Towards Proof Planning for High Integrity Software

Andrew Ireland

Dependable Systems Group
Computing and Electrical Engineering
Heriot-Watt University
Edinburgh

Some Critical Challenges

- Increased proof automation within the context of domain specific specification and implementation languages.
- Better integration of conjecture disproving capabilities within the formal verification process.
- Higher levels of interaction when formal verification fails.
- Development of “smarter” software development environments through hidden theorem proving components.
- Effective techniques for managing change within the formal development of software.

Motivation and Context

- Importance of formal methods within the development safety critical systems recognized by Foresight Panel (1995).
- Scalability and usability of formal methods remain key challenges to researchers.
- Automation and high-level interaction will play an important role in meeting challenges.
- Focus here of theorem proving as an enabling technology for code level verification.

Executive Summary

- To investigate the role of Proof Planning within the SPARK approach to high integrity Ada.
- In collaboration with Praxis Critical Systems and funded by the EPSRC Critical Systems programme – GR/R24081.
- The Team:
  - Andrew Ireland and Julian Richardson
  - Post-Doc Research Associate
  - Bill Ellis (Research Student)

Executive Summary

- Deductive and strategic support for the incremental development of high integrity software – scalability & usability.
- Proof planning provides a framework for representing strategic as well as theorem proving knowledge – pay-off in terms of proof discovery and high-level interaction.
- Time is ripe to investigate the proof discovery capabilities on "industrial strength" problems – SPARK is an ideal vehicle.

Background

- Foundations:
  - Floyd-Hoare inductive assertion method (deductive).
  - Assertion generation methods (strategic).
- First generation systems (Stanford Pascal Verifier, GYPSY):
  - Batch oriented.
  - Unmanageable verification conditions (VCs).
  - Feedback on proof failures poor, *i.e.* incorrect code or inconsistent specification or missing lemma ... ?
- Second generation systems (PENELOPE, SPARK):
  - Incremental development
  - VCs kept relatively small
  - Proof failures easier to understand and relate to code, specification and application theory.

Overview

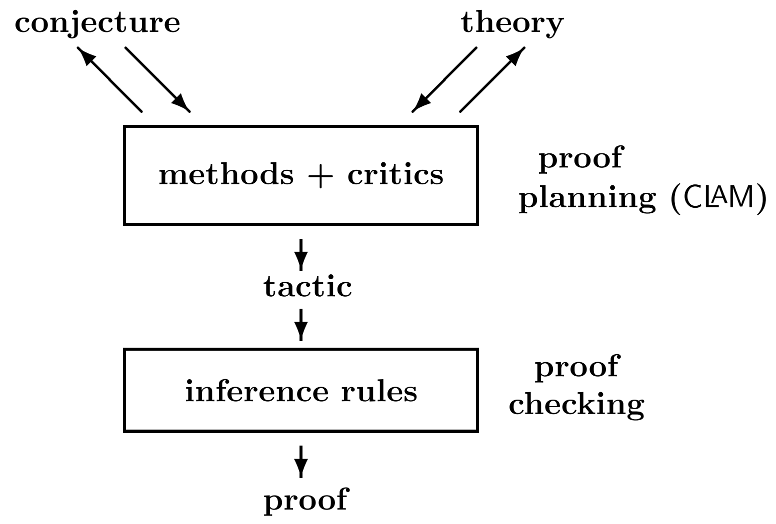
- Proof planning and proof discovery capabilities.
- Proposal to investigate these proof discovery capabilities within an industrial setting – SPARK Ada (Praxis Critical Systems) funded by EPSRC Critical Systems programme.
- The Team:
  - Andrew Ireland and Julian Richardson
  - Post-Doc Research Associate
  - Bill Ellis (Research Student)

Proof Plans

Proof plan = tactics + methods + critics

- **Proof tactics:** prescribe the structure of a proof at the level of the primitive rules of deduction.
- **Proof methods:** given a collection of general-purpose tactics, proof methods guide the construction of a special-purpose tactic for a given conjecture.
- **Proof critics:** when methods fail their associated critics suggest proof patches by analyzing partial success in terms of method preconditions and the overall proof plan instantiation.

Strategic-Deductive Separation



CLAM proof planner was developed within Alan Bundy's Mathematical Reasoning Group (University of Edinburgh).

Proof by Mathematical Induction

- Proof by mathematical induction is required if you want to reason about recursion, iteration, parameterized designs, ...
- Structural induction on lists:

  Conjecture: $\forall t : list(\tau).\ P(t)$

  gives rise to a base case:

  Goal: $P(nil)$

  and a step case:

  Given: $P(t)$

  Goal: $P(h :: t)$

Note *nil* denotes the empty list and :: is the list constructor.

A List Induction Example

- Conjecture:

$$\forall t : list(\tau).\ rot(len(t), t) = t$$

- Generalized conjecture:

$$\forall t : list(\tau).\forall l : list(\tau).\ rot(len(t), t @ l) = l @ t$$

- Base case:

  Goal: $rot(len(nil), nil @ l) = l @ nil$

- Step case:

  Given: $\forall l' : list(\tau).\ rot(len(t), t @ l') = l' @ t$

  Goal: $rot(len(h :: t), (h :: t) @ l) = l @ (h :: t)$

Definitions and Properties

$$\begin{array}{rcl}
len(nil) & = & 0 \\
len(X :: Y) & = & 1 + len(Y) \\
nil @ Z & = & Z \\
(X :: Y) @ Z & = & X :: (Y @ Z) \\
rot(0, Z) & = & Z \\
rot(1 + X, nil) & = & nil \\
rot(1 + X, Y :: Z) & = & rot(X, Z @ (Y :: nil)) \\
(X @ Y) @ Z & = & X @ (Y @ Z) \\
X @ (Y :: Z) & = & (X @ Y :: nil) @ Z
\end{array}$$

Step-case: Deductive Perspective

- **Given:**

$$\forall l' : list(\tau).\ rot(len(t), t \mathbin{@} l') = l' \mathbin{@} t \tag{1}$$

- **Goal:**

$$\begin{aligned}
rot(\underline{len(h :: t)}, \underline{(h :: t) \mathbin{@} l}) &= \underline{l \mathbin{@} (h :: t)} \\
\underline{rot(1 + len(t), h :: (t \mathbin{@} l))} &= (l \mathbin{@} h :: nil) \mathbin{@} t \\
rot(len(t), \underline{(t \mathbin{@} l) \mathbin{@} h :: nil}) &= (l \mathbin{@} h :: nil) \mathbin{@} t \\
rot(len(t), t \mathbin{@} (l \mathbin{@} h :: nil)) &= (l \mathbin{@} h :: nil) \mathbin{@} t
\end{aligned}$$

Match with (1), l' becomes $l \mathbin{@} h :: nil$

Step-case: Strategic Perspective

- **Given:**

$$\forall l' : list(\tau).\ rot(len(t), t \mathbin{@} l') = l' \mathbin{@} t$$

- **Goal:**

$$\left.rot(len(\boxed{h :: \underline{t}}^{\uparrow}), \boxed{h :: \underline{t}}^{\uparrow} \mathbin{@} \lfloor l \rfloor) = \lfloor l \rfloor \mathbin{@} \boxed{h :: \underline{t}}^{\uparrow}\right\}\ \text{identify mismatch}$$

$$\left.\begin{aligned}
rot(\boxed{1 + \underline{len(t)}}^{\uparrow}, \boxed{h :: \underline{t \mathbin{@} \lfloor l \rfloor}}^{\uparrow}) &= \left\lfloor \boxed{\underline{l} \mathbin{@} h :: nil}^{\downarrow} \right\rfloor \mathbin{@} t \\
rot(len(t), \boxed{\underline{t \mathbin{@} \lfloor l \rfloor} \mathbin{@} h :: nil}^{\downarrow}) &= \left\lfloor \boxed{\underline{l} \mathbin{@} h :: nil}^{\downarrow} \right\rfloor \mathbin{@} t \\
rot(len(t), t \mathbin{@} \left\lfloor \boxed{\underline{l} \mathbin{@} h :: nil}^{\downarrow} \right\rfloor) &= \left\lfloor \boxed{\underline{l} \mathbin{@} h :: nil}^{\downarrow} \right\rfloor \mathbin{@} t
\end{aligned}\right\}\ \text{reduce mismatch}$$

$$\left.rot(len(t), t \mathbin{@} (l \mathbin{@} h :: nil)) = (l \mathbin{@} h :: nil) \mathbin{@} t\right\}\ \text{exploit match}$$

Note: $\lfloor \ldots \rfloor$ delimit position of match variables.

Strategic vs Deductive

- **Strategic:** proof search guided by annotated rewrite rules:

$$
\begin{array}{rcl}
len(\boxed{X :: \underline{Y}}^{\uparrow}) & \Rightarrow & \boxed{1 + \underline{len(Y)}}^{\uparrow} \\
rot(\boxed{1 + \underline{X}}^{\uparrow}, \boxed{Y :: \underline{Z}}^{\uparrow}) & \Rightarrow & rot(X, \boxed{\underline{Z} \;@\; (Y :: nil)}^{\downarrow}) \\
\boxed{\underline{(X \;@\; Y) \;@\; Z}}^{\downarrow} & \Rightarrow & X \;@\; \boxed{(\underline{Y} \;@\; Z)}^{\downarrow}
\end{array}
$$

- **Deductive:** proof checking involves rewriting (no search):

$$
\begin{array}{rcl}
len(X :: Y) & \Rightarrow & 1 + len(Y) \\
rot(1 + X, Y :: Z) & \Rightarrow & rot(X, Z \;@\; (Y :: nil)) \\
(X \;@\; Y) \;@\; Z & \Rightarrow & X \;@\; (Y \;@\; Z)
\end{array}
$$

Note: rewriting + meta-data = ripple proof strategy.

Patterns of Success

Rippling-out:

$$
f_1(\ldots, f_n(\boxed{c_1(\underline{\ldots})}^{\uparrow}), \ldots) \qquad \boxed{c_n(\underline{f_1(\ldots, f_n(\ldots), \ldots)})}^{\uparrow}
$$

before **after**

Rippling-sideways:

$$
f_1(\boxed{c_1(\underline{\ldots})}^{\uparrow}, f_n(\ldots)) \qquad f_1(\ldots, \boxed{c_n(\underline{f_n(\ldots)})}^{\downarrow})
$$

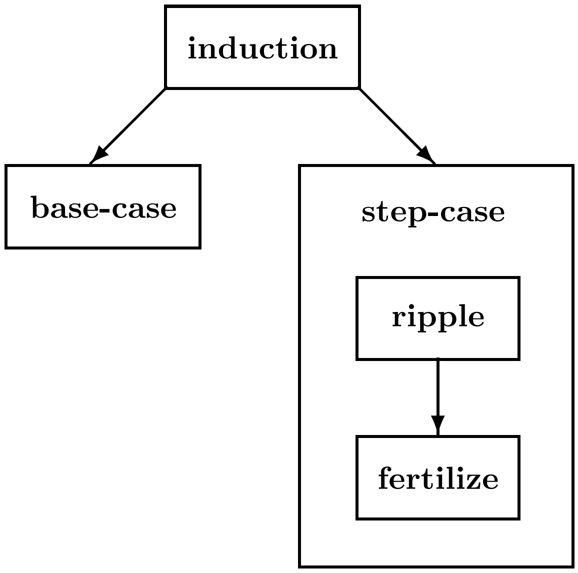
before **after**

Rippling-in:

$$
\boxed{c_n(\underline{f_1(\ldots, f_n(\ldots), \ldots)})}^{\downarrow} \qquad f_1(\ldots, f_n(\boxed{c_1(\underline{\ldots})}^{\downarrow}), \ldots)
$$

before **after**

A Proof Plan for Induction



General-purpose Tactics for Induction

- `base_case(Base):`

  $\quad$ `Base` $\equiv$

  `(...eval_def(Pos, Rule)...) then elementary`

- `step_case(Step):`

  `Step` $\equiv$ `ripple(Waves) then fertilize(IndHyp)`

  `Waves` $\equiv$ `...wave(Pos, [Rule, Dir])...`

- `induction(IndRule, Var:Typ)`

A Special-Purpose Tactic for Induction

- **Conjecture:**

$$\forall t : list(nat).\ rot(len(t), t) = t$$

- **Composite tactic** (generated automatically)**:**

```
generalize(...) then
   induction([v0::t],[t:nat list]) then
      [base_case(... then
             induction([v1::l],[l:nat list]) then
                [base_case(... then
                    elementary(...)),
                 step_case(ripple(...) then
                              fertilize(...) then
                                 elementary(...))]),
       step_case(ripple(...) then
                    fertilize(...))]
```


Playing Devil's Advocate

Attack: Why have a separate planning level? Why not use "smart-tactics" that have the heuristic strategies built-in?

Defence: Having a loose coupling between **strategic** (planning) and **deductive** (checking) levels gives three key benefits:

- Supports greater **flexibility** in the use of strategies than a conventional script or tactic-based environment.
- Promotes the **reusability** of strategies across different application domains and logics.
- Enables **synergy** between strategies developed within related disciplines to be exploited.

Automatic Conjecture Generalization

- **Given:**

$$rot(len(t), t) = t$$

- **Goal:**

$$rot(len(\boxed{h :: \underline{t}}^{\uparrow}), \boxed{h :: \underline{t}}^{\uparrow}) = \boxed{h :: \underline{t}}^{\uparrow}$$

$$\underbrace{rot(\boxed{1 + \underline{len(t)}}^{\uparrow}, \boxed{h :: \underline{t}}^{\uparrow})}_{\textbf{blocked}} = \boxed{h :: \underline{t}}^{\uparrow}$$

- Ripple strategy suggests patching the goal via the introduction of a new universally quantified variable, *i.e.* an accumulator variable.

Patch: Plan Guides Goal Refinement

- **Given:**

$$\forall l' : list(\tau).\ rot(len(t), F_1(t, l')) = G_1(t, l')$$

- **Goal:**

$$rot(len(\boxed{h :: \underline{t}}^{\uparrow}), F_1(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor l \rfloor)) = G_1(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor l \rfloor) \quad \Big\} \text{ identify mismatch}$$

$$\left.\begin{aligned} rot(\boxed{1 + \underline{len(t)}}^{\uparrow}, \boxed{h :: t \mathbin{@} \underline{F_2(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor l \rfloor)}}^{\uparrow}) &= \dots \\ rot(len(t), \boxed{\underline{t \mathbin{@} F_2(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor l \rfloor)} \mathbin{@} h :: nil}^{\downarrow}) &= \dots \\ rot(len(t), t \mathbin{@} \boxed{\underline{F_2(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor l \rfloor)} \mathbin{@} h :: nil}^{\downarrow}) &= \dots \end{aligned}\right\} \text{reduce mismatch}$$

Note: $F_1, G_1, \ldots$ denote second-order meta variables.

Patch: Plan Guides Goal Refinement

- **Given:**

$$\forall l' : list(\tau).\ rot(len(t), t \ @\ F_2(t, l')) = G_1(t, l')$$

- **Goal:**

$$\left.\begin{array}{lcl} \ldots & = & \ldots \\ \ldots & = & G_1(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor l \rfloor) \\ \ldots & = & \boxed{\underline{G_2(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor l \rfloor)} \ @\ h :: nil}^{\downarrow} \ @\ t \end{array}\right\} \text{reduce mismatch}$$

$$rot(len(t), t \ @\ (l \ @\ h :: nil)) \ = \ (l \ @\ h :: nil) \ @\ t \quad \Big\} \text{coerce match}$$

$F_1 \equiv \lambda x.\lambda y.\ x \ @\ F_2(x, y)$ $\quad$ $G_1 \equiv \lambda x.\lambda y.\ G_2(x, y) \ @\ x$
$F_2 \equiv \lambda x.\lambda y.\ y$ $\quad$ $G_2 \equiv \lambda x.\lambda y.\ y$

Automatic Lemma Discovery

- In order to prove:

$$\forall t : list(\tau).\forall l : list(\tau).\ rot(len(t), t \ @\ l) = l \ @\ t$$

- We need to synthesize and prove:

$$\begin{array}{rcl} X \ @\ (Y \ @\ Z) & = & (X \ @\ Y) \ @\ Z \\ X \ @\ (Y :: Z) & = & (X \ @\ Y :: nil) \ @\ Z \end{array}$$

CLAM generates such lemmata dynamically via proof critics.

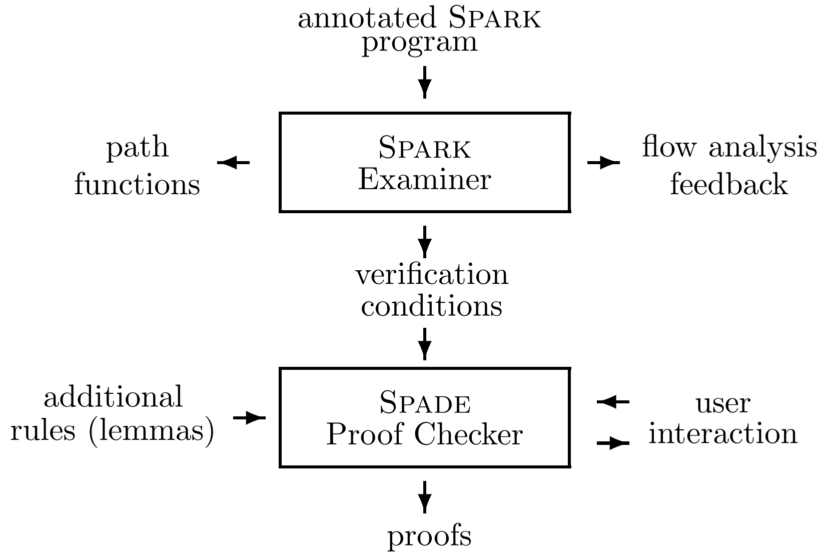
Proof Critics: Flexibility and Reusability

- Conjecture generalization
- Lemma discovery
- Induction revision
- Case analyze
- Replacing constants
- Adding new conjuncts
- Tail-invariants
- Range splitting

The SPARK Approach to High Integrity Software

- SPARK is an Ada subset designed for high integrity applications (Praxis Critical Systems).
- More than just a programming language – an approach to the production of high integrity software.
- Reflected in the SPARK tools that support flow analysis (SPARK Examiner) as well as formal verification (SPADE Proof Checker).

SPARK Examiner and SPADE Proof Checker



Based upon Figure 3.1 from (Barnes, 1995)

Research Proposal

- Research hypothesis:

  The coupling of CLAM *with the* SPADE *Proof Checker will significantly reduce the amount and sophistication of user interaction that is required in order to complete the formal verification of software written in* SPARK*.*

- Goals and challenges:
  - Proof discovery: reduce the number of lemmas (rules) and intermediate assertions that a SPADE user will typically be required to provide.
  - Proof management: improve the productivity of a SPARK programmer both at the level of coding as well as formal proof – interactive proof (program) critics, code reuse?

Summary

- Deductive and strategic support for the incremental development of high integrity software – scalability & usability.
- Proof planning provides a framework for representing strategic as well as theorem proving knowledge – pay-off in terms of proof discovery and high-level interaction.
- Time is ripe to investigate the proof discovery capabilities on "industrial strength" problems – SPARK is an ideal vehicle.

Dijkstra's Dutch Flag Problem

$\{(\forall i : 0 \leq i < \mathcal{N} : color(a[i]) = red \lor$
$\quad color(a[i]) = white \lor color(a[i]) = blue)\}$

$r := 0;$
while $(r < m)$ **do**
  if $(color(a[m-1]) = white)$ **then** $m := m - 1;$
  else if $(color(a[m-1]) = red)$ **then**
    begin
      $t := a[m-1]; a[m-1] := a[r]; a[r] := t; r := r + 1$
    end
  else begin
      $m := m - 1; w := w - 1; t := a[m]; a[m] := a[w]; a[r] := t$
    end

$\{(0 \leq r) \land (r \leq w) \land (w \leq \mathcal{N}) \land (\forall i : 0 \leq i < r : color(a[i]) = red) \land$
$\quad (\forall i : r \leq i < w : color(a[i]) = white) \land$
$\quad (\forall i : w \leq i < \mathcal{N} : color(a[i]) = blue)\}$

Dutch Flag: Loop Invariant Discovery

- **Given postcondition:**

$$
\begin{aligned}
&(0 \le r) \wedge (r \le w) \wedge (w \le \mathcal{N}) \wedge \\
&(\forall i : 0 \le i < r : color(a[i]) = red) \wedge \\
&(\forall i : r \le i < w : color(a[i]) = white) \wedge \\
&(\forall i : w \le i < \mathcal{N} : color(a[i]) = blue)
\end{aligned}
$$

- **Invariant discovered:**

$$
\begin{aligned}
&(0 \le r) \wedge (r \le m) \wedge (m \le w) \wedge (w \le \mathcal{N}) \wedge \\
&(\forall i : 0 \le i < r : color(a[i]) = red) \wedge \\
&(\forall i : m \le i < w : color(a[i]) = white) \wedge \\
&(\forall i : w \le i < \mathcal{N} : color(a[i]) = blue)
\end{aligned}
$$