The Why/Krakatoa/Caduceus Platform for Deductive Program Verification

Jean-Christophe Filliâtre

CNRS – Université Paris Sud

TYPES Summer School – August 30th, 2007

Introduction

Provers based on HOL are suitable tools to verify **purely functional** programs (see other lectures)

But how to verify an **imperative program** with your favorite prover?

for instance this one

```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,(
c+d)/2));return f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0<<q),0,0));}
```

Usual methods

- Floyd-Hoare logic
- Dijkstra's weakest preconditions

- could be formalized in the prover (deep embedding)
- could be applied by a tactic (shallow embedding)

$\Rightarrow$ would be **specific to this prover**

Which programming language?

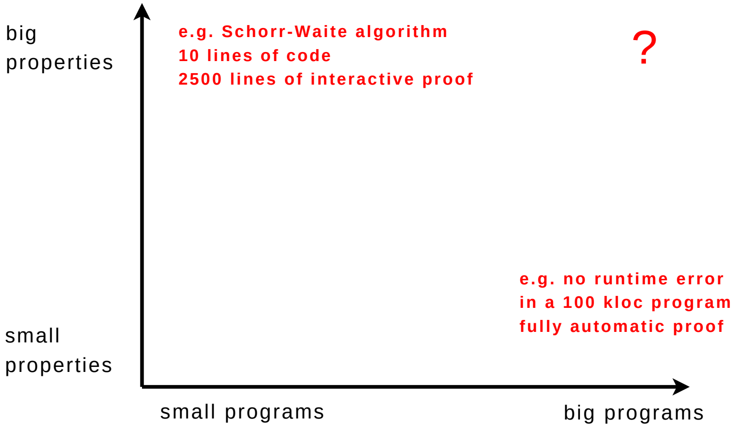
a realistic existing programming language such as C or Java?

- many constructs $\Rightarrow$ many rules
- would be **specific to this language**

The ProVal project — `http://proval.lri.fr/`

- general **goal**: prove behavioral properties of **pointer programs**
- pointer program = program manipulating data structures with **in-place mutable fields**
- we currently focus on **C** and **Java** programs

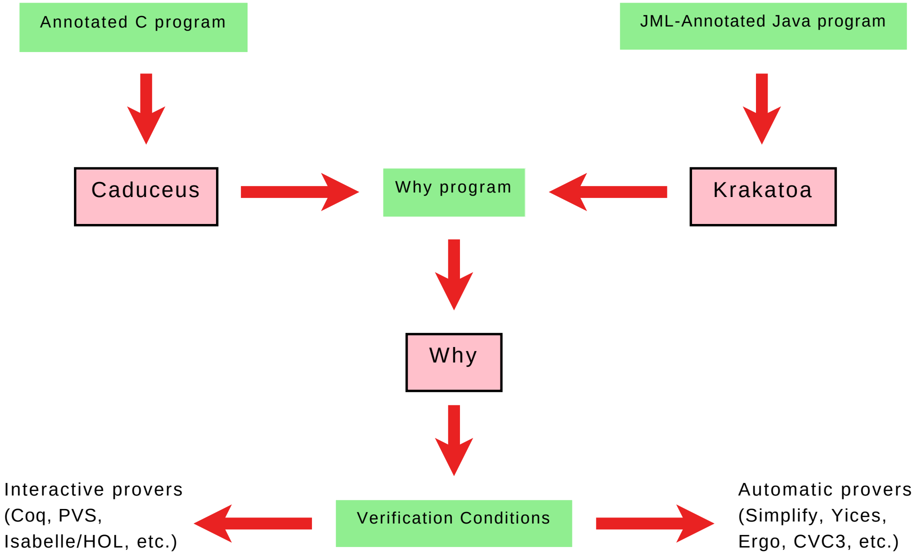
What kind of properties



Principles

- specification as **annotations** at the source code level
  - JML (Java Modeling Language) for Java
  - our own language for C (mostly JML-inspired)
- generation of **verification conditions** (VCs)
  - using Hoare logic / weakest preconditions
  - other similar approaches: static verification (ESC/Java, SPEC#), B method, etc.
- **multi-prover** approach
  - off-the-shelf provers, as many as possible
  - automatic provers (Simplify, Yices, Ergo, etc.)
  - proof assistants (Coq, PVS, Isabelle/HOL, etc.)

Platform Overview



Outline

1. An intermediate language for program verification
   1. syntax, typing, semantics, proof rules
   2. the Why tool
   3. multi-prover approach
2. Verifying C and Java programs
   1. specification languages
   2. models of program execution
3. A challenging case study

part I

An Intermediate Language for Program Verification

Basic Idea

makes program verification

- prover-independent but prover-aware
- language-independent

so that we can use it to verify C, Java, etc. programs with HOL provers but also with FO decision procedures

The essence of Hoare logic: assignment rule

$$\{ \ P[x \leftarrow E] \ \} \ x \ := \ E \ \{ \ P \ \}$$

1. absence of aliasing
2. side-effects free E **shared** between program and logic

Data types

Any purely applicative data type from the logic can be used in programs

Example: a data type `int` for integers with constants 0, 1, etc. and operations +, ∗, etc.
The pure expression 1+2 belongs to both programs and logic

A single data structure: the **reference** (mutable variable) containing **only pure values**, with no possible alias between two different references

Data types

Any purely applicative data type from the logic can be used in programs

Example: a data type `int` for integers with constants 0, 1, etc. and operations +, $*$, etc.
The pure expression 1+2 belongs to both programs and logic

A single data structure: the **reference** (mutable variable) containing **only pure values**, with no possible alias between two different references

Data types

Any purely applicative data type from the logic can be used in programs

Example: a data type `int` for integers with constants 0, 1, etc. and operations +, ∗, etc.
The pure expression 1+2 belongs to both programs and logic

A single data structure: the **reference** (mutable variable) containing **only pure values**, with no possible alias between two different references

Data types

Any purely applicative data type from the logic can be used in programs

Example: a data type `int` for integers with constants 0, 1, etc. and operations +, *, etc.
The pure expression 1+2 belongs to both programs and logic

A single data structure: the **reference** (mutable variable) containing **only pure values**, with no possible alias between two different references

ML syntax

No distinction between expressions and statements
⇒ less constructs
⇒ less rules

dereference	`!`x
assignment	x `:=` e
local variable	`let` x `=` e_1 `in` e_2
local reference	`let` x `= ref` e_1 `in` e_2
conditional	`if` e_1 `then` e_2 `else` e_3
loop	`while` e_1 `do` e_2 `done`

sequence e_1`;` e_2 ≡ `let _ =` e_1 `in` e_2

ML syntax

No distinction between expressions and statements
⇒ less constructs
⇒ less rules

dereference	`!`x
assignment	x `:=` e
local variable	`let` x `=` e_1 `in` e_2
local reference	`let` x `= ref` e_1 `in` e_2
conditional	`if` e_1 `then` e_2 `else` e_3
loop	`while` e_1 `do` e_2 `done`

sequence e_1 `;` e_2 $\equiv$ `let _ =` e_1 `in` e_2

ML syntax

No distinction between expressions and statements
⇒ less constructs
⇒ less rules

dereference	`!`x
assignment	x `:=` e
local variable	`let` x `=` e_1 `in` e_2
local reference	`let` x `= ref` e_1 `in` e_2
conditional	`if` e_1 `then` e_2 `else` e_3
loop	`while` e_1 `do` e_2 `done`
sequence e_1 `;` e_2	$\equiv$ `let _ =` e_1 `in` e_2

ML syntax

No distinction between expressions and statements
⇒ less constructs
⇒ less rules

dereference	`!`x
assignment	x `:=` e
local variable	`let` x `=` e_1 `in` e_2
local reference	`let` x `= ref` e_1 `in` e_2
conditional	`if` e_1 `then` e_2 `else` e_3
loop	`while` e_1 `do` e_2 `done`
sequence e_1 ; e_2	$\equiv$ `let _ =` e_1 `in` e_2

ML syntax

No distinction between expressions and statements
⇒ less constructs
⇒ less rules

dereference	`!`x
assignment	x `:=` e
local variable	`let` x `=` e_1 `in` e_2
local reference	`let` x `= ref` e_1 `in` e_2
conditional	`if` e_1 `then` e_2 `else` e_3
loop	`while` e_1 `do` e_2 `done`
sequence e_1 `;` e_2	$\equiv$ `let _ =` e_1 `in` e_2

ML syntax

No distinction between expressions and statements
$\Rightarrow$ less constructs
$\Rightarrow$ less rules

dereference	`!`x
assignment	x `:=` e
local variable	`let` x `=` e_1 `in` e_2
local reference	`let` x `= ref` e_1 `in` e_2
conditional	`if` e_1 `then` e_2 `else` e_3
loop	`while` e_1 `do` e_2 `done`

sequence e_1 ; e_2 $\equiv$ `let _ =` e_1 `in` e_2

ML syntax

No distinction between expressions and statements
$\Rightarrow$ less constructs
$\Rightarrow$ less rules

dereference	`!`x
assignment	x `:=` e
local variable	`let` x `=` e_1 `in` e_2
local reference	`let` x `= ref` e_1 `in` e_2
conditional	`if` e_1 `then` e_2 `else` e_3
loop	`while` e_1 `do` e_2 `done`

sequence e_1 `;` e_2 $\equiv$ `let _ =` e_1 `in` e_2

ML syntax

No distinction between expressions and statements
$\Rightarrow$ less constructs
$\Rightarrow$ less rules

dereference	`!`x
assignment	x `:=` e
local variable	`let` x `=` e_1 `in` e_2
local reference	`let` x `= ref` e_1 `in` e_2
conditional	`if` e_1 `then` e_2 `else` e_3
loop	`while` e_1 `do` e_2 `done`
sequence e_1; e_2	$\equiv$ `let _ =` e_1 `in` e_2

Annotations

- $\texttt{assert}\ \{p\};\ e$
- $e\ \{p\}$

Examples:

- $\texttt{assert}\ \{x > 0\};\ 1/x$
- $x\ \texttt{:=}\ 0\ \{!x = 0\}$
- $\texttt{if}\ !x > !y\ \texttt{then}\ !x\ \texttt{else}\ !y\ \{\mathit{result} \geq !x \land \mathit{result} \geq !y\}$
- $x\ \texttt{:=}\ !x + 1\ \{!x > \texttt{old}(!x)\}$

Annotations

- $\texttt{assert}\ \{p\};\ e$
- $e\ \{p\}$

Examples:

- $\texttt{assert}\ \{x > 0\};\ 1/x$
- $x\ := 0\ \{!x = 0\}$
- $\texttt{if}\ !x > !y\ \texttt{then}\ !x\ \texttt{else}\ !y\ \{\mathit{result} \geq !x \wedge \mathit{result} \geq !y\}$
- $x\ := !x + 1\ \{!x > \texttt{old}(!x)\}$

Annotations

- assert $\{p\}$; e
- e $\{p\}$

Examples:

- assert $\{x > 0\}$; $1/x$
- x := 0 $\{!x = 0\}$
- if $!x > !y$ then $!x$ else $!y$ $\{result \geq !x \wedge result \geq !y\}$
- x := $!x + 1$ $\{!x > \mathtt{old}(!x)\}$

Annotations

- $\texttt{assert}\ \{p\};\ e$
- $e\ \{p\}$

Examples:

- $\texttt{assert}\ \{x > 0\};\ 1/x$
- $x\ \texttt{:=}\ 0\ \{\texttt{!}x = 0\}$
- $\texttt{if}\ \texttt{!}x > \texttt{!}y\ \texttt{then}\ \texttt{!}x\ \texttt{else}\ \texttt{!}y\ \{\mathit{result} \geq \texttt{!}x \wedge \mathit{result} \geq \texttt{!}y\}$
- $x\ \texttt{:=}\ \texttt{!}x + 1\ \{\texttt{!}x > \texttt{old}(\texttt{!}x)\}$

Annotations (cont'd)

Loop invariant and variant

- while e_1 do {invariant p variant t} e_2 done

Example:

```
while !x < N do
  { invariant !x ≤ N variant N − !x }
  x := !x + 1
done
```

Annotations (cont'd)

Loop invariant and variant

- while e_1 do {invariant p variant t} e_2 done

Example:

```
while !x < N do
  { invariant !x ≤ N variant N − !x }
  x := !x + 1
done
```

Auxiliary variables

Used to denote the **intermediate** values of variables

Example: $\ldots\ \{!x = X\}\ \ldots\ \{!x > X\}\ \ldots$

We will use **labels** instead

- new construct L:e
- new annotation `at`(t, L)

Example:

$$\begin{array}{l} \quad \vdots \\ L : \texttt{while}\ \ldots\ \texttt{do}\ \{\ \texttt{invariant}\ !x \geq \texttt{at}(!x, L)\ \ldots\ \} \\ \qquad \ldots \\ \quad \texttt{done} \end{array}$$

Auxiliary variables

Used to denote the **intermediate** values of variables

Example: $\ldots\ \{!x = X\}\ \ldots\ \{!x > X\}\ \ldots$

We will use **labels** instead

- new construct L:e
- new annotation $\mathtt{at}(t, L)$

Example:

$$\begin{array}{l} \quad\vdots \\ L : \mathtt{while}\ \ldots\ \mathtt{do}\ \{\ \mathtt{invariant}\ !x \geq \mathtt{at}(!x, L)\ \ldots\ \} \\ \quad\quad \ldots \\ \quad \mathtt{done} \end{array}$$

Auxiliary variables

Used to denote the **intermediate** values of variables

Example: $\ldots\ \{!x = X\}\ \ldots\ \{!x > X\}\ \ldots$

We will use **labels** instead

- new construct L:e
- new annotation at(t, L)

Example:

$$\begin{array}{l} \quad\vdots \\ L : \texttt{while}\ \ldots\ \texttt{do}\ \{\ \texttt{invariant}\ !x \geq \texttt{at}(!x, L)\ \ldots\ \} \\ \qquad \ldots \\ \quad \texttt{done} \end{array}$$

Auxiliary variables

Used to denote the **intermediate** values of variables

Example: $\ldots\ \{!x = X\}\ \ldots\ \{!x > X\}\ \ldots$

We will use **labels** instead

- new construct $L\texttt{:}e$
- new annotation $\texttt{at}(t, L)$

Example:

$$\begin{array}{l} \quad\vdots \\ L : \texttt{while}\ \ldots\ \texttt{do}\ \{\ \texttt{invariant}\ !x \geq \texttt{at}(!x, L)\ \ldots\ \} \\ \qquad \ldots \\ \quad \texttt{done} \end{array}$$

Functions

A function declaration introduces a **precondition**

- $\texttt{fun}\ (x : \tau) \rightarrow \{p\}\ e$
- $\texttt{rec}\ f\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \beta\ \{\texttt{variant}\ t\} = \{p\}\ e$

Example:

$$\texttt{fun}\ (x : \texttt{int}\ \mathrm{ref}) \rightarrow \{!x > 0\}\ x\ \texttt{:=}\ !x - 1\ \{!x \geq 0\}$$

Functions

A function declaration introduces a **precondition**

- $\texttt{fun}\ (x : \tau) \rightarrow \{p\}\ e$
- $\texttt{rec}\ f\ (x_1 : \tau_1) \dots (x_n : \tau_n) : \beta\ \{\texttt{variant}\ t\} = \{p\}\ e$

Example:

$$\texttt{fun}\ (x : \texttt{int}\ \mathrm{ref}) \rightarrow \{!x > 0\}\ x := !x - 1\ \{!x \geq 0\}$$

Functions

A function declaration introduces a **precondition**

- $\texttt{fun}\ (x : \tau) \rightarrow \{p\}\ e$
- $\texttt{rec}\ f\ (x_1 : \tau_1) \ldots (x_n : \tau_n) : \beta\ \{\texttt{variant}\ t\} = \{p\}\ e$

Example:

$$\texttt{fun}\ (x : \texttt{int}\ \mathsf{ref}) \rightarrow \{\,!x > 0\}\ x \texttt{ := } !x - 1\ \{\,!x \geq 0\}$$

Modularity

A function declaration extends the ML function type with a **precondition**, an **effect** and a **postcondition**

$$f : \; x : \tau_1 \rightarrow \{p\} \, \tau_2 \; \texttt{reads} \, x_1, \ldots, x_n \; \texttt{writes} \, y_1, \ldots, y_m \, \{q\}$$

Example:

$$swap : \; x : \texttt{int} \; \mathsf{ref} \rightarrow y : \texttt{int} \; \mathsf{ref} \rightarrow$$
$$\{\} \, \texttt{unit} \; \texttt{writes} \, x, y \, \{!x = \texttt{old}(!y) \wedge \; !y = \texttt{old}(!x)\}$$

Modularity

A function declaration extends the ML function type with a **precondition**, an **effect** and a **postcondition**

$$f : \; x : \tau_1 \rightarrow \{p\}\, \tau_2 \; \texttt{reads}\, x_1, \ldots, x_n \; \texttt{writes}\, y_1, \ldots, y_m \, \{q\}$$

Example:

$$\begin{array}{l} \mathit{swap} : \; x : \texttt{int}\ \mathsf{ref} \rightarrow y : \texttt{int}\ \mathsf{ref} \rightarrow \\ \qquad\qquad \{\}\, \texttt{unit writes}\ x, y\, \{\texttt{!}x = \texttt{old}(\texttt{!}y) \;\wedge\; \texttt{!}y = \texttt{old}(\texttt{!}x)\} \end{array}$$

Exceptions

Finally, we introduce **exceptions** in our language

- a more realistic ML fragment
- to interpret abrupt statements like `return`, `break` or `continue`

new constructs

- `raise` $(E\ e) : \tau$
- `try` e_1 `with` $E\ x \rightarrow e_2$ `end`

Exceptions

Finally, we introduce **exceptions** in our language

- a more realistic ML fragment
- to interpret abrupt statements like `return`, `break` or `continue`

new constructs

- `raise` $(E\ e) : \tau$
- `try` e_1 `with` $E\ x \rightarrow e_2$ `end`

Exceptions

Finally, we introduce **exceptions** in our language

- a more realistic ML fragment
- to interpret abrupt statements like `return`, `break` or `continue`

new constructs

- `raise` $(E\ e) : \tau$
- `try` e_1 `with` $E\ x \rightarrow e_2$ `end`

Exceptions

The notion of postcondition is extended

$$\texttt{if } x < 0 \texttt{ then raise } \text{Negative} \texttt{ else } \mathit{sqrt}\ x$$
$$\{\ \mathit{result} \geq 0 \mid \text{Negative} \Rightarrow x < 0\ \}$$

So is the notion of effect

$$\mathit{div} :\ x : \texttt{int} \rightarrow y : \texttt{int} \rightarrow \{\ldots\}\, \texttt{int raises } \text{Negative}\, \{\ldots\}$$

Exceptions

The notion of postcondition is extended

$$\texttt{if } x < 0 \texttt{ then raise } \mathsf{Negative} \texttt{ else } \mathit{sqrt}\ x$$
$$\{\ \mathit{result} \geq 0 \mid \mathsf{Negative} \Rightarrow x < 0\ \}$$

So is the notion of effect

$$\mathit{div} :\ x : \texttt{int} \rightarrow y : \texttt{int} \rightarrow \{\ldots\}\, \texttt{int raises } \mathsf{Negative}\, \{\ldots\}$$

Loops and exceptions

We can replace the `while` loop by an **infinite loop**

- `loop` e {`invariant` p `variant` t}

and simulate the `while` loop using an exception

```
while e1 do {invariant p variant t} e2 done  ≡
  try
    loop if e1 then e2 else raise Exit
    {invariant p variant t}
  with Exit _ -> void end
```

simpler constructs ⇒ simpler typing and proof rules

Loops and exceptions

We can replace the `while` loop by an **infinite loop**

- `loop` e `{invariant` p `variant` t`}`

and simulate the `while` loop using an exception

```
while e1 do {invariant p variant t} e2 done  ≡
  try
    loop if e1 then e2 else raise Exit
    {invariant p variant t}
  with Exit _ -> void end
```

simpler constructs ⇒ simpler typing and proof rules

Loops and exceptions

We can replace the `while` loop by an **infinite loop**

- `loop` e {`invariant` p `variant` t}

and simulate the `while` loop using an exception

```
while e1 do {invariant p variant t} e2 done  ≡
  try
    loop if e1 then e2 else raise Exit
    {invariant p variant t}
  with Exit _ -> void end
```

simpler constructs ⇒ simpler typing and proof rules

Summary

Types

$$
\begin{array}{lcl}
\tau & ::= & \beta \mid \beta \text{ ref} \mid (x : \tau) \rightarrow \kappa \\
\kappa & ::= & \{p\}\, \tau\ \epsilon\, \{q\} \\
q & ::= & p; E \Rightarrow p; \ldots; E \Rightarrow p \\
\epsilon & ::= & \texttt{reads}\ x, \ldots, x\ \texttt{writes}\ x, \ldots, x\ \texttt{raises}\ E, \ldots, E
\end{array}
$$

Annotations

$$
\begin{array}{lcl}
t & ::= & c \mid x \mid !x \mid \phi(t, \ldots, t) \mid \texttt{old}(t) \mid \texttt{at}(t, L) \\
p & ::= & \text{True} \mid \text{False} \mid P(t, \ldots, t) \\
 & \mid & p \Rightarrow p \mid p \wedge p \mid p \vee p \mid \neg\, p \mid \forall x : \beta.p \mid \exists x : \beta.p
\end{array}
$$

Summary

Types

$$
\begin{array}{lcl}
\tau & ::= & \beta \mid \beta \text{ ref} \mid (x : \tau) \rightarrow \kappa \\
\kappa & ::= & \{p\}\, \tau\ \epsilon\, \{q\} \\
q & ::= & p; E \Rightarrow p; \ldots; E \Rightarrow p \\
\epsilon & ::= & \texttt{reads}\ x, \ldots, x\ \texttt{writes}\ x, \ldots, x\ \texttt{raises}\ E, \ldots, E
\end{array}
$$

Annotations

$$
\begin{array}{lcl}
t & ::= & c \mid x \mid \texttt{!}x \mid \phi(t, \ldots, t) \mid \texttt{old}(t) \mid \texttt{at}(t, L) \\
p & ::= & \text{True} \mid \text{False} \mid P(t, \ldots, t) \\
 & \mid & p \Rightarrow p \mid p \wedge p \mid p \vee p \mid \neg\, p \mid \forall x : \beta.p \mid \exists x : \beta.p
\end{array}
$$

Programs

$$
\begin{array}{lcl}
u & ::= & c \mid x \mid \texttt{!}x \mid \phi(u,\dots,u) \\
e & ::= & u \\
 & \mid & x \texttt{ := } e \\
 & \mid & \texttt{let } x \texttt{ = } e \texttt{ in } e \\
 & \mid & \texttt{let } x \texttt{ = ref } e \texttt{ in } e \\
 & \mid & \texttt{if } e \texttt{ then } e \texttt{ else } e \\
 & \mid & \texttt{loop } e \ \{\texttt{invariant } p \texttt{ variant } t\} \\
 & \mid & L\texttt{:}e \\
 & \mid & \texttt{raise } (E\ e) : \tau \\
 & \mid & \texttt{try } e \texttt{ with } E\ x \rightarrow e \texttt{ end} \\
 & \mid & \texttt{assert } \{p\};\ e \\
 & \mid & e\ \{q\} \\
 & \mid & \texttt{fun } (x:\tau) \rightarrow \{p\}\ e \\
 & \mid & \texttt{rec } x\ (x:\tau)\dots(x:\tau) : \beta\ \{\texttt{variant } t\} = \{p\}\ e \\
 & \mid & e\ e
\end{array}
$$

Typing

A typing judgment

$$\Gamma \vdash e : (\tau, \epsilon)$$

Rules given in the notes (page 24)

The main purpose is to **exclude aliases**

In particular, references can't escape their scopes

Typing

A typing judgment

$$\Gamma \vdash e : (\tau, \epsilon)$$

Rules given in the notes (page 24)

The main purpose is to **exclude aliases**
In particular, references can't escape their scopes

Semantics

Call-by-value semantics, with left to right evalutation

Big-step operational semantics given in the notes (page 26)

Proof Rules: Weakest Preconditions

We define the predicate $wp(e, q)$, called the weakest precondition for program e and postcondition q

Property: If $wp(e, q)$ holds, then e terminates and q holds at the end of execution (and all inner annotations are verified)

The converse holds for the fragment without loops and functions

The correctness of an annotated program e is thus $wp(e, \mathrm{True})$

Proof Rules: Weakest Preconditions

We define the predicate $wp(e, q)$, called the weakest precondition for program e and postcondition q

Property: If $wp(e, q)$ holds, then e terminates and q holds at the end of execution (and all inner annotations are verified)

The converse holds for the fragment without loops and functions

The correctness of an annotated program e is thus $wp(e, \mathrm{True})$

Proof Rules: Weakest Preconditions

We define the predicate $wp(e, q)$, called the weakest precondition for program e and postcondition q

Property: If $wp(e, q)$ holds, then e terminates and q holds at the end of execution (and all inner annotations are verified)

The converse holds for the fragment without loops and functions

The correctness of an annotated program e is thus $wp(e, \text{True})$

Proof Rules: Weakest Preconditions

We define the predicate $wp(e, q)$, called the weakest precondition for program e and postcondition q

Property: If $wp(e, q)$ holds, then e terminates and q holds at the end of execution (and all inner annotations are verified)

The converse holds for the fragment without loops and functions

The correctness of an annotated program e is thus $wp(e, \text{True})$

Definition of $wp(e, q)$

We actually define $wp(e, q; r)$ where

- q is the "normal" postcondition
- $r \equiv E_1 \Rightarrow q_1; \ldots; E_n \Rightarrow q_n$ is the set of "exceptional" post.

Basic constructs

$$wp(u, q; r) = q[result \leftarrow u]$$

$$wp(x \texttt{ := } e, q; r) = wp(e, q[result \leftarrow \texttt{void}; !x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = ref } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[!x \leftarrow result]; r)$$

$$wp(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, q; r) =$$
$$wp(e_1, \texttt{if } result \texttt{ then } wp(e_2, q; r) \texttt{ else } wp(e_3, q; r); r)$$

$$wp(L{:}\,e, q; r) = wp(e, q; r)[\mathrm{at}(t, L) \leftarrow t]$$

Basic constructs

$$wp(u, q; r) = q[result \leftarrow u]$$

$$wp(x \texttt{ := } e, q; r) = wp(e, q[result \leftarrow \texttt{void}; \texttt{!}x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = ref } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[\texttt{!}x \leftarrow result]; r)$$

$$\begin{array}{l} wp(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, q; r) = \\ \qquad wp(e_1, \texttt{if } result \texttt{ then } wp(e_2, q; r) \texttt{ else } wp(e_3, q; r); r) \end{array}$$

$$wp(L\texttt{:}e, q; r) = wp(e, q; r)[\mathrm{at}(t, L) \leftarrow t]$$

Basic constructs

$$wp(u, q; r) = q[result \leftarrow u]$$

$$wp(x \texttt{ := } e, q; r) = wp(e, q[result \leftarrow \texttt{void}; \texttt{!}x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = ref } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[\texttt{!}x \leftarrow result]; r)$$

$$\begin{aligned} & wp(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, q; r) = \\ & \quad wp(e_1, \texttt{if } result \texttt{ then } wp(e_2, q; r) \texttt{ else } wp(e_3, q; r); r) \end{aligned}$$

$$wp(L\texttt{:}e, q; r) = wp(e, q; r)[\mathrm{at}(t, L) \leftarrow t]$$

Basic constructs

$$wp(u, q; r) = q[result \leftarrow u]$$

$$wp(x \texttt{ := } e, q; r) = wp(e, q[result \leftarrow \texttt{void}; \texttt{!}x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = ref } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[\texttt{!}x \leftarrow result]; r)$$

$$\begin{array}{l} wp(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, q; r) = \\ \quad wp(e_1, \texttt{if } result \texttt{ then } wp(e_2, q; r) \texttt{ else } wp(e_3, q; r); r) \end{array}$$

$$wp(L\!: e, q; r) = wp(e, q; r)[\mathrm{at}(t, L) \leftarrow t]$$

Basic constructs

$$wp(u, q; r) = q[result \leftarrow u]$$

$$wp(x \texttt{ := } e, q; r) = wp(e, q[result \leftarrow \texttt{void}; \texttt{!}x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = ref } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[\texttt{!}x \leftarrow result]; r)$$

$$\begin{array}{l} wp(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, q; r) = \\ \qquad wp(e_1, \texttt{if } result \texttt{ then } wp(e_2, q; r) \texttt{ else } wp(e_3, q; r); r) \end{array}$$

$$wp(L\!: e, q; r) = wp(e, q; r)[\mathrm{at}(t, L) \leftarrow t]$$

Basic constructs

$$wp(u, q; r) = q[result \leftarrow u]$$

$$wp(x \texttt{ := } e, q; r) = wp(e, q[result \leftarrow \texttt{void}; \texttt{!}x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[x \leftarrow result]; r)$$

$$wp(\texttt{let } x \texttt{ = ref } e_1 \texttt{ in } e_2, q; r) = wp(e_1, wp(e_2, q; r)[\texttt{!}x \leftarrow result]; r)$$

$$wp(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3, q; r) =$$
$$wp(e_1, \texttt{if } result \texttt{ then } wp(e_2, q; r) \texttt{ else } wp(e_3, q; r); r)$$

$$wp(L\texttt{:}e, q; r) = wp(e, q; r)[\texttt{at}(t, L) \leftarrow t]$$

Traditional rules

Assignment of a side-effects free expression

$$wp(x \texttt{ := } u, q) = q[\texttt{!}x \leftarrow u]$$

Exception-free sequence

$$wp(e_1\texttt{ ; }e_2, q) = wp(e_1, wp(e_2, q))$$

Traditional rules

Assignment of a side-effects free expression

$$wp(x \texttt{ := } u, q) = q[\texttt{!}x \leftarrow u]$$

Exception-free sequence

$$wp(e_1 \texttt{;}\ e_2, q) = wp(e_1, wp(e_2, q))$$

Exceptions

$$wp(\texttt{raise}\ (E\ e) : \tau, q; r) = wp(e, r_E; r)$$

$$wp(\texttt{try}\ e_1\ \texttt{with}\ E\ x \rightarrow e_2\ \texttt{end}, q; r) = \\ wp(e_1, q; E \Rightarrow wp(e_2, q; r)[x \leftarrow result]; r)$$

Exceptions

$$wp(\texttt{raise}\ (E\ e) : \tau, q; r) = wp(e, r_E; r)$$

$$\begin{array}{l} wp(\texttt{try}\ e_1\ \texttt{with}\ E\ x \rightarrow e_2\ \texttt{end}, q; r) = \\ \quad wp(e_1, q; E \Rightarrow wp(e_2, q; r)[x \leftarrow \mathit{result}]; r) \end{array}$$

Annotations

$$wp(\texttt{assert } \{p\};\ e, q; r) = p \wedge wp(e, q; r)$$

$$wp(e\ \{q'; r'\}, q; r) = wp(e, q' \wedge q; r' \wedge r)$$

Annotations

$$
\begin{aligned}
wp(\texttt{assert}\ \{p\};\ e, q; r) &= p \wedge wp(e, q; r) \\
wp(e\ \{q'; r'\}, q; r) &= wp(e, q' \wedge q; r' \wedge r)
\end{aligned}
$$

Loops

$$wp(\texttt{loop}\ e\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\}, q; r) = \\ p\ \wedge\ \forall\omega.\ p \Rightarrow wp(L\!:e, p \wedge t < \texttt{at}(t, L); r)$$

where ω = the variables (possibly) modified by e

Usual while loop

$wp(\texttt{while}\ e_1\ \texttt{do}\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\}\ e_2\ \texttt{done}, q; r)$
$= p\ \wedge\ \forall\omega.\ p \Rightarrow$
$wp(L\!:\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ \texttt{raise}\ E, p \wedge t < \texttt{at}(t, L), E \Rightarrow q; r)$
$= p\ \wedge\ \forall\omega.\ p \Rightarrow$
$wp(e_1, \texttt{if}\ \mathit{result}\ \texttt{then}\ wp(e_2, p \wedge t < \texttt{at}(t, L))\ \texttt{else}\ q, r)[\texttt{at}(x, L) \leftarrow x]$

Loops

$$wp(\texttt{loop}\ e\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\}, q; r) =$$
$$p\ \wedge\ \forall\omega.\ p \Rightarrow wp(L{:}e, p \wedge t < \texttt{at}(t, L); r)$$

where ω = the variables (possibly) modified by e

Usual while loop

$$wp(\texttt{while}\ e_1\ \texttt{do}\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\}\ e_2\ \texttt{done}, q; r)$$
$$= p\ \wedge\ \forall\omega.\ p \Rightarrow$$
$$wp(L{:}\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else raise}\ E, p \wedge t < \texttt{at}(t, L), E \Rightarrow q; r)$$
$$= p\ \wedge\ \forall\omega.\ p \Rightarrow$$
$$wp(e_1, \texttt{if}\ \mathit{result}\ \texttt{then}\ wp(e_2, p \wedge t < \texttt{at}(t, L))\ \texttt{else}\ q, r)[\texttt{at}(x, L) \leftarrow x]$$

Loops

$$wp(\texttt{loop}\ e\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\}, q; r) = \\ p\ \wedge\ \forall\omega.\ p \Rightarrow wp(L\!:e, p \wedge t < \texttt{at}(t, L); r)$$

where ω = the variables (possibly) modified by e

Usual while loop

$$wp(\texttt{while}\ e_1\ \texttt{do}\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\}\ e_2\ \texttt{done}, q; r)$$
$$= p\ \wedge\ \forall\omega.\ p \Rightarrow$$
$$wp(L\!:\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else raise}\ E, p \wedge t < \texttt{at}(t, L), E \Rightarrow q; r)$$
$$= p\ \wedge\ \forall\omega.\ p \Rightarrow$$
$$wp(e_1, \texttt{if}\ \mathit{result}\ \texttt{then}\ wp(e_2, p \wedge t < \texttt{at}(t, L))\ \texttt{else}\ q, r)[\texttt{at}(x, L) \leftarrow x]$$

Loops

$$wp(\texttt{loop}\ e\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\}, q; r) =$$
$$p\ \wedge\ \forall\omega.\ p \Rightarrow wp(L\!: e, p \wedge t < \texttt{at}(t, L); r)$$

where ω = the variables (possibly) modified by e

Usual while loop

$$wp(\texttt{while}\ e_1\ \texttt{do}\ \{\texttt{invariant}\ p\ \texttt{variant}\ t\}\ e_2\ \texttt{done}, q; r)$$
$$= p\ \wedge\ \forall\omega.\ p \Rightarrow$$
$$wp(L\!: \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else raise}\ E, p \wedge t < \texttt{at}(t, L), E \Rightarrow q; r)$$
$$= p\ \wedge\ \forall\omega.\ p \Rightarrow$$
$$wp(e_1, \texttt{if}\ \mathit{result}\ \texttt{then}\ wp(e_2, p \wedge t < \texttt{at}(t, L))\ \texttt{else}\ q, r)[\texttt{at}(x, L) \leftarrow x]$$

Functions

$$wp(\texttt{fun}\ (x:\tau) \rightarrow \{p\}\ e, q; r) \quad = q\ \wedge\ \forall x.\forall \rho. p \Rightarrow wp(e, \mathsf{True})$$

$$wp(\texttt{rec}\ f\ (x_1:\tau_1)\dots(x_n:\tau_n):\tau\ \{\texttt{variant}\ t\} = \{p\}\ e, q; r)$$
$$= q\ \wedge\ \forall x_1.\dots.\forall x_n.\forall \rho. p \Rightarrow wp(L{:}e, \mathsf{True})$$

when computing $wp(L{:}e, \mathsf{True})$, f is assumed to have type

$$(x_1:\tau_1) \rightarrow \cdots \rightarrow (x_n:\tau_n) \rightarrow \{p \wedge t < \mathtt{at}(t, L)\}\ \tau\ \epsilon\ \{q\}$$

Functions

$$wp(\texttt{fun}\ (x:\tau) \rightarrow \{p\}\ e, q; r) \quad = q\ \wedge\ \forall x.\forall \rho.p \Rightarrow wp(e, \mathrm{True})$$

$$wp(\texttt{rec}\ f\ (x_1:\tau_1)\dots(x_n:\tau_n):\tau\ \{\texttt{variant}\ t\} = \{p\}\ e, q; r)$$
$$= q\ \wedge\ \forall x_1.\dots.\forall x_n.\forall \rho.p \Rightarrow wp(L\!:e, \mathrm{True})$$

when computing $wp(L\!:e, \mathrm{True})$, f is assumed to have type

$$(x_1:\tau_1) \rightarrow \cdots \rightarrow (x_n:\tau_n) \rightarrow \{p \wedge t < \texttt{at}(t, L)\}\, \tau\ \epsilon\, \{q\}$$

Function call

Simplified using

$$e_1\ e_2 \equiv \texttt{let}\ x_1\ \texttt{=}\ e_1\ \texttt{in}\ \texttt{let}\ x_2\ \texttt{=}\ e_2\ \texttt{in}\ x_1\ x_2$$

Assuming

$$x_1 \ :\ (x : \tau) \rightarrow \{p'\}\, \tau'\ \epsilon\, \{q'\}$$

we define

$$wp(x_1\ x_2, q) = p'[x \leftarrow x_2]\ \wedge\ \forall \omega.\forall result.(q'[x \leftarrow x_2] \Rightarrow q)[\texttt{old}(t) \leftarrow t]$$

Function call

Simplified using

$$e_1\ e_2 \equiv \texttt{let}\ x_1\ \texttt{=}\ e_1\ \texttt{in}\ \texttt{let}\ x_2\ \texttt{=}\ e_2\ \texttt{in}\ x_1\ x_2$$

Assuming

$$x_1\ :\ (x : \tau) \rightarrow \{p'\}\,\tau'\ \epsilon\,\{q'\}$$

we define

$$wp(x_1\ x_2, q) = p'[x \leftarrow x_2]\ \wedge\ \forall \omega. \forall result.(q'[x \leftarrow x_2] \Rightarrow q)[\texttt{old}(t) \leftarrow t]$$

Outline

1. An intermediate language for program verification
   1. syntax, typing, semantics, proof rules
   2. **the Why tool**
   3. multi-prover approach
2. Verifying C and Java programs
   1. specification languages
   2. models of program execution
3. A challenging case study

The Why Tool

This intermediate language is implemented in the Why tool

input = polymorphic first-order logic declarations + programs

output = logical declarations + goals, in the syntax of the selected prover

Logical Declarations

```
type t

logic zero : t

logic succ : t -> t

logic le : t, t -> prop

axiom a : forall x:t. le(zero,x)

goal g : le(zero, succ(zero))
```

Programs

```
parameter x : int ref

parameter g :
  b:t -> { x>=0 } t writes x { result=succ(b) and x=x@+1 }

let h (a:int) (b:t) =
  { x>=0 }
    if !x = a then x := 0;
    g (succ b)
  { result=succ(succ(b)) }

exception E
exception F of int
```

Usage

it is a compiler:

- `why --coq f.why` to produce a re-editable Coq file `f_why.v`
- `why --simplify f.why` to produce a Simplify script `f_why.sx`
- etc.

the following provers/formats are supported:

- Coq, PVS, Isabelle/HOL, HOL-light, HOL4, Mizar
- Simplify, Ergo, SMT (Yices, CVC3, etc.), CVC-Lite, haRVey, Zenon

there is a graphical user interface, `gwhy`

Example: Dijkstra's Dutch national flag

Goal: to sort an array where elements only have three different values (blue, white and red)

Algorithm

0	b	i	r	n
BLUE	WHITE	...to do...	RED	

$flag(t,\ n)\ \equiv$

$b \leftarrow 0$

$i \leftarrow 0$

$r \leftarrow n$

while $i < r$

case $t[i]$

BLUE : *swap* $t[b]$ *and* $t[i]$; $b \leftarrow b + 1$; $i \leftarrow i + 1$

WHITE : $i \leftarrow i + 1$

RED : $r \leftarrow r - 1$; *swap* $t[r]$ *and* $t[i]$

Correctness proof

we want to prove

- **termination**
- **absence of runtime error** = no array access out of bounds
- **behavioral correctness** = the final array is sorted **and** contains the same elements as the initial array

Modelization

We model

- colors using an **abstract datatype**
- arrays using references containing **functional arrays**

An abstract type for colors

```
type color

logic blue : color
logic white : color
logic red : color

predicate is_color(c:color) = c=blue or c=white or c=red

parameter eq_color  :
  c1:color -> c2:color ->
    {} bool { if result then c1=c2 else c1<>c2 }
```

Functional arrays

```
type color_array

logic acc : color_array, int -> color
logic upd : color_array, int, color -> color_array

axiom acc_upd_eq :
  forall a:color_array. forall i:int. forall c:color.
    acc(upd(a,i,c),i) = c

axiom acc_upd_neq :
  forall a:color_array. forall i,j:int. forall c:color.
    i <> j -> acc(upd(a,j,c),i) = acc(a,i)
```

Array bounds

```
logic length : color_array -> int

axiom length_update :
  forall a:color_array. forall i:int. forall c:color.
    length(upd(a,i,c)) = length(a)

parameter get :
  t:color_array ref -> i:int ->
    { 0<=i<length(t) } color reads t { result=acc(t,i) }

parameter set :
  t:color_array ref -> i:int -> c:color ->
    { 0<=i<length(t) } unit writes t { t=upd(t@,i,c) }
```

Array bounds

```
logic length : color_array -> int

axiom length_update :
  forall a:color_array. forall i:int. forall c:color.
    length(upd(a,i,c)) = length(a)

parameter get :
  t:color_array ref -> i:int ->
    { 0<=i<length(t) } color reads t { result=acc(t,i) }

parameter set :
  t:color_array ref -> i:int -> c:color ->
    { 0<=i<length(t) } unit writes t { t=upd(t@,i,c) }
```

The swap function

```
let swap (t:color_array ref) (i:int) (j:int) =
  { 0 <= i < length(t) and 0 <= j < length(t) }
  let u = get t i in
  set t i (get t j);
  set t j u
  { t = upd(upd(t@,i,acc(t@,j)), j, acc(t@,i)) }
```

5 proofs obligations

- 3 automatically discharged by Why
- 2 left to the user (and automatically discharged by Simplify)

The swap function

```
let swap (t:color_array ref) (i:int) (j:int) =
  { 0 <= i < length(t) and 0 <= j < length(t) }
  let u = get t i in
  set t i (get t j);
  set t j u
  { t = upd(upd(t@,i,acc(t@,j)), j, acc(t@,i)) }
```

5 proofs obligations

- 3 automatically discharged by Why
- 2 left to the user (and automatically discharged by Simplify)

Function code

```
let dutch_flag (t:color_array ref) (n:int) =
  let b = ref 0 in
  let i = ref 0 in
  let r = ref n in
  while !i < !r do
     if eq_color (get t !i) blue then begin
       swap t !b !i;
       b := !b + 1;
       i := !i + 1
     end else if eq_color (get t !i) white then
       i := !i + 1
     else begin
       r := !r - 1;
       swap t !r !i
     end
  done
```

Function specification

```
let dutch_flag (t:color_array ref) (n:int) =
  { 0 <= n and length(t) = n and
    forall k:int. 0 <= k < n -> is_color(acc(t,k)) }
  ⋮
  { (exists b:int. exists r:int.
      monochrome(t,0,b,blue) and
      monochrome(t,b,r,white) and
      monochrome(t,r,n,red))
    and permutation(t,t@,0,n-1) }
```

The monochrome property

```
predicate monochrome(t:color_array,i:int,j:int,c:color) =
  forall k:int. i<=k<j -> acc(t,k)=c
```

The permutation property

```
logic permutation : color_array, color_array, int, int -> prop

axiom permut_refl : forall t:color_array. forall l,r:int.
  permutation(t,t,l,r)

axiom permut_sym : forall t1,t2:color_array. forall l,r:int.
  permutation(t1,t2,l,r) -> permutation(t2,t1,l,r)

axiom permut_trans : forall t1,t2,t3:color_array. forall l,r:i
  permutation(t1,t2,l,r) -> permutation(t2,t3,l,r) ->
  permutation(t1,t3,l,r)

axiom permut_swap : forall t:color_array. forall l,r,i,j:int.
  l <= i <= r -> l <= j <= r ->
  permutation(t, upd(upd(t,i,acc(t,j)), j, acc(t,i)), l, r)
```

The permutation property

```
logic permutation : color_array, color_array, int, int -> prop

axiom permut_refl : forall t:color_array. forall l,r:int.
  permutation(t,t,l,r)

axiom permut_sym : forall t1,t2:color_array. forall l,r:int.
  permutation(t1,t2,l,r) -> permutation(t2,t1,l,r)

axiom permut_trans : forall t1,t2,t3:color_array. forall l,r:i
  permutation(t1,t2,l,r) -> permutation(t2,t3,l,r) ->
  permutation(t1,t3,l,r)

axiom permut_swap : forall t:color_array. forall l,r,i,j:int.
  l <= i <= r -> l <= j <= r ->
  permutation(t, upd(upd(t,i,acc(t,j)), j, acc(t,i)), l, r)
```

Loop invariant

```
⋮
init:
while !i < !r do
   { invariant
       0 <= b <= i and i <= r <= n and
       monochrome(t,0,b,blue) and
       monochrome(t,b,i,white) and
       monochrome(t,r,n,red) and
       length(t) = n and
       (forall k:int. 0 <= k < n -> is_color(acc(t,k))) and
       permutation(t,t@init,0,n-1)
     variant
       r - i }
   ⋮
done
```

Proof obligations

11 proof obligations

- loop invariant holds initially
- loop invariant is preserved and variant decreases (3 cases)
- `swap` precondition (twice)
- array access within bounds (twice)
- postcondition holds at the end of function execution

All automatically discharged by Simplify!

Outline

1. An intermediate language for program verification
   1. syntax, typing, semantics, proof rules
   2. the Why tool
   3. **multi-prover approach**
2. Verifying C and Java programs
   1. specification languages
   2. models of program execution
3. A challenging case study

Discharging the Verification Conditions

we want to use **off-the-shelf provers**, as many as possible

requirements

- first-order logic
- equality and arithmetic
- quantifiers (memory model, user algebraic models)

Provers Currently Supported

automatic decision procedures

- provers *a la* Nelson-Oppen
  - Simplify, Yices, Ergo
  - CVC Lite, CVC3
- resolution-based provers
  - haRVey, rv-sat
- tableaux-based provers
  - Zenon

interactive proof assistants

- Coq, PVS, Isabelle/HOL, HOL4, HOL-light, Mizar

Typing Issues

verification conditions are expressed in polymorphic first-order logic

need to be **translated** to logics with various type systems:

- unsorted logic (Simplify, Zenon)
- simply sorted logic (SMT provers)
- parametric polymorphism (CVC Lite, PVS)
- polymorphic logic (Ergo, Coq, Isabelle/HOL)

Typing Issues

erasing types is unsound

```
type color

logic white,black : color

axiom color: forall c:color. c=white or c=black
```

$$\forall c,\ c = \texttt{white} \lor c = \texttt{black} \ \vdash\ \bot$$

Type Encoding

several type encodings are used

- monomorphization
  - each polymorphic symbol is replace by several monomorphic types
  - may loop
- usual encoding "types-as-predicates"
  - $\forall x, \mathtt{nat}(x) \Rightarrow P(x)$
  - does not combine nicely with most provers
- new encoding with **type-decorated terms**
  Handling Polymorphism in Automated Deduction (CADE 21)

Trust in Prover Results

- some provers apply the de Bruijn principle and thus are **safe**
  - Coq, HOL family
- most provers **have to be trusted**
  - Simplify, Yices
  - PVS, Mizar
- some provers output **proof traces**
  - Ergo, CVC family, Zenon

Provers Collaboration

most of the time, we run the various provers **in parallel**, expecting at least one of them to discharge the VC

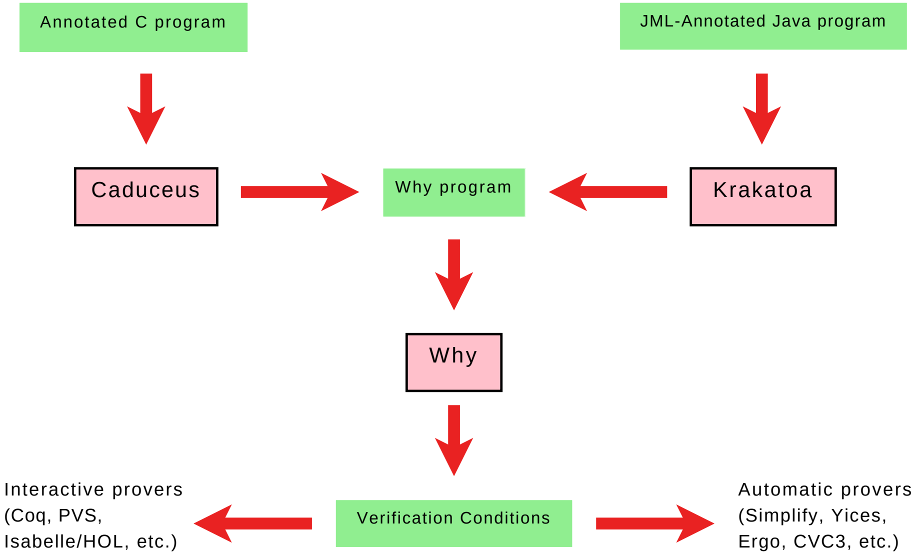
if not, we turn to interactive theorem provers

- no real collaboration between automatic provers
- from Coq or Isabelle, one can call automatic theorem provers
  - proofs are checked when available
  - results are trusted otherwise

part II

Verifying C and Java Programs

Platform Overview



Outline

1. An intermediate language for program verification
   1. syntax, typing, semantics, proof rules
   2. the Why tool
   3. multi-prover approach
2. Verifying C and Java programs
   1. **specification languages**
      - how to formally specify behaviors
   2. models of program execution
3. A challenging case study

Which language to specify behaviors?

Java already has a specification language: **JML** (Java Modeling Language) used in runtime assertion checking tools, ESC/Java, JACK, LOOP, CHASE

JML allows to specify

- precondition, postcondition and side-effects for methods
- invariant and variant for loops
- class invariants
- model fields ($\sim$ ghost code)

Which language to specify behaviors?

we designed a similar language for **C programs**, largely inspired by JML

additional features:

- pointer arithmetic
- algebraic models
  - any axiomatized theory can be used in specifications
  - no runtime assertion checking
- floating-point arithmetic
  - round errors can be specified

A First Example: Binary Search

binary search: search a sorted array of integers for a given value

famous example; see J. Bentley's *Programming Pearls*:
most programmers are wrong on their first attempt to write binary search

Binary Search (code)

```
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1, p = -1;
  while (l <= u ) {
    int m = (l + u) / 2;
    if (t[m] < v)
      l = m + 1;
    else if (t[m] > v)
      u = m - 1;
    else {
      p = m; break;
    }
  }
  return p;
}
```

Binary Search (spec)

we want to prove:

1. absence of runtime error
2. termination
3. behavioral correctness

Binary Search (spec)

```
/*@ requires
  @   n >= 0 &&
  @   \valid_range(t,0,n-1) &&
  @   \forall int k1, int k2;
  @     0 <= k1 <= k2 <= n-1 => t[k1] <= t[k2]
  @*/
int binary_search(int* t, int n, int v) {
  ...
}
```

Binary Search (spec)

```
/*@ requires
  @   n >= 0 &&
  @   \valid_range(t,0,n-1) &&
  @   \forall int k1, int k2;
  @     0 <= k1 <= k2 <= n-1 => t[k1] <= t[k2]
  @ ensures
  @   (\result >= 0 && t[\result] == v) ||
  @   (\result == -1 && \forall int k;
  @                          0 <= k < n => t[k] != v)
  @*/
int binary_search(int* t, int n, int v) {
  ...
}
```

Binary Search (spec)

```
/*@ requires ...
  @ ensures  ...
  @*/
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1, p = -1;
  /*@ variant u-l
    @*/
  while (l <= u ) {
    ...
  }
}
```

Binary Search (spec)

```
/*@ requires ...
  @ ensures  ...
  @*/
int binary_search(int* t, int n, int v) {
  int l = 0, u = n-1, p = -1;
  /*@ invariant
    @   0 <= l && u <= n-1 && p == -1 &&
    @   \forall int k;
    @     0 <= k < n => t[k] == v => l <= k <= u
    @ variant u-l
    @*/
  while (l <= u ) {
    ...
  }
}
```

Binary Search (proof)

DEMO

Algebraic Models

in JML, annotations are written using **pure Java code**
this is mandatory to perform **runtime assertion checking**

but it is often convenient to introduce **axiomatized theories** in order to annotate programs, that is

- abstract types
- function symbols, w or w/o definitions
- predicates, w or w/o definitions
- axioms

Example: Priority Queues

static data structure for a **priority queue** containing integers

```
void clear();       // empties the queue
void push(int x);   // inserts a new element
int  max();         // returns the maximal element
int  pop();         // removes and returns the maximal element
```

Bags

```
//@ type bag

//@ logic bag empty_bag()

//@ logic bag singleton_bag(int x)

//@ logic bag union_bag(bag b1, bag b2)

/*@ logic bag add_bag(int x, bag b)
  @   { union_bag(b, singleton_bag(x)) } */

//@ logic int occ_bag(int x, bag b)

/*@ predicate is_max_bag(bag b, int m) {
  @   occ_bag(m, b) >= 1 &&
  @   \forall int x; occ_bag(x,b) >= 1 => x <= m
  @ } */
```

Bags

```
//@ type bag

//@ logic bag empty_bag()

//@ logic bag singleton_bag(int x)

//@ logic bag union_bag(bag b1, bag b2)

/*@ logic bag add_bag(int x, bag b)
  @   { union_bag(b, singleton_bag(x)) } */

//@ logic int occ_bag(int x, bag b)

/*@ predicate is_max_bag(bag b, int m) {
  @   occ_bag(m, b) >= 1 &&
  @   \forall int x; occ_bag(x,b) >= 1 => x <= m
  @ } */
```

Bags

```
//@ type bag

//@ logic bag empty_bag()

//@ logic bag singleton_bag(int x)

//@ logic bag union_bag(bag b1, bag b2)

/*@ logic bag add_bag(int x, bag b)
  @   { union_bag(b, singleton_bag(x)) } */

//@ logic int occ_bag(int x, bag b)

/*@ predicate is_max_bag(bag b, int m) {
  @   occ_bag(m, b) >= 1 &&
  @   \forall int x; occ_bag(x,b) >= 1 => x <= m
  @ } */
```

Bags

```
//@ type bag

//@ logic bag empty_bag()

//@ logic bag singleton_bag(int x)

//@ logic bag union_bag(bag b1, bag b2)

/*@ logic bag add_bag(int x, bag b)
  @   { union_bag(b, singleton_bag(x)) } */

//@ logic int occ_bag(int x, bag b)

/*@ predicate is_max_bag(bag b, int m) {
  @   occ_bag(m, b) >= 1 &&
  @   \forall int x; occ_bag(x,b) >= 1 => x <= m
  @ } */
```

Bags

```
//@ type bag

//@ logic bag empty_bag()

//@ logic bag singleton_bag(int x)

//@ logic bag union_bag(bag b1, bag b2)

/*@ logic bag add_bag(int x, bag b)
  @   { union_bag(b, singleton_bag(x)) } */

//@ logic int occ_bag(int x, bag b)

/*@ predicate is_max_bag(bag b, int m) {
  @   occ_bag(m, b) >= 1 &&
  @   \forall int x; occ_bag(x,b) >= 1 => x <= m
  @ } */
```

Bags

```
//@ type bag

//@ logic bag empty_bag()

//@ logic bag singleton_bag(int x)

//@ logic bag union_bag(bag b1, bag b2)

/*@ logic bag add_bag(int x, bag b)
  @   { union_bag(b, singleton_bag(x)) } */

//@ logic int occ_bag(int x, bag b)

/*@ predicate is_max_bag(bag b, int m) {
  @   occ_bag(m, b) >= 1 &&
  @   \forall int x; occ_bag(x,b) >= 1 => x <= m
  @ } */
```

Bags

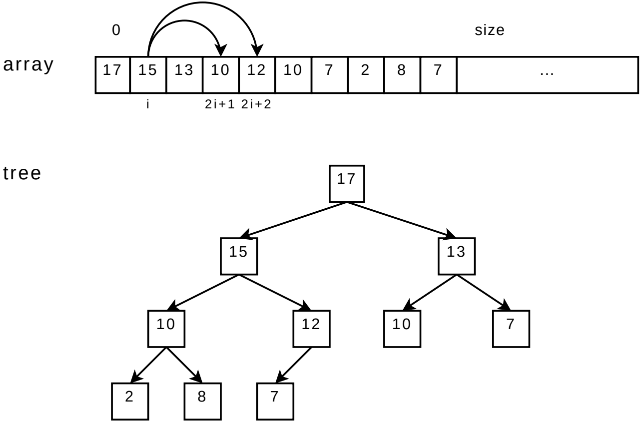
```
//@ type bag

//@ logic bag empty_bag()

//@ logic bag singleton_bag(int x)

//@ logic bag union_bag(bag b1, bag b2)

/*@ logic bag add_bag(int x, bag b)
  @   { union_bag(b, singleton_bag(x)) } */

//@ logic int occ_bag(int x, bag b)

/*@ predicate is_max_bag(bag b, int m) {
  @   occ_bag(m, b) >= 1 &&
  @   \forall int x; occ_bag(x,b) >= 1 => x <= m
  @ } */
```

Priority Queues (spec)

```
//@ logic bag model() { ... }

//@ ensures model() == empty_bag()
void clear();

//@ ensures model() == add_bag(x, \old(model()))
void push(int x);

//@ ensures is_max_bag(model(), \result)
int max();

/*@ ensures is_max_bag(\old(model()), \result) &&
  @         \old(model()) == add_bag(\result, model()) */
int pop();
```

Implementing Priority Queues

implementation: heap encoded in an array



bag { 2, 7, 7, 8, 10, 10, 12, 13, 15, 17 }

Trees

```
//@ type tree

//@ logic tree Empty()

//@ logic tree Node(tree l, int x, tree r)
```

Heaps

```
//@ predicate is_heap(tree t)

//@ axiom is_heap_def_1: is_heap(Empty())

/*@ axiom is_heap_def_2:
  @   \forall int x; is_heap(Node(Empty(), x, Empty()))
  @*/

/*@ axiom is_heap_def_3:
  @   \forall tree ll; \forall int lx;
  @   \forall tree lr; \forall int x;
  @     x >= lx => is_heap(Node(ll, lx, lr)) =>
  @     is_heap(Node(Node(ll, lx, lr), x, Empty()))
  @*/
```

...

Trees and Bags

```
//@ logic bag bag_of_tree(tree t)

/*@ axiom bag_of_tree_def_1:
  @   bag_of_tree(Empty()) == empty_bag()
  @*/

/*@ axiom bag_of_tree_def_2:
  @   \forall tree l; \forall int x; \forall tree r;
  @     bag_of_tree(Node(l, x, r)) ==
  @     add_bag(x, union_bag(bag_of_tree(l), bag_of_tree(r)))
  @*/
```

Trees and Arrays

```
//@ logic tree tree_of_array(int *t, int root, int bound)

/*@ axiom tree_of_array_def_2:
  @   \forall int *t; \forall int root; \forall int bound;
  @     0 <= root < bound =>
  @     tree_of_array(t, root, bound) ==
  @     Node(tree_of_array(t, 2*root+1, bound),
  @          t[root],
  @          tree_of_array(t, 2*root+2, bound))
  @*/
```

...

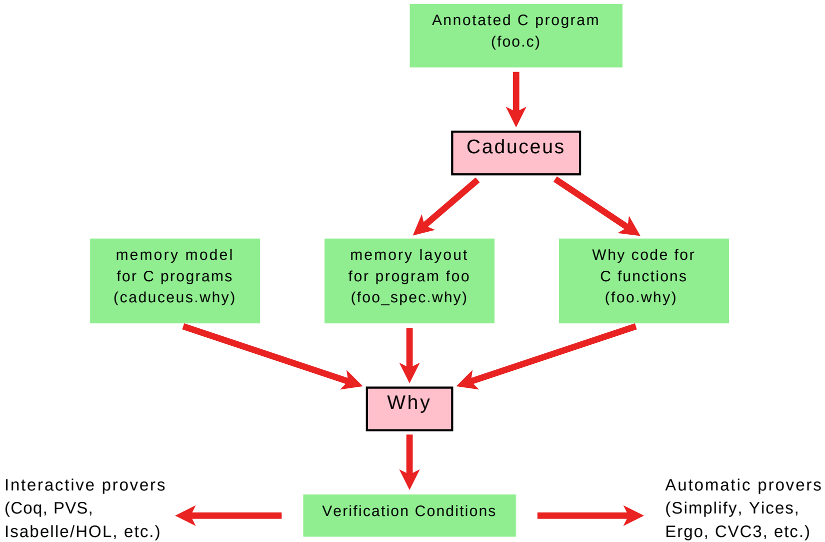
Priority Queues (spec)

```
#define MAXSIZE 100

int heap[MAXSIZE];

int size = 0;

//@ invariant size_inv : 0 <= size < MAXSIZE

//@ invariant is_heap: is_heap(tree_of_array(heap, 0, size))

/*@ logic bag model()
  @   { bag_of_tree(tree_of_array(heap, 0, size)) } */
```

Outline

1. An intermediate language for program verification
   1. syntax, typing, semantics, proof rules
   2. the Why tool
   3. multi-prover approach
2. Verifying C and Java programs
   1. specification languages
   2. **models of program execution**
      - translation of pointer programs to alias-free Why programs
3. A challenging case study

Generating the Verification Conditions



To Pointer Programs to Alias-Free Programs

naive idea: model the **memory as a big array**

using the theory of arrays

$$\texttt{acc} : \texttt{mem}, \texttt{int} \rightarrow \texttt{int}$$
$$\texttt{upd} : \texttt{mem}, \texttt{int}, \texttt{int} \rightarrow \texttt{mem}$$

$$\forall m\, p\, v,\ \texttt{acc}(\texttt{upd}(m, p, v), p) = v$$
$$\forall m\, p_1\, p_2\, v,\ p_1 \neq p_2 \Rightarrow \texttt{acc}(\texttt{upd}(m, p_1, v), p_2) = \texttt{acc}(m, p_2)$$

Naive Memory Model

then the C program

```
int x;
int y;
x = 0;
y = 1;
//@ assert x == 0
```

becomes

$$m := \mathtt{upd}(m, x, 0);$$
$$m := \mathtt{upd}(m, y, 1);$$
$$\mathtt{assert}\ \mathtt{acc}(m, x) = 0$$

the verification condition is

$$\mathtt{acc}(\mathtt{upd}(\mathtt{upd}(m, x, 0), y, 0), x) = 0$$

Memory Model for Pointer Programs

we use the **component-as-array** model (Burstall-Bornat)

each structure/object field is mapped to a different array

relies on the property **“two different fields cannot be aliased”**

strong consequence: prevents pointer casts and unions (a priori)

Benefits of the Component-As-Array Model

```
struct S { int x; int y; } p;
...
p.x = 0;
p.y = 1;
//@ assert p.x == 0
```

becomes

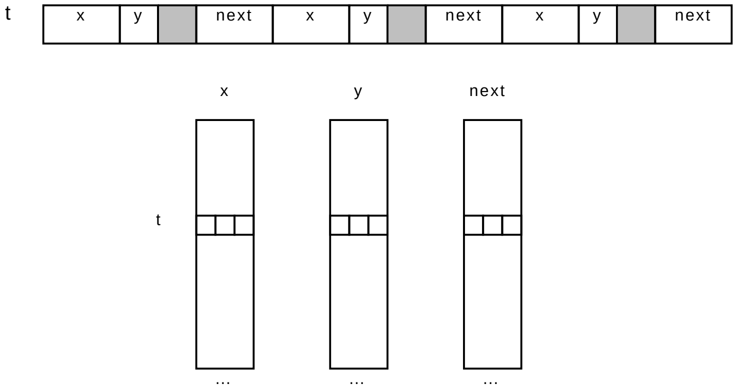
$x := \mathtt{upd}(x, p, 0);$
$y := \mathtt{upd}(y, p, 1);$
$\mathtt{assert}\ \mathtt{acc}(x, p) = 0$

the verification condition is

$$\mathtt{acc}(\mathtt{upd}(x, p, 0), p) = 0$$

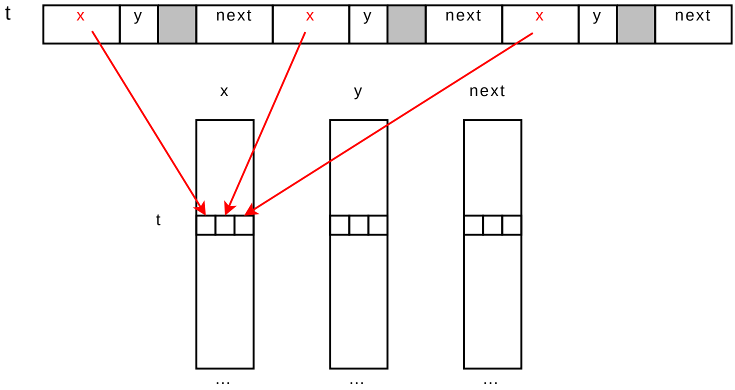
Component-As-Array Model and Pointer Arithmetic

```
struct S { int x; short y; struct S *next; } t[3];
```



Component-As-Array Model and Pointer Arithmetic

```
struct S { int x; short y; struct S *next; } t[3];
```



Separation Analysis

on top of Burstall-Bornat model, we add some **separation analysis**

- each pointer is assigned a **zone**
- zones are **unified** when pointers are assigned / compared
- functions are **polymorphic** wrt zones

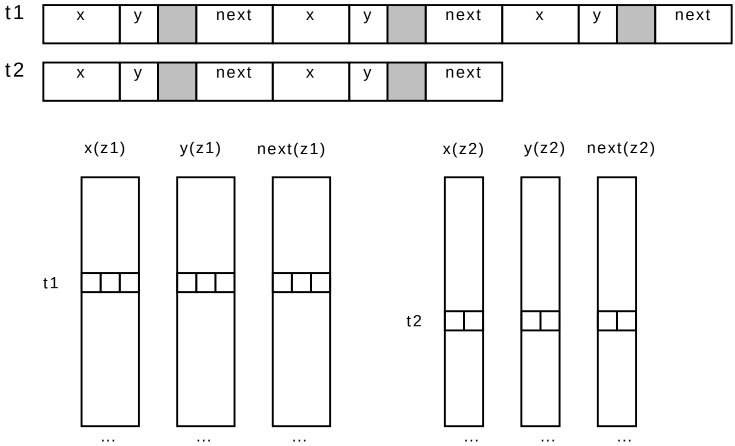
similar to ML-type inference

then the component-as-array model is refined according to zones

Separation Analysis for Deductive Verification (HAV'07)

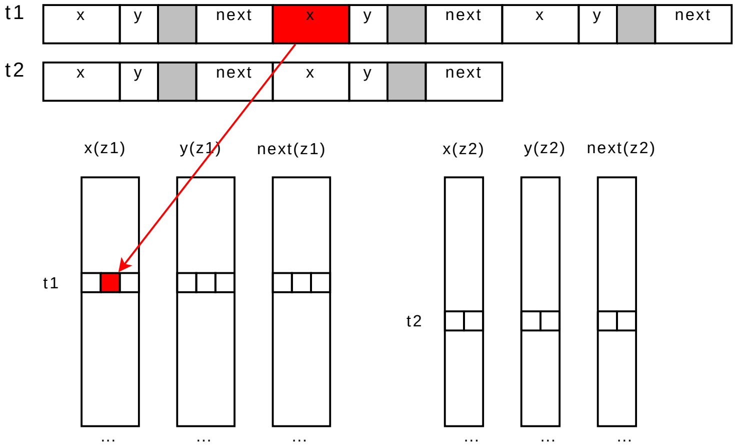
Separation Analysis

```
struct S { int x; short y; struct S *next; } t1[3], t2[2];
```



Separation Analysis

```
struct S { int x; short y; struct S *next; } t1[3], t2[2];
```



Example

little challenge for program verification proposed by P. Müller:

> *count the number n of non-zero values in an integer array t, then copy these values in a freshly allocated array of size n*

`t`	2	1	0	4	0	5	3	0

`count=5`

`u`	2	1	4	5	3

P. Müller's Example (code)

```
void m(int t[], int length) {
  int count=0, i, *u;

  for (i=0 ; i < length; i++)
    if (t[i] > 0) count++;

  u = (int *)calloc(count,sizeof(int));
  count = 0;

  for (i=0 ; i < length; i++)
    if (t[i] > 0) u[count++] = t[i];
}
```

P. Müller's Example (spec)

```
void m(int t[], int length) {
  int count=0, i, *u;
  //@ invariant count == num_of_pos(0,i-1,t) ...
  for (i=0 ; i < length; i++)
    if (t[i] > 0) count++;
  //@ assert count == num_of_pos(0,length-1,t)
  u = (int *)calloc(count,sizeof(int));
  count = 0;
  //@ invariant count == num_of_pos(0,i-1,t) ...
  for (i=0 ; i < length; i++)
    if (t[i] > 0) u[count++] = t[i];
}
```

P. Müller's Example (proof)

12 verification conditions

- without separation analysis: 10/12 automatically proved
- with separation analysis: 12/12 automatically proved

DEMO

Integer Arithmetic

up to now, we did not consider integer arithmetic

there are basically three ways to model arithmetic

- **exact**: all computations are interpreted using mathematical integers; thus it **assumes** that there is no overflow
- **bounded**: the user have to **prove** that there is no integer overflow
- **modulo**: overflows are possible and modulo arithmetic is used; it is **faithful** to machine arithmetic

Overflows in Binary Search

we proved binary search using exact arithmetic

let us prove that there is no overflow

DEMO

Modelling Integer Arithmetic

difficulty: we do not want to lose the ability of provers to handle arithmetic

thus we cannot simply axiomatize machine arithmetic using new abstract data types

Bounded Arithmetic

consider signed 32-bit integers

```
type int32

logic of_int32: int32 -> int

axiom int32_domain:
  forall x:int32.
    -2147483648 <= of_int32(x) <= 2147483647

parameter int32_of_int:
  x:int ->
    { -2147483648 <= x <= 2147483647 }
    int32
    { of_int32(result) = x }
```

Bounded Arithmetic

consider the C fragment

```
(x + 1) * y
```

it is translated into

```
int32_of_int
  ((of_int32
     (int32_of_int
       ((of_int32 x) + (of_int32 (int32_of_int 1)))))
   *
   (of_int32 y))
```

Bounded Arithmetic

in practice, most proof obligations are easy to solve

```
int f(int n) {
  int i = 0;
  while (i < n) {
    ...
    i++;
  }
}
```

we do not even need to insert annotations

Modulo Arithmetic

```
type int32

logic of_int32: int32 -> int

axiom int32_domain:
  forall x:int32. -2147483648 <= of_int32(x) <= 2147483647

logic mod_int32: int -> int

parameter int32_of_int:
  x:int -> { } int32 { of_int32(result) = mod_int32(x) }

axiom mod_int32_id:
  forall x:int.
    -2147483648 <= x <= 2147483647 -> mod_int32(x) = x
...
```

Outline

1. An intermediate language for program verification
   1. syntax, typing, semantics, proof rules
   2. the Why tool
   3. multi-prover approach
2. Verifying C and Java programs
   1. specification languages
   2. models of program execution
3. **A challenging case study**

A challenging case study

challenge for **the verified program of the month**:

```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,(
c+d)/2));return f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0<<q),0,0));}
```

appears on a web page collecting C signature programs

due to Marcel van Kervinck (author of MSCP, a chess program)

Unobfuscating...

```
int t(int a, int b, int c) {
  int d, e=a&~b&~c, f=1;
  if (a)
    for (f=0; d=e&-e; e-=d)
      f += t(a-d, (b+d)*2, (c+d)/2);
  return f;
}

int main(int q) {
  scanf("%d", &q);
  printf("%d\n", t(~(~0<<q), 0, 0));
}
```

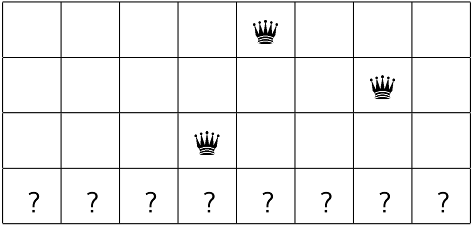
this program reads an integer *n*
and prints the number of solutions to the *n*-queens problem

How does it work?

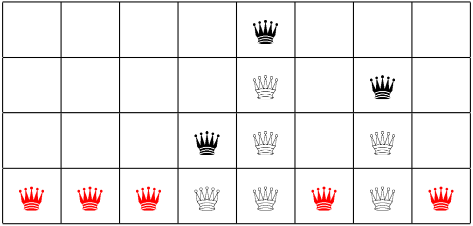
- backtracking algorithm (no better way to solve the n-queens)
- integers used as **sets** (bit vectors)

integers	sets
`0`	$\emptyset$
`a&b`	$a \cap b$
`a+b`	$a \cup b$, when $a \cap b = \emptyset$
`a-b`	$a \setminus b$, when $b \subseteq a$
`~a`	$\complement a$
`a&-a`	$min\_elt(a)$, when $a \neq \emptyset$
`~(~0<<n)`	$\{0, 1, \ldots, n-1\}$
`a*2`	$\{i+1 \mid i \in a\}$, written $S(a)$
`a/2`	$\{i-1 \mid i \in a \wedge i \neq 0\}$, written $P(a)$

What a, b and c mean

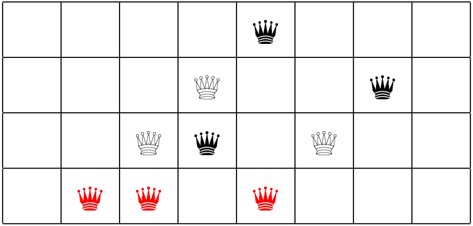


What a, b and c mean



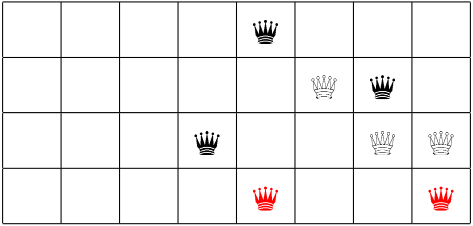
a = columns to be filled = 11100101_2

What a, b and c mean



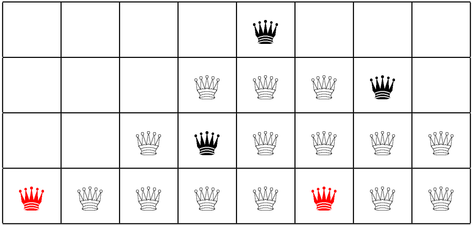
b = positions to avoid because of left diagonals = 01101000_2

What a, b and c mean



c = positions to avoid because of right diagonals = 00001001_2

What a, b and c mean



$a\&\tilde{}b\&\tilde{}c$ = positions to try = 10000100_2

Now it is clear

```
int t(int a, int b, int c) {
  int d, e=a&~b&~c, f=1;
  if (a)
    for (f=0; d=e&-e; e-=d)
      f += t(a-d, (b+d)*2, (c+d)/2);
  return f;
}

int queens(int n) {
  return t(~(~0<<n), 0, 0);
}
```

Abstract finite sets

```
//@ type iset

//@ predicate in_(int x, iset s)

/*@ predicate included(iset a, iset b)
  @ { \forall int i; in_(i,a) => in_(i,b) } */

//@ logic iset empty()

//@ axiom empty_def : \forall int i; !in_(i,empty())
```

...

total: **66 lines** of functions, predicates and axioms

C ints as abstract sets

```
//@ logic iset iset(int x)

/*@ axiom iset_c_zero : \forall int x;
  @   iset(x)==empty() <=> x==0 */

/*@ axiom iset_c_min_elt :
  @   \forall int x; x != 0 =>
  @     iset(x&-x) == singleton(min_elt(iset(x))) */

/*@ axiom iset_c_diff : \forall int a, int b;
  @   iset(a&~b) == diff(iset(a), iset(b)) */
```

...

total: **27 lines**

Termination

```
int t(int a, int b, int c){
  int d, e=a&~b&~c, f=1;
  if (a)
    //@ variant card(iset(e-d))
    for (f=0; d=e&-e; e-=d) {
      f += t(a-d,(b+d)*2,(c+d)/2);
    }
  return f;
}
```

3 verification conditions, all proved automatically

similarly for the termination of the recursive function:
7 verification conditions, all proved automatically

Soundness

how to express that we compute the right number,
since the program is not storing anything,
not even the current solution?

answer: by introducing **ghost code** to perform the missing operations

Ghost code

ghost code can be regarded as regular code, as soon as

- ghost code does not modify program data
- program code does not access ghost data

ghost data is purely logical $\Rightarrow$ ne need to check the validity of pointers

Program instrumented with ghost code

```
//@ int** sol;
//@ int s;
//@ int* col;
//@ int k;

int t(int a, int b, int c) {
  int d, e=a&~b&~c, f=1;
  if (a)
    for (f=0; d=e&-e; e-=d) {
      //@ col[k] = min_elt(d);
      //@ k++;
      f += t3(a-d, (b+d)*2, (c+d)/2);
      //@ k--;
    }
  //@ else
  //@   store_solution();
  return f;
}
```

Program instrumented with ghost code (cont'd)

```
/*@ requires solution(col)
  @ assigns  s, sol[s][0..N()-1]
  @ ensures  s==\old(s)+1 && eq_sol(sol[\old(s)], col)
  @*/
void store_solution();

/*@ requires
  @   n == N() && s == 0 && k == 0
  @ ensures
  @   \result == s &&
  @   sorted(sol, 0, s) &&
  @   \forall int* t; solution(t) <=>
  @       (\exists int i; 0<=i<\result && eq_sol(t,sol[i]))
  @*/
int queens(int n) { return t(~(~0<<n), 0, 0); }
```

Finally, we get...

256 lines of code and specification

regarding VCs:

- main function `queens`: **15** verification conditions
  - **all** proved automatically (Simplify, Ergo or Yices)
- recursive function `t`: **51** verification conditions
  - **42** proved automatically: 41 by Simplify, 37 by Ergo and 35 by Yices
  - **9** proved manually using Coq (and Simplify)

Conclusion

Summary

the Why/Krakatoa/Caduceus platform features

- behavioral specification languages for C and Java programs, at source code level
- deductive program verification using original memory models
- multi-provers backend (interactive and automatic)

free software under GPL license; see http://why.lri.fr/

successfully applied on both

- academic case studies (Schorr-Waite, N-queens, list reversal, etc.)
- industrial case studies (Gemalto, Dassault Aviation, France Telecom)

Other Features

other features not covered in this lecture

- **floating point arithmetic**
  - allows to specify rounding and method errors
  - *Formal Verification of Floating-Point Programs* (ARITH 18)
- **pruning strategies** to help decision procedures on large VCs
  - *A Graph-based Strategy for the Selection of Hypotheses* (FTP 2007)

Ongoing Work & Future Work

ongoing work

- ownership: when class/type invariants must hold?
- C unions & pointer casts

future work

- verification of ML programs