Deductive Proof using Frama-C

Allan Blanchard (allan.blanchard@cea.fr)

December 4, 2025 @ MPRI

CEA-List, Université Paris-Saclay, Software Safety and Security Lab

Before we start

- 6 lectures
  - incl. one quick test
  - incl. one project
- 1 exam

SOFTWARE HORROR STORIES

Software is hard. — DONALD KNUTH

...

- 1996: *Ariane 5* explodes — erroneous *float-to-int* conversion
- 1999: *Mars Climate Orbiter* explodes — error in units

...

SOFTWARE HORROR STORIES

Software is hard. — DONALD KNUTH

...

- 1996: *Ariane 5* explodes — erroneous *float-to-int* conversion
- 1999: *Mars Climate Orbiter* explodes — error in units

...

- 2006: *Debian SSH bug* — predictible RNG (fixed in 2008)
- 2012: *Heartbleed* — out of bounds access (fixed in 2014)
- 1989: *Shellshock* — insufficient input controls (fixed in 2014)

...

Example with binary search

A very simple program, and a famous bug ...

- 1946, first algorithm
- 1960, first complete algorithm

EXAMPLE WITH BINARY SEARCH

A very simple program, and a famous bug ...

- 1946, first algorithm
- 1960, first complete algorithm
- 2006, *Nearly All Binary Searches and Mergesorts are Broken*

EXAMPLE WITH BINARY SEARCH

A very simple program, and a famous bug ...

- 1946, first algorithm
- 1960, first complete algorithm
- 2006, *Nearly All Binary Searches and Mergesorts are Broken*

For example, in JDK:

```
1 int mid = (low + high) / 2;
2 int midVal = a[mid];
```

EXAMPLE WITH BINARY SEARCH

A very simple program, and a famous bug ...

- 1946, first algorithm
- 1960, first complete algorithm
- 2006, *Nearly All Binary Searches and Mergesorts are Broken*

For example, in JDK:

```
1 int mid = (low + high) / 2;
2 int midVal = a[mid];
```

Possible fix:

```
1 int mid = low + (high - low) / 2;
2 int midVal = a[mid];
```

WHAT DO WE WANT?

Ideally, we would like to guarantee that programs have no errors

- informally: “it does what we expect”
- formally: we’ll see

Is it actually possible?

WHAT DO WE WANT?

Ideally, we would like to guarantee that programs have no errors

- informally: “it does what we expect”
- formally: we’ll see

Is it actually possible? No (sorry)

PERFECT TOOL

Which perfect tool to guarantee that a program P satisfies its specification S?

1. We give *P* and *S* to a tool *T*,
2. we type “enter” (the analysis is full automatic),
3. *T* analyzes *P* and *S* without executing it *P*,
4. *T* instantly gives an answer:
   - yes: *P* satisfies *S*,
   - no: there is a problem at location *l* in *P* wrt to *S*.

Which perfect tool to guarantee that a program P satisfies its specification S?

1. We give P and S to a tool T,
2. we type “enter” (the analysis is full automatic),
3. T analyzes P and S without executing it P,
4. T instantly gives an answer:
   - yes: P satisfies S,
   - no: there is a problem at location l in P wrt to S.

The Universe does not like this

Thus, it is not possible

Rice Theorem

All non-trivial semantic properties of programs are undecidable.

We cannot have a perfect tool :(

REALITY IS SAD

Rice Theorem

All non-trivial semantic properties of programs are undecidable.

We cannot have a perfect tool :(

But there are different tradeoffs

EXAMPLE OF TRADEOFFS

Abandon “static” and “exhaustive”

We get testing

Abandon termination

We get model checking

Abandon “exact”

We get abstract interpretation

Abandon “automatic”

We get deductive verification (and we will work on that)

1 ⟩ provide a specification, a mathematic model
2 ⟩ build a proof that the code correspond to the specification

1 ⟩ provide a specification, a mathematic model
2 ⟩ build a proof that the code correspond to the specification

First program proof: Alan Turing, 1949

```
1 u := 1
2 for r = 0 to n - 1 do
3   v := u
4   for s = 1 to r do
5     u := u + v
```

1 ⟩ provide a specification, a mathematic model
2 ⟩ build a proof that the code correspond to the specification

First program proof: Alan Turing, 1949

Theoretical foundation: Floyd-Hoare logic, 1969

Deductive verification

1 ⟩ provide a specification, a mathematic model
2 ⟩ build a proof that the code correspond to the specification

First program proof: Alan Turing, 1949

Theoretical foundation: Floyd-Hoare logic, 1969

First practical success: Metro 14, 1998

Tool: Atelier B, approach by refinement

SUCCESS STORIES

- A380 Flight control, 2005
  unit proofs of functional properties
  tool: Caveat, deductive verification
- Hyper-V, 2008
  tools: VCC + automatic prover Z3, deductive verification
- CompCert, 2009
  tool: Rocq, correct by construction
- seL4, 2009
  tool: Isabelle/HOL, deductive verification
- CakeML, 2016
  tool: HOL4, deductive verification

In these lectures, we will use C, for now limited to:

- integer types,
- sequences of instructions, conditional and while loops,
- simple variables.

Example

```
1 int sum = 1;
2 int count = 0;
3 while(sum <= n){
4   count++;
5   sum = sum + 2 * count + 1;
6 }
7 return count;
```

What does this program return for a given `n`?

EXAMPLE — *ISQRT*

```
1 int sum = 1;
2 int count = 0;
3 while(sum <= n){
4   count++;
5   sum = sum + 2 * count + 1;
6 }
7 return count;
```

What does this program return for a given `n`? Informal specification:

- in the end, `count` is the floor of the square root of `n`,
- for example for `n` = 42, the returned value is 6

$$\{P\}\ c\ \{Q\}$$

- P precondition (logic formula)
- c program
- Q postcondition (logic formula)

Meaning

If we execute c from a memory state that satisfies P, then either the execution diverges, or it terminates in a memory state that satisfies Q.
This is called partial correction, termination is not proved.

Valid Hoare triples:

- $\{x = 1\}\ x = x + 2;\ \{x = 3\}$
- $\{x = y\}\ x + y\ \{\texttt{result} = 2y\}$
- $\{\exists v.\ x = 4v\}\ x + 42\ \{\exists w.\ \texttt{result} = 2w\}$
- $\{\texttt{true}\}$ `while (true) { skip; }` $\{\boxed{\texttt{false}}\}$

Valid Hoare triples:

- $\{x = 1\}\ x = x + 2;\ \{x = 3\}$
- $\{x = y\}\ x + y\ \{\texttt{result} = 2y\}$
- $\{\exists v.\ x = 4v\}\ x + 42\ \{\exists w.\ \texttt{result} = 2w\}$
- $\{\texttt{true}\}$ `while (true) { skip; }` $\{\boxed{\texttt{false}}\}$

Beware!

- After this loop *any* property can be proved
- Termination might be crucial

$\{\}\ \mathit{ISQRT}\ \{\}$

Valid Hoare triples:

- $\{x = 1\}\ x = x + 2;\ \{x = 3\}$
- $\{x = y\}\ x + y\ \{\texttt{result} = 2y\}$
- $\{\exists v.\ x = 4v\}\ x + 42\ \{\exists w.\ \texttt{result} = 2w\}$
- $\{\texttt{true}\}$ `while (true) { skip; }` $\{\boxed{\texttt{false}}\}$

Beware!

- After this loop *any* property can be proved
- Termination might be crucial

$\{?\}$ *ISQRT* $\{?\}$

Valid Hoare triples:

- $\{x = 1\}\ x = x + 2;\ \{x = 3\}$
- $\{x = y\}\ x + y\ \{\texttt{result} = 2y\}$
- $\{\exists v.\ x = 4v\}\ x + 42\ \{\exists w.\ \texttt{result} = 2w\}$
- $\{\texttt{true}\}$ `while (true) { skip; }` $\{\boxed{\texttt{false}}\}$

Beware!

- After this loop *any* property can be proved
- Termination might be crucial

$$\{n \geqslant 0\}\ \mathit{ISQRT}\ \{?\}$$

Valid Hoare triples:

- $\{x = 1\}\ x = x + 2;\ \{x = 3\}$
- $\{x = y\}\ x + y\ \{\texttt{result} = 2y\}$
- $\{\exists v.\, x = 4v\}\ x + 42\ \{\exists w.\, \texttt{result} = 2w\}$
- $\{\texttt{true}\}\ \texttt{while (true) \{ skip; \}}\ \{\boxed{\texttt{false}}\}$

Beware!

- After this loop *any* property can be proved
- Termination might be crucial

$$\{n \geqslant 0\}\ \mathit{ISQRT}\ \{\texttt{result}^2 \leqslant n < (\texttt{result} + 1)^2\}$$

Hoare rules

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}$$

$$\frac{}{\{P[x \mapsto t]\}\ x = t;\ \{P\}}$$

$$\frac{\{P\}\ c_1\ \{Q\} \qquad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1\ ;\ c_2\ \{R\}}$$

$$\frac{\{P \wedge e \neq 0\}\ c_1\ \{Q\} \qquad \{P \wedge e = 0\}\ c_2\ \{Q\}}{\{P\}\ \texttt{if}\ (e)\ \{\ c_1;\ \}\ \texttt{else}\ \{\ c_2;\ \}\ \{Q\}}$$

$$\frac{\{I \wedge e \neq 0\}\ c\ \{I\}}{\{I\}\ \texttt{while}\ (e)\ \{\ c;\ \}\ \{I \wedge e = 0\}}$$

$$\frac{P \Rightarrow P' \qquad \{P'\}\ c\ \{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\ c;\ \{Q\}}$$

- These rules define a semantics, not an algorithm,

Remarks

- These rules define a semantics, not an algorithm,
- *Sequence:* how to find R?

Remarks

- These rules define a semantics, not an algorithm,
- *Sequence:* how to find R?
- *Loops:* how to find I?

Remarks

- These rules define a semantics, not an algorithm,
- *Sequence:* how to find R?
- *Loops:* how to find I?
- *Consequence:* when should we use it?

Remarks

- These rules define a semantics, not an algorithm,
- *Sequence:* how to find R?
- *Loops:* how to find I?
- *Consequence:* when should we use it?
- *Runtime errors?*

Remarks

- These rules define a semantics, not an algorithm,
- *Sequence:* how to find R?
- *Loops:* how to find I?
- *Consequence:* when should we use it?
- *Runtime errors?*
- *Assignment:* it works *backward* ...

Example

Proof of $\{x = 1\}\, x = x + 2;\, \{x = 3\}$

$$\dfrac{x = 1 \Rightarrow x + 2 = 3 \qquad \dfrac{\{(x = 3)[x \mapsto x + 2]\}\, x = x + 2;\, \{x = 3\}}{\{x + 2 = 3\}\, x = x + 2;\, \{x = 3\}}}{\{x = 1\}\, x = x + 2;\, \{x = 3\}}$$

What we want ...

An algorithm that establishes program correction

Predicate transformer (Dijkstra)

$\mathrm{WP}(c, Q)$

c program

Q postcondition

computes the weakest precondition P such that $\{P\}\ c\ \{Q\}$

$$x = 3 \star x \star y; \qquad \{\, x \text{ is even} \,\}$$

$$\{\ 3xy \text{ is even}\ \} \quad x = 3 \star x \star y; \qquad \{\ x \text{ is even}\ \}$$

Intuition

$$\{\ 3xy \text{ is even}\ \}\quad x = 3 \star x \star y;\qquad \{\ x \text{ is even}\ \}$$

$$\{\ Q[s]\ \}\quad x = s;\qquad \{\ Q[x]\ \}$$

Intuition

$\{\ 3xy \text{ is even}\ \}\quad x = 3 \star x \star y;\quad \{\ x \text{ is even}\ \}$

$\{\ Q[s]\ \}\quad x = s;\quad \{\ Q[x]\ \}$

```
{ Q }  if (e) { c1 ; }
       else { c2 ; }
```

Intuition

$$\{\ 3xy \text{ is even}\ \}\quad x = 3 \star x \star y;\quad \{\ x \text{ is even}\ \}$$

$$\{\ Q[s]\ \}\quad x = s;\quad \{\ Q[x]\ \}$$

$$\texttt{if}\ (e)\ \{\ c_1\ Q;\}\quad \texttt{else}\ \{\ c_2\ Q\ ;\}\quad \{\ Q\ \}$$

Intuition

$\{\ 3xy \text{ is even}\ \}\quad x = 3 \star x \star y;\quad \{\ x \text{ is even}\ \}$

$\{\ Q[s]\ \}\quad x = s;\quad \{\ Q[x]\ \}$

```
if (e) { P1 c1 Q;}   { Q }
else { P2 c2 Q ;}
```

$$\{\ 3xy \text{ is even}\ \}\quad x = 3 \star x \star y;\quad \{\ x \text{ is even}\ \}$$

$$\{\ Q[s]\ \}\quad x = s;\quad \{\ Q[x]\ \}$$

$$\{\ \text{if } e \text{ then } P_1 \text{ else } P_2\ \}\quad \texttt{if}\ (e)\ \{\ P_1\ c_1\ Q;\}\ \texttt{else}\ \{\ P_2\ c_2\ Q\ ;\}\quad \{\ Q\ \}$$

Intuition

$$\{\ 3xy \text{ is even}\ \}\quad x = 3 \star x \star y;\quad \{\ x \text{ is even}\ \}$$

$$\{\ Q[s]\ \}\quad x = s;\quad \{\ Q[x]\ \}$$

$$\{\ \text{if } e \text{ then } P_1 \text{ else } P_2\ \}\quad \texttt{if}\ (e)\ \{\ P_1\ c_1\ Q;\}\ \texttt{else}\ \{\ P_2\ c_2\ Q\ ;\}\quad \{\ Q\ \}$$

$$\texttt{if}\ (e)\ \{\quad c\quad ;\}\quad \{\ Q\ \}$$

$$\{\ 3xy \text{ is even}\ \}\quad x = 3 \star x \star y;\quad \{\ x \text{ is even}\ \}$$

$$\{\ Q[s]\ \}\quad x = s;\quad \{\ Q[x]\ \}$$

$$\{\ \text{if } e \text{ then } P_1 \text{ else } P_2\ \}\quad \texttt{if}\ (e)\ \{\ P_1\ c_1\ Q;\}\ \texttt{else}\ \{\ P_2\ c_2\ Q\ ;\}\quad \{\ Q\ \}$$

$$\texttt{if}\ (e)\ \{\ P\ c\ Q;\}\quad \{\ Q\ \}$$

$$\{\ 3xy \text{ is even}\ \}\quad x = 3 \star x \star y;\quad \{\ x \text{ is even}\ \}$$

$$\{\ Q[s]\ \}\quad x = s;\quad \{\ Q[x]\ \}$$

$$\{\ \text{if } e \text{ then } P_1 \text{ else } P_2\ \}\quad \texttt{if}\ (e)\ \{\ P_1\ c_1\ Q;\}\ \texttt{else}\ \{\ P_2\ c_2\ Q\ ;\}\quad \{\ Q\ \}$$

$$\{\ \text{if } e \text{ then } P \text{ else } Q\ \}\quad \texttt{if}\ (e)\ \{\ P\ c\ Q;\}\quad \{\ Q\ \}$$

$$\{\ 3xy \text{ is even}\ \}\quad x = 3 \star x \star y;\quad \{\ x \text{ is even}\ \}$$

$$\{\ Q[s]\ \}\quad x = s;\quad \{\ Q[x]\ \}$$

$$\{\ \text{if } e \text{ then } P_1 \text{ else } P_2\ \}\quad \texttt{if}\ (e)\ \{\ P_1\ c_1\ Q;\}\ \texttt{else}\ \{\ P_2\ c_2\ Q\ ;\}\quad \{\ Q\ \}$$

$$\{\ \text{if } e \text{ then } P \text{ else } Q\ \}\quad \texttt{if}\ (e)\ \{\ P\ c\ Q;\}\quad \{\ Q\ \}$$

$$\texttt{while}\ (e)\ \{\ c;\ \}\quad \{\ Q\ \}$$

$\{\ 3xy \text{ is even}\ \}$	$x = 3 \star x \star y;$	$\{\ x \text{ is even}\ \}$
$\{\ Q[s]\ \}$	$x = s;$	$\{\ Q[x]\ \}$
$\{$ if e then P_1 else P_2 $\}$	`if` $(e)\ \{\ P_1\ c_1\ Q;\}$ `else` $\{\ P_2\ c_2\ Q\ ;\}$	$\{\ Q\ \}$
$\{$ if e then P else Q $\}$	`if` $(e)\ \{\ P\ c\ Q;\}$	$\{\ Q\ \}$
?	`while` $(e)\ \{\ c;\ \}$	$\{\ Q\ \}$

WP Definition

$$\mathrm{WP}(\texttt{skip}, Q) \equiv Q$$

$$\mathrm{WP}(x = t;, Q) \equiv Q[x \mapsto t]$$

$$\mathrm{WP}(c_1 \ ; \ c_2, Q) \equiv \mathrm{WP}(c_1, \mathrm{WP}(c_2, Q))$$

$$\mathrm{WP}(\texttt{if}\ (t)\ \{\ c_1;\ \}\ \texttt{else}\ \{\ c_2;\ \}, Q) \equiv (t \to \mathrm{WP}(c_1, Q)) \wedge (\neg t \to \mathrm{WP}(c_2, Q))$$

```
if (odd q) { r = r + p;; };
p = p + p; ;
q = half q;
```

```
if (odd q) { ;}
    r = r + p;
else
    skip;
p = p + p; ;
q = half q;
```

```
if (odd q) { ;}
    r = r + p;
else
    skip;
p = p + p; ;
q = half q;
```

$Q[p, q, r]$

```
if (odd q) { ;}
    r = r + p;
else
    skip;
```

$$
\begin{array}{l}
p = p + p;\ ; \\
Q[p, \texttt{half}\ q, r] \\
q = \texttt{half}\ q; \\
Q[p, q, r]
\end{array}
$$

```
if (odd q) { ;}

    r = r + p;

else

    skip;
```

$Q[p + p, \texttt{half}\ q, r]$

$p = p + p;$;

$Q[p, \texttt{half}\ q, r]$

$q = \texttt{half}\ q;$

$Q[p, q, r]$

```
if (odd q) { ;}

    r = r + p;
  Q[p + p, half q, r]
else

    skip;
  Q[p + p, half q, r]
p = p + p; ;
Q[p, half q, r]
q = half q;
Q[p, q, r]
```

$$
\begin{array}{l}
\texttt{if (odd } q\texttt{) \{ ;\}} \\
\quad Q[p + p, \texttt{half } q, r + p] \\
\quad\quad r = r + p; \\
\quad Q[p + p, \texttt{half } q, r] \\
\texttt{else} \\
\quad Q[p + p, \texttt{half } q, r] \\
\quad\quad \texttt{skip;} \\
\quad Q[p + p, \texttt{half } q, r] \\
\quad p = p + p; \texttt{ ;} \\
Q[p, \texttt{half } q, r] \\
\quad q = \texttt{half } q; \\
Q[p, q, r]
\end{array}
$$

$$
\begin{array}{l}
\quad (\texttt{odd}\ q \to Q[p + p, \texttt{half}\ q, r + p]) \wedge \\
(\neg\ \texttt{odd}\ q \to Q[p + p, \texttt{half}\ q, r]) \\
\quad \texttt{if}\ (\texttt{odd}\ q)\ \{\ ;\} \\
\qquad Q[p + p, \texttt{half}\ q, r + p] \\
\qquad\quad r = r + p; \\
\qquad Q[p + p, \texttt{half}\ q, r] \\
\quad \texttt{else} \\
\qquad Q[p + p, \texttt{half}\ q, r] \\
\qquad\quad \texttt{skip}; \\
\qquad Q[p + p, \texttt{half}\ q, r] \\
\quad p = p + p;\ ; \\
Q[p, \texttt{half}\ q, r] \\
\quad q = \texttt{half}\ q; \\
Q[p, q, r]
\end{array}
$$

$$\mathrm{WP}(\texttt{while}\ (t)\ \{\ e;\ \},Q) \equiv$$

$\exists J$: `Prop`.	an invariant J
$J \wedge$	which is true when we start
$\forall x_1 \ldots x_k.$	and remains true
$(J \wedge\ \ t \rightarrow \mathrm{WP}(e, J)) \wedge$	after any iteration,
$(J \wedge \neg t \rightarrow Q)$	is enough to prove Q

$x_1, \ldots, x_k$ variables modified by e

We do not know the values modified after n iterations

- we have to prove Q and that J is preserved for arbitrary values
- J must give all necessary information about the memory state

WP: annotated loops

Finding an invariant is hard is the general case

- it is equivalent to proving Q by induction

We must help tools with annotations:

$$
\begin{aligned}
&\mathrm{WP}(\texttt{invariant } J \texttt{ while } (t)\ \{\ e;\ \}, Q) \equiv \\
&\quad J \wedge \\
&\quad \forall x_1 \ldots x_k. \\
&\qquad (J \wedge\ \ t \rightarrow \mathrm{WP}(e, J)) \wedge \\
&\qquad (J \wedge \neg t \rightarrow Q)
\end{aligned}
$$

the provided invariant J
is true when we start,
and remains true
after an iteration
and is enough to prove Q

$x_1, \ldots, x_k$ variables modified by e

Example

```
1 r = 0 ;
2 i = 0 ;
3 while(i < x){
4   r = r + y ;
5   i++ ;
6 }
```

Coherence

For all program c and postcondition Q, $\{\mathrm{WP}(c, Q)\}\ c\ \{Q\}$ is valid.

Proof by induction on the structure of p.

Consequence

To prove that $\{P\}\ c\ \{Q\}$ is valid, we just have to prove $P \Rightarrow \mathrm{WP}(c, Q)$.

This is what deductive verification tools do

Let's start with a simple question

How many undefined behaviors are there in the ISO C?

How many undefined behaviors are there in the ISO C?

191

LET'S START WITH A SIMPLE QUESTION

How many undefined behaviors are there in the ISO C?

191

Simple ⟶ Subtle

- Modulo 0
- Shift with negative exponent
- Strict aliasing violation
- Data-race

Spot the potential undefined behaviors

```
#define N 42

void foo(int t[N], int *p, int *q){


   int i = ++(*p) + (*q)++ ;
   t[0] = t[i]+1;
}
```

Spot the potential undefined behaviors

```
#define N 42

void foo(int t[N], int *p, int *q){
   assert(t != 0 && p != 0 && q != 0);
   assert(0 <= *p + *q + 1 < N);
   int i = ++(*p) + (*q)++ ;
   t[0] = t[i]+1;
}
```

Spot the potential undefined behaviors

- l.3 : not `NULL` is not enough

```
#define N 42

void foo(int t[N], int *p, int *q){
   assert(t != 0 && p != 0 && q != 0);
   assert(0 <= *p + *q + 1 < N);
   int i = ++(*p) + (*q)++ ;
   t[0] = t[i]+1;
}
```

Spot the potential undefined behaviors

› I.3 : not `NULL` is not enough

```
#define N 42

void foo(int t[N], int *p, int *q){
   assert(t != 0 && p != 0 && q != 0);
   assert(0 <= *p + *q + 1 < N);
   int i = ++(*p) + (*q)++ ;
   t[0] = t[i]+1;
}
```

Spot the potential undefined behaviors

```
#define N 42

void foo(int t[N], int *p, int *q){
   assert(t != 0 && p != 0 && q != 0);
   assert(0 <= *p + *q + 1 < N);
   int i = ++(*p) + (*q)++ ;
   t[0] = t[i]+1;
}
```

- l.3 : not `NULL` is not enough
- l.4 : `*p`, `*q` uninitialized
- l.4 : integer overflow/underflow

Spot the potential undefined behaviors

```
#define N 42

void foo(int t[N], int *p, int *q){
   assert(t != 0 && p != 0 && q != 0);
   assert(0 <= *p + *q + 1 < N);
   int i = ++(*p) + (*q)++ ;
   t[0] = t[i]+1;
}
```

- l.3 : not `NULL` is not enough
- l.4 : `*p`, `*q` uninitialized
- l.4 : integer overflow/underflow
- l.5 : `*p`, `*q` uninitialized
- l.5 : integer overflow/underflow
- l.5 : multiple side-effects (`p == q`)

Spot the potential undefined behaviors

```
#define N 42

void foo(int t[N], int *p, int *q){
   assert(t != 0 && p != 0 && q != 0);
   assert(0 <= *p + *q + 1 < N);
   int i = ++(*p) + (*q)++ ;
   t[0] = t[i]+1;
}
```

- l.3 : not `NULL` is not enough
- l.4 : `*p`, `*q` uninitialized
- l.4 : integer overflow/underflow
- l.5 : `*p`, `*q` uninitialized
- l.5 : integer overflow/underflow
- l.5 : multiple side-effects

Spot the potential undefined behaviors

```
#define N 42

void foo(int t[N], int *p, int *q){
   assert(t != 0 && p != 0 && q != 0);
   assert(0 <= *p + *q + 1 < N);
   int i = ++(*p) + (*q)++ ;
   t[0] = t[i]+1;
}
```

- l.3 : not `NULL` is not enough
- l.4 : `*p`, `*q` uninitialized
- l.4 : integer overflow/underflow
- l.5 : `*p`, `*q` uninitialized
- l.5 : integer overflow/underflow
- l.5 : multiple side-effects
- l.6 : `t[i]` uninitialized
- l.6 : integer overflow
- l.6 : buffer overflow

Spot the potential undefined behaviors

```
#define N 42

void foo(int t[N], int *p, int *q){
   assert(t != 0 && p != 0 && q != 0);
   assert(0 <= *p + *q + 1 < N);
   int i = ++(*p) + (*q)++ ;
   t[0] = t[i]+1;
}
```

- l.3 : not `NULL` is not enough
- l.4 : `*p`, `*q` uninitialized
- l.4 : integer overflow/underflow
- l.5 : `*p`, `*q` uninitialized
- l.5 : integer overflow/underflow
- l.5 : multiple side-effects
- l.6 : `t[i]` uninitialized
- l.6 : integer overflow
- l.6 : **buffer overflow**

Collaborative tools for analysis of C source code

- Developed at CEA-List, mostly open-source
- Current version Frama-C 31.0 Gallium
- ACSL annotation language
- Extensible plug-in oriented platform
  - collaboration of analyzes over the same code
  - inter plug-in communication through ACSL formulas

https://www.frama-c.com/

Several tools inside a single platform

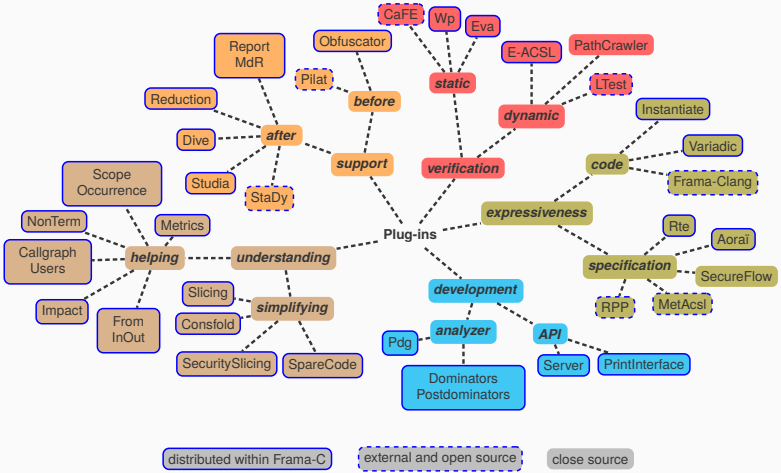
- plug-in architecture like in Eclipse
- tools provided as plug-ins
  - over 30 plug-ins in the open-source distribution
  - close-source plug-ins, either at CEA (about 10) or outside
- a common kernel
  - provides a uniform setting
  - provides general services
  - synthesizes useful information

- mostly developed in OCaml ($\approx$ 286 kloc in the open-source distribution)
- initially based on Cil [Necula et al., CC'02]
- library dedicated to analysis of C code

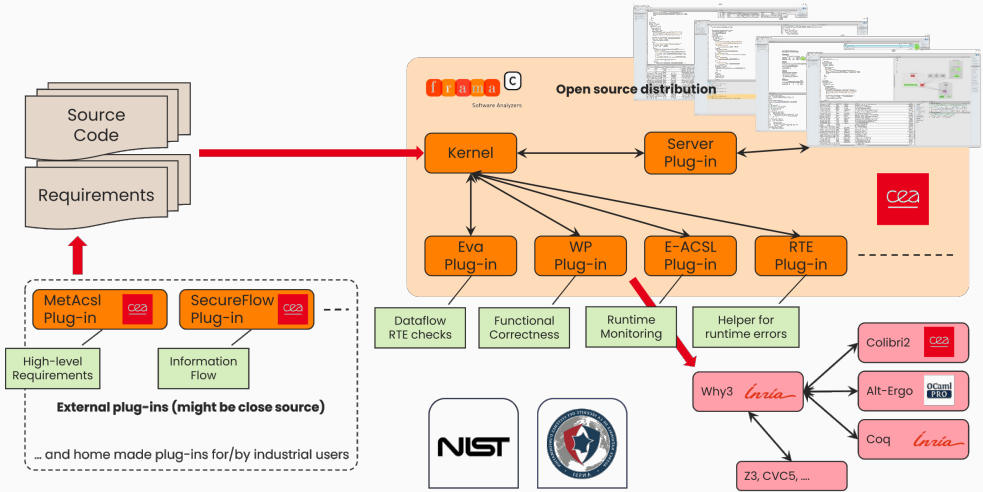
Development of plug-ins by third party

- dedicated plug-ins for specific tasks
- dedicated plug-ins for fine-grained parameterization
- extensions of existing analyzers

The Frama-C plug-ins gallery



BIG PICTURE


Source Code
Requirements
Open source distribution
Kernel
Server Plug-in
Eva Plug-in
WP Plug-in
E-ACSL Plug-in
RTE Plug-in
MetAcsl Plug-in
SecureFlow Plug-in
High-level Requirements
Information Flow
External plug-ins (might be close source)
... and home made plug-ins for/by industrial users
Dataflow RTE checks
Functional Correctness
Runtime Monitoring
Helper for runtime errors
Why3
Colibri2
Alt-Ergo
Coq
Z3, CVC5, ...
NIST

What is the role of the WP plug-in?

Deductive verification tool

- prove the functional correctness of programs
- prove absence of runtime errors
- assure conformance of a program with respect to its specification

ACSL: ANSI/ISO C Specification Language

A behavioral specification language

- Based on the notion of contract, like in Eiffel, JML
- Allows users to specify functional properties of programs
- Allows communication between various plug-ins
- Independent of a particular analysis

Basic Components

- Typed first-order logic with pure C expressions
- C types + $\mathbb{Z}$ (integer) and $\mathbb{R}$ (real)
- Built-ins predicates and logic functions, particularly over pointers

Contracts

- Goal: specification of imperative functions
- Approach: give assertions (i.e. properties) about the functions
  - Precondition is supposed to be true on entry (ensured by the caller)
  - Postcondition must be true on exit (ensured by the function)
- Nothing is guaranteed when the precondition is not satisfied
- Termination may be guaranteed or not (total or partial correctness)

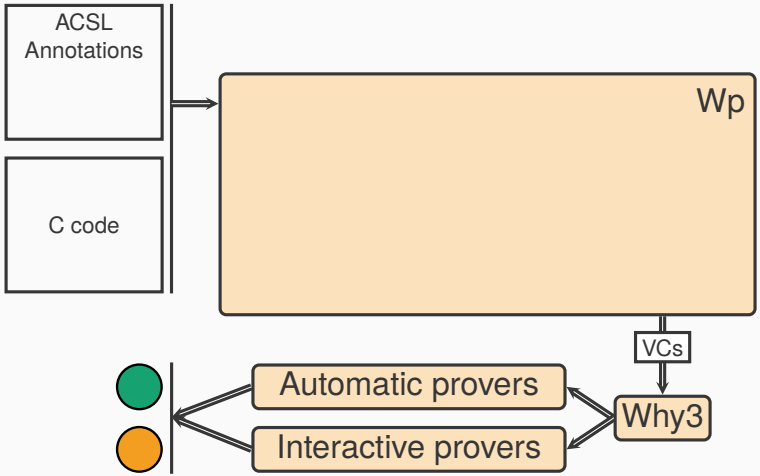
Primary role of contracts

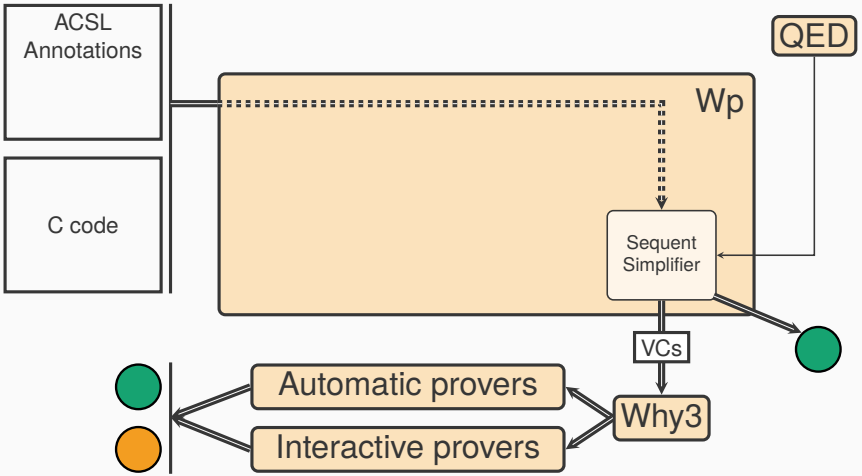
- Must reflect the informal specification
- Should not be modified just to suit the verification tasks

Basic elements of an ACSL contract

- **requires**: precondition of the function
- **ensures**: postcondition of the function

```
1 /*@ requires Bounds: -10 <= x < 10 ;
2     ensures  Result: \result == x + 1 ;
3 */
4 int add_one(int x){
5   return x + 1 ;
6 }
```


ACSL
Annotations
C code
Wp
VCs
Automatic provers
Interactive provers
Why3


ACSL
Annotations
C code
QED
Wp
Sequent
Simplifier
VCs
Automatic provers
Interactive provers
Why3

Check whether everyone can have Frama-C ...

```c
int alphabet_letter(char c){
  if( ('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z') ) return 1 ;
  else return 0 ;
}
```

```c
int day_of(int m){
  int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 } ;
  return days[m-1] ;
}
```

```c
int last_angle(int first, int second){
  return 180 - first - second ;
}
```

```c
int add(int x, int y){
  return x+y ;
}
```

```c
int distance(int a, int b){
  if(a < b) return b - a ;
  else return a - b ;
}
```

We now consider function calls.

```
1 f(ps, ...);
```

The weakest precondition rule is:

$$\begin{array}{rcl} \mathrm{WP}(f(\vec{t}), Q)) & \equiv & P_f[p_i \mapsto t_i] \\ & \wedge & \forall \vec{v}, \quad (Q_f[p_i \mapsto t_i, \mathit{here}(a_j) \mapsto v_j, \mathit{old}(a_j) \mapsto a_j] \Rightarrow Q[\mathit{here}(a_j) \mapsto v_j]) \end{array}$$

P_f	precondition of f	$\vec{a}$	lvalues modified in f
Q_f	postcondition of f	$\vec{v}$	fresh variables
$\vec{p}$	formal parameters of f		

Modular proof: when verifying a function call, we only use the contract, not its code.

How can we know which lvalues are modified?

- no simple procedure in C (even if we could get an over-approximation)
- $\rightarrow$ ask the user for this information

```
assigns v1, v2, ... vN;
```

- part of the postcondition
- specifies lvalues that are modified by the function
- nothing to say about local variables:
  - a function is authorized to modify them
  - a postcondition cannot talk about it anyway
- the `\nothing` keyword denotes an empty set

Example

Specify and prove the correction of the `swap` function:

```
1 void swap(int* p, int* q){
2   int tmp = *p ;
3   *p = *q ;
4   *q = tmp ;
5 }
```

Example

```
1 /*@ assigns *p, *q ;
2     ensures *p == \old(*q) && *q == \old(*p); */
3 void swap(int* p, int* q){
4   int tmp = *p ;
5   *p = *q ;
6   *q = tmp ;
7 }
```

Example

Apply WP rules on this example:

```
1 int a = 2;
2 int b = 4;
3 swap(&a, &b);
4 // Post: a == 4 && b == 2 ;
```

Example

$$\mathrm{WP}(\boldsymbol{p}, a = 4 \wedge b = 2) =$$
$$\mathrm{WP}(\boldsymbol{a} = \mathbf{2}, \mathrm{WP}(\boldsymbol{b} = \mathbf{4}, \mathrm{WP}(\boldsymbol{swap}(\&a, \&b), a = 4 \wedge b = 2)))$$

Example

$$\mathrm{WP}(p, a = 4 \wedge b = 2) =$$
$$\mathrm{WP}(a = 2, \mathrm{WP}(b = 4, \mathrm{WP}(\mathit{swap}(\&a, \&b), a = 4 \wedge b = 2)))$$

$$\mathrm{WP}(\mathit{swap}(\&a, \&b), Q) =$$
$$\forall v_a v_b, \quad Q_f$$
$$[p \mapsto \&a, q \mapsto \&b]$$
$$[\mathit{here}(*(\&a)) \mapsto v_a, \mathit{here}(*(\&b)) \mapsto v_b]$$
$$[\mathit{old}(*(\&a)) \mapsto *(\&a), \mathit{old}(*(\&b)) \mapsto *(\&b)]$$
$$\Rightarrow Q[\mathit{here}(*(\&a)) \mapsto v_a, \mathit{here}(*(\&b)) \mapsto v_b])$$

Example

$$\mathrm{WP}(p, a = 4 \wedge b = 2) =$$
$$\mathrm{WP}(a = 2, \mathrm{WP}(b = 4, \mathrm{WP}(swap(\&a, \&b), a = 4 \wedge b = 2)))$$

$$\mathrm{WP}(swap(\&a, \&b), Q) =$$
$$\forall v_a v_b, \quad *p = old(*q) \wedge *q = old(*p)$$
$$[p \mapsto \&a, q \mapsto \&b]$$
$$[here(*(\&a)) \mapsto v_a, here(*(\&b)) \mapsto v_b]$$
$$[old(*(\&a)) \mapsto *(\&a), old(*(\&b)) \mapsto *(\&b)]$$
$$\Rightarrow Q[here(*(\&a)) \mapsto v_a, here(*(\&b)) \mapsto v_b])$$

Example

$$\mathrm{WP}(p, a = 4 \wedge b = 2) =$$
$$\quad \mathrm{WP}(a = 2, \mathrm{WP}(b = 4, \mathrm{WP}(\mathit{swap}(\&a, \&b), a = 4 \wedge b = 2)))$$

$$\mathrm{WP}(\mathit{swap}(\&a, \&b), Q) =$$
$$\forall v_a v_b, \quad *(\&a) = \mathit{old}(*(\&b)) \wedge *(\&b) = \mathit{old}(*(\&a))$$
$$[\mathit{here}(*(\&a)) \mapsto v_a, \mathit{here}(*(\&b)) \mapsto v_b]$$
$$[\mathit{old}(*(\&a)) \mapsto *(\&a), \mathit{old}(*(\&b)) \mapsto *(\&b)]$$
$$\Rightarrow Q[\mathit{here}(*(\&a)) \mapsto v_a, \mathit{here}(*(\&b)) \mapsto v_b])$$

$$\mathrm{WP}(p, a = 4 \wedge b = 2) =$$
$$\mathrm{WP}(a = 2, \mathrm{WP}(b = 4, \mathrm{WP}(\mathit{swap}(\&a, \&b), a = 4 \wedge b = 2)))$$

$$\mathrm{WP}(\mathit{swap}(\&a, \&b), Q) =$$
$$\forall v_a v_b, \quad \mathit{here}(*(\&a)) = \mathit{old}(*(\&b)) \wedge \mathit{here}(*(\&b)) = \mathit{old}(*(\&a))$$
$$[\mathit{here}(*(\&a)) \mapsto v_a, \mathit{here}(*(\&b)) \mapsto v_b]$$
$$[\mathit{old}(*(\&a)) \mapsto *(\&a), \mathit{old}(*(\&b)) \mapsto *(\&b)]$$
$$\Rightarrow Q[\mathit{here}(*(\&a)) \mapsto v_a, \mathit{here}(*(\&b)) \mapsto v_b])$$

Example

$$\mathrm{WP}(p, a = 4 \wedge b = 2) =$$
$$\mathrm{WP}(a = 2, \mathrm{WP}(b = 4, \mathrm{WP}(swap(\&a, \&b), a = 4 \wedge b = 2)))$$

$$\mathrm{WP}(swap(\&a, \&b), Q) =$$
$$\forall v_a v_b, \quad v_a = old(*(\&b)) \wedge v_b = old(*(\&a))$$
$$[old(*(\&a)) \mapsto *(\&a), old(*(\&b)) \mapsto *(\&b)]$$
$$\Rightarrow Q[here(*(\&a)) \mapsto v_a, here(*(\&b)) \mapsto v_b])$$

Example

$$\mathrm{WP}(p, a = 4 \wedge b = 2) =$$
$$\mathrm{WP}(a = 2, \mathrm{WP}(b = 4, \mathrm{WP}(\mathit{swap}(\&a, \&b), a = 4 \wedge b = 2)))$$

$$\mathrm{WP}(\mathit{swap}(\&a, \&b), Q) =$$
$$\forall v_a v_b, \quad v_a = *(\&b) \wedge v_b = *(\&a) \Rightarrow Q[\mathit{here}(*(\&a)) \mapsto v_a, \mathit{here}(*(\&b)) \mapsto v_b])$$

$$\mathrm{WP}(p, a = 4 \wedge b = 2) =$$
$$\mathrm{WP}(a = 2, \mathrm{WP}(b = 4, \mathrm{WP}(swap(\&a, \&b), a = 4 \wedge b = 2)))$$

$$\mathrm{WP}(swap(\&a, \&b), Q) =$$
$$\forall v_a v_b, v_a = b \wedge v_b = a \Rightarrow Q[a \mapsto v_a, b \mapsto v_b]$$

Example

$$\mathrm{WP}(\mathbf{p}, a = 4 \wedge b = 2) =$$
$$\mathrm{WP}(\mathbf{a} = \mathbf{2}, \mathrm{WP}(\mathbf{b} = \mathbf{4}, \mathrm{WP}(\mathbf{swap}(\&a, \&b), a = 4 \wedge b = 2)))$$

$$\mathrm{WP}(\mathbf{swap}(\&a, \&b), Q) =$$
$$\forall v_a v_b, v_a = b \wedge v_b = a \Rightarrow (a = 4 \wedge b = 2)[a \mapsto v_a, b \mapsto v_b]$$

Example

$$\mathrm{WP}(\mathbf{p}, a = 4 \wedge b = 2) =$$
$$\mathrm{WP}(\mathbf{a} = \mathbf{2}, \mathrm{WP}(\mathbf{b} = \mathbf{4}, \mathrm{WP}(\mathbf{swap}(\&a, \&b), a = 4 \wedge b = 2)))$$

$$\mathrm{WP}(\mathbf{swap}(\&a, \&b), Q) =$$
$$\forall v_a v_b, v_a = b \wedge v_b = a \Rightarrow v_a = 4 \wedge v_b = 2$$

Example

$$\mathrm{WP}(p, a = 4 \land b = 2) =$$
$$\mathrm{WP}(a = 2, \mathrm{WP}(b = 4, \mathrm{WP}(swap(\&a, \&b), a = 4 \land b = 2)))$$

$$\mathrm{WP}(b = 4, \forall v_a v_b, v_a = b \land v_b = a \Rightarrow v_a = 4 \land v_b = 2) =$$
$$(\forall v_a v_b, v_a = b \land v_b = a \Rightarrow v_a = 4 \land v_b = 2)[b \mapsto 4]$$

Example

$$\mathrm{WP}(p, a = 4 \wedge b = 2) =$$
$$\mathrm{WP}(a = 2, \mathrm{WP}(b = 4, \mathrm{WP}(swap(\&a, \&b), a = 4 \wedge b = 2)))$$

$$\mathrm{WP}(b = 4, \forall v_a v_b, v_a = b \wedge v_b = a \Rightarrow v_a = 4 \wedge v_b = 2) =$$
$$\forall v_a v_b, v_a = 4 \wedge v_b = a \Rightarrow v_a = 4 \wedge v_b = 2$$

Example

$$\mathrm{WP}(p, a = 4 \wedge b = 2) =$$
$$\mathrm{WP}(a = 2, \mathrm{WP}(b = 4, \mathrm{WP}(swap(\&a, \&b), a = 4 \wedge b = 2)))$$

$$\mathrm{WP}(a = 2, \forall v_a v_b, v_a = 4 \wedge v_b = a \Rightarrow v_a = 4 \wedge v_b = 2) =$$
$$(\forall v_a v_b, v_a = 4 \wedge v_b = a \Rightarrow v_a = 4 \wedge v_b = 2)[a \mapsto 2]$$

Example

$$\mathrm{WP}(p, a = 4 \wedge b = 2) =$$
$$\mathrm{WP}(a = 2, \mathrm{WP}(b = 4, \mathrm{WP}(swap(\&a, \&b), a = 4 \wedge b = 2)))$$

$$\mathrm{WP}(a = 4, \forall v_a v_b, v_a = 4 \wedge v_b = a \Rightarrow v_a = 4 \wedge v_b = 2) =$$
$$\forall v_a v_b, v_a = 4 \wedge v_b = 2 \Rightarrow v_a = 4 \wedge v_b = 2$$

$$\mathrm{WP}(p, a = 4 \wedge b = 2) =$$
$$\mathrm{WP}(a = 2, \mathrm{WP}(b = 4, \mathrm{WP}(swap(\&a, \&b), a = 4 \wedge b = 2)))$$

$$\forall v_a v_b, v_a = 4 \wedge v_b = 2 \Rightarrow v_a = 4 \wedge v_b = 2$$

Use WP to prove this example:

```
1  extern int h ;
2
3  int main(void){
4    h = 42;
5    int a = 2;
6    int b = 4;
7    swap(&a, &b);
8    //@ assert a == 4 && b == 2 ;
9  }
```

A partition of features: behaviors

```
1  /*@
2    behavior <name>:
3      assumes  ... ;
4      requires ... ;
5      ensures  ... ;
6    ...
7    disjoint behaviors ;
8    complete behaviors ;
9  */
10 void function( ... ){ ... }
```

- reflect the need to partition the specification
- allows checking completeness and disjointness

Behaviors

```
/*@ requires R;       // global precondition
  @ ensures E;        // global postcondition
  @ assigns L;        // possibly assigned lvalues
  @
  @ behavior b1:      // behavior b1
  @   assumes A1;     // guard for b1
  @   requires R1;    // precondition for b1
  @   ensures E1;     // postcondition for b1
  @   assigns L1;     // possibly assigned lvalues when b1 is enabled
  @ ...
  @ behavior bn:
  @   assumes An;
  @   requires Rn;
  @   ensures En;
  @   assigns Ln;
  @
  @ complete behaviors;
  @ disjoint behaviors; */
```

- the caller must satisfy preconditions:

$$R \wedge \bigwedge_{i \in [1..n]} (A_i \implies R_i)$$

- the caller must satisfy preconditions:

$$R \wedge \bigwedge_{i \in [1..n]} (A_i \implies R_i)$$

- the callee must guarantee postconditions:

$$E \wedge \bigwedge_{i \in [1..n]} (\texttt{\textbackslash old}(A_i) \implies E_i)$$

- the modifiable lvalues are in L; and
- for all $i \in [1..n]$, if A_i is valid in the initial state, then the modifiable lvalues are in L_i.

- the caller must satisfy preconditions:

$$R \wedge \bigwedge_{i \in [1..n]} (A_i \implies R_i)$$

- the callee must guarantee postconditions:

$$E \wedge \bigwedge_{i \in [1..n]} (\texttt{\textbackslash old}(A_i) \implies E_i)$$

- the modifiable lvalues are in L; and
- for all $i \in [1..n]$, if A_i is valid in the initial state, then the modifiable lvalues are in L_i.
- **`complete behaviors`**: all cases are covered ($\bigvee_{i \in [1..n]} A_i$ is valid in the initial state)
- **`disjoint behaviors`**: all cases are mutually exclusive

Behaviors implementation in WP

Beware: currently WP can prove an `assigns` clause in a behavior but cannot use it.

- so don't use it :)
- use a global `assigns` + additional postconditions

Example

Specify the following function with behaviors:

```
1 int days_of(int m){
2   int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 } ;
3   return days[m-1] ;
4 }
```

Remember binary search

Some operations may fail during the execution if their safety conditions are not satisfied.
We want to take that is account in the axiomatic semantics:

$\{P\}\ c\ \{Q\}$ if we execute c
from a memory state that satisfies P,
then there is no runtime error
and, either the execution diverges, or it terminates
in a memory state that satisfies Q.

Strategies when dealing with runtime errors

We have several ways to deal with that:

- extend the WP calculus to deal with RTEs,
- use functions for operations that may fail,
- generate special annotations that must be verified locally.

Partial operators

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & t \texttt{ div } t & \text{euclidean division} \\
 & | & a \texttt{[} t \texttt{]} & \text{array access} \\
 & | & \ldots &
\end{array}
$$

with special WP rules:

$$
\begin{aligned}
\mathrm{WP}(t_1 \texttt{ div } t_2, Q) &\equiv t_2 \neq 0 \ \wedge \ Q[\texttt{result} \mapsto (t_1 \texttt{ div } t_2)] \\
\mathrm{WP}(a\texttt{[}t\texttt{]}, Q) &\equiv 0 \leqslant t < |a| \ \wedge \ Q[\texttt{result} \mapsto a\texttt{[}t\texttt{]}] \\
&\ldots
\end{aligned}
$$

Not really practical: we prefer to have a unified way to deal with that

Use function contracts

We can delegate this work to function contracts:

```
1  /*@ requires b != 0 ;
2      assigns  \nothing ;
3      ensures  \result == a / b; */
4  int div(int a, int b){
5    return a / b ;
6  }
7
8  /*@ requires \valid_read(p + i)
9      assigns  \nothing ;
10     ensures  \result == p[i]; */
11 int get(int* p, int i){
12   return p[i] ;
13 }
```

- usable in practice when the language is designed for that (e.g. Why3)
- for language like C, it is not really practical

Assertions

```
1   //@ assert P ;
```

Hoare rule:

$$\overline{\{R \wedge Q\}\,\texttt{assert}\ R\,\{Q\}}$$

Weakest precondition:

$$\mathrm{WP}(\texttt{assert}\ R, Q) \quad = \quad (1)\, R \wedge Q \quad = \quad (2)\, R \wedge (R \rightarrow Q)$$

In practice, we use version 2.

The RTE plug-in of Frama-C

Invalid acces

```
1 //@ assert \valid(p+i) ;
2 int x = p[i] ;
```

Division by zero

```
3 //@ assert d != 0 ;
4 int x = 42 / d ;
```

Integer overflow

```
5 //@ assert INT_MIN <= a+b;
6 //@ assert a+b <= INT_MAX;
7 int x = a + b ;
```

- Initialization,
- `float` to `int`,
- invalid shifts,
- ...

Integers: ACSL vs. C

- In C
  - bounded integers, may overflow
- In ACSL
  - mathematical integers, no overflow
  - total underspecified functions
    - defined for their entire domain
    - but the value might not be known or make sense
  - no erros: `1/0 == 1/0` is fine
  - can be casted to a C type:

$$(\tau)n \equiv n \mod 2^{8\times \text{sizeof}(\tau)}$$

Integers: ACSL vs. C

- In C
  - bounded integers, may overflow
- In ACSL
  - mathematical integers, no overflow
  - total underspecified functions
    - defined for their entire domain
    - but the value might not be known or make sense
  - no erros: `1/0 == 1/0` is fine
  - can be casted to a C type:

$$(\tau)n \equiv n \mod 2^{8\times \text{sizeof}(\tau)}$$

- are the following predicates valid?
  - `\forall integer x,y; \exists integer z; x+y == z`
  - `\forall int x,y; \exists int z; x+y == z`

Pray to the Absolute

Specify and prove the following function guaranteeing absence of runtime errors:

```
1 int abs(int x){
2   return x < 0 ? -x : x ;
3 }
```

Non termination might hide problems.

$\{P\}\ c\ \{Q\}$ if we execute c
from a memory state that satisfies P,
then there is no runtime error
and the execution terminates
in a memory state that satisfies Q.

This is called total correction.

We can demonstrate that:

- each loop has a finite number of iterations
- each call generates a finite number of recursive calls

We can demonstrate that:

- each loop has a finite number of iterations
- each call generates a finite number of recursive calls

Solution: prove that each iteration of a loop and each recursive call makes a given value decrease,
this value is called a *variant* wrt a well-founded relation
(i.e. without infinitely decreasing chain)

For example for signed integers:

$$i \prec j \quad = \quad i < j \wedge 0 \leq j$$

```
1 //@ loop variant t ;
```

At the end of the iteration:

- we prove that `\at(t, Here)` $\prec$ `\at(t, LoopCurrent)`

Specify and prove the following function:

```
1  int f(int x, int y){
2    int r = 0 ;
3    int i = 0 ;
4    while(i < x){
5      r = r + y ;
6      i++;
7    }
8    return r ;
9  }
```

```
1 //@ decreases t ;
```

At each internal call:

- we prove that `\at(t, Here)`* $\prec$ `\at(t, Pre)`
- * as instantiated in the call.

For mutually recursive functions:

- each function has its own term,
- all functions must share the same relation.

Example

Prove the termination of the following function:

```
1 int sum(int fst, int lst, int acc){
2   if (fst >= lst) return acc ;
3   else return sum(fst+1, lst, fst+acc) ;
4 }
```

MAX

```
1 int max(int a, int b){
2   return a < b ? b : a ;
3 }
```

MAX

```
1 /*@ ensures \result >= a && \result >= b ; */
2 int max(int a, int b){
3   return a < b ? b : a ;
4 }
```

MAX

```
1  /*@ ensures \result == a || \result == b ;
2      ensures \result >= a && \result >= b ;
3      assigns \nothing ;
4  */
5  int max(int a, int b){
6    return a < b ? b : a ;
7  }
```

Precision and correctness of a specification

Imprecise specification

- see previous example about `max`
- but it is not necessarily a problem: it depends on what we want to prove
- and we do not want to have a specification that is too complex

Incorrect specification

- the specification does not reflect what we have in mind
- dramatic: we might prove that something bad happens without knowing
- ... or prove nothing at all ...

Detecting unsatisfiable preconditions

WP option: `-wp-smoke-tests`

- tries to prove that the precondition is always false
- if it succeeds, the precondition is certainly wrong
- if it does not, it is, at least, not trivially wrong

```
1 //@ requires x < 0 && x > 0 ;
2 void f(int x){}
```

Dead behaviors are more subtle

- the function can be called
- but an entire part of its contract might be dead for no good reason

```
1 /*@ requires x < 0 ;
2     behavior B1:
3       assumes x > 0 ;
4 */
5 void f(int x){}
```

A wrong requirement might also kill program paths

- the function can be called
- but some code in the function is dead in all calls
- or simply unverified ...

```
1 //@ requires x < 0 ;
2 void f(int x){
3   if(x > 0){
4     x++;
5   }
6 }
```

LETTER

```c
enum Kind { VOWEL, CONSONANT };

enum Kind kind_of_letter(char c){
  if(c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' || c == 'y')
    return VOWEL;
  else
    return CONSONANT;
}
```

```
/*@ requires 'a' <= c <= 'z' ;
    assigns \nothing;
    behavior vowel:
      assumes c \in { 'a', 'e', 'i', 'o', 'u', 'y' };
      ensures \result == VOWEL;
    behavior consonant:
      assumes ! (c \in { 'a', 'e', 'i', 'o', 'u', 'y' });
      ensures \result == CONSONANT;
    complete behaviors;
    disjoint behaviors; */
enum Kind kind_of_letter(char c){
  if(c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u' || c == 'y')
    return VOWEL;
  else
    return CONSONANT;
}
```

DAYS-OF

```c
int f(int y){
  return ((y % 4 == 0) && (y % 100 !=0)) || (y % 400==0) ;
}

int day_of(int m, int y){
  int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 } ;
  return days[m-1] + (m == 2 ? f(y) ? 1 : 0 : 0);
}
```

```c
enum note { N200, N100, N50, N20, N10, N5, N2, N1 };
int const values[] = { 200, 100, 50, 20, 10, 5, 2, 1 };

int remove_max_notes(enum note n, int* rest){
  int num = (*rest)/values[n];
  (*rest) -= num * values[n];
  return num;
}

int make_change(int amount, int received, int change[8]){
  if(amount > received) return -1;
  int rest = received - amount ;
  change[N200] = remove_max_notes(N200, &rest);
  change[N100] = remove_max_notes(N100, &rest);
  //...
  return 0;
}
```

FORALL, EXISTS ...

```
1  #include <stddef.h>
2
3  /*@
4    assigns \nothing ;
5    ensures \result <==> \true ; // some property about value instead
6  */
7  int pred(int value){ /* your code */ }
8
9  int forall_pred(const int* array, size_t len){}
10 int exists_pred(const int* array, size_t len){}
11 int none_pred(const int* array, size_t len){}
12 int not_all_pred(const int* array, size_t len){}
```

Step 1: `loop assigns`

Start with the `loop assigns` specification

- it is generally simple to guess,
- do not try to be too smart, prefer additional invariant to super precise assigns,
- e.g. `loop assigns x[0 .. i] --> loop assigns x[0 .. len-1]`

Step 2: bound assigned variables

Then bound the assigned variables

- in particular indexes,
- these invariant are generally easy to guess,
- you should put them at the beginning of the list.

For most loops

This is the right moment to guess a suitable `loop variant`

- e.g. `while (i < e) ... --> loop variant e - i`

Step 3: Provide the "main" invariant

Link the loop invariant with the postcondition

- use the postcondition itself as a guide,
- e.g. `ensures P(n) +` `while(i < n) --> loop invariant P(i)`
- it can also be the guard of a behavior,
- e.g. when the result depends on the state at pre
- you should put these invariants at the end of the list

Step 4: (Optional) add "glue"

It may not be enough

- ... when using voluntarily imprecise `loop assigns`
- ... when proving the interesting invariant relies on additional implicit invariant
- ...
- you should put them between invariants of Step 1 and Step 3

RESET

```c
#include <stddef.h>

void reset(int* array, size_t length){
  for(size_t i = 0; i < length; ++i)
    array[i] = 0;
}
```

```
1  #include <stddef.h>
2
3  size_t search(int* arr, size_t len, int value){
4    for(size_t i = 0; i < len; i++)
5      if(arr[i] == value) return i;
6    return len;
7  }
```

```
1  #include <stddef.h>
2
3  size_t bsearch(int* arr, size_t len, int value){
4    if(len == 0) return 0 ;
5
6    size_t low = 0 ;
7    size_t up = len ;
8
9    while(low < up){
10     size_t mid = low + (up - low) / 2 ;
11     if      (arr[mid] > value) up = mid ;
12     else if(arr[mid] < value) low = mid+1 ;
13     else return mid ;
14   }
15   return len ;
16 }
```

SEARCH-AND-REPLACE

```c
#include <stddef.h>

void search_and_replace(int* array, size_t length, int old, int new){
  for(size_t i = 0; i < length; ++i){
    if(array[i] == old) array[i] = new;
  }
}
```

Problem ...

```
1 // Pre: { *p == 0 && *q == 0 }
2 // { 42 == 42 && *q == 0;} !!!
3 *p = 42
4 // { *p == 42 && *q == 0;}
5
```

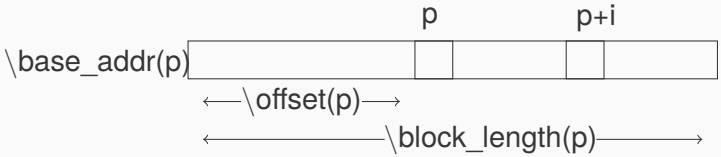
Hoare logic and WP calculus require absence of aliasing

- Why3 relaxes this restriction thanks to static verification of aliases
- We must point out the external dependencies for functions
- else, static checks fail

For programs with arbitrary pointers, we need more sophisticated approaches

- memory models
- handle alias in VCs (separation logic, dynamic frames, ...)

- a pointer gives access to a memory block
- pointer = base address + offset



- `\base_addr`, `\offset` and `\block_length` are ACSL logic functions
- `\base_addr(p)` is the address of the beginning of the block that contains `p`, that is, the pointer with its base and offset 0
- `\offset` is the offset (in bytes) between `p` and its base address,
- `\block_length` is the length (in bytes) of the block that contains `p`

- In imperative programming, programs are sequences of commands
- For example, the assignment `x := x+1;` update the memory state that precedes the assignment in order to associate to x its old value incremented by 1.
- A memory model is a way to model how a program stores and accesses to its data during the execution.
- A memory model is useful to :
  - formalize imperative languages semantics
  - verify properties about imperative programs
  - prove programs transformations
  - analyze concurrent programs behaviors

First memory model

- type $\texttt{pointer}_\tau$ to a value of type τ + type $\texttt{memory}$ for memory states
- a function to read through a pointer:

$$\texttt{read} : \texttt{memory} \to \texttt{pointer}_\tau \to \tau$$

- a function to write through a pointer:

$$\texttt{write} : \texttt{memory} \to \texttt{pointer}_\tau \to \tau \to \texttt{memory}$$

- algebraic laws for composition "read after write":

$$\texttt{read}\,(\texttt{write}\; m\; p\; v)\; p = v$$
$$\texttt{read}\,(\texttt{write}\; m\; p\; v)\; p' = \texttt{read m p}' \qquad \text{if } p \neq p'$$

- these laws allow reasoning about memory operations
- the memory is seen as an array of types values
- allows modeling arrays and (typed) pointer arithmetic

A language with pointers

- we add pointers to the language:

$$\begin{array}{rcll} e & ::= & \ldots & \text{same as before} \\ & | & \star e & \text{access to a pointed value} \end{array}$$

- assignment in Hoare logic without pointers:

$$\overline{\{\, P[x \mapsto e] \,\}\; x := e \;\{P\}}$$

1 ⟩ we add a global memory state:

$$\texttt{mem}: \quad \texttt{memory}$$

2 ⟩ assignment rule for a pointed value:

$$\overline{\{\ P[\mathit{mem} \mapsto \texttt{write}\ \mathit{mem}\ p\ e]\ \}\ \star p := e\ \{P\}}$$

3 ⟩ modeling the read of a pointed value: during substitutions replace in e each occurrence of $*p$ with *read mem p*.

› Example:

$$\overline{\{\ \texttt{read}\ \mathit{mem}\ p + 1\ =\ 1\ \}\ x := \star p + 1\ \{x = 1\}}$$

Exercise 5

1 ⟩ Show that the following Hoare triple is provable

$$\{\ \texttt{read}\ \mathit{mem}\ p = 0\ \}\ \star p := \star p + 1\ \{\ \texttt{read}\ \mathit{mem}\ p = 1\ \}$$

2 ⟩ Define an algebraic law that characterize the following memory isolation property for base addresses: if p and q are 2 pointers with different bases, then writing through p does not modify $\star\, q$.

Without memory model

- we represent program variables with logic variables
- no notion of memory
- pros:
  - verification conditions are easy to prove
- cons:
  - we cannot express aliasing
  - we cannot model pointers
- in practice, this is the Hoare model in Frama-C/WP

With our simple memory model

- we see memory as an array of values
- no memory separation
- pros:
  - aliasing and pointer arithmetic can be modeled
- cons:
  - the typing is ... strange
  - when we modify $*p$, if we want to prove that $*p'$ is unchanged, we have to prove $p \neq p'$,
  - such proofs are generally complex and not automatic
- consequence: this model is useless in practice

In practice

- pointers, with pointer arithmetics,
- allocations/deallocations,
- addresses of variables,
- complex datatypes (arrays, structures, unions),
- (heterogeneous) casts,
- ...

Which memory model should we use?

- none is perfect,
- compromise between automatic and expressiveness

- According to the C norm, automatic variables are allocated in distinct memory blocks. We then can introduce a separation hypothesis about these blocks, for example for global variables:

$$\begin{array}{l}\forall \text{ global variables } G_1, G_2,\\ \forall i, j,\\ \quad 0 \leq i < \backslash block\_length(\&G_1) \implies\\ \quad 0 \leq j < \backslash block\_length(\&G_2) \implies\\ \quad \&G_1 + i \neq \&G_2 + j\end{array}$$

- we can model each variable using a separated logic variable,
- modifying a variable does not modify the other ones,
- improves automation

(Simplified) Default WP memory model

- the memory is typed but contains different sorts of values,
- each sort of fundamental value has its own memory,
- compound types are built on top of that,
- pros:
  - modifying a value of type `int`, does not modify values of type `float`
- cons:
  - no heterogeneous casts

First example with float and int

```
1 void function(int* i, int* j, float* f){
2   *i = 42 ;
3   *f = 1. ;
4   *j = 24 ;
5   //@ check same_int: *i == 42 ;
6   //@ check same_flt: *f == 1. ;
7 }
```

`\at` and `\old`

Talking about a value at a program point different from the current one.

- **`\at`**`(x, L)` denotes the value of `x` (term or predicate), at a program label `L` that can also be a predefined label,
- predefined labels:
  - **`Here`** the current program point,
  - **`Pre`** (resp. **`Post`**): program point just before (resp. just after) the function call,
  - **`Old`**, mostly similar to **`Pre`** for us, can only be used in **`ensures`** and **`assigns`**,
  - **`LoopEntry`** (resp. **`LoopCurrent`**): program point just at the beginning (resp. just before the current iteration) of a loop,
  - `Init` program point after program startup.
- syntactical sugar: **`\old`**`(t)` $\equiv$ **`\at`**`(t, Old)`

WP Reference model

When a function `f` always accesses to a pointer `p` via `*p` only, `f` cannot create an alias to `p`.

- we can assume this and model these pointers with the Hoare model!
- Typed+Ref does that automatically,
- ... but this is a strong hypothesis!

Exercise 6 – Model choice

```
# frama-c swap.c -wp
[kernel] Parsing swap.c
[wp] 3 goals scheduled
[wp] Proved goals:    5 / 5
  Terminating:       1
  Unreachable:       1
  Qed:               2  (0.81ms)
  Alt-Ergo 2.6.0:    1  (19ms)
```

```
# frama-c swap.c -wp -wp-model +ref
[kernel] Parsing swap.c
[wp] 2 goals scheduled
[wp] Proved goals:    4 / 4
  Terminating:      1
  Unreachable:      1
  Qed:              2
[wp] swap.c:4: Warning:
  Model hypotheses for 'swap':
  /*@
    behavior wp_typed_ref:
      requires \valid(a);
      requires \valid(b);
      requires \separated(a, b);
  */
  void swap(int *a, int *b);
```

ACSL: function and predicates about memory

Properties about pointers

Logic functions:

- `\base_addr(p)`: base address of a pointer
- `\block_length(p)`: length of the memory block
- `\offset(p)`: offset between the base and the pointer
- ...

Predicates:

- `\valid(p)`: `p` points to a memory block
- `\initialized(p)`: the pointer value is initialized
- `\separated(p, q)`: `p` and `q` point to different memory blocks
- ...

Other memory models ...

- Burstall-Bornat model
  - separated fields for objects and structures
  - ease reasoning about structures and objects
- Bytes model
  - models the memory as an array of bytes
  - super precise but costly
- Region model
  - choose memory model according to an alias analysis

ACSL does not specify a memory model! This choice is left to tools

Why do we need to guide the proof process?

- we want to prove programs with loops
- sometimes, directly proving the important formula is not that easy
- we want to increase proof speed

Different kinds of assertions

Verify/admit something at a program point

- `check` verify that the property is true
- `admit` assume that the property is true
- `assert` first, verify that it is true, then, assume that it is

Examples

```
1 ivette assert-1.c -wp
```

What is the purpose of each kind of assertion

- `check`: something that must be verified but might pollute other proofs
- `admit`: kill program paths to focus current effort on something else
- `assert`: guide proof by introducing cuts, we will see that later

Predicates and logic functions

```
1 /*@ predicate is_42(int* p) =
2       *p == 42 ;
3 */
4 /*@ logic integer val_of_p(int* p) =
5       *p ;
6 */
```

Memory parameters

Logic functions and predicates can be parameterized with labels

- they represent program points
- they are not ordered *per se*

```
1  /*@ predicate unchanged{L1, L2}(int *p) =
2        \at(*p, L1) == \at(*p, L2) ; */
3
4  int main(void){
5    int x = 4, y = 5 ;
6   L: y++ ;
7    //@ assert unchanged{L, Here}(&x) ;
8  }
```

Call is by value

- Using `\at` operator on a logic value does not make sense

```
1  /*@ predicate unchanged{L1, L2}(int i) =
2        \at(i, L1) == \at(i, L2) ; */
3
4  int main(void){
5    int x = 4, y = 5 ;
6   L: x++ ;
7    //@ assert unchanged{L, Here}(x) ; // Woops
8  }
```

Search and replace

```
1 #include <stddef.h>
2
3 void search_and_replace(int* array, size_t length, int old, int new){
4   for(size_t i = 0; i < length; ++i){
5     if(array[i] == old) array[i] = new;
6   }
7 }
```

- define predicate `unchanged`
- define predicate `replaced`
- use them for the specification and the proof

Search and replace - 1

```
1  /*@ predicate unchanged{L1, L2}(int* a, integer fst, integer last) =
2        \forall integer j;
3          fst <= j < last ==> \at(a[j], L1) == \at(a[j], L2);
4
5      predicate replaced{L1, L2}(int* a, integer len, int old, int new) =
6        (\forall integer j;
7           0 <= j < len && \at(a[j], L1) == old ==>
8             \at(a[j], L2) == new) &&
9        (\forall integer j;
10          0 <= j < len && \at(a[j], L1) != old ==>
11            \at(a[j], L2) == \at(a[j], L1));
12 */
```

Search and replace - 2

```
1  /*@ requires \valid(arr + (0 .. len-1));
2      assigns arr[0 .. len-1];
3      ensures replaced{Pre, Here}(arr, len, old, new) ;
4  */
5  void search_and_replace(int* arr, size_t len, int old, int new){
6    /*@ loop invariant 0 <= i <= len;
7        loop invariant unchanged{Pre, Here}(arr, i, len) ;
8        loop invariant replaced{Pre, Here}(arr, i, old, new) ;
9        loop assigns i, arr[0 .. len-1];
10       loop variant len-i; */
11   for(size_t i = 0; i < len; ++i)
12     if(arr[i] == old) arr[i] = new;
13 }
```

Spot the problem:

```
1  /*@ logic integer count(int v, int* a, integer b, integer e) =
2        (e <= b) ? 0 : (a[b] == v ? 1 : 0) + count(v, a, b+1, e) ;
3
4      predicate unchanged{L1, L2}(int* a, integer fst, integer last) =
5        \forall integer j;
6          fst <= j < last ==> \at(a[j], L1) == \at(a[j], L2);
7  */
8  /*@ assigns *r ;
9      ensures unchanged{Pre, Here}(a, 0, n) ; */
10 void do_something(int* r, int *a, size_t n);
11
12 /*@ assigns *r ;
13     ensures
14     \forall int v ; count{Pre}(v, a, 0, n) == count{Post}(v, a, 0, n) ; */
15 void f(int* r, int *a, size_t n){
16   do_something(r, a, n) ;
17 }
```

Counting

```
1  /*@ requires \separated(r, a + (0 .. n-1)) ;
2      assigns *r ;
3      ensures  unchanged{Pre, Here}(a, 0, n) ;
4  */
5  void do_something(int* r, int *a, size_t n);
6
7  /*@ requires \separated(r, a + (0 .. n-1)) ;
8      assigns *r ;
9      ensures
10       \forall int v ; count{Pre}(v, a, 0, n) == count{Post}(v, a, 0, n) ;
11 */
12 void f(int* r, int *a, size_t n){
13   do_something(r, a, n) ;
14 }
```

Yet, the proof fails

Reusable proof of a deduction step

- prove that a step of deduction is correct in isolation
- so that the solver can use it when it makes sense
- **beware:** by default a lemma talks about *any* memory state

```
1 /*@ lemma its_name {L1, ...}:
2       \forall ... ; P ;
3 */
```

Counting

```
1  /*@ lemma unchanged_counter{L1, L2}:
2        \forall int v, int* p, integer b, e ;
3          unchanged{L1, L2}(p, b, e) ==>
4            count{L1}(v, p, b, e) == count{L2}(v, p, b, e) ;
5  */
6
7  /*@ requires \separated(r, a + (0 .. n-1)) ;
8      assigns *r ;
9      ensures
10       \forall int v ; count{Pre}(v, a, 0, n) == count{Post}(v, a, 0, n) ;
11 */
12 void f(int* r, int *a, size_t n){
13   do_something(r, a, n) ;
14 }
```

Some functions cannot be directly defined

- in particular partial functions
- one can use axioms in such a case
- or sometimes axioms are just more efficiently handled by solvers

```
1 /*@ axiomatic Strlen {
2       logic integer strlen(char* s) reads s[0..] ;
3       axiom non_negative_strlen:
4         \forall char* s ; strlen(s) >= 0 ;
5     }
6 */
```

Axiomatic definitions - reads clause

The reads clause is important!

It tells WP when the information about the predicate/function should be invalidated

```
1  /*@ axiomatic Ax {
2        predicate P(int* a) ;
3        axiom a: \forall int* a ; P(a) <==> *a == 42 ;
4      }
5  */
6
7  /*@ requires P(a) ;
8      ensures  P(a) ;  // <<< PROVED
9  */
10 void function(int *a){ *a = 0 ; }
```

Example

```
/*@ 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);
    }
*/
```

- both the prototype and the cases can have labels
- Frama-C does not check the inductive is well-founded (but Why3 does)
- **cases are axioms**

Inductive

- more efficiently compiled to the logic
- Why3 can check that they are well-founded
- predicates

Axiomatic

- can model logic functions
- can group several functions

Again, smoke tests can help detect wrong axiomatic definitions

- **But again, you cannot be sure that everything is fine**

Global context vs. local context

The *visible* VC embeds the local context

- preconditions
- effects of the instructions
- admitted local properties

The global context is not visible by default, it embeds:

- WP Why3 models
- WP axioms about the shape of the memory
- user defined axioms and lemmas
- **limitation:** currently it is hard to see exactly this list of properties

Example

```
1  /*@ axiomatic Ax {
2        predicate P(int* x) reads *x ;
3        predicate Q(int* x) reads *x ;
4        axiom a1: \forall int* x ; *x <= 42 ==> P(x) ;
5        axiom a2: \forall int* x ; P(x) ==> Q(x) ;
6      }
7  */
8
9  /*@ requires P(x) ;
10     ensures  Q(x) ; */
11 void function(int* x){
12   if(*x < 42){
13     (*x) ++ ;
14   } else {
15     *x = 42 ;
16     //@ assert P(x) ;
17   }
18 }
```

Using assertions to trigger lemmas

Lemmas are applied automatically

- ... but not out of nowhere
- “something” in the context must tell them that it is at least possible
- generally it is a term that matches the premise of the lemma to use
- it can appear:
  - in the contract (+ transitive transformations),
  - in another lemma,
  - in the local context because of the code (rare for complex lemmas)
  - in the local context because of a clever assertion

Toy exercise (I)

```
/*@
  axiomatic Ax {
    predicate X{L1, L2}(int* p, integer l)
        reads \at(p[0 .. l-1], L1), \at(p[0 .. l-1], L2) ;
    predicate Y{L1, L2}(int* p, integer l)
        reads \at(p[0 .. l-1], L1), \at(p[0 .. l-1], L2) ;

    axiom Ax_axiom_XY {L1,L2}:
      \forall int* p, integer l, i ;
        0 <= i <= l ==> X{L1, L2}(p, i) ==> Y{L1, L2}(p, l) ;
    axiom transitive{L1,L2,L3}:
      \forall int* p, integer l ;
        Y{L1,L2}(p, l) ==> Y{L2,L3}(p, l) ==> Y{L1,L3}(p, l);
  }
*/
```

Toy exercise (II)

```
1  /*@
2    assigns p[0 .. l-1] ;
3    ensures X{Pre, Post}(p, l) ;
4  */
5  void f(int* p, unsigned l);
6
7  /*@
8    ensures Y{Pre,Post}(p, l);
9  */
10 void g(int* p, unsigned l){
11   f(p, l) ;
12   f(p, l) ;
13 }
```

Increase cell

```c
#include <stddef.h>

void inc_cell(int* array, size_t len, size_t i){
  array[i]++ ;
}
```

Ghost code: Principle

Observing without interfering

- regular C code
- except that it cannot interfere with actual code
- can explicit the implicit behavior of the code

Ghost code

- starts with `//@` (or `/*@`)
- ghost annotations are surrounded by `/@` and `@/`

```
1 int x = 24 ;
2 //@ ghost int v = x ;
3
4 /*@ ghost
5     if(v < 10){
6       v++ ;
7       /@ assert v <= 10 ; @/
8     }
9 */
```

Ghost code must not break control flow

- Frama-C checks that ghost code has the same control flow as if it was not there

```
1  extern int value ;
2  int main(void){
3    if(value){
4      return 1
5    } /*@ ghost else {
6      return 0 ; //< raises an error
7    } */
8
9    return 0 ;
10 }
```

Ghost code can store information

- ghost variables form the ghost memory
- this memory is separated from the normal memory
- Frama-C checks this, but needs help for pointers using the `\ghost` qualifier

```
1  int v ;
2  /*@ ghost
3    int g = v ;
4    int* p = &v ;          // legal
5    *p = 1 ;               // illegal: *p is "normal" memory
6    int \ghost * q = &v ; // illegal: v type is "normal" memory
7    int \ghost * r = &g ; // legal
8    *r = 1 ;               // legal
9  */
```

Important known limitation

Memory typing is strict and prevent some legitimate usage

- a normal function cannot be called in ghost code to treat ghost memory
- even if it would be perfectly fine semantically

```
1 int a[10] ;
2 //@ ghost int g[10] ;
3
4 int x = search(a, 10, 5) ; // OK
5 //@ ghost int y = search(g, 10, 5) ; // KO
```

Lemmas are fine but

- they can be hard to trigger,
- proving them might not be that easy,
- sometimes a bit more code can give a lot of information

Functions are already cuts in the global proof of the system

- could we use functions as lemmas?
- yes, these are called lemma functions

General idea

- premises are `requires`
- conclusion is `ensures`
- existential can be modeled as the result of the function

Example with a single memory state

```
1  /*@ predicate not_in(int v, int* arr, integer len) =
2        \forall integer i ; 0 <= i < len ==> arr[i] != v ;
3  */
4
5  /*@ lemma not_in_occ_0:
6        \forall int v, int* arr, integer len ;
7          not_in(v, arr, len) ==> l_occurrences_of(v, arr, 0, len) == 0 ;
8  */
```

Example with a single memory state

```
/*@ ghost
  /@ requires not_in(v, arr, len) ;
     assigns \nothing ;
     ensures l_occurrences_of(v, arr, 0, len) == 0 ;
  @/
  void not_in_occ_0(int v, int* arr, size_t len){
    /@ loop invariant 0 <= i <= len ;
       loop invariant l_occurrences_of(v, arr, 0, i) == 0 ;
       loop assigns i ;
       loop variant len - i ;
    @/
    for(size_t i = 0 ; i < len ; ++i){
      /@ assert arr[i] != v ; @/
    }
  }
*/
```

Example with a single memory state

```
1 /*@ requires not_in(new, a, len) ; */
2 void function(int* a, size_t len, int new){
3   //@ ghost not_in_occ_0(new, a, len) ;
4   //@ assert l_occurrences_of(new, a, 0, len) == 0 ;
5 }
```

Example with several memory states

```
/*@ axiomatic Sum_arr{
      logic integer sum(int* arr, integer b, integer e)
        reads arr[b .. (e-1)];
      axiom empty:
        \forall int* a, integer b, e;
          b >= e ==> sum(a,b,e) == 0;
      axiom range:
        \forall int* a, integer b, e;
          b < e ==> sum(a,b,e) == sum(a,b,e-1)+a[e-1];
    }
*/
```

```
/*@ requires \valid(a + (0 .. len-1)) && i < len ; */
void function(int* a, size_t len, size_t i){
  a[i] ++ ; a[i] -- ;
  //@ assert sum(a, 0, len) == sum{Pre}(a, 0, len);
}
```

Example with several memory states

```
1  #define unchanged_sum(_L1, _L2, _arr, _beg, _end)                    \
2    /@ assert unchanged{_L1, _L2}(_arr, _beg, _end) ; @/               \
3    /@ loop invariant _beg <= _i <= _end ;                             \
4       loop invariant sum{_L1}(_arr, _beg, _i) ==                      \
5                      sum{_L2}(_arr, _beg, _i) ;                       \
6       loop assigns _i ;                                               \
7       loop variant _end - _i ;                                        \
8     @/                                                                \
9    for(size_t _i = _beg ; _i < _end ; ++_i){                          \
10     /@ assert \at(_arr[\at(_i, Here)], _L1) ==                       \
11               \at(_arr[\at(_i, Here)], _L2); @/                      \
12   }                                                                  \
13   /@ assert sum{_L1}(_arr, _beg, _end) == sum{_L2}(_arr, _beg, _end) ; @/
```

Example with several memory states

```
1  /*@ requires \valid(a + (0 .. len-1)) ;
2      requires i < len ;
3  */
4  void function(int* a, size_t len, size_t i){
5    a[i] ++ ;
6    a[i] -- ;
7    //@ ghost unchanged_sum(Pre, Here, a, 0, len) ;
8    //@ assert sum(a, 0, len) == sum{Pre}(a, 0, len);
9  }
```

Ghost code in ACSL and Frama-C is not the silver bullet

- logic types are currently not supported in ghost code
- ghost fields of structures are not supported
- C ghost code does not allow functions with multiple memory labels

But it is super efficient to reduce the need of a proof assistant

```
1
2  #include <stddef.h>
3
4  /*@ ghost
5      void occ_bounds(int v, int* arr, size_t len){}
6      void occ_monotonic(int v, int* arr, size_t pos, size_t more){}
7      void occ_0_implies_not_in(int v, int* arr, size_t len){}
8      size_t occ_pos_implies_in(int v, int* arr, size_t len){}
9      void occ_pos_implies_exists(int v, int* arr, size_t len){}
10 */
```

Sum integers

```
1  #include <limits.h>
2
3  /*@
4    logic integer sum_of_n_integers(integer n) =
5      (n <= 0) ? 0 : sum_of_n_integers(n-1) + n ;
6  */
7
8  /*@
9    requires n*(n+1) <= UINT_MAX ;
10   assigns \nothing ;
11   ensures \result == sum_of_n_integers(n);
12 */
13 unsigned sum_n(unsigned n){
14   return (n*(n+1))/2 ;
15 }
```