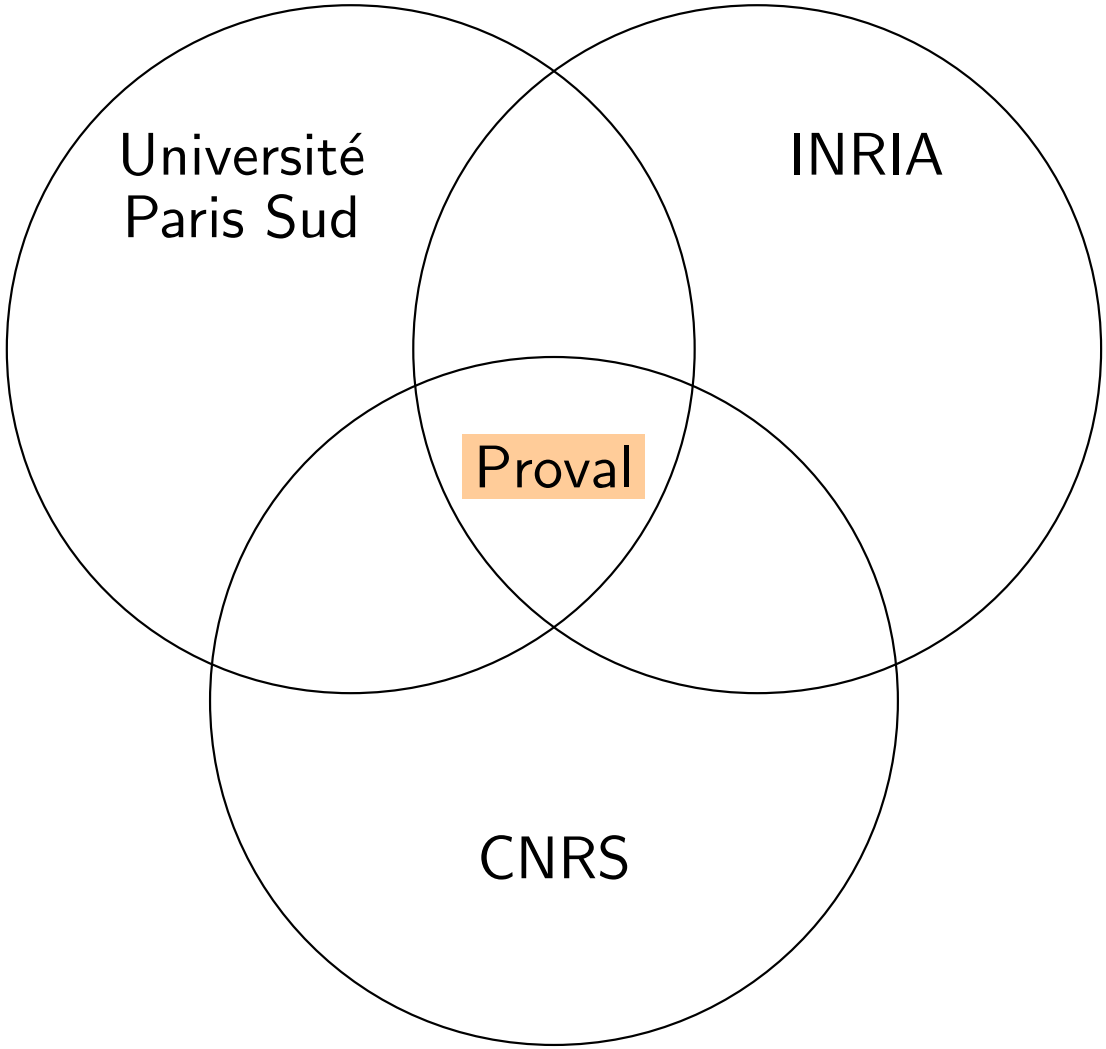
Verifying C and Java programs

Jean-Christophe Filliâtre

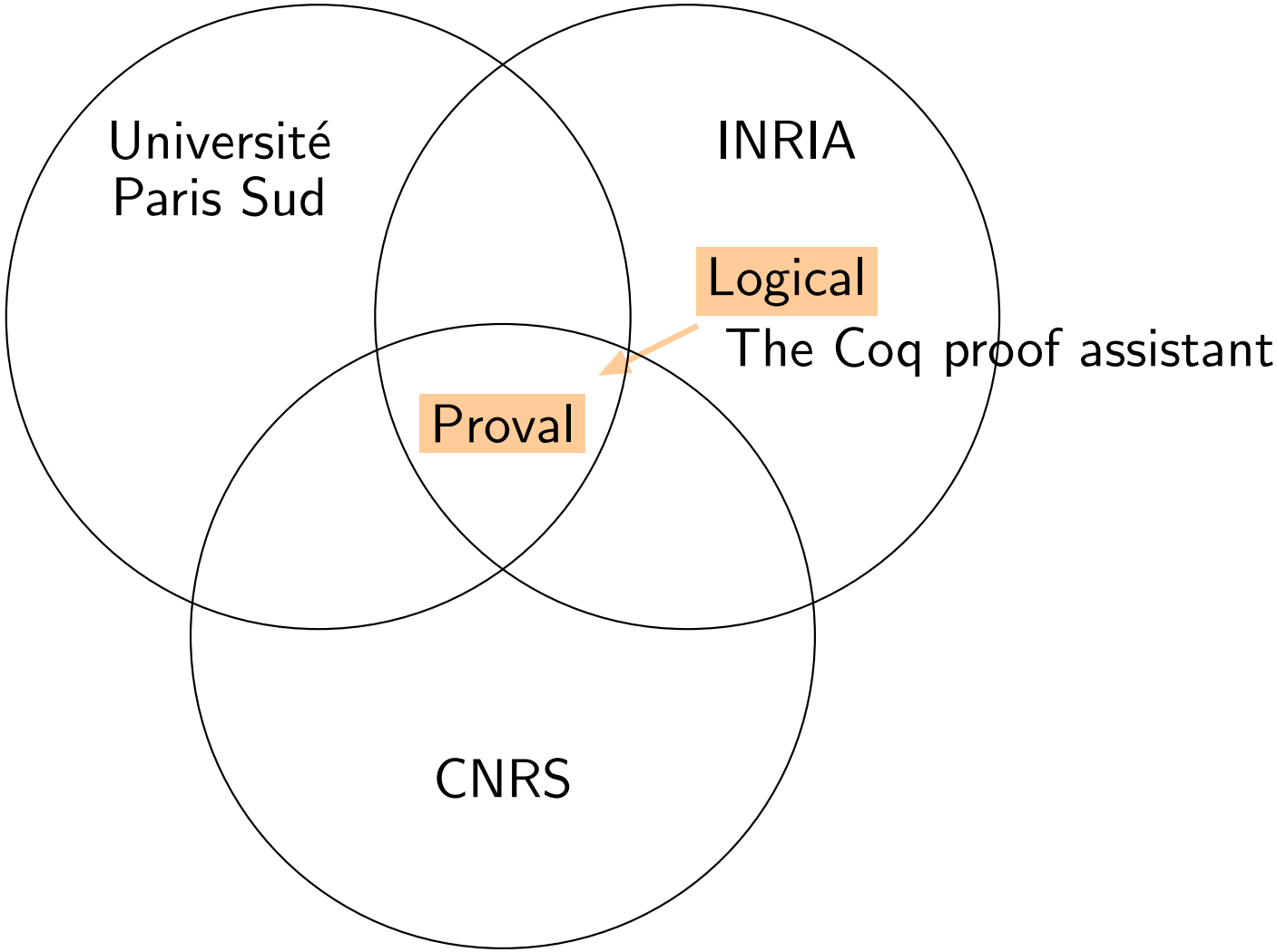
CNRS – Université Paris Sud

Queen Mary University – April 28th, 2005

The Proval project (proval.lri.fr)



The Proval project (proval.lri.fr)



Credo

We are interested in verifying big properties of (small) programs
($\neq$ small properties of big programs e.g. BLAST at Microsoft)

We believe that

- program verification requires interactive proof
  (i.e. fully automatic proof is hopeless)
- verification tools must be safe

Credo

We are interested in verifying big properties of (small) programs
($\neq$ small properties of big programs e.g. BLAST at Microsoft)

We believe that

- program verification requires interactive proof
  (i.e. fully automatic proof is hopeless)
- verification tools must be safe

Credo

We are interested in verifying big properties of (small) programs
($\neq$ small properties of big programs e.g. BLAST at Microsoft)

We believe that

- program verification requires interactive proof
  (i.e. fully automatic proof is hopeless)
- verification tools must be safe

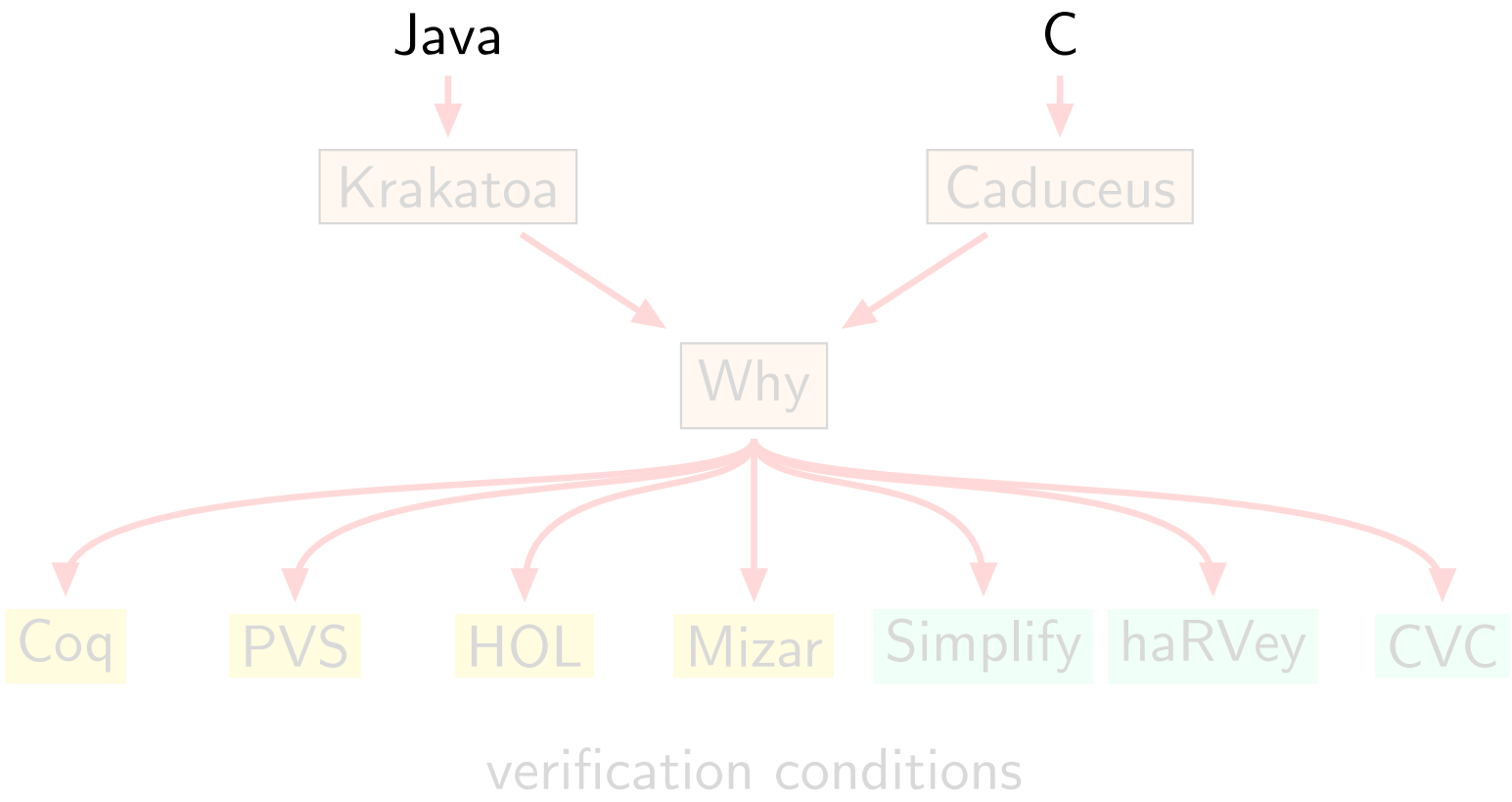
Credo

We are interested in verifying big properties of (small) programs
($\neq$ small properties of big programs e.g. BLAST at Microsoft)

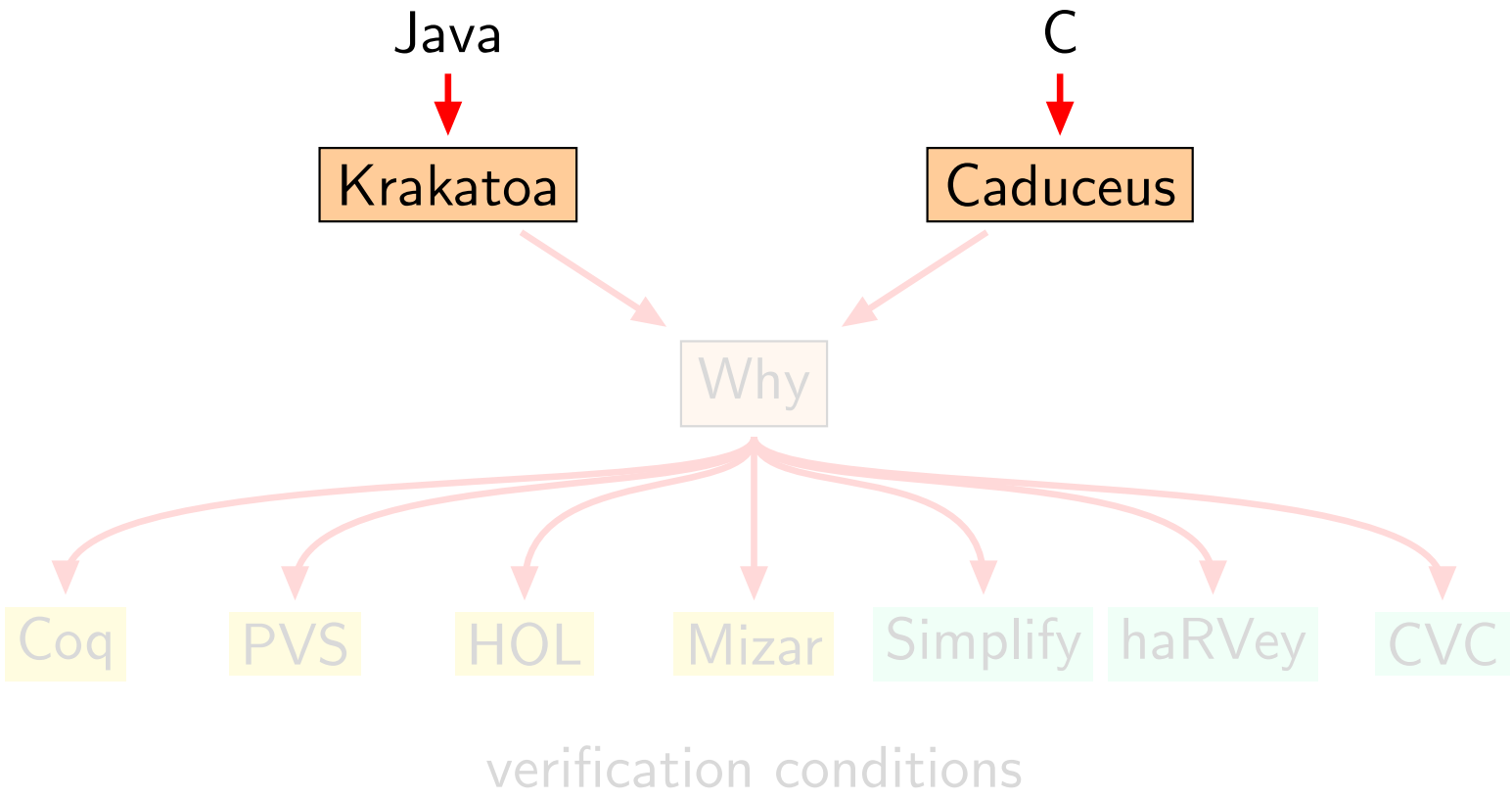
We believe that

- program verification requires interactive proof
  (i.e. fully automatic proof is hopeless)
- verification tools must be safe

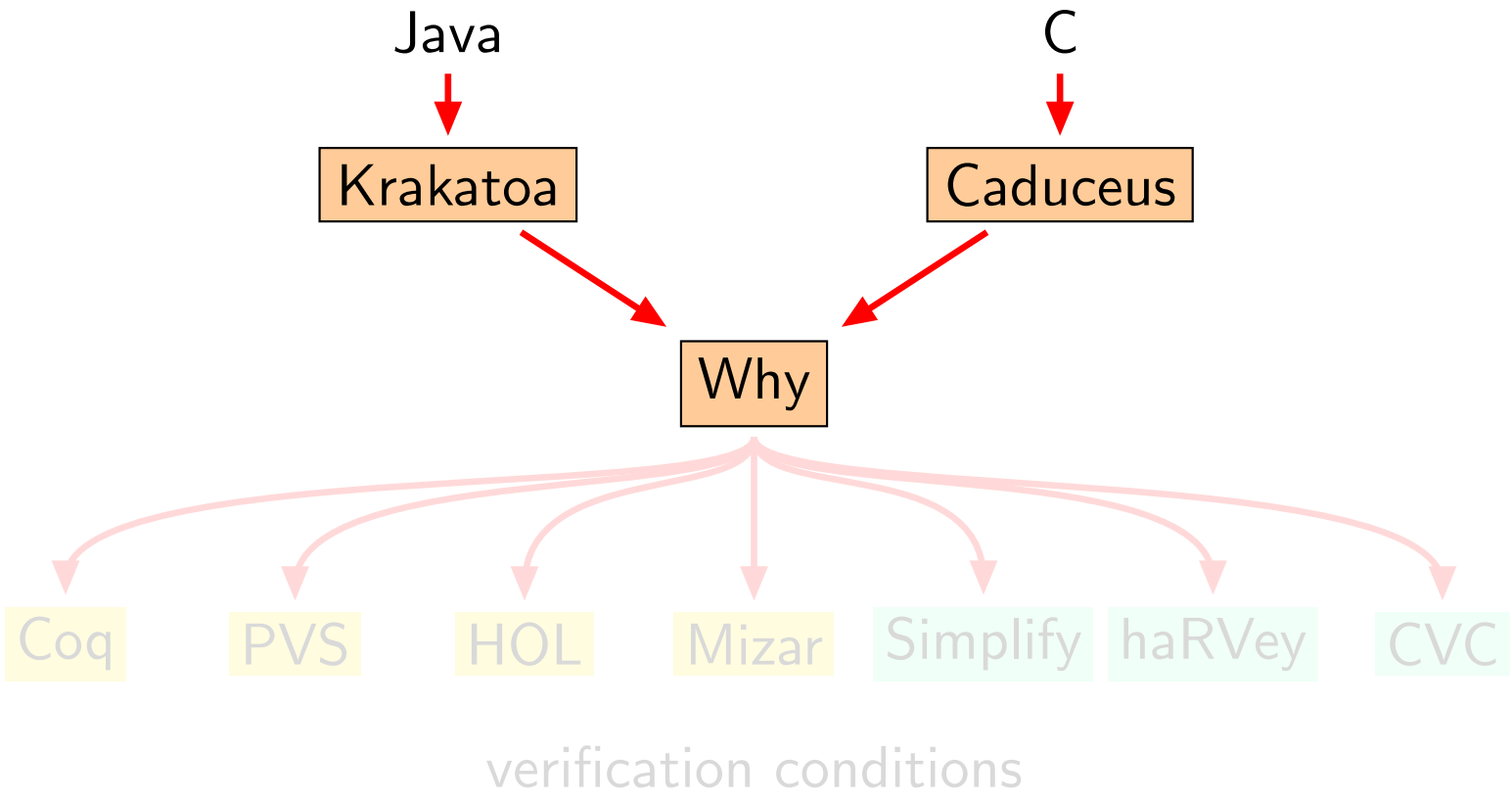
Overview of our tools



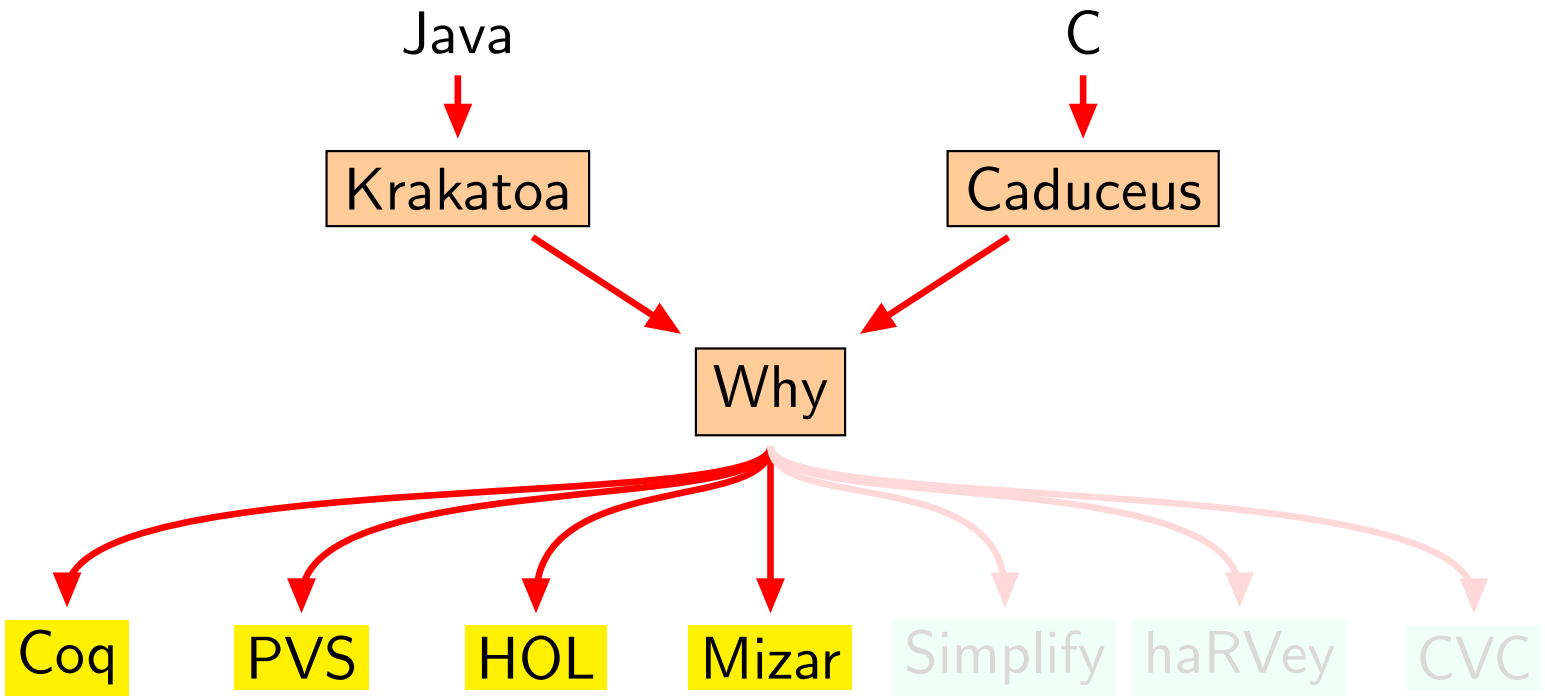
Overview of our tools



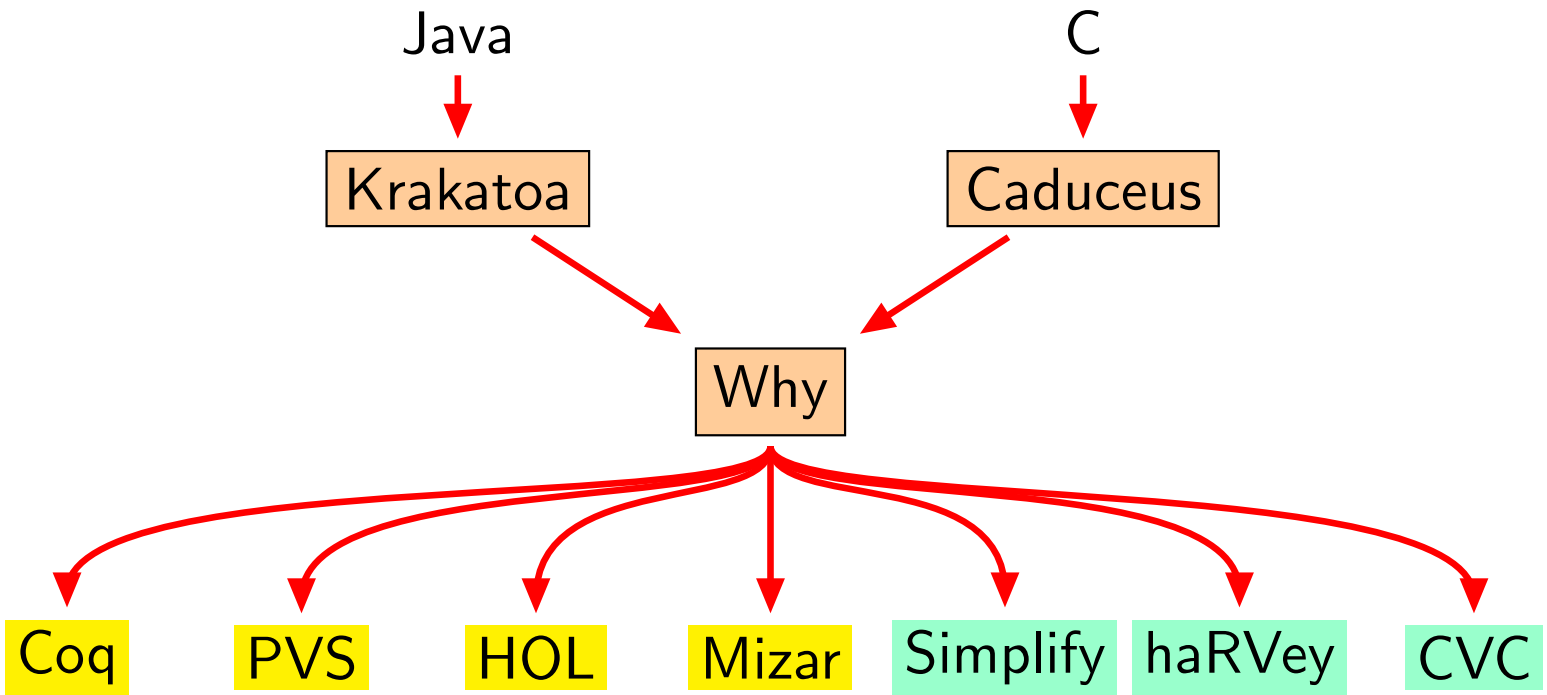
Overview of our tools



Overview of our tools



Overview of our tools



Plan

1. The Why tool
   - Hoare logic revisited
   - Interpretation of imperative programs in Type Theory
   - Monads and effects
2. Verifying C and Java programs
   - Illustration on examples
   - Finding an adequate memory model
3. Perspectives
   - Verification of ML programs

Plan

1. The Why tool
   - Hoare logic revisited
   - Interpretation of imperative programs in Type Theory
   - Monads and effects
2. Verifying C and Java programs
   - Illustration on examples
   - Finding an adequate memory model
3. Perspectives
   - Verification of ML programs

Plan

1. The Why tool
   - Hoare logic revisited
   - Interpretation of imperative programs in Type Theory
   - Monads and effects
2. Verifying C and Java programs
   - Illustration on examples
   - Finding an adequate memory model
3. Perspectives
   - Verification of ML programs

Part I

The Why tool

Capture the essence of Hoare logic

1. absence of aliasing
2. embedding of the logic inside the programming language

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

We look for

- as much genericity as possible
  (w.r.t. input language / logic / back-end provers)
- a safe method (not yet another VCG)

Capture the essence of Hoare logic

1. absence of aliasing
2. embedding of the logic inside the programming language

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

We look for

- as much genericity as possible
  (w.r.t. input language / logic / back-end provers)
- a safe method (not yet another VCG)

Capture the essence of Hoare logic

1. absence of aliasing
2. embedding of the logic inside the programming language

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

We look for

- as much genericity as possible
  (w.r.t. input language / logic / back-end provers)
- a safe method (not yet another VCG)

Key idea

Translate an imperative program into Type Theory using monads

From a program $\{P\}\ p\ \{Q\}$, build a proof

$$\hat{p} : \forall x_1, \ldots, x_n.\ P \Rightarrow \exists y_1, \ldots, y_m.\ Q$$

$\hat{p}$ is of course incomplete: the holes are the proof obligations

Key idea

Translate an imperative program into Type Theory using monads

From a program $\{P\}\ p\ \{Q\}$, build a proof

$$\hat{p} : \forall x_1, \ldots, x_n.\ P \Rightarrow \exists y_1, \ldots, y_m.\ Q$$

$\hat{p}$ is of course incomplete: the holes are the proof obligations

Key idea

Translate an imperative program into Type Theory using monads

From a program $\{P\}\ p\ \{Q\}$, build a proof

$$\hat{p} : \forall x_1, \ldots, x_n.\ P \Rightarrow \exists y_1, \ldots, y_m.\ Q$$

$\hat{p}$ is of course incomplete: the holes are the proof obligations

Example

to establish

$$\{x \geq 0\}\ x\ :=\ !x + 1\ \{x > 0\}$$

we build a proof of

$$\forall x_0.\ x_0 \geq 0 \Rightarrow \exists x_1.\ x_1 > 0$$

which is

$$\lambda x_0.\ \lambda P : x_0 \geq 0.\ \text{let } x_1 = x_0 + 1 \text{ in } (\text{exists } x_1\ (? : x_1 > 0))$$

resulting in one obligation

$$\forall x_0.\ x_0 \geq 0 \Rightarrow \forall x_1.\ x_1 = x_0 + 1 \Rightarrow x_1 > 0$$

Example

to establish

$$\{x \geq 0\}\ x\ :=\ !x + 1\ \{x > 0\}$$

we build a proof of

$$\forall x_0.\ x_0 \geq 0 \Rightarrow \exists x_1.\ x_1 > 0$$

which is

$$\lambda x_0.\ \lambda P : x_0 \geq 0.\ \text{let } x_1 = x_0 + 1 \text{ in } (\text{exists } x_1\ (? : x_1 > 0))$$

resulting in one obligation

$$\forall x_0.\ x_0 \geq 0 \Rightarrow \forall x_1.\ x_1 = x_0 + 1 \Rightarrow x_1 > 0$$

Example

to establish

$$\{x \geq 0\}\ x\ :=\ !x + 1\ \{x > 0\}$$

we build a proof of

$$\forall x_0.\ x_0 \geq 0 \Rightarrow \exists x_1.\ x_1 > 0$$

which is

$$\lambda x_0.\ \lambda P : x_0 \geq 0.\ \text{let } x_1 = x_0 + 1 \text{ in } (\text{exists } x_1\ (? : x_1 > 0))$$

resulting in one obligation

$$\forall x_0.\ x_0 \geq 0 \Rightarrow \forall x_1.\ x_1 = x_0 + 1 \Rightarrow x_1 > 0$$

Example

to establish

$$\{x \geq 0\}\ x\ :=\ !x + 1\ \{x > 0\}$$

we build a proof of

$$\forall x_0.\ x_0 \geq 0 \Rightarrow \exists x_1.\ x_1 > 0$$

which is

$$\lambda x_0.\ \lambda P : x_0 \geq 0.\ \text{let } x_1 = x_0 + 1 \text{ in } (\text{exists } x_1\ (? : x_1 > 0))$$

resulting in one obligation

$$\forall x_0.\ x_0 \geq 0 \Rightarrow \forall x_1.\ x_1 = x_0 + 1 \Rightarrow x_1 > 0$$

Programs

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \mathsf{fun}\ x : \tau \to e \mid e\ e \mid !x \mid x := e \mid \mathsf{ref}\ e \\
& \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{rec}\ f\ (\vec{x} : \vec{\tau}) : \tau = e \\
\\
\tau & ::= & \iota \mid \iota\ \mathsf{ref} \mid \tau \to \tau
\end{array}
$$

derived constructs:

$$
e_1; e_2 \equiv \mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2
$$

$$
\begin{array}{rcl}
\mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{done} & \equiv & (\mathsf{rec}\ w\ (u : \mathsf{unit}) : \mathsf{unit} = \\
& & \quad \mathsf{if}\ e_1\ \mathsf{then}\ (e_2; w\ ())\ \mathsf{else}\ ())\ ()
\end{array}
$$

Programs

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \mathsf{fun}\ x : \tau \to e \mid e\ e \mid !x \mid x := e \mid \mathsf{ref}\ e \\
& \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \mathsf{rec}\ f\ (\vec{x} : \vec{\tau}) : \tau = e \\
\tau & ::= & \iota \mid \iota\ \mathsf{ref} \mid \tau \to \tau
\end{array}
$$

derived constructs:

$$e_1; e_2 \equiv \mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$$

$$\mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{done} \equiv (\mathsf{rec}\ w\ (u : \mathsf{unit}) : \mathsf{unit} = \mathsf{if}\ e_1\ \mathsf{then}\ (e_2; w\ ())\ \mathsf{else}\ ())\ ()$$

Programs

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid \text{fun } x : \tau \to e \mid e\ e \mid !x \mid x := e \mid \text{ref } e \\
& \mid & \text{if } e \text{ then } e \text{ else } e \mid \text{let } x = e \text{ in } e \mid \text{rec } f\ (\vec{x} : \vec{\tau}) : \tau = e \\
\tau & ::= & \iota \mid \iota \text{ ref} \mid \tau \to \tau
\end{array}
$$

derived constructs:

$$
\begin{array}{rcl}
e_1; e_2 & \equiv & \text{let } \_ = e_1 \text{ in } e_2 \\
\text{while } e_1 \text{ do } e_2 \text{ done} & \equiv & (\text{rec } w\ (u : \text{unit}) : \text{unit} = \\
& & \quad \text{if } e_1 \text{ then } (e_2; w\ ()) \text{ else } ())\ ()
\end{array}
$$

Adding annotations

$$
\begin{array}{rcll}
e & ::= & c \mid x \mid \text{fun } x : \tau \to e \mid e\ e \mid !x \mid x := e \mid \text{ref } e & \\
 & \mid & \text{if } e \text{ then } e \text{ else } e \mid \text{let } x = e \text{ in } e \mid \text{rec } f\ (\vec{x} : \vec{\tau}) : \kappa = e & \\
 & \mid & \{P\}\ e\ \{Q\} & \\
\tau & ::= & \iota \mid \iota \text{ ref} \mid x : \tau \to \kappa & \text{type of value} \\
\kappa & ::= & (P, \tau, \epsilon, Q) & \text{type of computation} \\
\epsilon & ::= & \text{reads } \vec{x} \text{ writes } \vec{x} &
\end{array}
$$

the precondition P may mention $!x$

the postcondition Q may mention $!x$, $!\overleftarrow{x}$ and *result*

Adding annotations

$$
\begin{array}{rcll}
e & ::= & c \mid x \mid \text{fun } x : \tau \to e \mid e\ e \mid\ !x \mid x := e \mid \text{ref } e & \\
 & \mid & \text{if } e \text{ then } e \text{ else } e \mid \text{let } x = e \text{ in } e \mid \text{rec } f\ (\vec{x} : \vec{\tau}) : \kappa = e & \\
 & \mid & \{P\}\ e\ \{Q\} & \\
 & & & \\
\tau & ::= & \iota \mid \iota \text{ ref} \mid x : \tau \to \kappa & \text{type of value} \\
\kappa & ::= & (P, \tau, \epsilon, Q) & \text{type of computation} \\
 & & & \\
\epsilon & ::= & \text{reads } \vec{x} \text{ writes } \vec{x} &
\end{array}
$$

the precondition P may mention $!x$
the postcondition Q may mention $!x$, $!\overleftarrow{x}$ and *result*

Typing with effects

In an environment $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ of values we infer the type with effects of a program e

$$\Gamma \vdash e : \kappa$$

In particular, typing

- forbids aliasing
- prevents a local reference to escape its scope

Typing with effects

In an environment $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ of values we infer the type with effects of a program e

$$\Gamma \vdash e : \kappa$$

In particular, typing

- forbids aliasing
- prevents a local reference to escape its scope

Interpretation in CIC

A program e of type $\kappa = (P, \tau, \epsilon, Q)$ where $\epsilon =$ reads $\vec{x}$ writes $\vec{y}$ is interpreted in CIC as

$$\hat{e} : \forall \vec{x}.\ P \Rightarrow \exists \vec{y}, \mathit{result}.\ Q$$

The interpretation makes use of monads parameterized by effects
$\Rightarrow$ the addition of exceptions was straightforward

Interpretation in CIC

A program e of type $\kappa = (P, \tau, \epsilon, Q)$ where $\epsilon =$ reads $\vec{x}$ writes $\vec{y}$ is interpreted in CIC as

$$\hat{e} : \forall \vec{x}.\ P \Rightarrow \exists \vec{y}, \mathit{result}.\ Q$$

The interpretation makes use of monads parameterized by effects
$\Rightarrow$ the addition of exceptions was straightforward

Soundness

We define an operational semantics for our programs (following Wright and Felleisen's *A syntactic approach to type soundness*)
$\theta.e \mapsto^\star \theta'.v$

First we show that the computational part of $\hat{e}$ respects the semantics of e

$$\exists \theta', v.\, \theta.e \mapsto^\star \theta'.v \iff (\overline{e}\,\theta(x_1)\ldots\theta_n(x_n)) = (\theta'(y_1), \ldots, \theta'(y_m), v)$$

Then we show that if all the obligations in $\hat{e}$ can be replaced by proof terms, in such a way that $\hat{e}$ becomes well-typed in CIC, then e satisfies its specification κ.

Soundness

We define an operational semantics for our programs (following Wright and Felleisen's *A syntactic approach to type soundness*)
$\theta.e \mapsto^\star \theta'.v$

First we show that the computational part of $\hat{e}$ respects the semantics of e

$$\exists \theta', v.\ \theta.e \mapsto^\star \theta'.v \iff (\overline{e}\,\theta(x_1)\ldots\theta_n(x_n)) = (\theta'(y_1),\ldots,\theta'(y_m),v)$$

Then we show that if all the obligations in $\hat{e}$ can be replaced by proof terms, in such a way that $\hat{e}$ becomes well-typed in CIC, then e satisfies its specification κ.

Soundness

We define an operational semantics for our programs (following Wright and Felleisen's *A syntactic approach to type soundness*)
$\theta.e \mapsto^\star \theta'.v$

First we show that the computational part of $\hat{e}$ respects the semantics of e

$$\exists \theta', v.\ \theta.e \mapsto^\star \theta'.v \iff (\overline{e}\,\theta(x_1)\ldots\theta_n(x_n)) = (\theta'(y_1),\ldots,\theta'(y_m),v)$$

Then we show that if all the obligations in $\hat{e}$ can be replaced by proof terms, in such a way that $\hat{e}$ becomes well-typed in CIC, then e satisfies its specification κ.

Completeness

The method is incomplete if the program does not contain enough annotations

As usual, computing weakest preconditions dispenses the user from writing all annotations

In practice, inserting pre-, post- and loop invariants is all we need to do

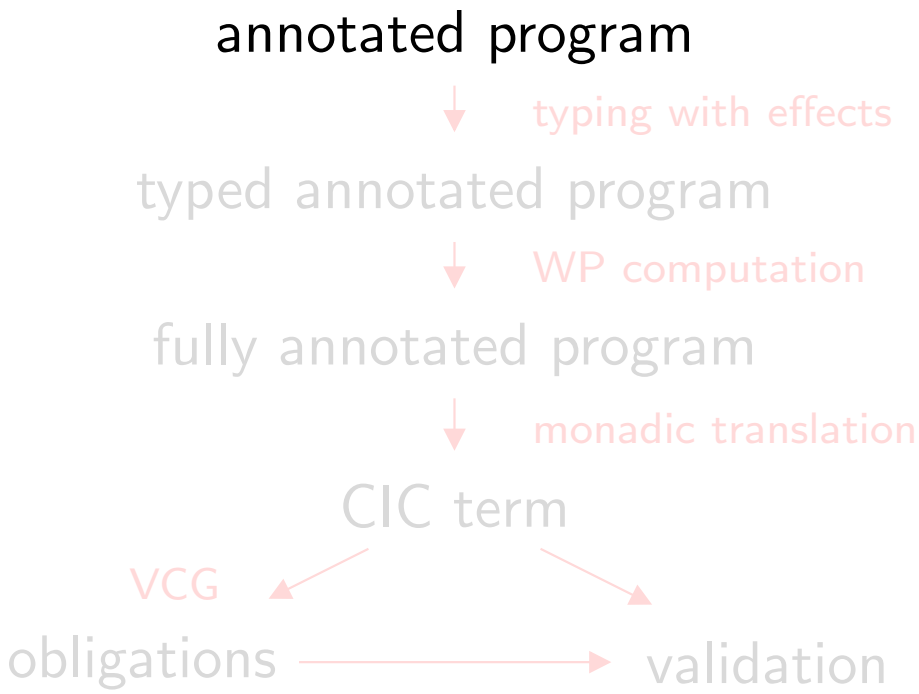
Completeness

The method is incomplete if the program does not contain enough annotations

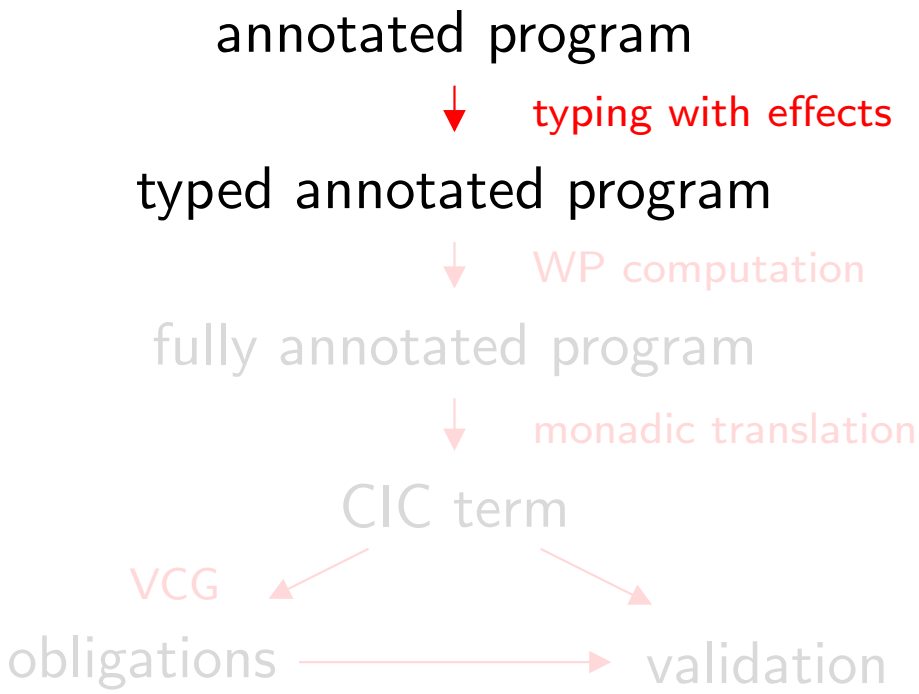
As usual, computing weakest preconditions dispenses the user from writing all annotations

In practice, inserting pre-, post- and loop invariants is all we need to do

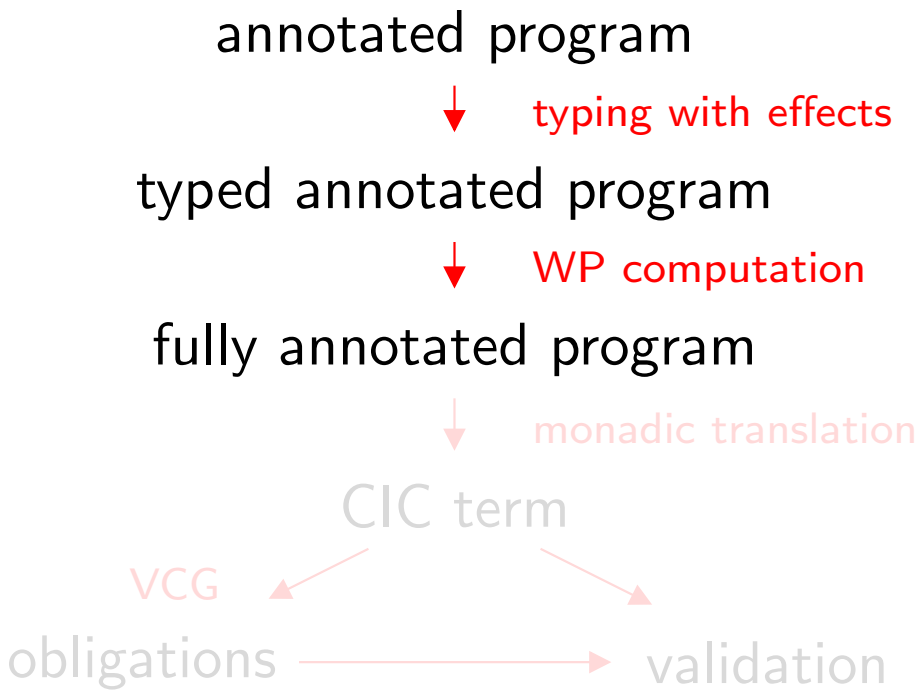
Steps in the Why tool



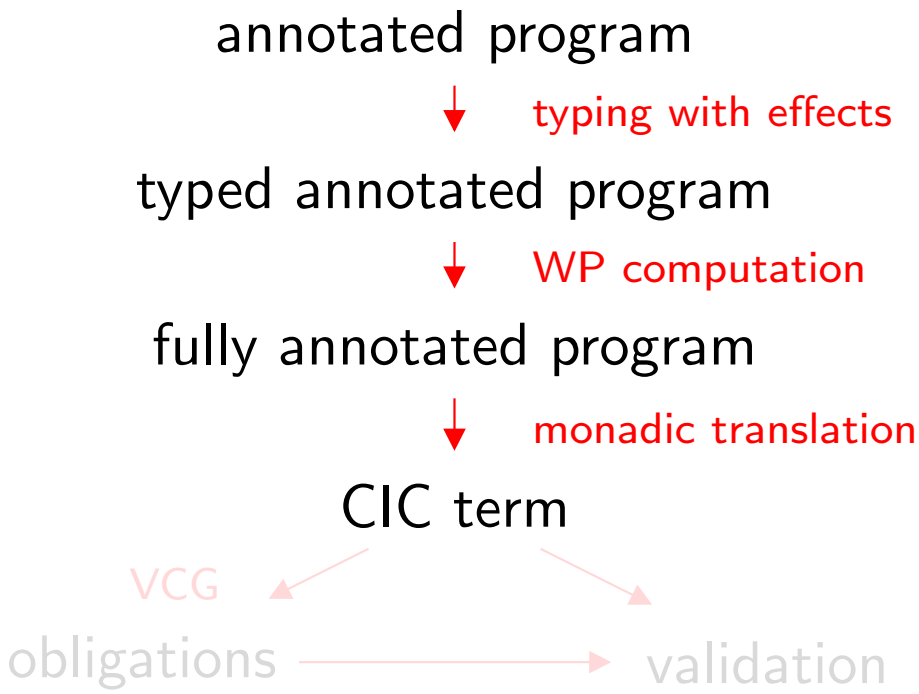
Steps in the Why tool



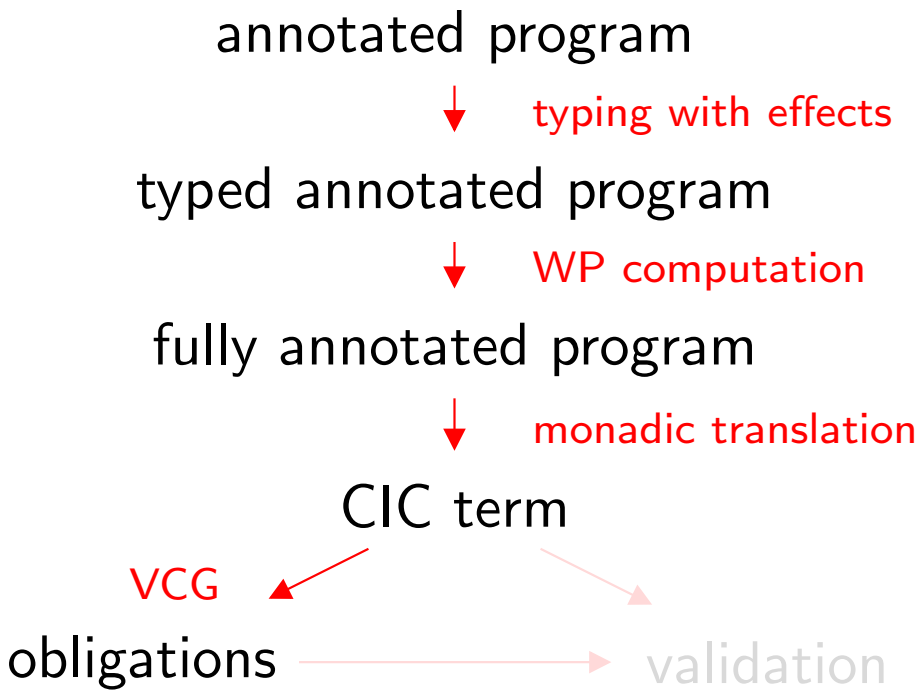
Steps in the Why tool



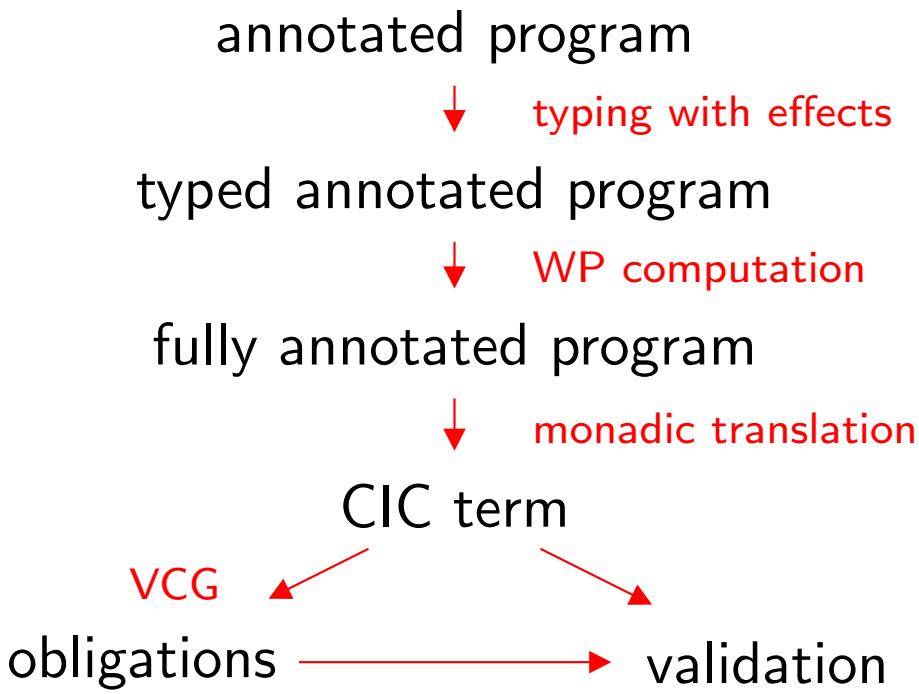
Steps in the Why tool



Steps in the Why tool



Steps in the Why tool



The Why tool in practice

The tool includes additional features

- exceptions
  - $e ::= \ldots \mid \text{raise } (E\ e) \mid \text{try } e \text{ with } E\ x \Rightarrow e$
  - $\epsilon ::= \text{reads } \vec{x} \text{ writes } \vec{x} \text{ raises } \vec{E}$
  - postcondition $Q; E_1 \Rightarrow Q_1; \ldots; E_n \Rightarrow Q_n$
- modularity
  one can declare types / logic functions / predicates / programs
- polymorphism
  but no polymorphism w.r.t effects

The Why tool in practice

The tool includes additional features

- exceptions
  - $e ::= \ldots \mid \text{raise } (E\ e) \mid \text{try } e \text{ with } E\ x \Rightarrow e$
  - $\epsilon ::= \text{reads } \vec{x} \text{ writes } \vec{x} \text{ raises } \vec{E}$
  - postcondition $Q; E_1 \Rightarrow Q_1; \ldots; E_n \Rightarrow Q_n$
- modularity
  one can declare types / logic functions / predicates / programs
- polymorphism
  but no polymorphism w.r.t effects

The Why tool in practice

The tool includes additional features

- exceptions
  - $e ::= \ldots \mid \text{raise } (E\ e) \mid \text{try } e \text{ with } E\ x \Rightarrow e$
  - $\epsilon ::= \text{reads } \vec{x} \text{ writes } \vec{x} \text{ raises } \vec{E}$
  - postcondition $Q; E_1 \Rightarrow Q_1; \ldots; E_n \Rightarrow Q_n$

- modularity
  one can declare types / logic functions / predicates / programs

- polymorphism
  but no polymorphism w.r.t effects

Logic

Annotations are not written in CIC but in a logical subset common to a maximal set of provers

polymorphic multi-sorted first-order logic
(with equality and arithmetic)

actually, we only need

- to embed logical terms inside programs
- minimal logic ($\forall \wedge \Rightarrow$) to compute weakest preconditions

Logic

Annotations are not written in CIC but in a logical subset common to a maximal set of provers

polymorphic multi-sorted first-order logic
(with equality and arithmetic)

actually, we only need

- to embed logical terms inside programs
- minimal logic ($\forall \wedge \Rightarrow$) to compute weakest preconditions

Logic

Annotations are not written in CIC but in a logical subset common to a maximal set of provers

polymorphic multi-sorted first-order logic
(with equality and arithmetic)

actually, we only need

- to embed logical terms inside programs
- minimal logic ($\forall \wedge \Rightarrow$) to compute weakest preconditions

Example

```
type 'a array

logic length : 'a array -> int
logic acc : 'a array, int -> 'a
logic upd : 'a array, int, 'a -> 'a array

parameter get : t:'a array ref -> i:int ->
   { 0 <= i < length(t) } 'a reads t { result = acc(t, i) }

parameter set : t:'a array ref -> i:int -> v:'a ->
   { 0 <= i < length(t) } unit writes t { t = upd(t@, i, v) }
```

Example (cont'd)

```
exception Found of int
exception Not_found

let search (t : int array ref) =
  {}
  try
    let i = ref 0 in begin
    while !i < (length !t) do
      { invariant 0 <= i and forall k:int. 0 <= k < i -> acc(t,k) <> 0
        variant length(t) - i }
      if (get t !i) = 0 then raise (Found !i);
      i := !i + 1
    done;
    raise Not_found : int
    end
  with Found x ->
    x
  end
  { acc(t, result) = 0
  | Not_found => forall k:int. 0 <= k < length(t) -> acc(t,k) <> 0 }
```

Example (cont'd)

We get the usual Hoare logic proof obligations:

1. loop invariant holds initially
2. precondition of `(get t !i)`
3. preservation of the loop invariant
4. decreasing of the variant (termination)
5. final postcondition when terminating normally
6. final postcondition when `Not_found` is raised

all of them are automatically discharged by Simplify

Note: we can still type-check the validation (with Coq)

Example (cont'd)

We get the usual Hoare logic proof obligations:

1. loop invariant holds initially
2. precondition of `(get t !i)`
3. preservation of the loop invariant
4. decreasing of the variant (termination)
5. final postcondition when terminating normally
6. final postcondition when `Not_found` is raised

all of them are automatically discharged by Simplify

Note: we can still type-check the validation (with Coq)

Example (cont'd)

We get the usual Hoare logic proof obligations:

1. loop invariant holds initially
2. precondition of `(get t !i)`
3. preservation of the loop invariant
4. decreasing of the variant (termination)
5. final postcondition when terminating normally
6. final postcondition when `Not_found` is raised

all of them are automatically discharged by Simplify

Note: we can still type-check the validation (with Coq)

Part II

Verifying C and Java programs

We propose a new approach for the verification of Java and C programs and two tools

- Krakatoa: verification of Java programs
  - specified using JML (Java Modeling Language)
- Caduceus: verification of C programs
  - specification language à la JML

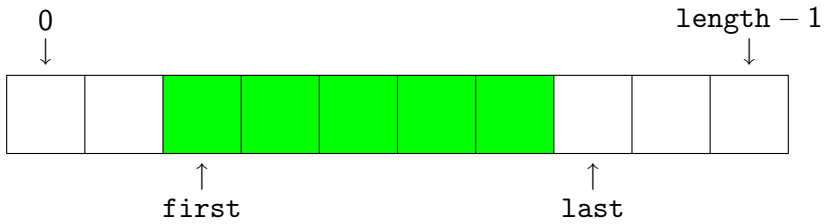
Both are currently under experimentation on industrial code (Axalto / Dassault Aviation)

We propose a new approach for the verification of Java and C programs and two tools

- Krakatoa: verification of Java programs
  - specified using JML (Java Modeling Language)
- Caduceus: verification of C programs
  - specification language à la JML

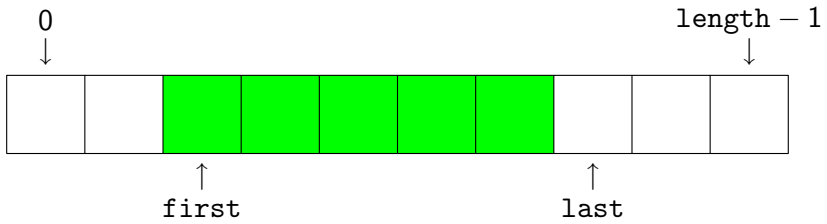
Both are currently under experimentation on industrial code (Axalto / Dassault Aviation)

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

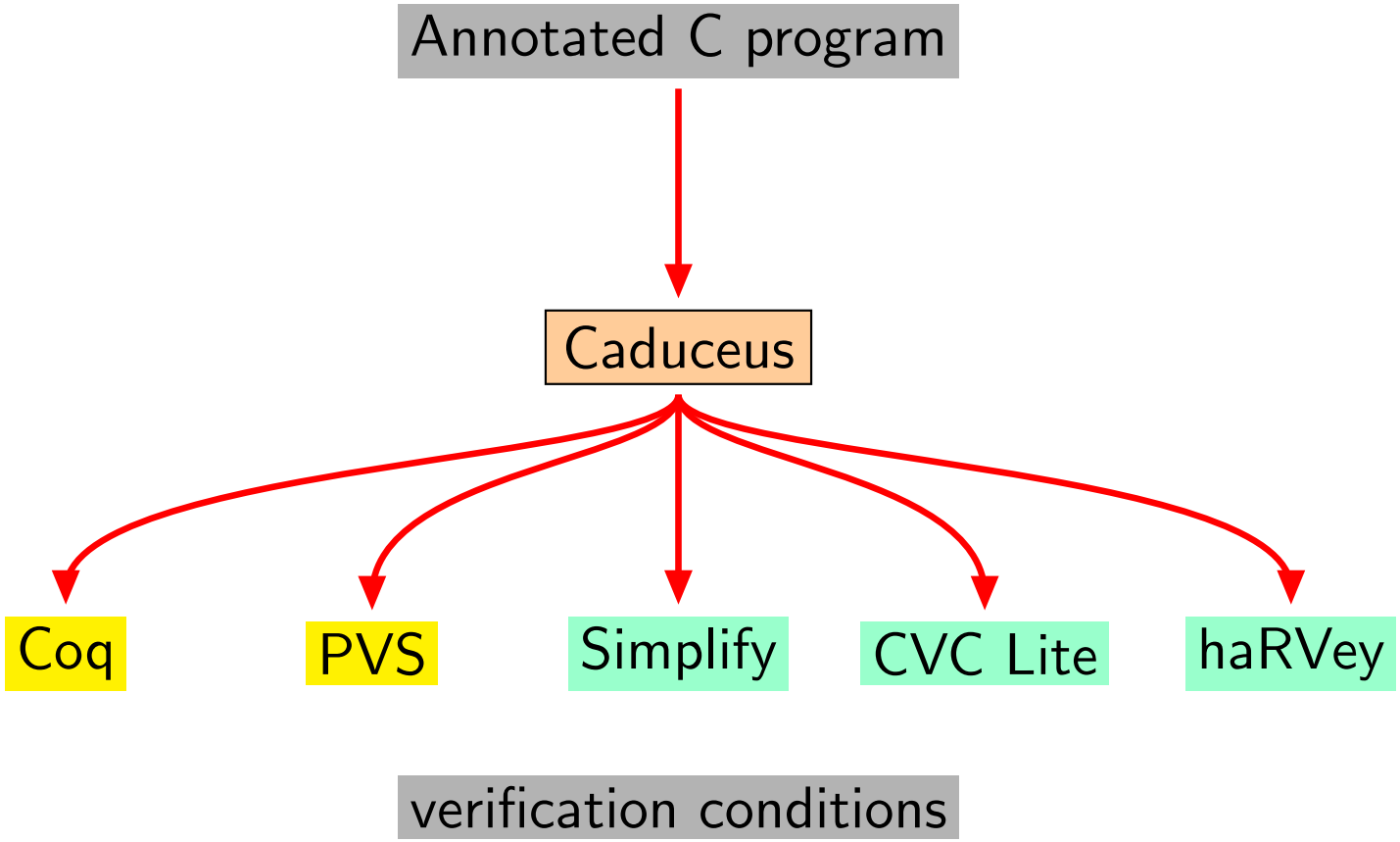
```
/*@ requires !q.full
  @ assigns  q.empty, q.full, q.last, q.contents[q.last]
  @ ensures  !q.empty && q.contents[\old(q.last)] == c
  @*/
void push(char c);

/*@ 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'
}
```

Multi-prover architecture



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`

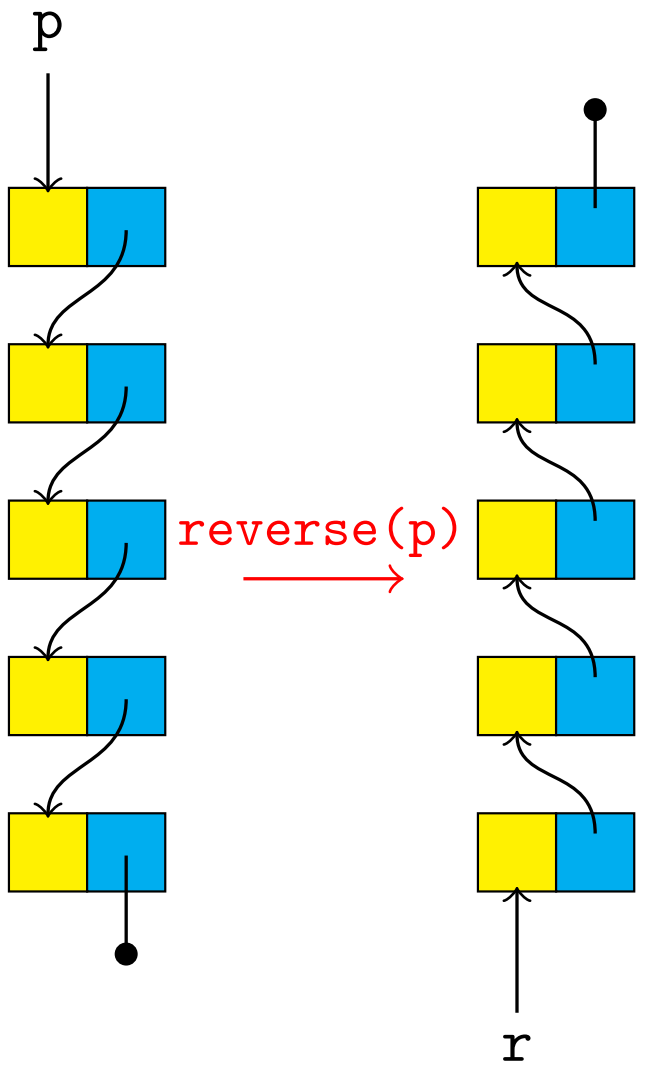
Proofs of these obligations

- 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

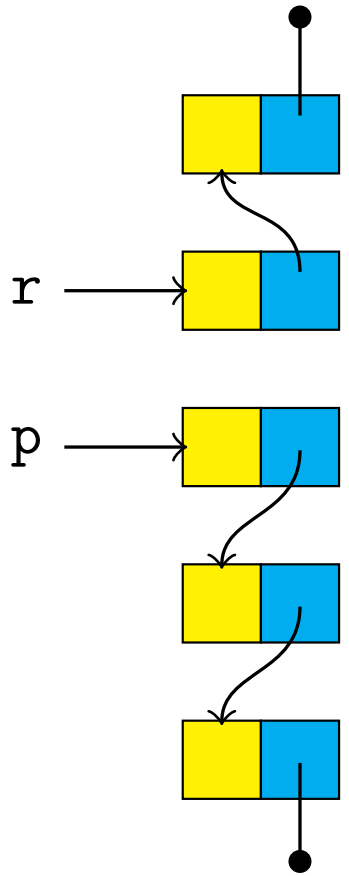
```
/* llist(p,l) specifies that l is the list of pointers
   from p to NULL following tl fields */
//@ predicate llist(list p, plist l) reads p->tl

// 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

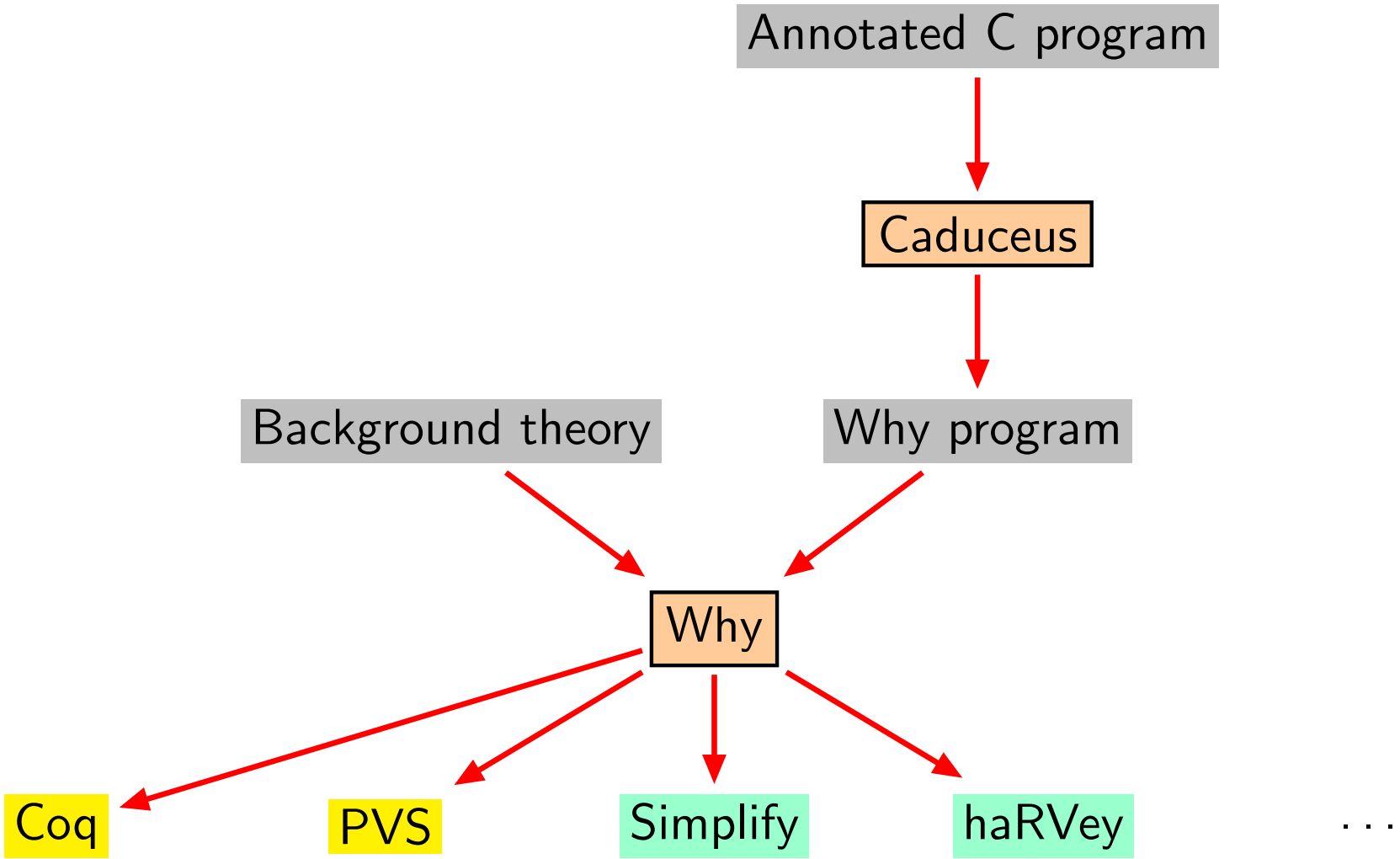
- Graph marking algorithm
- Considered as a benchmark for the verification of pointer programs (Bornat, 1999, Jape system) (Nipkow-Mehta, 2003, Isabelle/HOL)
- 12 verification conditions
- With Simplify: 33%
- With Coq: 100%, with 2362 lines of tactics

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

Underlying technique

Underlying technique



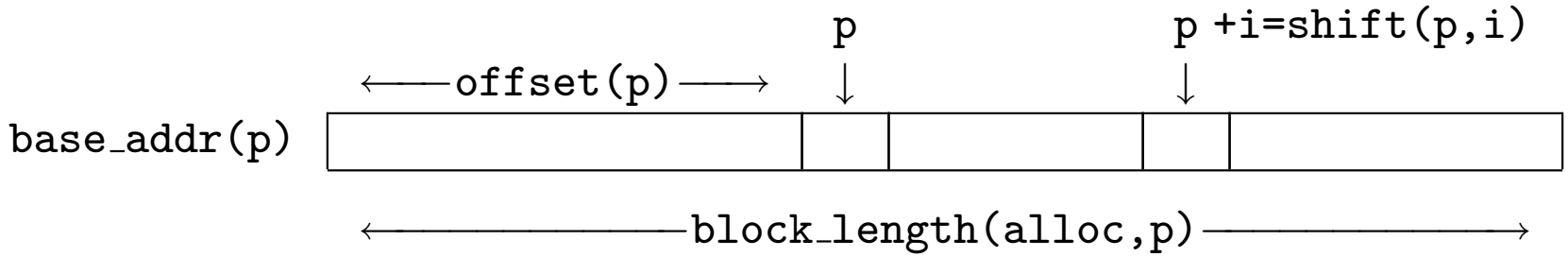
Modeling C memory heap

- 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

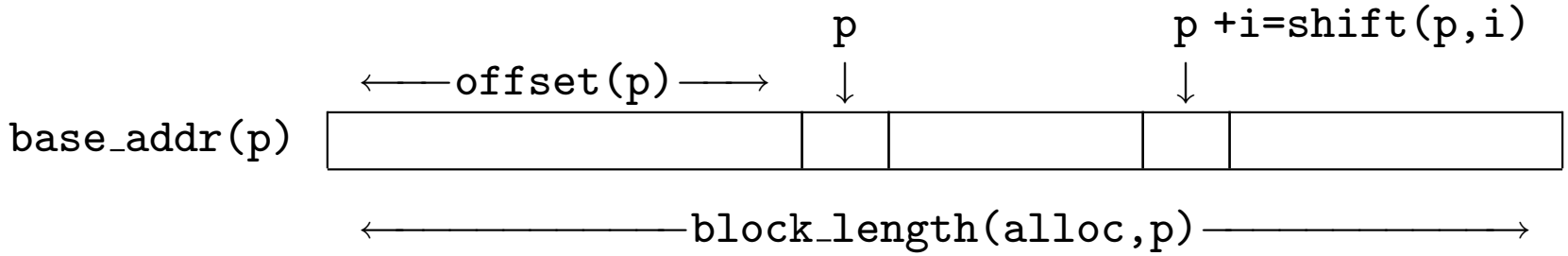
- Burstall-Bornat model: memory partition according to structure fields
- We extend this idea to handle C arrays and pointer arithmetic: a memory block is



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

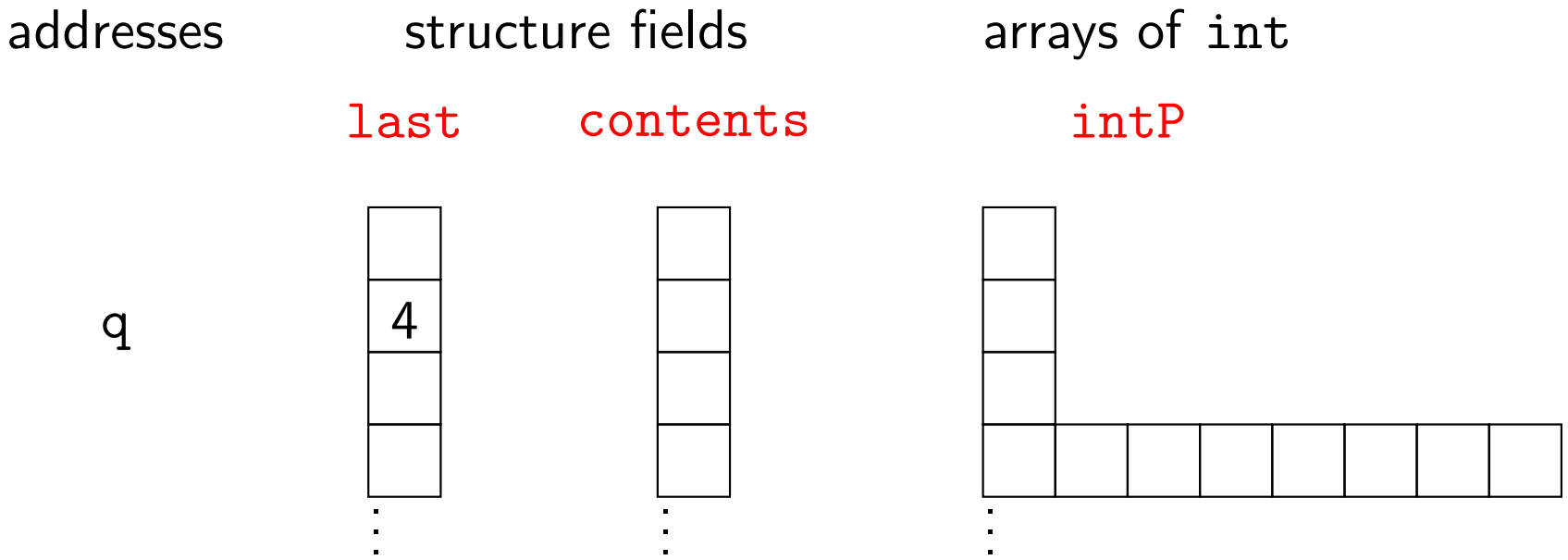
Modeling C memory heap

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



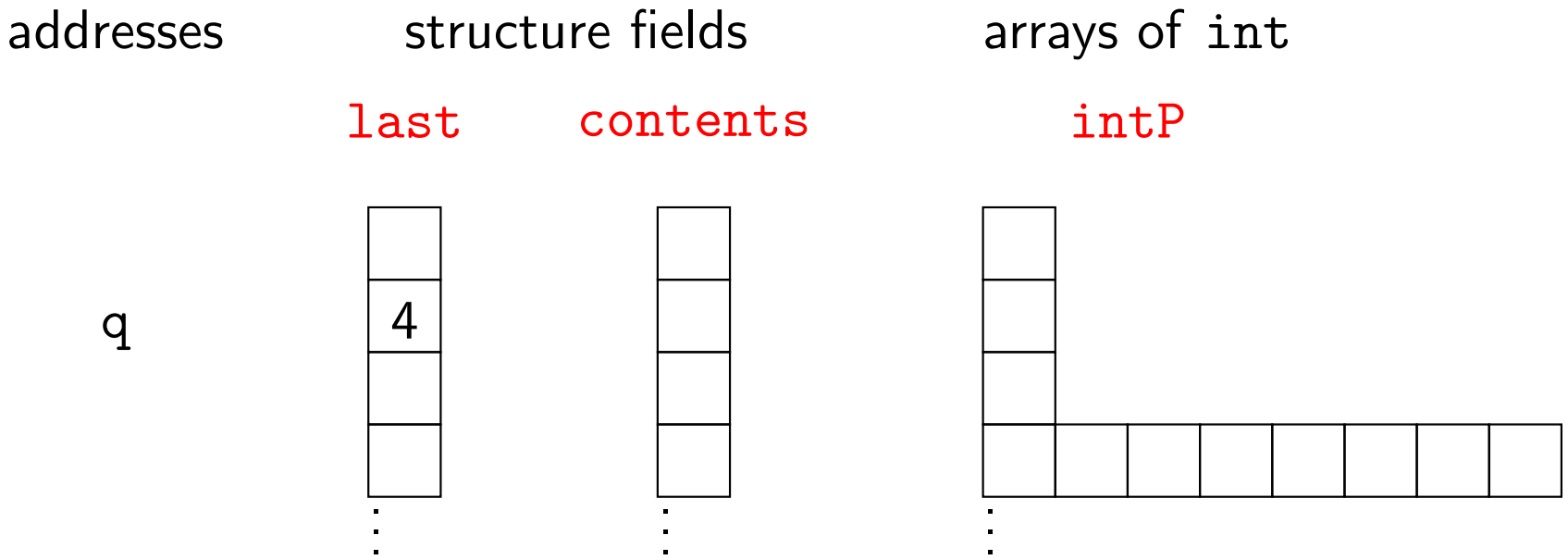
- Each structure field is a map from addresses to memory blocks

Burstall-Bornat model: example of character queue



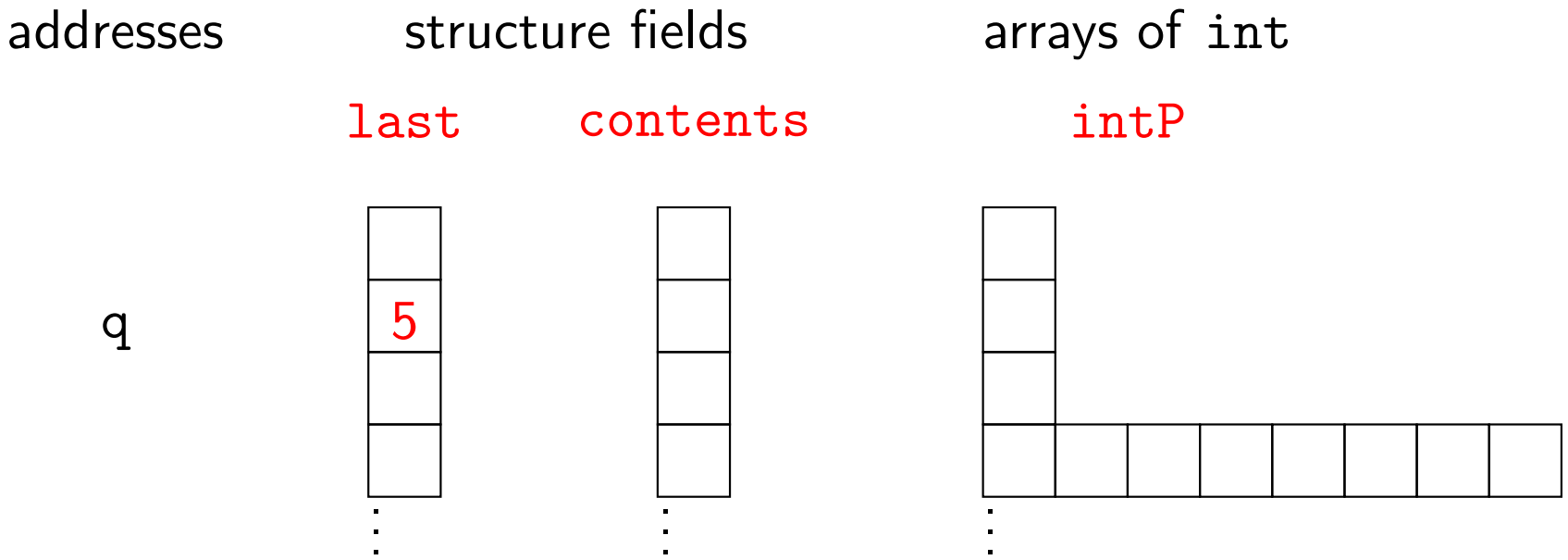

```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

Burstall-Bornat model: example of character queue



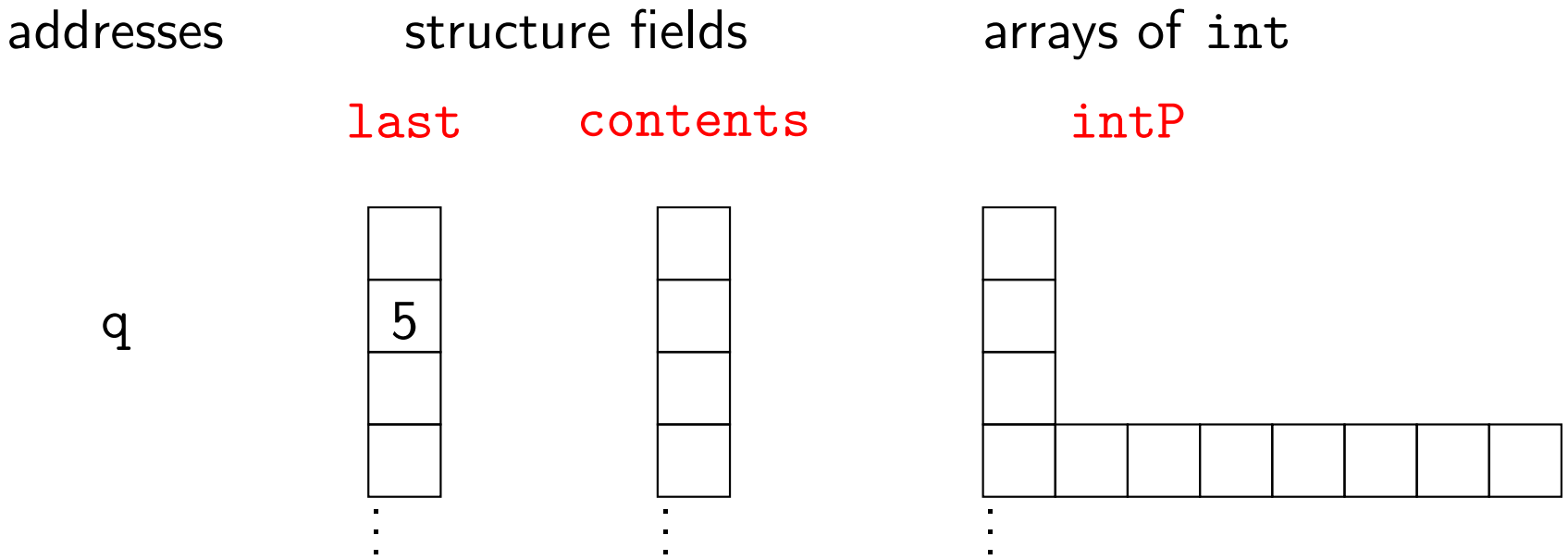

```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

Burstall-Bornat model: example of character queue



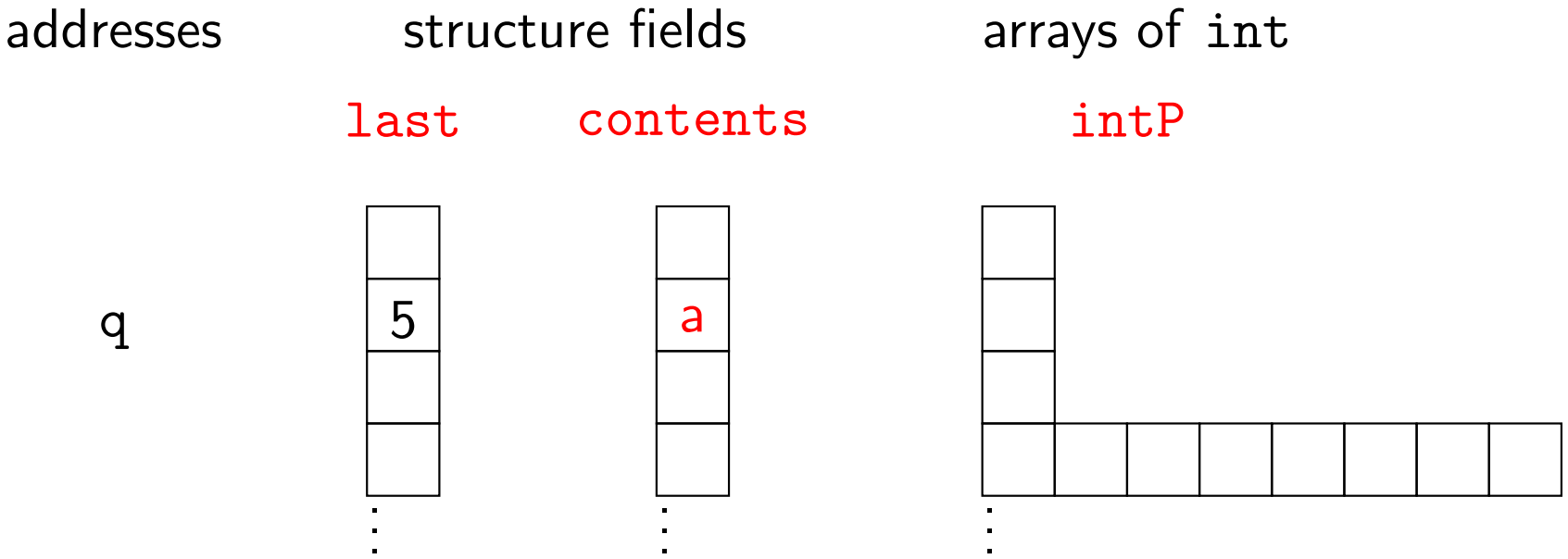

```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

Burstall-Bornat model: example of character queue




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

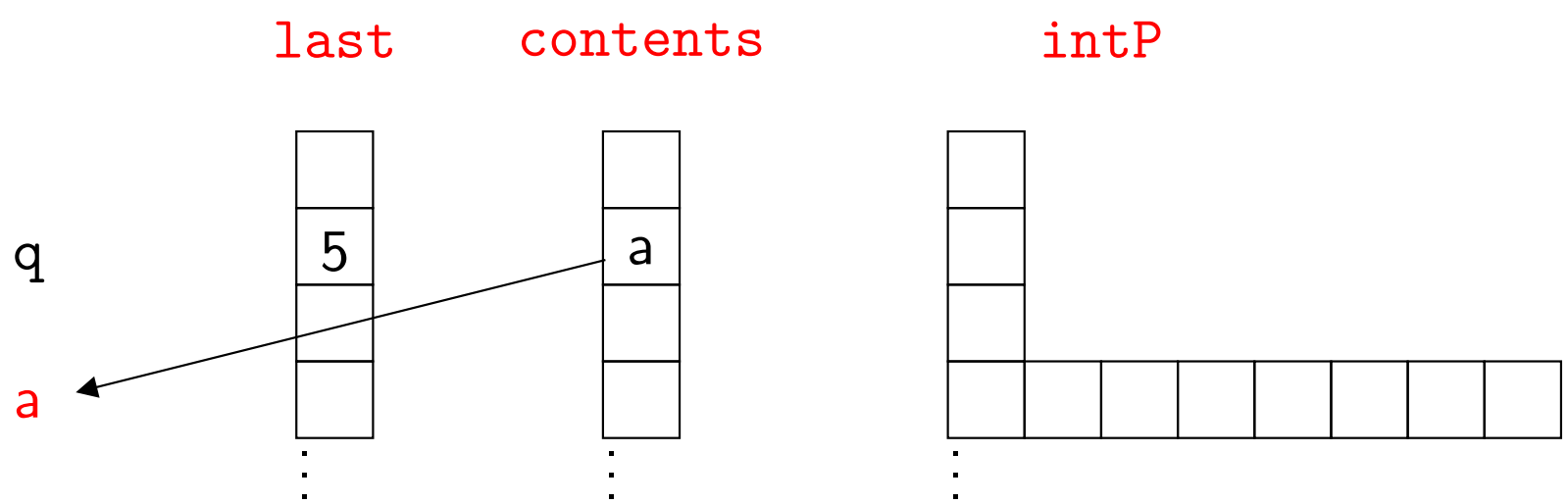
Burstall-Bornat model: example of character queue




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

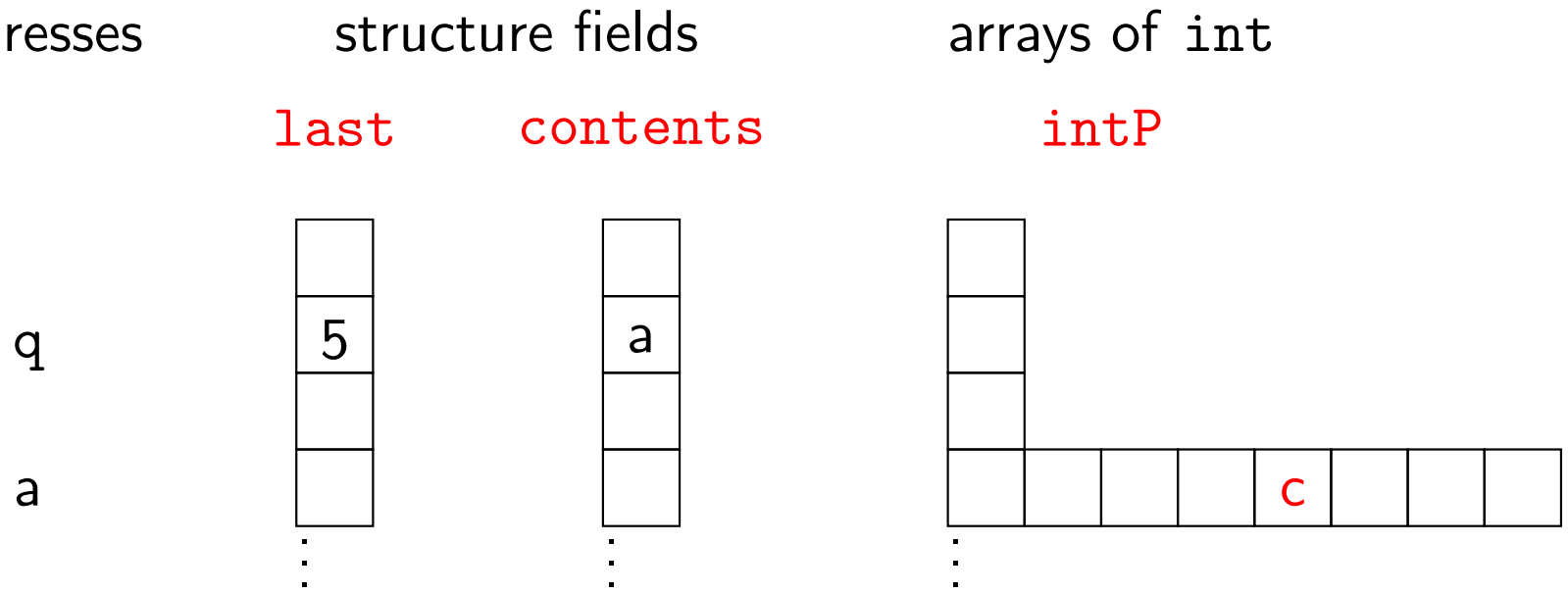
Burstall-Bornat model: example of character queue

addresses structure fields arrays of `int`



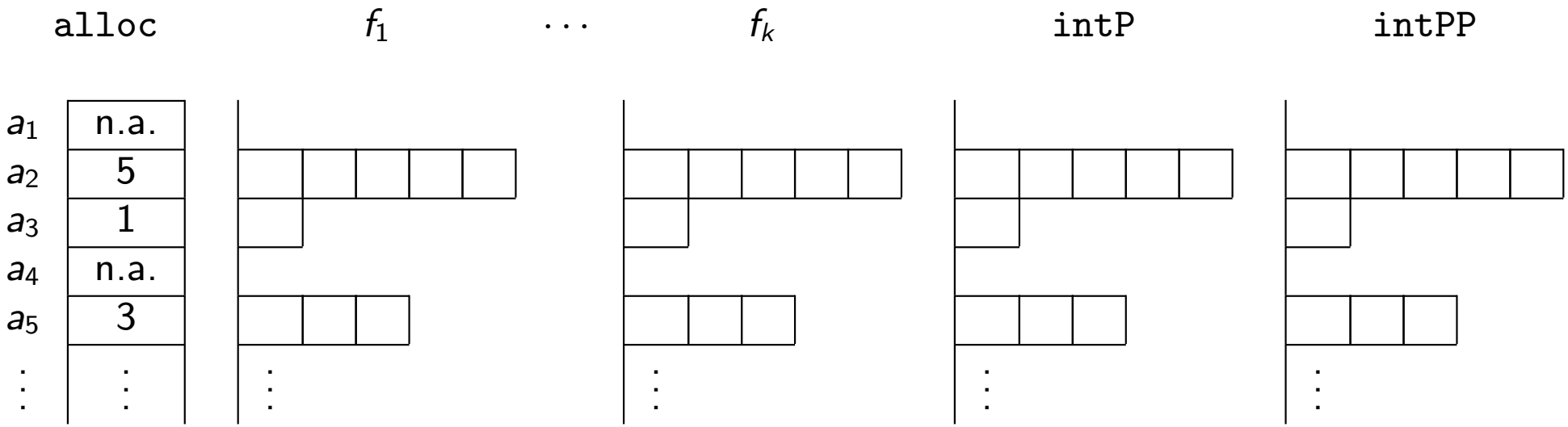

```
q.contents[q.last++] = c
    q.last <- 4 + 1
    *(q.contents+4) <- c
```

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

Soundness

Soundness of the translation not formally proved, but

- based on the Why tool, which is sound
  - the Why validation can be type-checked
- the modelling is quite simple
- ultimately, consistency of the background theory can be proved (e.g. by a realization in Coq)

Soundness

Soundness of the translation not formally proved, but

- based on the Why tool, which is sound
  - the Why validation can be type-checked
- the modelling is quite simple
- ultimately, consistency of the background theory can be proved (e.g. by a realization in Coq)

Java programs: the Krakatoa tool

Same methodology, with a very similar model

- one map for each class attribute
- a pointer is valid if it is not `null` (no pointer arithmetic)

But the JML sometimes lacks a precise semantics

Java programs: the Krakatoa tool

Same methodology, with a very similar model

- one map for each class attribute
- a pointer is valid if it is not `null` (no pointer arithmetic)

But the JML sometimes lacks a precise semantics

Part III

Conclusion and perspectives

Conclusion

- We are able to certify non trivial programs
- We support a large subset of ANSI C and Java/JML
- Prototypes 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
- Prototypes 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
- C unions

Limitations of the model

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

Current limitations / work in progress

Limitations of the tools

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

Limitations of the model

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

Verification of ML programs

The Burstall-Bornat model could apply (one map for each mutable structure field); includes references as a particular case

- `type 'a ref = { mutable contents : 'a }`

The Why approach reaches its limits: the type system is not powerful enough to handle the full combination of higher-order and polymorphism

A good specification language has to be designed first

Verification of ML programs

The Burstall-Bornat model could apply (one map for each mutable structure field); includes references as a particular case

- `type 'a ref = { mutable contents : 'a }`

The Why approach reaches its limits: the type system is not powerful enough to handle the full combination of higher-order and polymorphism

A good specification language has to be designed first

Verification of ML programs

The Burstall-Bornat model could apply (one map for each mutable structure field); includes references as a particular case

- `type 'a ref = { mutable contents : 'a }`

The Why approach reaches its limits: the type system is not powerful enough to handle the full combination of higher-order and polymorphism

A good specification language has to be designed first

Verification of ML programs

The Burstall-Bornat model could apply (one map for each mutable structure field); includes references as a particular case

- `type 'a ref = { mutable contents : 'a }`

The Why approach reaches its limits: the type system is not powerful enough to handle the full combination of higher-order and polymorphism

A good specification language has to be designed first