Why: VCGen multi-linguagem, multi-demonstrador

why(3) é uma ferramenta que produz condições de verificação a partir de programas anotados:

- podem ser em Java, ML ou C
- os demonstradores podem ser

  automáticos: Simplify, Yices, Alt-ergo, CVC3, Z3 etc.

  interactivos: Coq, Isabelle, Mizar, etc.
- Satisfaz o critério de Bruijn: uma vez demonstradas as obrigações de prova, uma demonstração de que o programa satisfaz as especificações é construída e verificado o seu tipo, automaticamente.
- why traduz o programa anotado num programa funcional anotado que incluí o modelo de memória da linguagem imperativa/objectos.
- `why.lri.fr` e `why3.lri.fr`

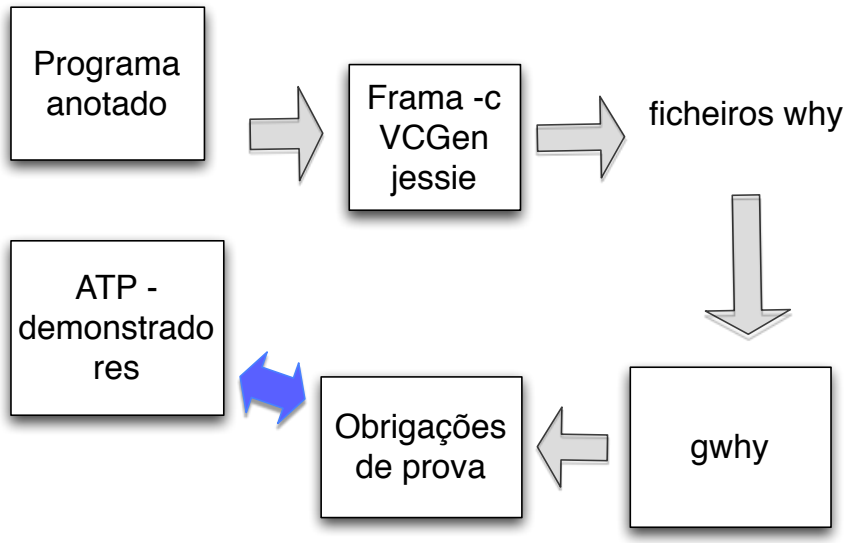
Frama-C

Ambiente integrado para a análise de programas fonte em C. `frama-c.cea.fr`

Inclui vários *plugins* entre eles *Jessie* o VCGen para C baseado no `why`. Os programas são anotados com a linguagem ACSL (inspirada no JML) mas orientada para:

- Verificação estática
- Verificação dedutiva

Frama-C com ATP



Um linguagem de especificação

- ACSL (ANSI/ISO **C** Specification Language).
- Uma linguagem de anotação para programas em **C**.
  - Cada função em **C** é anotada com uma especificação ACSL - o contracto da função.
  - A verificação de um programa consistindo num conjunto de funções é modular: assumindo que o resto das funções é correcto verificamos a correcção de uma função relativamente ao seu contracto.
  - Um programa é correcto se todas as suas funções o forem.
  - As funções podem ser anotadas com condições *frame* e os ciclos com variantes e invariantes.

ACSL:linguagem de especificação para o ANSI C

Anotações entre /*@ */ ou //@

Expressões lógicas: expressões C com alguns constructores novos: com um *backslash* antes: `forall`, `exists`, `c?e1:e2`

Tipos: de C ou lógicos: `integer`, `real` and `boolean`. Há coersão para os tipos de C.

Contractos funcionais: `requires`,`ensures`,`assigns`, `assumes`,`behavior`,`decreases`, `terminates`

Etiquetas lógicas: com um *backslash* antes: `old`,`result`, `at(e,id)`

Anotações de comandos: `loop invariant`, `assert`, `loop variant`

Jessie

Permite a verificação de

- Propriedades de segurança: limites de valores (overflows, etc); integridade das referências a variáveis indexadas e apontadores
- Propriedades funcionais dos programas

`frama-c -jessie max.c`

(* `frama-c -jessie -jessie-atp alt-ergo max.c` *)

(* `frama-c -jessie -jessie-gui max.c` *) (* obsoloetas? *)

Linguagem de especificação ACSL

As condições de verificação são agrupadas em

Segurança

Funcional Comportamento por omissão

Comportamentos normais `behavior`

Comportamentos definidos pelo utilizador

Contratos funcionais

```
  /*@ requires P ;
@ behavior b1 :
@ assumes A1 ;
@ requires R1 ;
@ assigns L1 ;
@ ensures E1 ;
@ behavior b2 :
@ assumes A2 ;
@ requires R2 ;
@ assigns L2 ;
@ ensures E2 ;
@*/
```

Contratos funcionais

A semântica é:

- A chamada da função tem de ser feita num estado tal que P & & $(A_1 \Rightarrow R_1)$ & & $(A_2 \Rightarrow R_2)$ se verifica.
- A função retorna um estado tal que $\backslash old(A_i) \Rightarrow E_i$ para cada i.
- para cada i, se a função é chamada num estado em que A_i se verifica, toda a memória alocada nesse estado e que não está em L_i fica alocada e com os valores que tinha no estado final.

Verificação de Segurança

Memória validade dos acessos a memória alocada

Inteiros verifica a não existência de *overflows* e que as operações aritméticas são executadas correctamente (não existência de divisão por zero).

Terminação

`pragma JessieIntegerModel(math)` (ou `exact`) com este pragma só é necessário verificar a memória porque supõe-se que os inteiros têm precisão infinita.

Outros valores:

- `strict` com precisão limitada e para cada operação é nec. garantir que não há overflow
- `modulo` modela os processadores reais com operações modulo 2^n.

Exemplo: pesquisa binária

```
//@ requires n >= 0 && \valid_range(t,0,n-1);
int binary_search(int* t, int n, int v) {
int l = 0, u = n-1, p = -1;
//@ loop invariant 0 <= l && u <= n-1;
```

É necessário (pré-condição) que `t` seja válido entre 0 e $n-1$ e que $n \geq 0$.No invariante de ciclo `l` e `u` devem ser estar entre 0 e $n-1$.

Segurança de *overflow* de inteiros

Sem o pragma anterior (corresponde a `strict`)

é necessário garantir que $l+u$ não ultrapassa os limites de um inteiro de 32 bits.

Neste caso basta mudar a instrução:

`int m = (l + u) / 2;`

para

`int m = l + (u - l) / 2;`

Verificação funcional

Exemplo: pesquisa binária

O resultado tem de ser um índice entre os definidos ou -1:

`@   ensures -1 <= \result <= n-1;`

Para demonstrar isto é necessário fortalecer o invariante de ciclo:

`@ 0 <= l && u <= n -1 &&  -1 <= p <= n-1;` Neste caso deve-se ainda dividir o comportamento em dois: um quando há sucesso (v está em t) outro quando há falhanço (v não está em t).

`maxarray` em C com anotações ACSL

```
int size, u[],max;

/*@ requires size >=1 && \valid_range(u,0,size-1);
  @ ensures 0 <= max < size &&
  @    (\forall int a; 0 <= a < size ==> u[a] <= u [max]);
  @*/
void maxarray() {
  int i = 1;
  max = 0;

  /*@ loop invariant
    @    1 <= i <= size && 0 <= max < i &&
    @    (\forall int a; 0 <= a < i ==> u[a] <= u[max]);
    @ loop variant
    @    size-i;
    @*/
  while (i < size) {
    if (u[i] > u[max]) max = i;
    i = i+1;
  }
}
```

Exemplo - maxarray

- Tanto o input `(size,u)` como o output `(max)` são variáveis globais.
- As linhas (comentadas) que antecedem a função, contêm:
  - a pré-condição da função assinalada por `require`

  - a pós-condição da função assinalada por `ensures`

- As linhas (comentadas) que antecedem o ciclo while, contêm o variante e invariante do ciclo.
- A condição `valid_range` é usada para definir uma pré-condição de segurança.

Predicados e funções lógicas

- Podemos usar predicados e funções no nível lógico.
- Em ACSL, declarações lógicas são usadas para definir axiomatizações.
- Uma axiomatização define tipos, funções e predicados declarando os respectivos axiomas.
- Vamos considerar como exemplo, uma função para tabelar factoriais, para a qual definimos uma axiomatização da função factorial.
- A axiomatização da função factorial contém
  - `predicate isfact(integer n, integer r);`
  - `logic integer fact(integer n);`

ACSL - predicados e funções lógicas

- `predicate`, `logic`, `axiom`, `lemma`
- predicados indutivos: `inductive`
- axiomatização: `axiomatic`
- tipos polimórficos : ex. `type list <A>= Nil | Cons(A,list<A>)`
- definições recursivas de funções
- construções lógicas de ordem superior: `\lambda`, funções de agregação `\max,\min,\sum,\product,\numof`
- definições híbridas: com argumentos de tipos lógicos e do C
- módulos

ACSL - predicados e funções

```
//@ predicate is_positive(integer x)= x>0;
/*@ logic integer(real x) =
@ x>0.0 : 1 ? (x<0.0 : -1: 0);
@*/
```

ACSL - predicados indutivos

```
/*@ inductive P (x1 , . . . , xn ) {
@ case c1 : p1 ;
...
@ case ck : pk ;
@ }
@*/
```

Onde cada `ci` é um identificador e `pi` uma proposição. A semântica é `P` ser o menor ponto fixo dos casos, i.e o menor predicado que os verifica. Para que exista, os `pi` podem ser p.e. claúsulas de Horn.

ACSL - predicados indutivos

O seguinte predicado define o máximo divisor comum entre dois inteiros:

```
/*@ inductive is_gcd(integer a, integer b, integer d) {
@ case gcd_zero:
@ \forall integer n; is_gcd(n,0,n)
@ case gcd_succ:
@  \forall integer a,b,d; is_gcd(b, a % b, d)==> is_gcd(a,b,d)
@ }
@*/
```