

An Introduction to Why3

Ming-Hsien Tsai

2020/11/25

Why3

- Why3 (<http://why3.lri.fr>) is a platform for deductive program verification
- Why3 is developed in the team-project Toccata (formerly ProVal) at Inria Saclay-Île-de-France / LRI Univ Paris-Saclay / CNRS
- Why3 is a full redesign of Why
- A simple online version is available at <http://why3.lri.fr/try/>

Why3

- Inside Why3 -

- Why3 provides
 - WhyML: a rich language for specification and programming
 - a verification condition generator (to various provers)
 - a standard library of logical theories (integer and real arithmetic, Boolean operations, sets, maps, etc.)
 - basic programming data structures (arrays, queues, hash tables, etc.)
 - an automated extraction mechanism (to OCaml programs)

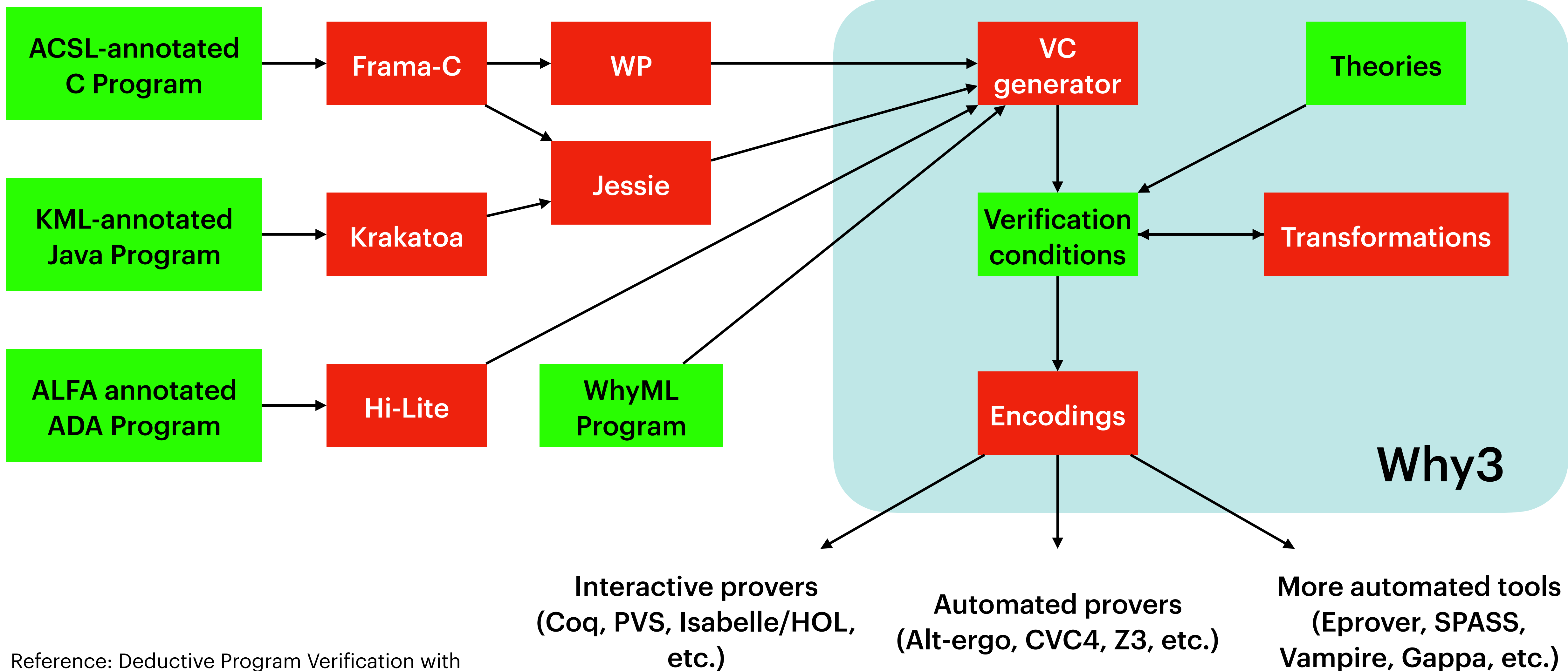
Why3

- Usage -

- Logic only
 - Axiomatize a ring structure and prove lemmas
- Implementation and verification of algorithms and data structures
 - Implement a sorting algorithm and verify its correctness
- As an intermediate verification language
 - Convert a program in another programming language to a WhyML program
- As a library

Why3

- Data Flow -



Why3

- Extensions -

- Why3 is extensible via three kinds of plugins
 - Parsers (new input formats)
 - Transformations (to be used in drivers)
 - Printers (to add support for new provers)

Related Tools

- Boogie 2 (developed by Microsoft)
 - Spec#
 - Z3
- What4 (developed by Galois)
 - Haskell

Installation and Configuration

- Install via opam
 - `$ opam install why3 why3-coq why3-ide`
- Install via apt-get
 - `$ apt-get install why3 why3-coq`
- Configuration
 - `$ why3 config --detect`
- List detected provers
 - `$ why3 --list-provers`

Known provers:

Alt-Ergo 2.3.3

CVC3 2.4.1

CVC4 1.6

CVC4 1.6 (counterexamples)

Coq 8.9.1

Gappa 1.3.3

MathSAT5 5.5.2

Z3 4.8.8 (counterexamples)

Z3 4.8.9 (counterexamples)

Hello Proofs

- hello_proof.mlw -

module and theory
are interchangeable

uppercase initial is required

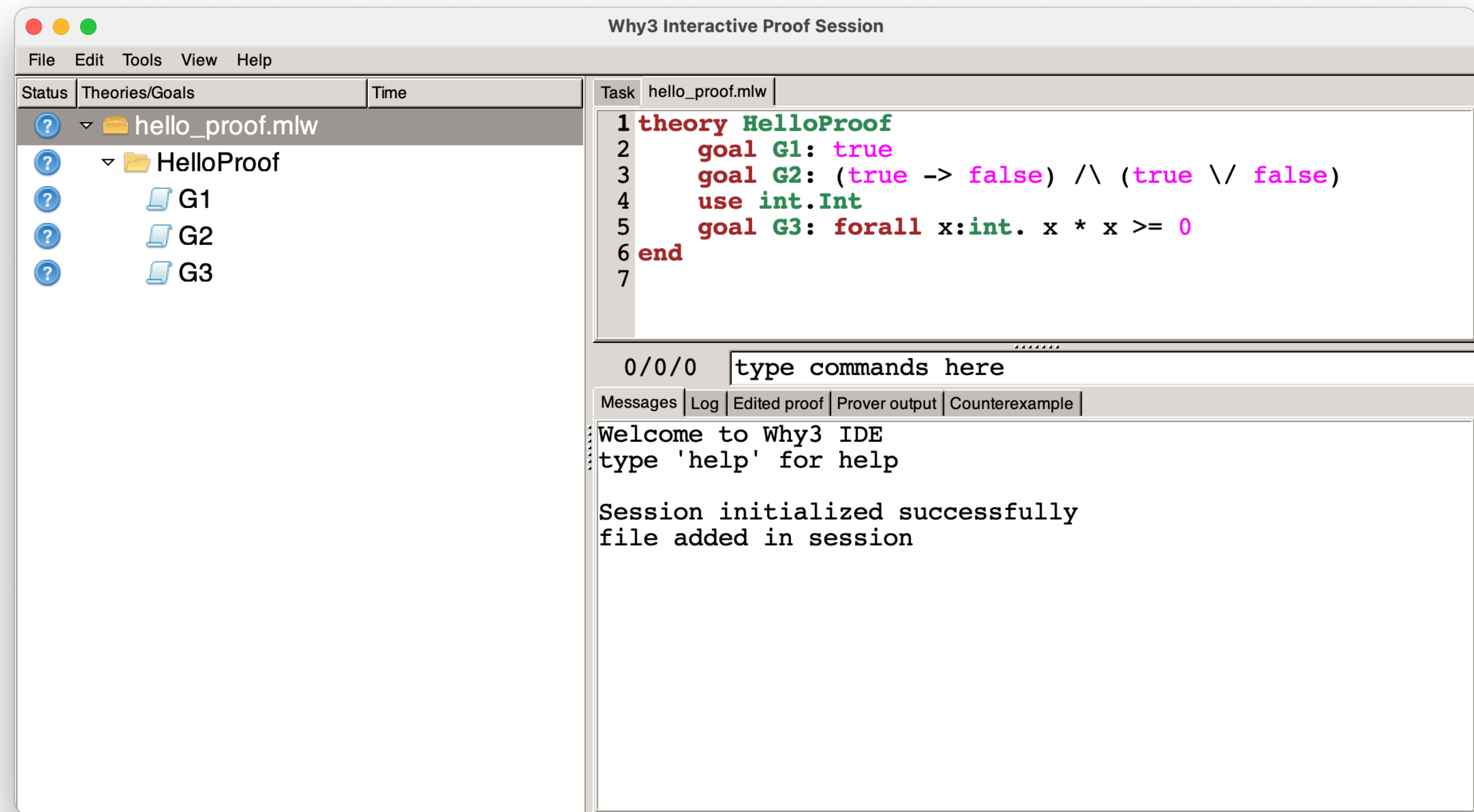
```
theory HelloProof  
  goal G1: true  
  goal G2: (true -> false) /\ (true \/ false)  
  use int.Int  
  goal G3: forall x:int. x * x >= 0  
end
```

hello_proof.mlw

Hello Proofs

- Why3 IDE -

- Launch the why3 IDE
 - `$ why3 ide hello_proof.mlw`



```
theory HelloProof

  goal G1: true

  goal G2: (true -> false) /\ (true \/ false)

  use int.Int

  goal G3: forall x:int. x * x >= 0

end
```

hello_proof.mlw



Hello Proofs

- Tasks -

The screenshot shows the Why3 Interactive Proof Session IDE. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help".

The interface is divided into several panels:

- Left Panel (Theories/Goals):** A tree view showing the project structure. It includes a folder "hello_proof.mlw" containing a sub-folder "HelloProof" with three goals: "G1", "G2", and "G3". "G3" is currently selected and highlighted.
- Task Panel (hello_proof.mlw):** A code editor showing the goal definition for G3:

```
1 ----- Local Context -----  
2  
3 ----- Goal -----  
4  
5 goal G3 : forall x:int. (x * x) >= 0  
6  
7
```
- Command Line:** A text input field with the prompt "0/0/0 type commands here".
- Messages Panel:** A log area with tabs for "Messages", "Log", "Edited proof", "Prover output", and "Counterexample". The current message is:

```
Welcome to Why3 IDE  
type 'help' for help  
  
Session initialized successfully  
file added in session
```

Hello Proofs

- Calling Provers in IDE -

The screenshot shows the Why3 Interactive Proof Session IDE. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help".

The interface is divided into several panes:

- Task Pane:** Shows the current task as "hello_proof.mlw".
- Code Editor:** Contains the following proof script:

```
1 theory HelloProof
2   goal G1: true
   goal G2: (true -> false) /\ (true \/ false)
   use int.Int
   goal G3: forall x:int. x * x >= 0
end
```
- Goals Pane:** Lists the goals G1, G2, and G3, each with a question mark icon.
- Prover Selection Menu:** A dropdown menu is open, showing the following options:
 - Alt-Ergo 2.3.3
 - Coq 8.9.1
 - CVC3 2.4.1
 - CVC4 1.6
 - Gappa 1.3.3
 - MathSAT5 5.5.2
 - Auto level 0
 - Auto level 1
 - Auto level 2
 - Auto level 3
 - Split VC
 - Edit
 - Get Counterexamples
 - Replay valid obsolete proofs
 - Replay all obsolete proofs
 - Clean node
 - Remove node
 - Interrupt
- Command Line:** A text input field with the placeholder "type commands here".
- Output Pane:** Shows the following output:

```
0/0
Welcome to Why3 IDE
Get 'help' for help
Reasoning initialized successfully
Goal added in session
```

Hello Proofs

- Calling Provers in IDE -

The screenshot shows the Why3 Interactive Proof Session IDE. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help".

The interface is divided into several panels:

- Left Panel (Theories/Goals):** A tree view showing the project structure. The root is "hello_proof.mlw", which contains a sub-theory "HelloProof". Under "HelloProof", there are three goals: "G1" (checked), "G2" (question mark), and "G3" (checked). A prover configuration "Alt-Ergo 2.3.3" with a time limit of "0.01" is associated with goal G2.
- Task Panel (hello_proof.mlw):** A code editor showing the following code:

```
1 theory HelloProof
2   goal G1: true
3   goal G2: (true -> false) /\ (true \/ false)
4   use int.Int
5   goal G3: forall x:int. x * x >= 0
6 end
7
```
- Progress Panel:** Shows "0/0/0" and a progress bar.
- Messages Panel:** Includes tabs for "Messages", "Log", "Edited proof", "Prover output", and "Counterexample". The "Messages" tab is currently selected and is empty.

Hello Proofs

- Applying Transformations -

The screenshot shows the Why3 Interactive Proof Session interface. The main window displays the following code:

```
1 theory HelloProof
2   goal G1: true
3   goal G2: (true -> false) /\ (true \/ false)
4   use int.Int
5   goal G3: forall x:int. x * x >= 0
6 end
```

The left sidebar shows the project structure:

- hello_proof.mlw
 - HelloProof
 - G1
 - G2
 - Alt-Ergo 2.3.3
 - G3

A context menu is open over goal G3, listing various solvers and actions:

Alt-Ergo 2.3.3	VoidSymbol
Coq 8.9.1	VoidSymbol
CVC3 2.4.1	VoidSymbol
CVC4 1.6	VoidSymbol
Gappa 1.3.3	VoidSymbol
MathSAT5 5.5.2	VoidSymbol
<hr/>	
Auto level 0	0
Auto level 1	1
Auto level 2	2
Auto level 3	3
Split VC	S
<hr/>	
Edit	E
Get Counterexamples	G
Replay valid obsolete proofs	R
Replay all obsolete proofs	
Clean node	C
Remove node	⊗
Interrupt	

The bottom of the interface shows tabs for "Prover output" and "Counterexample".

Hello Proofs

- Applying Transformations -

The screenshot displays the Why3 Interactive Proof Session interface. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help".

The left sidebar shows a tree view of the project structure:

- hello_proof.mlw
 - HelloProof
 - G1 (checked)
 - G2
 - Alt-Ergo 2.3.3 (0.01)
 - split_vc
 - 0
 - 1
 - G3 (checked)

```
Task hello_proof.mlw
1 theory HelloProof
2   goal G1: true
3   goal G2: (true -> false) /\ (true \/ false)
4   use int.Int
5   goal G3: forall x:int. x * x >= 0
6 end
7
```

0/0/0

Messages | Log | Edited proof | Prover output | Counterexample

Hello Proofs

- Applying Transformations -

The screenshot displays the Why3 Interactive Proof Session interface. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help".

The left pane shows a tree view of the proof session:

- hello_proof.mlw
 - HelloProof
 - G1 (checked)
 - G2
 - Alt-Ergo 2.3.3 (Time: 0.01)
 - split_vc
 - 0
 - 1 (highlighted)
 - G3 (checked)

The right pane shows the proof script for "hello_proof.mlw":

```
1 theory HelloProof
2   goal G1: true
3   goal G2: (true -> false) /\ (true \/ false)
4   use int.Int
5   goal G3: forall x:int. x * x >= 0
6 end
7
```

Below the script, the progress indicator shows "0/0/0". The bottom pane has tabs for "Messages", "Log", "Edited proof", "Prover output", and "Counterexample".

Hello Proofs

- Applying Transformations -

The screenshot shows the Why3 Interactive Proof Session interface. The main window displays the following code:

```

1 theory HelloProof
2   goal G1: true
3   goal G2: (true -> false) /\ (true \/ false)
4   use int.Int
5   goal G3: forall x:int. x * x >= 0
6 end
7

```

The left sidebar shows a tree view of the project structure:

- hello_proof.mlw
 - HelloProof
 - G1
 - G2
 - Alt-Ergo 2.3.3 0.01
 - split_vc
 - 0
 - 1
 - G3

A context menu is open over goal G3, listing various solvers and actions:

Alt-Ergo 2.3.3	VoidSymbol
Coq 8.9.1	VoidSymbol
CVC3 2.4.1	VoidSymbol
CVC4 1.6	VoidSymbol
Gappa 1.3.3	VoidSymbol
MathSAT5 5.5.2	VoidSymbol
<hr/>	
Auto level 0	0
Auto level 1	1
Auto level 2	2
Auto level 3	3
Split VC	S
<hr/>	
Edit	E
Get Counterexamples	G
Replay valid obsolete proofs	R
Replay all obsolete proofs	
Clean node	C
Remove node	⊗
Interrupt	

The bottom of the interface shows tabs for 'Edited proof', 'Prover output', and 'Counterexample'.

Hello Proofs

- Applying Transformations -

The screenshot displays the Why3 Interactive Proof Session interface. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help".

The interface is divided into several panels:

- Left Panel (Theories/Goals):** A tree view showing the project structure. The root is "hello_proof.mlw", which contains a sub-project "HelloProof". Under "HelloProof", there are three goals: "G1" (checked), "G2" (unchecked), and "split_vc" (unchecked). "split_vc" is expanded to show a sub-goal "0" (unchecked) and a goal "G3" (checked). The "Alt-Ergo 2.3.3" solver is associated with "G2" and "0", with a time of 0.01.
- Task Panel (hello_proof.mlw):** A code editor showing the proof script:

```
1 theory HelloProof
2   goal G1: true
3   goal G2: (true -> false) /\ (true \/ false)
4   use int.Int
5   goal G3: forall x:int. x * x >= 0
6 end
7
```
- Progress Panel:** Shows "0/0/0" and a progress bar.
- Messages Panel:** Includes tabs for "Log", "Edited proof", "Prover output", and "Counterexample".

Hello Proofs

- Modifying the Input -

The screenshot displays the Why3 Interactive Proof Session interface. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help".

The left pane shows a tree view of the proof session:

- hello_proof.mlw
 - HelloProof
 - G1 (checked)
 - G2
 - split_vc
 - 0
 - Alt-Ergo 2.3.3 (0.01)
 - 1 (checked)
 - Alt-Ergo 2.3.3 (0.01)
 - G3 (checked)

The right pane shows the code editor for "hello_proof.mlw":1 theory HelloProof
2 goal G1: true
3 goal G2: (true -> false) /\ (true \/ false)
4 use int.Int
5 goal G3: forall x:int. x * x >= 0
6 end
7

Below the code editor, there is a progress indicator "0/0/0" and a "Messages" section with tabs for "Log", "Edited proof", "Prover output", and "Counterexample".

Hello Proofs

- Modifying the Input -

The screenshot shows the Why3 Interactive Proof Session interface. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help". The "File" menu is open, showing options: "Add file to session", "Preferences", "Save session", "Save files", "Save session and files" (with shortcut ^S), "Save all and Refresh session" (with shortcut ^R), and "Quit" (with shortcut ^Q). The main area is divided into three panes. The left pane shows a tree view of the proof session with nodes: "G2", "split_vc", "0", "Alt-Ergo 2.3.3" (0.01), "1", "Alt-Ergo 2.3.3" (0.01), and "G3". The middle pane shows the task editor for "*hello_proof.mlw" with the following code:

```
1 theory HelloProof
2   goal G1: true
3   goal G2: (false | -> false) /\ (true \/ false)
4   use int.Int
5   goal G3: forall x:int. x * x >= 0
6 end
7
```

The right pane shows the prover output, currently displaying "0/0/0". Below the prover output is a tabbed interface with tabs for "Messages", "Log", "Edited proof", "Prover output", and "Counterexample".

Hello Proofs

- Modifying the Input -

The screenshot displays the Why3 Interactive Proof Session interface. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help".

The interface is divided into several panels:

- Left Panel (Theories/Goals):** A tree view showing the project structure. The root is "hello_proof.mlw", which contains a folder "HelloProof". Inside "HelloProof", there are goals "G1", "G2", and "G3". "G2" is expanded to show a tactic "split_vc" with sub-goals "0" and "1". A red dashed box highlights the entry "Alt-Ergo 2.3.3 0.01 (obsolete)" associated with goal "G2".
- Right Panel (Task):** Shows the source code for the proof task "hello_proof.mlw". The code is as follows:

```
1 theory HelloProof
2   goal G1: true
3   goal G2: (false -> false) /\ (true \/ false)
4   use int.Int
5   goal G3: forall x:int. x * x >= 0
6 end
7
```
- Bottom Panel (Messages):** Displays a message: "/Users/mht208/Work/Test/why3/hello_proof.mlw was saved" and "Session refresh successful".

Hello Proofs

- Replaying Obsolete Proofs -

The screenshot shows the Why3 Interactive Proof Session interface. The title bar reads "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help". The "Tools" menu is open, showing various options. The "Replay all obsolete proofs" option is highlighted. The main window displays a code editor with the following code:

```
1 theory HelloProof
2   goal G1: true
3   goal G2: (false -> false) /\ (true \/ false)
4   use int.Int
5   goal G3: forall x:int. x * x >= 0
6 end
7
```

Below the code editor, there is a progress indicator showing "0/0/0". The "Messages" tab is active, displaying the message "Session refresh successful". The "Log" tab shows "01 (obsolete)".

Hello Proofs

- Replaying Obsolete Proofs -

The screenshot shows the Why3 Interactive Proof Session interface. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help".

The left sidebar displays a project structure:

- hello_proof.mlw (checked)
- └─ HelloProof (checked)

The main editor area shows the code for the `HelloProof` theory:

```
1 theory HelloProof
2   goal G1: true
3   goal G2: (false -> false) /\ (true \/ false)
4   use int.Int
5   goal G3: forall x:int. x * x >= 0
6 end
7
```

Below the code editor, there is a progress indicator showing "0/0/0" and a "Messages" panel with the text "Session refresh successful".

Hello Proofs

- Prove with Coq -

The screenshot shows the Why3 Interactive Proof Session IDE. The main window displays a Coq proof script for a theory named `HelloProof`. The script contains two goals: `G1: true` and `G2: (true -> false) /\ (true \/ false)`. A `forall` statement is also present: `forall x:int. x * x >= 0`. The IDE interface includes a menu bar (File, Edit, Tools, View, Help), a status bar, and a task pane showing the current file `hello_proof.mlw`. A context menu is open over goal `G1`, listing various solvers and proof tactics. The output pane shows the result of the proof: `Why3 IDE`, `for help`, `alized successfully`, and `session`.

Status	Theories/Goals	Time	Task
?	hello_proof.mlw		hello_proof.mlw
?	HelloProof		
?	G1		
?	G2		
?	G3		

```

1 theory HelloProof
2   goal G1: true
3   goal G2: (true -> false) /\ (true \/ false)
4   use int.Int
5   : forall x:int. x * x >= 0

```

Solver	VoidSymbol
Alt-Ergo 2.3.3	VoidSymbol
Coq 8.9.1	VoidSymbol
CVC3 2.4.1	VoidSymbol
CVC4 1.6	VoidSymbol
Gappa 1.3.3	VoidSymbol
MathSAT5 5.5.2	VoidSymbol

Tactic	Time
Auto level 0	0
Auto level 1	1
Auto level 2	2
Auto level 3	3
Split VC	S

Action	Key
Edit	E
Get Counterexamples	G
Replay valid obsolete proofs	R
Replay all obsolete proofs	
Clean node	C
Remove node	⊗
Interrupt	

proof | Prover output | Counterexample |

Why3 IDE
for help
alized successfully
session



Hello Proofs

- Prove with Coq -

The screenshot shows the Why3 Interactive Proof Session interface. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help".

The interface is divided into several panels:

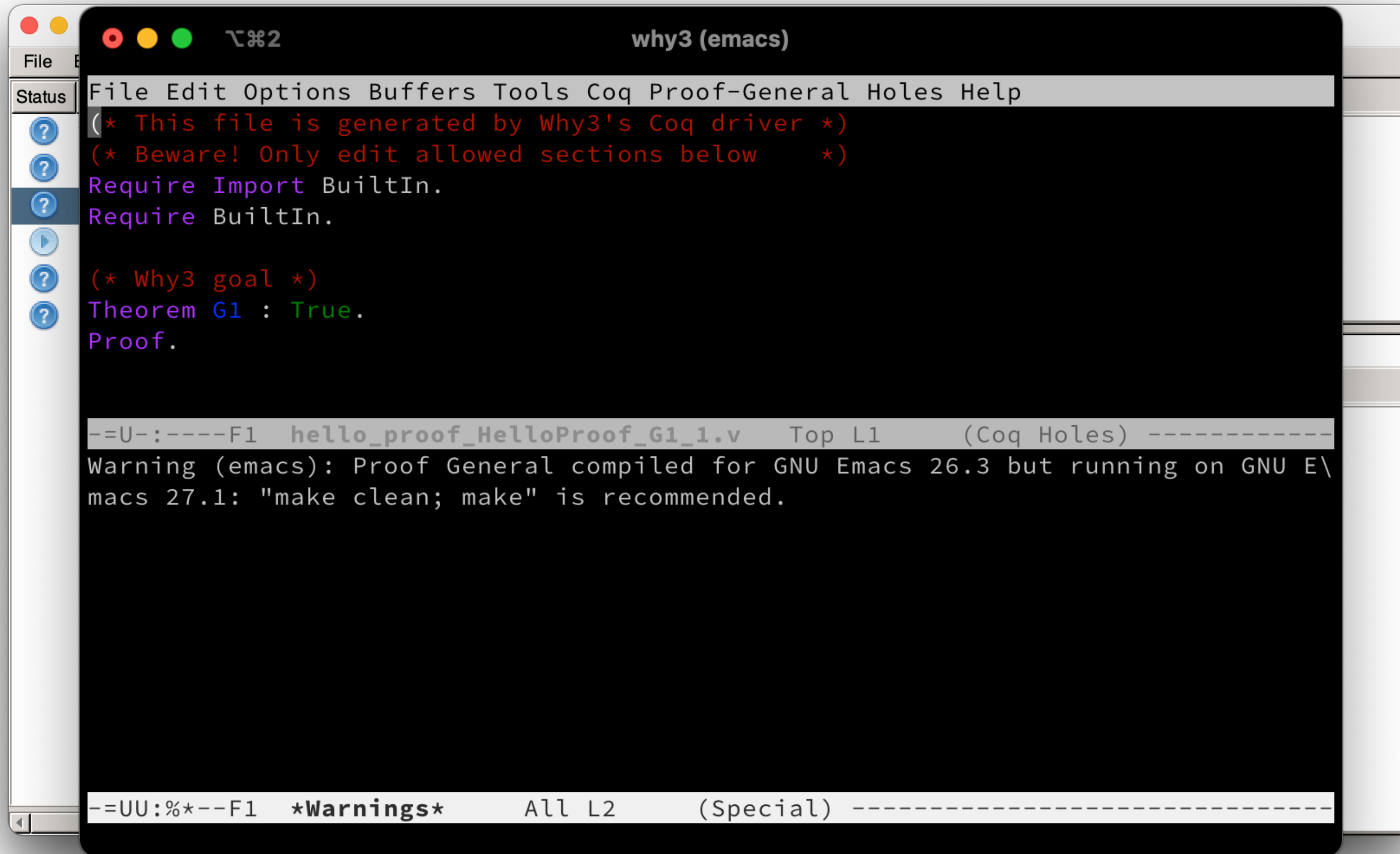
- Left Panel (Theories/Goals):** A tree view showing the project structure. It includes a folder "hello_proof.mlw", a sub-folder "HelloProof", and a goal "G1" which is currently selected. Below "G1", there are three entries: "Coq 8.9.1 (running) [limit=...]", "G2", and "G3".
- Right Panel (Task):** A code editor showing the Coq proof script for "hello_proof.mlw". The script is as follows:

```
1 theory HelloProof
2   goal G1: true
3   goal G2: (true -> false) /\ (true \/ false)
4   use int.Int
5   goal G3: forall x:int. x * x >= 0
6 end
7
```
- Bottom Panel (Messages):** A panel with tabs for "Messages", "Log", "Edited proof", "Prover output", and "Counterexample". The "Messages" tab is currently active and is empty.



Hello Proofs

- Prove with Coq -



```
File Edit Options Buffers Tools Coq Proof-General Holes Help
(* This file is generated by Why3's Coq driver *)
(* Beware! Only edit allowed sections below *)
Require Import BuiltIn.
Require BuiltIn.

(* Why3 goal *)
Theorem G1 : True.
Proof.

--U-:----F1 hello_proof_HelloProof_G1_1.v Top L1 (Coq Holes) -----
Warning (emacs): Proof General compiled for GNU Emacs 26.3 but running on GNU E\
macs 27.1: "make clean; make" is recommended.

--UU:%*--F1 *Warnings* All L2 (Special) -----
```



Hello Proofs

- Prove with Coq -



```
File Edit Options Buffers Tools Coq Proof-General Holes Help
(* This file is generated by Why3's Coq driver *)
(* Beware! Only edit allowed sections below *)
Require Import BuiltIn.
Require BuiltIn.

(* Why3 goal *)
Theorem G1 : True.
Proof.
trivial.

--=U-:----F1 hello_proof>HelloProof_G1_1.v Top L9 (Coq Holes) -----
Warning (emacs): Proof General compiled for GNU Emacs 26.3 but running on GNU E\
macs 27.1: "make clean; make" is recommended.

--=UU:%*--F1 *Warnings* All L2 (Special) -----
Wrote /Users/mht208/Work/Test/why3/hello_proof/hello_proof>HelloProof_G1_1.v
```



Hello Proofs

- Prove with Coq -

The screenshot shows the Why3 Interactive Proof Session interface. The main window displays a Coq proof script for a theory named `HelloProof`. The script contains three lines of code:

```
1 theory HelloProof
2   goal G1: true
3   goal G2: (true -> false) /\ (true \/ false)
forall x:int. x * x >= 0
```

A context menu is open over the goal `G1`. The menu items are:

- Alt-Ergo 2.3.3 VoidSymbol
- Coq 8.9.1 VoidSymbol
- CVC3 2.4.1 VoidSymbol
- CVC4 1.6 VoidSymbol
- Gappa 1.3.3 VoidSymbol
- MathSAT5 5.5.2 VoidSymbol
- Auto level 0 0
- Auto level 1 1
- Auto level 2 2
- Auto level 3 3
- Split VC S
- Edit E
- Get Counterexamples G
- Replay valid obsolete proofs R
- Replay all obsolete proofs**
- Clean node C
- Remove node
- Interrupt

The status bar at the bottom of the window shows the file path: `wrote /Users/mnt208/work/test/why3/hello_proof/hello_proof_HelloProof_G1_1.v`



Hello Proofs

- Prove with Coq -

The screenshot shows the Why3 Interactive Proof Session interface. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help".

The interface is divided into several sections:

- Left Panel (Theories/Goals):** A tree view showing the project structure. It includes a folder "hello_proof.mlw" containing a sub-folder "HelloProof". Under "HelloProof", there are three goals: "G1" (marked with a green checkmark), "G2" (selected with a blue highlight), and "G3".
- Task Panel (hello_proof.mlw):** A code editor showing the Coq proof script:

```
1 theory HelloProof
2   goal G1: true
3   goal G2: (true -> false) /\ (true \/ false)
4   use int.Int
5   goal G3: forall x:int. x * x >= 0
6 end
7
```
- Progress Bar:** A progress indicator showing "0/0/0" and a small progress bar.
- Messages Panel:** A panel with tabs for "Messages", "Log", "Edited proof", "Prover output", and "Counterexample". The "Messages" tab is currently selected and is empty.

At the bottom of the window, a status bar shows the file path: "wrote /Users/mnt208/work/test/why3/hello_proof/hello_proof_HelloProof_G1_1.v".



Hello Proofs

- Displaying VCs via Why3 Commands -

- \$ why3 prove hello_proof.mlw

```
theory HelloProof
  (* use why3.BuiltIn.BuiltIn *)
  (* use why3.Bool.Bool *)
  (* use why3.Unit.Unit *)
  goal G1 : true
  goal G2 : (true -> false) /\ (true \/ false)
  (* use int.Int *)
  goal G3 : forall x:int. (x * x) >= 0
end
```

Hello Proofs

- Proving via Why3 Commands -

- `$ why3 prove -P Alt-Ergo hello_proof.mlw`

```
hello_proof.mlw HelloProof G1: Valid (0.00s, 0 steps)
hello_proof.mlw HelloProof G2: Unknown () (0.01s)
hello_proof.mlw HelloProof G3: Valid (0.00s, 0 steps)
```

- `$ why3 prove -P Alt-Ergo hello_proof.mlw -T HelloProof -G G2 -G G3`

```
hello_proof.mlw HelloProof G2: Unknown () (0.01s)
hello_proof.mlw HelloProof G3: Valid (0.00s, 0 steps)
```

Hello Proofs

- Transforming via Why3 Commands -

- `$ why3 --list-transforms`

```
Known splitting transformations:
```

```
...
```

```
split_goal_right
```

- `$ why3 prove -P Alt-Ergo hello_proof.mlw -a split_vc -T HelloProof -G G2`

```
hello_proof.mlw HelloProof G2: Unknown () (0.01s)
```

```
hello_proof.mlw HelloProof G2: Valid (0.00s, 0 steps)
```


WhyML

- WhyML is a first-order language with polymorphic types, pattern matching, and inductive predicates
- A WhyML source file contains a list of modules
- Each module contains a list of declarations, including
 - Logical declarations (types, functions, predicates, axioms, lemmas, goals)
 - Program data types
 - Program declarations and definitions (assignment, sequence, loops, exceptions, ghost parameters and ghost code, annotations)

WhyML

- Types -

- A type can be
 - an abstract type,
 - an alias type,
 - an algebraic data type,
 - a record type,
 - a range type, or
 - a float type

WhyML

- Abstract Types -

```
type t
```

- Properties about the abstract type can be axiomatized
- The abstract type can be instantiated by theory cloning
 - Related axioms may become provable

WhyML

- Alias Types -

```
type t = list int
```

- A shorthand of a type expression

WhyML

- Algebraic Data Types -

```
type list 'a = Nil | Cons 'a (list 'a)
```

type constructors (with uppercase initials)

- Algebraic data types can be polymorphic
- A data type can be recursive
- A tuple type is a particular case of algebraic data types, with a single constructor

```
type ipair = (int, int)
```

WhyML

- Record Types -

```
type t = { mutable v: int; is_pos: bool }  
  invariant { is_pos = true -> v > 0 }  
  by { v = 3; is_pos = true }
```

- Record types can be polymorphic
- A record type can be recursive
- Existence of at least one record instance is ensured by generating one additional verification condition

Access field v of (x : t) by

- **v x, or**
- **x.v**

```
exists v:int, is_pos:bool. is_pos = true -> v > 0
```

WhyML

- Record Types -

```
type t = { mutable v: int; is_pos: bool }  
  invariant { is_pos = true -> v > 0 }  
  by { v = 3; is_pos = true }
```

ease the verification

- Record types can be polymorphic
- A record type can be recursive
- Existence of at least one record instance is ensured by generating one additional verification condition

Access field v of (x : t) by

- **v x, or**
- **x.v**

```
exists v:int, is_pos:bool. is_pos = true -> v > 0
```

WhyML

- Equality of Record Types -

```
type t = { mutable v: int; is_pos: bool }  
  invariant { is_pos = true -> v > 0 }  
  by { v = 3; is_pos = true }
```

```
goal G: forall n m: t. v n = v m -> is_pos n = is_pos m -> n = m
```



Failed to prove

WhyML

- Equality of Record Types -

What we get from the definition of byte after `why3 prove`

```
goal t'vc : true -> 3 > 0
```

```
type t
```

```
function v t : int
```

```
function is_pos t : bool
```

```
t'invariant :
```

```
forall self:t [is_pos self | v self]. is_pos self = True -> v self > 0
```

WhyML

- Private Record Types -

```
type t = private { mutable v: int; is_pos: bool }  
  
val create (n: int) : t  
  ensures { result.v = n /\ result.is_pos = true -> result.v > 0 }
```

- A record type can be private
 - No instance can be created!
- Private record types are usually used to build interfaces
 - Can be refined with a non-private type

WhyML

- Range Types -

```
type byte = <range 0 15>

val function (+) (n: byte) (m: byte) : byte
  ensures { byte'int result = mod (byte'int n + byte'int m) 16 }

axiom to_int_extensionality:
  forall n, m: byte. byte'int n = byte'int m -> n = m
```

- Must import int.Int
- For every range type r , a function $r'int$ projecting a term of type r to its integer value is generated automatically

Instances are 0:byte, 1:byte, ..., 15:byte

WhyML

- Function and Predicate Symbols -

Every `function` below can be replaced by `predicate`

- Functions can be defined or declared by the following keywords
 - `let`: definition of program function
 - `val`: declaration of a program function
 - `let function`: definition of a pure program function, can be used in specifications
 - `val function`: declaration of a pure program function, can be used in specifications
 - `function`: definition or declaration of a logical function symbol, which can also be used as a program function in ghost code
 - `let lemma`: definition of a special pure program function

pure: side-effect free

WhyML

- Examples of Functions -

- An abstract, logical function:

```
function append (list 'a) (list 'a) : list 'a
```

- A recursive function which can be used in programs and specifications:

```
let rec function length (l: list 'a) : int =  
  match l with  
  | Nil          -> 0  
  | Cons _ r    -> 1 + length r  
end
```

Add ``rec`` after `let` for recursive programs

WhyML

- Termination of Recursive Definitions -

- Why3 automatically verifies that recursive definitions are terminating by looking for an appropriate lexicographic order of arguments that guarantees a structural descent
- If Why3 fails, provide a variant in the function specification

```
function length (l: list 'a) : int =  
  match l with  
  | Nil          -> 0  
  | Cons _ r    -> 1 + length r  
end
```

No ``rec`` is needed for logical function symbols

WhyML

- Partial Correctness -

- Partial correctness of program functions is ensured by adding the `diverges` clause

```
let rec ackermann (m n: int) : int
  requires { 0 <= m /\ 0 <= n }
  diverges
  =
  if m = 0 then n + 1
  else if n = 0 then ackermann (m - 1, 1)
  else ackermann (m - 1, ackermann (m, n - 1))
```

WhyML

- Ghost Code -

- Ghost functions can be defined or declared by adding the keyword `ghost` after `let` or `val`
- Ghost code is never translated into executable code
- Ghost code can never affect the computation of the program

Exercise

- 64-bit machine integers
 - Define a type for 64-bit machine integers
 - Define a multiplication function
- Balanced binary trees
 - Define a type for binary trees
 - Define a function to compute the height of a binary tree
 - Define a type for balanced binary trees

WhyML

- Inductive Predicates -

- Such a predicate is the least relation satisfying a set of clauses

```
inductive sub (list 'a) (list 'a) =  
  | empty: sub (Nil: list 'a) (Nil: list 'a)  
  | cons : forall x: 'a. forall s1 s2: list 'a.  
           sub s1 s2 -> sub (Cons x s1) (Cons x s2)  
  | dive : forall x: 'a. forall s1 s2: list 'a.  
           sub s1 s2 -> sub s1 (Cons x s2)
```

- Standard positivity restrictions apply to ensure the existence of a least fixed point

WhyML

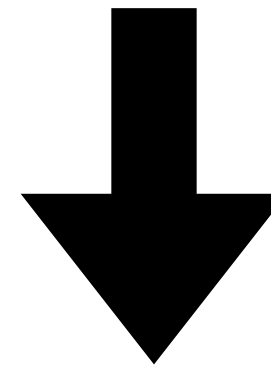
- Terms and Formulas -

- The first-order language in Why3 is extended, both in terms and formulas, with pattern matching, let-expressions, and conditional (if-then-else) expressions
- To make distinction between terms and formulas
 - conditional expressions are allowed in terms but
 - is lifted to the level of formulas

WhyML

- Lifting Conditional Expressions -

```
function max (x y: int) : int = if x >= y then x else y
```



```
function max (x y: int) : int
```

```
axiom max_listed:
```

```
(x >= y -> max x y = x)
```

```
/\
```

```
(not (x >= y) -> max x y = y)
```

First Example

- Find an element in a sorted list
 - What is a list?
 - What is a sorted list?
 - How to implement the algorithm?
 - What are the specifications?

Polymorphic Lists

```
module List
  type list 'a = Nil | Cons 'a (list 'a)
  let predicate is_nil (l:list 'a)
    ensures { result <-> l = Nil }
  = match l with
    Nil      -> true
    | Cons _ _ -> false
  end
end
```

Length of a List

```
module Length
  use int.Int
  use List
  let rec function length (l: list 'a) : int =
    match l with
    | Nil          -> 0
    | Cons _ r    -> 1 + length r
  end
  lemma Length_nonnegative: forall l: list 'a. length l >= 0
  lemma Length_nil: forall l: list 'a. length l = 0 <-> l = Nil
end
```



Length of a List

```
module Length
  use int.Int
  use List
  let rec function length (l: list 'a) : int =
    match l with
    | Nil          -> 0
    | Cons _ r    -> 1 + length r
  end
  lemma Length_nonnegative: forall l: list 'a. length l >= 0
  lemma Length_nil: forall l: list 'a. length l = 0 <-> l = Nil
end
```

Prove the lemmas automatically?



Length of a List

- Proving with Transformations -

The transformation *induction_ty_lex* transforms the following goal

```
goal G: forall l: list 'a. length l >= 0
```

into this one:

When induction can be applied to several variables,
the transformation picks one heuristically

```
goal G: forall l: list 'a.  
  match l with  
  | Nil          -> length l >= 0  
  | Cons a l1 -> length l1 >= 0 -> length l >= 0  
end
```

Membership in a List

```
module Mem
  use List

  predicate mem (x: 'a) (l: list 'a) =
    match l with
    | Nil          -> false
    | Cons y r    -> x = y \/\ mem x r
    end
end
```

Use VS Clone

- When we use a theory, there is no duplication
- When we clone a theory, a local copy of its declarations is created
 - Any axiom is automatically turned into a lemma (or remains an axiom if explicitly indicated)
 - Lemmas in the module being cloned are not reproved
 - Functions defined in the module being cloned are not reproved

Sorted Lists

- Definition -

```
module Sorted
  use List
  type t
  predicate le t t
  clone relations.Transitive with type t = t, predicate rel = le, axiom Trans
  inductive sorted (l: list t) =
    | Sorted_Nil: sorted Nil
    | Sorted_One: forall x: t. sorted (Cons x Nil)
    | Sorted_Two: forall x y: t, l: list t.
      le x y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))
  ...
```

Sorted Lists

- Definition -

```
module Sorted
  use L
  type
  predi
  clone
  induc
  | S
  | S
  | S
  ...
  module EndoRelation
    type t
    predicate rel t t
  end
  module Transitive
    clone export EndoRelation
    axiom Trans : forall x y z:t. rel x y -> rel y z -> rel x z
  end
end
```

Sorted Lists

- Definition -

```
module Sorted
  use List
  type t
  predicate le t t
  clone relations.Transitive with type t = t, predicate rel = le, axiom Trans
  inductive sorted (l: list t) =
    | Sorted_Nil: sorted Nil
    | Sorted_One: forall x: t. sorted (Cons x Nil)
    | Sorted_Two: forall x y: t, l: list t.
      le x y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))
  ...
```

Sorted Lists

- Lemmas -


```
module Sorted
...
use Mem
lemma sorted_mem:
  forall x: t, l: list t.
  (forall y: t. mem y l -> le x y) /\ sorted l <-> sorted (Cons x l)
use Append
lemma sorted_append:
  forall l1 [@induction] l2: list t.
  (sorted l1 /\ sorted l2 /\ (forall x y: t. mem x l1 -> mem y l2 -> le x y))
  <-> sorted (l1 ++ l2)
end
```

Force induction over
one particular variable



Sorted Lists of Integers

```
module SortedInt  
  
  use int.Int  
  clone export Sorted with type t = int,  
    predicate le = (<=),  
    goal Transitive.Trans  
  
end
```



Transitivity can be proven now

Find a Value in a Sorted List of Integers

```
module FindInSortedList
  use int.Int, List, Mem, SortedInt
  lemma Sorted_not_mem: forall x y : int, l : list int.
    x < y -> sorted (Cons y l) -> not mem x (Cons y l)

  let rec find x l
    requires { sorted l }
    variant { l }
    ensures { result = True <-> mem x l }
  = match l with
    | Nil -> False
    | Cons y r -> x = y || x > y && find x r
  end
end
```

Find a Value in a Sorted List of Integers

- Generated Verification Condition -

```
goal find'vc :  
  forall x:int, l:list int.  
    sorted1 l ->  
    A /\ B
```

Definitions of A and B are given in the following two slides

Find a Value in a Sorted List of Integers

- Generated Verification Condition: A -

```
match l with
| Nil -> true
| Cons y r ->
    not x = y ->
    x > y ->
    match l with
    | Nil -> false
    | Cons _ f -> f = r
    end /\ sorted1 r
end
```

Prove **requires** and **variant** for the recursive call

Find a Value in a Sorted List of Integers

- Generated Verification Condition: B -

```
(forall result:bool.  
  match l with  
  | Nil -> result = False  
  | Cons y r ->  
    if x = y then result = True  
    else result  
      = (if x > y then if mem x r then True else False  
         else False)  
  end -> result = True <-> mem x l)
```

Prove ensures

Find a Value in a Sorted List of Integers

- Code Extraction -

The OCaml code can be extracted by the following command
\$ why3 extract -D ocaml64 find_sorted.mlw

```
type 'a list =  
  | Nil  
  | Cons of 'a * 'a list  
...  
let rec find (x: Z.t) (l: (Z.t) list) : bool =  
  match l with  
  | Nil -> false  
  | Cons (y, r) -> Z.equal x y || Z.gt x y && find x r  
...
```

Interpreting WhyML Programs

Add the following code to module FindInSortedList

```
let test () =  
  let l = Cons 3 (Cons 5 (Cons 8 (Cons 13 (Cons 21 Nil)))) in  
  find 9 l
```

Run function test with the following command:

```
$ why3 execute find_sorted.mlw FindInSortedList.test
```

(assume FindInSortedList is defined in file find_sorted.mlw)



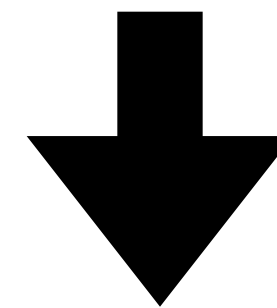
More Transformations

- Besides `split_vc` and `induction_ty_lex`, there are many other transformations
 - Most of them behave the same as in Coq
- Read the manual for more details

More Transformations

- apply -

```
predicate is_even int
predicate is_zero int
axiom zero_is_even: forall x: int. is_zero x -> is_even x
goal G: is_even 0
```



apply zero_is_even

```
predicate is_even int
predicate is_zero int
axiom zero_is_even: forall x: int. is_zero x -> is_even x
goal G: is_zero 0
```


Apply Transformations with Arguments

- IDE Case -

The screenshot shows the Why3 Interactive Proof Session IDE. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help".

The interface is divided into several sections:

- Left Panel:** A tree view showing the project structure. It includes a "Status" column with question marks, a "Theories/Goals" column with a folder icon for "transform_apply.mlw", a sub-folder "Top", and a goal "G".
- Task Panel:** Shows the current task "transform_apply.mlw" with a list of commands:

```
1 predicate is_even int
2 predicate is_zero int
3 axiom zero_is_zero: is_zero 0
4 axiom zero_is_even: forall x: int. is_zero x -> is_even x
5 goal G: is_even 0
6
```
- Command Input:** A text box containing "0/0/0" and "type commands here", which is highlighted with a red rectangle.
- Messages Panel:** Contains the following text:

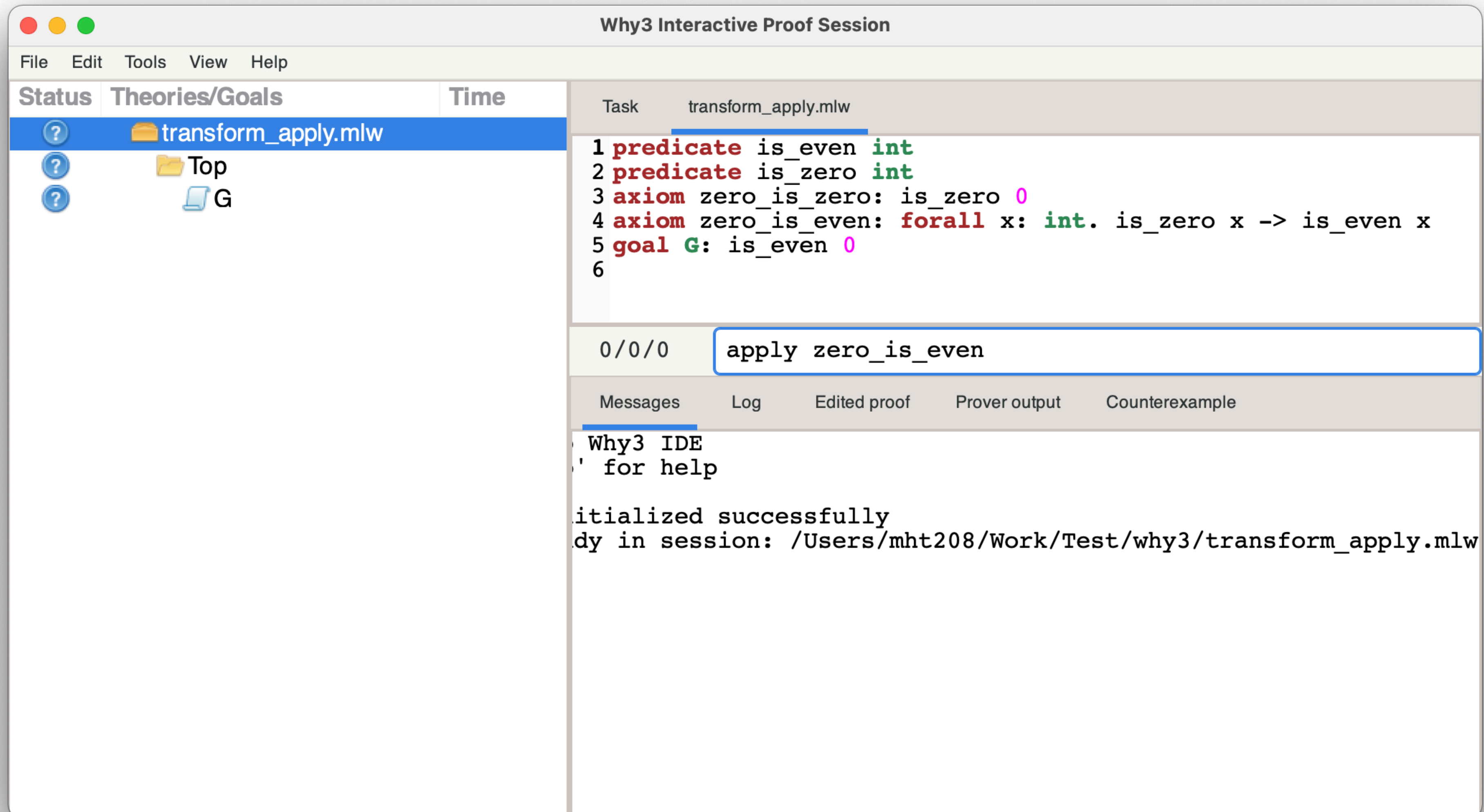
```
Why3 IDE
' for help

itialized successfully
dy in session: /Users/mht208/Work/Test/why3/transform_apply.mlw
```



Apply Transformations with Arguments

- IDE Case -



The screenshot shows the Why3 Interactive Proof Session IDE. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help".

The left sidebar shows a tree view with the following structure:

- transform_apply.mlw (selected)
- Top
- G

The main editor area shows the following code:

```
1 predicate is_even int
2 predicate is_zero int
3 axiom zero_is_zero: is_zero 0
4 axiom zero_is_even: forall x: int. is_zero x -> is_even x
5 goal G: is_even 0
6
```

Below the code, there is a command input field with the text "0/0/0" and "apply zero_is_even".

The bottom panel shows the "Messages" tab with the following output:

```
Why3 IDE
' for help

itialized successfully
dy in session: /Users/mht208/Work/Test/why3/transform_apply.mlw
```



Apply Transformations with Arguments

- IDE Case -

The screenshot shows the Why3 Interactive Proof Session IDE. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help".

The interface is divided into several sections:

- Left Panel (Theories/Goals):** A tree view showing the project structure. The root is "transform_apply.mlw", which contains a "Top" folder. Inside "Top" is a goal "G". Under "G", there is a goal "0 [apply premises]".
- Task Panel (transform_apply.mlw):** A list of code lines:

```
1 predicate is_even int
2 predicate is_zero int
3 axiom zero_is_zero: is_zero 0
4 axiom zero_is_even: forall x: int. is_zero x -> is_even x
5 goal G: is_even 0
6
```
- Progress Bar:** Shows "0/0/0" and a progress indicator.
- Messages Panel:** A tabbed area with "Messages", "Log", "Edited proof", "Prover output", and "Counterexample". The "Messages" tab is currently selected and is empty.



Apply Transformations with Arguments

- IDE Case -

The screenshot shows the Why3 Interactive Proof Session IDE. The window title is "Why3 Interactive Proof Session". The menu bar includes "File", "Edit", "Tools", "View", and "Help".

The interface is divided into several panels:

- Theories/Goals:** A tree view on the left showing the project structure. The root is "transform_apply.mlw", which contains a "Top" folder. Inside "Top" is a goal "G". Under "G", there is a tactic "apply zero_is_even" and a goal "0 [apply premises]". The goal "0 [apply premises]" is currently selected and highlighted in blue.
- Task:** A panel on the right showing the current task "transform_apply.mlw". It displays the following code:

```
8  
9 axiom zero_is_even : forall x:int. is_zero x -> is_even x  
10  
11 ----- Goal -----  
12  
13 goal G : is_zero 0  
14  
15 |
```
- Progress:** A small display showing "0/0/0" and an empty input field.
- Messages:** A panel at the bottom with tabs for "Messages", "Log", "Edited proof", "Prover output", and "Counterexample". The "Messages" tab is currently selected and empty.



Apply Transformations with Arguments

- Command-Line Case -

1. Run `why3 shell` to open an interactive shell

```
why3 (why3\ shell)
> why3 shell transform_apply.mlw
Welcome to Why3 shell. Type 'help' for help.
> root
File /Users/mht208/Work/Test/why3/transform_apply.mlw is:
predicate is_even int
predicate is_zero int
axiom zero_is_zero: is_zero 0
axiom zero_is_even: forall x: int. is_zero x -> is_even x
goal G: is_even 0
Session initialized successfully
root File transform_apply.mlw, id 1;
  [ Theory Top, id: 2; [{ Goal=G, id = 3; parent=Top;
  [] [] }]];

```

proof tree



Apply Transformations with Arguments

- Command-Line Case -

2. Select goal by `goto n` where n is the ID of the goal

```
why3 (why3\ shell)
goto 3
root File transform_apply.mlw, id 1;
  [ Theory Top, id: 2; [**{ Goal=G, id = 3; parent=Top; [] [] } **]];
No goal
> Goal is:
----- Local Context -----

predicate is_even int

predicate is_zero int

axiom zero_is_zero : is_zero 0

axiom zero_is_even : forall x:int. is_zero x -> is_even x

----- Goal -----

goal G : is_even 0
```

Apply Transformations with Arguments

- Command-Line Case -

3. Apply the `apply` transformation

```
why3 (why3\ shell)
apply zero_is_even
Goal is:
----- Local Context -----

predicate is_even int
predicate is_zero int

axiom zero_is_zero : is_zero 0

axiom zero_is_even : forall x:int. is_zero x -> is_even x

----- Goal -----

goal G : is_even 0

> █
```

Apply Transformations with Arguments

- Command-Line Case -

4. Goto the node of the generated sub-goal

```
why3 (why3\ shell)

goal G : is_even 0

p
root File transform_apply.mlw, id 1;
  [ Theory Top, id: 2;
    [**{ Goal=G, id = 3;
      parent=Top;
      []
      [{ Trans=apply zero_is_even;
        args=;
        parent=G;
        [{ Goal=0 [apply premises], id = 5;
          parent=apply zero_is_even;
          []
          [] }] }] } **]];

Goal is:
```

p: print the session

Apply Transformations with Arguments

- Command-Line Case -

4. Goto the node of the generated sub-goal

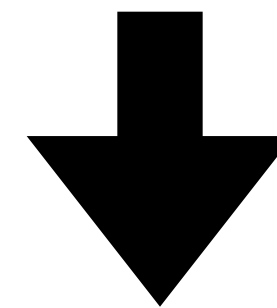
```
why3 (why3\ shell)
No goal
> Goal is:
----- Local Context -----
predicate is_even int
predicate is_zero int
axiom zero_is_zero : is_zero 0
axiom zero_is_even : forall x:int. is_zero x -> is_even x
----- Goal -----
goal G : is_zero 0
```

after `goto 5`

More Transformations

- apply with -

```
axiom ac: a = c
axiom cb: c = b
axiom transitivity : forall x y z:int. x = y -> y = z -> x = z
goal G1 : a = b
```



apply transitivity with c

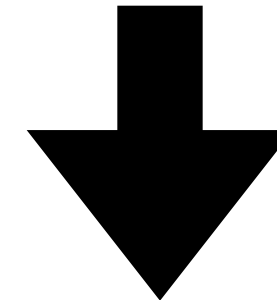
```
axiom ac: a = c
axiom cb: c = b
axiom transitivity : forall x y
z:int. x = y -> y = z -> x = z
goal G1 : a = c
```

```
axiom ac: a = c
axiom cb: c = b
axiom transitivity : forall x y
z:int. x = y -> y = z -> x = z
goal G1 : c = b
```

More Transformations

- case -

```
constant x : int
constant y : int
goal G: if x = 0 then y = 2 else y = 3
```



case (x = 0)

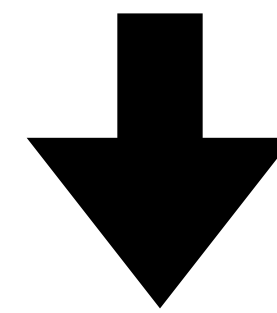
```
constant x : int
constant y : int
axiom h : x = 0
goal G: if x = 0 then y = 2 else y
= 3
```

```
constant x : int
constant y : int
axiom h : not x = 0
goal G: if x = 0 then y = 2 else y
= 3
```

More Transformations

- intros -

```
goal G : forall x y: int. x + y = y + x
```



intros x,y

```
----- Local Context -----
```

```
constant x : int
```

```
constant y : int
```

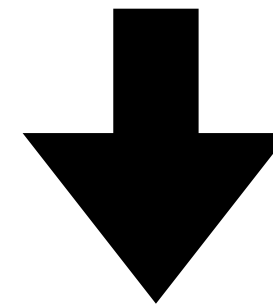
```
----- Goal -----
```

```
goal G : (x + y) = (y + x)
```

More Transformations

- split_goal_right -

```
goal G: (p /\ q) -> (r /\ s)
```

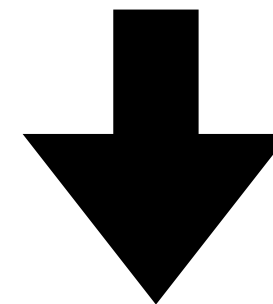


split_goal_right

```
goal G: (p /\ q) -> r
```

```
goal G: (p /\ q) -> s
```

```
goal G: (p /\ q) -> ((r /\ s) \/ (p /\ q))
```



split_goal_right

```
goal G: (p /\ q) -> ((r /\ s) \/ (p /\ q))
```

More Transformations

- induction -

```
----- Local Context -----  
constant x : int  
----- Goal -----  
goal G : forall y: int. (x + y) = (y + x)
```



induction x

```
-----  
constant x  
Init : x <=  
-----  
goal G : fo
```

```
----- Local Context -----  
constant x : int  
Init : 0 < x  
Hrec : forall n:int. n < x -> (forall y:int. (n + y) = (y + n))  
----- Goal -----  
goal G : forall y:int. (x + y) = (y + x)
```

Second Example

We want to prove the following lemma

```
lemma pigeonhole :  
  forall n m: int. forall f: int -> int.  
    (0 <= m < n) ->  
    (forall i. 0 <= i < n -> 0 <= f i < m) ->  
    (exists i1, i2. 0 <= i1 < i2 < n /\ f i1 = f i2)
```

Can we prove it automatically with Why3?

How do you informally prove it?

Pigeonhole

- Ghost Code -

```
use int.Int, set.Fset, ref.Ref

let rec ghost below (n: int) : fset int
  requires { 0 <= n }
  ensures  { forall i. mem i result <-> 0 <= i < n }
  ensures  { cardinal result = n }
  variant  { n }
= if n = 0 then empty else add (n-1) (below (n-1))
```


Pigeonhole

- Lemma Function -

- `let lemma` defines a special pure program function which serves
- not as an actual code to execute
 - but to prove the function's contract as a lemma

```
let lemma pigeonhole (n m: int) (f: int -> int)
  requires { 0 <= m < n }
  requires { forall i. 0 <= i < n -> 0 <= f i < m }
  ensures  { exists i1, i2. 0 <= i1 < i2 < n /\ f i1 = f i2 }
=
...
```

Pigeonhole

- Lemma Function -

```
let lemma pigeonhole (n m: int) (f: int -> int)
```

```
...
```

```
=
```

```
let s = ref empty in
```

```
for i = 0 to n-1 do
```

```
  invariant { cardinal !s = i }
```

```
  invariant { forall x. mem x !s <-> (exists j. 0 <= j < i /\ x = f j) }
```

```
  if mem (f i) !s then return;
```

```
  s := add (f i) !s
```

```
done;
```

```
let b = below m in assert { subset !s b };
```

```
absurd
```

absurd: denotes an unreachable piece of code

Usage of Pigeonhole

- Consider a mex (minimum excluded) problem below
 - Given a finite subset of integers, find the smallest nonnegative integer that does not belong to the subset
- Assume the subset is given as an array A
 - If n is the size of the array, it is clear that $0 \leq \text{mex} \leq n$ (pigeonhole principle)
- A simple algorithm
 - marks values that belongs to $0..n-1$ in some external Boolean array of length n , and then
 - scans the Boolean array to find the first unused value

Mex

- mem -

```
module MexArray
  use int.Int
  use map.Map
  use array.Array
  use ref.Refint
  use pigeon.Pigeonhole

  predicate mem (x: int) (a: array int) =
    exists i. 0 <= i < length a && a[i] = x

  ...
end
```

Mex

- Specification -

```
let mex (a: array int) : int
  ensures { 0 <= result <= length a }
  ensures { not (mem result a) }
  ensures { forall x. 0 <= x < result -> mem x a }
= ...
```

Mex

- First Scan -

```
= let n = length a in
  let used = make n false in
  let ghost idx = ref (fun i -> i) in (* the position of each marked value *)
  for i = 0 to n - 1 do
    invariant { forall x. 0 <= x < n -> used[x] ->
      mem x a && 0 <= !idx x < n && a[!idx x] = x }
    invariant { forall j. 0 <= j < i -> 0 <= a[j] < n ->
      used[a[j]] }
    let x = a[i] in
    if 0 <= x && x < n then begin used[x] <- true; idx := set !idx x i end
  done;
```

Mex

- Second Scan -

```
let r = ref 0 in
let ghost posn = ref (-1) in
while !r < n && used[!r] do
  invariant { 0 <= !r <= n }
  invariant { forall j. 0 <= j < !r -> used[j] && 0 <= !idx j < n }
  invariant { if !posn >= 0 then 0 <= !posn < n && a[!posn] = n
              else forall j. 0 <= j < !r -> a[j] <> n }

  variant { n - !r }
  if a[!r] = n then posn := !r;
  incr r
done;
(* we cannot have !r=n (all values marked) and !posn>=0 at the same time *)
if !r = n && !posn >= 0 then pigeonhole (n+1) n (set !idx n !posn);
!r
```

Mex

- Without/With Pigeonhole -

```

----- Local Context -----
-
H4 : 0 <= ((n - 1) + 1)
LoopInvariant3 :
  forall x:int.
    0 <= x /\ x < n ->
      used[x] = True ->
        mem x a && (0 <= (idx @ x) /\ (idx @ x) < n) && a[idx @ x] = x
LoopInvariant2 :
  forall j:int.
    0 <= j /\ j < ((n - 1) + 1) -> 0 <= a[j] /\ a[j] < n -> used[a[j]] = True
LoopInvariant1 :
  forall j:int.
    0 <= j /\ j < r -> used[j] = True && 0 <= (idx @ j) /\ (idx @ j) < n
LoopInvariant :
  if posn >= 0 then (0 <= posn /\ posn < n) && a[posn] = n
  else forall j:int. 0 <= j /\ j < r -> not a[j] = n
h2 : r = n
h1 : mem r a
h : posn >= 0
----- Goal -----

goal mex'vc : not mem r a

```

an unproven task without pigeonhole

```

----- Local Context -----
...
LoopInvariant3 :
  forall x:int.
    0 <= x /\ x < n ->
      used[x] = True ->
        mem x a && (0 <= (idx @ x) /\ (idx @ x) < n) && a[idx @ x] = x
LoopInvariant2 :
  forall j:int.
    0 <= j /\ j < ((n - 1) + 1) -> 0 <= a[j] /\ a[j] < n -> used[a[j]] = True
LoopInvariant1 :
  forall j:int.
    0 <= j /\ j < r -> used[j] = True && 0 <= (idx @ j) /\ (idx @ j) < n
LoopInvariant :
  if posn >= 0 then (0 <= posn /\ posn < n) && a[posn] = n
  else forall j:int. 0 <= j /\ j < r -> not a[j] = n
H1 : not (if r < n then used[r] else False) = True
H :
  not (r = n && posn >= 0) \/\
  (let o = set idx n posn in
   exists i1:int, i2:int.
     (0 <= i1 /\ i1 < i2 /\ i2 < (n + 1)) /\ (o @ i1) = (o @ i2))
h2 : r = n
h1 : mem r a
h : posn >= 0
----- Goal -----

goal mex'vc : not mem r a

```

the previously unproven task with pigeonhole

Mex

- Without/With Pigeonhole -

Find the difference between the two tasks by

1. Remove ``if ... then pigeonhole (n+1) n (set !idx n !posn);``
2. Apply transformation `split_vc`
3. Prove all subgoals with `alt-ergo`
4. Apply the following transformations to the unproven task
 1. `case (r = n)`
 2. `case (mem r a)`
 3. `case (posn >= 0)`
5. Add the removed line back and do again

an unproven task without pigeonhole

the previously unproven task with pigeonhole

Third Example

- Nistonacci Numbers -

The following function is to be implemented

$$\text{nist}(n) = \begin{cases} n & \text{if } n < 2 \\ \text{nist}(n - 2) + 2 * \text{nist}(n - 1) & \text{otherwise} \end{cases}$$

where n is a natural number

We want to prove that the implementation `nist_impl` satisfies:

forall $n: \text{int}. n \geq 0 \rightarrow \text{nist_impl}(n) \geq n$

Nistonacci Numbers

- Implementation -

```
let rec function nist_impl (n: int): int
  requires { n >= 0 }
  variant { n }
  ensures { result >= n }
= let x = ref 0 in
  let y = ref 1 in
  for i=0 to n-1 do
    invariant { !x = nist_impl(i) }
    invariant { !y = nist_impl(i + 1) }
    let tmp = !x in
      x := !y;
      y := tmp + 2 * !y
  done;
!x
```

Error:
unbound function or predicate symbol 'nist_impl'

Nistonacci Numbers

- Reference Implementation -

```
let rec ghost function nist (n: int): int
  requires { n >= 0 }
  variant { n }
= if n < 2 then n
  else nist (n-2) + 2 * nist (n-1)
```

```
let function nist_impl (n: int): int
  requires { n >= 0 }
  ensures { result = nist n }
= let x = ref 0 in
  let y = ref 1 in
  for i=0 to n-1 do
    invariant { !x = nist i }
    invariant { !y = nist (i + 1) }
    ...
  ...
```

Nistonacci Numbers

- Correctness -

```
lemma nist_impl_correct:  
  forall n: int. n >= 0 -> nist_impl(n) >= n
```

Can be proven by transformations:

1. intros n
2. induction n

Nistonacci Numbers

- Get Rid of Transformations -

```
let rec lemma nist_ge_n (n: int)
  requires { n >= 0 }
  variant { n }
  ensures { nist(n) >= n }
= if n >= 2 then
  begin
    nist_ge_n (n-1);
    nist_ge_n (n-2)
  end
```

introduce induction hypotheses

References

- Why3: Shepherd Your Herd of Provers (BOOGIE 2011)
- Expressing Polymorphic Types in a Many-Sorted Language (FroCos 2011)
- Why3 -- Where Programs Meet Provers (ESOP 2013)
- Why3 manual
- Why3 examples (<http://toccata.lri.fr/gallery/why3.en.html>)