Program verification with Why3

Marc Schoolderman

February 7, 2019

Preliminaries

The first part of this course is on Why3

- Main sessions: thursdays, 8:30 HG00.068

- Tutorial hour: next wednesday, 10:30 MERC I 00.28
  Slot will not always be used – check announcements!

Useful if you remember something from:

- Mathematical Structures
- Assertion and Argumentation
- Semantics and Correctness
- Functional Programming

Practicalities

Reference materials
Why3 Tutorial by J.C. Filliâtre:

`why3.lri.fr/vtsa-18/notes-why3.pdf`

Manual:

`why3.lri.fr/manual.pdf`

Exercises: see Brightspace

- Important you do these!
- **Work in pairs!**
- Deadline: tuesday 12:00

Help? Contact: `m.schoolderman@cs.ru.nl`

Motivation

Formal verification

Validation
Writing the correct program

- Are informal requirements captured in a *specification*?

Verification
Writing the program correctly

- Does the program match the specification?

Formal verification

Validation
Writing the correct program

- Are informal requirements captured in a *specification*?

Verification
Writing the program correctly

- Does the program match the specification?

Formal verification
The art of using rigorous, mathematical techniques for verification

- Prove that a program matches a **formal specification**!

The dawn of computing science

1948 Manchester Baby: first programmable computer

The dawn of computing science

1948 Manchester Baby: first programmable computer

1949 Early program proof by Turing for computing $n!$

The dawn of computing science

1948 Manchester Baby: first programmable computer

1949 Early program proof by Turing for computing $n!$

… **crickets** …

The dawn of computing science

1948	Manchester Baby: first programmable computer
1949	Early program proof by Turing for computing $n!$
	… **crickets** …
1967	Floyd: “Assigning Meanings to Programs”
1969	Hoare logic (Axiomatic semantics)
1975	Dijkstra: weakest preconditions

The dawn of computing science

1948 Manchester Baby: first programmable computer

1949 Early program proof by Turing for computing $n!$

… **crickets** …

1967 Floyd: "Assigning Meanings to Programs"

1969 Hoare logic (Axiomatic semantics)

1975 Dijkstra: weakest preconditions

… fast forward …

The dawn of computing science

1948 Manchester Baby: first programmable computer

1949 Early program proof by Turing for computing $n!$

… **crickets** …

1967 Floyd: "Assigning Meanings to Programs"

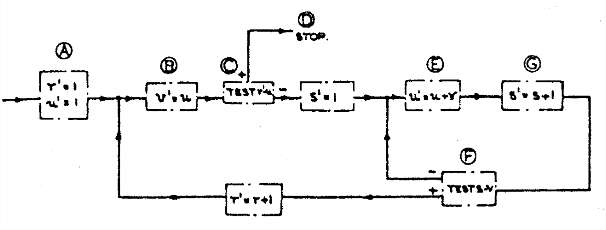
1969 Hoare logic (Axiomatic semantics)

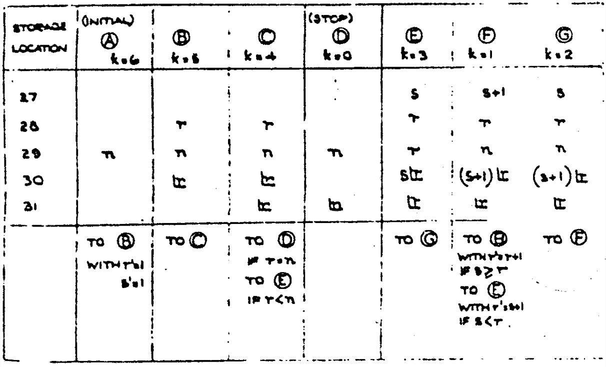
1975 Dijkstra: weakest preconditions

… fast forward …

2019 Formal verification seldomly used

Turing's notes





STORAGE LOCATION	(INITIAL) Ⓐ $k=6$	Ⓑ $k=5$	Ⓒ $k=4$	(STOP) Ⓓ $k=0$	Ⓔ $k=3$	Ⓕ $k=1$	Ⓖ $k=2$
27					s	$s+1$	s
28		r	r		r	r	r
29	n	n	n	n	r	n	n
30		$\underline{\lvert r}$	$\underline{\lvert r}$		$s\underline{\lvert r}$	$(s+1)\underline{\lvert r}$	$(s+1)\underline{\lvert r}$
31			$\underline{\lvert r}$	$\underline{\lvert n}$	$\underline{\lvert r}$	$\underline{\lvert r}$	$\underline{\lvert r}$
	TO Ⓑ WITH $r'=1$ $u'=1$	TO Ⓒ	TO Ⓓ IF $r=n$ TO Ⓔ IF $r<n$		TO Ⓖ	TO Ⓑ WITH $r'=r+1$ IF $s\geq r$ TO Ⓔ WITH $s'=s+1$ IF $s<r$.	TO Ⓕ

Critical software bugs

Testing shows the presence, not the absence of bugs – Dijkstra

1996	Ariane 5: uncaught runtime exception
1999	NASA: confused imperial and metric system
2008	Debian OpenSSL: RNG seeded with well-known data
2009	Toyota "unintended acceleration": stack overflow
2014	Apple SSL: `goto fail`
2014	HeartBleed: buffer overflow
2014	ShellShock: Incorrect input processing (undetected for 25 years)

More stories:

`http://www.cs.tau.ac.il/~nachumd/horror.html`

Bugs in the Java JDK

2006 “Nearly All Binary Searches and Mergesorts are Broken” – Joshua Bloch:[¶]

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

[¶] https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html

Bugs in the Java JDK

2006 "Nearly All Binary Searches and Mergesorts are Broken" – Joshua Bloch:[¶]

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

Will cause overflow if `low+high`$\geq 2^{31}$!

[¶] https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html

Bugs in the Java JDK

2006 “Nearly All Binary Searches and Mergesorts are Broken” – Joshua Bloch:[¶]

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

Will cause overflow if `low+high`$\geq 2^{31}$! Fixing this in C:

```
int mid = low + (high−low)/2
```

[¶] `https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html`

What can we do about this?

Proofs can show the absence of (certain) bugs.

What can we do about this?

Proofs can show the absence of (certain) bugs.

The Bad News
Formal proofs are *hard*, *tedious*, *time-consuming*, and *error-prone*.

What can we do about this?

Proofs can show the absence of (certain) bugs.

The Bad News
Formal proofs are *hard*, *tedious*, *time-consuming*, and *error-prone*.

The Good News
Major advances in *computional power* and *artificial intelligence*

- Interactive theorem provers: Coq, PVS, Isabelle/HOL
- Fully automated provers: Z3, CVC4, E, Alt-Ergo, ...

Another reason to use machine intelligence

Do you trust your own proofs?

Do you trust other people's proofs?

Another reason to use machine intelligence

Do you trust your own proofs?

Do you trust other people's proofs?

Cryptanalysis of OCB2 **– Inoue & Minematsu**

- OCB2: authenticated encryption, 'proven secure' in 2004
- Broken in 2018?!
  `https://eprint.iacr.org/2018/1040.pdf`

Some high-profile stories

AMD K5	Verification of `fdiv` using ACL2 (1995)
Paris Métro	Driverless *Ligne 14* verified using B-Method (1998)
Hyper-V	Hypervisor verified using VCC and Z3 (2005)
CompCert	C compiler verified using Coq (2009)
seL4	micro-kernel verified using Isabelle/HOL (2009)

Work in progress:

Project Everest	Verified HTTPS stack using F*
CakeML	Bootstrapping, verified compiler for ML

The state of the art

Modern systems for program verification

- Why3 (INRIA)
- F* (Microsoft Research)
- Frama-C/WP (INRIA+CEA)

Modern systems for program verification

- Why3 (INRIA)
- F* (Microsoft Research)
- Frama-C/WP (INRIA+CEA)

Common complaint from industry:
"Give us a system that we can actually use"

Modern systems for program verification

- Why3 (INRIA)
- F* (Microsoft Research)
- Frama-C/WP (INRIA+CEA)

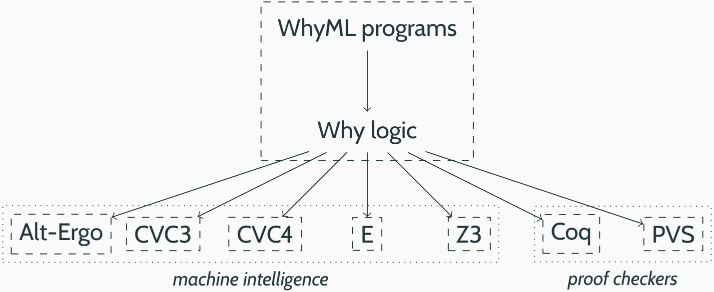
Common complaint from industry:
“Give us a system that we can actually use”

Let’s see how far we get with Why3

- Recently had major updates
- Expertise present at Radboud

The Why3 Platform

specifications + programs ↦ verification conditions


WhyML programs

Why logic

Alt-Ergo

CVC3

CVC4

E

Z3

Coq

PVS

machine intelligence

proof checkers

Layers of Why3

WhyML consists of two parts:

1. A pure logic system
    - Usage: theorem proving.
2. A programming language
    - Usage: modelling programs, intermediate language

Three ways of using Why3

❶ A pure logic system

- First order logic + pure functions (no side effects)
- Proofs discharged by *automatic provers*
- Why3 keeps track of dependencies between proofs
- Ability to produce (potential) *counter-examples*

Why3 logic

```
module Example

use int.Int

predicate odd (x: int) = exists k: int. x = 2*k+1

function sqr (x: int): int = x*x

lemma odd_square:
  forall x: int. odd x -> odd (sqr x)

end
```

WhyML

❷ WhyML as a programming language

- Imperative programming (`while` loops, mutable data)
- Function contracts: pre- and postconditions
- Algebraic data types with pattern matching
- Type inference (like Haskell, ML)
- Control-flow: `break`, `continue`, `return`
- Why3 generates verification conditions

WhyML programs

```
let foo (x: int): int
  requires { x >= 0 }
  ensures { result >= 0 }
= let z = ref 0 in
  let odd = ref 1 in
  let sum = ref 1 in
  while !sum <= x do
    z := !z + 1;
    odd := !odd + 2;
    sum := !sum + !odd;
  done;
  return !z
```

Theoretical background

Hoare logic

Pre- and postconditions: $\{P\}$ **S** $\{Q\}$

- *Partial correctness*: if P holds, and we run **S**, then Q holds when it terminates.
- **Note**: Doesn't say **S** will actually terminate!

Comes with derivation rules:

```
{ x = 42 }
{ x+1 = 43 }
y := x + 1
{ y = 43 }
```

$$\{P[x \mapsto a]\}\mathbf{x{:=}a}\{P\}$$

$$\frac{\{P\}\mathbf{S}_1\{Q\} \quad \{Q\}\mathbf{S}_2\{R\}}{\{P\}\mathbf{S}_1;\mathbf{S}_2\{R\}}$$

Do you enjoy Hoare logic?

```
{ x >= 0 }
while x > 0 do
  x := x-1
done
{ x = 0 }
```

Do you enjoy Hoare logic?

```
{ x >= 0 }

while x > 0 do


  x := x-1


done

{ x = 0 }
```

Do you enjoy Hoare logic?

```
{ x >= 0 }
{ INV }
while x > 0 do
  { x > 0 /\ INV }

  x := x-1

  { INV }
done
{ not (x > 0) /\ INV }
{ x = 0 }
```

Do you enjoy Hoare logic?

```
{ x >= 0 }
{ x >= 0 }
while x > 0 do
  { x > 0 /\ x >= 0 }

  x := x-1

  { x >= 0 }
done
{ not (x > 0) /\ x >= 0 }
{ x = 0 }
```

Do you enjoy Hoare logic?

```
{ x >= 0 }
{ x >= 0 }
while x > 0 do
  { x > 0 /\ x >= 0 }
  { x-1 >= 0 }
  x := x-1
  { x >= 0 }
  { x >= 0 }
done
{ not (x > 0) /\ x >= 0 }
{ x = 0 }
```

Some observations

Hoare logic proofs are mostly mechanical, except:

- Finding loop invariants
- Proving that one condition follows from another

And can only show *partial correctness*!

Weakest liberal precondition

A predicate $wlp(S, Q)$, so that $\{wlp(S, Q)\}$ **S** $\{Q\}$

- Instead of deriving $\{P\}$ **S** $\{Q\}$, just show $P \rightarrow wlp(S, Q)$

Weakest liberal precondition

A predicate $wlp(S, Q)$, so that $\{wlp(S, Q)\}$ **S** $\{Q\}$

- Instead of deriving $\{P\}$ **S** $\{Q\}$, just show $P \rightarrow wlp(S, Q)$

$$\begin{aligned}
wlp(\texttt{x:=e}, Q) &= Q[x \mapsto e] \\
wlp(\texttt{e1;e2}, Q) &= wlp(e1, wlp(e2, Q)) \\
wlp(\texttt{if b then e1 else e2}, Q) &= (b \rightarrow wlp(e1, Q)) \land (\neg b \rightarrow wlp(e2, Q)) \\
wlp(\texttt{while b do S}, Q) &= INV \land \\
&\quad \forall v \in S.INV \rightarrow (b \rightarrow wlp(S, INV)) \land (\neg b \rightarrow Q)
\end{aligned}$$

wlp: *Computation*

$$
\begin{aligned}
wlp(\textbf{while...done}, x = 0) = {} & INV \wedge \\
& \forall x. INV \rightarrow (x > 0 \rightarrow wlp(\textbf{x:=x-1}, INV)) \wedge \\
& (\neg(x > 0) \rightarrow x = 0)
\end{aligned}
$$

```
{ x >= 0 }
while x > 0 do invariant x >= 0
  x := x-1
done
{ x = 0 }
```

wlp: *Computation*

$$\begin{aligned} wlp(\textbf{while}\ldots\textbf{done}, x = 0) = x \geq 0 \land \\ \forall x. x \geq 0 \rightarrow (x > 0 \rightarrow wlp(\textbf{x:=x-1}, x \geq 0)) \land \\ (\neg(x > 0) \rightarrow x = 0) \end{aligned}$$

```
{ x >= 0 }
while x > 0 do invariant x >= 0
  x := x-1
done
{ x = 0 }
```

wlp: Computation

$$
\begin{aligned}
wlp(\textbf{while\ldots done}, x = 0) = x \geq 0 \wedge \\
\forall x. x \geq 0 \to (x > 0 \to x - 1 \geq 0) \wedge \\
(\neg(x > 0) \to x = 0)
\end{aligned}
$$

```
{ x >= 0 }
while x > 0 do invariant x >= 0
  x := x-1
done
{ x = 0 }
```

wlp: *Computation*

This is the verification condition:

$$x \geq 0 \longrightarrow wlp(\textbf{while} \ldots \textbf{done}, x = 0)$$

wlp: Computation

This is the verification condition:

$$\begin{aligned} & x \geq 0 \longrightarrow x \geq 0 \wedge \\ & \quad \forall x. x \geq 0 \rightarrow (x > 0 \rightarrow x - 1 \geq 0) \wedge (\neg(x > 0) \rightarrow x = 0) \end{aligned}$$

wlp: *Computation*

This is the verification condition:

$$
\begin{aligned}
& x \geq 0 \longrightarrow x \geq 0 \wedge \\
& \quad \forall x . x \geq 0 \rightarrow (x > 0 \rightarrow x - 1 \geq 0) \wedge (\neg(x > 0) \rightarrow x = 0)
\end{aligned}
$$

This is what Why3 will do for you:

- Why3 computes (more or less) exactly this.
- Why3 will also do this proof for you.

What do we want to prove?

Partial correctness $\{P\}$ **S** $\{Q\}$

Termination Prove that **S** terminates.

Total correctness Partial correctness + termination

What do we want to prove?

Partial correctness $\{P\}$ **S** $\{Q\}$

Termination Prove that **S** terminates.

Total correctness Partial correctness + termination

Partial correctness on its own can be weak.

```
while not sorted a do
  tmp := a[0]; a[0] := a[1]; a[1] := tmp;
done
{ sorted a }
```

Proving termination

To prove termination of `while` loops, find some quantity that:

- Gets smaller every iteration
- Never becomes negative

Proving termination

To prove termination of `while` loops, find some quantity that:

- Gets smaller every iteration
- Never becomes negative

We call this the **variant**.

```
{ x >= 0 }
while x > 0 do
  invariant x >= 0
  variant x
  x := x-1
done
{ x = 0 }
```

Practical matters: the Why3 toolbox

Programming in WhyML

If you can program, you can program in WhyML!

- Programs can be run directly (`why3 execute`)

Built-in types: `bool`, `int`, `real`

- Data is **immutable** by default
- Mutable data can be stored in *references*: `ref int`

WhyML has annotations for:

- Function contracts: `requires`, `ensures`
- While loops: `invariant`, `variant`
- Assertions: `assert`

WhyML programs

```
let foo (x: int): int
  requires { ... }
  ensures { ... }
= let z:   ref int = ref 0 in
  let odd: ref int = ref 1 in
  let sum: ref int = ref 1 in
  while !sum <= x do
    invariant { ... }
    variant { ... }
    z := !z + 1;
    odd := !odd + 2;
    assert { ... };
    sum := !sum + !odd;
  done;
  return !z
```

Logical language

Pure WhyML expressions + first order logic

Quick syntax guide:

$x \wedge y$	`x/\y`
$x \vee y$	`x\/y`
$\neg y$	`not x`
$x \rightarrow y$	`x->y`
$x \leftrightarrow y$	`x<->y`
$\forall x \in T[P(x)]$	`forall x:t.  p x`
$\exists x \in T[P(x)]$	`exists x:t.  p x`

Logic vs. Code

Pure logic:

```
function double (x: int): int =
  2*x


predicate divides (d n: int) =
  exists q: int. n = q*d
```

Program code:

```
let double (x: int): int =
  2*x


let divides (d n: int): bool =
  d = 0 && n = 0 || mod n d = 0
```

Logic vs. Code

❶ Logical expressions can only be used in annotations

Pure logic:

```
function double (x: int): int =
  2*x


predicate divides (d n: int) =
  exists q: int. n = q*d
```

Program code:

```
let double (x: int): int =
  2*x


let divides (d n: int): bool =
  d = 0 && n = 0 || mod n d = 0
```

Logic vs. Code

❶ Logical expressions can only be used in annotations

❷ To reason about programs, you generally need contracts

Pure logic:

```
function double (x: int): int =
  2*x


predicate divides (d n: int) =
  exists q: int. n = q*d
```

Program code:

```
let double (x: int): int
  ensures { result = 2*x }
= 2*x

let divides (d n: int): bool
  ensures { result <->
            exists q: int. n = q*d }
= d = 0 && n = 0 || mod n d = 0
```

Logic vs. Code

Often we can avoid repeating ourselves:

Usable in both logical formulas and programs:

```
let function double (x: int): int =
  2*x

let predicate divides (d n: int)
  ensures { result <-> exists q: int. n = q*d }
= d = 0 && n = 0 || mod n d = 0
```

Why3 IDE

Proving programs is done in the Why3 IDE (`why3 ide`)

- Has a *program view* and a *logical view*
- Allows editing the program
- Access provers with right click
- Logical formulas can be manipulated
  - Strategies: automated splitting & proving
  - Transformations: fine-grained control
- State can be saved and returned to later

Let’s do something!

Example

What does this compute?

```
let foo (x: int): int
= let z   = ref 0 in
  let odd = ref 1 in
  let sum = ref 1 in
  while !sum <= x do
    z := !z + 1;
    odd := !odd + 2;
    sum := !sum + !odd;
  done;
  return !z
```

Example

What does this compute?

```
let foo (x: int): int
= let z   = ref 0 in
  let odd = ref 1 in
  let sum = ref 1 in
  while !sum <= x do
    z := !z + 1;
    odd := !odd + 2;
    sum := !sum + !odd;
  done;
  return !z
```

!z	!odd	!sum

Example

What does this compute?

```
let foo (x: int): int
= let z   = ref 0 in
  let odd = ref 1 in
  let sum = ref 1 in
  while !sum <= x do
    z := !z + 1;
    odd := !odd + 2;
    sum := !sum + !odd;
  done;
  return !z
```

!z	!odd	!sum
0	1	1

Example

What does this compute?

```
let foo (x: int): int
= let z   = ref 0 in
  let odd = ref 1 in
  let sum = ref 1 in
  while !sum <= x do
    z := !z + 1;
    odd := !odd + 2;
    sum := !sum + !odd;
  done;
  return !z
```

!z	!odd	!sum
0	1	1
1	3	4

Example

What does this compute?

```
let foo (x: int): int
= let z   = ref 0 in
  let odd = ref 1 in
  let sum = ref 1 in
  while !sum <= x do
    z := !z + 1;
    odd := !odd + 2;
    sum := !sum + !odd;
  done;
  return !z
```

!z	!odd	!sum
0	1	1
1	3	4
2	5	9

Example

What does this compute?

```
let foo (x: int): int
= let z   = ref 0 in
  let odd = ref 1 in
  let sum = ref 1 in
  while !sum <= x do
    z := !z + 1;
    odd := !odd + 2;
    sum := !sum + !odd;
  done;
  return !z
```

!z	!odd	!sum
0	1	1
1	3	4
2	5	9
3	7	16

Example

What does this compute?

```
let foo (x: int): int
= let z   = ref 0 in
  let odd = ref 1 in
  let sum = ref 1 in
  while !sum <= x do
    z := !z + 1;
    odd := !odd + 2;
    sum := !sum + !odd;
  done;
  return !z
```

!z	!odd	!sum
0	1	1
1	3	4
2	5	9
3	7	16
4	9	25

Example

What does this compute?

```
let foo (x: int): int
= let z   = ref 0 in
  let odd = ref 1 in
  let sum = ref 1 in
  while !sum <= x do
    z := !z + 1;
    odd := !odd + 2;
    sum := !sum + !odd;
  done;
  return !z
```

!z	!odd	!sum
0	1	1
1	3	4
2	5	9
3	7	16
4	9	25
5	11	36

Example

What does this compute?

```
let foo (x: int): int
= let z   = ref 0 in
  let odd = ref 1 in
  let sum = ref 1 in
  while !sum <= x do
    z := !z + 1;
    odd := !odd + 2;
    sum := !sum + !odd;
  done;
  return !z
```

!z	!odd	!sum
0	1	1
1	3	4
2	5	9
3	7	16
4	9	25
5	11	36
6	13	49

Example

What does this compute?

```
let foo (x: int): int
= let z   = ref 0 in
  let odd = ref 1 in
  let sum = ref 1 in
  while !sum <= x do
    z := !z + 1;
    odd := !odd + 2;
    sum := !sum + !odd;
  done;
  return !z
```

!z	!odd	!sum
0	1	1
1	3	4
2	5	9
3	7	16
4	9	25
5	11	36
6	13	49
7	15	64

Demo!

How to find good invariants?

[redacted]

Find a property that generalizes initial and end condition.

```
assert { !i = 0 };
let j = ref 9 in
while !i < 10 do
  invariant { ... }
  i := !i + 1;
  j := !j - 1;
done
assert { !i = 9 };
```

How to find good invariants?

Find a property that generalizes initial and end condition.

```
assert { !i = 0 };
let j = ref 9 in
while !i < 10 do
  invariant { !i + !j = 9 }
  i := !i + 1;
  j := !j - 1;
done
assert { !i = 9 };
```

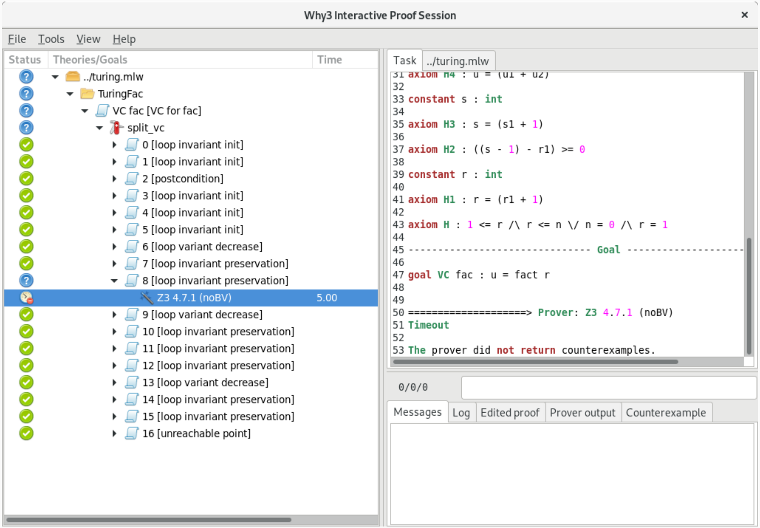
Conclusion

Summary

- Why3 is a platform for automating total correctness proofs
- Uses powerful SMT solvers to do tedious proofs
- WhyML: Logical formulas + Program code
- Functions: specify *contracts*
- `while` loops: specify *invariants* and *variants*

Final comment

Step 1: Frustration

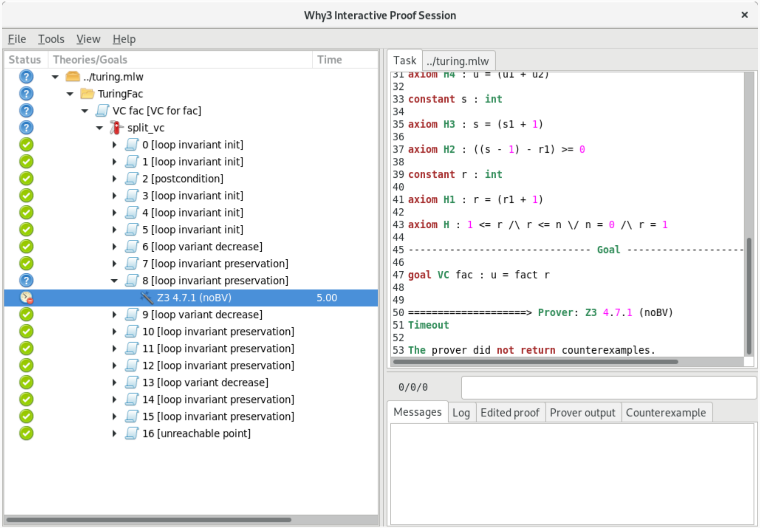


Status	Theories/Goals	Time
?	../turing.mlw	
?	TuringFac	
?	VC fac [VC for fac]	
?	split_vc	
✓	0 [loop invariant init]	
✓	1 [loop invariant init]	
✓	2 [postcondition]	
✓	3 [loop invariant init]	
✓	4 [loop invariant init]	
✓	5 [loop invariant init]	
✓	6 [loop variant decrease]	
✓	7 [loop invariant preservation]	
?	8 [loop invariant preservation]	
	Z3 4.7.1 (noBV)	5.00
✓	9 [loop variant decrease]	
✓	10 [loop invariant preservation]	
✓	11 [loop invariant preservation]	
✓	12 [loop invariant preservation]	
✓	13 [loop variant decrease]	
✓	14 [loop invariant preservation]	
✓	15 [loop invariant preservation]	
✓	16 [unreachable point]	

```
axiom H4 : u = (u1 + u2)

constant s : int

axiom H3 : s = (s1 + 1)

axiom H2 : ((s - 1) - r1) >= 0

constant r : int

axiom H1 : r = (r1 + 1)

axiom H : 1 <= r /\ r <= n \/ n = 0 /\ r = 1

------------------------------- Goal -------------------

goal VC fac : u = fact r


===================> Prover: Z3 4.7.1 (noBV)
Timeout

The prover did not return counterexamples.
```

Final comment

Step 2: ...



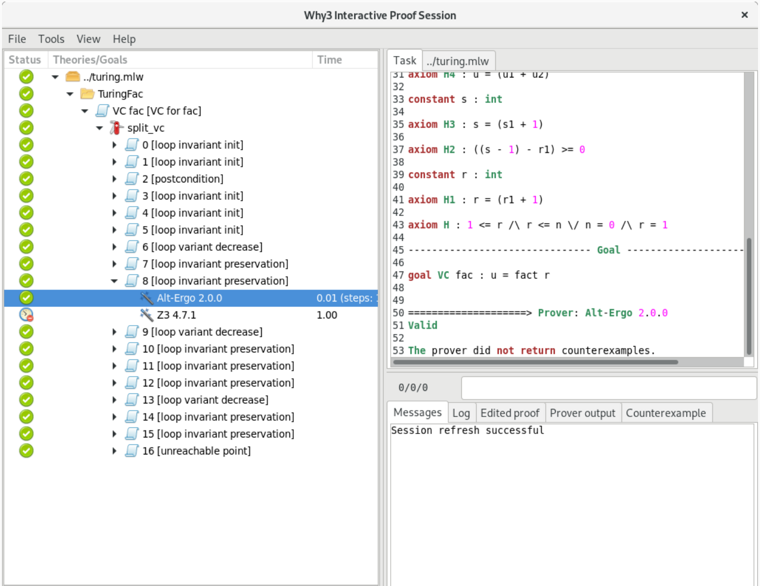
Status	Theories/Goals	Time
?	../turing.mlw	
?	TuringFac	
?	VC fac [VC for fac]	
?	split_vc	
✓	0 [loop invariant init]	
✓	1 [loop invariant init]	
✓	2 [postcondition]	
✓	3 [loop invariant init]	
✓	4 [loop invariant init]	
✓	5 [loop invariant init]	
✓	6 [loop variant decrease]	
✓	7 [loop invariant preservation]	
?	8 [loop invariant preservation]	
⏱	Z3 4.7.1 (noBV)	5.00
✓	9 [loop variant decrease]	
✓	10 [loop invariant preservation]	
✓	11 [loop invariant preservation]	
✓	12 [loop invariant preservation]	
✓	13 [loop variant decrease]	
✓	14 [loop invariant preservation]	
✓	15 [loop invariant preservation]	
✓	16 [unreachable point]	

```
axiom H4 : u = (u1 + u2)

constant s : int

axiom H3 : s = (s1 + 1)

axiom H2 : ((s - 1) - r1) >= 0

constant r : int

axiom H1 : r = (r1 + 1)

axiom H : 1 <= r /\ r <= n \/ n = 0 /\ r = 1

------------------------------- Goal -------------------

goal VC fac : u = fact r


====================> Prover: Z3 4.7.1 (noBV)
Timeout

The prover did not return counterexamples.
```

```
0/0/0
```

Final comment

Step 3: Dopamine rush!



Status	Theories/Goals	Time
✓	../turing.mlw	
✓	TuringFac	
✓	VC fac [VC for fac]	
✓	split_vc	
✓	0 [loop invariant init]	
✓	1 [loop invariant init]	
✓	2 [postcondition]	
✓	3 [loop invariant init]	
✓	4 [loop invariant init]	
✓	5 [loop invariant init]	
✓	6 [loop variant decrease]	
✓	7 [loop invariant preservation]	
✓	8 [loop invariant preservation]	
✓	Alt-Ergo 2.0.0	0.01 (steps:
⏲	Z3 4.7.1	1.00
✓	9 [loop variant decrease]	
✓	10 [loop invariant preservation]	
✓	11 [loop invariant preservation]	
✓	12 [loop invariant preservation]	
✓	13 [loop variant decrease]	
✓	14 [loop invariant preservation]	
✓	15 [loop invariant preservation]	
✓	16 [unreachable point]	

```
axiom H4 : u = (u1 + u2)

constant s : int

axiom H3 : s = (s1 + 1)

axiom H2 : ((s - 1) - r1) >= 0

constant r : int

axiom H1 : r = (r1 + 1)

axiom H : 1 <= r /\ r <= n \/ n = 0 /\ r = 1

------------------------------- Goal -------------------

goal VC fac : u = fact r


===================> Prover: Alt-Ergo 2.0.0
Valid

The prover did not return counterexamples.
```

```
0/0/0
```

```
Session refresh successful
```