Introduction to the Why tool

Jean-Christophe Filliâtre

CNRS – Université Paris Sud

TYPES summer school – August 25th, 2005

Provers based on HOL are suitable tools to verify purely functional programs (see Tuesday's lecture)

What if you want to verify an imperative program with your favorite prover?

Provers based on HOL are suitable tools to verify purely functional programs (see Tuesday's lecture)

What if you want to verify an imperative program with your favorite prover?

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

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

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

Which programming language?

a realistic existing programming language such as C or Java?

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

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 Why tool

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 Why tool

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

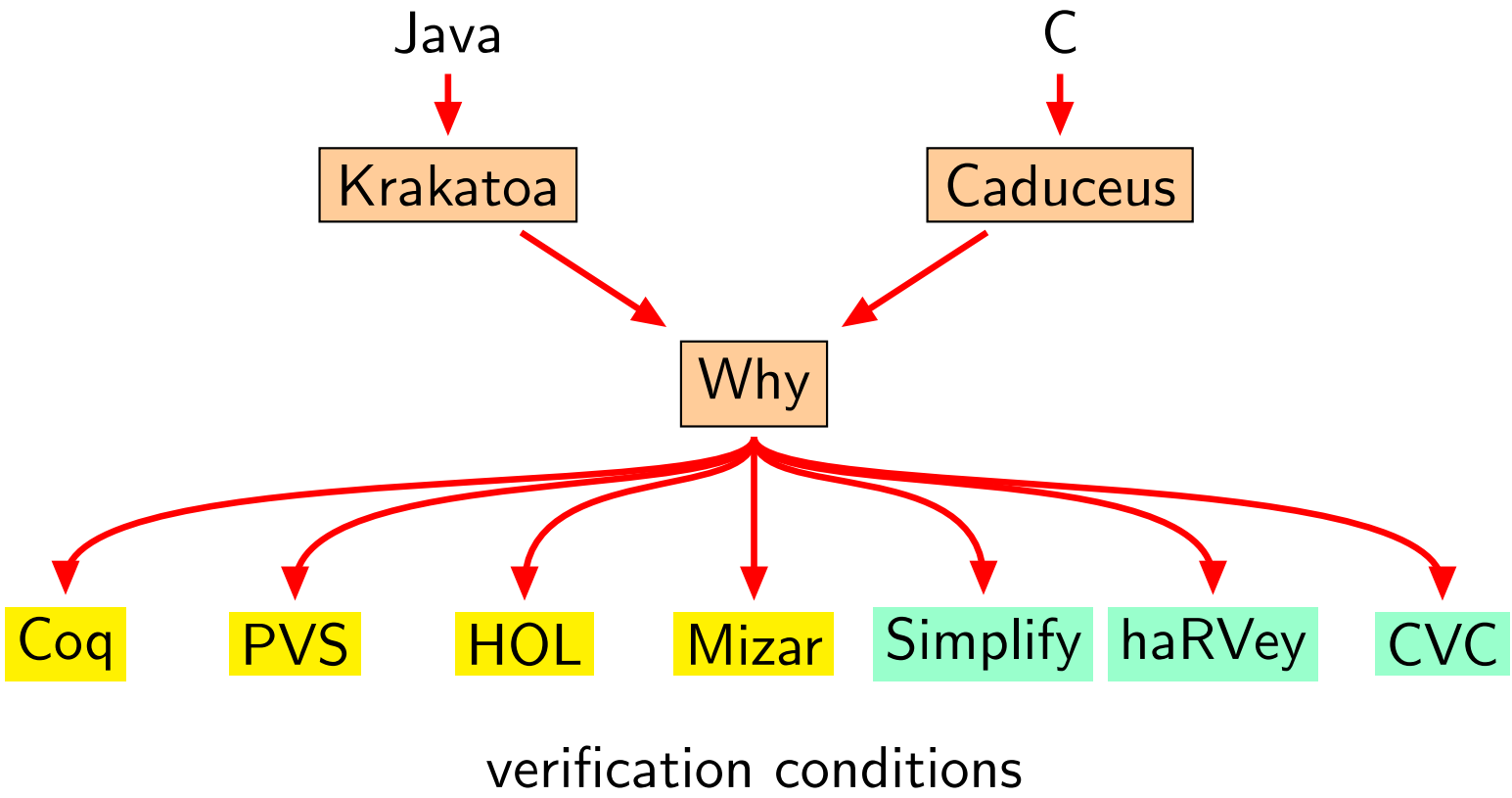
The Why tool

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

An intermediate language



Outline

1. A language for program verification
   - syntax
   - typing
   - semantics
2. Proof rules
3. The WHY tool
   - Dijkstra's Dutch flag
4. Verification of C programs

Outline

1. A language for program verification
   - syntax
   - typing
   - semantics
2. Proof rules
3. The WHY tool
   - Dijkstra's Dutch flag
4. Verification of C programs

Outline

1. A language for program verification
   - syntax
   - typing
   - semantics
2. Proof rules
3. The WHY tool
   - Dijkstra's Dutch flag
4. Verification of C programs

Outline

1. A language for program verification
   - syntax
   - typing
   - semantics
2. Proof rules
3. The WHY tool
   - Dijkstra's Dutch flag
4. Verification of C programs

Part I

A language for program verification

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

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

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

$\Rightarrow$ 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

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

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:

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

Annotations

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

Examples:

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

Annotations

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

Examples:

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

Annotations

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

Examples:

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

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
```

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:

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

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:

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

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:

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

Modularity

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

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

Example:

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

Modularity

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

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

Example:

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

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:

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

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 $\texttt{at}(t, L)$

Example:

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

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:

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

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 $\texttt{at}(t, L)$

Example:

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

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` $E\ x \longrightarrow e_2$ `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` $E\ x \longrightarrow e_2$ `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` $E\ x \longrightarrow e_2$ `end`

Exceptions

The notion of postcondition is extended

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

So is the notion of effect

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

Exceptions

The notion of postcondition is extended

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

So is the notion of effect

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

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 $\Rightarrow$ 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

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

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

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

Programs

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

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)

Part II

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 $wp(e, \mathrm{True})$

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 $wp(e, \mathrm{True})$

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 $wp(e, \mathrm{True})$

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 $wp(e, \mathrm{True})$

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

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

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

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

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

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

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

Basic constructs

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

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

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

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

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

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

Basic constructs

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

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

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

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

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

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

Basic constructs

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

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

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

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

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

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

Basic constructs

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

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

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

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

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

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

Basic constructs

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

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

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

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

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

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

Traditional rules

Assignment of a side-effects free expression

$$wp(x := u, q) = q[x \leftarrow u]$$

Exception-free sequence

$$wp(e_1 ; e_2, q) = wp(e_1, wp(e_2, q))$$

Traditional rules

Assignment of a side-effects free expression

$$wp(x := u, q) = q[x \leftarrow u]$$

Exception-free sequence

$$wp(e_1 ; e_2, q) = wp(e_1, wp(e_2, q))$$

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

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

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

Annotations

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

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

Annotations

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

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

Loops

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

where $\omega =$ 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

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

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

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

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

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

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

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

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

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

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

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

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

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

Function call

Simplified using

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

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

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

Assuming

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

we define

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

Part III

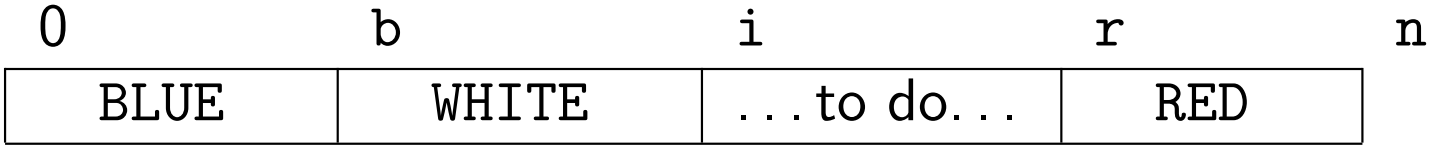
The WHY tool

Dijkstra's Dutch national flag

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

Dijkstra's Dutch national flag

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



A few lines of C code

```
typedef enum { BLUE, WHITE, RED } color;

void swap(int t[], int i, int j) {
  color c = t[i]; t[i] = t[j]; t[j] = c;
}

void flag(int t[], int n) {
  int b = 0, i = 0, r = n;
  while (i < r) {
    switch (t[i]) {
    case BLUE: swap(t, b++, i++); break;
    case WHITE: i++; break;
    case RED: swap(t, --r, i); break;
    }
  }
}
```

Modelization

We are not verifying the C code, but rather the algorithm

We model

- colors with an abstract datetype
- arrays using references containing functional arrays

Modelization

We are not verifying the C code, but rather the algorithm

We model

- colors with an abstract datetype
- 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 :
  ∀a:color_array. ∀i:int. ∀c:color.
    acc(upd(a,i,c),i) = c

axiom acc_upd_neq :
  ∀a:color_array. ∀i,j:int. ∀c:color.
    i ≠ j → acc(upd(a,j,c),i) = acc(a,i)
```

Array bounds

```
logic length : color_array → int

axiom length_update :
  ∀a:color_array. ∀i:int. ∀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 :
  ∀a:color_array. ∀i:int. ∀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

```
predicate monochrome(t:color_array,i:int,j:int,c:color) =
  ∀k:int. i<=k<j → acc(t,k)=c

let dutch_flag (t:color_array ref) (n:int) =
  { 0 <= n and length(t) = n and
    ∀k:int. 0 <= k < n → is_color(acc(t,k)) }
  ⋮
  { ∃b:int. ∃r:int.
      monochrome(t,0,b,blue) and
      monochrome(t,b,r,white) and
      monochrome(t,r,n,red) }
```

Function specification

```
predicate monochrome(t:color_array,i:int,j:int,c:color) =
  ∀k:int. i<=k<j → acc(t,k)=c

let dutch_flag (t:color_array ref) (n:int) =
  { 0 <= n and length(t) = n and
    ∀k:int. 0 <= k < n → is_color(acc(t,k)) }
  ⋮
  { ∃b:int. ∃r:int.
      monochrome(t,0,b,blue) and
      monochrome(t,b,r,white) and
      monochrome(t,r,n,red) }
```

Loop invariant

```
  ⋮
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
               ∀k:int. 0 <= k < n → is_color(acc(t,k))
     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!

Note: to be exhaustive, one has to show that the set of elements was not changed

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!

Note: to be exhaustive, one has to show that the set of elements was not changed

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

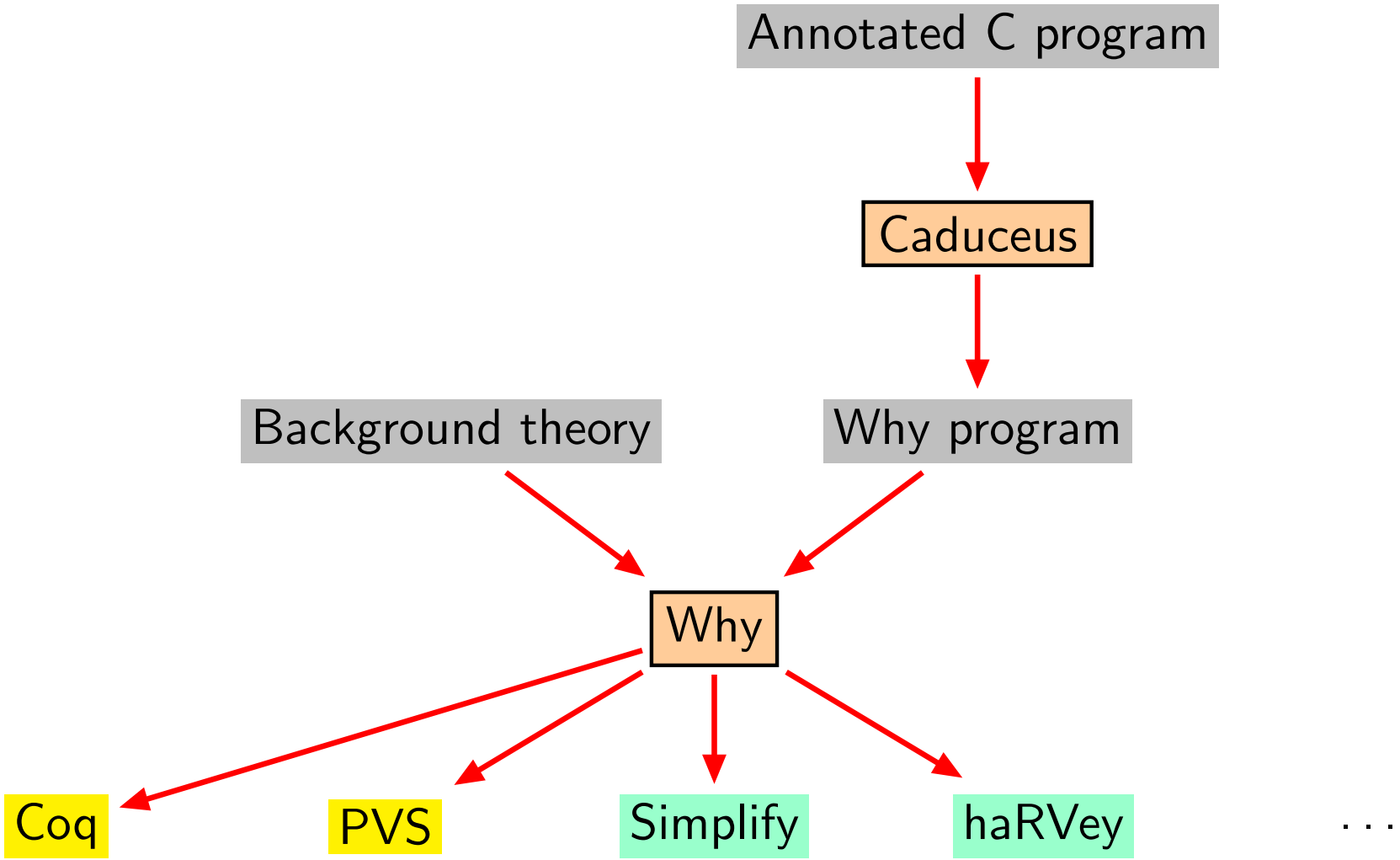
All automatically discharged by Simplify!

Note: to be exhaustive, one has to show that the set of elements was not changed

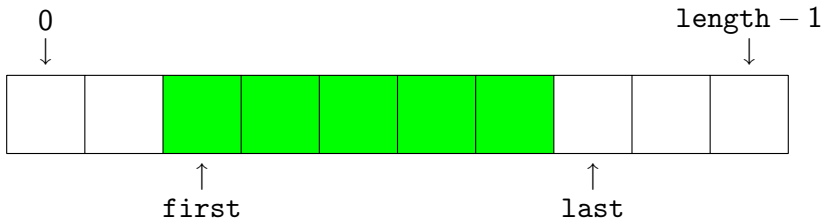
Part IV

Verification of C programs

Overview

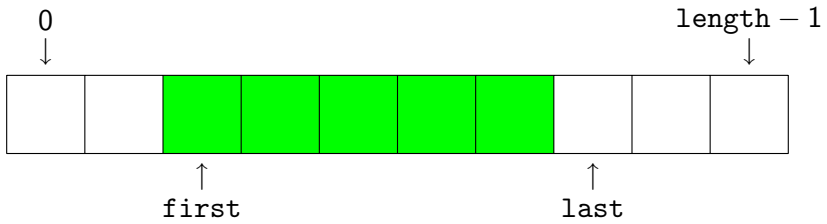

Annotated C program
Caduceus
Background theory
Why program
Why
Coq
PVS
Simplify
haRVey
...

Example: character queue as circular array




```
struct queue {
  char contents[];
  int length;
  int first, last;
  unsigned int empty, full :1;
} q;
/*@ invariant q_invariant :
  @   \valid_range(q.contents, 0, q.length-1) &&
  @   0 <= q.first < q.length &&
  @   0 <= q.last < q.length
  @*/
```

Example: character queue as circular array




```
struct queue {
  char contents[];
  int length;
  int first, last;
  unsigned int empty, full :1;
} q;
/*@ invariant q_invariant :
  @   \valid_range(q.contents, 0, q.length-1) &&
  @   0 <= q.first < q.length &&
  @   0 <= q.last < q.length
  @*/
```

Example continued: specifying functions

```
/*@ requires !q.full
  @ assigns  q.empty, q.full, q.last, q.contents[q.last]
  @ ensures  !q.empty && q.contents[\old(q.last)] == c
  @*/
void push(char c);

/*@ requires !q.empty
  @ assigns q.empty, q.full, q.first
  @ ensures !q.full && \result == q.contents[\old(q.first)]
  @*/
char pop();
```

Example continued: body for push function

```
/*@ requires !q.full
  @ assigns  q.empty, q.full, q.last, q.contents[q.last]
  @ ensures  !q.empty && q.contents[\old(q.last)] == c
  @*/
void push(char c) {
  q.contents[q.last++] = c;      // insert 'c' in the queue
  if (q.last == q.length)
        q.last = 0;              // wrap if needed
  q.empty = 0;                   // queue is not empty
  q.full = (q.first == q.last); // queue is full if
                                 // 'last' reaches 'first'
}
```

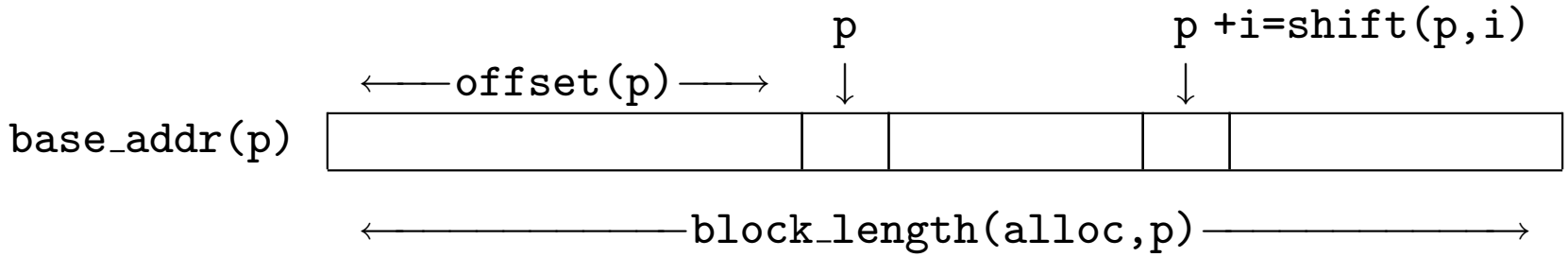
Modeling C memory heap

- Burstall-Bornat model: memory partition according to structure fields
- We extend this idea to handle C arrays and pointer arithmetic: a memory block is

- Each structure field is a map from addresses to memory blocks

Modeling C memory heap

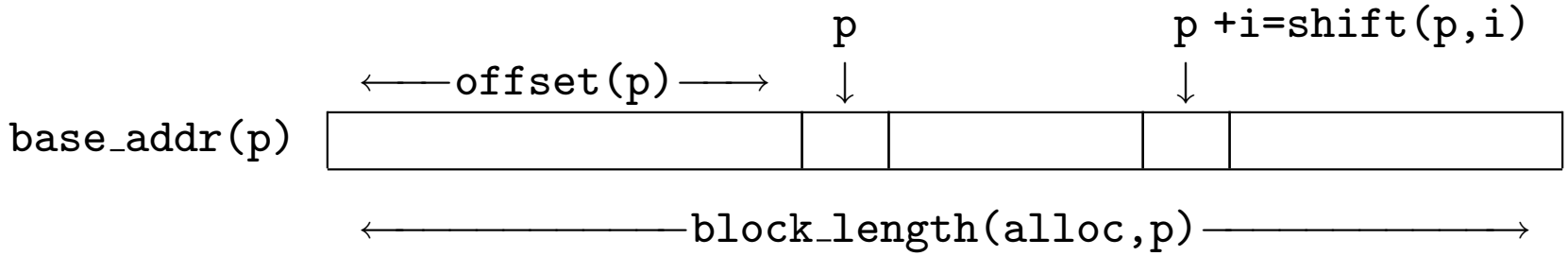
- Burstall-Bornat model: memory partition according to structure fields
- We extend this idea to handle C arrays and pointer arithmetic: a memory block is



- Each structure field is a map from addresses to memory blocks

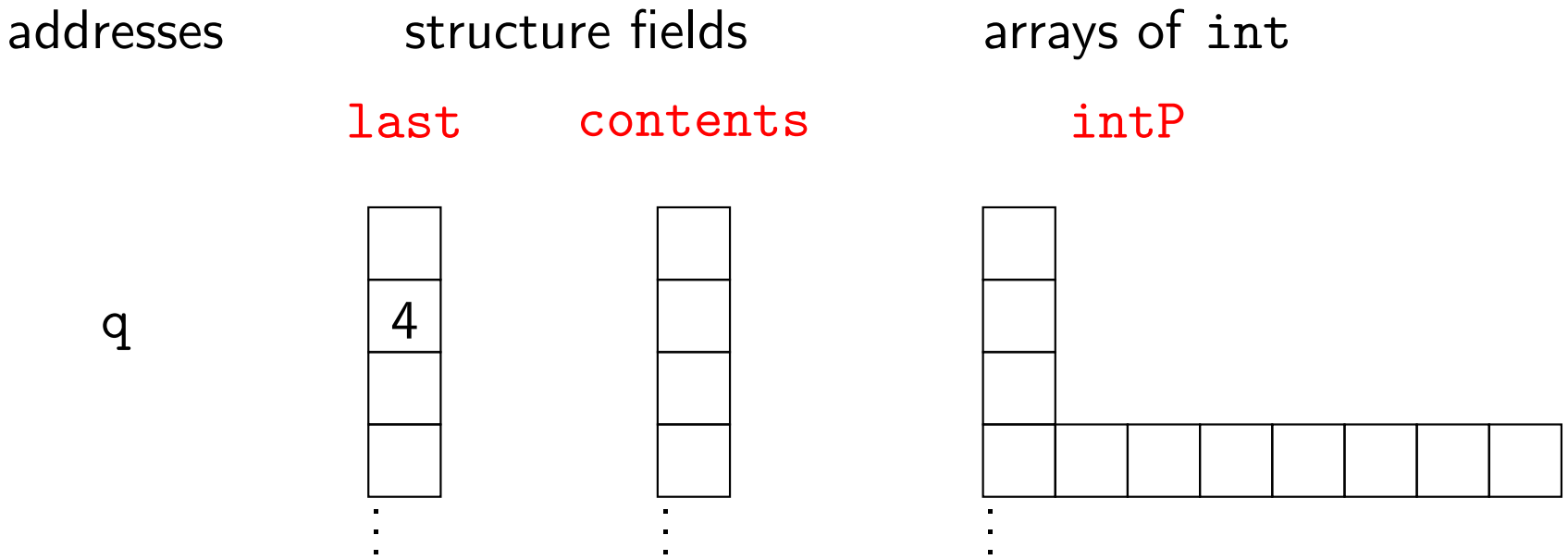
Modeling C memory heap

- Burstall-Bornat model: memory partition according to structure fields
- We extend this idea to handle C arrays and pointer arithmetic: a memory block is



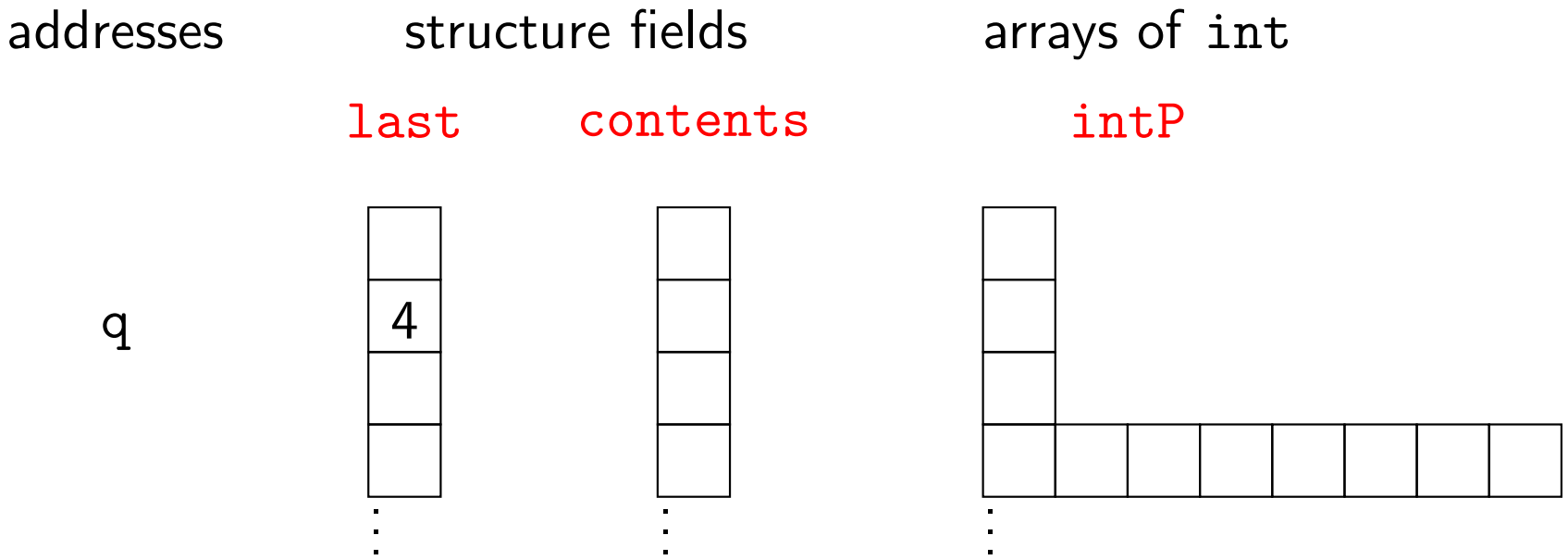
- Each structure field is a map from addresses to memory blocks

Burstall-Bornat model: example of character queue



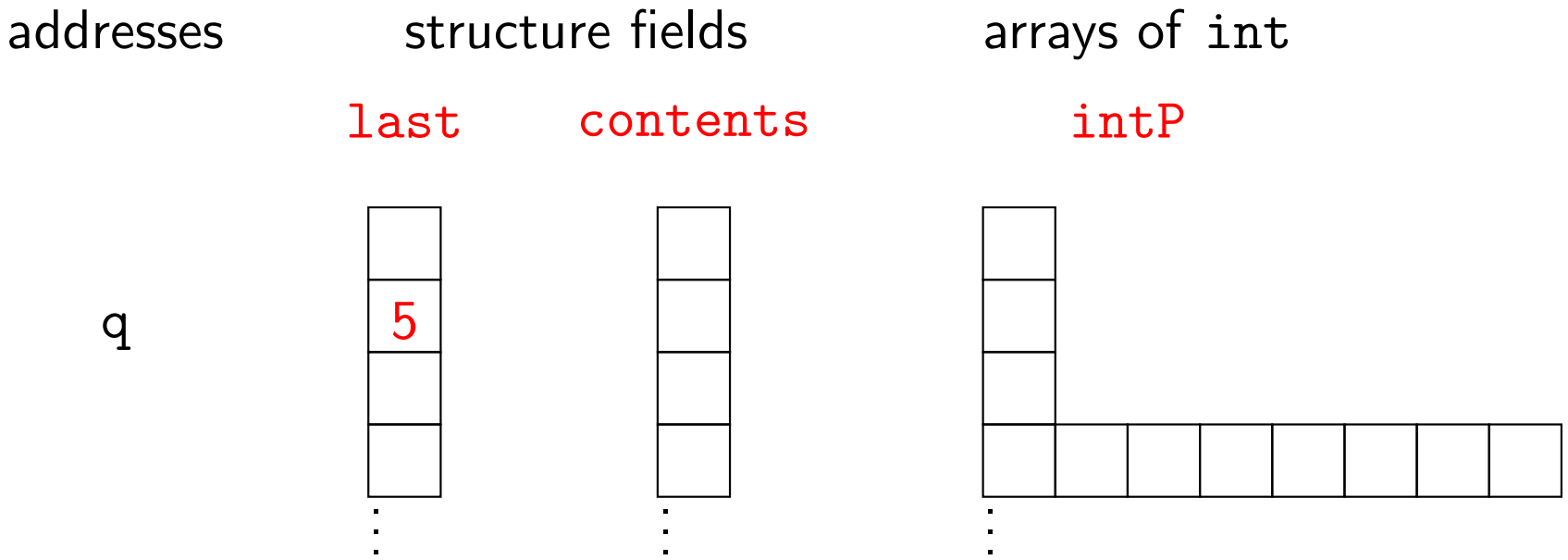

```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

Burstall-Bornat model: example of character queue



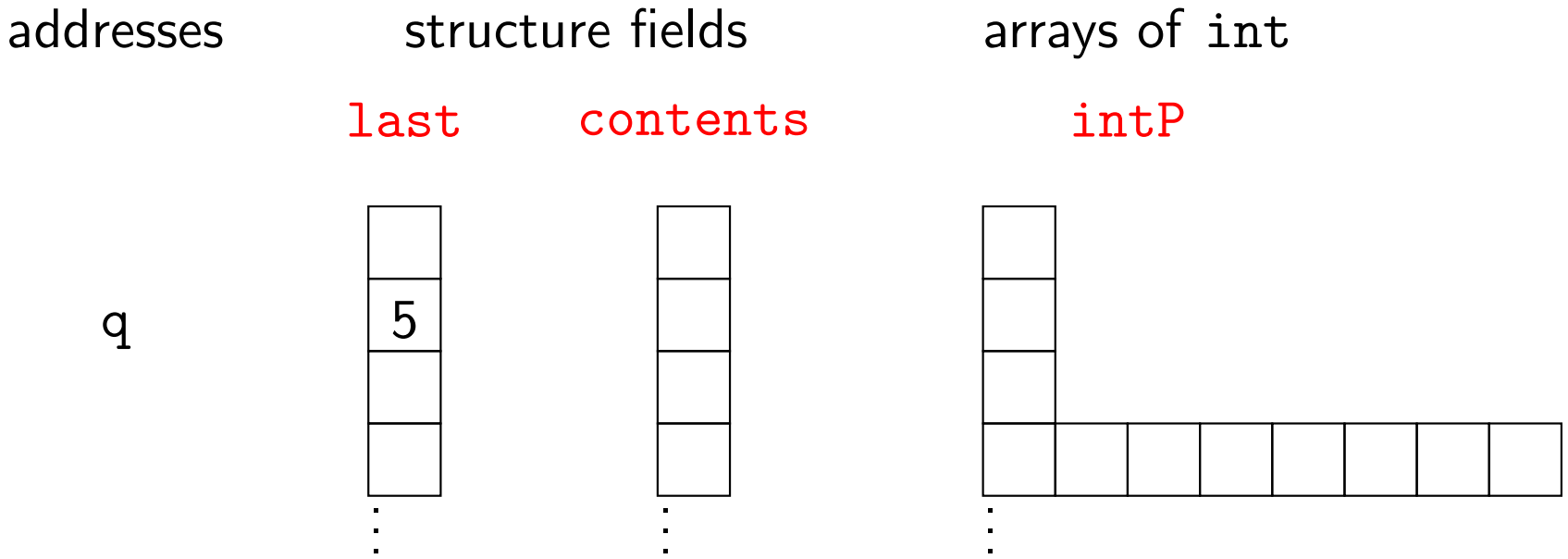

```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

Burstall-Bornat model: example of character queue



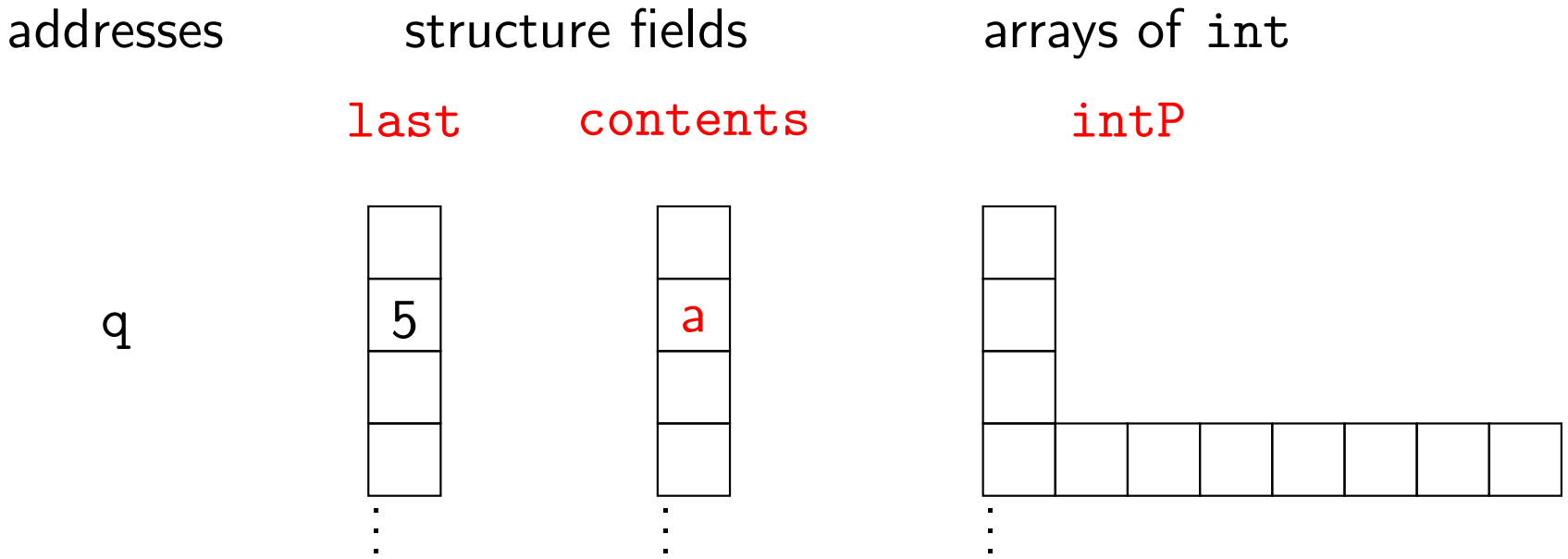

```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

Burstall-Bornat model: example of character queue




```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

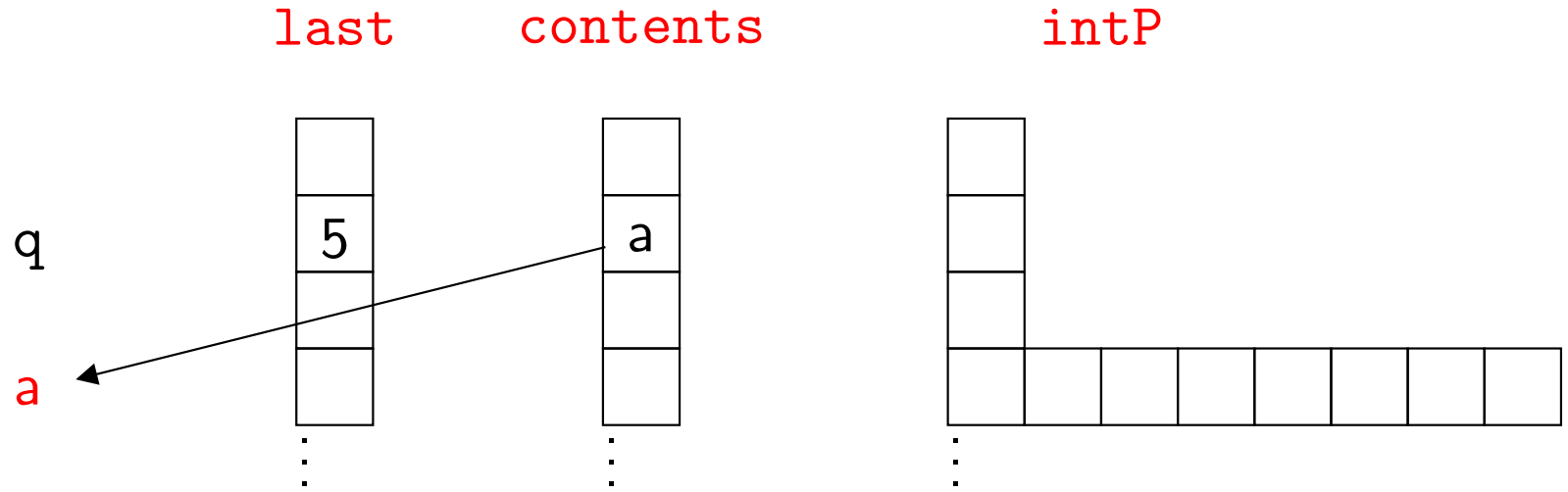
Burstall-Bornat model: example of character queue




```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

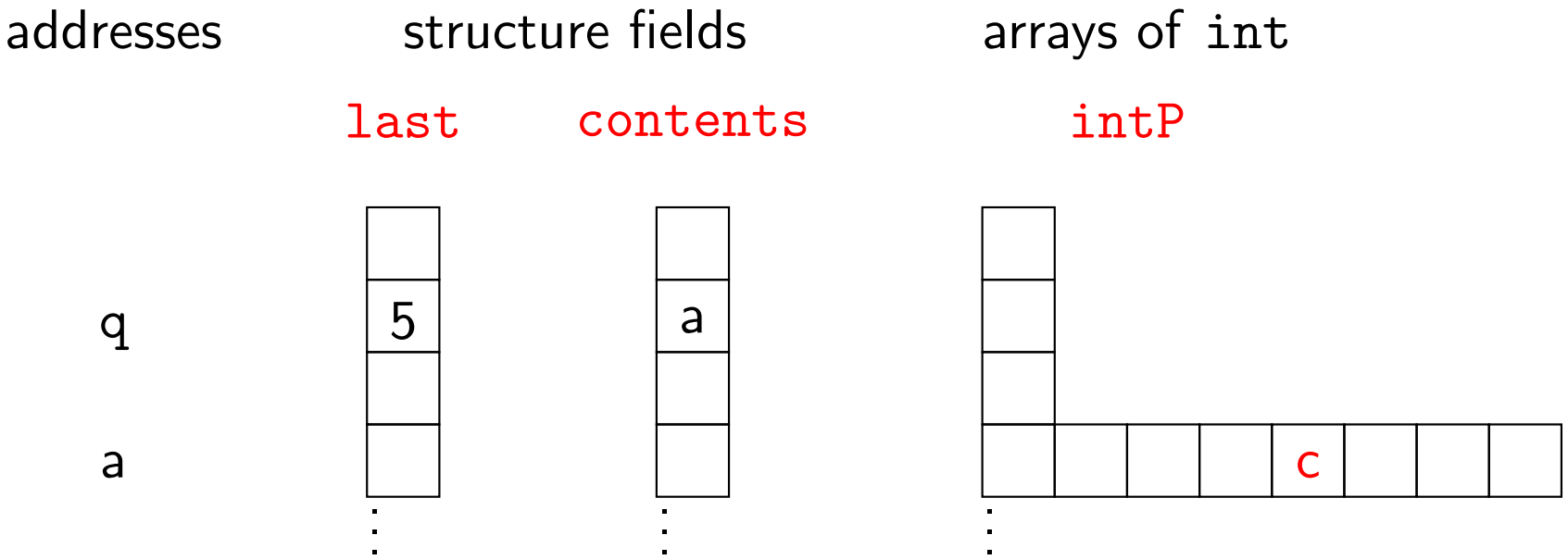
Burstall-Bornat model: example of character queue

addresses structure fields arrays of `int`



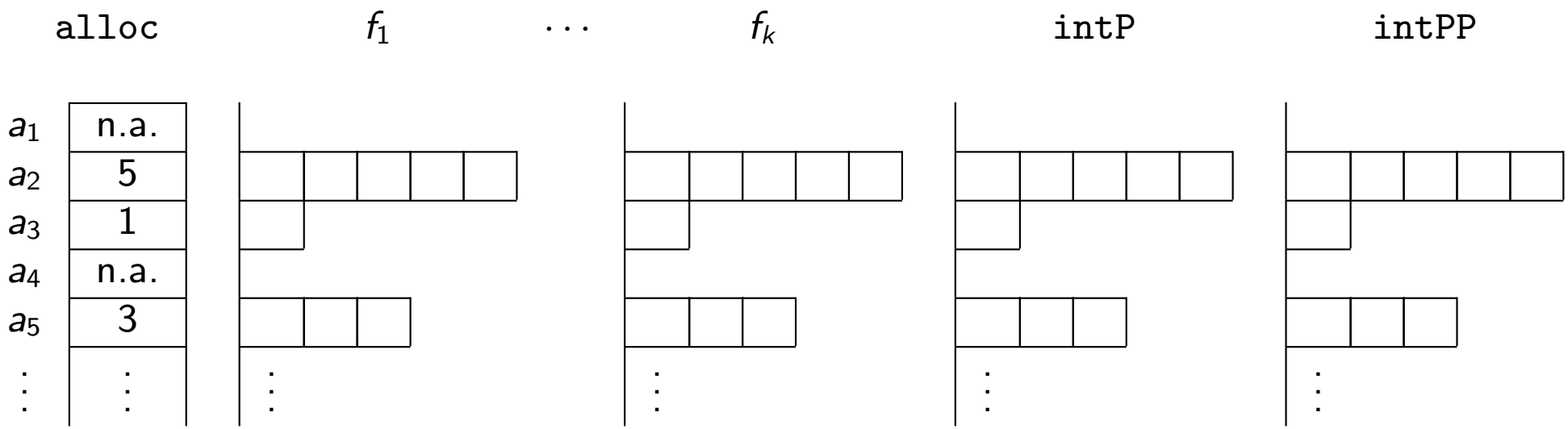

```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

Burstall-Bornat model: example of character queue




```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

General structure of C memory heap



Translation of C statements into Why

The C statement

```
q.contents[q.last++] = c
```

becomes in Why:

```
assert { valid(!alloc,q) };    // proof obligation
let tmp1 = acc(!last,q) in     // tmp1 <- q.last
last := upd(!last,q,tmp1+1);   // q.last <- tmp1+1
let tmp2 = shift(acc(!contents,q),tmp1) in
                               // tmp2 <- q.contents + tmp1
assert { valid(!alloc,tmp2) };// proof obligation
intP := upd(!intP,tmp2,c)      // *tmp2 <- c
```

Axiomatization

- The abstract Why functions `acc`, `upd`, `shift`, etc. are specified by axioms: the background theory
- Excerpt from this theory:

```
acc(upd(t,i,v),i) = v
i <> j -> acc(upd(t,i,v),j) = acc(t,j)
shift(p,0) = p
shift(shift(p,i),j) = shift(p,i+j)
...
```

- An important part of this theory is dedicated to assigns clauses

Example continued: certification of `push` function

Caduceus produces 3 verification conditions expressing that

- the code of `push` contains no unallocated pointer dereference (e.g. assignment of `q.contents[q.last++]` is valid)
- the postcondition and the `assigns` clause of `push` are established
- the invariant `q_invariant` is preserved by `push`

Proofs of these obligations

- with Simplify (100%) and CVC Lite (67%)
- with Coq (100%), very easy (6 lines of tactics)

Example continued: certification of `push` function

Caduceus produces 3 verification conditions expressing that

- the code of `push` contains no unallocated pointer dereference (e.g. assignment of `q.contents[q.last++]` is valid)
- the postcondition and the `assigns` clause of `push` are established
- the invariant `q_invariant` is preserved by `push`

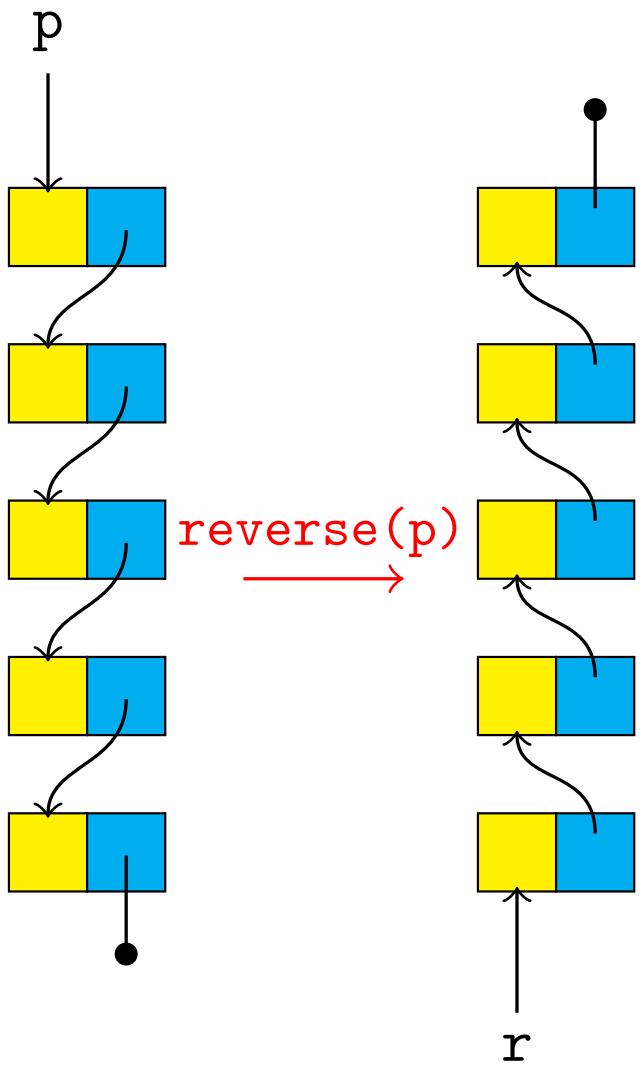
Proofs of these obligations

- with Simplify (100%) and CVC Lite (67%)
- with Coq (100%), very easy (6 lines of tactics)

Example: in-place list reversal

```
typedef struct struct_list {
  int hd;
  struct struct_list *tl;
} *list;

list reverse(list p) {
  list r = NULL;
  while (p != NULL) {
    list q = p ;
    p = p->tl;
    q->tl = r;
    r = q;
  }
  return r;
}
```



Introduction of new logical types and functions

- New predicates and functions can be introduced

```
// logical finite list of pointers
//@ logic plist nil()
//@ logic plist cons(list p, plist l)

// concatenation and reversal
//@ logic plist app(plist l1, plist l2)
//@ logic plist rev(plist pl)
```

- Axioms may be given, e.g.

```
//@ axiom app_nil : \forall plist l; app(nil(),l) == l
```

Introduction of new logical types and functions

- New predicates and functions can be introduced

```
// logical finite list of pointers
//@ logic plist nil()
//@ logic plist cons(list p, plist l)

// concatenation and reversal
//@ logic plist app(plist l1, plist l2)
//@ logic plist rev(plist pl)
```

- Axioms may be given, e.g.

```
//@ axiom app_nil : \forall plist l; app(nil(),l) == l
```

Specification of list reversal

```
/* llist(p,l) specifies that l is the list of pointers
   from p to NULL following tl fields */
//@ predicate llist(list p, plist l) reads p->tl

// is_list(p) specifies that p is finite
//@ predicate is_list(list p) { \exists plist l ; llist(p,l) }

/*@ requires is_list(p)
  @ ensures \forall plist l;
  @    \old(llist(p, l)) => llist(\result, rev(l))  */
list reverse(list p);
```

Specification of list reversal

```
/* llist(p,l) specifies that l is the list of pointers
   from p to NULL following tl fields */
//@ predicate llist(list p, plist l) reads p->tl

// is_list(p) specifies that p is finite
//@ predicate is_list(list p) { \exists plist l ; llist(p,l) }
```

```
/*@ requires is_list(p)
  @ ensures \forall plist l;
  @    \old(llist(p, l)) => llist(\result, rev(l))  */
list reverse(list p);
```

Specification of list reversal

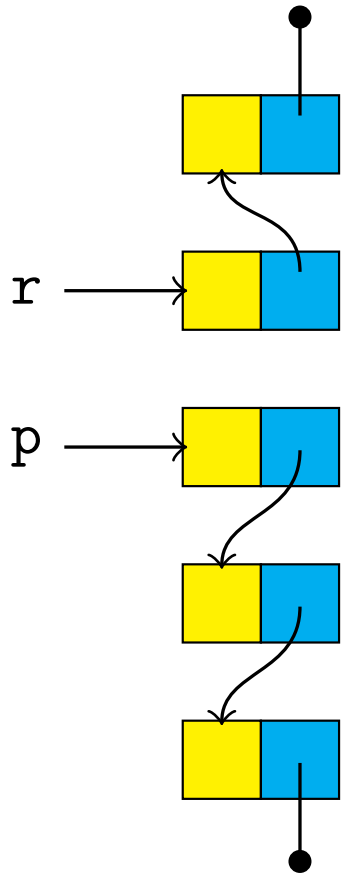
```
/* llist(p,l) specifies that l is the list of pointers
   from p to NULL following tl fields */
//@ predicate llist(list p, plist l) reads p->tl

// is_list(p) specifies that p is finite
//@ predicate is_list(list p) { \exists plist l ; llist(p,l) }

/*@ requires is_list(p)
  @ ensures \forall plist l;
  @    \old(llist(p, l)) => llist(\result, rev(l))  */
list reverse(list p);
```

Annotating the code of list reversal

```
list reverse(list p) {
  list r = NULL;
/*@ invariant
    \exists plist lp; \exists plist lr;
      llist(p, lp) && llist(r, lr) &&
      disjoint(lp, lr) &&
      \forall plist l; \old(llist(p, l)) =>
        app(rev(lp), lr) == rev(l)
  @ variant length(p) for length_order */
  while (p != NULL) {
    list q = p;
    p = p->tl; q->tl = r; r = q;
  }
  return r;
}
```



Certification of list reversal

- 7 verification conditions
- With Simplify: 71%
- With Coq: 100%, with 661 lines of tactics

Example: Schorr-Waite algorithm

- Graph marking algorithm
- Considered as a benchmark for the verification of pointer programs (Bornat, 1999, Jape system) (Nipkow-Mehta, 2003, Isabelle/HOL)
- 12 verification conditions
- With Simplify: 33%
- With Coq: 100%, with 2362 lines of tactics

Example: Schorr-Waite algorithm

- Graph marking algorithm
- Considered as a benchmark for the verification of pointer programs (Bornat, 1999, Jape system) (Nipkow-Mehta, 2003, Isabelle/HOL)
- 12 verification conditions
- With Simplify: 33%
- With Coq: 100%, with 2362 lines of tactics

Part V

Conclusion

Conclusion

- We are able to certify non trivial programs
- We support a large subset of ANSI C and Java/JML
- Tools freely available
  - http://why.lri.fr/
  - http://caduceus.lri.fr/
  - http://krakatoa.lri.fr/

But scaling up issues show up on large programs:

- Generated proof obligations can get large
- Clear need for assistance to write specifications
- Need for more automation of proofs, cooperation of provers

Conclusion

- We are able to certify non trivial programs
- We support a large subset of ANSI C and Java/JML
- Tools freely available
  - http://why.lri.fr/
  - http://caduceus.lri.fr/
  - http://krakatoa.lri.fr/

But scaling up issues show up on large programs:

- Generated proof obligations can get large
- Clear need for assistance to write specifications
- Need for more automation of proofs, cooperation of provers

Current limitations / work in progress

Limitations of the tools

- (mutually) recursive functions
- arithmetic overflow
- floating point arithmetic

Limitations of the C model

- pointer cast
- unions
- non ANSI (i.e. compiler dependent) features

Current limitations / work in progress

Limitations of the tools

- (mutually) recursive functions
- arithmetic overflow
- floating point arithmetic

Limitations of the C model

- pointer cast
- unions
- non ANSI (i.e. compiler dependent) features

Next challenge

Verification of ML programs (with side-effects)

Next challenge

Verification of ML programs (with side-effects)