Tinker, Tailor, Solver, Proof

Writing Graphical Proof Strategies in Tinker

Gudmund Grov
Yuhui Lin

Aleks Kissinger

funded by EPSRC grants: EP/H023852, EP/H024204 and EP/J001058, the John Templeton Foundation and the Office of Naval Research
Here's the plan:

**Theory: Proof-strategy graphs**

- Based on string diagrams, used in e.g. category theory and physics
- Evaluation by diagram rewriting

**Tool**

- Run as tactic within TP
- Evaluation:
- Implemented for ProofPower
Here's the plan:

Theory

- Based on e.g. category theory and physics
- Evaluation by

Tool: Tinker

- Run as tactic within TP
- Evaluation: automatic OR interactive
- Implemented for Isabelle and ProofPower
Stack-based strategies

LCF-style provers operate on open goals using tactics:

\[ t : \text{goal} \rightarrow [\text{goal}] \]

...and use stack based goal propagation:

pop first goal
apply tactic
push new sub-goal(s)
Stack-based strategies

Proof strategies are built from tactics using **tactical** combinators:

\[
\text{t}_1\ \text{THEN}\ \text{t}_2 \quad \text{t}_1\ \text{OR}\ \text{t}_2 \quad \text{REPEAT}\ \text{t}
\]
Stack-based strategies

\[ \text{tac mytac} := t_1 \text{ THEN } t_2 \text{ THEN } t_2 \text{ THEN } t_3 \]

\[ \text{mytac}(g) := \]

\[ | \]

\[ \]
Stack-based strategies

tac mytac := \( t_1 \) THEN \( t_2 \) THEN \( t_2 \) THEN \( t_3 \)

mytac(g) :=
Stack-based strategies

\[ \text{tac mytac} := t_1 \text{ THEN } t_2 \text{ THEN } t_2 \text{ THEN } t_3 \]

\[ \text{mytac}(g) := t_1 \]

Diagram:

```
    t_1
     /
    / \n   /   
  /     
 t_2  t_2
     
```
Stack-based strategies

\[
\text{tac mytac} := t_1 \text{ THEN } \underbrace{t_2 \text{ THEN } t_2 \text{ THEN } t_3}
\]

\[
\text{mytac}(g) :=
\]

\[
\begin{array}{c}
\text{t}_1 \\
\end{array}
\]
Stack-based strategies

tag \text{mytac} := t_1 \text{ THEN } t_2 \text{ THEN } t_2 \text{ THEN } t_3

\text{mytac}(g) := t_1
Stack-based strategies

\[\text{tac} \ \text{mytac} := t_1 \ \text{THEN} \ t_2 \ \text{THEN} \ \overbrace{t_2} \ \text{THEN} \ t_3\]

\[\text{mytac}(g) :=\]

\[t_1\]

\[t_2\]

\[t_2\]
Stack-based strategies

tac mytac := t₁ THEN t₂ THEN t₂ THEN t₃

mytac(g) :=

\[
\begin{array}{c}
\text{t₁} \\
\text{t₂} \\
\text{t₂}
\end{array}
\]
Stack-based strategies

tac mytac := t₁ THEN t₂ THEN t₂ THEN \( t₃ \)

mytac(g) :=

Diagram:

```
  t₁
  /
 t₂ ── t₂
  
  \( t₃ \)
```
Stack-based strategies

tac mytac := t₁ THEN t₂ THEN t₂ THEN t₃

mytac(g) :=

\[
\begin{tikzpicture}
  \node (t1) at (0,0) {$t_1$};
  \node (t2) at (-1,-1) {$t_2$};
  \node (t3) at (1,-1) {$t_3$};
  \path (t1) -- (t2);
  \path (t1) -- (t3);
\end{tikzpicture}
\]
But sometimes it goes wrong....

Suppose we replace $t_1$ with the “improved” tactic $t_x$
Tactic based proving

\[ \text{tac mytac} := t_x \text{ THEN } t_2 \text{ THEN } t_2 \text{ THEN } t_3 \]

\[ \text{mytac}(g) := \]
Tactic based proving

\( \text{tac mytac := } t_x \text{ THEN } t_2 \text{ THEN } t_2 \text{ THEN } t_3 \)

\( \text{mytac}(g) := \)
Tactic based proving

\texttt{tac mytac := } t_x \text{ THEN } t_2 \text{ THEN } t_2 \text{ THEN } t_3

\texttt{mytac(g) := } t_x
Tactic based proving

tac mytac := $t_x$ THEN $t_2$ \(\uparrow\) THEN $t_2$ THEN $t_3$

mytac(g) := $t_x$
Tactic based proving

tac mytac := $t_x$ THEN $t_2$ THEN $t_2$ THEN $t_3$

mytac(g) :=

```
  tx
 /|
/  |
 t2
```
Tactic based proving

\[ \text{tac mytac} := t_x \text{ THEN } t_2 \text{ THEN } \frac{t_2}{t_2} \text{ THEN } t_3 \]

\[ \text{mytac}(g) := \]

![Diagram showing tactic application and proof steps]
Tactic based proving

tac mytac := \( t_x \) THEN \( t_2 \) THEN \( t_2 \) THEN \( t_3 \)

mytac(g) :=
Debugging

where did it go wrong?

tac mytac := t_x THEN t_2 THEN t_2 THEN t_3
Debugging
where did it go wrong?

tac mytac := t_x THEN t_2 THEN \( \frac{t_2}{\text{error}} \) THEN t_3
Debugging
where did it go wrong?

tac  mytac := t_x  THEN  t_2  THEN  t_2  THEN  t_3

actual error

error
Debugging
where did it go wrong?

tac mytac := $t_x$ THEN $t_2$ THEN $t_2$ THEN $t_3$

or here

error
t_2 may also succeed here creating unexpected sub-goals
Bugs may be easy to spot for this example, but what if...
fun z_basic_prove_tac (thms: THM list) : TACTIC = (  
  TRY_T all_var_elim_asm_tac THEN  
  DROP_ASMS_T (MAP_EVERY (strip_asm_tac o  
    (fn thm => rewrite_rule thms thm  
      handle (Fail _) => thm)) o rev) THEN  
  TRY_T (rewrite_tac thms) THEN  
  REPEAT strip_tac THEN  
  TRY_T all_var_elim_asm_tac THEN_TRY  
  (z_quantifiers_elim_tac THEN  
    (fn gl => let val ciz = set_check_is_z false;  
      val res = (EXTEND_PC_T1 "mmp1" all_asm_fc_tac[] THEN  
        (basic_res_tac2 3 [eq_refl_thm]  
          ORELSE_T basic_res_tac3 3 [eq_refl_thm])) gl;  
        val _ = set_check_is_z ciz; in res end))));

fun z_basic_prove_tac (thms: THM list) : TACTIC = (  TRY_T all_var_elim_asm_tac THEN  DROP_ASMS_T (MAP_EVERY (strip_asm_tac o  (fn thm => rewrite_rule thms thm  handle (Fail _) => thm)) o rev) THEN  (TRY_T (rewrite_tac thms)) THEN  REPEAT strip_tac THEN  TRY_T all_var_elim_asm_tac THEN_TRY  (z_quantifiers_elim_tac THEN  (fn gl => let val ciz = set_check_is_z false;  val res = (EXTEND_PC_T1 "mmp1" all_asm_fc_tac[] THEN  (basic_res_tac2 3 [eq_refl_thm]  ORELS_T basic_res_tac3 3 [eq_refl_thm])  val _ = set_check_is_z ciz; in res end)));  error
fun z_basic_prove_tac (thms: THM list) : TACTIC = (
    TRY_T all_var_elim_asm_tac THEN
    DROP_ASM (MAP_EVERY (strip_asm_tac o
        (fn thm => rewrite_rule thms thm
            handle (Fail _) => thm)) o rev) THEN
    (TRY_T (rewrite_tac thms)) THEN
    REPEAT strip_tac THEN
    TRY_T all_var_elim_asm_tac THEN_TRY
    (z_quantifiers_elim_tac THEN
        (fn gl => let
            val ciz = set_check_is_z false;
            val res = (EXTEND_PC_T1 "mmp1" all_asm_fc_tac[] THEN
                (basic_res_tac2 3 [eq_refl_thm]
                    ORELSE_T basic_res_tac3 3 [eq_refl_thm]
                val _ = set_check_is_z ciz; in res end))))
)
(in thm => rewrite_rule thms thm
    handle (Fail _) => thm)) o rev) THEN
(TRY_T (rewrite_tac thms)) THEN
REPEAT strip_tac THEN
TRY_T all_var_elim_asm_tac THEN_TRY
(z_quantifiers_elim_tac THEN
(fn gl => let
    val ciz = set_check_is_z false;
    val res = (EXTEND_PC_T1 "mmp1" all_asm_fc_tac[] THEN
        (basic_res_tac2 3 [eq_refl_thm]
           ORELSE_T basic_res_tac3 3 [eq_refl_thm])) gl;
    val _ = set_check_is_z ciz; in res end
    )))

error
fun z_basic_prove_tac (thms: THM list) : TACTIC =
    TRY_T all_var_elim_asm_tac THEN
    DROP_ASMS_T (MAP_EVERY (strip_asm_tac o
        (fn thm => rewrite_rule thms thm
            handle (Fail _) => thm)) o rev) THEN
    TRY_T (rewrite_tac thms) THEN
    REPEAT strip_tac THEN
    TRY_T all_var_elim_asm_tac THEN_TRY
    (z_quantifiers_elim_tac THEN
        (fn gl => let
            val ciz = set_check_is_z false;
            (basic_res_tac2 3 [eq_refl_thm]
                ORELSE_T basic_res_tac3 3 [eq_refl_thm])) gl;
            val _ = set_check_is_z ciz; in res end
        (fn thm => rewrite_rule thms thm
            handle (Fail _) => thm)) o rev) THEN
    TRY_T (rewrite_tac thms) THEN
    REPEAT strip_tac THEN
    TRY_T all_var_elim_asm_tac THEN_TRY
    (z_quantifiers_elim_tac THEN
        (fn gl => let
            val ciz = set_check_is_z false;
            (basic_res_tac2 3 [eq_refl_thm]
                ORELSE_T basic_res_tac3 3 [eq_refl_thm])) gl;
            val _ = set_check_is_z ciz; in res end
        (fn thm => rewrite_rule thms thm
            handle (Fail _) => thm)) o rev) THEN
    (TRY_T (rewrite_tac thms)) THEN
    REPEAT strip_tac THEN
    TRY_T all_var_elim_asm_tac THEN_TRY
    (z_quantifiers_elim_tac THEN
        (fn gl => let
            val ciz = set_check_is_z false;
            (basic_res_tac2 3 [eq_refl_thm]
                ORELSE_T basic_res_tac3 3 [eq_refl_thm])) gl;
            val _ = set_check_is_z ciz; in res end
        (fn thm => rewrite_rule thms thm
            handle (Fail _) => thm)) o rev) THEN
    (TRY_T (rewrite_tac thms)) THEN
    REPEAT strip_tac THEN
fun _basic_prove_tac (thms: THM list) : TACTIC = (
    TRY_T all_var_elim_asm_tac THEN
    DROP_ASMS_T (MAP_EVERY (strip_asm_tac o
        (fn thm => rewrite_rule thms thm
            handle (Fail _) => thm)) o rev) THEN
    (TRY_T (rewrite_tac thms)) THEN
    REPEAT strip_tac THEN
    TRY_T all_var_elim_asm_tac THEN_TRY
    (z_quantifiers_elim_tac THEN
        (fn gl => let
            val ciz = set_check_is_z false
            (basic_res_tac2 3 [eq_refl_thm]
                ORELSE_T basic_res_tac3 3 [eq_refl_thm])) gl;
            val _ = set_check_is_z ciz; in res end
        (fn thm => rewrite_rule thms thm
            handle (Fail _) => thm)) o rev) THEN
    (TRY_T (rewrite_tac thms)) THEN
    REPEAT strip_tac THEN
    TRY_T all_var_elim_asm_tac THEN_TRY
    (z_quantifiers_elim_tac THEN
        (fn gl => let
            val ciz = set_check_is_z false;
            (basic_res_tac2 3 [eq_refl_thm]
                ORELSE_T basic_res_tac3 3 [eq_refl_thm])) gl;
            val _ = set_check_is_z ciz; in res end
        (fn thm => rewrite_rule thms thm
            handle (Fail _) => thm)) o rev) THEN
    (TRY_T (rewrite_tac thms)) THEN
    REPEAT strip_tac THEN
Instead of...

```plaintext
TRY_T all_var_elim_asm_tac THEN
DROP_ASMS_T (MAP_EVERY (strip_asm_tac o
  (fn thm => rewrite_rule thms thm
   handle (Fail _) => thm)) o rev) THEN
(TRY_T (rewrite_tac thms)) THEN
REPEAT strip_tac THEN
TRY_T all_var_elim_asm_tac THEN_TRY
(z_quantifiers_elim_tac THEN
(fn gl => let _val ciz = set_check_is_z false;
  val res = (EXTEND_PC_T1 '"mmp1" all_asm_fc_tac[]
    THEN (basic_res_tac2 3 [eq_refl_thm]
    ORELSE_T basic_res_tac3 3 [eq_refl_thm])) gl;
  val _ = set_check_is_z ciz; in res end)));```

... think of a proof strategy as a pipe network
Pipes connect tactics

The type of pipe used ensures correct composition

a tactic

another tactic
Loops

Repetition is simply a feedback pipe
Passing goals

Goals are **passed** to the next tactic using the **pipe**

A goal must **fit** in the pipe it is in.
Passing goals

Multiple goals can pass down the same pipe during the course of evaluation

abstracts over goal number and order
Hierarchies

Networks can be **structured** so a tactic can itself be a pipe network.
String diagrams give an abstract way to represent many kinds of processes. They consist of:

- **Nodes**
- **Typed wires**
- **Dangling wires for inputs and outputs**
String diagrams are composed by **plugging** dangling output wires with dangling input wires.

Connecting wires must have **same type**.
Proof-strategy graphs (PSGraphs) are a type of string diagram, with:

tactic-nodes

goal-nodes

tac1

g

tac2

goal types
A tactic-node can be an **atomic tactic**, provided by the theorem prover.
PSGraph tactic nodes

...or a **graph tactic**, which contains:

- one graph
  (hierarchical evaluation)

  \[
  \begin{array}{c}
  \text{one graph} \\
  (\text{hierarchical evaluation})
  \end{array}
  \]

- many graphs
  (hierarchy + branching)

  \[
  \begin{array}{c}
  \text{many graphs} \\
  (\text{hierarchy + branching})
  \end{array}
  \]
Each open goal is represented by a **goal node** in the graph.
Goal types

Wires are labelled by goal types, which are predicates over goals:

\[
\text{goaltype} := \text{top\_symbol}([\text{string}]) \\
| \text{not(goaltype)} \\
| ... 
\]

Tactic nodes can only be plugged together if their input/output types match.
Example

Repeated **forall introduction** can be represented as follows
Evaluation

Evaluation begins by placing a goal node on an input

...and terminates when all remaining goals are on outputs.
Evaluation

Goal-nodes are moved around via graph rewrite rules, which are generated on-the-fly by tactic evaluation:

consume one input goal node
produce new goal nodes on outputs

\[ \text{tac}(g) = [h, i, j] \]
Branching

The output wire of a subgoal is chosen based on its goal type. If multiple wires match a single goal (or if tac is non-deterministic), evaluation can branch:
Hierarchies

Graph tactic-nodes are evaluated in a similarly, but with PSGraph evaluation replacing the call to underlying tactic:

\[
\begin{align*}
\text{one input goal-node added to} & \hspace{1cm} \text{G evaluated recursively} & \hspace{1cm} \text{output goals added as outputs} \\
\text{the associated input of G} & & \text{to the graph tactic-node}
\end{align*}
\]
Hierarchies

If a tactic-node contains multiple graphs, they can be evaluated either in OR-style or ORELSE-style:

- Non-deterministic evaluation:
  \[
  \begin{array}{c}
  \text{non-deterministic} \\
  \text{evaluation:}
  \end{array} \\
  \begin{array}{c}
  = \text{OR} \\
  \end{array} \\
  \begin{array}{c}
  \begin{array}{c}
  \text{graph 1}
  \end{array} \\
  \begin{array}{c}
  \text{graph 2}
  \end{array} \\
  \begin{array}{c}
  \text{graph 3}
  \end{array} \\
  \end{array} \\
  \]

- First successful evaluation:
  \[
  \begin{array}{c}
  \text{first successful} \\
  \text{evaluation:}
  \end{array} \\
  \begin{array}{c}
  = \text{ORELSE}
  \end{array} \\
  \begin{array}{c}
  \begin{array}{c}
  \text{graph 1}
  \end{array} \\
  \begin{array}{c}
  \text{graph 2}
  \end{array} \\
  \begin{array}{c}
  \text{graph 3}
  \end{array} \\
  \end{array} \\
  \]
Tinker

A tool for building PSGraphs...

...and evaluating them.