Practical Proof Planning for Formal Methods

Alan Bundy

School of informatics

University of Edinburgh

Deduction and Software Engineering

- Formal methods of system development offer correctness assurance.

  Can deal with infinite state systems without abstraction.

  Complementary to model checking and testing.

- Produces proof obligations which are long and complicated.

  Machine assistance required to reduce error and tedium.

- Proof search produces combinatorial explosion.

  Reasoning about repetition especially explosive.

  Automatic search often defeated.

  Interactive proof highly skilled and time-consuming.

Why Proof Planning?

- Captures common patterns of reasoning.
- Heirarchical proof structure:

  enhances understanding of proof;

  tames combinatorial explosion.
- Also captures common patterns of failure analysis and repair.

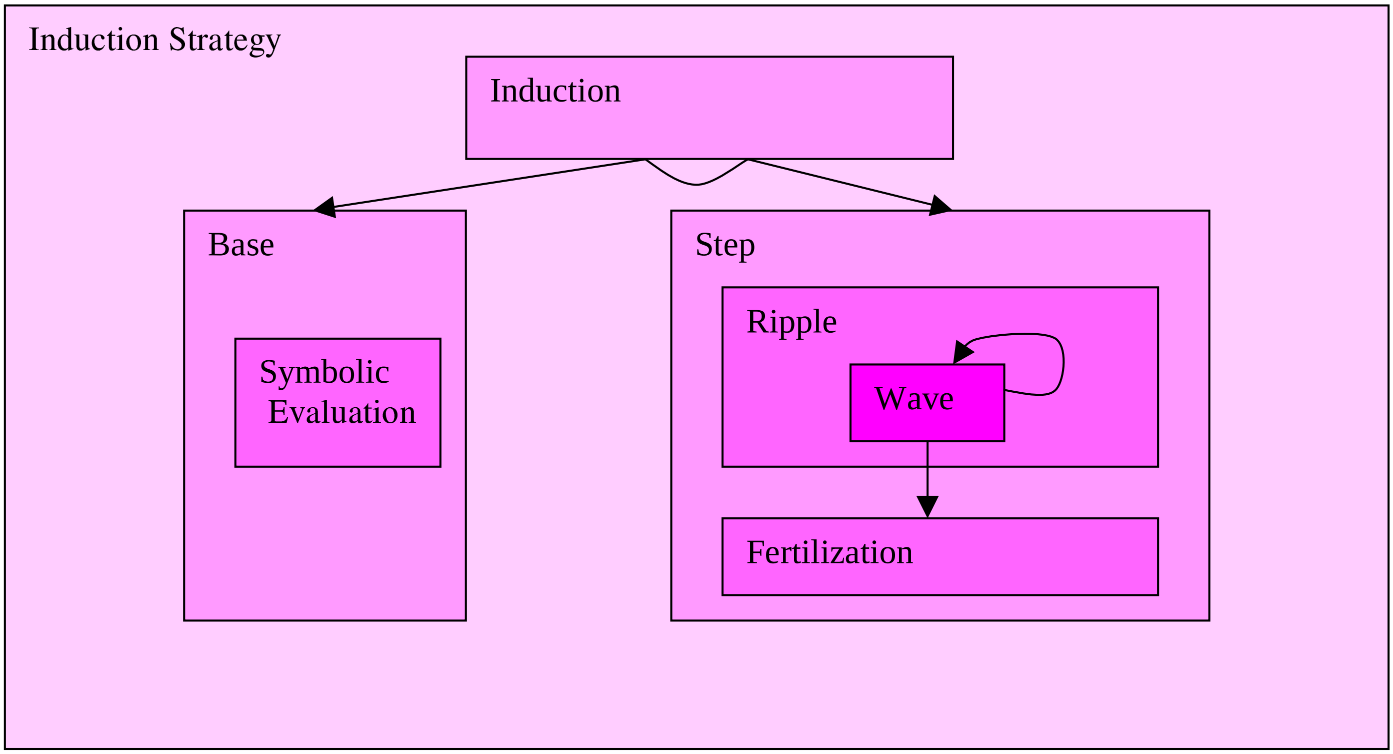
  Thereby, automates discovery of ‘eureka’ steps.
- Hence, supports both interactive and automated proof.

What is Proof Planning

- Represent common patterns of reasoning as tactics.
- Specify each tactic in a method.
- Reason with specifications to form proof plan from tactics.
- Build customised proof plan for each conjecture.
- Represent common patterns of proof failure as critics.
- Patch failed proof attempts with critics.

General-Purpose Proof Plans

A Strategy for Inductive Proof: ind_strat



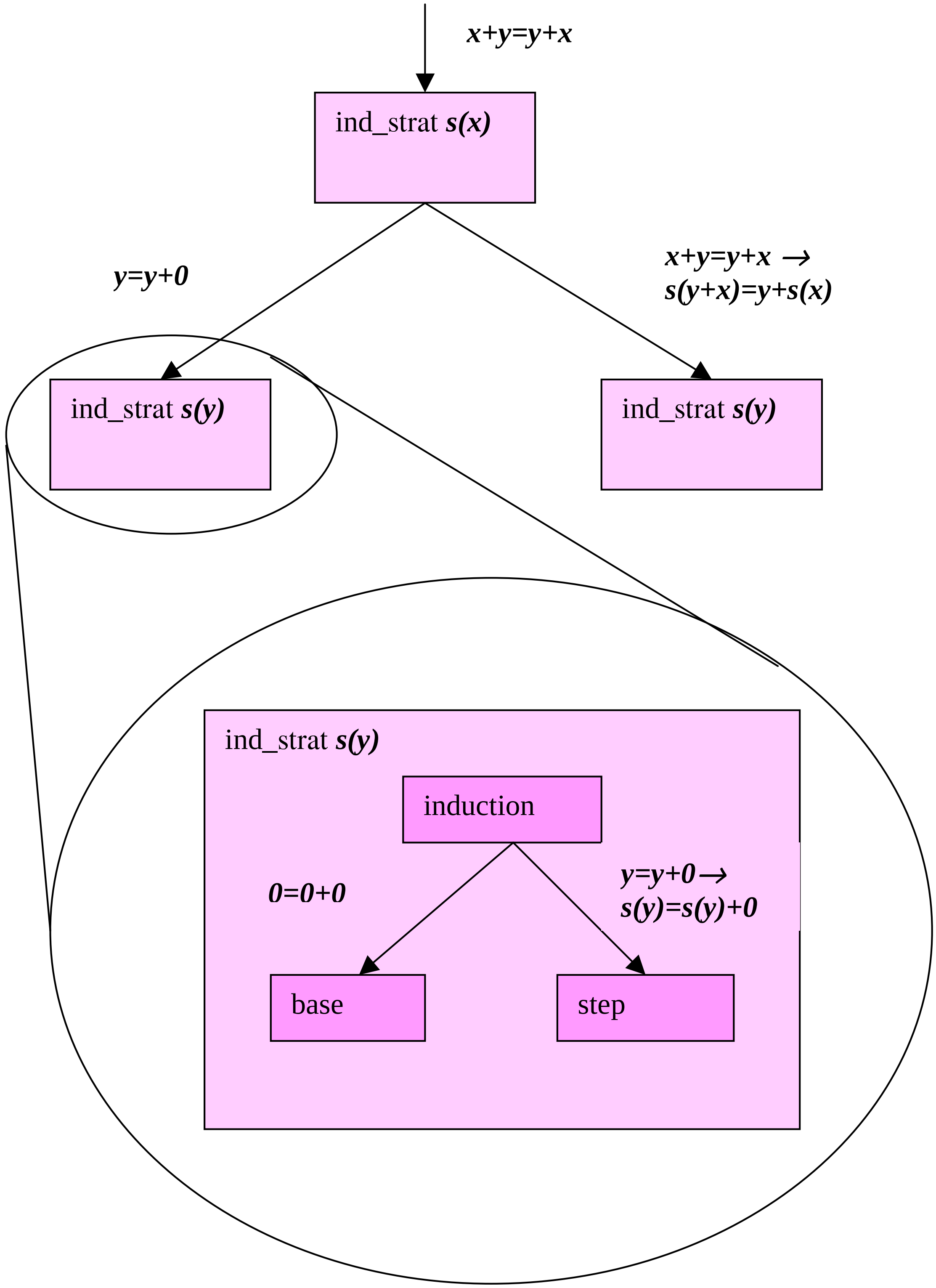
Preconditions:

Declarative: Rippling must be possible in step cases.

Procedural: Look-ahead to choose induction rule that will permit rippling.

Special-Purpose Proof Plan

Commutativity of +:



x+y=y+x

ind_strat ***s(x)***

y=y+0

x+y=y+x →
s(y+x)=y+s(x)

ind_strat ***s(y)***

ind_strat ***s(y)***

ind_strat ***s(y)***

induction

0=0+0

y=y+0 →
s(y)=s(y)+0

base

step

Empirical Success of Proof Planning

- Implemented in *Clam*/λ*Clam* and Ωmega proof planners.
- Successfully tested on a wide range domains: formal methods, mathematics, configuration, game playing, ...
- Solution of eureka problems using critics, *e.g.* lemma discovery and generalisation.
- Applications to software/hardware verification/synthesis.
- Ωmega linked to 3^{rd} party provers, CAS, constraint solvers, *etc.*

Advantages of Proof Planning

- Reduction in search: larger steps, fewer options.
- Multi-level proof explanation: supports interaction.
- Non-standard proof exploration,

  least-commitment devices:

  meta-variables and constraints.

  agent-based proof planning.
- Framework for inter-operating reasoners.

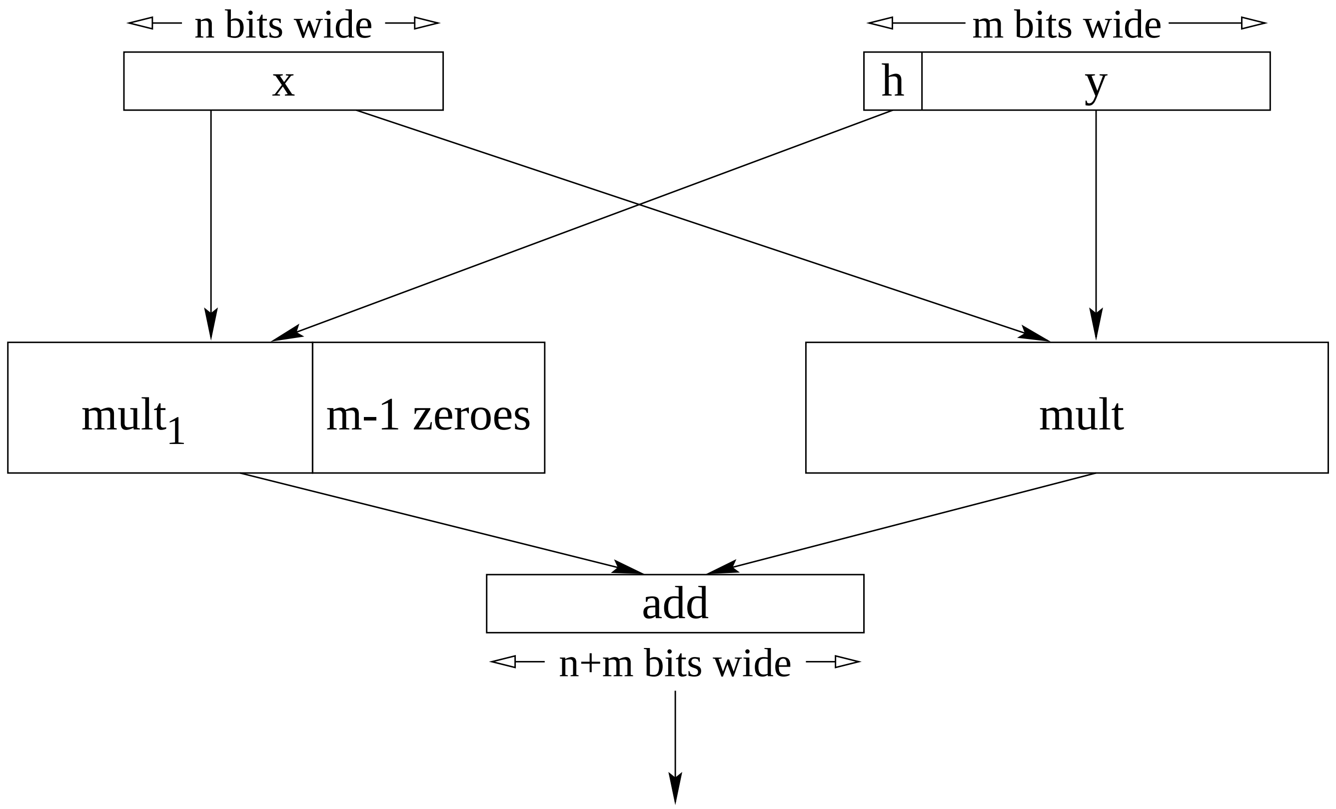
Hardware Verification

- Proof planning applied to hardware verification in PhD of Francisco Cantu,

  supervised by Alan Bundy, David Basin and Alan Smaill.
- Existing induction proof plans readily adapted.
- Significant sequential and combinational circuits verified automatically,

  e.g. multipliers, Gordon computer.
- Plans robust under minor modifications of specifications/implementations.

Parallel Multiplier Circuit



n bits wide

x

m bits wide

h

y

mult_1

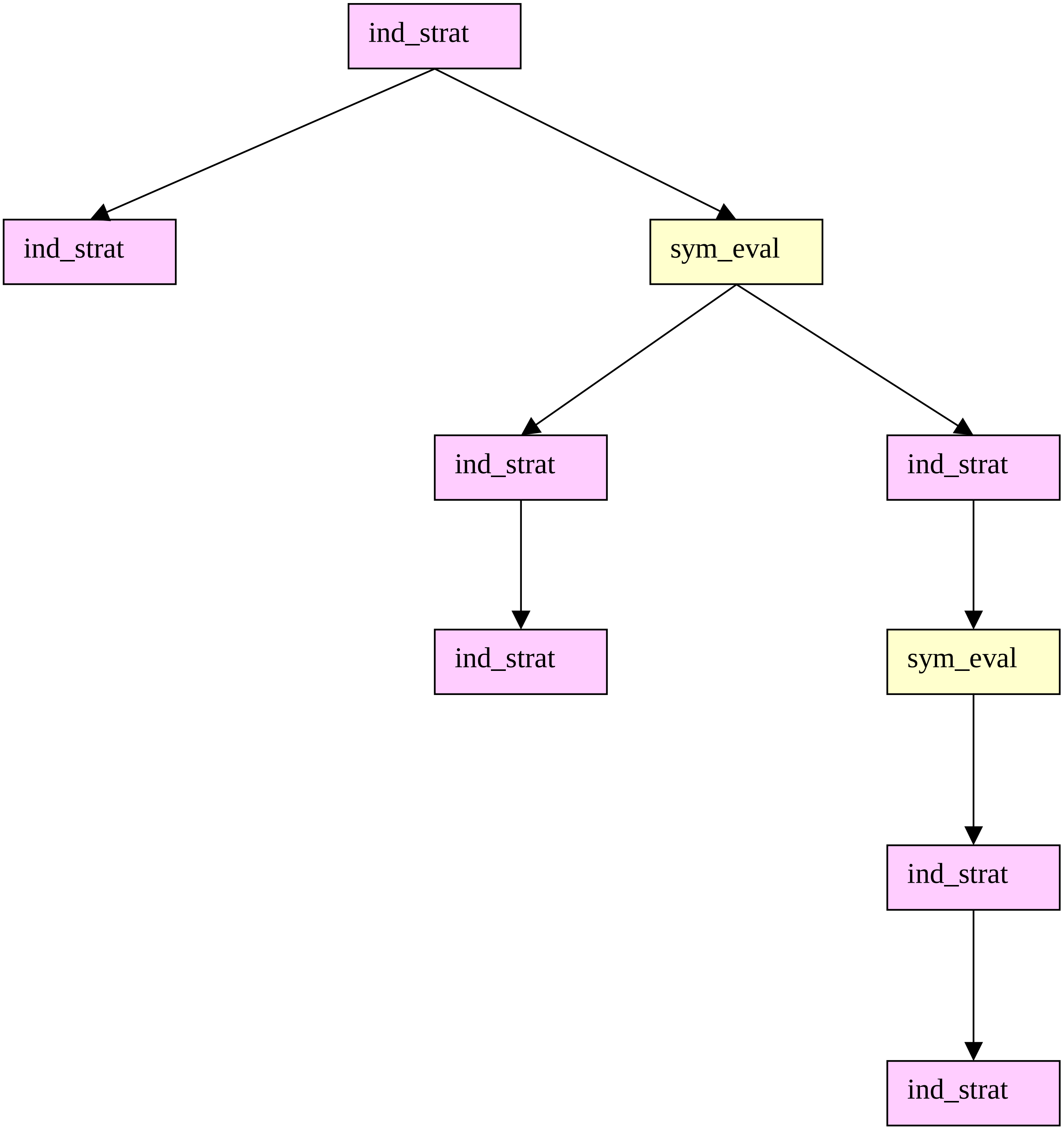
m-1 zeroes

mult

add

n+m bits wide

Parallel Multiplier: Proof Plan

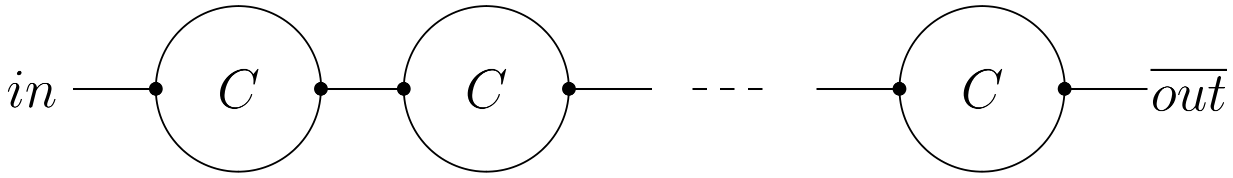


Distributed System Verification

- Proof planning applied to verification of CCS programs in PhD of Raúl Monroy,

  supervised by Alan Bundy, Andrew Ireland, Jane Hesketh and Ian Green
- Extends model checking by dealing with infinite-state, parameterised systems (VIPSs).
- CCS Expansion rule applicable repeatedly and to any proper sub-expression.
- Existing induction proof plans adapted and extended with new methods: Generalisation and UFI, equation.
- Significant VIPSs verified.

Linked Buffer



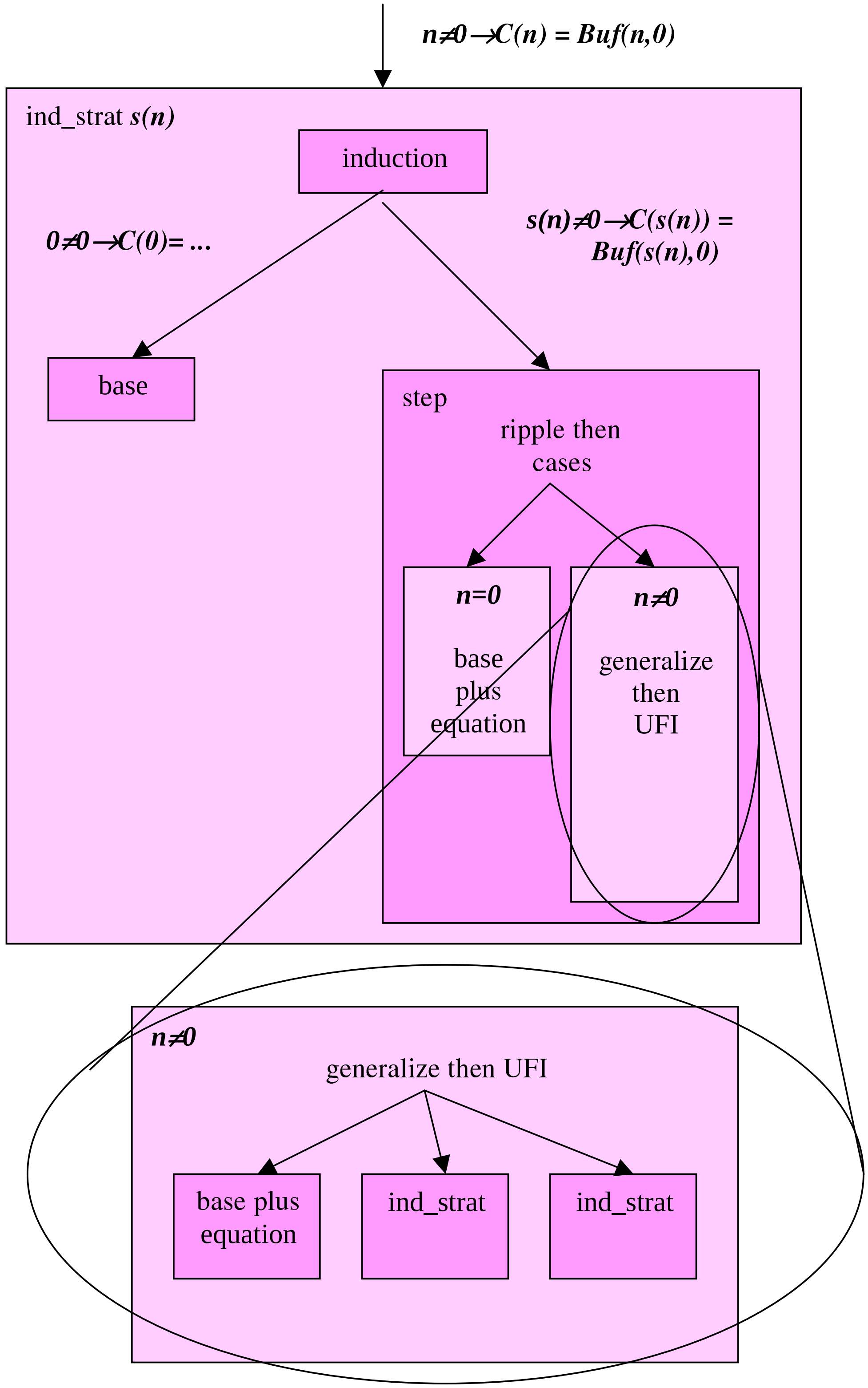
- Model

$$\begin{aligned} n = 0 \rightarrow \quad & C^{s(n)} \overset{\text{def}}{=} C \\ n \neq 0 \rightarrow \quad & C^{s(n)} \overset{\text{def}}{=} C \frown C^{(n)} \end{aligned}$$

- Specification

$$\begin{aligned} n \neq 0 \quad \rightarrow \quad & Buf_{\langle n,0\rangle} \overset{\text{def}}{=} in.Buf_{\langle n,s(0)\rangle} \\ n > s(k) \quad \rightarrow \quad & Buf_{\langle n,s(k)\rangle} \overset{\text{def}}{=} in.Buf_{\langle n,s(s(k))\rangle} + \overline{out}.Buf_{\langle n,k\rangle} \\ n = s(k) \quad \rightarrow \quad & Buf_{\langle n,s(k)\rangle} \overset{\text{def}}{=} \overline{out}.Buf_{\langle n,k\rangle} \end{aligned}$$

Linked Buffer: Proof Plan



$n \neq 0 \rightarrow C(n) = Buf(n,0)$

ind_strat $s(n)$

induction

$0 \neq 0 \rightarrow C(0) = \ldots$

$s(n) \neq 0 \rightarrow C(s(n)) =$
$Buf(s(n),0)$

base

step

ripple then
cases

$n=0$

base
plus
equation

$n \neq 0$

generalize
then
UFI

$n \neq 0$

generalize then UFI

base plus
equation

ind_strat

ind_strat

Logic Program Synthesis

- Proof planning applied to pure logic program synthesis in PhD of Ina Kraan,

  supervised by Alan Bundy, David Basin and Ian Green.
- Existing induction proof plans readily adapted.
- Verify program with unknown implementation: meta-variable P.

  $$spec(\overrightarrow{args}) \leftrightarrow P(\overrightarrow{args})$$

  Instantiate P via higher-order unification during proof.

  Proof plans help control expanded search.

  Called middle-out reasoning.
- Similar trick to discover appropriate induction – and hence recursion.
- Many logic programs synthesised from specifications.

Catch 22 in Deductive Synthesis of Recursion

- Induction causes infinite branching:
  induction rule for every well-ordering in every data-structure.
- Standard heuristics construct induction from recursive programs in conjecture.
- This fails for synthesis, since:
  - recursive structure of synthesised program dual to induction in proof,
  - do *not* want to be limited to recursion in *specification*.
- Middle-out reasoning technique allows induction to be *independent* of recursions in conjecture.

Proof Plans and Program Schemas

- Unified view of proof plans and program schemas,

  work by Julian Richardson (Edinburgh + Heriot Watt) and Pierre Flener (Uppsala).

- Deductive synthesis of programs via proofs, ensures correspondence.

- Schema guided programming via proof planning.
  - Proof method for each program schema, *e.g.* divide and conquer.
  - Provides legal and heuristic pre-conditions for use.
  - Constructs plan for any proof obligations, including synthesis proof.
  - Generates code via deductive synthesis.

Imperative Program Verification

- Proof planning applied to imperative program verification in PhD of Jamie Stark, supervised by Andrew Ireland (Heriot Watt).
- Existing induction proof plans adapted, with loop invariant approximating induction formula, and central role for rippling.
- Inductive proof critics adapted to loop invariant critic, failure of verification proof suggests revision of loop invariant, technique subsumes and extends many heuristics in literature.
- Many significant imperative programs verified, and applications to hardware suggested.

Example: Exponentiation

Program specification:

```
$\{x = \mathcal{X} \wedge y = \mathcal{Y}\}$
$r := 1;$
**while** $(y > 0)$ **do**
**begin**
    $r := r * x;$
    $y := y - 1$
**end**
$\{r = exp(\mathcal{X}, \mathcal{Y}) \wedge \neg(y > 0)\}$
```

where *exp* is defined by:

$$\begin{aligned} exp(X, 0) &= 1 \\ exp(X, Y + 1) &= exp(X, Y) * X \end{aligned}$$

Loop verification condition:

$$r = exp(\mathcal{X}, \mathcal{Y} - y) \wedge x = \mathcal{X} \wedge y > 0 \rightarrow \\ r * x = exp(\mathcal{X}, \mathcal{Y} - (y - 1))$$

A Loop Invariant Critic

- Simple default loop invariant assumed initially:

  $r = exp(\mathcal{X}, \mathcal{Y}) \wedge \neg(y > 0).$

- In step case, rippling is blocked:

  $\boxed{\boxed{r} * x}^{\uparrow} = exp(\mathcal{X}, \mathcal{Y}) \ldots .$

- Critic suggests generalisation:

  $r = exp(\mathcal{X}, \boldsymbol{F(\mathcal{Y}, r, x, y)}) \ldots$

- Middle-out reasoning instantiates F to produce:

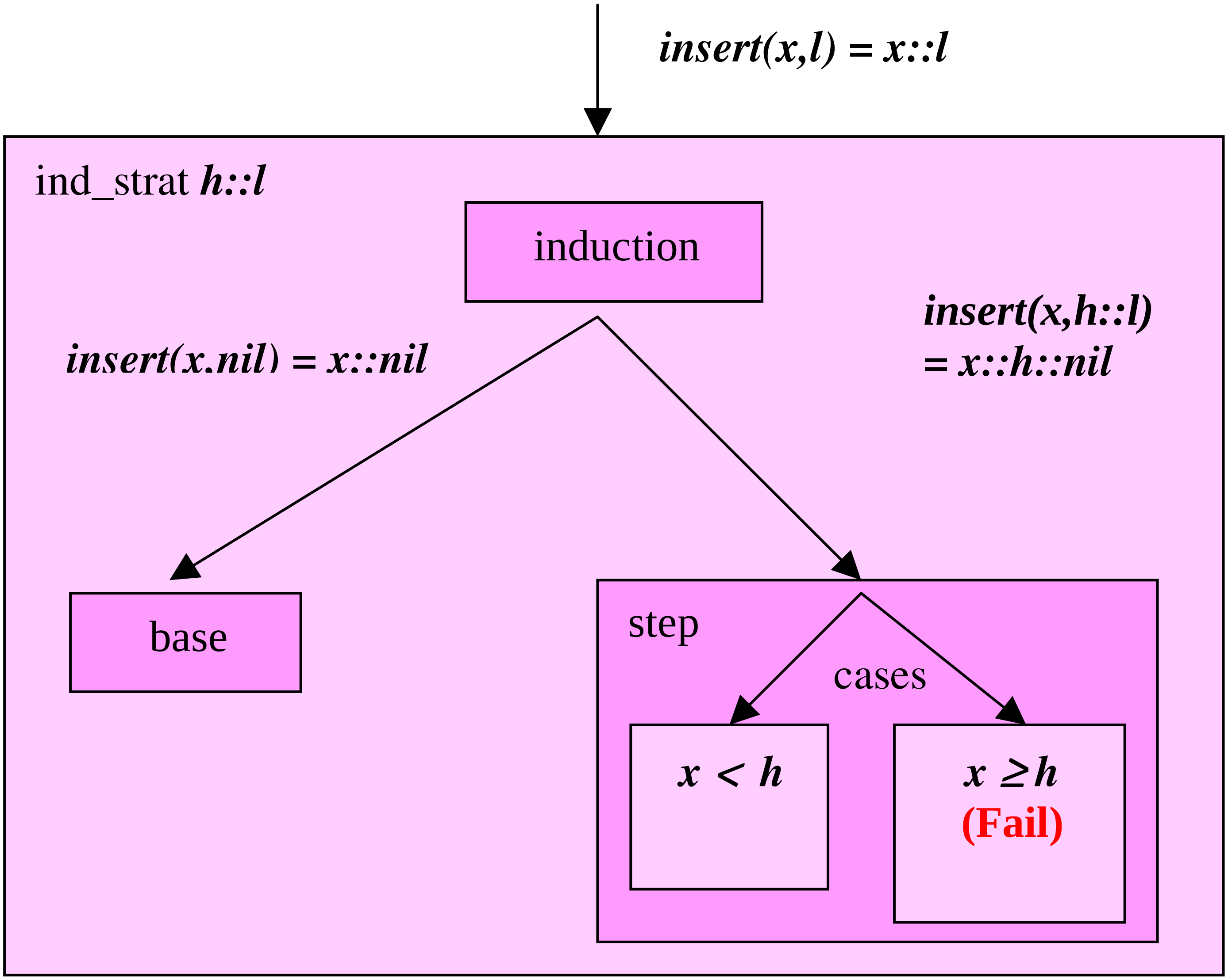
  $r = exp(\mathcal{X}, \boldsymbol{\mathcal{Y} - y}) \ldots,$

  as revised invariant, which succeeds.

Correction of Faulty Conjectures

- Proof planning applied to correction of faulty conjectures in MSc (and recent work) of Raúl Monroy,
  supervised by Alan Bundy.
- Most initial programs faulty, so verification conjectures faulty.
- Correct by inserting consistent, non-trivial conditions,
  may require defining new recursive predicates.
- Synthesise new condition from failed proof,
  proof plan identifies critical failures.
- Many interesting conjecture corrections synthesised.

Faulty Insert Program: Proof Plan



insert(x,l) = x::l

ind_strat ***h::l***

induction

insert(x.nil) = x::nil

insert(x,h::l)
= x::h::nil

base

step

cases

x < h

x ≥ h
(Fail)

Some Conjectures Corrected

Faulty Conjecture	Patch
$X > \mathrm{half}(X)$	$X \neq 0$
$\mathrm{insert}(X, L) = X :: L$	$\mathrm{P}_0(L, X)$
$\mathrm{sort}(\mathrm{app}(A, B)) = \mathrm{app}(\mathrm{sort}(A), \mathrm{sort}(B))$	$\mathrm{P}_{\mathrm{stap}}(A, B)$

where

- $\mathrm{P}_0(L, X) \leftrightarrow \forall Y \in L.\ X \leq Y.$
- $\mathrm{P}_{\mathrm{stap}}(A, B) \leftrightarrow \forall X \in A. \forall Y \in B.\ X \leq Y.$

Future Prospects and Challenges

- Application of proof planning to software engineering in the large,

  e.g. Grid application rapid (re-)assembly.

- Proof planning's role in the integration of reasoning systems:

  'brute-force' provers, CAS, constraint solvers, decision procedures, counter-example finders, conjecture makers, *etc.*

- Automatic discovery of new proof plans,

  e.g. via data mining proof corpora.

Conclusion

- Proof planning guides search in proof obligations, including ‘eureka’ steps, via middle-out reasoning and critics.
- Successfully applied in verification and synthesis of both hardware and software.

  Hard problems solved, *e.g.* Gordon Computer verification.
- Can extend state of the art in automation and lift level of interaction.