Deductive Program Verification with Why3 - A Tutorial, Jean-Christophe Filliâtre, Lecture at EJCP 2015 (link)
- An Introduction to Deductive Program Verification, Jean-Christophe Filliâtre, Lecture at the Sixth Summer School on Formal Techniques (link)

site: Why3 - Where Programs Meet Provers (link)

interface web why3: try why3 online (link)

Why3, numa imagem

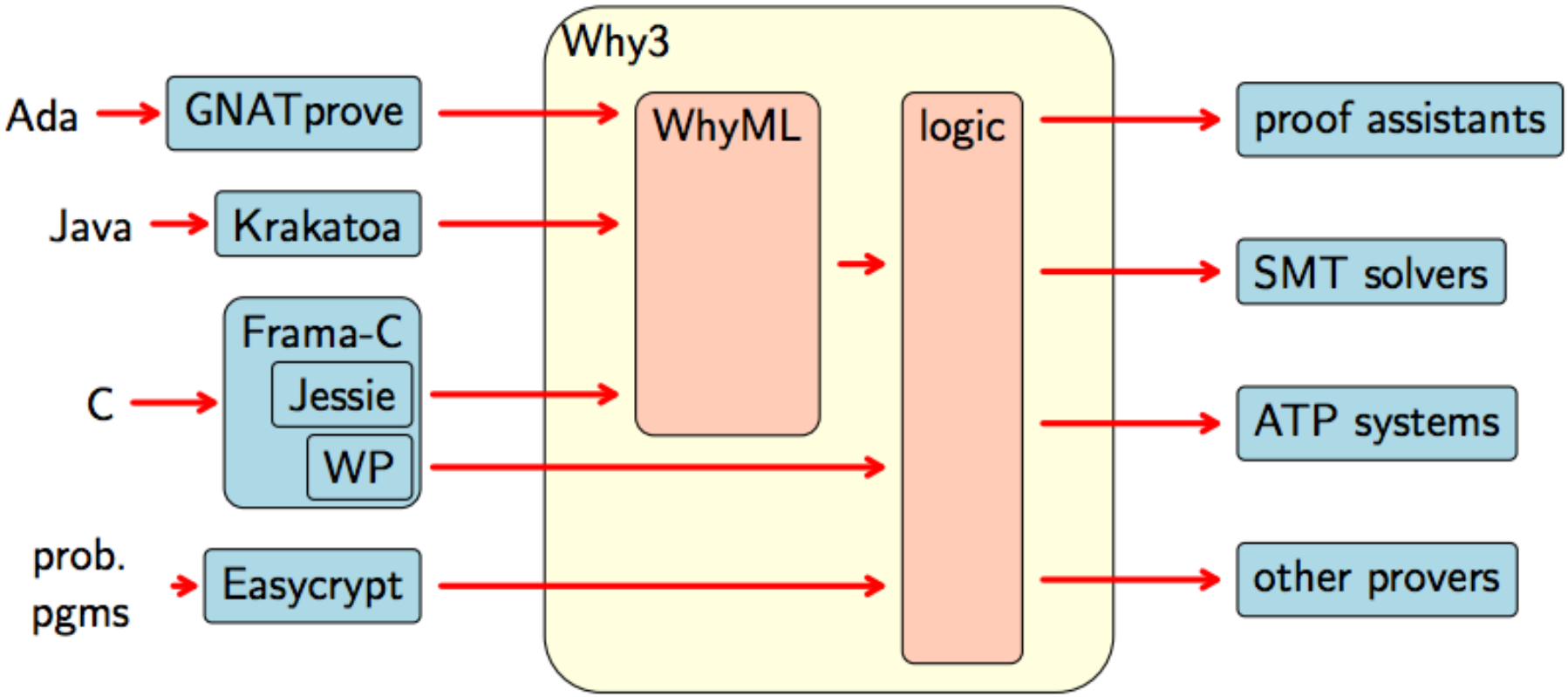


os cenários clássicos de utilização de Why3 são os seguintes:

- como uma **linguagem lógica** associada aos mecanismos de prova associados
  (como front-end conveniente para muitos provers)
- como uma **linguagem de programação** com mecanismos de prova de algoritmos
- como uma **linguagem intermédia** para a verificação de programas escritos em diversas linguagens de programação (C, Ada, Java, etc.)

  dispõe de uma API rica que permite embeber as suas funcionalidades em aplicações externas

Why3 como ferramenta de suporte



Why3 como uma interface à lógica da programação

a lógica do Why3: essencialmente uma lógica de primeira ordem polimórfica com:

- tipos de dados algébricos (mutuamente) recursivos
- símbolos de funções/predicados (mutuamente) recursivos
- predicados (mutuamente) (co-)indutivos
- construções let-in. match-with, if-then-else

estaremos confinados na primeira ordem? há formas controladas de ter ordem superior nos termos (**expressões lambda**)

declarações na lógica Why3

tipos

- abstractos: `type t`
- alias: `type t = list int`
- algébricos `type list 'a = Nil | Cons 'a (list 'a)`

funções, predicados

- não interpretados: `function f int : int`
- definidos: `predicate non_empty (l: list 'a) = l <> Nil`

predicados indutivos: `inductive trans t t = ...`

axiomas, objectivos de prova, lemas:

- `goal G: forall x:int . x>= 0 -> x*x > 0`

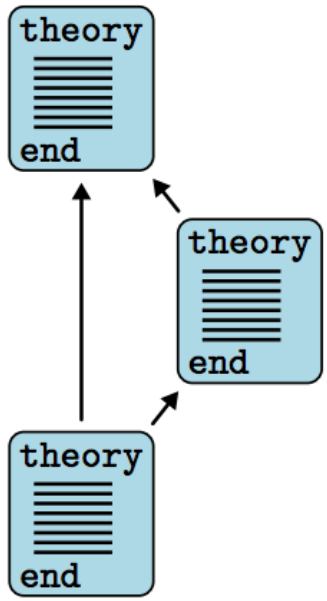
as declarações lógicas podem (e devem!) ser organizadas em **teorias**
uma teoria T_1 pode ser

usada (**use**) numa teoria T_2

- os símbolos de T_1 são ***partilhados***
- os axiomas de T_1 permanecem axiomas
- os lemas de T_1 tornam-se axiomas
- os *goals* de T_1 são ignorados

clonada (**clone**) numa outra teoria T_3

- as declarações de T_1 são **copiados** ou **substituídos**
- os axiomas de T_1permanecem axiomas ou tornam-se lemma/*goals*
- lemas de T_1 tornam-se axiomas
- os *goals* de T_1 são ignorados



Why3 como uma linguagem lógica

```
theory EndoRelation
  type t
  predicate rel t t
end

theory Reflexive
  clone export EndoRelation
  axiom Refl : ∀ x:t. rel x x
end

theory Irreflexive
  clone export EndoRelation
  axiom Strict : ∀ x:t. not rel x x
end

theory Transitive
  clone export EndoRelation
  axiom Trans : ∀ x y z:t.
      rel x y → rel y z → rel x z
end
```

```
theory List
  type list α = Nil | Cons α (list α)
end

theory Length
  use import int.Int
  use import List

  function length (l: list α) : int =
    match l with
    | Nil       → 0
    | Cons _ r → 1 + length r
    end

  lemma Length_nonnegative:
        ∀ l: list α. length l ≥ 0

  lemma Length_nil: ∀ l: list α.
        length l = 0 ↔ l = Nil
end
```

Why3 como uma linguagem lógica

```
theory Sorted
  use import List
  type t
  predicate le t t

  clone relations.Transitive with type t = t, predicate rel = le

  inductive sorted (l: list t) =
    | Sorted_Nil:    sorted Nil
    | Sorted_One:    ∀ x: t. sorted (Cons x Nil)
    | Sorted_Two:
        ∀ x y: t, l: list t.
        le x y → sorted (Cons y l) → sorted (Cons x (Cons y l))

  use import Mem
  lemma sorted_mem:
    ∀ x: t, l: list t.
    (∀ y: t. mem y l → le x y) ∧ sorted l ↔ sorted (Cons x l)
 (...)
end
```

a plataforma Why3 propõe uma **linguagem de programação** (*à la* ml) **WhyML** e um VCGen que permite processar tanto conteúdo computacional como especificação de comportamento ao nível da camada lógica de Why3

WhyML:

- sintaxe à la ML
- polimórfica
- *pattern-matching*
- excepções
- estrutura de dados mutáveis

terminação

Why3 **requer** que todas as funções *lógicas* terminem

trata-se de um dos critérios para assegurar a consistência lógica

e no que diz respeito a funções/programas?

a necessidade da terminação não se mantém nos programas

há no Why3 uma diferença entre a camada lógica e a camada programática, ao contrário por exemplo, do que acontece no sistema Coq

assim podemos provar

a **correção parcial**:
se as precondições são verificadas
e se o programa termina
então as pós-condições são garantidas

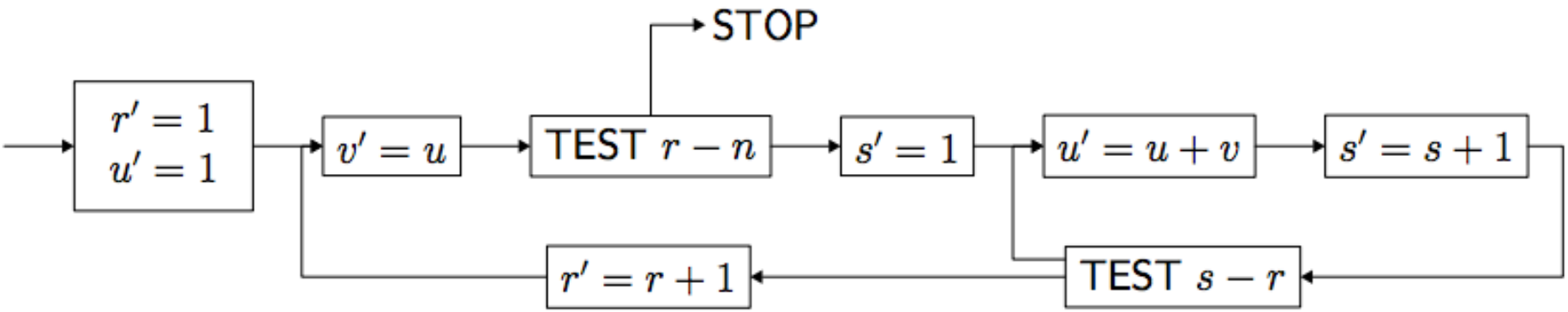
ou

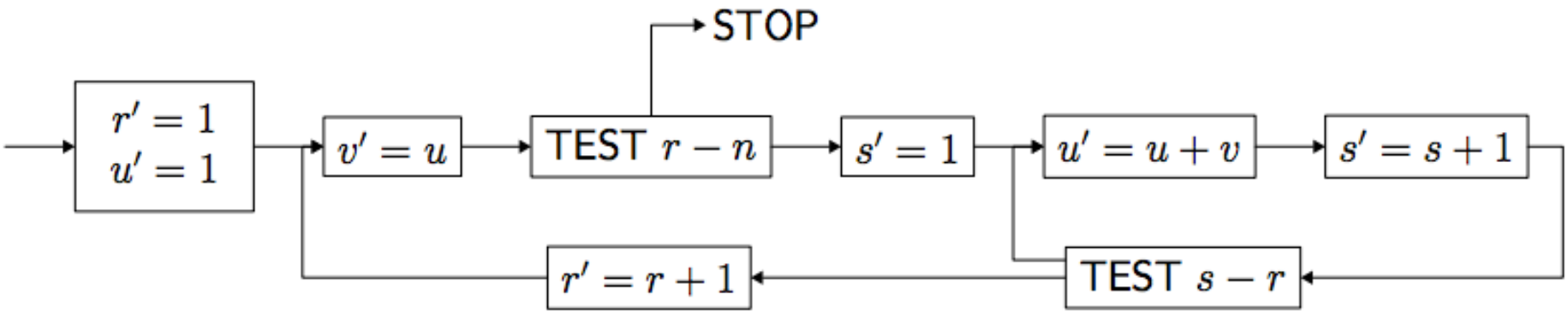
a **correção total**:
se as precondições são verificadas
então o programa termina
e as pós-condições são garantidas

mas convenhamos que provar a correção parcial de um programa que não termina é um exercício estéril...

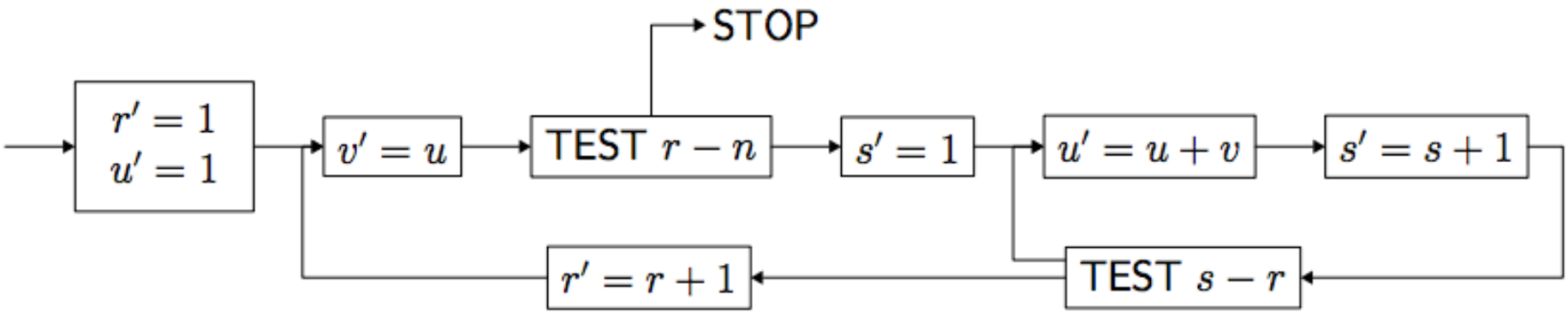
A. M. Turing. **Checking a Large Routine**. 1949.

a factorial, só com somas e antes das primeiras linguagens de programação

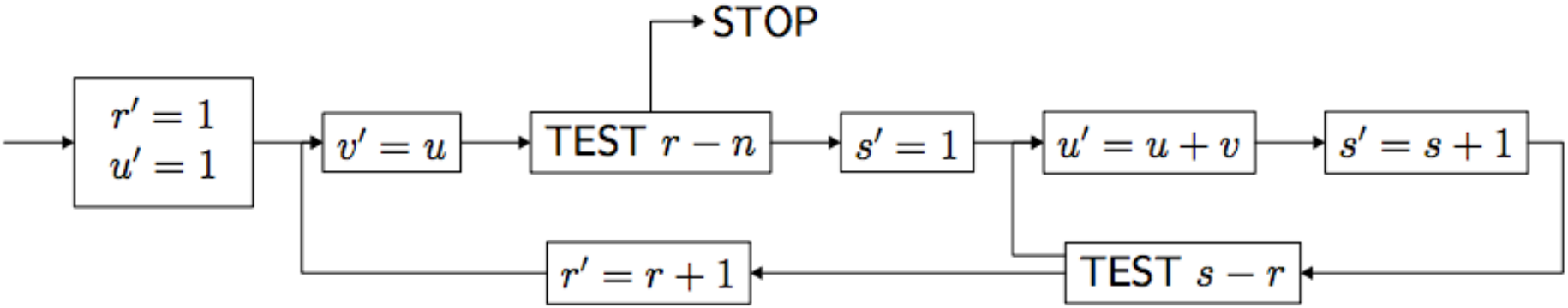




$u \leftarrow 1$
for $r = 0$ **to** $n - 1$ **do**
  $v \leftarrow u$
  for $s = 1$ **to** r **do**
    $u \leftarrow u + v$



precondition $\{n \geq 0\}$
$u \leftarrow 1$
`for` $r = 0$ `to` $n - 1$ `do`
  $v \leftarrow u$
  `for` $s = 1$ `to` r `do`
    $u \leftarrow u + v$
postcondition $\{u = fact(n)\}$



precondition $\{n \geq 0\}$
$u \leftarrow 1$
`for` $r = 0$ `to` $n - 1$ `do` invariant $\{u = fact(r)\}$
  $v \leftarrow u$
  `for` $s = 1$ `to` r `do` invariant $\{u = s \times fact(r)\}$
    $u \leftarrow u + v$
postcondition $\{u = fact(n)\}$

```
module CheckingALargeRoutine

  use import int.Int
  use import int.Fact
  use import ref.Ref

  let routine (n: int) requires { n ≥ 0 } ensures { result = fact n } =
    let u = ref 1 in
    for r = 0 to n-1 do invariant { !u = fact r }
      let v = !u in
      for s = 1 to r do invariant { !u = s * fact r }
        u := !u + v
      done
    done;
    !u

end
```

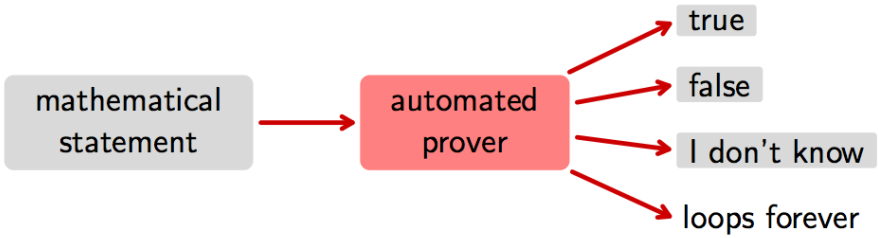
```
function fact(int) : int
axiom fact0: fact(0) = 1
axiom factn: ∀ n:int. n ≥ 1 → fact(n) = n * fact(n-1)
```

```
goal vc: ∀ n:int. n ≥ 0 →
  (0 > n - 1 → 1 = fact(n)) ∧
  (0 ≤ n - 1 →
    1 = fact(0) ∧
    (∀ u:int.
      (∀ r:int. 0 ≤ r ∧ r ≤ n - 1 → u = fact(r) →
        (1 > r → u = fact(r + 1)) ∧
        (1 ≤ r →
          u = 1 * fact(r) ∧
          (∀ u1:int.
            (∀ s:int. 1 ≤ s ∧ s ≤ r → u1 = s * fact(r) →
              (∀ u2:int.
                 u2 = u1 + u → u2 = (s + 1) * fact(r))) ∧
              (u1 = (r + 1) * fact(r) → u1 = fact(r + 1))))) ∧
      (u = fact((n - 1) + 1) → u = fact(n))))
```

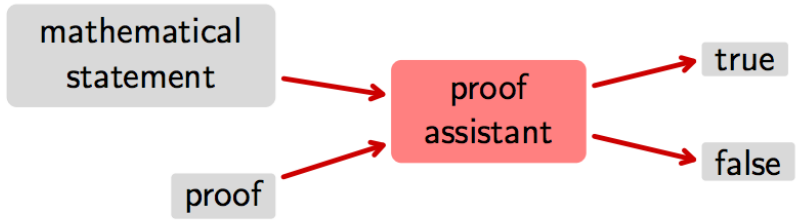
o que fazer com esta condição de verificação?

prova manual? possível (os pioneiros assim faziam...) mas é em geral um esforço inglório, longo e sujeito a erros

prova computacionalmente sistematizada? o nosso **focus** com a ajuda de Why3

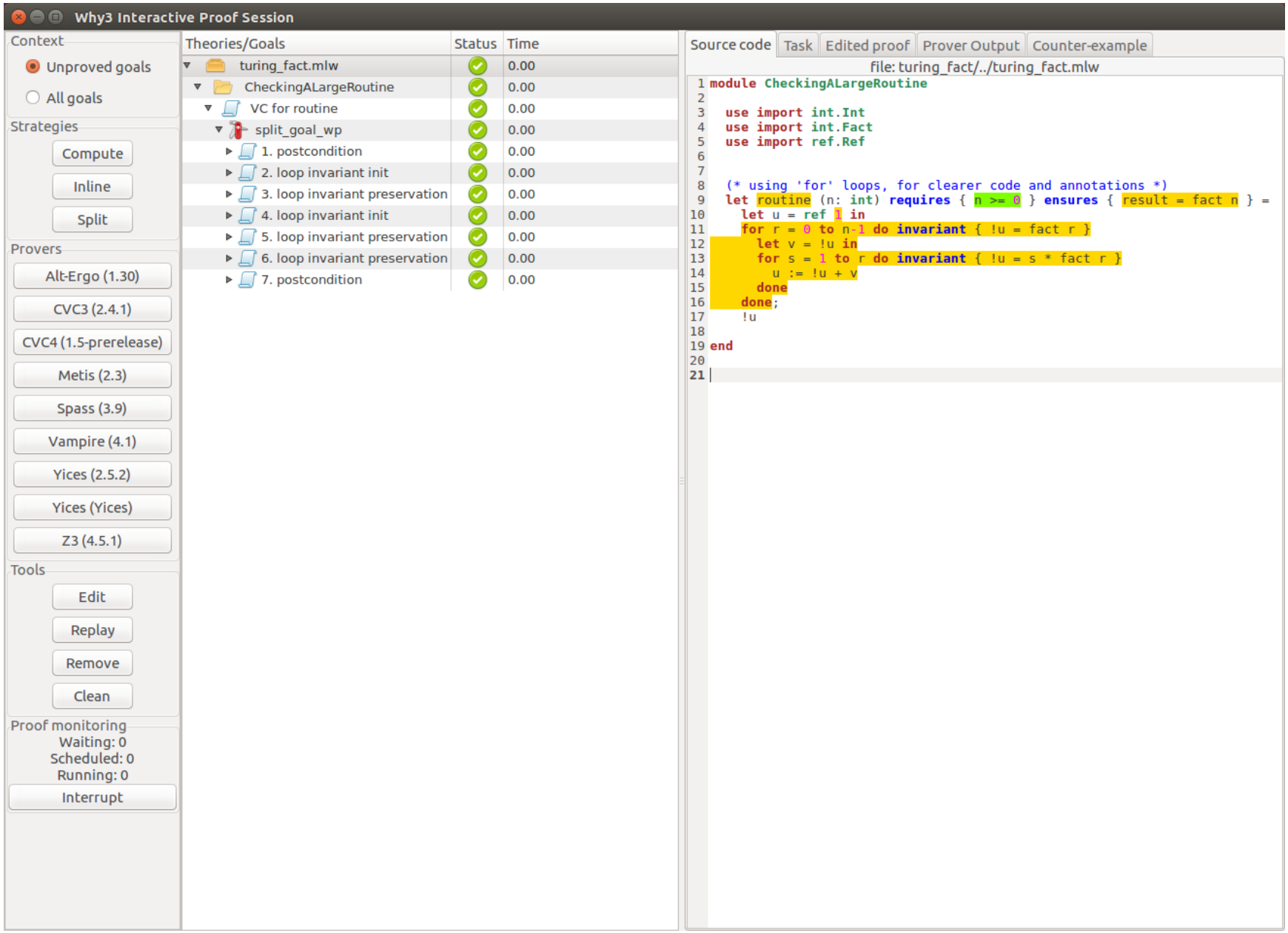


ex: Z3, Alt-Ergo, CVC4, Vampire, SPASS, etc.



ex: Coq, Isabelle, PVS, Hol Light, etc.

a factorial de Turing



$$VC(\texttt{let f x requires \{ P \} ensures \{ Q \} = e}) = \\ \forall \mathbf{x}.\, P \Rightarrow WP(e, Q)$$

onde $WP(e, Q)$ é a **pre-condição mais fraca** para o programa e que satisfaz a pós-condição Q

$$WP(\texttt{t}, Q) =$$
$$\quad Q[\texttt{result} \leftarrow \texttt{t}]$$

$$WP(\texttt{x} := \texttt{t}, Q) =$$
$$\quad Q[\texttt{x} \leftarrow \texttt{t}]$$

$$WP(\texttt{e1; e2}, Q) =$$
$$\quad WP(\texttt{e1}, WP(\texttt{e2}, Q))$$

$$WP(\texttt{if b then e1 else e2}, Q) =$$
$$\quad \texttt{if b then } WP(\texttt{e1}, Q) \texttt{ else } WP(\texttt{e2}, Q)$$

$$WP(\texttt{while b do invariant \{ I \} e done}, Q) =$$
$$\quad \texttt{I} \wedge \forall x_1, \ldots, x_n. I \Rightarrow \texttt{if b then } WP(\texttt{e}, I) \texttt{ else } Q$$

em vez da aplicação da substituição no cálculo das VCs, introduz-se novas variáveis

$$\begin{array}{ll} \texttt{x} := \texttt{!x} + 1; & \forall x_1.x1 = x_0 + 1 \implies \\ \texttt{x} := \texttt{!x} * \texttt{!y}; & \forall x_2.x2 = x_1 \times y \implies \\ \ldots & \ldots \end{array}$$

a geração de VCs deve tomar conta de muitos mais construcções sintácticas, como a aplicação de funções, *pattern-matching*, ciclos *for*, etc.

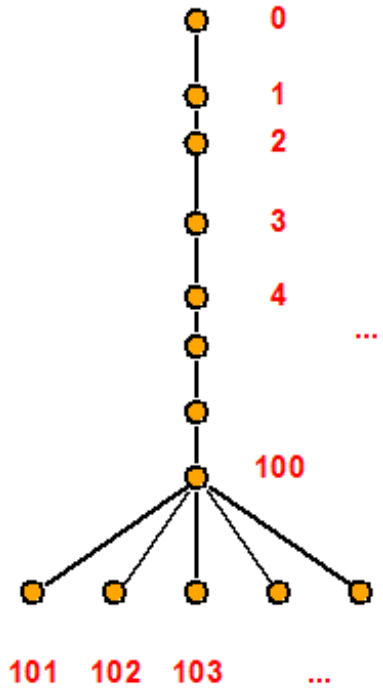
destaca-se em particular a existência em Why3 das pos-condições que representam retorno de funções em condições excepcionais

```
exception Invalid

let f (...) : t
requires {...P...}
ensures {...Q...}
raise {Invalid → ... pos-condição excepcional ...}    = ...definição de f...
```

a **função 91**, definida por Mc Carthy (com o Amir Pnueli e Zohar Manna), em 1970 (link wikipedia)

$$f(n) = \begin{cases} n - 10 & \text{se } n > 100 \\ f(f(n+11)) & \text{senão} \end{cases}$$

```
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
```



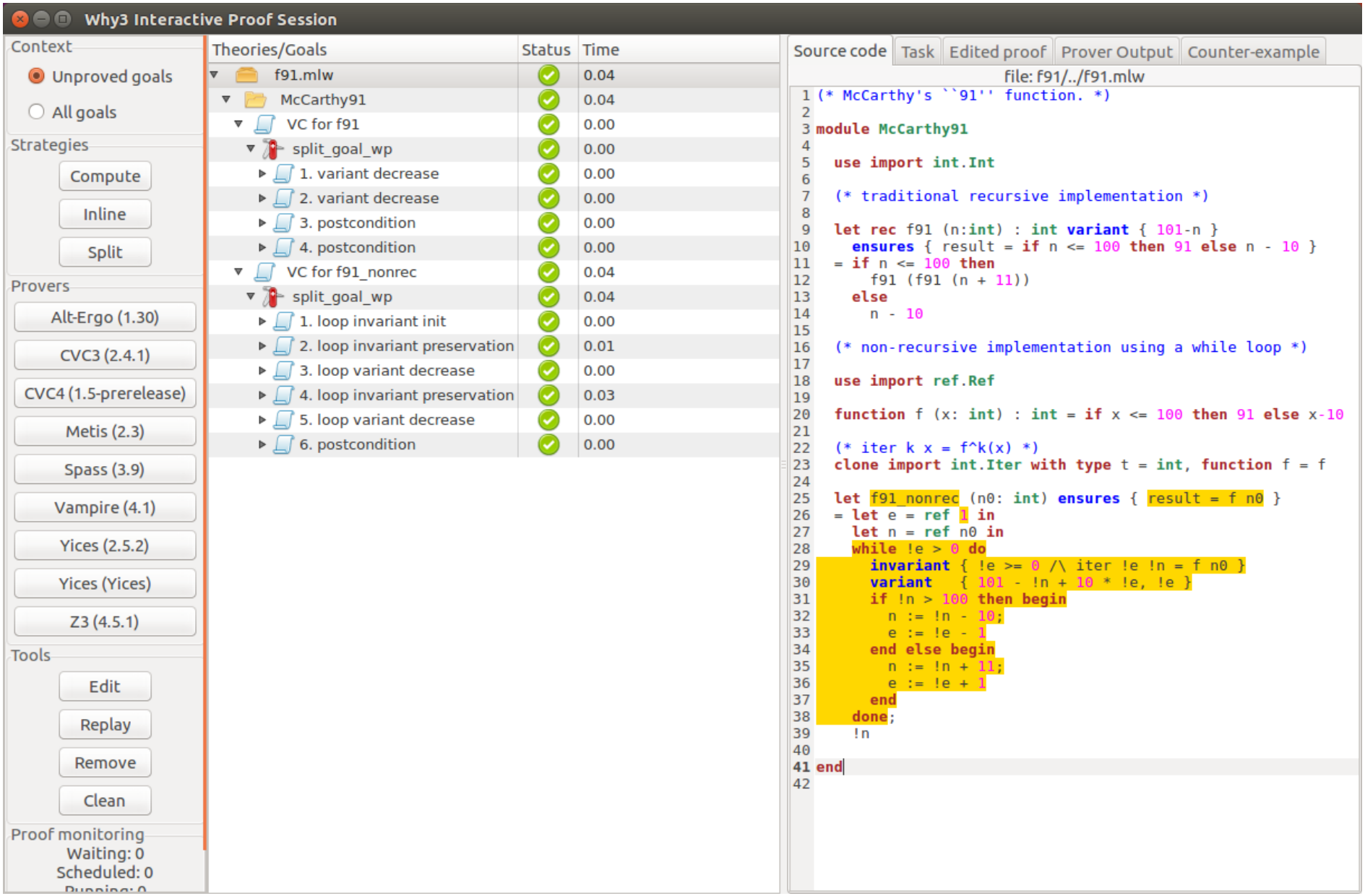
a função 91 (versão recursiva)

```
module McCarthy91

 use import int.Int

 function f (x: int) : int =
     if x ≤ 100 then 91 else x-10

 let rec f91 (n:int) : int
    variant { 101-n }
    ensures { result = f n }
 = if n ≤ 100 then
       f91 (f91 (n + 11))
    else
       n - 10
```

a função 91 (versão iterativa)

```
use import ref.Ref
(* iter k x = f^k(x) *)
clone import int.Iter  with type t = int, function f = f

let f91_nonrec (n0: int)
  ensures { result = f n0 }
= let e = ref 1 in
  let n = ref n0 in
  while !e > 0 do
    invariant { !e ≥ 0 ∧ iter !e !n = f n0 }
    variant   { 101 - !n + 10 * !e, !e }
    if !n > 100 then begin
      n := !n - 10;   e := !e - 1
    end else begin
      n := !n + 11;    e := !e + 1
    end
  done;
  !n
end
```

prova why3 da função 91



a terminação de um ciclo ou de uma função recursiva é assegurada por um **variante**

notação: **variant** $\{t_1, \ldots, t_n\}$
significado:

- ordem lexicográfica sobre os tuplos $t_1, \ldots, t_n$ em que
- a relação de ordem sobre cada t_i pode ser:
  - $y \prec x \triangleq y < x \wedge 0 \leq x$ se t_i é de tipo *int*
  - a ordem **subtermo próprio**, se o tipo de t_i é um tipo algébrico
  - uma ordem **bem fundada** r fornecida pelo utilizador (neste caso a notação é: **variant** $\{t \ \mathit{with} \ r\}$)

como já o referimos, a correção parcial é uma propriedade relativamente fraca visto que a não-terminação retira muito do sentido de uma prova de correção

a não-terminação é um **efeito**

Why3 regista-a e avisa o utilizador quando está em falta a garantia da sua ausência

a não ser que esteja presente uma cláusula explítica designada de **diverge**

dados mutáveis, vectores

só existe um tipo de dado mutável nativo em Why3

estruturas com **campos mutáveis**

como em OCaml

por exemplo, as referências são definidas desta forma

```
type ref α = { mutable contents "model_trace:" : α }

function (!) (x: ref α) : α = x.contents

let ref (v: α) ensures { result = { contents = v } }
    = { contents = v }

let (!) (r:ref α) ensures { result = !r } = r.contents

let (:=) (r:ref α) (v:α) ensures { !r = v } = r.contents ← v
```

a biblioteca Why3 introduz os vectores da seguinte forma

```
type array α model { length : int; mutable elts : map int α }
      invariant { 0 ≤ self.length }
```

onde

- `map` é o tipo lógico dos mapas puramente aplicativos
- a palavra chave **model** indica que o tipo *array* α é um tipo abstracto nos programas que se comporta logicamente conforme a dada estrutura
- o **invariante de tipo** é requerido a cada passagem de parâmetro de um valor deste tipo, quando um elemento deste tipo é criado ou modificado

não sendo os vectores definidos por um tipo concreto, mas sim por um tipo abstracto (os vectores são modelados), não podemos definir funções programáticas sobre vectores, mas sim declará-las e axiomatizá-las

exemplos:

```
(...) use import map.Map as M (...)

  function ([]) (a: array α) (i: int) : α = get a i
  function ([←]) (a: array α) (i: int) (v: α) : array α = set a i v

  val ([]) (a: array α) (i: int) : α
    requires { "expl:index in array bounds" 0 ≤ i < length a }
    ensures  { result = a[i] }

  val ([]←) (a: array α) (i: int) (v: α) : unit writes {a}
    requires { "expl:index in array bounds" 0 ≤ i < length a }
    ensures  { a.elts = M.set (old a.elts) i v }

  val length (a: array α) : int ensures { result = a.length }
```

quando escrevemos `a[i]`

- é um mero atalho sintáctico para `(Map.get a.elts i)`
- não provamos que *i* está dentro do âmbito do vector considerado (por definição, `a.elts` é um mapa sobre todos os inteiros)

no contexto de uma votação, os votos são recolhidos num vector

dado um multi-conjunto de N votos

A	A	A	C	C	B	B	C	C	C	B	C	C

determinar quem ganhou, caso haja maioria absoluta

MJRTY - uma solução elegante

uma solução desenhada por R. Boyer e S. Moore em 1980

em tempo linear

em espaço constante (recorrendo só a 3 variáveis)

Chapter 5

MJRTY—A Fast Majority Vote Algorithm[1]

Robert S. Boyer and J Strother Moore

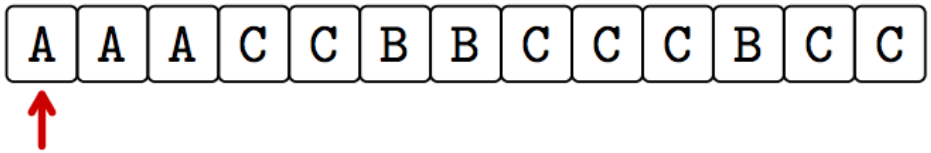
Computer Sciences Department
University of Texas at Austin
and
Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas

Abstract

A new algorithm is presented for determining which, if any, of an arbitrary number of candidates has received a majority of the votes cast in an election. The number of comparisons required is at most twice the number of votes. Furthermore, the algorithm uses storage in a way that permits an efficient use of magnetic tape. A Fortran version of the algorithm is exhibited. The Fortran code has been proved correct by a mechanical verification system for Fortran. The system and the proof are discussed.

[1]The work described here was conducted in the Computer Science Laboratory of SRI International and suported in part by NASA Contract NAS1-15528, NSF Grant MCS-7904081, and ONR Contract N00014-75-C-0816 1981. A brief history of this work is given in the concluding section.

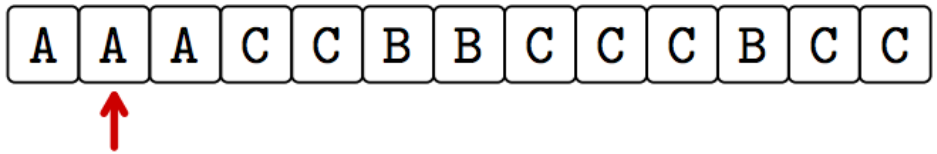
MJRTY - como funciona?



$\texttt{cand} = A$

$\texttt{k} = 1$

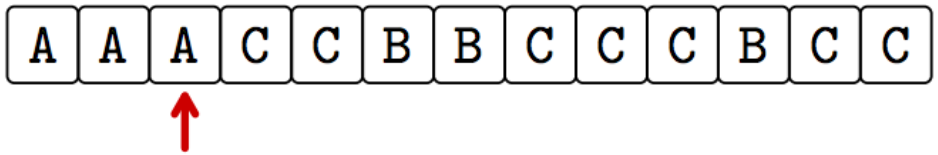
MJRTY - como funciona?



$\texttt{cand} = A$

$\texttt{k} = 2$

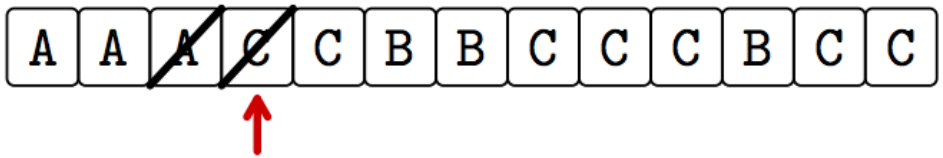
MJRTY - como funciona?



$\texttt{cand} = A$

$\texttt{k} = 3$

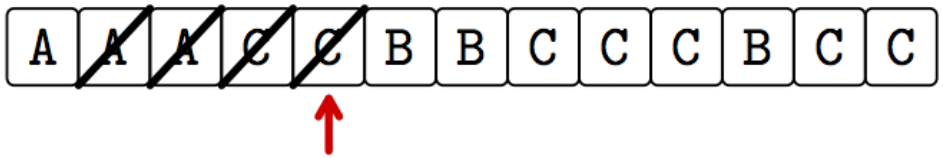
MJRTY - como funciona?



$\texttt{cand} = A$

$\texttt{k} = 2$

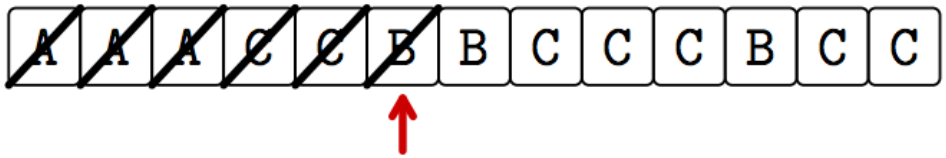
MJRTY - como funciona?



$\texttt{cand} = A$

$\texttt{k} = 1$

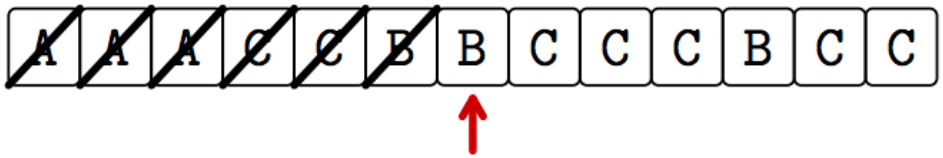
MJRTY - como funciona?



$\texttt{cand} = A$

$\texttt{k} = 0$

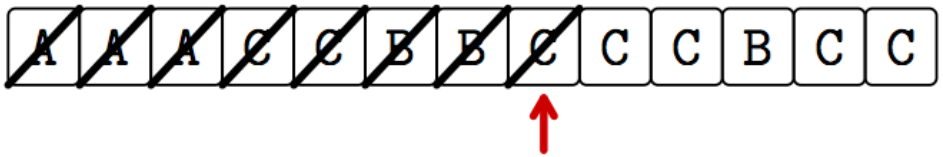
MJRTY - como funciona?



$\texttt{cand} = B$

$\texttt{k} = 1$

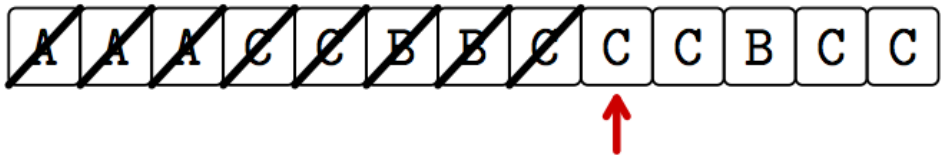
MJRTY - como funciona?



$\texttt{cand} = B$

$\texttt{k} = 0$

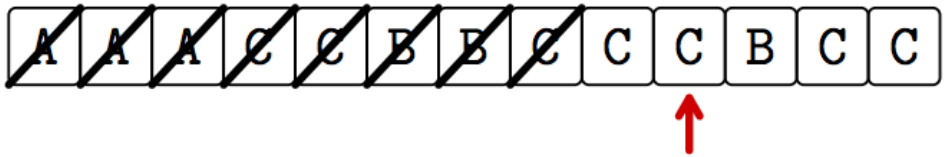
MJRTY - como funciona?



$\texttt{cand} = C$

$\texttt{k} = 1$

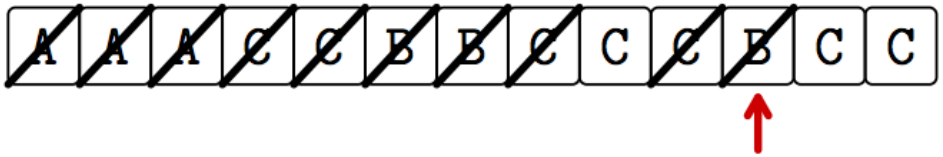
MJRTY - como funciona?



$\texttt{cand} = C$

$\texttt{k} = 2$

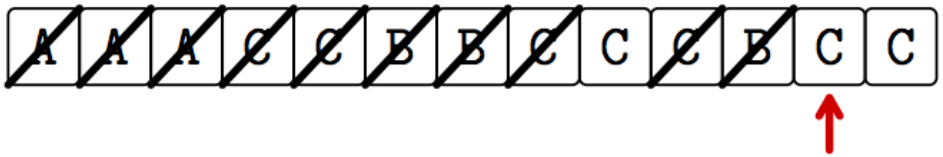
MJRTY - como funciona?



$\texttt{cand} = C$

$\texttt{k} = 1$

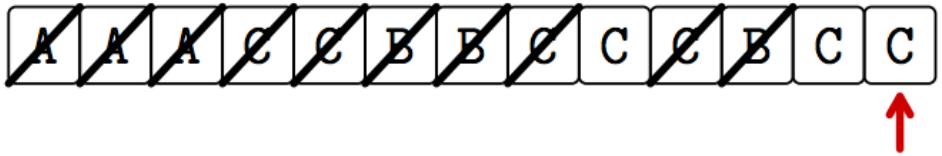
MJRTY - como funciona?



$\texttt{cand} = C$

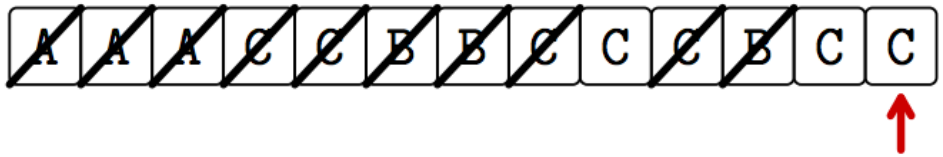
$\texttt{k} = 2$

MJRTY - como funciona?



$\texttt{cand} = C$

$\texttt{k} = 3$



$\texttt{cand} = C$
$\texttt{k} = 3$

C teve o maior número de votos

em seguida, verifica-se se C teve efectiva maioria absoluta
(neste caso particular tem... $7 > 13/2$)

MJRTY em Why3

```
let mjrty (a: array candidate) =
  let n = length a in
  let cand = ref a[0] in let k = ref 0 in
  for i = 0 to n-1 do
    if !k = 0 then begin cand := a[i]; k := 1 end
    else if !cand = a[i] then incr k else decr k
  done;
  if !k = 0 then raise Not_found;
  try
    if 2 * !k > n then raise Found; k := 0;
    for i = 0 to n-1 do
      if a[i] = !cand then begin
        incr k; if 2 * !k > n then raise Found
      end
    done;
    raise Not_found
  with Found →
    !cand
  end
```

- pre-condição

```
let mjrty (a: array candidate)
  requires {1 ≤ length a}
```

- pos-condição no caso de sucesso

```
ensures { 2 * numof a result 0 (length a) > length a}
```

- pos-condição no caso de falha

```
raises { Not_found →
                ∀ c : candidate.
                2 * numof a c 0 (length a) ≤ length a }
```

sendo `numof a v l u` (do módulo `NumOfEq`)o número de $a[i]_{(l \leq i < u)}$ tais que $a[i] = v$

de notar que a multiplicação por dois no lado esquerdo das desigualdades explica-se pelo melhor suporte das ferramentas de prova automática à multiplicação relativamente à divisão (por dois no lado direito, neste caso)

- invariante do primeiro ciclo

```
for i = 0 to n-1 do
    invariant { 0 ≤ !k ≤ numof    a !cand 0 i}
    invariant { 2 * (numof a !cand 0 i - !k) ≤ i - !k }
    invariant { ∀ c: candidate. c 6= !cand →
                        2 * numof a c 0 i ≤ i - !k }
        ...
```

- invariante do segundo ciclo

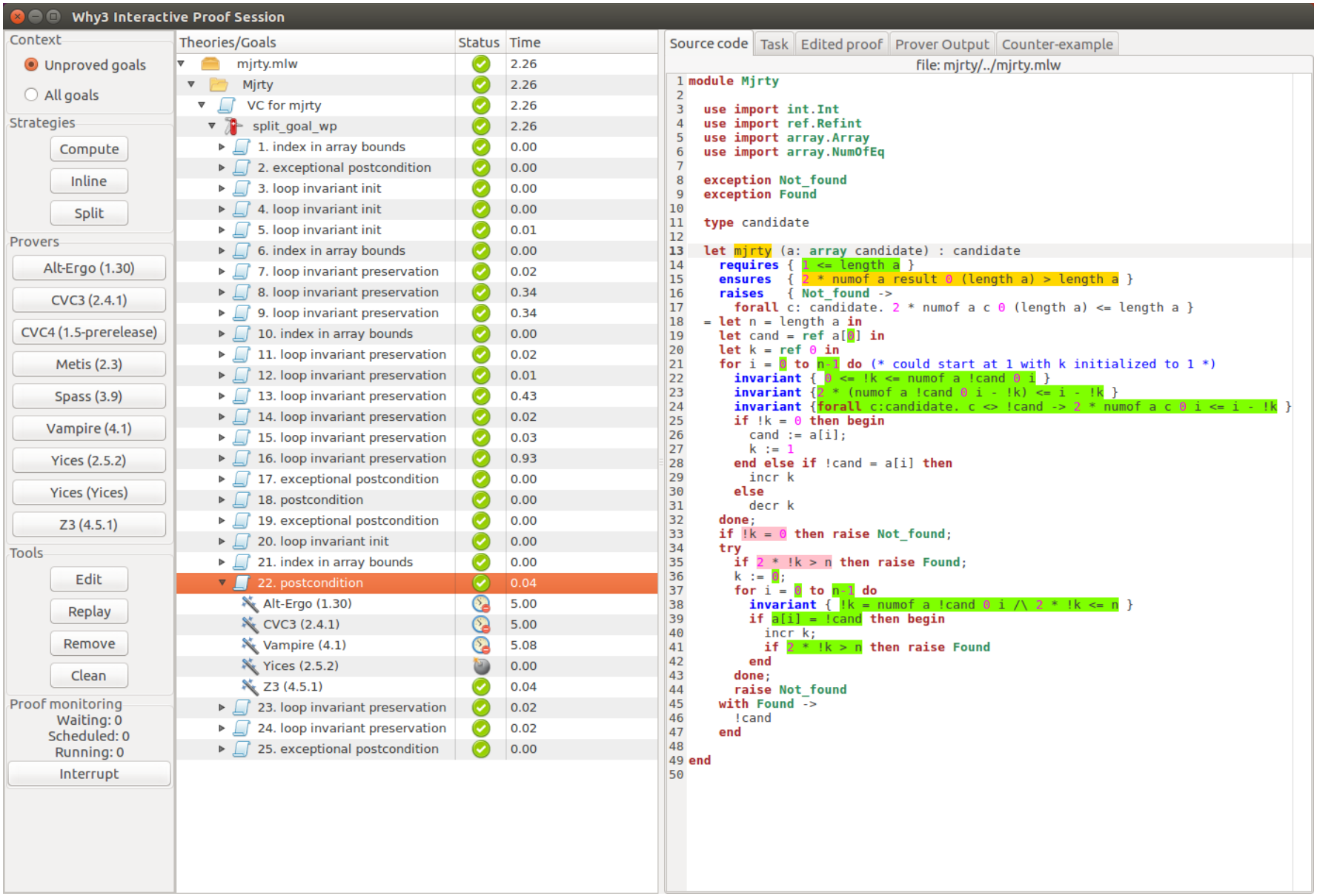
```
for i = 0 to n-1 do
    invariant { !k = numof a !cand 0 i }
    invariant { 2 * !k ≤ n }
    ...
```

as condições de verificação expressam:

- ***safety***
  - acesso ao vector dentro do limite do seu comprimento
  - terminação
- **correção funcional**
  - os invariantes de ciclo são verificados à entrada do ciclo e preservados pelas iterações desse
  - tendo em conta as pre-condições, as pos-condições são garantidas

todos são provados automaticamente

MJRTY no Why3



podemos extrair este código WhyML para OCaml

```
why3 extract -D ocaml64 -D mjrty -T mjrty.Mjrty -o .
```

socorremo-nos aqui de dois ***drivers***

- um *driver* genérico (disponível com o Why3) para a tradução para OCaml 64-bits
  (indica que se pretende traduzir `int` para `Zarith`, `array` para os vectores OCaml, etc.)
- um *driver* criado especificamente para este exemplo, nomeadamente

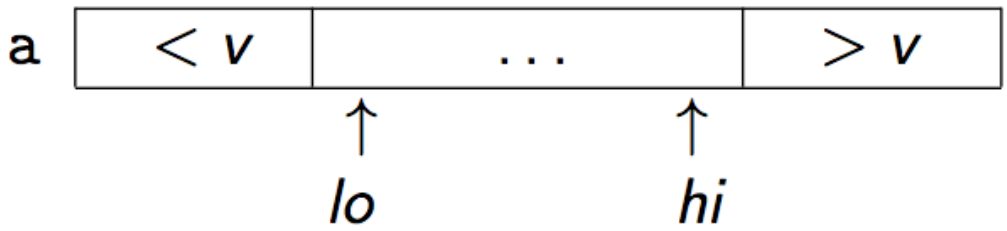
```
module mjrty.Mjrty
  syntax type candidate "char"
end
```

um exemplo possível de compilação

```
ocamlopt ...  zarith.cmxa why3extract.cmxa mjrty__Mjrty.ml test_mjrty.ml
```

Want one cat? here it is

```
lo := 0
hi := len(a) - 1
while lo <= hi do
  m := lo + (hi - lo)/2
  if a[m] < v
     lo := m + 1
  else if a[m] > v
     hi := m - 1
  else
     return m
return -1
```



```
let binary search (a : array int) (v : int) : int
  requires { ∀ i j. 0 ≤ i ≤ j < length a → a[i] ≤ a[j] }
  ensures { (0 ≤ result < length a ∧ a[result] = v)
      ∨ (result = -1 ∧ ∀ i. 0 ≤ i < length a → a[i] ≠ v) }
```

ou ainda

```
let binary_search (a : array int) (v : int)
  requires { ∀ i j : int. 0 ≤ i ≤ j < length a → a[i] ≤ a[j] }
  ensures { 0 ≤ result < length a ∧ a[result] = v }
  raises { Not_found → ∀ i:int. 0 ≤ i < length a → a[i] ≠ v}
```

```
let binary search (a : array int) (v : int) : int
  requires { ∀ i j. 0 ≤ i ≤ j < length a → a[i] ≤ a[j] }
  ensures { (0 ≤ result < length a  →  a[result] = v)
      ∧ (result = -1 ∧ ∀ i. 0 ≤ i < length a → a[i] ≠ v) }
```

o programa pode agora devolve o valor -2 e mesmo assim ser demonstrado correcto

```
let binary search (a: array int) (v: int) : int
  requires { ∀ i j. 0 ≤ i ≤ j < length a → a[i] ≤ a[j] }
  ensures { 0 ≤ result < length a → a[result] = v
    ∧ result = -1 → ∀ i. 0 ≤ i < length a → a[i] ≠ v }
```

(note a ausência de parêntesis)
o programa pode agora devolver 42 e mesmo assim ser demonstrado correcto

antes de realizar qualquer esforço de prova (o *why*, recorda-se), tenha em atenção a escrita de especificações comportamentais acertadas (*what*)

mais ainda, tenha um revisor que concorde com o teor da especificação

senão, o processo de prova não tem nem faz sentido...

compromisso:

há muitas formas de escrever a mesma especificação

algumas se adequam mais a leitura (por nós), outras são melhores para o seu processo por ferramentas de prova (automáticas)

tomando o exemplo do que é *estar ordenado*, no lugar de

$$\text{requires } \{\ \forall\ i\ j.\ 0 \leq i \leq j < \text{length a} \rightarrow a[i] \leq a[j]\ \}$$

poderíamos ter escrito

$$\text{requires } \{\ \forall\ i.\ 0 \leq i < \text{length a - 1} \rightarrow a[i] \leq a[i + 1]\ \}$$

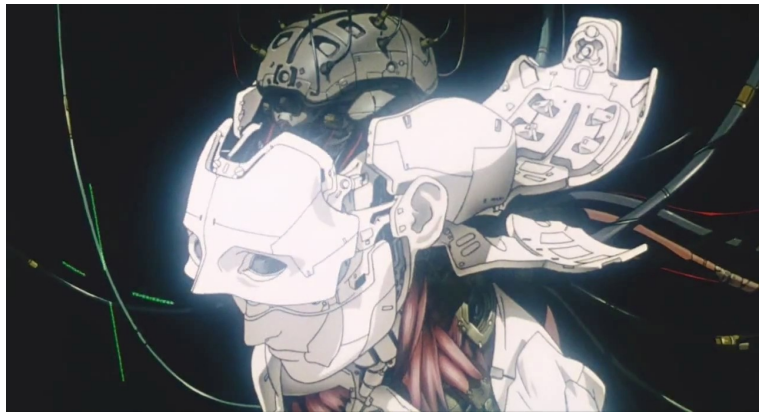
embora equivalentes, a última formulação requer indução para ser demonstrada

ora, a indução, tipicamente, requer capacidades de demonstração para além dos habituais demonstradores automáticos

no exemplo da pesquisa binária, é bastante fácil equivocar-se sobre o calculo dos limites (por exemplo escrever `lo := m` no lugar de `lo := m +1`)
se não está a demonstrar a terminação, sempre pode ainda demonstrar a correcção, mas tratar-se-á de **correcção parcial**

ghost code

trata-se de dados e código que é adicionado ao programa com o **propósito exclusivo** do seu uso nas especificações ou provas



imaginemos que procuramos o menor número de Fibonacci que seja maior ou igual de que um dado n

```
    a, b <- 0, 1
    while a < n do
        a, b <- b, a + b
    return a
```

poderíamos apresentar a especificação seguinte

```
let a = ref 0 in
let b = ref 1 in
while !a < n do
    invariant { ∃ i. i≥0 && !a = fib i && !b = fib (i+1) }
```

mas as quantificações existencias geram VCs de demonstração automática difícil

no lugar da solução anterior, podemos guardar memória do valor adequado para i numa **referência ghost**

```
let a = ref 0 in
let b = ref 1 in
let ghost i = ref 0 in (* ghost data *)
while !a < n do
      invariant { !i ≥ 0 && !a = fib !i && !b = fib (!i+1) }
      ...
      i := !i + 1 (* ghost statement *)
 done
```

desta forma, em vez de exigir ao demonstrador automático que adivinhe o valor correcto para esse i, fornecemo-lo

- código *ghost* pode ler valores do programa, mas não pode modificá-los
- código *ghost* não pode modificar o fluxo de controlo do programa
- o código *normal* não vê os valores *ghost*

consequência: o código ghost pode ser **removido** sem modificação observável do comportamento do programa

(e é de facto removido pelo processo de extração para OCaml)

uma função

```
let f (x: t) : unit requires { P } ensures { Q } = ...
```

que é pura (i.e. não modifica dados globais) e que termina pode ser transformada **automaticamente** no lema

```
lemma f: ∀ x: t. P → Q
```

a declaração `let lemma` indica ao Why3 para realizar tal operação

```
let rec lemma fib_pos (n: int) : unit
requires { n ≥ 1 }
variant { n }
ensures { fib n ≥ 1 } =
if n > 2 then begin fib_pos (n - 2); fib_pos (n - 1) end
```

automaticamente obtemos

```
lemma fib_pos : ∀ n: int. n≥1 → fib n ≥ 1
```

- foi estabelecida uma **prova por indução**, graças ao *variant* ao cálculo das precondições mais fracas
- podemos fazer chamadas explícitas à função lema

```
fib_pos 42; (* this is ghost code *)
...
```

poupa os provedores automáticos propondo-lhes uma instanciação directa do lema

o código ghost pode ser usado com proveito para modelar o conteúdo de estruturas de dados

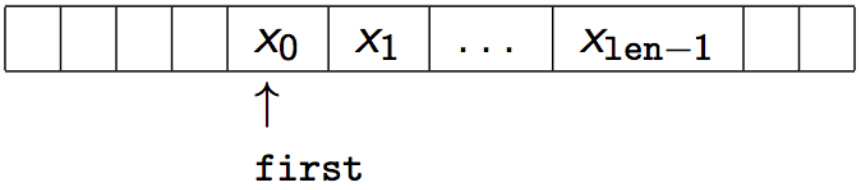
imaginemos, pode exemplo que queremos modelar uma fila com uma capacidade máxima

```
type queue α
val create: int → queue α
val push: α → queue α → unit
val pop: queue α → α
```

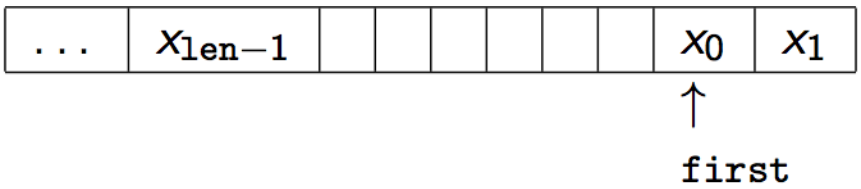
pode ser implementada a custa de um vector

```
type buffer α = {
        mutable first: int;
        mutable len : int;
                      data : array α;
 }
```

elementos em número `len` são armazenados, a partir do índice `first`



mas o arquivo destes elementos pode extender-se comodamente para além do fim "físico" do vector



por forma a especificar as operações sobre estas filas, gostaríamos de **modelar** o conteúdos das filas como. por exemplo, sequências de elementos

uma forma de o fazer é usar **código ghost**

juntamos dois **campos ghost** para modelar o conteúdo da fila

```
type queue α = {
  ...
  ghost capacity: int;
  ghost mutable sequence: Seq.seq α;
}
```

podemos então usá-los na especificação das operações desejadas

```
val create (n: int) (dummy: α) : queue α
    requires { n > 0 }
    ensures { result.capacity = n }
    ensures { result.sequence = Seq.empty }

val push (q: queue α) (x: α) : unit
    requires { Seq.length q.sequence < q.capacity }
    writes   { q.sequence }
    ensures { q.sequence = Seq.snoc (old q.sequence) x }

val pop (q: queue α) : α
    requires { Seq.length q.sequence > 0 }
    writes   { q.sequence }
    ensures { result = (old q.sequence)[0] }
    ensures { q.sequence = (old q.sequence)[1 ..] }
```

com base nesta especificação, é possível provar algum código que usa estas filas

```
let harness () =
  let q = create 10 0 in
  push q 1;
  push q 2;
  push q 3;
  let x = pop q in assert { x = 1 };
  let x = pop q in assert { x = 2 };
  let x = pop q in assert { x = 3 };
  ()
```

para compatibilizar os campos habituais dos campos ghost, utilizamos o mecanismo do invariante de tipo que estabelece o seguinte:

```
type buffer α =
    ...
  invariant {
    self.capacity = Array.length.data ∧
    0 ≤ self.first < self.capacity ∧
    0 ≤ self.len ≤ self.capacity ∧
    self.len = Seq.length self.sequence ∧
    ∀ i: int. 0 ≤ i < self.len →
          (self.first + i < self.capacity →
              Seq.get self.sequence i = self.data[self.first + i]) ∧
          (0 ≤ self.first + i - self.capacity →
              Seq.get self.sequence i = self.data[self.first + i - self.capacity]) }
```

já encontramos este conceito

tal invariante de tipo está assegurado nos **limites das funções** que manipulam valores deste tipo
logo

- é **assumido** à entrada da função
- **deve ser verificado**
  - quando uma função (que manopla pelo menos um valor de tal tipo) é invocada
  - quando a função termina, para os valores deste tipo que foram alterados ou que são devolvidos

código ghost é acrescentado para alterar os campos ghost em conformidade

um exemplo:

```
let push (b: buffer α) (x: α) : unit
 =
 ghost b.sequence ← Seq.snoc b.sequence x;
 let i = b.first + b.len in
 let n = Array.length b.data in
 b.data[if i ≥ n then i - n else i] ← x;
 b.len < b.len + 1
```

programação puramente aplicativa

uma **ideia chave** da lógica de Hoare

any types and symbols from the logic can be used in programs

de notar que já usamos alguns tipos desta forma, por exemplo o tipo `int`

podemos então proceder da mesma forma com tipos de dados algébricos

na biblioteca , podemos encontrar

```
type bool = True | False                    (* ver bool.Bool *)
type option α = None | Some α               (* ver option.Option*)
type list α = Nil | Cons α (list α)         (* ver in list.List *)
```

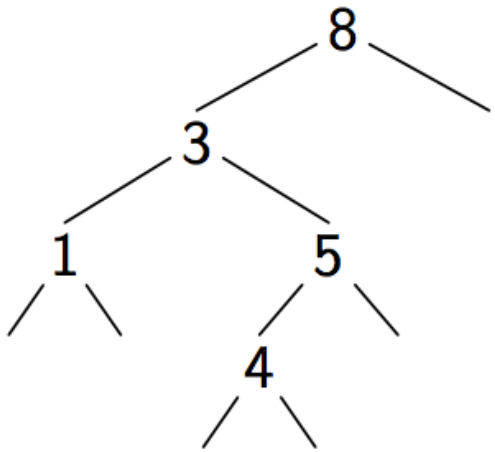
consideremos as árvores binárias de um tipo `elt`

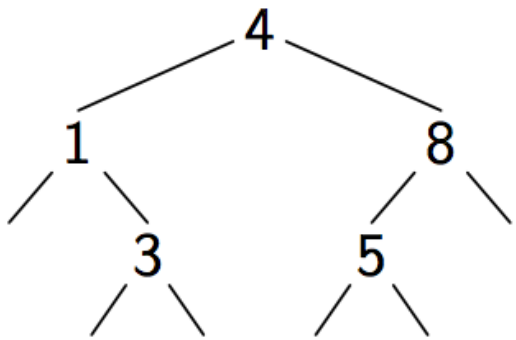
```
type elt

type tree =
  | Empty
  | Node tree elt tree
```

e o seguinte problema

dadas duas árvores binárias,

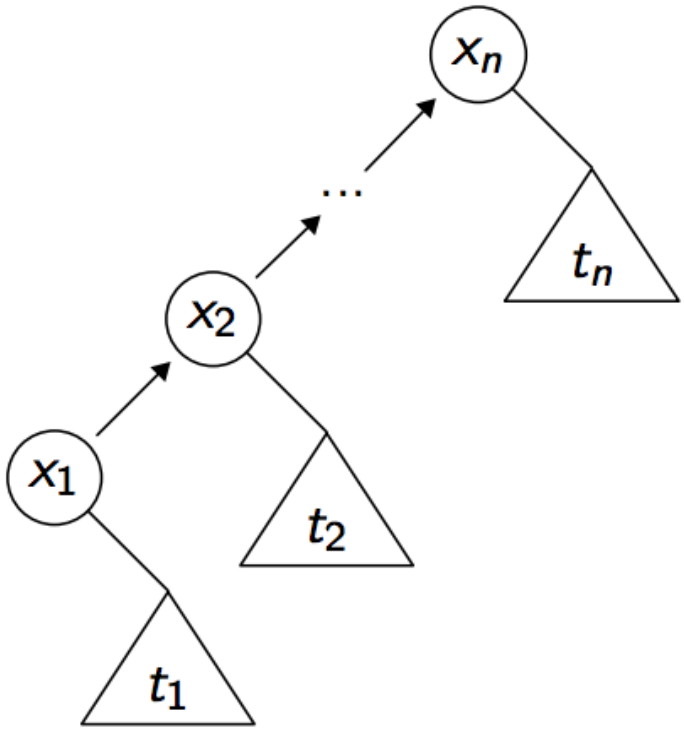
poderão elas conter os mesmos elementos quando objecto de uma travessia ordenada?



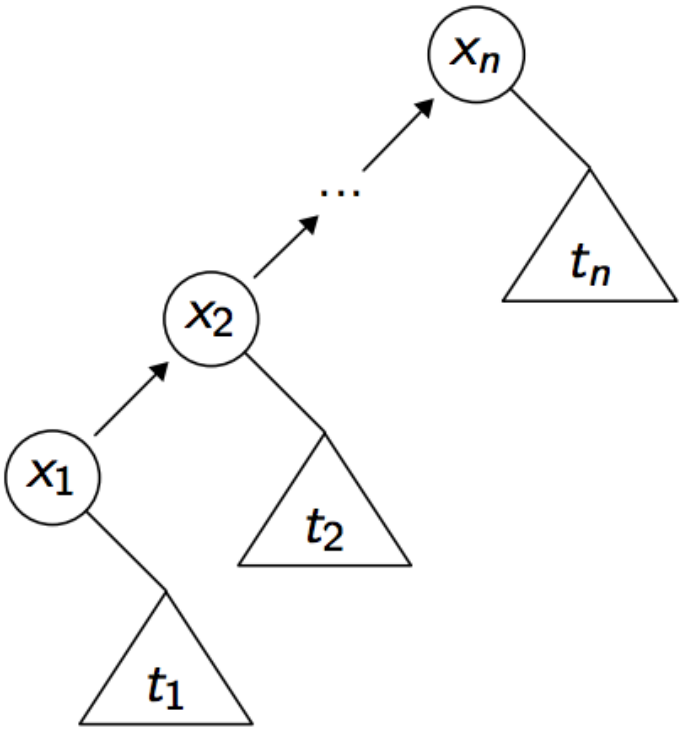


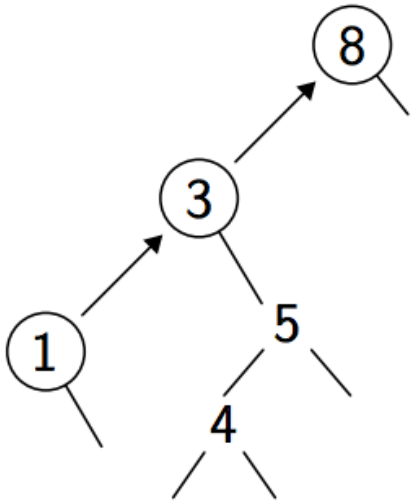
```
function elements (t: tree) : list elt = match t with
  | Empty → Nil
  | Node l x r → elements l ++ Cons x (elements r)
end

let same fringe (t1 t2: tree) : bool
  ensures { result=True ↔ elements t1 = elements t2 }
  =
  ...
```

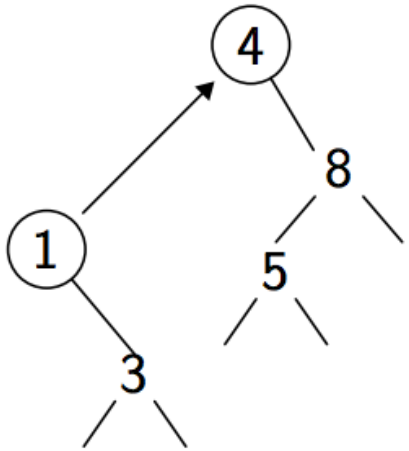
uma solução: olhar para o ramo esquerdo como uma lista, de baixo para cima



uma solução: olhar para o ramo esquerdo como uma lista, de baixo para cima







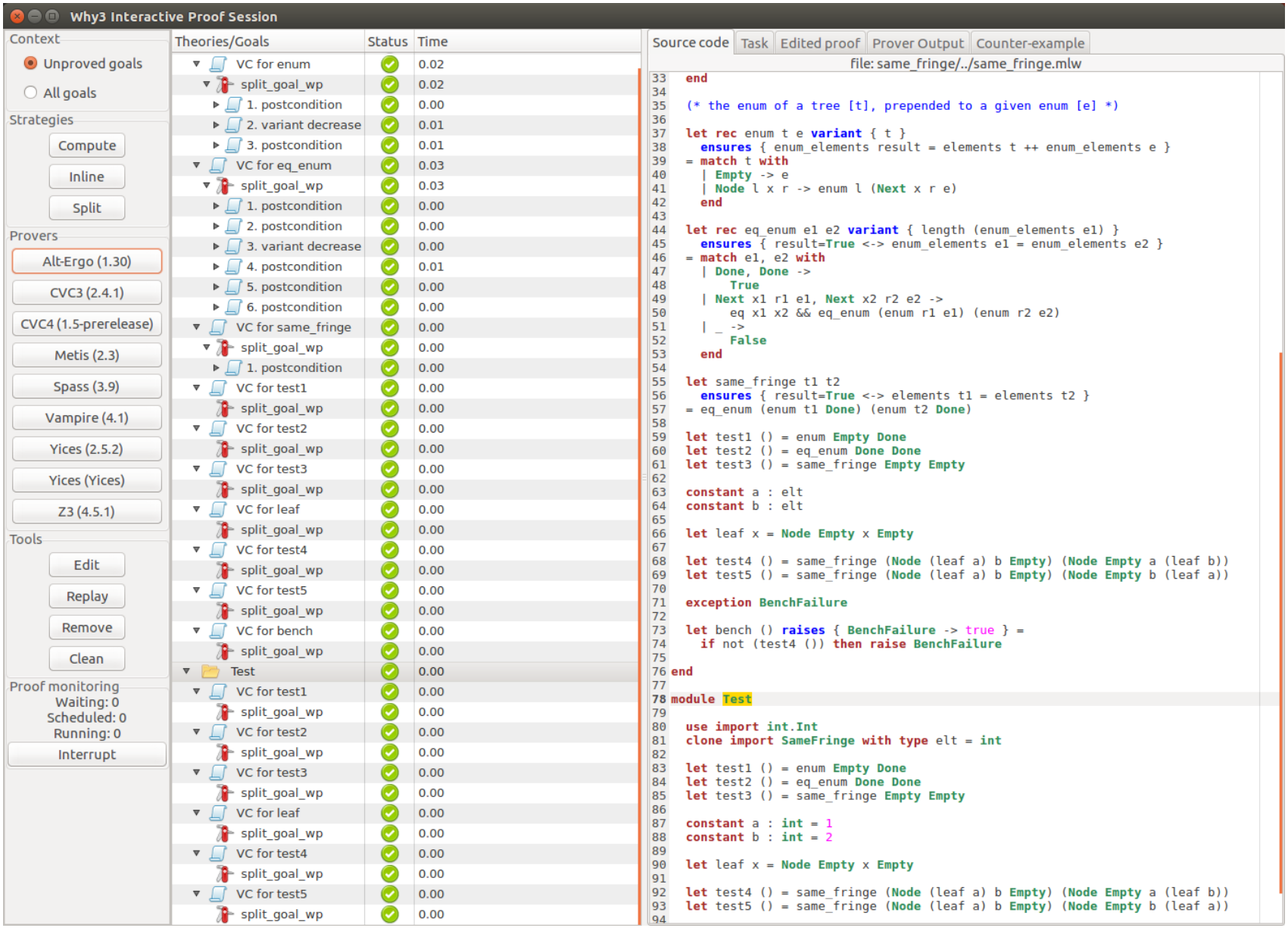
```
(* binary trees with elements at the nodes *)

type elt

val eq (x y: elt) : bool
  ensures { result ↔ x=y }

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
```

```
(* the left spine of a tree, as a bottom-up list *)

type enum =
  | Done
  | Next elt tree enum

function enum_elements (e : enum) : list elt = match e with
  | Done → Nil
  | Next x r e → Cons x (elements r ++ enum_elements e)
end

(* the enum of a tree [t], prepended to a given enum [e] *)

let rec enum t e variant { t }
  ensures { enum_elements result = elements t ++ enum_elements e }
= match t with
  | Empty → e
  | Node l x r → enum l (Next x r e)
  end
```

```
let same_fringe t1 t2
  ensures { result=True ↔ elements t1 = elements t2 }
= eq_enum (enum t1 Done) (enum t2 Done)
```

```
let test1 () = enum Empty Done
let test2 () = eq_enum Done Done
let test3 () = same_fringe Empty Empty


constant a : elt
constant b : elt

let leaf x = Node Empty x Empty

let test4 () = same_fringe (Node (leaf a) b Empty) (Node Empty a (leaf b))
let test5 () = same_fringe (Node (leaf a) b Empty) (Node Empty b (leaf a))

exception BenchFailure

let bench () raises { BenchFailure → true } =
  if not (test4 ()) then raise BenchFailure
```

same fringe


Why3 Interactive Proof Session
Context
Unproved goals
All goals
Strategies
Compute
Inline
Split
Provers
Alt-Ergo (1.30)
CVC3 (2.4.1)
CVC4 (1.5-prerelease)
Metis (2.3)
Spass (3.9)
Vampire (4.1)
Yices (2.5.2)
Yices (Yices)
Z3 (4.5.1)
Tools
Edit
Replay
Remove
Clean
Proof monitoring
Waiting: 0
Scheduled: 0
Running: 0
Interrupt
Theories/Goals
Status
Time
VC for enum 0.02
split_goal_wp 0.02
1. postcondition 0.00
2. variant decrease 0.01
3. postcondition 0.01
VC for eq_enum 0.03
split_goal_wp 0.03
1. postcondition 0.00
2. postcondition 0.00
3. variant decrease 0.00
4. postcondition 0.01
5. postcondition 0.00
6. postcondition 0.00
VC for same_fringe 0.00
split_goal_wp 0.00
1. postcondition 0.00
VC for test1 0.00
split_goal_wp 0.00
VC for test2 0.00
split_goal_wp 0.00
VC for test3 0.00
split_goal_wp 0.00
VC for leaf 0.00
split_goal_wp 0.00
VC for test4 0.00
split_goal_wp 0.00
VC for test5 0.00
split_goal_wp 0.00
VC for bench 0.00
split_goal_wp 0.00
Test 0.00
VC for test1 0.00
split_goal_wp 0.00
VC for test2 0.00
split_goal_wp 0.00
VC for test3 0.00
split_goal_wp 0.00
VC for leaf 0.00
split_goal_wp 0.00
VC for test4 0.00
split_goal_wp 0.00
VC for test5 0.00
split_goal_wp 0.00
Source code
Task
Edited proof
Prover Output
Counter-example
file: same_fringe/../same_fringe.mlw
33 end
34
35 (* the enum of a tree [t], prepended to a given enum [e] *)
36
37 let rec enum t e variant { t }
38 ensures { enum_elements result = elements t ++ enum_elements e }
39 = match t with
40 | Empty -> e
41 | Node l x r -> enum l (Next x r e)
42 end
43
44 let rec eq_enum e1 e2 variant { length (enum_elements e1) }
45 ensures { result=True <-> enum_elements e1 = enum_elements e2 }
46 = match e1, e2 with
47 | Done, Done ->
48 True
49 | Next x1 r1 e1, Next x2 r2 e2 ->
50 eq x1 x2 && eq_enum (enum r1 e1) (enum r2 e2)
51 | _ ->
52 False
53 end
54
55 let same_fringe t1 t2
56 ensures { result=True <-> elements t1 = elements t2 }
57 = eq_enum (enum t1 Done) (enum t2 Done)
58
59 let test1 () = enum Empty Done
60 let test2 () = eq_enum Done Done
61 let test3 () = same_fringe Empty Empty
62
63 constant a : elt
64 constant b : elt
65
66 let leaf x = Node Empty x Empty
67
68 let test4 () = same_fringe (Node (leaf a) b Empty) (Node Empty a (leaf b))
69 let test5 () = same_fringe (Node (leaf a) b Empty) (Node Empty b (leaf a))
70
71 exception BenchFailure
72
73 let bench () raises { BenchFailure -> true } =
74 if not (test4 ()) then raise BenchFailure
75
76 end
77
78 module Test
79
80 use import int.Int
81 clone import SameFringe with type elt = int
82
83 let test1 () = enum Empty Done
84 let test2 () = eq_enum Done Done
85 let test3 () = same_fringe Empty Empty
86
87 constant a : int = 1
88 constant b : int = 2
89
90 let leaf x = Node Empty x Empty
91
92 let test4 () = same_fringe (Node (leaf a) b Empty) (Node Empty a (leaf b))
93 let test5 () = same_fringe (Node (leaf a) b Empty) (Node Empty b (leaf a))
94

aritmética por computador

modelemos a aritmética 32-bit com sinal

proveito:

- podemos assim tentar assegurar a ausência de *overflow* aritmético
- modelamos mais fielmente a operações aritméticas tais como são feitas por um computador (i.e. tendo em conta os *overflows*)

um **requisito essencial** para isso: não queremos perder a capacidade que os SMT tem em calcular, em fazer aritmética

modelemos a aritmética 32-bit com sinal com o tipo seguinte

```
type int32
```

o valor inteiro de tal tipo é-nos dado pela função

```
function toint int32 : int
```

ideia chave: nas anotações, usamos somente o tipo `int`

ou seja, uma variável programática introduzida como `x: int32` é referenciada nas anotações lógicas como `toint x`

definimos o âmbito dos inteiros 32-bits com sinal

```
function min int: int = - 0x8000 0000 (* -2^31 *)
function max int: int = 0x7FFF FFFF (* 2^31-1 *)
```

e estabelecemos

```
axiom int32 domain:
  ∀ x: int32. min int ≤ toint x ≤ max int
```

... ou até mesmo criá-los

```
val ofint (x: int) : int32
  requires { min int ≤ x ≤ max int }
  ensures { toint result = x }
```

cada expressão programática tal como

$$x + y$$

é traduzida para

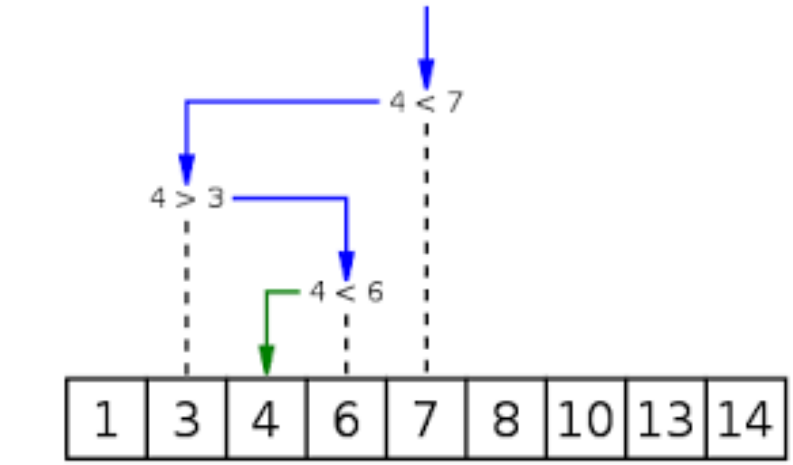
$$\text{ofint (toint x + toint y)}$$

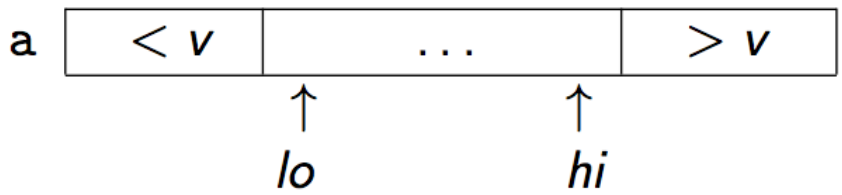
tal tratamento assegura a ausência de *overflow* aritmética

mas com o custo de um maior numero de VCs por demonstrar

verifiquemos a ausência de *overflow* aritmético no contexto da pesquisa binária

```
let binary_search (a : array int32) (v : int32)
 = try
     let l = ref (of_int 0) in
     let u = ref (length a -
of_int 1) in
     while !l ≤ !u do
       let m = (!l + !u) / of_int 2 in
       if a[m] < v then
         l := m + of_int 1
       else if a[m] > v then
         u := m - of_int 1
       else
         raise (Break m)
     done;
     raise Not_found
   with Break i →    i
   end
```

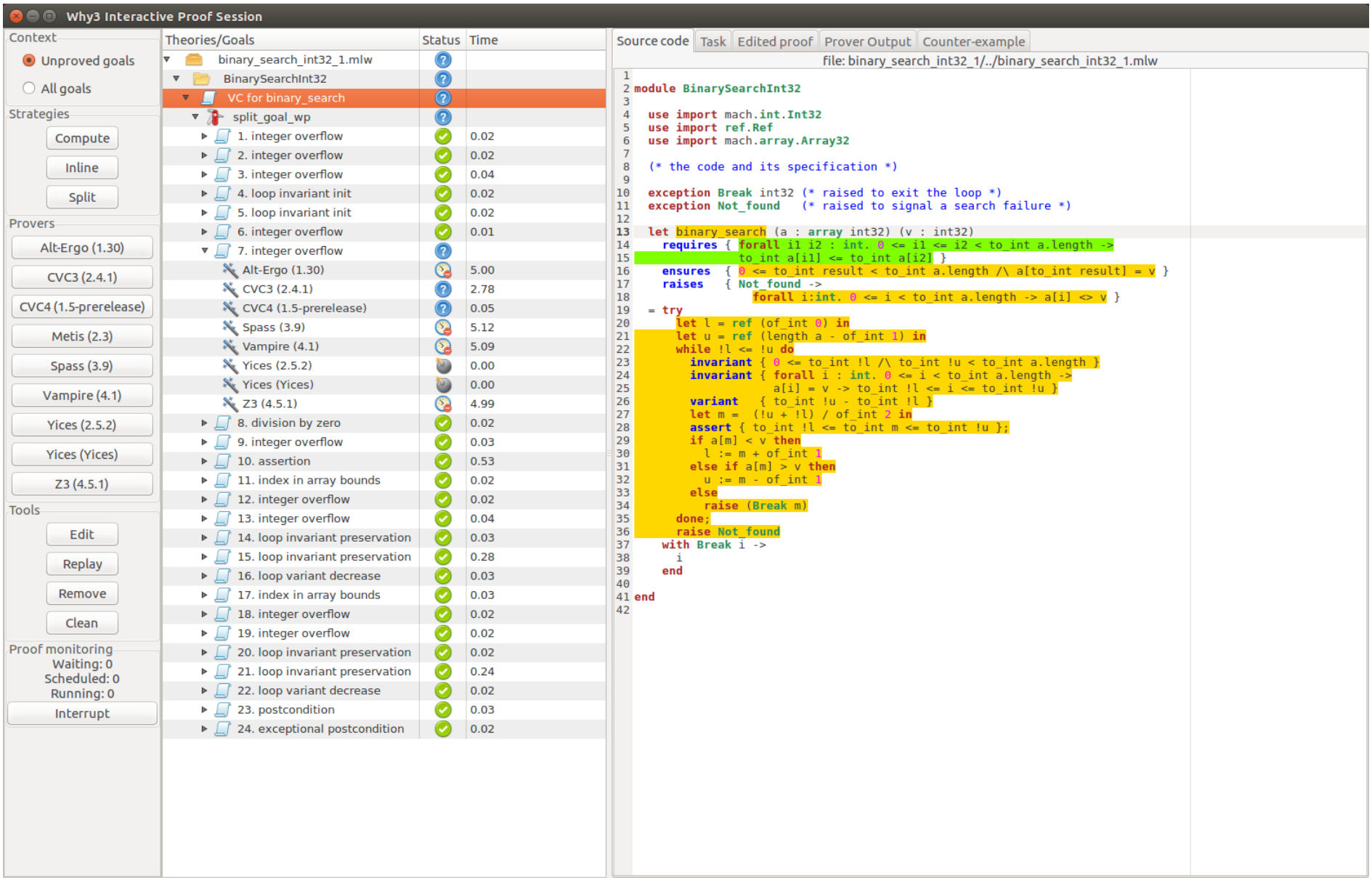




versão com as devidas especificações comportamentais

```
let binary_search (a : array int32) (v : int32)
    requires { ∀ i1 i2 : int. 0 ≤ i1 ≤ i2 < to_int a.length →
                 to_int a[i1] ≤ to_int a[i2] }
    ensures  { 0 ≤ to_int result < to_int a.length ∧ a[to_int result] = v }
    raises   { Not_found →∀ i:int. 0 ≤ i < to_int a.length → a[i] ≠ v }
  = try
      let l = ref (of_int 0) in
      let u = ref (length a - of_int 1) in
      while !l ≤ !u do
        invariant { 0 ≤ to_int !l ∧ to_int !u < to_int a.length }
        invariant { ∀ i : int. 0 ≤ i < to_int a.length →
                      a[i] = v → to_int !l ≤ i ≤ to_int !u }
        variant   { to_int !u - to_int !l }
        let m = (!l + !u) / of_int 2 in
        assert { to_int !l ≤ to_int m ≤ to_int !u };
        if a[m] < v then       l := m + of_int 1
        else if a[m] > v then  u := m - of_int 1
        else  raise (Break m)
      done;
      raise Not_found
    with Break i →  i
    end
```

pesquisa binária com why3



a prova não se consegue concluir com sucesso... porque encontramos um bug!

o cálculo

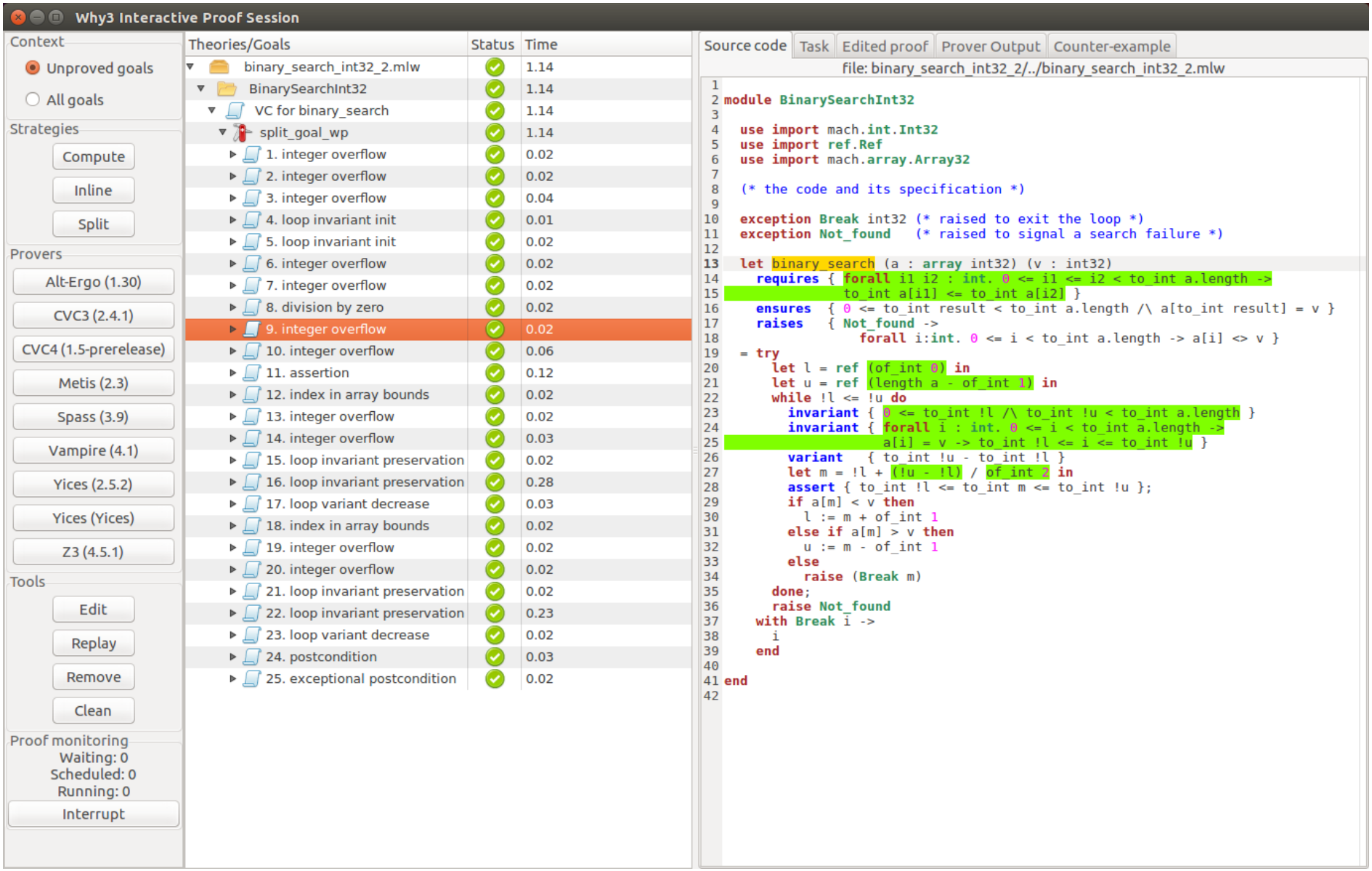
```
let m = (!l + !u) / 2 in
```

pode provocar um *overflow* aritmético
por exemplo com vectores de grande tamanho (onde !l + !u ultrapassa o valor máximo para os inteiros 32 bits com sinal)

uma correção possível é:

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

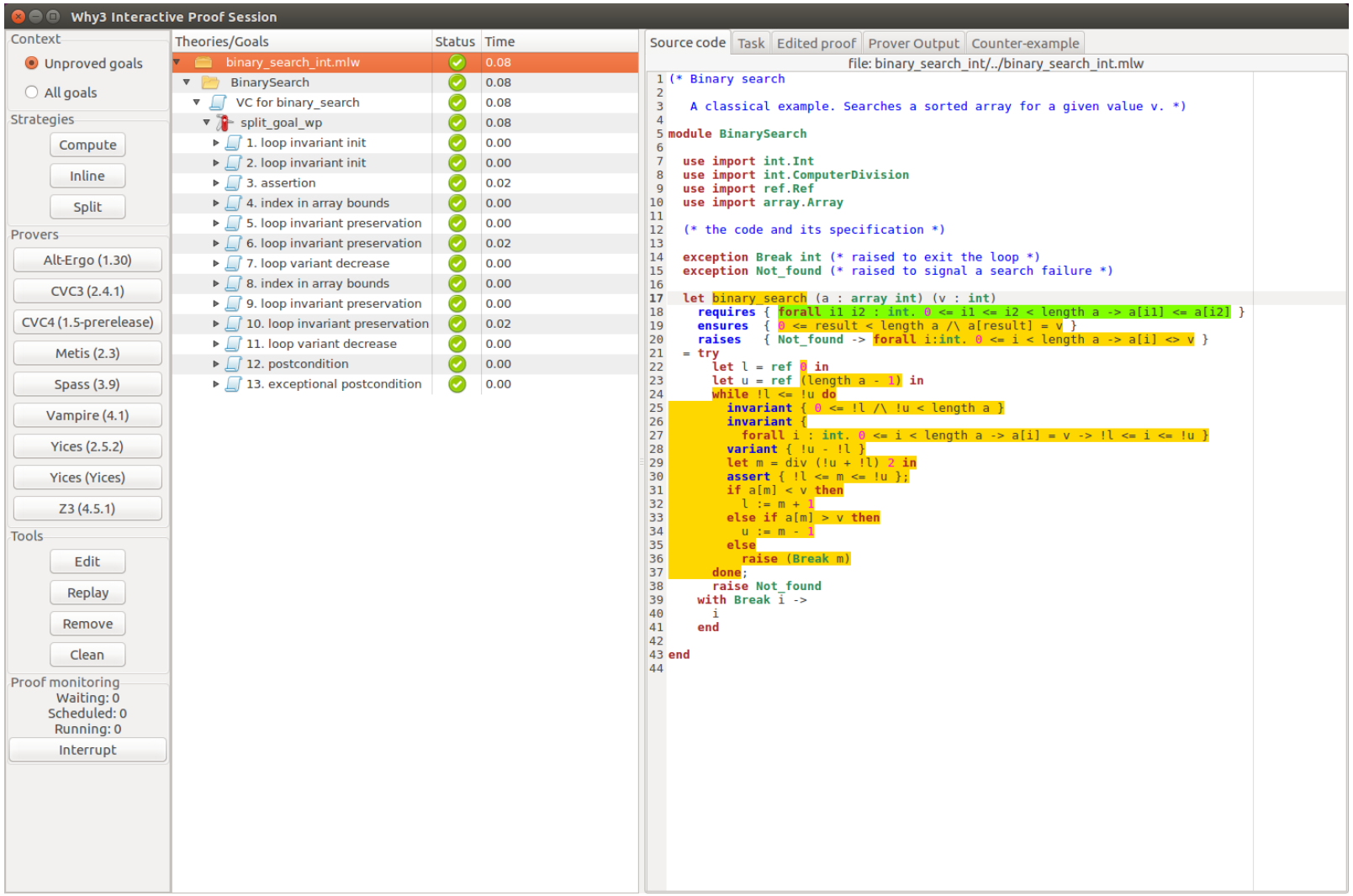
pesquisa binária corrigida com why3



para realçar a importância da representação dos dados,
se avaliamos este código com o why3

```
let binary_search (a : array int) (v : int)
    requires { ∀ i1 i2 : int. 0 ≤ i1 ≤ i2 < length a → a[i1] ≤ a[i2] }
    ensures  { 0 ≤ result < length a ∧ a[result] = v }
    raises   { Not_found → ∀ i:int. 0 ≤ i < length a → a[i] ≠ v }
 = try
     let l = ref 0 in
     let u = ref (length a - 1) in
     while !l ≤ !u do
       invariant { 0 ≤ !l ∧ !u < length a }
       invariant { ∀ i : int. 0 ≤ i < length a → a[i] = v → !l ≤ i ≤ !u }
       variant { !u - !l }
       let m = !l + div (!u - !l) 2 in
       assert { !l ≤ m ≤ !u };
       if a[m] < v then  l := m + 1
       else if a[m] > v then u := m - 1
       else raise (Break m)
     done;
     raise Not_found
   with Break i →    i
   end
```

obtemos


Why3 Interactive Proof Session
Context
Unproved goals
All goals
Strategies
Compute
Inline
Split
Provers
Alt-Ergo (1.30)
CVC3 (2.4.1)
CVC4 (1.5-prerelease)
Metis (2.3)
Spass (3.9)
Vampire (4.1)
Yices (2.5.2)
Yices (Yices)
Z3 (4.5.1)
Tools
Edit
Replay
Remove
Clean
Proof monitoring
Waiting: 0
Scheduled: 0
Running: 0
Interrupt
Theories/Goals
Status
Time
binary_search_int.mlw 0.08
BinarySearch 0.08
VC for binary_search 0.08
split_goal_wp 0.08
1. loop invariant init 0.00
2. loop invariant init 0.00
3. assertion 0.02
4. index in array bounds 0.00
5. loop invariant preservation 0.00
6. loop invariant preservation 0.02
7. loop variant decrease 0.00
8. index in array bounds 0.00
9. loop invariant preservation 0.00
10. loop invariant preservation 0.02
11. loop variant decrease 0.00
12. postcondition 0.00
13. exceptional postcondition 0.00
Source code
Task
Edited proof
Prover Output
Counter-example
file: binary_search_int/../binary_search_int.mlw
(* Binary search
A classical example. Searches a sorted array for a given value v. *)
module BinarySearch
use import int.Int
use import int.ComputerDivision
use import ref.Ref
use import array.Array
(* the code and its specification *)
exception Break int (* raised to exit the loop *)
exception Not_found (* raised to signal a search failure *)
let binary_search (a : array int) (v : int)
requires { forall i1 i2 : int. 0 <= i1 <= i2 < length a -> a[i1] <= a[i2] }
ensures { 0 <= result < length a /\ a[result] = v }
raises { Not_found -> forall i:int. 0 <= i < length a -> a[i] <> v }
= try
let l = ref 0 in
let u = ref (length a - 1) in
while !l <= !u do
invariant { 0 <= !l /\ !u < length a }
invariant {
forall i : int. 0 <= i < length a -> a[i] = v -> !l <= i <= !u }
variant { !u - !l }
let m = div (!u + !l) 2 in
assert { !l <= m <= !u };
if a[m] < v then
l := m + 1
else if a[m] > v then
u := m - 1
else
raise (Break m)
done;
raise Not_found
with Break i ->
i
end
end

Vous chantiez ? j'en suis fort aise.
Eh bien : dansez maintenant.

Problema:

Sejam y e z dois inteiros naturais (arbitrariamente grandes).
Seja $ds : \mathbb{N} \to \mathbb{N}$ a função que calcula a soma dos dígitos do inteiro parâmetro.

Encontrar o menor $x \in \mathbb{N}$ tal que $ds(x) = y$ e $x > z$.

$z = 1324165332710, \quad y = 38, \quad x = 1324165332800$
$z = 763453, \quad y = 13, \quad x = 800005$
$z = 12732738716543, \quad y = 345, \quad x = 39999999999999 \cdots 9999999999999$

(Alguns dos) Fatos:

- $0 \leq ds(a_1 \cdots a_n) \leq 9 \times n$.
- Menor natural com a soma dos seus dígitos (ds) igual a v

$$(v \ mod \ 9)\underbrace{9 \cdots 9}_{v/9}$$

- $A \triangleq aa_1 \cdots a_n \ \in \mathbb{N}$ (com $0 \leq a < 9$) e $S \in \mathbb{N}$.
  Se $9 \times n < S \leq 9 \times (n+1)$ então são precisos pelo menos $(n+1)$ algarismos para um inteiro com esta ds.
  $B \triangleq bb_1 \cdots b_n$ o menor inteiro tal que $ds(B) = S \wedge B > A$.
  Então $b > a$ e $b_1 \cdots b_n$ é o menor inteiro de ds $S - b$.

Ideia

$A \triangleq a_1 \cdots a_n$, a soma pretendida é S.
Encontrar o m "***mais a direita possível***" e o menor b tal que

$$a_1 \cdots a_{m-1} a_m a_{m+1} \cdots a_n$$

$$a_m < b \leq 9$$

e

$$ds(a_1 \cdots a_{m-1}) = S' \quad \wedge \quad 0 \leq S - (S' + b) \leq (n - m) \times 9$$

EUREKA!

$$\underbrace{a_1 \cdots a_{m-1}}_{S'} \underbrace{b}_{b} \underbrace{\square_{m+1} \cdots \square_n}_{S-(S'+b)}$$

OCaml - soma de dígitos de um inteiro

```
let ys = Sys.argv.(1) (* y as a string *)
let zs = Sys.argv.(2) (* z as a string *)
let n = String.length zs
let y = int_of_string ys
(* y/9 = length of the smallest number
whose digit sum is y*)
let max_digits = 1 + max n (y / 9)

(* x - the computed digits sequence,
   in reverse order. Initial value: z *)
let x = Array.create max_digits 0
for i = 0 to n - 1 do
  x.(n - 1 - i) <-
   Char.code zs.[i] - Char.code '0'
done

let s = ref 0 in
  for i = 0 to max_digits - 1 do
       s := !s + x.(i) done;
  for d = 0 to max_digits - 1 do
  (* s is the sum of digits d..n-1 *)
  (* solution with digits > d intacts,
  and digit d increased by 1 or more *)
    for c = x.(d) + 1 to 9 do
  (* delta = difference between y and
  s when substituting digit x.(d) by c*)
    let delta = y - !s - c + x.(d) in
   (* if delta is positive and <= 9 * d,
    then c is the new value for x.(d)*)
    if 0 <= delta && delta <= 9 * d
    then begin
     x.(d) <- c;
     let k = delta / 9 in
       for i = 0 to d-1 do
         x.(i) <- if i < k then 9
                  else if i = k then
                    delta mod 9
                  else 0
       done;
       for i = max d (n-1) downto 0 do
        print_int x.(i) done;
        print_newline (); exit 0
    end
  done;
  s := !s - x.(d)
done
```

Why3 - soma de dígitos de um inteiro

```
(* 1. Safety: we only prove that array access are within bounds
   (and termination, implicitely proved since we only have for loops) *)

let search_safety () =
  requires { length x = m }
  ensures { true }
  raises { Success -> true }
  'Init:
  let s = ref 0 in
  for i = 0 to m - 1 do
    s := !s + x[i]
  done;
  for d = 0 to m - 1 do
    invariant { length x = m }
    for c = x[d] + 1 to 9 do
      invariant { length x = m }
      let delta = y - !s - c + x[d] in
      if 0 <= delta && delta <= 9 * d
      then begin
       x[d] <- c;
       let k = div delta 9 in
       for i = 0 to d - 1 do
         invariant { length x = m }
         if i < k then x[i] <- 9
         else if i = k
          then x[i] <- mod delta 9
          else x[i] <- 0
       done;
       raise Success
      end
    done;
    s := !s - x[d]
  done
```

```
(* x[0..m-1] is a well-formed integer i.e. has digits in 0..9 *)
predicate is_integer (x : M.map int int) =
  forall k : int. 0 <= k < m -> 0 <= M.get x k <= 9
```

```
(* x1 > x2 since x1[d] > x2[d] and x1[d+1..m-1] = x2[d+1..m-1] *)
predicate gt_digit (x1 x2 : M.map int int) (d : int) =
  is_integer x1 /\ is_integer x2 /\ 0 <= d < m /\
  M.get x1 d > M.get x2 d
  /\ forall k : int. d < k < m -> M.get x1 k = M.get x2 k
```

$$sum\ x.elts\ i\ j\ = x[i] + x[i+1] + \cdots + x[j-1]$$

Why3 - Soma de dígitos de um inteiro

```
(* 2. Correctness, part 1: when Success is raised, x contains an integer
      with digit sum y *)
let search () =
  requires { length x = m /\
                  is_integer x.elts }
  ensures { true }
  raises { Success -> is_integer x.elts
          /\ sum x.elts 0 m = y }
  'Init:
  let s = ref 0 in
  for i = 0 to m - 1 do
    invariant { !s = sum x.elts 0 i }
    s := !s + x[i]
  done;
  assert { !s = sum x.elts 0 m };
  for d = 0 to m - 1 do
    invariant {
      x = (at x 'Init) /\
      !s = sum x.elts d m }
    for c = x[d] + 1 to 9 do
      invariant { x = (at x 'Init) }
      let delta = y - !s - c + x[d] in
        if 0 <= delta && delta <= 9 * d
        then begin
          x[d] <- c;
          assert {sum x.elts d m = y-delta};
          let k = div delta 9 in
          assert { k <= d };
          for i = 0 to d - 1 do
            invariant { length x = m /\
            is_integer x.elts /\
            sum x.elts d m = y - delta /\
            sum x.elts 0 i =
            if i <= k then 9*i else delta}
          if i < k then x[i] <- 9
          else if i = k
          then x[i] <- (mod delta 9)
          else x[i] <- 0
          done;
          assert {sum x.elts 0 d = delta};
          raise Success
        end
    done;
    s := !s - x[d]
  done
```

$$interp\ x\ i\ j = \text{ o inteiro constituído pelos algarismos } x[j-1]\cdots x[i]$$

$$interp9\ x\ i\ j = power\ 10\ i \times (interp\ x\ i\ j + 1) - 1 \quad (i.e.\ x[j-1]\cdots x[i]9\cdots 9)$$

$$sum\_digit\ n = \begin{cases} \text{se } n \leq 0 \text{ então a soma é } 0 \\ \text{senão é } sum\_digits\ (div\ n\ 10) + mod\ n\ 10 \end{cases}$$

$$smallest\ 20 = [2;9;9] \qquad smallest\_size\ 20 = 3$$

$$\forall y : int.y \geq 0 \rightarrow sum\ (smallest\ y)\ 0\ (smallest\_size\ y) = y$$

Why3 - Soma de dígitos de um inteiro

```
(* 3. Correctness, part 2: we now prove that, on success, x contains the
   smallest integer > old(x) with digit sum y    and
   4. Completeness: we always raise the Success exception *)

let search_smallest () =
 requires { length x = m /\ is_integer x.elts /\
   (* m= max_digit, x has at most n digits *)
   forall k : int. n <= k < m -> x[k] = 0
 }
 ensures { false }
 raises { Success -> is_integer x.elts /\ sum x.elts 0 m = y /\
             interp x.elts 0 m > interp (old x.elts) 0 m /\
             forall u : int. interp (old x.elts) 0 m < u < interp x.elts 0 m ->
                          sum_digits u <> y }
  'Init:
  let s = ref 0 in
  for i = 0 to m - 1 do
    invariant { !s = sum x.elts 0 i }
    s := !s + x[i]
  done;
  assert { !s = sum x.elts 0 m };
```

Why3 - Soma de dígitos de um inteiro

```
for d = 0 to m - 1 do
  invariant {
   x = (at x 'Init) /\
   !s = sum x.elts d m /\
   forall u : int.
     interp (at x.elts 'Init) 0 m < u <= interp9 x.elts d m
                      -> sum_digits u <> y
  }
   for c = x[d] + 1 to 9 do
     invariant {
      x = (at x 'Init) /\
      forall c' : int. x[d] < c' < c ->
       forall u : int.
       interp (at x.elts 'Init) 0 m < u <= interp9 (M.set x.elts d c') d m ->
       sum_digits u <> y }
     let delta = y - !s - c + x[d] in
     if 0 <= delta && delta <= 9 * d then begin
       assert { smallest_size delta <= d };
       x[d] <- c;
       assert { sum x.elts d m = y - delta };
       assert { gt_digit x.elts (at x.elts 'Init) d };
```

```
      let k = div delta 9 in
      assert { k <= d };
      for i = 0 to d - 1 do
        invariant {
         length x = m /\ is_integer x.elts /\
         sum x.elts d m = y - delta /\
         sum x.elts 0 i = (if i <= k then 9*i else delta) /\
         (forall j : int. 0 <= j < i ->
           (j <  smallest_size delta -> x[j] = M.get (smallest delta) j) /\
           (j >= smallest_size delta -> x[j] = 0)) /\
           gt_digit x.elts (at x.elts 'Init) d }
        if i < k then x[i] <- 9
        else if i = k then x[i] <- mod delta 9
        else x[i] <- 0;
        assert { is_integer x.elts }
      done;
      assert { sum x.elts 0 d = delta };
      assert { interp x.elts 0 d = interp (smallest delta) 0 d };
      raise Success
    end
  done;
  s := !s - x[d]
done
```

Why3 como backend de verificação

- **Como verificar programas não estruturados?**

- Como verificar programas ARM?
- Como verificar o custo computacional de um programa ARM?

Complexity Checking via verificação dedutiva - Vantagens?

- Permite capturar os custos de programas expressivos
- Pode servir de certificado de custo (para além do custo em si)

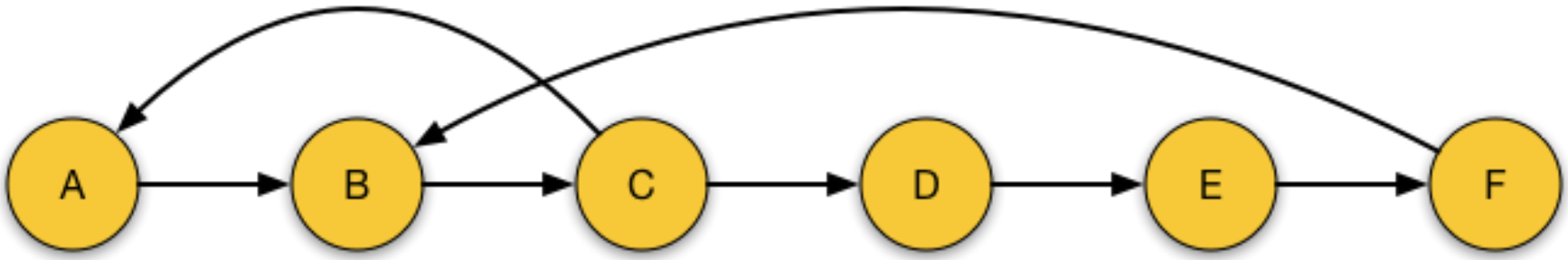
- ARM = Advanced Risc Machine
- Popular em sistemas embebidos e em plataformas móveis

ARM7:

- Arquitetura 32 bits
- 37 registos. Deles, 31 de uso comum
- 1 CPSR (Current Program State Register)
- 5 SPSR (Saved Program Status Registers)

```
mov r0, r3
cmp r0, r1
addge r0, r0, r2
```

- Como adaptar a lógica de Hoare (e respetivo WPC) a CFGs como este?
- Onde colocar asserções, invariantes?



Processo de sequencialização - um exemplo

```
.L0:
   sub r0, r0, #1
.L1:
   sub r0, r0, #2
   cmp r0, #0
   bgt .L0
   add r0, r0, #1
   cmp r0, #0
   bgt .L1
```

```
   Pre
.L0:
   sub r0, r0, #1
.L1:
   sub r0, r0, #2
   I1
   cmp r0, #0
   bgt .L0
   add r0, r0, #1
   I2
   cmp r0, #0
   bgt .L1
   Post
```

Processo de sequencialização - um exemplo

$seq_1 \triangleq$
```
assume Pre
sub r0, r0, #1
sub r0, r0, #1
assert l1
```

$seq_2 \triangleq$
```
assume l1
cmp r0, #0
assume cmp > 0
sub r0, r0, #1
sub r0, r0, #2
assert l1
```

$seq_3 \triangleq$
```
assume l1
cmp r0, #0
assume cmp <= 0
add r0, r0, #1
assert l2
```

$seq_4 \triangleq$
```
assume l2
cmp r0, #0
assume cmp > 0
sub r0, r0, #2
assert l1
```

$seq_5 \triangleq$
```
assume l2
cmp r0, #0
assume cmp <= 0
assert Post
```

Propriedade de correção do método

Se os programas sequenciais anotados são provados corretos, então o programa anotado original global é correto

Ferramenta de suporte às provas de programas ARM - Army

- Foi implementada uma ferramenta que implementa o método: **ARMY**
  - Input: um programa ARM anotado
  - Output: um conjuntos de programas sequenciais (sem instruções de salto)
- Mas, como usar o $\mathrm{why3}$ para demonstrar corretos estes programas sequenciais?
- Como fazer para que o $\mathrm{why3}$ entenda ARM?

Ferramenta de suporte às provas de programas ARM - Army

Solução? $\Longrightarrow$

- Extender a sintaxe do why3
- Definir modelo de execução e de memória + Embeber a semântica dos *opcodes*

do ARM no why3.

A ferramenta de suporte gera os programas num formato adequado para o why3: **ARMY : ARM in whY**

```
val cmp (c: cond) (rd: register) (s_o: shifter_operand) :
      unit reads {registers.contents}
           writes {cmp_result.contents}
ensures {
  (cmp_result.contents =
     (!registers)[rd] - ((addressing_mode_1 s_o registers)) ↔
     condition_passed c !cpsr) }
```

O custo como uma propriedade funcional

$$s_1; s_2; \ldots; s_n$$

$$cost = \sum_{i=1}^{n} cost_i \ + \ cost_0$$

$$\{P\} s_1; s_2; \ldots; s_n \{Q\}$$

$$\Downarrow$$

$$\{P \wedge cost = cost_0\} s_1; s_2; \ldots; s_n \{Q \wedge cost = \sum_{i=1}^{n} cost_i \ + \ cost_0\}$$

semântica de ldr com o modelo de custo computacional

```
type arm_cycles_counter 'a model{mutable n: 'a;
  mutable s: 'a; mutable i: 'a}
val ghost counter: arm_cycles_counter int
⋮
val ldr (c: cond) (rd: register) (m: addressing mode):
 unit reads {memory.contents, registers.contents,
             counter.s, counter.n, counter.i}
      writes{registers.contents, counter.s, counter.n,
             counter.i}
ensures {
 (!registers=(old !registers)
    [rd←!memory[old(addressing_mode_2 a registers)]]
    ↔ (condition_passed c !cpsr)) ⋀
 (counter.s = old counter.s + 1) ⋀
 (counter.n = old counter.n + 1) ⋀
 (counter.i = old counter.i + 1) }
```

Fatorial via Turing

```
let routine (n:int) =
int r := 0
int u := 1
while r < n do
    int s := 1
    int v = u
    while s <= r do
        u := u + v
        s := s + 1
    done
    r := r + 1
done
return u
```

```
routine:
    mov r2, #0
    mov r3, #1
    b L2
L5: mov r4, #1
    mov r5, r3
    b L3
L4: add r3, r5, r3
    add r4, r4, #1
L3: cmp r4, r2
    ble L4
    add r2, r2, #1
L2: cmp r2, r1
    blt L5
    mov r0, r3
```

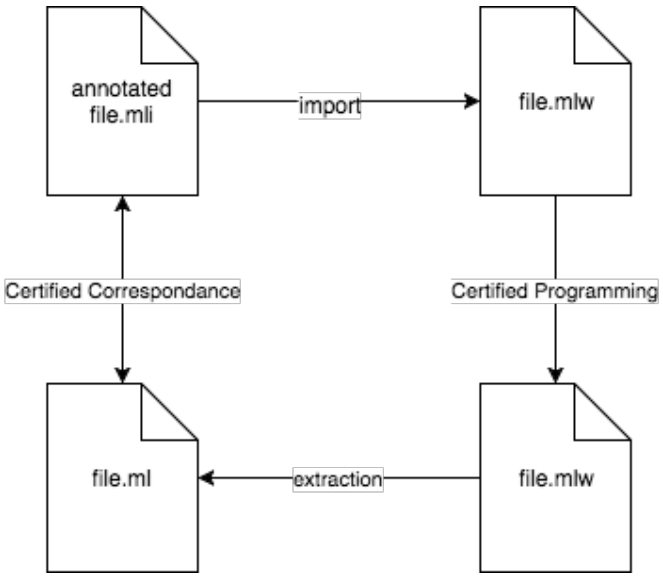
```
routine:
    @c r1 ≥ 0 c@
    @cc counter.s = 0 cc@
    mov r2, #0
    mov r3, #1
    b L2
L5: mov r4, #1
    mov r5, r3
    b L3
L4: add r3, r5, r3
    add r4, r4, #1
L3: @c 1 ≤ r4 ≤ r2 + 1∧ r2 < r1 c@
    @b r3 = r4 ∗ fact r2 ∧ r5 = fact r2 b@
    @cc 2 ∗ counter.s = 5 ∗ (r2 ∗ (r2 − 1)) +
      10 ∗ r4 + 8 + 18 ∗ r2 cc@
    cmp r4, r2
    ble L4
    add r2, r2, #1
L2: @c 0 ≤ r2 ≤ r1 c@
    @b r3 = fact r2 b@
    @cc 2 ∗ counter.s = 5 ∗ (r2 ∗ (r2 − 1)) +
      18 ∗ r2 + 6 cc@
    cmp r2, r1
    blt L5
    mov r0, r3
  @a r0 = fact r1 ∧ 2 ∗ counter.s = 5 ∗ (r1
    ∗ (r1 − 1)) + 18 ∗ r1 + 12 a@
```

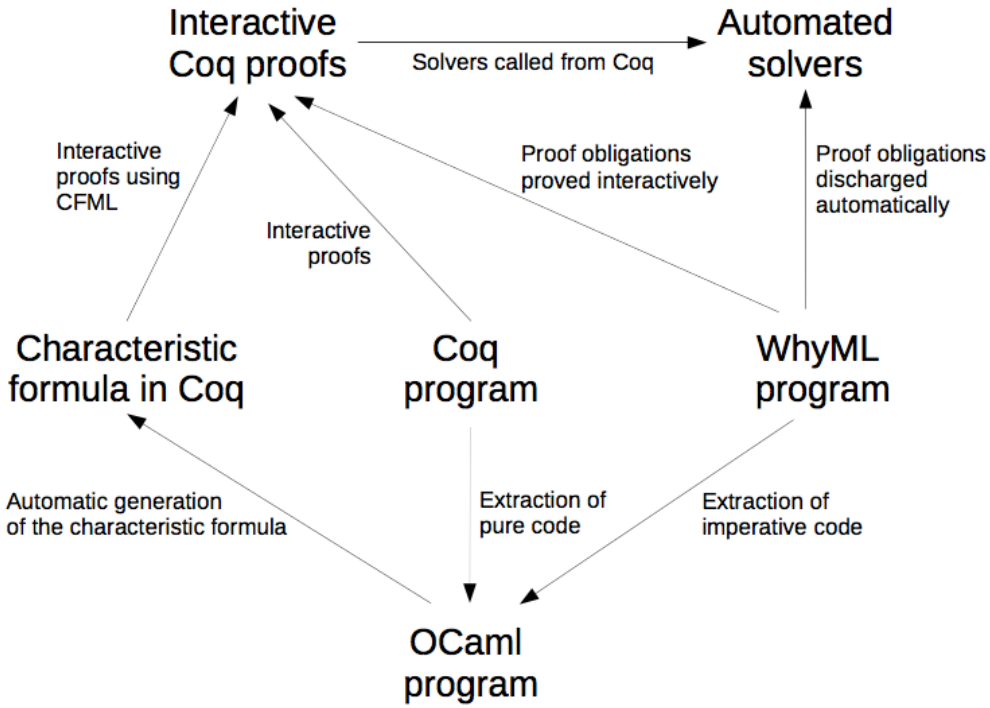
conclusão

ficou muito por apresentar

como o why3 controla fenómenos de aliasing, calcula condições de verificação, modela apontadores ou a heap, como calcula com vírgula flutuante, etc.

como se estrutura a API do why3, etc.





Esta apresentação

Programar bem = um exercício criativo que se socorre de argumentos rigorosos claros e bem estabelecidos.

A tecnologia já permite um suporte computacional!

no futuro...?

Uma plataforma integrada para
programação + **verificação** + **análise de complexidade**

O que Xanana Gusmão disse, e o que eu percebi...

Se soubesse, em vez de brincar aos computadores, teria aprendido matemática.