

HECTOR

a software model checker
with cooperating analysis plugins

Nathaniel Charlton and Michael Huth
Imperial College London

Introduction

- HECTOR targets imperative heap-manipulating programs
 - uses abstraction to extract finite model from program
- Checks linear-time safety properties using variant of LTL
 - e.g. memory safety, assertion violations
 - results sound but imprecise
- Main idea investigated:

Use several different abstraction domains,
cooperatively, to analyse the target program

Why we combine abstractions

- many specialised abstraction domains have been developed to focus on different types of program behaviour
 - e.g. dedicated domains for linked data structures, polynomial invariants, secure data-flow, object type analysis for OOP
- But in “real” programs these things are all mixed together
- Because the domains are not independent, they can use each others’ intermediate results to provide more precision

How we combine abstractions

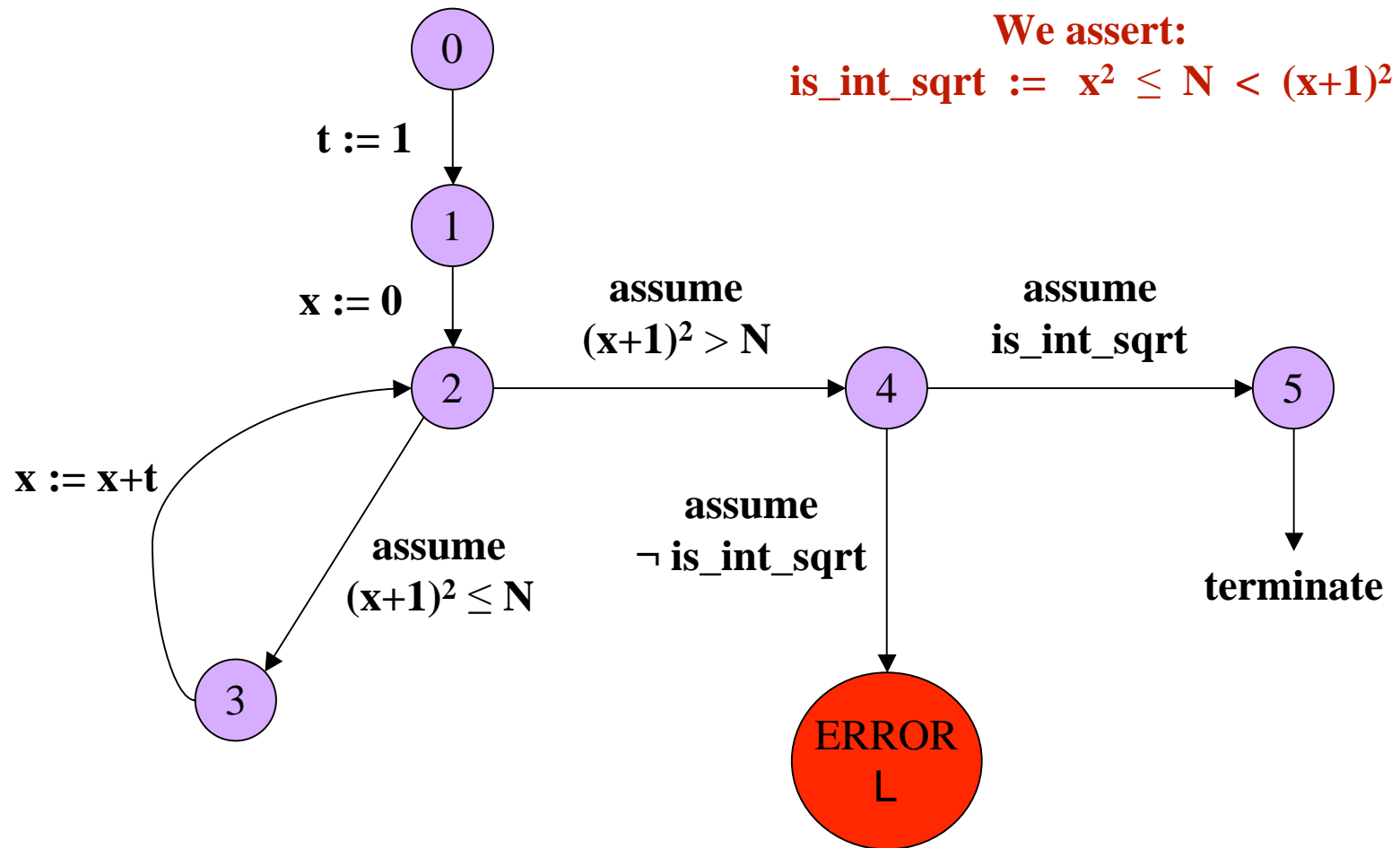
- Wrap each abstraction domain behind a common interface, as an “analysis module”
- This interface allows analysis modules to exchange information about program state, expressed as formulae of a single common logic
- Generic model construction and checking algorithms work with whatever analysis modules are provided
- Common logic is: **First Order + Transitive Closure**
 - need to express reachability in the heap
 - but don't want to make reasoning too difficult

Aiming for modularity

This design decision strives for **modularity**:

- Implement one thing at a time:
 - When writing e.g. a shape analysis, just concentrate on shapes...
- When implementing a new analysis, **just need to make it “understand” the single common logic**
 - Then your new analysis will automatically cooperate with the existing ones

Example 😊



Hector: create new model

Program to analyse

Program control flow graphs:

```
%% This is what we want at the end, i.e. what it means to have correctly  
%% calculated the integer square root.
```

```
shorthand(is_int_sqrt, and(lte(times(x, x), n), lt(n, times(plus(x,1), plus(x,1))))).
```

```
method(intsqrt, [n], [x, t]).
```

```
edge(intsqrt, 0, assConst(t, 1), 1).
```

```
edge(intsqrt, 1, assConst(x, 0), 2).
```

```
edge(intsqrt, 2, assume(lte(times(plus(x,1), plus(x,1)), n)), 3).
```

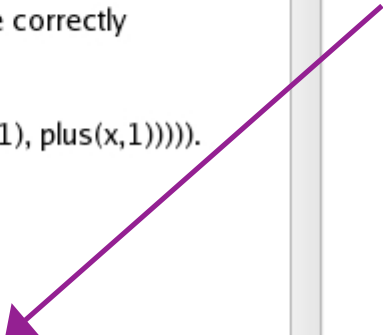
```
edge(intsqrt, 2, assume(gt(times(plus(x,1), plus(x,1)), n)), 4).
```

```
edge(intsqrt, 3, assArith(x, plus, x, t), 2).
```

```
edge(intsqrt, 4, assume(not(is_int_sqrt)), asserterror).
```

```
edge(intsqrt, 4, assume(is_int_sqrt), 5).
```

Step 1: enter textual
representation of
program's CFGs



Trivector Predicate Abstraction

Enabled

Enter abstraction predicates for method intsqrt

```
lte(times(x, x), n).  
gte(n, 0).  
lte(times(plus(x,1), plus(x,1)), n).  
eq(t, 1).
```

Step 2: Configure the abstraction

E.g. for predicate abstraction, choose the abstraction predicates

Monomial Predicate Abstraction

Enabled

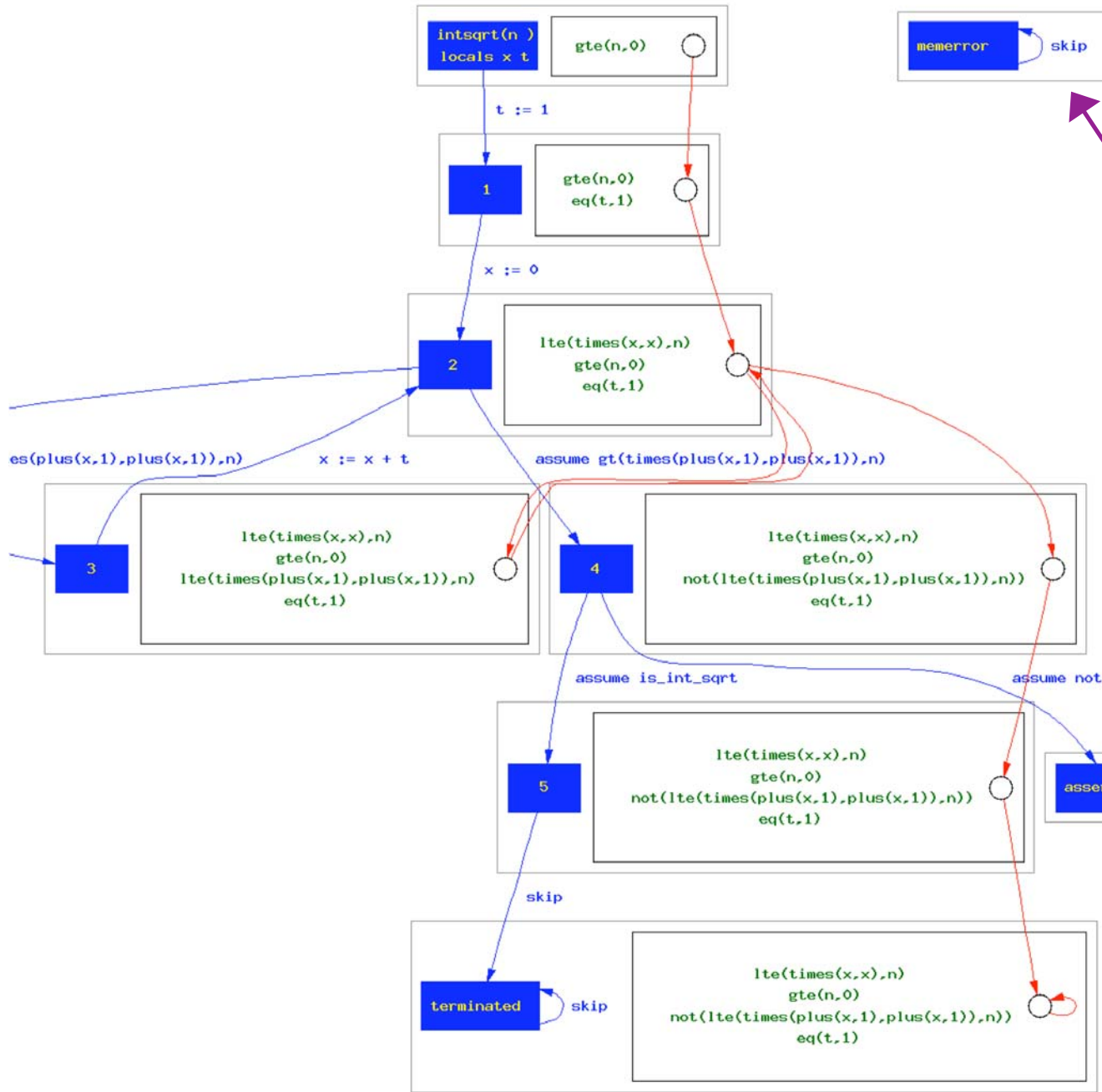
Enter abstraction predicates for method intsqrt

Can enable or disable each analysis module

Shape analysis

Enabled

Fields to ignore:

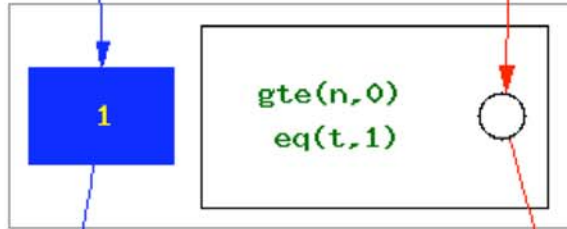


Error states
unreachable –
program is OK

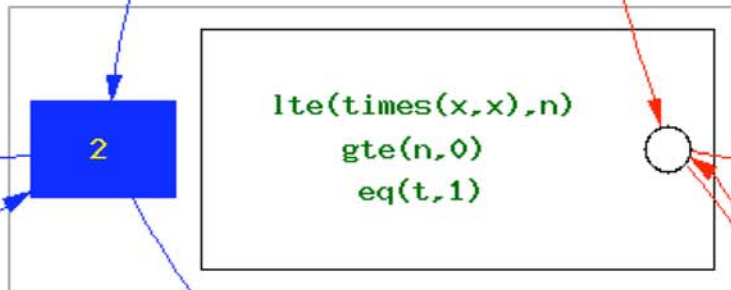


`memerror`

`t := 1`



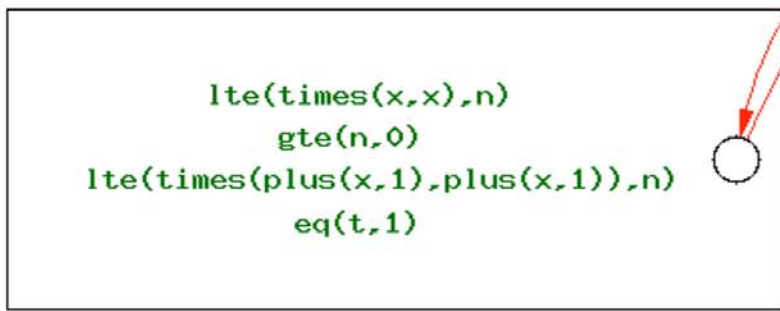
`x := 0`

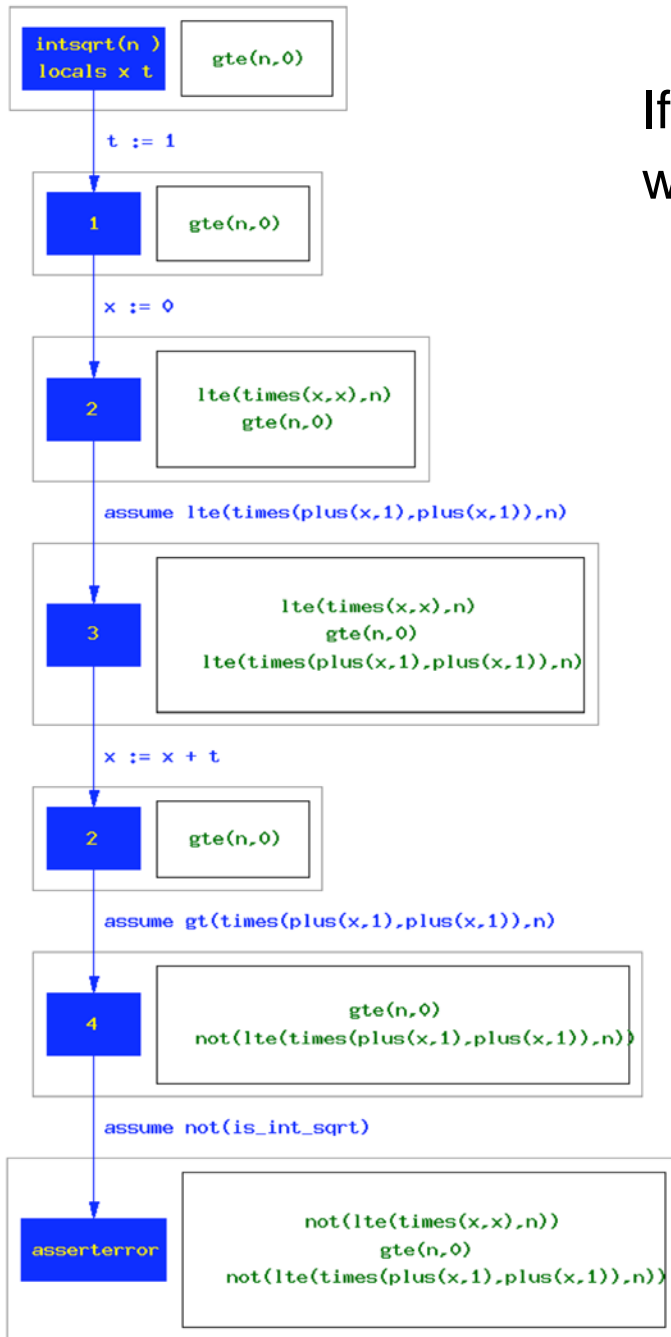


`plus(x,1),n`

`x := x + t`

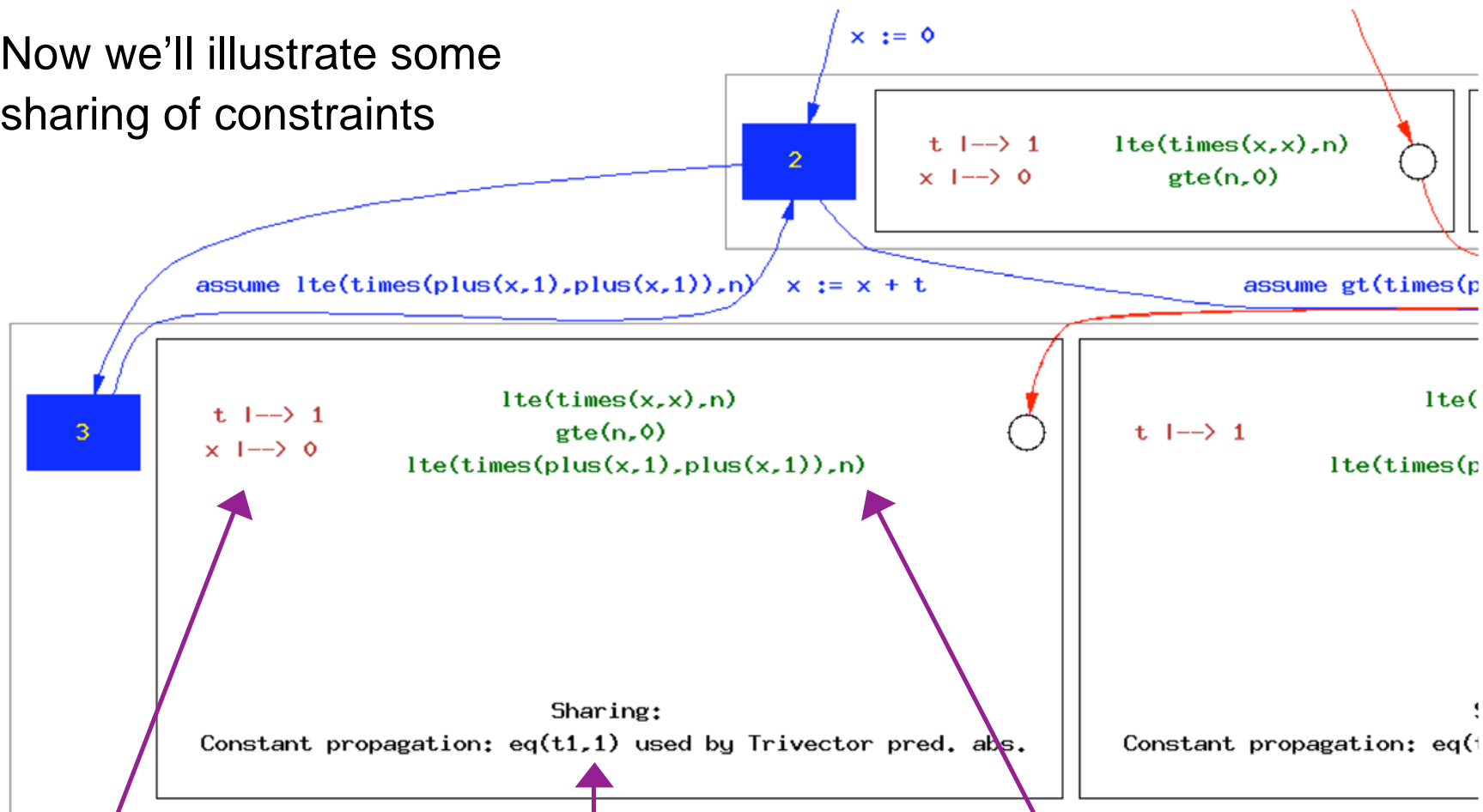
`assume gt(times(plus(x,1),plus(x,1)),n)`





If we forget to include the predicate $t=1$, we get a putative counterexample trace

Now we'll illustrate some sharing of constraints



Abstract value for constant propagation module

Reports which shared formulae were used

Abstract value for predicate abstraction module

Generally, only a small proportion of the shared formulae will be useful. Usually, unused sharing is not shown.

`assume is_int_sqrt`

5

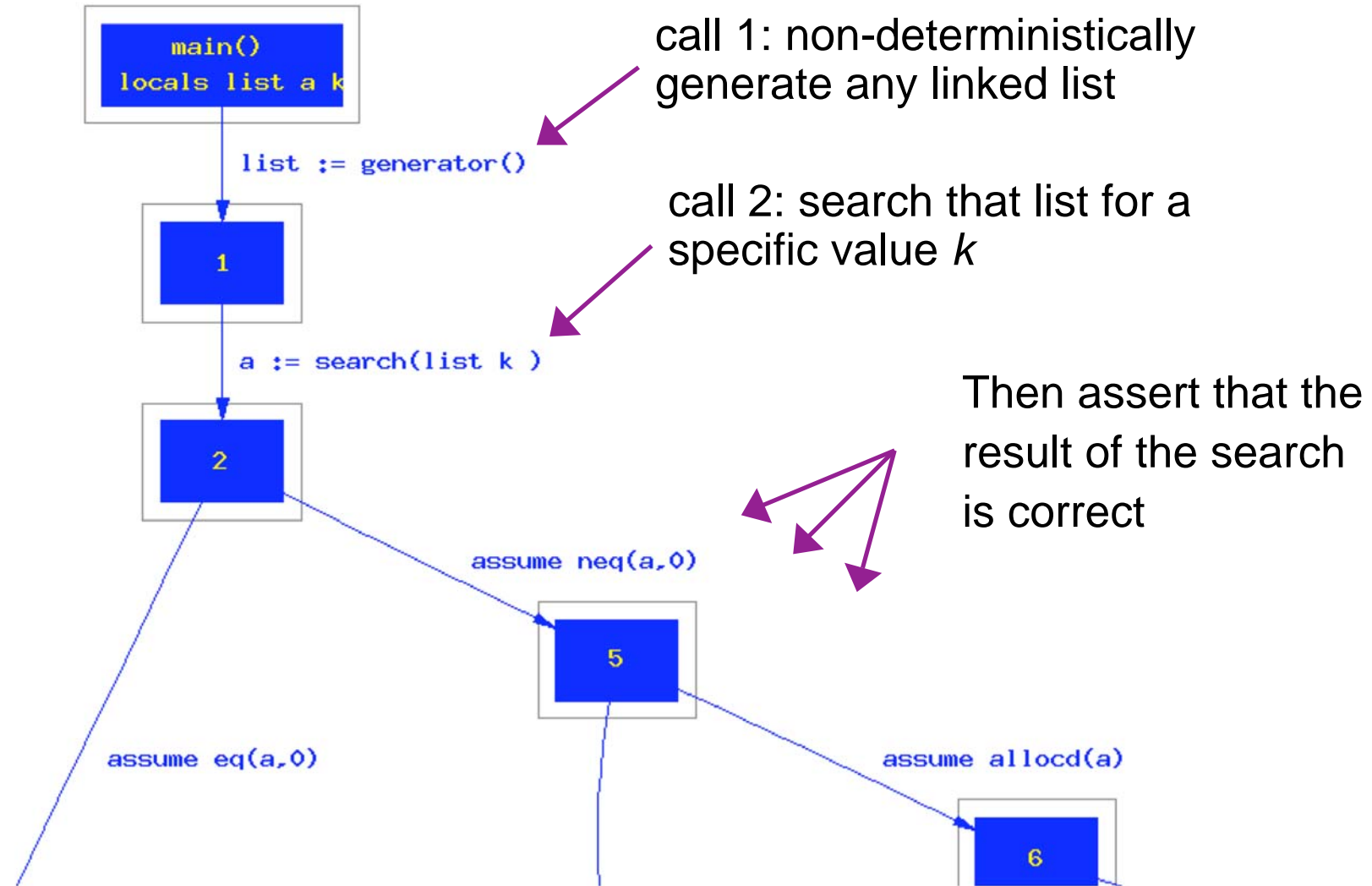
```
t |--> 1
x |--> 0
      lte(times(x,x),n)
      gte(n,0)
      not(lte(times(plus(x,1),plus(x,1)),n))
```

Sharing:

```
Trivector pred. abs.: lte(times(x1,x1),n1) unused
Trivector pred. abs.: gte(n1,0) unused
Trivector pred. abs.: not(lte(times(plus(x1,1),plus(x1,1)),n1)) unused
Constant propagation: eq(t1,1) unused
Constant propagation: eq(x1,0) unused
```

Here several formulae are shared, but none prove to be useful.

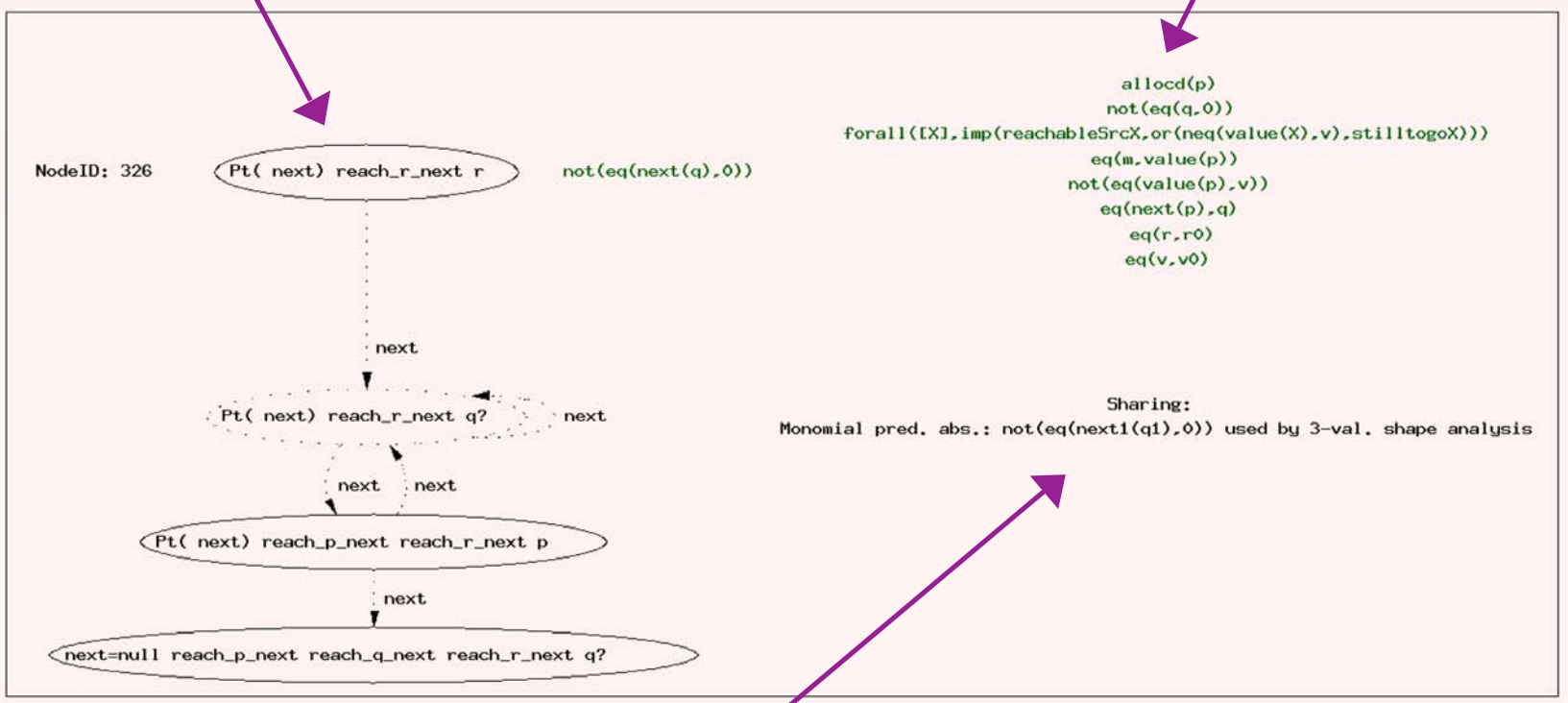
Ex 2: data structures and procedures



Abstract value for
shape analysis module
(invokes TVLA software)

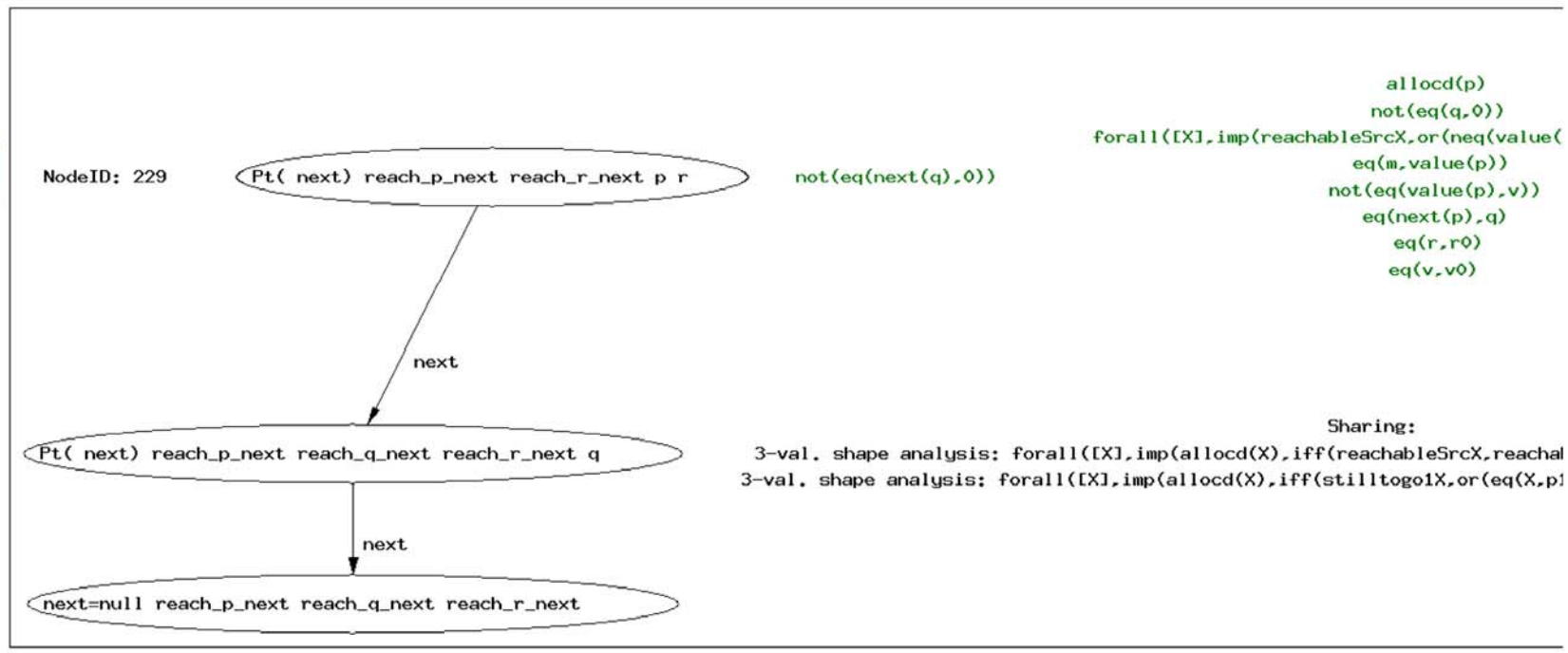
Abstract value
for predicate
abstraction
module

5



Here, the predicate abstraction module provides a fact which is used by the shape analysis module

7



Here it's the reverse: the **shape analysis module shares a fact about reachability** which is used by the predicate abstraction module

Predicate abstraction treats TC subformulae as uninterpreted predicates, and so the shape analysis module must do all the reachability reasoning.

Checking LTL safety properties

$$\begin{aligned} \Phi ::= & \Phi \wedge \Phi \mid \Phi \vee \Phi \quad \mid G\Phi \mid X\Phi \mid \Phi W \Phi \\ & \mid p \mid \neg p \mid \text{inMethod}(M) \mid \neg \text{inMethod}(M) \end{aligned}$$

Typical grammar for LTL safety properties, except that
“propositions” are now formulae from FO + TC logic e.g.:

$$G \neg \left(\begin{array}{l} \text{inMethod}(\textit{search}) \\ \wedge \neg \exists X : \textit{reachable}X \wedge \textit{value}(X) = v \end{array} \right)$$

Sharing machinery reused during checking, to evaluate
“propositions”

Model check

Absence of memory errors

Absence of assertion violations

Absence of memory errors and assertion violations

Always

Never

until

Custom property

Trace to location of method

Trace to above location, via location of method

Prefer "strong" counterexample

Determinise automaton

some standard patterns
set up for model checking

or enter an ad-hoc LTL query



One gets a nice counterexample trace



Summary

- **HECTOR** is a software model checker for imperative heap-manipulating programs
 - checks linear time safety LTL properties
 - “atomic” propositions are FO + TC formulae
- **Key feature: abstraction domains combined in a modular way by exchanging formulae in a single common logic**
- Aim: Enable the use of different abstraction domains cooperatively, yet still maintain a clean separation between aspects of the analysis

For more details, see my website

<http://www.doc.ic.ac.uk/~nac103>

or find me at my exhibition booth. 😊