SMT solver & program verification

Lecturer: Yu-Fang Chen

Credits:

Thanks to Yu-Chia Chen for making the slides.
The contents are based on the Slides of Ming-Hsien Tsai, Anthony Lin and David Mantre

First-order logic

Limitations of propositional logic

- Consider the following classical argument:

(1) All men are mortal
(2) Socrates is a man

Therefore: Socrates is mortal

- Can you express this in propositional logic?

Limitations of propositional logic

- Here is an attempt:

Problem: How big is this formula?

(1) All men are mortal

$$Man(Socrates) \to Mortal(Socrates)$$

$$Man(Plato) \to Mortal(Plato)$$

...

(2) Socrates is a man

$$Man(Socrates)$$

Therefore: Socrates is mortal

$$Mortal(Socrates)$$

A better solution

- Extend the logic to easily refer to "all men"

$$\forall x. Man(x) \rightarrow Mortal(x)$$

quantifier

- Read (verbose): *"For all x, if x is a man, then x is mortal"*
- Note: Proposition are now "predicates" which depend on x
- Observation: two lines vs. billions of line

What else can you say in FOL?

- There is a man who is not married

$$\exists x. man(x) \land \neg married(x)$$

- Every person has a mother

$$\forall x. person(x) \rightarrow \left(\exists y. motherOf(y, x)\right)$$

- Some person have two mobile phones

$$\exists x \exists y \exists z. person(x) \land mp(y, x) \land mp(z, x) \land z \neq y$$

So, is it true that …?

- Q: FOL is just PL with quantifiers and more complex "propositions"?
- A: Yes, pretty much. But this is much more complex in fact!

Ponderables

- Quantifiers "quantify" over what?
- Which of the following sentence are "true"?

$$\exists x.\, man(x) \land \neg married(x)$$

$$\big(\exists x.\, man(x)\big) \to \big(\forall y.\, man(x)\big)$$

$$\big(\forall x.\, man(x)\big) \to \big(\exists y.\, man(x)\big)$$

$$\forall x.\, man(x) \to \max(x)$$

First-order logic (FOL) syntax

“Atoms” (simplified)

- Examples of “atomic formulas” (“atoms”) in FOL:

$man(x)$ — variable

$even(1)$ — Constant

$mp(y, x)$ — Predicate /Relation

- Relations have arities (# arguments):
  - $man, even$ have arity 1
  - mp has arity 2
- Relation with arity 0 is a proposition, e.g. $man(\text{“}John\text{”})$

“Atoms” (simplified)

- Variables: $x, y, \ldots$
- Function symbols (with arities): $f/2, +/2, sin/1, \pi/0, \ldots$
- Constants (0-ary function): $0, 1, \pi, “John”, \ldots$
- Terms: variables/constants/functions-over-terms
- Relation symbols (with arities): $man/1, mp/2, =/2$
- Definition: If R/i is a relation symbol with arity i and each of $t_1, t_2, \ldots, t_i$ is a term, then $R(t_1, t_2, \ldots, t_i)$ is an atomic formula

"Formulas"

- As in boolean logic, build formulas from propositions with:

$$\neg, \quad \wedge, \quad \vee, \quad \rightarrow, \quad \leftrightarrow$$

- In addition, formulas can be "quantified":

  If F is a formula and x is a variable, then

  $\forall x. F$ Is a formula

  $\exists x. F$ Is a formula

Exercise

- How do you build the following formulas?

$$\exists x. man(x) \land \neg married(x)$$

$$\big(\exists x. man(x)\big) \to \big(\forall y. man(x)\big)$$

$$\big(\forall x. man(x)\big) \to \big(\exists y. man(x)\big)$$

Exercise

Which are FOL formulas?

- $\exists y \forall x. \big(R(z) \rightarrow R(x)\big)$
- $1 + 3 \times 20$ or $+(1,\ \times(3, 20)\ )$
- $pow(x, n) + pow(y, n) = pow(z, n)$
- $\forall x. \neg\, pow(x, n) \leftrightarrow n = 1$
- $\exists x \exists f. f(x) = 0$

" $=$ " is a relation symbol

Exercise

Which are FOL formulas?

- $\exists y \forall x. \big(R(z) \rightarrow R(x)\big)$
- $1 + 3 \times 20$ ✗
- $pow(x, n) + pow(y, n) = pow(z, n)$
- $\forall x. \neg\, pow(x, n) \leftrightarrow n = 1$ ✗
- $\exists x \exists f. f(x) = 0$ ✗

" $=$ " is a relation symbol

More exercise

- Give a definition of FOL formulas by induction/grammar

More exercise

- $F ::= R(t_1, \ldots t_n)$
  $\mid \neg F \mid F \wedge F \mid F \vee F \mid F \rightarrow F \mid F \leftrightarrow F$
  $\mid \exists x. F \mid \forall x. F$
- $t ::= f(t_1, \ldots t_n)$
  $\mid x$

x is a variable

Semantics of FOL

Interpretations

What do the quantifiers quantify over?

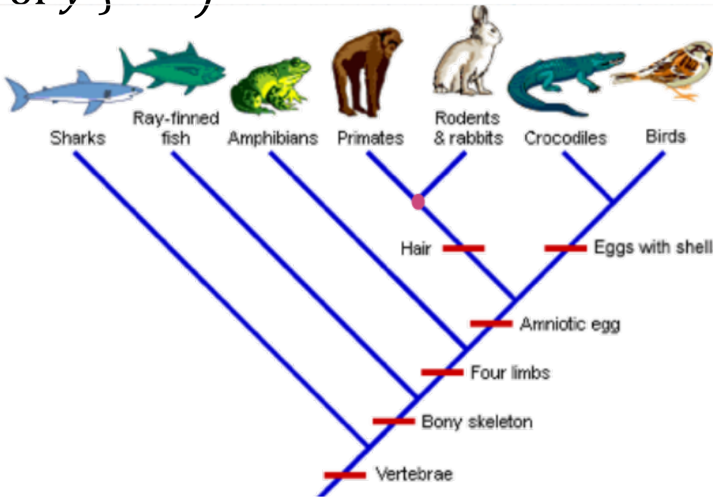
- Domains D (a.k.a. universe)
- An assignment function I mapping:
  - Each variable x to an element in D
  - Each function symbol f/n to a n-arity function $\overbrace{D \times \cdots \times D}^{n} \to D$
  - Each relation symbol R/n to a n-arity relation $\overbrace{D \times \cdots \times D}^{n} \to \mathbb{B}$

Example: phylogeny tree

- Relation symbols: $\leq/2, extant/1, extinct/1$
- Assignment:

$$D = \{Sharks, Birds, \dots\}$$

$$I = \begin{cases} extant/1 \mapsto \{Sharks \mapsto T, Birds \mapsto T \dots\} \\ extinct/1 \mapsto \{Sharks \mapsto F, Birds \mapsto F \dots\} \\ \leq/2 \mapsto \{\ (x, y) |\ x \text{ is subclass of } y\ \} \end{cases}$$



Example: Integer Linear Arithmetic (N,+)

- Function symbol: $+/2,\ \ 0/0, 1/0,\ \ \ldots$
- Predicate symbol: $=/2$
- Assignment:

$$D = \{0,1,2,3, \ldots\}$$
$$I(0) = \{(\,) \mapsto 0\},\ \ I(1) = \{(\,) \mapsto 1\}, \ldots$$
$$I(+) = \{(0,0) \mapsto 0, (0,1) \mapsto 1 \ldots\}$$
$$I(=) = \{(0,0) \mapsto T, (0,1) \mapsto F \ldots\}$$

Truth depends on interpretations

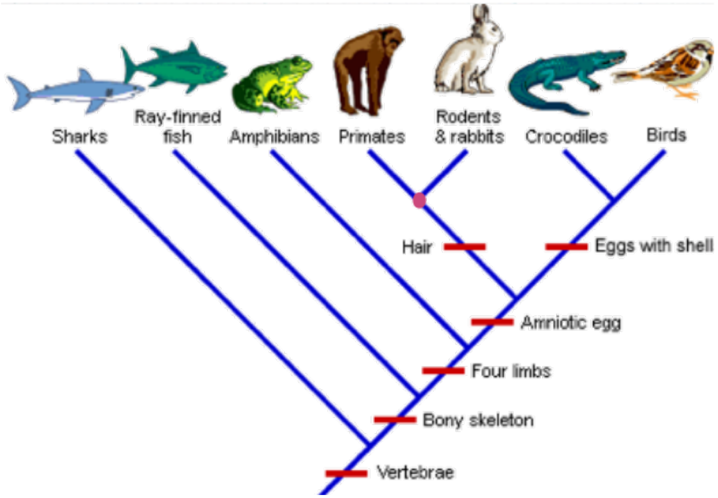
- The truth/falsehood of an FOL formula depends on interpretations (just as in PL).
- Need to define whether P is true in I ($I \vDash P$, or $I(P) = T$) by induction on P:
  - Atom: $I \vDash R(x, y)$ iff $\big(I(x), I(y)\big) \mapsto T$ is in $I(R)$
  - AND: $I \vDash P \wedge Q$ iff $I \vDash P$ and $I \vDash Q$
  - OR: $I \vDash P \vee Q$ iff $I \vDash P$ or $I \vDash Q$
  - NOT: $I \vDash \neg P$ iff $I \nvDash F$
- Note: $I\big(f(t_1, \dots, t_n)\big) = I(f)\big(I(t_1), \dots, I(t_n)\big)$

Example 1

F: $z \leq x \land z \leq y$

Interpretation:

- $x = \text{“}Primates\text{”}$
- $y = \text{“}Rodent\text{”}$
- $z =$



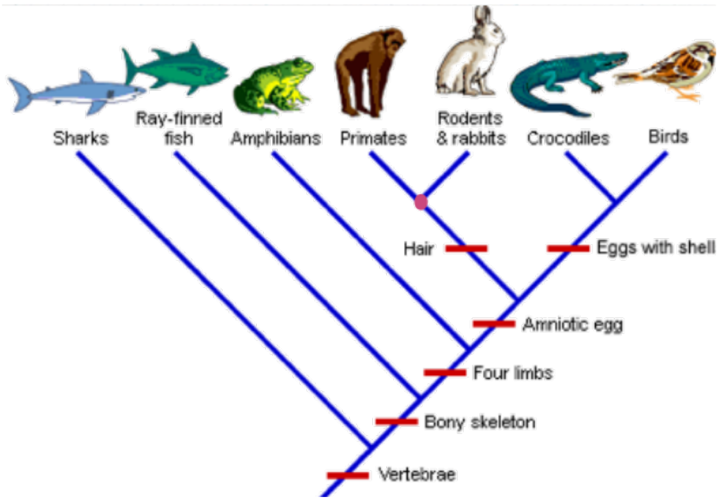
- Is F true in this interpretation?

Example 2

F: $z \leq x \land z \leq y$

Interpretation:

- $x =$ "$Primates$"
- $y =$ "$Rodent$"
- $z =$ "$Crocodiles$"



- Is F true in this interpretation?

Semantics of ∀ and ∃

Extending $I(P)$ to formulas with quantifiers:

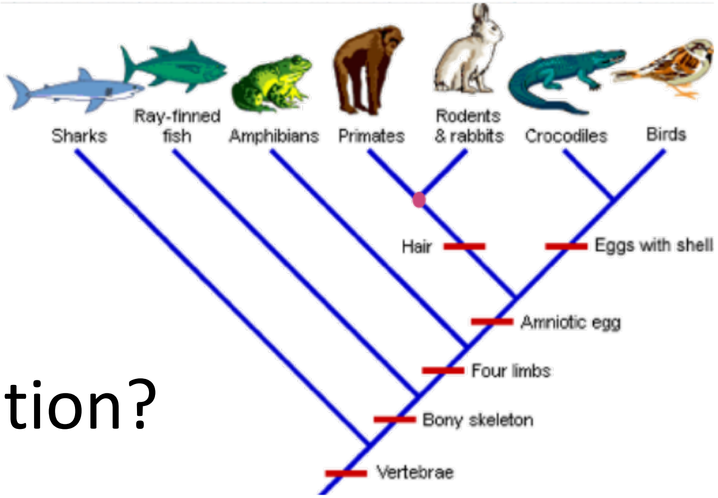
- Forall: $I \vDash \forall x. P \quad \text{iff} \quad I[a/x](P) = T$ for all a in D
- Exists: $I \vDash \exists x. P \quad \text{iff} \quad I[a/x](P) = T$ for some a in D

- Note: $I[a/x] = \{x \mapsto a \dots\}$

Example 1

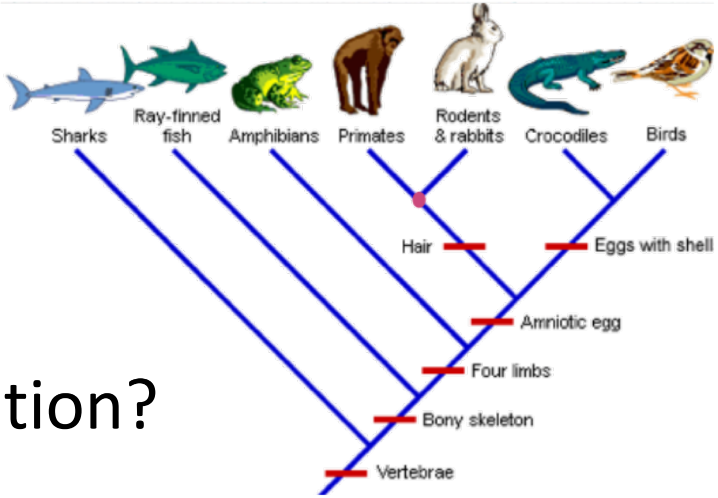
- F: $\exists x, y, z. (z \leq x \wedge z \leq y)$

- Is F true in this interpretation?



Example 2

- $F: \forall x, y, z. (x \le y \land y \le z \to x \le z)$



- Is F true in this interpretation?

Exercise 1

- Formally express that every two species have a common ancestor.
- Show that this is true in the phylogeny tree interpretation.

Exercise 2

Consider the following interpretation (social network):

- Relations: $Friends/2$
- $D = \{people\}$
- $I(Friends) = \{\ (x, y) : \ x \text{ is a friend of } y\ \}$

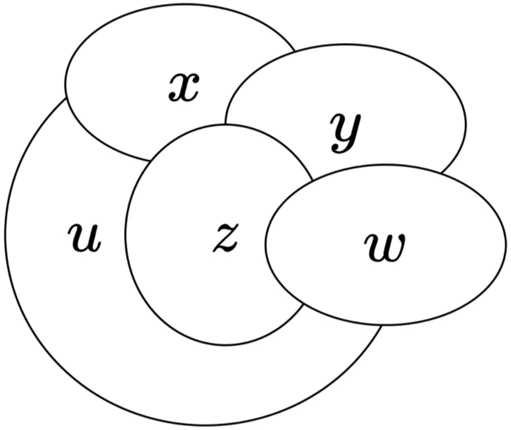
Express (the famous) six-degree of separation:

"The distance between any two people in this graph is six or less"

Exercise 3

Show that it is possible to have 3-coloring for this graph

- Relation: $=/2$
- Variables: u, w, x, y, z
- $D = \{R, G, B\}$



Exercise 4

In the linear arithmetic $(N, +)$ model, argue the following formulas are true:

- $\forall x \exists y. y > x$
- $\forall x \exists y. y + y = x \lor y + y + 1 = x$

Exercise 5

Consider the interpretation:

$$D = \{ 0,1,\dots,8 \}$$

$$I(R) = \{ (x,y) \mid y = x - z, z = 1,2,3 \}$$

Prove that the formula is false:

$$\forall x_1 \exists y_1 \forall x_2 \exists y_2. \big(R(x_1,y_1) \land R(y_1,x_2) \land R(x_2,y_2) \land R(y_2,0)\big)$$

Try SMT solver

```
(set-logic LIA)

(define-fun R ((x Int) (y Int)) Bool
  (or (= y (- x 1)) (= y (- x 2)) (= y (- x 3)))
)

(assert
  (forall ((x1 Int))
    (exists ((y1 Int))
      (forall ((x2 Int))
        (exists ((y2 Int))
          (and
            (R x1 y1)
            (R y1 x2)
            (R x2 y2)
            (R y2 0)
          )
        )
      )
    )
  )
)

(check-sat)
```

Exercise 6

Consider the interpretation:

$$D = \{integer\}$$

$$I(R) = \{\ (x, y) \mid y = x - z, z = 1,2,3\ \}$$

Prove that the formula below is true:

$$\forall x. \Big((\exists w. 4w = x) \rightarrow \forall z \exists y. \big(R(x, z) \rightarrow R(z, y) \land (\exists w. 4w = y) \big) \Big)$$

Note: $4w$ is a "macro" for $w + w + w + w$ (even this is a macro)

Try SMT solver

```
(set-logic LIA)
(define-fun R ((x Int) (y Int)) Bool
  (or (= y (- x 1)) (= y (- x 2)) (= y (- x 3)))
)

(assert
  (forall ((x Int))
    (exists ((w Int))
      (=>
        (= (+ w w w w) x)
        (forall ((z Int))
          (exists ((y Int))
            (=>
              (R x z)
              (and
                (R z y)
                (exists ((w Int)) (= (+ w w w w) y))
              )
            )
          )
        )
      )
    )
  )
)

(check-sat)
```

Satisfiability / Validity / Equivalence

Satisfiability/validity/ (semantic) equivalence

- A formula is satisfiable if it is true in some interpretation
- A formula is valid if it is true in all interpretations
- Two formulas are equivalent if their truth values are the same under all interpretations

Exercises

Show that all the following examples are satisfiable!

$$\exists x. man(x) \land \neg married(x)$$

$$\left(\exists x. man(x)\right) \to \left(\forall y. man(x)\right)$$

$$\left(\forall x. man(x)\right) \to \left(\exists y. man(x)\right)$$

$$\forall x. man(x) \to \max(x)$$

Exercises

Point out valid and invalid formulas!

$$\exists x. man(x) \land \neg married(x)$$

$$\left(\exists x. man(x)\right) \rightarrow \left(\forall y. man(x)\right)$$

$$\left(\forall x. man(x)\right) \rightarrow \left(\exists y. man(x)\right)$$

$$\forall x. man(x) \rightarrow \max(x)$$

More exercises

- Prove that the following formulas are valid

$$\forall x. \big(Man(x) \rightarrow Mortal(x)\big) \wedge Man(Socrates)$$

$$\rightarrow Mortal(Socrates)$$

- Prove that the following formula is not valid

$$(\exists x. P(x) \wedge \exists x. R(x)) \rightarrow \big(\exists x. P(x) \wedge R(x)\big)$$

Some equivalences

- Equivalences from boolean logic carry over to FOL
- New ones, e.g. De Morgan's Laws for FOL:

$$\neg\exists x.\neg F \quad \equiv \quad \forall x.F$$

$$\neg\forall x.\neg F \quad \equiv \quad \exists x.F$$

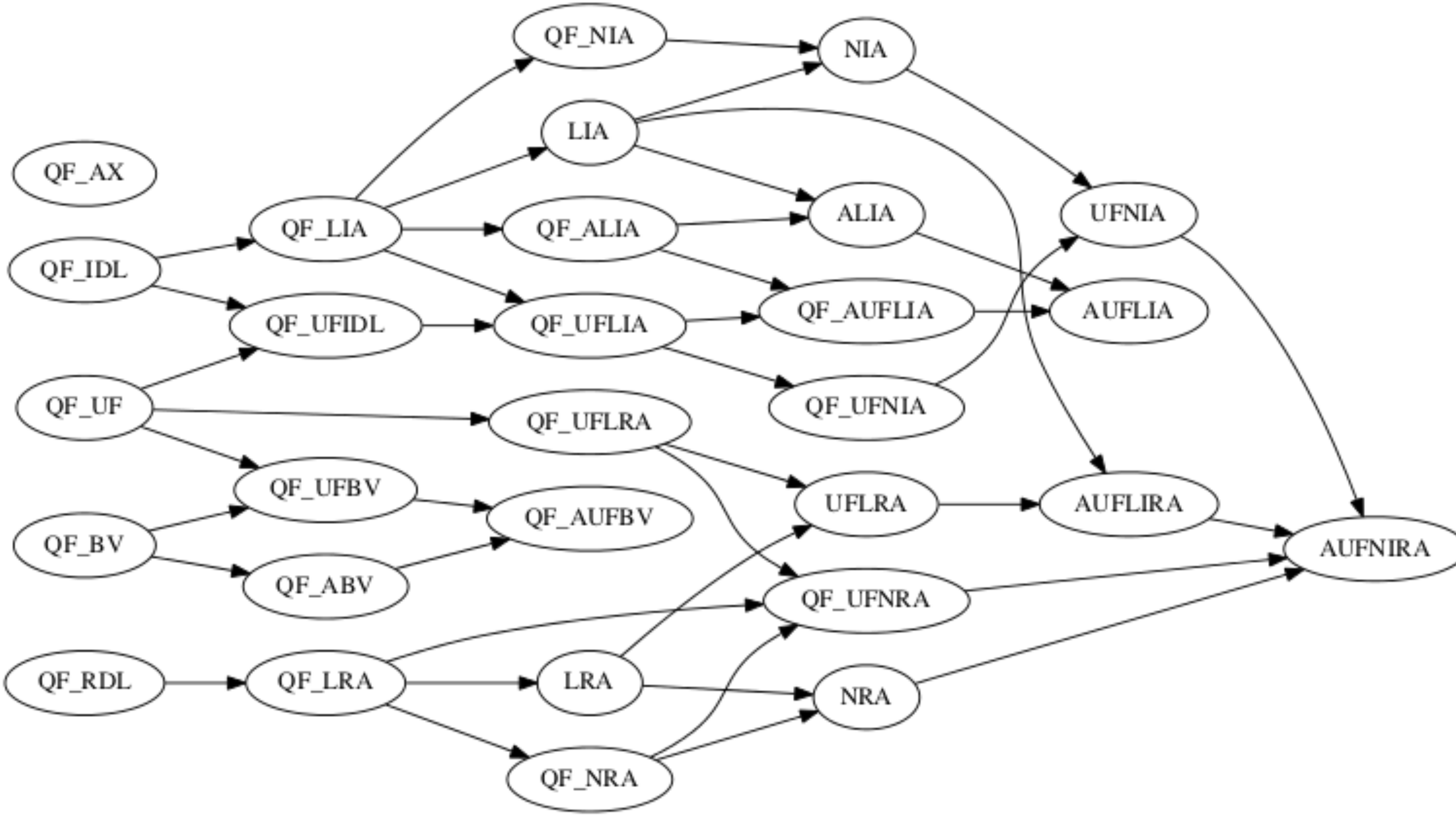
Exercise

Prove De Morgan‘s Laws!

Ponderables

- What’s the connection between satisfiability/validity/equivalence?
- Could you give an algorithm for checking satisfiability/validity/equivalence?
- What about the same problem over “finite interpretations”? Over “finite interpretations of size k”?

Roadmap for FOL after this



Some more tutorial questions

Free variables

Define this by induction on formula F:

- $free\big(R(x,y)\big) = \{x,y\}$
- $free(F \land F') = free(F) \cup free(F')$
- $free(\neg F) = free(F)$
- $free(\forall x. F) = free(F) \backslash \{x\}$
- $free(\exists x. F) = free(F) \backslash \{x\}$

Exercises

What are the free variables of the formulas:

$$\exists x.\, even(x)$$

$$\big(\forall x.\, R(x)\big) \land Z(x)$$

More equivalences

If x is not free in the formula G, then:

$$(\forall x. F) \land G \equiv \forall x. (F \land G)$$

$$(\exists x. F) \land G \equiv \exists x. (F \land G)$$

Software Verification

Outline

- What is program verification
- Hoare logic
- Weakest precondition
- Other ways to verify program
  - Static single assignment form
  - Symbolic execution

Assertions

- A time snapshot of a program execution is a state, which maps program variables to their values at that time.
- A program execution is an evolution of states.
- An assertion is a statement about states of a program.

$$x < 2^{51} \land y < 2^{15}$$

$$res \equiv (x \cdot y) \mod 2^{255-19}$$

- Most interesting assertions can be expressed in FOL.

Program verification

- Prove program property by formulating:
  - Assertions as pre-/post-conditions in FOL
  - Program variables as FOL variables

Pre- and post-conditions

- Put an assertion at the entry point of a program to specify the requirements of inputs: pre-condition
- Put an assertion at the exit point of a program to specify the guarantees of outputs: post-condition

Hoare logic

- Hoare logic is an axiomatic approach to program correctness
- Properties of programs can be verified in a deductive manner: applying inference rules to a set of axioms
- Different program languages may need different inference rules
- It is possible to automate the deductive verification

Hoare triples

- A program C annotated with pre-condition P and post-condition Q is a Hoare triple: $\{ P \} C \{ Q \}$
- Validity of a Hoare triple
  - Partial correctness: If the program starts with a state satisfying P and terminates at a final state, then the final state satisfies Q
  - Total correctness: If the program starts with a state satisfying P, then the program must terminate at a final state and the final state satisfies Q
- If a Hoare triple is interpreted as total correctness, it is sometimes written as $\langle P \rangle C \langle Q \rangle$

Specifications

- A program specification can be written as a Hoare triple, plus assertions inserted in the program
- If the Hoare triple can be shown to be valid, then the program satisfies the specification
- For a function that returns a result, we use the variable res to represent the returned result.

Examples

- $\{y \neq 0\}\ \ div(x, y)\ \ \{res = x \,/\, y\}$
- $\{size(ls)\ =\ n\}\, sort(ls, n)\, \{sorted(ls)\ \wedge\ size(ls)\ =\ n\}$
  - size and sorted are first-order functions
- $\{\ x<y\ \wedge\ y<z\ \wedge\ z<w\ \wedge\ w+x=y+z\ \wedge\ x+y=z+w\ \}\ C\ \{\ Q\ \}$
  - always valid for integer variables $x, y, z,$ and w

Be careful of writing specifications

Exercise

- Let max be a function that returns the maximal number between two input numbers. Write a specification of max as precise as possible.
  - { ? } max(x, y) { ? }
- Write the specification of a function that concatenates two integer lists. You may define other functions of list and use them in the specification.
  - list ::= nil | cons(Int, list)

Assignment

$$x := e$$

- Assume that the evaluation of e does not cause any side-effect
- $P[e/x]$: change x to e in P
- Which one is correct?
  - $\{P\}\ x :=\ e\ \{P[e/x]\}$ ✗ $\{x > 0\}\ x := 2\ \{2 > 0\}$
  - $\{Q[e/x]\}\ x :=\ e\ \{Q\}$ ○ $\{2 > 0\}\ x := 2\ \{x > 0\}$

Assignment – more examples

- $\{x - 1 \geq 0\}\ x := x - 1\ \{x \geq 0\}$
- $\{x < x + y\}\ z \coloneqq x\ \{z < z + y\}$
- $\{x \geq x\}\ z \coloneqq x\ \{z \geq x\}$

Assignment – axiom

$$\overline{\{Q[e/x]\}\ x \coloneqq e\ \{Q\}}\ \text{Assign}$$

- No side-effect: only x is changed
- x in post-condition has a new value same as e to satisfy Q
- What if x does not have value same as e?
  - Change x to e would satisfy Q

Multiple assignment

$$x_1, x_2, \ldots, x_n := e_1, e_2, \ldots, e_n$$

where x's are distinct variables

$$\frac{}{\{\, Q[e_1, e_2, \ldots, e_n / x_1, x_2, \ldots, x_n\}\ x_1, x_2, \ldots, x_n \coloneqq e_1, e_2, \ldots, e_n\ \{Q\}}\ \text{MultiAssign}$$

- $Q[e_1, e_2, \ldots, e_n / x_1, x_2, \ldots, x_n]$ is the result of simultaneous substitution
- $(x < y)[y, x / x, y] = (y < x)$

Proof rules

$$\frac{}{\{\ Q[e/x]\ \}\ x := e\ \{\ Q\ \}}\ \text{Assign}$$

$$\frac{\{\ P \wedge B\ \}\ S_1\ \{\ Q\ \} \qquad \{\ P \wedge \neg B\ \}\ S_2\ \{\ Q\ \}}{\{\ P\ \}\ \mathbf{If}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}\ \{\ Q\ \}}\ \text{Conditional}$$

$$\frac{}{\{\ Q\ \}\ \mathbf{skip}\ \{\ Q\ \}}\ \text{Skip}$$

$$\frac{\{\ P \wedge B\ \}\ S\ \{\ Q\ \} \qquad P \wedge \neg B \rightarrow Q}{\{\ P\ \}\ \mathbf{If}\ B\ \mathbf{then}\ S\ \mathbf{fi}\ \{\ Q\ \}}\ \text{If-Then}$$

$$\frac{\{\ P\ \}\ S_1\ \{\ Q\ \} \qquad \{\ Q\ \}\ S_2\ \{\ R\ \}}{\{\ P\ \}\ S_1;\ S_2\ \{\ R\ \}}\ \text{Sequence}$$

$$\frac{\{\ P \wedge B\ \}\ S\ \{\ P\ \}}{\{\ P\ \}\ \mathbf{while}\ B\ \mathbf{do}\ S\ \mathbf{od}\ \{\ P \wedge \neg B\ \}}\ \text{While}$$

$$\frac{P \rightarrow P' \qquad \{\ P'\ \}\ S\ \{\ Q'\ \} \qquad Q' \rightarrow Q}{\{\ P\ \}\ S\ \{\ Q\ \}}\ \text{Consequence}$$

Conditional

$\{\top\}$

If $x < y$ **then**

  $res \coloneqq y$

else

  $res \coloneqq x$

fi

$\{res \geq x \wedge res \geq y\}$

Conditional

$$\frac{}{\{Q[e/x]\}\ x := e\ \{Q\}}\text{Assign}$$

$\{\top\}$
If $x < y$ **then**
  $res := y$
else
  $res := x$
fi
$\{res \geq x \land res \geq y\}$

$\{\top\}$
If $x < y$ **then**
  $\{y \geq x \land y \geq y\}$
  $res := y$
  $\{res \geq x \land res \geq y\}$ — Assign
else
  $\{x \geq x \land x \geq y\}$
  $res := x$
  $\{res \geq x \land res \geq y\}$ — Assign
fi
$\{res \geq x \land res \geq y\}$

Conditional

$$\frac{P \rightarrow P' \quad \{P'\}\, S\, \{Q'\} \quad Q' \rightarrow Q}{\{P\}\, S\, \{Q\}} \text{Consequence}$$

$\{\top\}$

If $x < y$ **then**

$res := y$

else

$res := x$

fi

$\{res \geq x \wedge res \geq y\}$

$\{\top\}$

If $x < y$ **then**

$\{\top \wedge x < y\}$

$\{y \geq x \wedge y \geq y\}$

$res := y$

$\{res \geq x \wedge res \geq y\}$

Consequence

else

$\{\top \wedge x \geq y\}$

$\{x \geq x \wedge x \geq y\}$

$res := x$

$\{res \geq x \wedge res \geq y\}$

Consequence

fi

$\{res \geq x \wedge res \geq y\}$

Conditional

$$\frac{\{P \wedge B\}\ S_1\ \{Q\} \quad \{P \wedge \neg B\}\ S_2\ \{Q\}}{\{P\}\ \textbf{If}\ B\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{Q\}}\ \text{Conditional}$$

$\{\top\}$
If $x < y$ **then**
  $res := y$
else
  $res := x$
fi
$\{res \geq x \wedge res \geq y\}$

$\{\top\}$
If $x < y$ **then**
  $\{\top \wedge x < y\}$
  $\{y \geq x \wedge y \geq y\}$
  $res := y$
  $\{res \geq x \wedge res \geq y\}$
else
  $\{\top \wedge x \geq y\}$
  $\{x \geq x \wedge x \geq y\}$
  $res := x$
  $\{res \geq x \wedge res \geq y\}$
fi
$\{res \geq x \wedge res \geq y\}$

Conditional

While

$$\frac{\{P \wedge B\}\; S\; \{P\}}{\{P\}\; \textbf{while}\;\; B\; \textbf{do}\;\; S\; \textbf{od}\; \{P \wedge \neg B\}}\text{ While}$$

- P in the While rule is a *loop invariant*
- Invariant: an assertion that always holds whenever the program reaches it
- Loop invariants are usually specified manually
- For some classes of assertions, loop invariants can be synthesized

While – example

$$\overline{\{Q[e/x]\}\ x \coloneqq e\ \{Q\}}\ \text{Assign}$$

$\{\ s = \text{""}\ \}$
while $|s| < 10$ **do**
  $s \coloneqq \text{concat}(\text{"a"}, s, \text{"b"})$
od
$\{s \notin L((a+b)^* ba (a+b)^*)\}$

$\{\ s = \text{""}\ \}$

while $|s| < 10$ **do**

  $\{\ \text{concat}(\text{"a"}, s, \text{"b"}) \in L(a^* b^*)\ \}$
  $s \coloneqq \text{concat}(\text{"a"}, s, \text{"b"})$
  $\{\ s \in L(a^* b^*)\ \}$
od

$\{\ s \notin L((a+b)^* ba (a+b)^*)\}$

While – example

$$\frac{P \to P' \quad \{P'\}\, S\, \{Q'\} \quad Q' \to Q}{\{P\}\, S\, \{Q\}}\text{Consequence}$$

$\{\, s = \text{""} \,\}$
while $|s| < 10$ **do**
  $s \coloneqq \text{concat}(\text{"a"}, s, \text{"b"})$
od
$\{s \notin L((a+b)^* ba (a+b)^*)\}$

$\{\, s = \text{""} \,\}$

while $|s| < 10$ **do**
  $\{\, s \in L(a^* b^*) \wedge |s| < 10 \,\}$
  $\{\, \text{concat}(\text{"a"}, s, \text{"b"}) \in L(a^* b^*) \,\}$
  $s \coloneqq \text{concat}(\text{"a"}, s, \text{"b"})$
  $\{\, s \in L(a^* b^*) \,\}$
od

$\{\, s \notin L((a+b)^* ba (a+b)^*)\}$

While – example

$$\frac{\{\, P \land B \,\}\ S\ \{\, P \,\}}{\{\, P \,\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{\, P \land \neg B \,\}}\text{While}$$

$\{\, s = "" \,\}$
while $|s| < 10$ **do**
  $s \coloneqq \text{concat}("a", s, "b")$
od
$\{s \notin L((a+b)^* ba (a+b)^*)\}$

$\{\, s = "" \,\}$
$\{\, s \in L(a^* b^*) \,\}$
while $|s| < 10$ **do**
  $\{\, s \in L(a^* b^*) \land |s| < 10 \,\}$
  $\{\, \text{concat}("a", s, "b") \in L(a^* b^*) \,\}$
  $s \coloneqq \text{concat}("a", s, "b")$
  $\{\, s \in L(a^* b^*) \,\}$
od
$\{\, s \in L(a^* b^*) \land |s| \geq 10 \,\}$
$\{\, s \notin L((a+b)^* ba (a+b)^*)\}$

While – example

$$\frac{P \to P' \quad \{P'\}\, S\, \{Q'\} \quad Q' \to Q}{\{P\}\, S\, \{Q\}}\ \text{Consequence}$$

$\{\, s = \text{""} \,\}$
while $|s| < 10$ **do**
  $s := \text{concat}(\text{"a"}, s, \text{"b"})$
od
$\{s \notin L((a+b)^* ba (a+b)^*)\}$

$\{\, s = \text{""} \,\}$
$\{\, s \in L(a^* b^*) \,\}$
while $|s| < 10$ **do**
  $\{\, s \in L(a^* b^*) \land |s| < 10 \,\}$
  $\{\, \text{concat}(\text{"a"}, s, \text{"b"}) \in L(a^* b^*) \,\}$
  $s := \text{concat}(\text{"a"}, s, \text{"b"})$
  $\{\, s \in L(a^* b^*) \,\}$
od
$\{\, s \in L(a^* b^*) \land |s| \geq 10 \,\}$
$\{\, s \notin L((a+b)^* ba (a+b)^*)\}$

Try SMT solver

```
(set-logic QF_SLIA)
(set-option :incremental true)

(declare-fun s () String)
(define-const a RegLan (str.to_re "a"))
(define-const b RegLan (str.to_re "b"))
;; anbm := L(a*b*)
(define-const anbm RegLan
  (re.++ (re.* a) (re.* b))
)
```

```
;; s="" => s in L(a*b*)
(assert
  (and
    (= s "")
    (not (str.in_re s anbm))
  )
)
(check-sat)
;; s in L(a*b*) AND |s| < 10 => "a"++s++"b" in L(a*b*)
(assert
  (and
    (str.in_re s anbm)
    (< (str.len s) 10)
    (not (str.in_re (str.++ "a" s "b") anbm))
  )
)
(check-sat)
;; s in L(a*b*) AND |s| >= 10 => s not in
L([ab]*ba[ab]*)
(assert
  (and
    (str.in_re s anbm)
    (>= (str.len s) 10)
    (not(not (str.in_re s
      (re.++ (re.union a b) b a (re.union a b))
    )))
  )
)
(check-sat)
```

Exercise

- Complete the proof outline.

$$\{x \geq 0 \land y \geq 0 \land \gcd(x, y) = \gcd(m, n)\}$$

$$\begin{array}{l}
\textbf{while } x \neq 0 \land y \neq 0 \textbf{ do} \\
\quad \textbf{if } x < y \textbf{ then} \\
\quad\quad x, y := y, x \\
\quad \textbf{fi;} \\
\quad x := x - y \\
\textbf{od}
\end{array}$$

$$\left\{\begin{array}{l} \quad (x = 0 \land y \geq 0 \land y = \gcd(x, y) = \gcd(m, n)) \\ \lor\ (x \geq 0 \land y = 0 \land x = \gcd(x, y) = \gcd(m, n)) \end{array}\right\}$$

While – total correctness

- For total correctness, loops must terminate
- How to ensure this in annotations?
  - specify a rank function that decreases after every loop body

$$\frac{\{\,P \wedge B\,\}\ S\,\{\,P\,\} \quad \{\,P \wedge B \wedge t = Z\,\}\ S\,\{\,t < Z\,\} \quad P \wedge B \rightarrow t \geq 0}{\{\,P\,\}\ \mathbf{while}\ B\ \mathbf{do}\ \ S\ \mathbf{od}\,\{\,P \wedge \neg B\,\}}\ \text{While Total}$$

t is a rank function

Rank function – example

$$\frac{\{P \wedge B\}\ S\ \{P\} \quad \{P \wedge B \wedge t = Z\}\ S\ \{t < Z\} \quad P \wedge B \rightarrow t \geq 0}{\{P\}\ \mathbf{while}\ B\ \mathbf{do}\ \ S\ \mathbf{od}\ \{P \wedge \neg B\}}\ \text{While Total}$$

- What is the rank function?

$\{x \geq 0 \wedge y > 0 \wedge x \equiv m\ (\mathrm{mod}\ y)\}$
while $x \geq y$ **do**
  $x := x - y$
od
$\{x \geq 0 \wedge y > 0 \wedge x \equiv m (\mathrm{mod}\ y) \wedge x < y\}$

Rank function – example

$$\frac{\{\,P \land B\,\}\ S\ \{\,P\,\}\ \checkmark \quad \{\,P \land B \land t = Z\,\}\ S\ \{\,t < Z\,\} \quad P \land B \rightarrow t \geq 0}{\{\,P\,\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{\,P \land \neg B\,\}}\ \text{While Total}$$

- What is the rank function?

$\{x \geq 0 \land y > 0 \land x \equiv m \pmod y\}$
while $x \geq y$ **do**
  $\{x \geq 0 \land y > 0 \land x \equiv m \pmod y \land x \geq y \}$
  $\{x - y \geq 0 \land y > 0 \land x - y \equiv m \pmod y\}$
  $x := x - y$ — Assign
  $\{x \geq 0 \land y > 0 \land x \equiv m \pmod y\}$

od
$\{x \geq 0 \land y > 0 \land x \equiv m \pmod y \land x < y\}$

Rank function – example

$$\frac{\{P \wedge B\}\ S\ \{P\}^{\checkmark} \quad \{P \wedge B \wedge t = Z\}\ S\ \{t < Z\}^{\checkmark} \quad P \wedge B \to t \geq 0}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{P \wedge \neg B\}}\ \text{While Total}$$

- What is the rank function?

$\{x \geq 0 \wedge y > 0 \wedge x \equiv m \pmod y\}$
while $x \geq y$ **do**
  $\{x \geq 0 \wedge y > 0 \wedge x \equiv m \pmod y \wedge x \geq y \wedge x - y = Z\}$
  $\{y > 0 \wedge x - y = Z\}$
  $x := x - y$ — Assign
  $\{y > 0 \wedge x = Z\}$
  $\{x - y < Z\}$
od
$\{x \geq 0 \wedge y > 0 \wedge x \equiv m \pmod y \wedge x < y\}$

Rank function – example

$$\frac{\{P \land B\}\ S\ \{P\}^{\checkmark} \quad \{P \land B \land t = Z\}\ S\ \{t < Z\}^{\checkmark} \quad P \land B \to t \geq 0^{\checkmark}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ S\ \textbf{od}\ \{P \land \neg B\}}\ \text{While Total}$$

- What is the rank function?

$\{x \geq 0 \land y > 0 \land x \equiv m \pmod{y} \land x \geq y \to x - y \geq 0\}$

$\{x \geq 0 \land y > 0 \land x \equiv m \pmod{y}\}$

while $x \geq y$ **do**

$\{x \geq 0 \land y > 0 \land x \equiv m(\text{mod}\ y) \land x \geq y \land x - y = Z\}$

$\{y > 0 \land x - y = Z\}$

$x := x - y$

$\{y > 0 \land x = Z\}$

$\{x - y < Z\}$

od

$\{x \geq 0 \land y > 0 \land x \equiv m(\text{mod}\ y) \land x < y\}$

Weakest precondition

- Weakest precondition: the weakest precondition that guarantees termination of the program in a state satisfying the postcondition
- $wp(S, Q)$ is the weakest precondition of a program S and a postcondition Q
- $wp(S,\cdot)$ is a predicate transformer that transforms a postcondition to a weakest precondition
- $wp(S,\cdot)$ can be seen as the semantics of S

Hoare triple as wp

- When total correctness is meant, $\{P\}\ S\ \{Q\}$ is another notation for $P \to wp(S, Q)$
- $P \to wp(S, Q)$: P entails $wp(S, Q)$

Properties of wp

- Axioms:
  - Law of the Excluded Miracle: $wp(S, false) \equiv false$
  - Distributivity of Conjunction: $wp(S, Q_1) \land wp(S, Q_2) \equiv wp(S, Q_1 \land Q_2)$
  - Distributivity of Disjunction for deterministic S: $wp(S, Q_1) \lor wp(S, Q_2) \equiv wp(S, Q_1 \lor Q_2)$
- Derived:
  - Law of Monotonicity: if $Q_1 \rightarrow Q_2$, then $wp(S, Q_1) \rightarrow wp(S, Q_2)$
  - Distributivity of Disjunction for nondeterministic S: $wp(S, Q_1) \lor wp(S, Q_2) \equiv wp(S, Q_1 \lor Q_2)$

wp: Skip and abort

- $wp(\textbf{skip},\ \ Q) = Q$
- $wp(\textbf{abort},\ \ Q) = false$

wp: Assignment and sequence

- $wp(x := e,\ Q) = Q[e/x]$
- $wp(S_1;\ S_2,\ Q) = wp\big(S_1, wp(S_2,\ Q)\big)$

Example

$$wp(x := x - 5;\ x := x * 2, x > 20)$$

$$= wp\big(x := x - 5, wp(x := x * 2, x > 20)\big)$$

$$= wp(x := x - 5, x * 2 > 20)$$

$$= (x - 5) * 2 > 20$$

$$= x > 15$$

wp: Conditional

- $wp(\textbf{if } B \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi}, \ Q)$

$$= \big(B \land wp(S_1, Q)\big) \lor \big(\neg B \land wp(S_2, Q)\big)$$

- $wp(\textbf{if } B \textbf{ then } S \textbf{ fi}, \ Q)$

$$= \big(B \land wp(S, Q)\big) \lor (\neg B \land Q)$$

Example

$$wp(\textbf{if}\ x < y\ \textbf{then}\ x := y\ \textbf{fi}, x \geq y)$$

$$= \big(x{<}y \wedge wp(x := y, x{\geq}y)\big) \vee (\neg(x{<}y) \wedge x{\geq}y)$$

$$= (x < y \wedge y \geq y) \vee (\neg(x < y) \wedge x \geq y)$$

$$\Leftrightarrow \mathrm{T}$$

wp: While

- while B do S od is equivalent to
  - $\textbf{if } B \textbf{ then } (S; \textbf{if } B \textbf{ then } (S; \textbf{if } B \textbf{ then}(\ldots) \textbf{ fi}) \textbf{ fi}) \textbf{ fi}$
- Thus, $wp(\textbf{while } B \textbf{ do } S \textbf{ od}, Q) = (\neg B \wedge Q) \vee$

$$B \wedge wp\big(S, (\neg B \wedge Q)\big) \vee$$

$$B \wedge wp\Big(S, B \wedge wp\big(S, (\neg B \wedge Q)\big)\Big) \ldots$$

- Define
  - $H(Q, 0) \equiv \neg B \wedge Q$
  - $H(Q, k) \equiv B \wedge wp\big(S, H(Q, k-1)\big)$
- $wp(\textbf{while } B \textbf{ do } S \textbf{ od}, Q) = \exists k.\, 0 \leq k \wedge H(Q, k)$

wp: Theorem of while

$$\frac{\{\,P \wedge B\,\}\ S\,\{\,P\,\} \quad \{\,P \wedge B \wedge t = Z\,\}\ S\,\{\,t < Z\,\} \quad P \wedge B \rightarrow t \geq 0}{\{\,P\,\}\ \textbf{while}\ B\ \textbf{do}\ \ S\ \textbf{od}\ \{\,P \wedge \neg B\,\}}\ \text{While Total}$$

Suppose there exist an invariant P and an integer-valued expression t such that

- $P \wedge B \rightarrow wp(S, P)$,
- $P \wedge B \rightarrow (t \geq 0)$, and
- $P \wedge B \wedge (t = Z) \rightarrow \ wp(S, t < Z)$, where Z is a rigid variable.

Then $P \rightarrow wp(\textbf{while}\ B\ \textbf{do}\ S\ \textbf{od},\ \ P \wedge \neg B)$

Verification condition generation

$\{ P \}$

S_1
$\{ R \}$

S_2

S_3
$\{ Q \}$

Verification condition:

Verification condition generation

$\{ P \}$

S_1
$\{ R \}$
$\{ wp(S_2, wp(S_3, Q))\}$
S_2
$\{ wp(S_3, Q) \}$
S_3
$\{ Q \}$

Verification condition:

1. $R \rightarrow wp\big(S_2, wp(S_3, Q)\big)$

Verification condition generation

$\{\ P\ \}$

$\{\ wp(S_1, R)\ \}$

S_1

$\{\ R\ \}$

$\{\ wp(S_2, wp(S_3, Q))\}$

S_2

$\{\ wp(S_3, Q)\ \}$

S_3

$\{\ Q\ \}$

Verification condition:

1. $R \to wp\big(S_2, wp(S_3, Q)\big)$
2. $P \to wp(S_1, R)$

Verification condition generation

$\{ P \}$
$\{ wp(S_1, R$
$\wedge\, wp(S_2, wp(S_3, Q)))\}$
S_1
$\{ R \}$
$\{ wp(S_2, wp(S_3, Q))\}$
S_2
$\{ wp(S_3, Q) \}$
S_3
$\{ Q \}$

Verification condition:

1. $P \rightarrow wp(S_1, R \wedge wp(S_2, wp(S_3, Q)))$

Exercise

- Compute $wp(x := \ x+2;\ y := \ y-2, x+y=0)$
- Compute $wp(\textbf{If}\ x < y\ \textbf{then}\ res := y\ \textbf{else}\ res := x\ \textbf{fi},\ \ res \geq x \wedge \ res \geq y)$

Example wp

$\{\ x > y\ \}$

$\{\ x \geq y\ \}$

$t := x$ $\quad \cdot\,[x/t]$

$\{\ t \geq y\ \}$

$x := y$ $\quad \cdot\,[y/x]$

$\{\ t \geq x\ \}$

$y := t$ $\quad \cdot\,[t/y]$

$\{\ y \geq x\ \}$

$x > y \rightarrow x \geq y$

Example $\boldsymbol{wp}$

- $P \wedge B \rightarrow wp(S, P)$
- $P \wedge B \rightarrow (t \geq 0)$
- $P \wedge B \wedge (t = Z) \rightarrow wp(S, t < Z)$

$$wp(x \coloneqq e, P) = P[e/x]$$

$\{x \geq 100 \wedge y \geq 10\}$

while $x \geq y$ **do**

$x \coloneqq x - y$

od

$\{y > x \wedge x \geq 0\}$

Decide:

$P: x \geq 0 \wedge y > 0, \qquad t: x$

while $x \geq y$ **do**

$\{x - y \geq 0 \wedge y > 0\} \qquad \{x - y < Z\}$

$x \coloneqq x - y$

$\{x \geq 0 \wedge y > 0\} \qquad \{x < Z\}$

od

wp

Example wp

- $P \land B \to wp(S, P)$
- $P \land B \to (t \geq 0)$
- $P \land B \land (t = Z) \to wp(S, t < Z)$

$$wp(x \coloneqq e, P) = P[e/x]$$

$\{x \geq 100 \land y \geq 10\}$

while $x \geq y$ **do**

$x \coloneqq x - y$

od

$\{y > x \land x \geq 0\}$

Decide:

$P: x \geq 0 \land y > 0, \qquad t: x$

while $x \geq y$ **do**
$\{x - y \geq 0 \land y > 0\} \qquad \{x - y < Z\}$
$x \coloneqq x - y$
$\{x \geq 0 \land y > 0\} \qquad \{x < Z\}$
od

Verify followings are valid:

- $(x \geq 0 \land y > 0 \land x \geq y) \to (x - y \geq 0 \land y > 0)$
- $(x \geq 0 \land y > 0 \land x \geq y) \to x \geq 0$
- $(x \geq 0 \land y > 0 \land x \geq y \land x = Z) \to x - y < Z$

Example wp

- $P \wedge B \to wp(S, P)$
- $P \wedge B \to (t \geq 0)$
- $P \wedge B \wedge (t = Z) \to wp(S, t < Z)$

$$P \to wp(\textbf{while } B \textbf{ do } S \textbf{ od}, P \wedge \neg B)$$

$\{x \geq 100 \wedge y \geq 10\}$

while $x \geq y$ **do**

$x := x - y$

od

$\{y > x \wedge x \geq 0\}$

Decide:

$P: x \geq 0 \wedge y > 0, \qquad t: x$

while $x \geq y$ **do**
$\{x - y \geq 0 \wedge y > 0\}$ $\{x - y < Z\}$
$x := x - y$
$\{x \geq 0 \wedge y > 0\}$ $\{x < Z\}$
od

If followings are valid: ...

Then $x \geq 0 \wedge y > 0 \to wp(\textbf{while} \ldots, x \geq 0 \wedge y > 0 \wedge x < y)$

Verify validity of:

- $x \geq 100 \wedge y \geq 10 \to x \geq 0 \wedge y > 0$
- $x \geq 0 \wedge y > 0 \wedge x < y \to y > x \wedge x \geq 0$

Static single assignment form

$\{\, x > y \,\}$
$t := x$
$x := y$
$y := t$
$\{\, y \geq x \,\}$

F:
$x_0 > y_0 \wedge$
$t_1 = x_0 \wedge$
$x_1 = y_0 \wedge$
$y_1 = t_1 \wedge$
$y_1 \ngeq x_1$

Verify satisfiability of F

Example SSA

Decide:

Inv: $x \geq 0 \wedge y > 0$

$\{y \geq 100 \wedge x \geq 10\}$
$x, y := y, x$
$\{x \geq 0 \wedge y > 0\}$
while $x \geq y$ **do**
  $x := x - y$
od
$\{x \geq 0 \wedge y > 0 \wedge x < y\}$
$\{y > x \wedge x \geq 0\}$

pre–condition–check: $(y_0 \geq 100 \wedge x_0 \geq 10) \wedge (x_1 = y_0) \wedge (y_1 = x_0) \wedge \neg(x_1 \geq 0 \wedge y_1 > 0)$

Inv–check: $(x_0 \geq 0 \wedge y_0 > 0 \wedge x_0 \geq y_0) \wedge (x_1 = x_0 - y_0) \wedge \neg(x_1 \geq 0 \wedge y_0 \geq 0)$

post–condition–check: $(x_0 \geq 0 \wedge y_0 > 0 \wedge x_0 < y_0) \wedge \neg(y_0 > x_0 \wedge x_0 \geq 0)$

Symbolic execution

$\{ x > y \}$	$x \mapsto a, y \mapsto b, t \mapsto c$
$t := x$	$x \mapsto a, y \mapsto b, t \mapsto a$
$x := y$	$x \mapsto b, y \mapsto b, t \mapsto a$
$y := t$	$x \mapsto b, y \mapsto a, t \mapsto a$
$\{ y \geq x \}$	$a > b \rightarrow a \geq b$

Symbolic execution – example

Decide:

Inv: $x \geq 0 \wedge y > 0$

pre-condition-check: $(a \geq 100 \wedge b \geq 10)$

$\wedge \neg(b \geq 0 \wedge a > 0)$

$\{y \geq 100 \wedge x \geq 10\}$ $x \mapsto a, y \mapsto b$

$x, y := y, x$ $x \mapsto b, y \mapsto a$

$\{x \geq 0 \wedge y > 0\}$ $x \mapsto a', y \mapsto b'$

while $x \geq y$ **do** $x \mapsto a', y \mapsto b', a' \geq b'$

$x := x - y$ $x \mapsto a' - b', y \mapsto b', a' \geq b'$

od

$\{x \geq 0 \wedge y > 0 \wedge x < y\}$ $x \mapsto a'', y \mapsto b''$

$\{y > x \wedge x \geq 0\}$ $x \mapsto b'', y \mapsto a''$

Inv-check: $(a' \geq 0 \wedge b' > 0) \wedge$

$(a' \geq b') \wedge$ ← path-condition

$\neg(a' - b' \geq 0 \wedge b' > 0)$

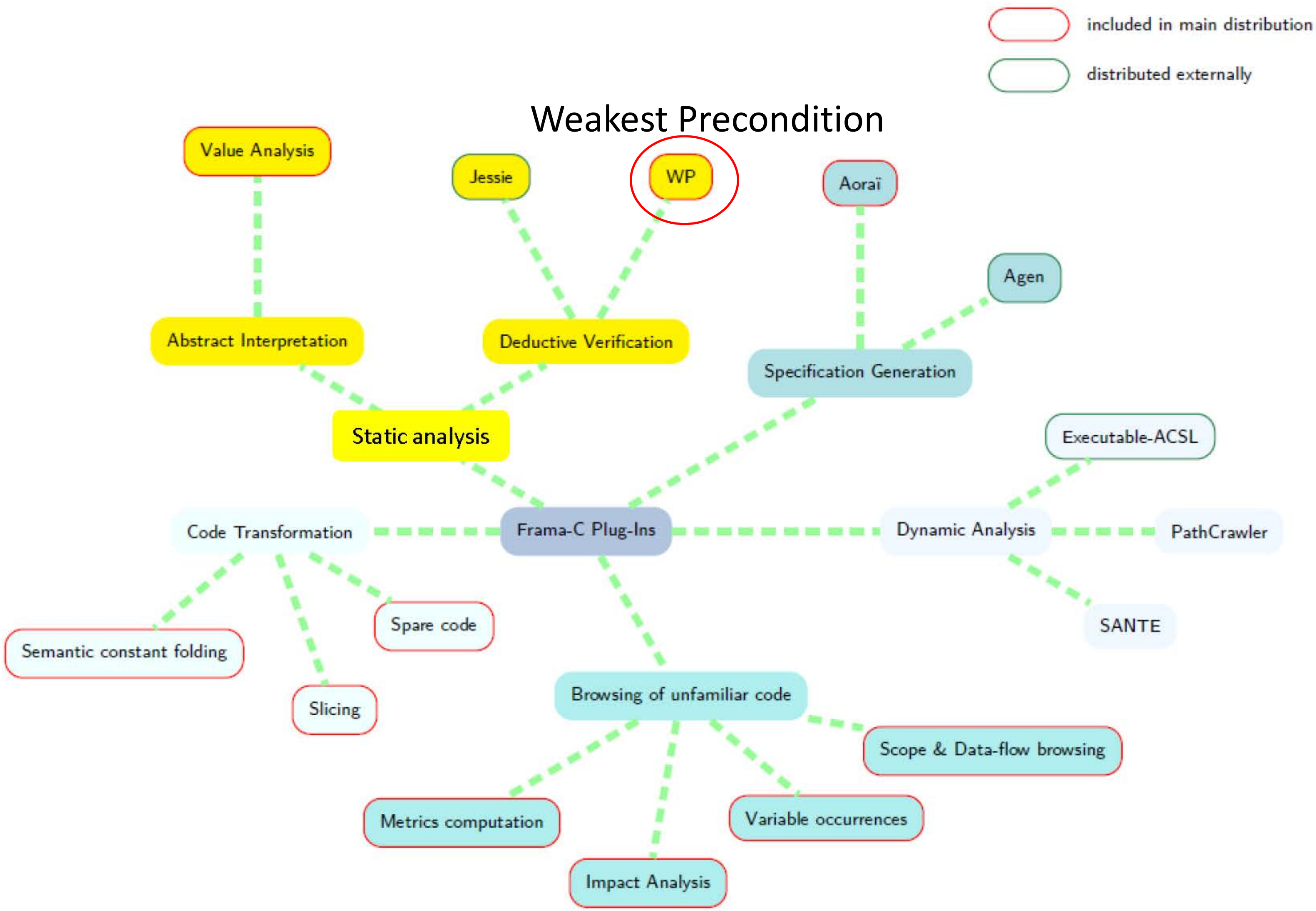
post-condition-check: $(a'' \geq 0 \wedge b'' > 0 \wedge a'' < b'') \wedge \neg(b'' > a'' \wedge a'' \geq 0)$

frama-C

What is Frama-C?

- **Frama-C** is **FRAM**ework for St**A**tic of **C** language
- Build upon
  - A **core** to read C files and build **Abstract Syntax Trees**
  - A set of **plug-ins** to do **static analyses** and **annotate** those syntax trees
  - **Collaboration** of plug-ins
    - A plug-in can **use** the analysis of **another** plug-in
- Purposes
  - **Static analyses** of C code
  - **Transformation** of C code
  - Framework to **build tools** analyzing and manipulating C code
    - New plug-ins programmed in **OCaml** language

Frama-C plugins



INTERLUDE: WHY DOING FORMAL VERIFICATION?

Questions on a simple program

- What **does** the following program?
- Is it **correct**?

```
int abs(int x){
  if (x < 0)
    return -x;
  else
    return x;
}
```

Answers on a simple program

- The program computes the **absolute value** of x
- It is **buggy**!
  - If **$x == -2^{31}$**, 2^{31} cannot be represented in binary two's complement!
    - C's int goes from -2^{31} (-2147483648) to $2^{31}-1$ (2147483647)
- A formal tool (like Frama-C) can **catch** it
  - "**frama-c-gui** -wp -wp-rte abs.c"
  - **Systematically**!!
    - Of course a programmer **knows** about such issues…
    - … but he might **forget** it while doing more complex things

```
int abs(int x)
{
  int __retres;
  if (x < 0) {
    /*@ assert rte: signed_overflow: -2147483647 ≤ x; */
    __retres = - x;
    goto return_label;
  }
  else {
    __retres = x;
    goto return_label;
  }
  return_label: /* internal */ return __retres;
}
```

Cannot be proved

THE NOTION OF “CONTRACT”

The notion of “contract”

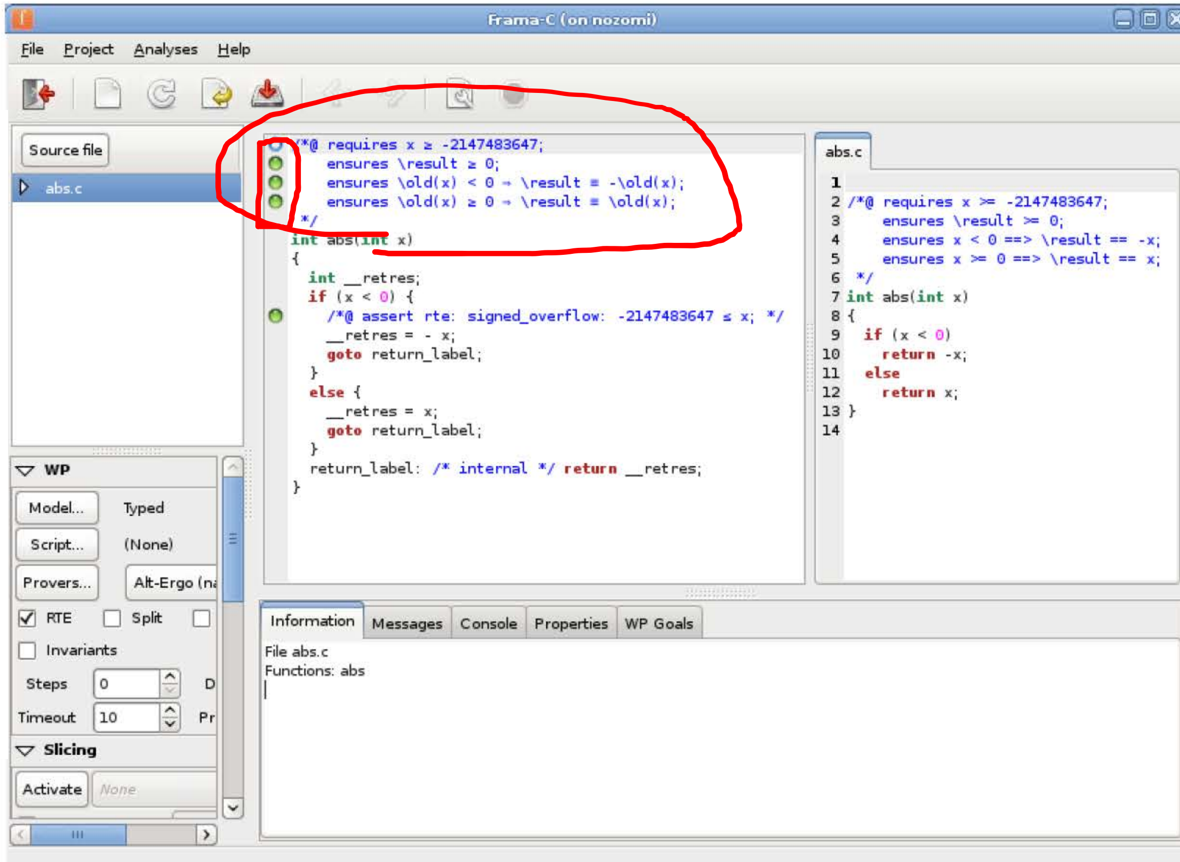
- **Contract** of a function defines
  - What the function **requires** from the outside world
  - What the function **ensures** to the outside world
    - Provided the “requires” part is fulfilled!
- Similar to **business** contract
- Going back to our **abs()** function
  - abs() requires that **$x > -2^{31}$**: `requires x >= - 2147483647;`
  - abs() ensures that
    - Its result is **positive**: `ensures \result >= 0;`
    - Its result is **–x if x is negative**, x otherwise:
      - `ensures x < 0 ==> \result == -x;`
      - `ensures x >= 0 ==> \result == x;`
  - “**\result**” denotes function result
  - Using Frama-C **notation**:

Formal annotation

```
/*@ requires x >= -2147483647;
    ensures \result >= 0;
    ensures x < 0 ==> \result == -x;
    ensures x >= 0 ==> \result == x;
 */
```

Use of Frama-C/WP tool on abs()

- Call with "**frama-c-gui -wp -wp-rte** abs.c"
  - **-wp**: call WP plug-in
  - **-wp-rte**: call RTE plug-in that inserts additional checks for Run Time Errors



BASIC USE OF FRAMA-C/WP THROUGH EXAMPLES

Function call and contract

- A contract is an "**opaque**" specification of function behavior
  - Function **callers** only see the **contract**
    - Contract **considered correct** even if not proved
  - If **no** contract… unknown behavior! (default contract)
- **DEMO** on call.c: "frama-c-gui -wp -wp-rte call.c"
  - Initial state: **all** proved
  - Show farenheit_to_celsius() "**requires**" not fulfilled
    - farenheit_to_celsius() and main() "**ensures**" still **proved**
  - Show farenheit_to_celsius() "**ensures**" not fulfilled
    - main() "**ensures**" still **proved**
- **Everything** should be proved to guarantee the program correct !

– Initial state: all proved (call-proved.c)

```
/*@ requires 0 ≤ fahr ≤ 300;
    ensures \result ≡ (5 * (\old(fahr) - 32)) / 9;
 */
int fahrenheit_to_celsius(int fahr)
{
  int __retres;
  /*@ assert rte: signed_overflow: -2147483648 ≤ fahr - 32; */
  /*@ assert rte: signed_overflow: -2147483648 ≤ 5 * (int)(fahr - 32); */
  /*@ assert rte: signed_overflow: 5 * (int)(fahr - 32) ≤ 2147483647; */
  __retres = (5 * (fahr - 32)) / 9;
  return __retres;
}
```

– Initial state: **all** proved (call-proved.c)

```
/*@ ensures \result ≡ -2000 ∨ (-18 ≤ \result ≤ 300); */
int main(int fahr)
{
  int __retres;
  if (fahr >= 0) {
    if (fahr <= 300) {
      int tmp;
      tmp = fahrenheit_to_celsius(fahr);
      {
        __retres = tmp;
        goto return_label;
      }
    }
    else {
      {
        __retres = -2000;
        goto return_label;
      }
    }
  }
  else {
    {
      __retres = -2000;
      goto return_label;
    }
  }
  return_label: return __retres;
}
```

- Show farenheit_to_celsius() "**requires**" not fulfilled (call-1.c)
  - farenheit_to_celsius() and main() "**ensures**" still **proved**

```c
/*@ ensures \result ≡ -2000 ∨ (-18 ≤ \result ≤ 300); */
int main(int fahr)
{
  int __retres;
  if (fahr >= 0) {
    if (fahr <= 400) {
      int tmp;
      tmp = fahrenheit_to_celsius(fahr);
      {
        __retres = tmp;
        goto return_label;
      }
    }
    else {
      {
        __retres = -2000;
        goto return_label;
      }
    }
  }
  else {
    {
      __retres = -
      goto return_
    }
  }
  return_label: re
}
```

```c
/*@ requires 0 ≤ fahr ≤ 300;
    ensures \result ≡ (5 * (\old(fahr) - 32)) / 9;
 */
int fahrenheit_to_celsius(int fahr)
{
  int __retres;
  /*@ assert rte: signed_overflow: -2147483648 ≤ fahr - 32; */
  /*@ assert rte: signed_overflow: -2147483648 ≤ 5 * (int)(fahr - 32); */
  /*@ assert rte: signed_overflow: 5 * (int)(fahr - 32) ≤ 2147483647; */
  __retres = (5 * (fahr - 32)) / 9;
  return __retres;
}
```

– Show farenheit_to_celsius() "**ensures**" not fulfilled (call-2.c)

- main() "**ensures**" still **proved**

```
/*@ ensures \result ≡ -2000 ∨ (-18 ≤ \result ≤ 300); */
int main(int fahr)
{
  int __retres;
  if (fahr >= 0) {
    if (fahr <= 300) {
      int tmp;
      tmp = fahrenheit_to_celsius(fahr);
      {
        __retres = tmp;
        goto return_label;
      }
    }
    else {
      {
        __retres = -2000;
        goto return_label;
      }
    }
  }
  else {
    {
      __retres = -2000;
      goto return_label;
    }
  }
  return_label: return __retres;
}
```

```
/*@ requires 0 ≤ fahr ≤ 300;
    ensures \result ≡ (5 * (\old(fahr) - 32)) / 9;
 */
int fahrenheit_to_celsius(int fahr)
{
  int __retres;
  __retres = 0;
  return __retres;
}
```

Old and new values, pointers: swap()

- In a contract, need to express:
  - **Validity** of pointers
  - For a variable x, value of x at function **entrance** and **exit**
- **Informal** specification
  - "Exchange two integer values pointed by pointers"
  - **Prototype**: `void swap(int *a, int *b)`
- What is swap() **formal** specification?
  - **Requires**: the pointers need to be **valid**
    - "**\valid(a)**": pointer a is valid
  - **Ensures**: the pointed values are **swapped**
    - "**\old(a)**": value of a at function **entrance** (in function contract ensures)
    - "**a**": value of a at function **exit**

swap() contract and code

- **Contract** and **code**

```
/*@ requires \valid(a) && \valid(b);
    ensures (*a == \old(*b) && *b == \old(*a));
 */
void swap(int *a, int *b){
  int tmp;

  tmp = *a;
  *a = *b;
  *b = tmp;
}
```

- **DEMO**: “frama-c-gui -wp -wp-rte swap.c”

Side note: Frama-C operators in specification

- Several **operators** useful in specification
  - Similar to **C** notation

Operator	Informal meaning	Formal meaning (C notation)
!p	**NOT** p	!p
p && q	p **AND** q	p && q
p || q	p **OR** q	p || q
p ==> q	**IF** p **THEN** q	(p ? q : 1)
p <==> q	p **IF AND ONLY IF** q	p == q

- No logical "IF p THEN q1 **ELSE** q2"
  - Use "(**p ==>** q1) **&&** (**!p ==>** q2)" instead
  - Or more simply "p **?** q1 **:** q2"

swap() variation: two elements in an array

- **Informal** specification
  - "In array a[] of size n, exchange array elements indexed by n1 and n2"
- **Prototype**:
  - `void array_swap(int n, int a[], int n1, int n2)`
- What is its **formal** specification?
  - The indexes are within array **bounds**
    - `requires n >= 0 && 0 <= n1 < n && 0 <= n2 < n;`
  - The array a[] is **valid** memory area up to cell number n
    - `requires \valid(a+(0..n-1));` (similar to `&a[0]` valid, …, `&a[n]` valid)
  - The indexed values are **swapped**
    - `ensures (a[n1] == \old(a[n2]) && a[n2] == \old(a[n1]));`

array_swap() contract and code

- **Contract** and **code**

```
/*@ requires n >= 0 && 0 <= n1 < n && 0 <= n2 < n;
    requires \valid(a+(0..n-1));
    ensures (a[n1] == \old(a[n2]) && a[n2] == \old(a[n1]));
 */
void array_swap(int n, int a[], int n1, int n2){
  int tmp;

  tmp = a[n1];
  a[n1] = a[n2];
  a[n2] = tmp;
}
```

- **DEMO**: “frama-c-gui -wp -wp-rte array_swap.c”

A MORE COMPLEX EXAMPLE WITH WP: FIND()

find() specification

- **Informal** specification
  - "Return the index of an occurrence of v in a[]"
  - "Array a[] is of size n, value v and n are integers"

- **Prototype**:

```
int find(int n, const int a[], int v)
```

- What is its **formal** specification**?**
  - We will elaborate it through some unit **tests**

Case 1: find() finds v in a[]

- **Informal** specification
  - "Return the index of an occurrence of v in a[]"
  - "Array a[] is of size n, value v and n are integers"
- **Prototype**:

```
int find(int n, const int a[], int v)
```

- find() **finds v** in a[]

```
int a[5] = { 9, 7, 8, 9, 6 };

int const f1 = find(5, a, 8);
assert(f1 == 2);
```

- **Formally**

```
ensures 0 <= \result < n ==> a[\result] == v;
```

Case 2: find() does not find v in a[]

- **Informal** specification
  - "Return the index of an occurrence of v in a[]"
  - "Array a[] is of size n, value v and n are integers"
  - "**Returns -1 if v is not found**"
- Prototype:

```
int find(int n, const int a[], int v)
```

- find() **does not find v** in a[]

```
int a[5] = { 9, 7, 8, 9, 6 };

int const f2 = find(5, a, 15);
assert(f2 == -1);
```

- **Formally**
  - If find() returns -1, then
    - for all index i, if i is in a[] bounds then a[i] != v

```
ensures \result == -1
          ==> (\forall integer i; 0 <= i < n ==> a[i] != v);
```

Side note: types used in ACSL annotations

- In ACSL, **distinction** between C program and mathematical **types**

C program type	Mathematical type
`int, short`	`integer` (•)
`float, double`	`real` (•)

- Usually one uses mathematical types for annotations
  - "`\forall integer i; ...`"
    - And not "`\forall int i; ...`"
    - It simplifies generated Verification Condition (not need to add restrictions on `int` range)

Case 3: find() does not modify a[]

- Would it be a **valid** find()**?**

```
int find(int n, int a[], int v){
 if (n > 0) {
    a[0] = v;
    return 0;
  } else
    return -1;
}
```

- We can express it formally
  - `assigns \nothing;`
  - Note: “**const**” expressed it formally but Frama-C does **not understand** “const”

Case 4: valid input and returned values

- **Informal** specification
  - “Array a[] is of size n, value v and n are integers”

- **Formal** specification?
  - `requires 0 <= n && \valid(a+(0..n-1));`

- **Informal** specification
  - “find() result is between -1 and n (excluded)

- **Formal** specification?
  - `ensures -1 <= \result < n;`

Wrap-up: find() formal contract

```
/*@ requires 0 <= n && \valid(a+(0..n-1));
    assigns \nothing;
    ensures \result == -1
          ==> (\forall integer i; 0 <= i < n ==> a[i] != v);
    ensures 0 <= \result < n ==> a[\result] == v;
    ensures -1 <= \result < n;
 */
```

find() code

- **DEMO**: how to **prove** find() code?
  - “frama-c-gui -wp -wp-rte find.c”

```
/*@ requires 0 <= n && \valid(a+(0..n-1));
    assigns \nothing;
    ensures \result == -1
            ==> (\forall integer i; 0 <= i < n ==> a[i] != v);
    ensures 0 <= \result < n ==> a[\result] == v;
    ensures -1 <= \result < n;
 */
int find(int n, const int a[], int v){
  int i;

  for (i=0; i < n; i++) {
    if (a[i] == v) {
      return i;    }
  }

  return -1;
}
```


Loops: how to handle them?

- Main rule: **loops** are "**opaque**"
  - So one needs to **add** needed **annotations** to help automatic provers prove desired properties
  - loop **invariant**, loop **assigns**, loop **variant**

- Loop **invariant**: property always true in a loop
  - Should be **true** at loop **entry**
  - Should be **true** at each loop **iteration**
    - Even if **no** iterations are possible
  - Should be true at loop **exit**

Example of loop invariant (1/2)

- "Loop index is between **0** and **n** (inclusive)"

```
/*@ requires 0 <= n && \valid(a+(0..n-1));
    assigns \nothing;
    ensures \result == -1
           ==> (\forall integer i; 0 <= i < n ==> a[i] != v);
    ensures 0 <= \result < n ==> a[\result] == v;
    ensures -1 <= \result < n;
 */

int find(int n, const int a[], int v){
  int i;

/*@
    loop invariant 0 <= i <= n;


*/
  for (i=0; i < n; i++) {
    if (a[i] == v) {
      return i;    }
  }

  return -1;
}
```

Example of loop invariant (2/2)

- "**Up to** index i, value v is still **not found**"

```
/*@ requires 0 <= n && \valid(a+(0..n-1));
    assigns \nothing;
    ensures \result == -1
           ==> (\forall integer i; 0 <= i < n ==> a[i] != v);
    ensures 0 <= \result < n ==> a[\result] == v;
    ensures -1 <= \result < n;
 */

int find(int n, const int a[], int v){
  int i;

/*@
    loop invariant 0 <= i <= n;
    loop invariant \forall integer j; 0 <= j < i ==> a[j] != v;

*/
  for (i=0; i < n; i++) {
    if (a[i] == v) {
      return i;    }
  }

  return -1;
}
```

We build progressively the desired property

Loop assigns and loop variant

- Loop **assigns**: what is assigned within the loop
- Loop **variant**: to prove **termination**
  - Show a metric **strictly decreasing** at each loop iteration and **bounded** by 0

```
int find(int n, const int a[], int v){
  int i;

/*@ loop invariant 0 <= i <= n;
    loop invariant \forall integer j; 0 <= j < i ==> a[j] != v;
    loop assigns i;
    loop variant n - i;
 */
  for (i=0; i < n; i++) {
    if (a[i] == v) {
      return i;     }
  }

  return -1;
}
```

find() final proved code

- “frama-c-gui -wp -wp-rte find-proved.c”

```
/*@ requires 0 <= n && \valid(a+(0..n-1));
    assigns \nothing;
    ensures \result == -1
           ==> (\forall integer i; 0 <= i < n ==> a[i] != v);
    ensures 0 <= \result < n ==> a[\result] == v;
    ensures -1 <= \result < n;
 */

int find(int n, const int a[], int v){
  int i;

/*@ loop invariant 0 <= i <= n;
    loop invariant \forall integer j; 0 <= j < i ==> a[j] != v;
    loop assigns i;
    loop variant n - i; */
  for (i=0; i < n; i++) {
    if (a[i] == v) {
      return i;    }
  }

  return -1;
}
```

A note on proof with WP

- **More** annotations than code!
  - **8** lines of **code**
  - **10** lines of **annotations**
- Because what we prove is **complicated**
  - A loop, in **all** possible cases!
- It corresponds to **exhaustive** test!

```
/*@ requires 0 <= n && \valid(a+(0..n-1));
   assigns \nothing;
   ensures \result == -1
        ==> (\forall integer i; 0 <= i < n ==> a[i] != v);
   ensures 0 <= \result < n ==> a[\result] == v;
   ensures -1 <= \result < n;
 */

int find(int n, const int a[], int v){
 int i;

/*@ loop invariant 0 <= i <= n;
   loop invariant \forall integer j; 0 <= j < i ==> a[j] != v;
   loop assigns i;
   loop variant n - i; */
 for (i=0; i < n; i++) {
   if (a[i] == v) {
    return i;   }
 }

 return -1;
}
```

BEHAVIORS: CLEAN CONTRACTS

find() contract using behaviors

- "frama-c-gui -wp -wp-rte find-behavior.c"

```
/*@ requires 0 <= n && \valid(a+(0..n-1));
    assigns \nothing;

    behavior found:
      assumes \exists integer i; 0 <= i < n && a[i] == v;
      ensures a[\result] == v;

    behavior not_found:
      assumes \forall integer i; 0 <= i < n ==> a[i] != v;
      ensures \result == -1;

    complete behaviors;
    disjoint behaviors;
 */
```

Array contains v at an index i

In that case return the correct index

Array does not contain v for all possible indexes i

In that case return -1

We cover all behaviors

All behaviors consider different cases

How to write clean contracts?

- Important to write **clean** contracts
  - Improve **readability**: contract is a readable **specification**
    - Help **understand** the code (e.g. in code review)
    - But such specification can be **mechanically** checked!
      - **No** more out-dated comments
  - Help proofs
- "**Behaviors**" can be use to separate several cases
  - **Name** each behavior
  - Give a "**sub-contract**" for each behavior
    - assumes, requires, ensures
- **Bonus**: one can additionally **check** that all behaviors…
  - …Cover **all** possible inputs (**complete** behaviors)
  - …Cover **different** cases (**disjoint** behaviors)

Side note: \exists and \forall operators

- To express something over a **range** of values
- Examples
  - `int a[5] = {1, 5, 3, 2, 1};`
  - `\exists integer i; 0 <= i < 5 && a[i] == 1;`

i	-1	0	1	2	3	4	5
a[i]	?	1	5	3	2	1	?
0 <= i < 5	✗	✓	✓	✓	✓	✓	✗
a[i] == 1	✗	✓	✗	✗	✗	✓	✗

  - `\forall integer i; 0 <= i < 5 ==> a[i] != 4;`

i	-1	0	1	2	3	4	5
a[i]	?	1	5	3	2	1	?
0 <= i < 5	✗	✓	✓	✓	✓	✓	✗
a[i] != 4	✗	✓	✓	✓	✓	✓	✗

Side note: opposite expressions

- **Opposite** expressions: 1st example
  - `int a[5] = {1, 5, 3, 2, 1};`

`\exists index i; a[i] == 1`

`\forall index i; a[i] != 1`

i	0	1	2	3	4	
a[i]	1	5	3	2	1	
a[i] == 1	✓	✗	✗	✗	✓	True ✓
a[i] != 1	✗	✓	✓	✓	✗	False ✗

- Still **opposite** expressions (with proper indexing)
  - `\exists integer i; 0 <= i < n && a[i] == v;`

  vs.
  - `\forall integer i; 0 <= i < n ==> a[i] != v;`

Homework: LRU Cache (lruCache_0.c)

```c
void enqueue(int* queue, int size, int page){
    int pos = -1; // position of page in queue
    int i = 0;

    for(; i<size; ++i){
        if(queue[i] == page){
            pos = i;
            break;
        }
    }


    int start = -1;

    if(pos!=-1){
        start = pos+1;
        queue[pos] = -1;
    }
    else{
        start = 1;
    }

    i = start;
    for(; i<size; ++i){
        queue[i-1] = queue[i];
        queue[i] = -1;
    }
    queue[size-1] = page;
}
```

```
/*@ predicate Unique{L}(int *a, integer size) =
   \forall integer i,j; 0 <= i < j < size && a[i]!=-1 &&
a[j]!=-1 ==> a[i] != a[j] ;

  requires 0<size<2147483645 && page >= 0;
  requires \valid( queue+(0..size-1) );
  requires Unique(queue, size);
  ensures queue[size-1] == page;
  ensures Unique(queue, size);
*/
```