Tinker, Tailor, Solver, Proof

Writing Graphical Proof Strategies in Tinker

Gudmund Grov
Yuhui Lin

Aleks Kissinger

HERIOT WATT UNIVERSITY

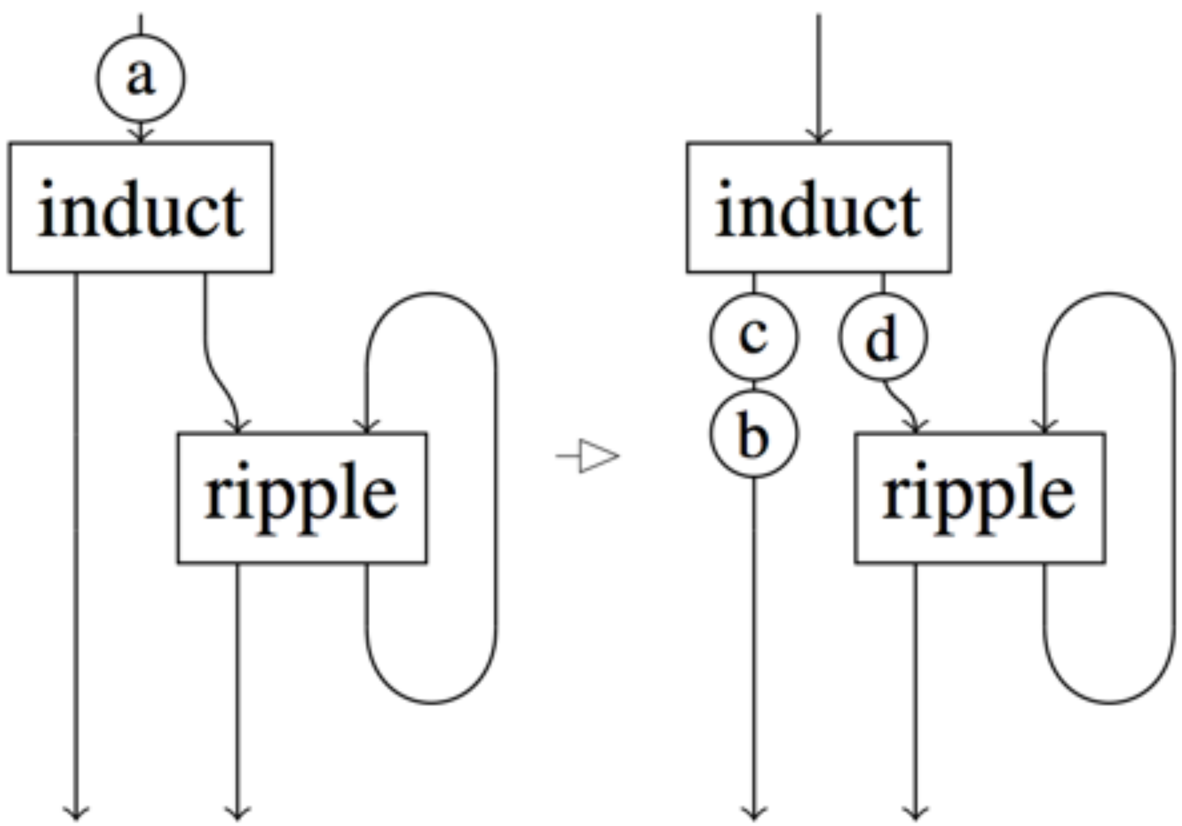
UNIVERSITY OF OXFORD

AI4FM

funded by EPSRC grants: EP/H023852, EP/H024204 and EP/J001058, the John Templeton Foundation and the Office of Navel 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

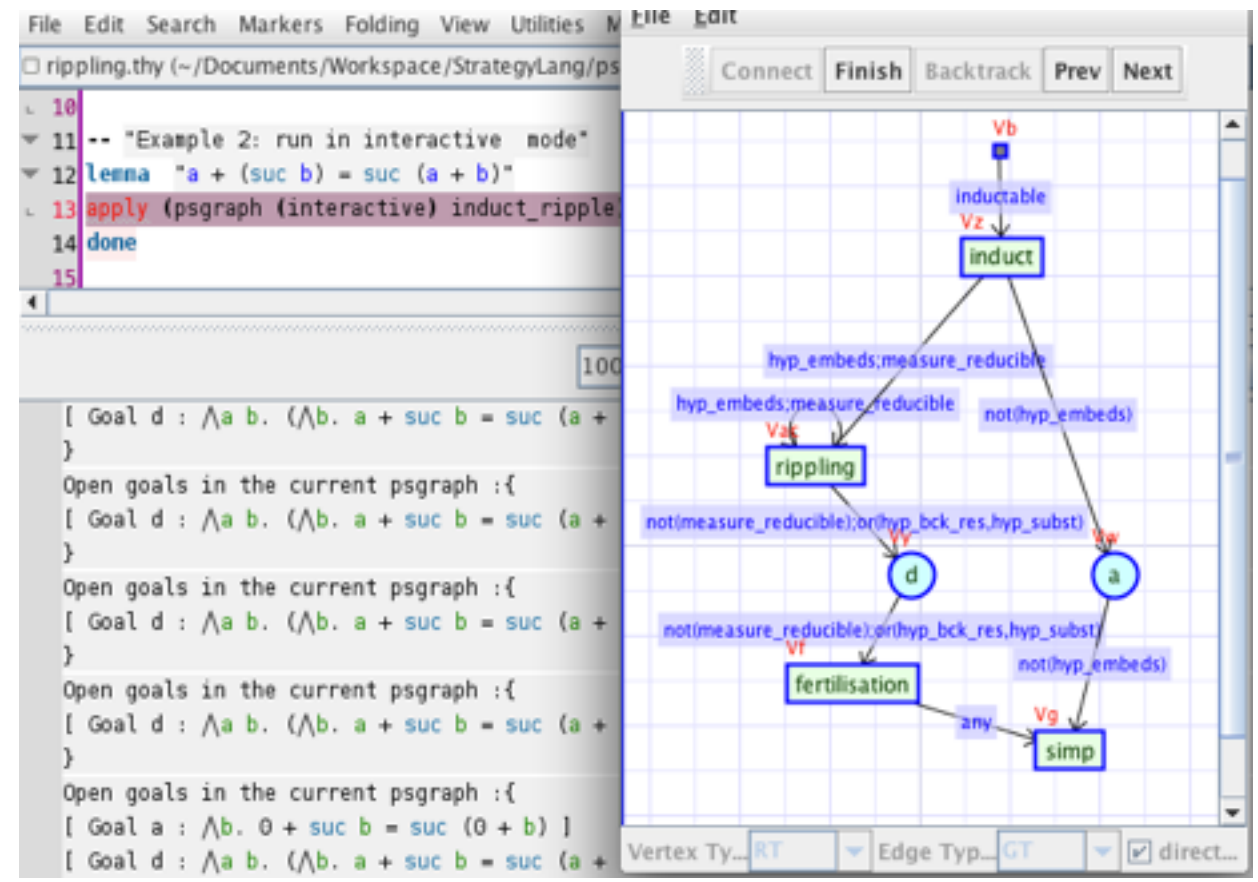
- Run as tactic within TP
- Evaluation:
- Implemented for **ProofPower**

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**

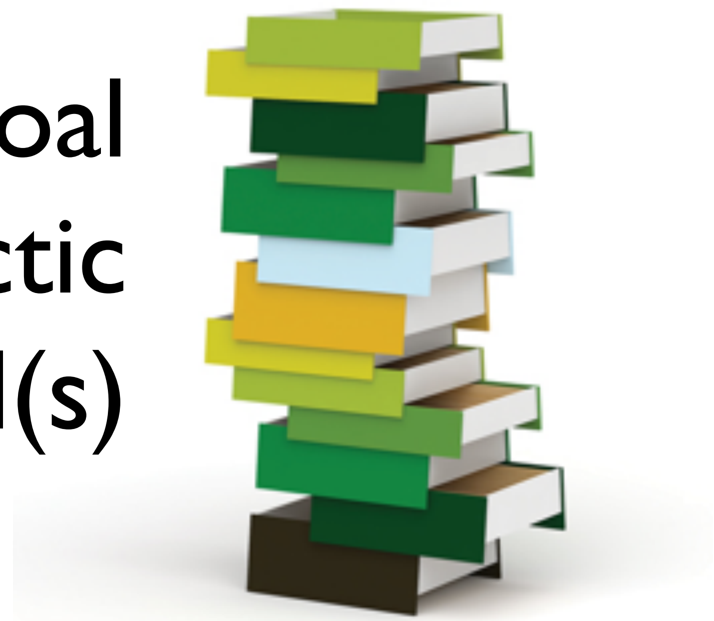
Stack-based strategies

LCF-style provers operate on open goals using **tactics**:

```
t: goal -> [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:

`t₁ THEN t₂`   `t₁ OR t₂`   `REPEAT t`

Stack-based strategies

```
tac mytac := t1 THEN t2 THEN t2 THEN t3
```

|

```
mytac(g) :=
```

Stack-based strategies

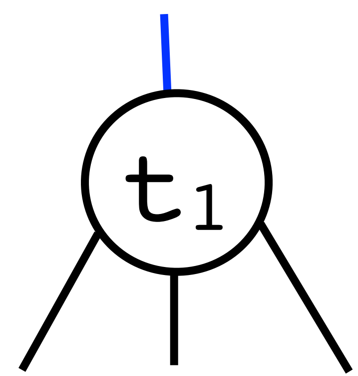
```
tac mytac := t₁ THEN t₂ THEN t₂ THEN t₃
```

```
mytac(g) :=
```

Stack-based strategies

tac mytac := t_1 THEN t_2 THEN t_2 THEN t_3

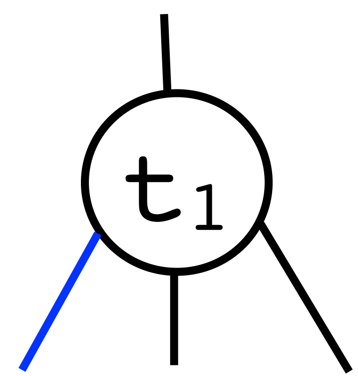
mytac(g) :=



Stack-based strategies

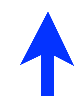
tac mytac := t_1 THEN t_2 THEN t_2 THEN t_3

mytac(g) :=

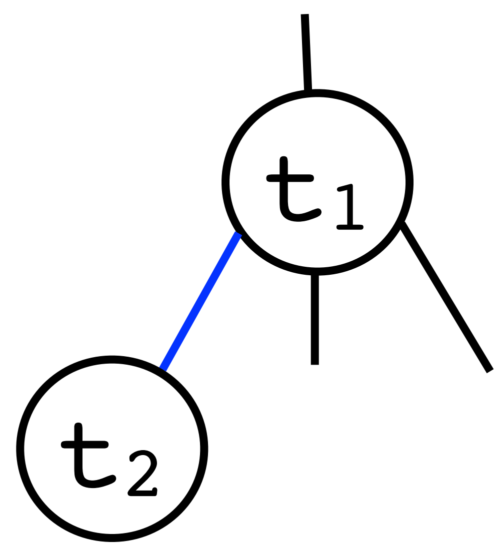


Stack-based strategies

tac mytac := t_1 THEN t_2 THEN t_2 THEN t_3



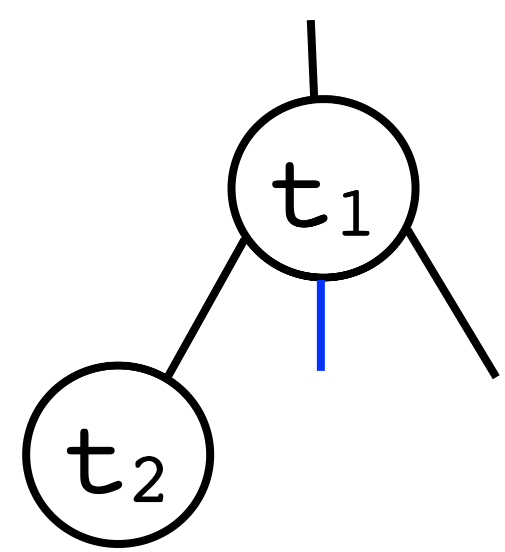
mytac(g) :=



Stack-based strategies

tac `mytac := ` t_1 `THEN` t_2 `THEN` $\underline{t_2}$ `THEN` t_3

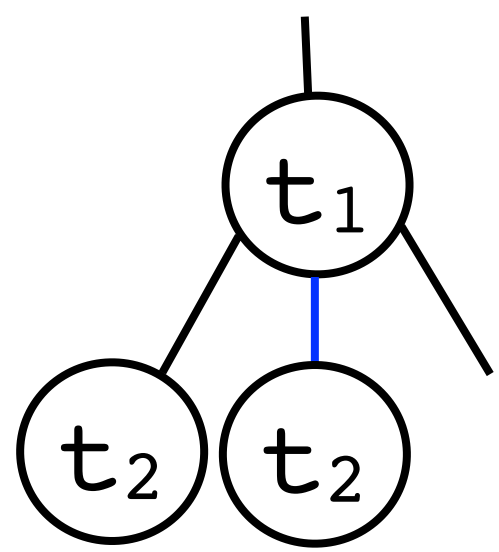
`mytac(g) :=`



Stack-based strategies

tac mytac := t_1 THEN t_2 THEN t_2 THEN t_3

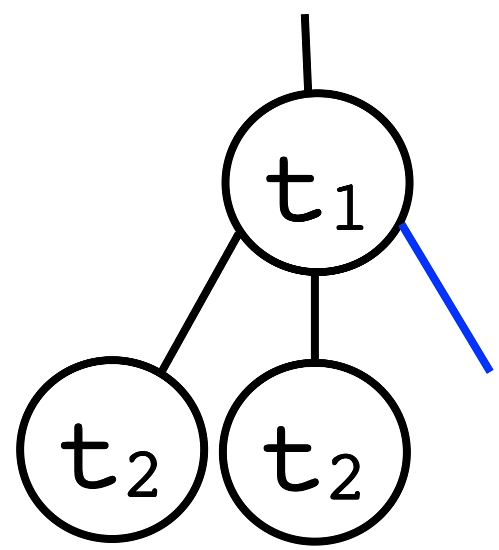
mytac(g) :=



Stack-based strategies

tac mytac := t_1 THEN t_2 THEN t_2 THEN t_3

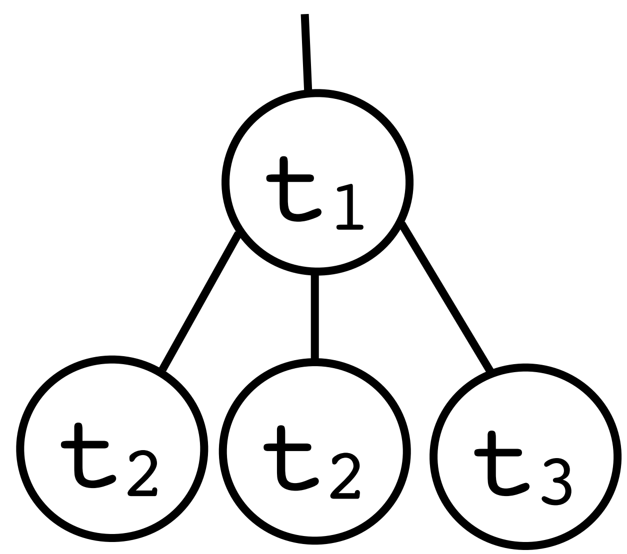
mytac(g) :=



Stack-based strategies

tac mytac := t_1 THEN t_2 THEN t_2 THEN t_3

mytac(g) :=



But sometimes it goes wrong....

Suppose we replace $\texttt{t}_1$ with the "improved" tactic $\texttt{t}_\texttt{x}$

Tactic based proving

```
tac mytac := tx THEN t2 THEN t2 THEN t3
```

```
mytac(g) :=
```

Tactic based proving

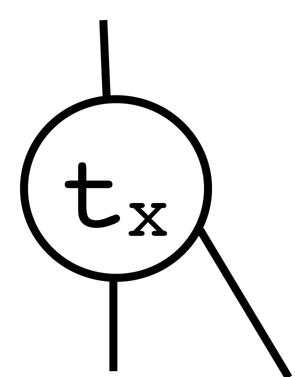
```
tac mytac := tx THEN t2 THEN t2 THEN t3
```

```
mytac(g) :=
```

Tactic based proving

tac mytac := t_x THEN t_2 THEN t_2 THEN t_3

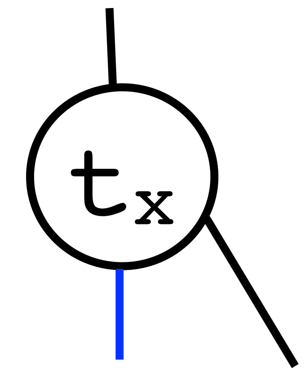
mytac(g) :=



Tactic based proving

```
tac mytac := tx THEN t2 THEN t2 THEN t3
```

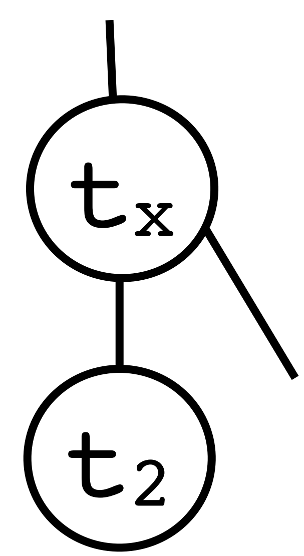
mytac(g) :=



Tactic based proving

tac mytac := t_x THEN t_2 THEN t_2 THEN t_3

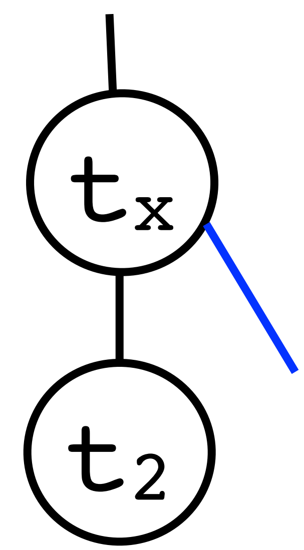
mytac(g) :=



Tactic based proving

tac mytac := t_x THEN t_2 THEN $\underline{t_2}$ THEN t_3

mytac(g) :=



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 <u>t_2</u> THEN t_3

error

Debugging

where did it go wrong?

actual
error

tac mytac := t_x THEN t_2 THEN t_2 THEN t_3

error

Debugging

where did it go wrong?

or
here

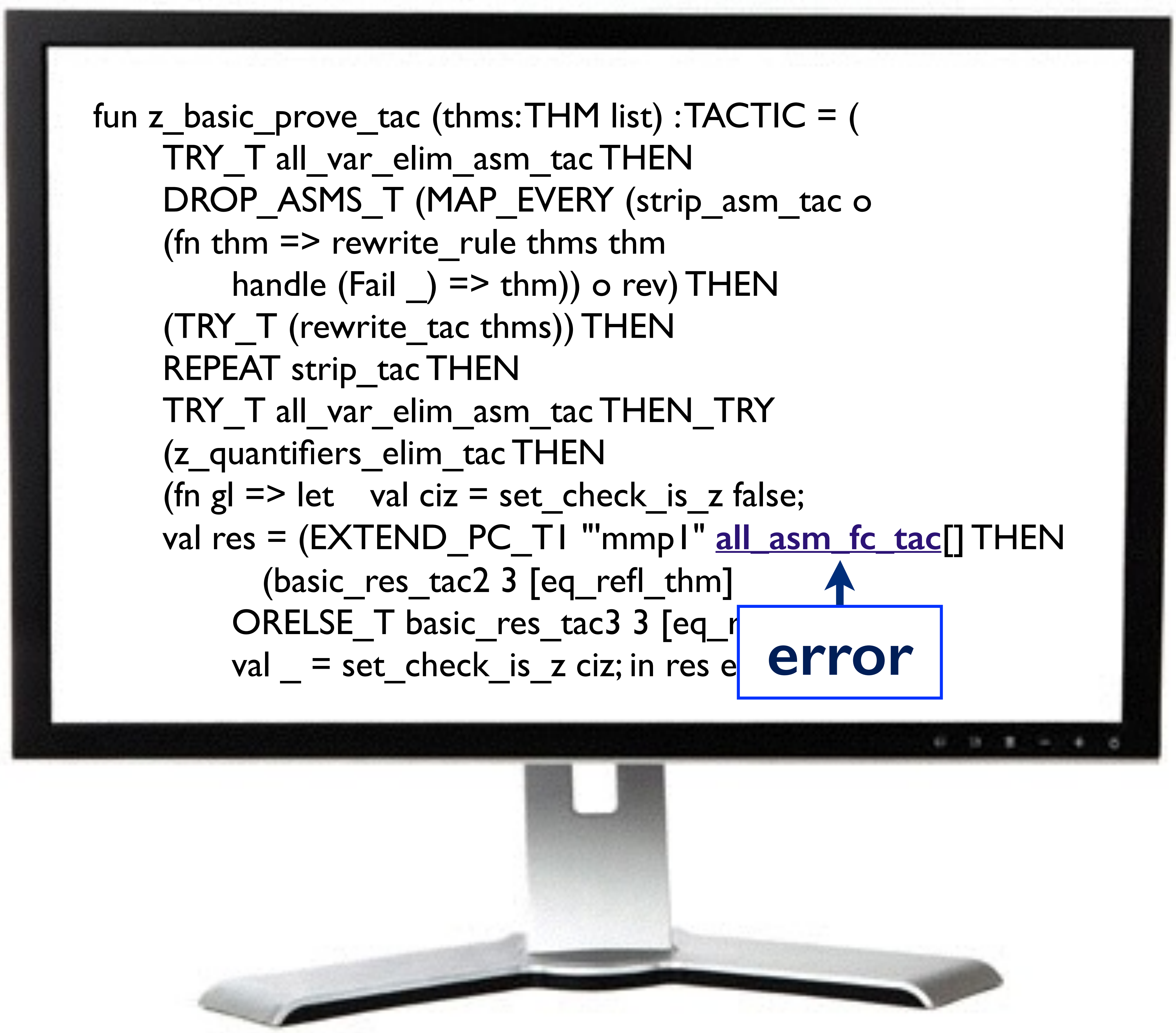
$\texttt{\textbf{tac} mytac := } t_x\ \underline{\texttt{THEN}\ t_2}\ \texttt{THEN}\ \underline{t_2}\ \texttt{THEN}\ t_3$

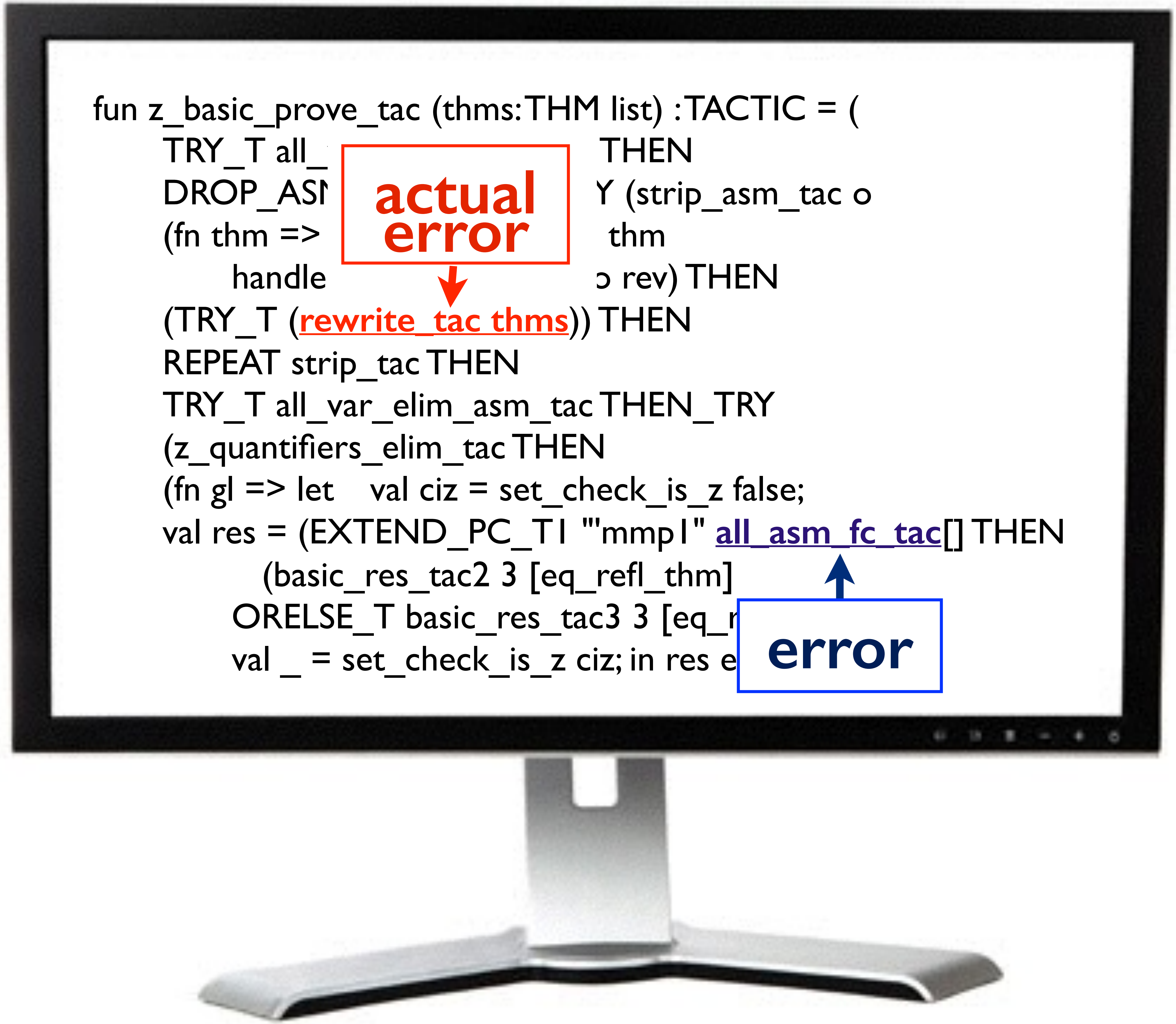
error

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]
ORELSE_T basic_res_tac3 3 [eq_r
val _ = set_check_is_z ciz; in res e
error



fun z_basic_prove_tac (thms:THM list) :TACTIC = (
TRY_T all_ THEN
DROP_ASI Y (strip_asm_tac o
(fn thm => thm
handle ɔ 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_r
val _ = set_check_is_z ciz; in res e
actual error
error


or..

```
(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

)));
```

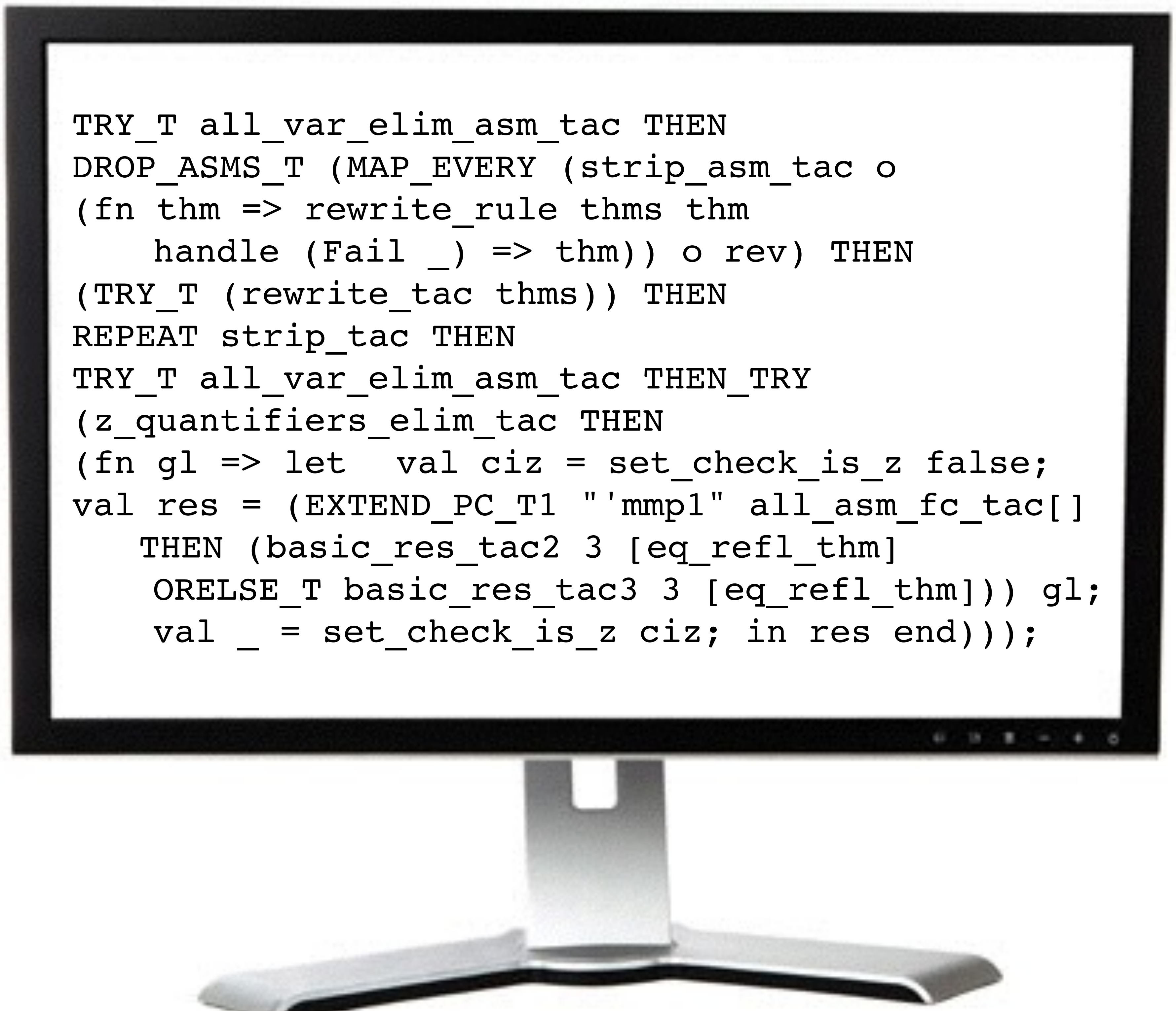
error

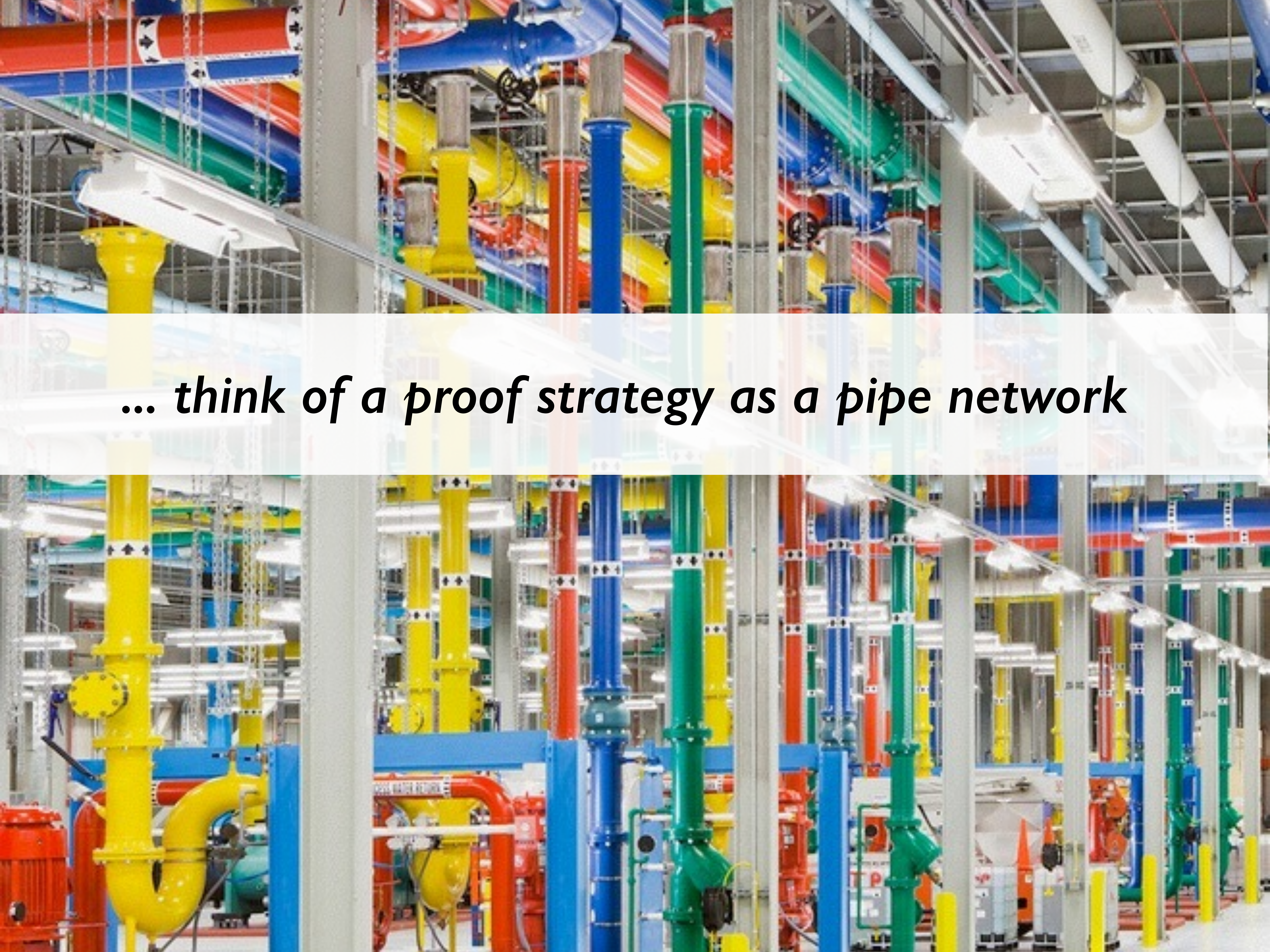
```
fun z_basic_prove_tac (thms: THM list) : TACTIC = (
    TRY_T all_var_elim_asm_tac THEN
    DROP_ASMS_T (MAP_EVERY (strip_asm_tac o
    (fn thm => rewrite_rule thms thm
        handle (Fail _) => thm)) o rev) THEN
    (TRY_T (rewrite_tac thms)) THEN
    REPEAT strip_tac THEN
    TRY_T all_var_elim_asm_tac 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
```

```
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
```

actual error

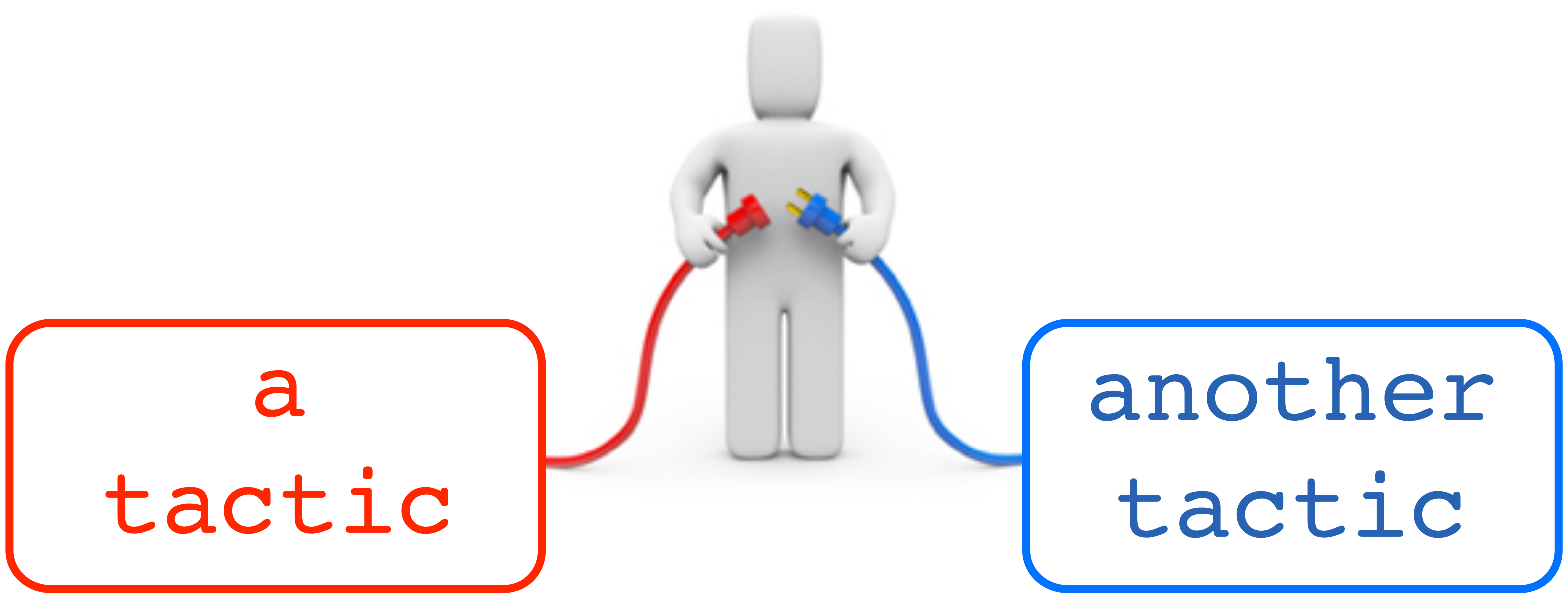
Instead of...


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



Loops

Repetition is simply a **feedback** pipe

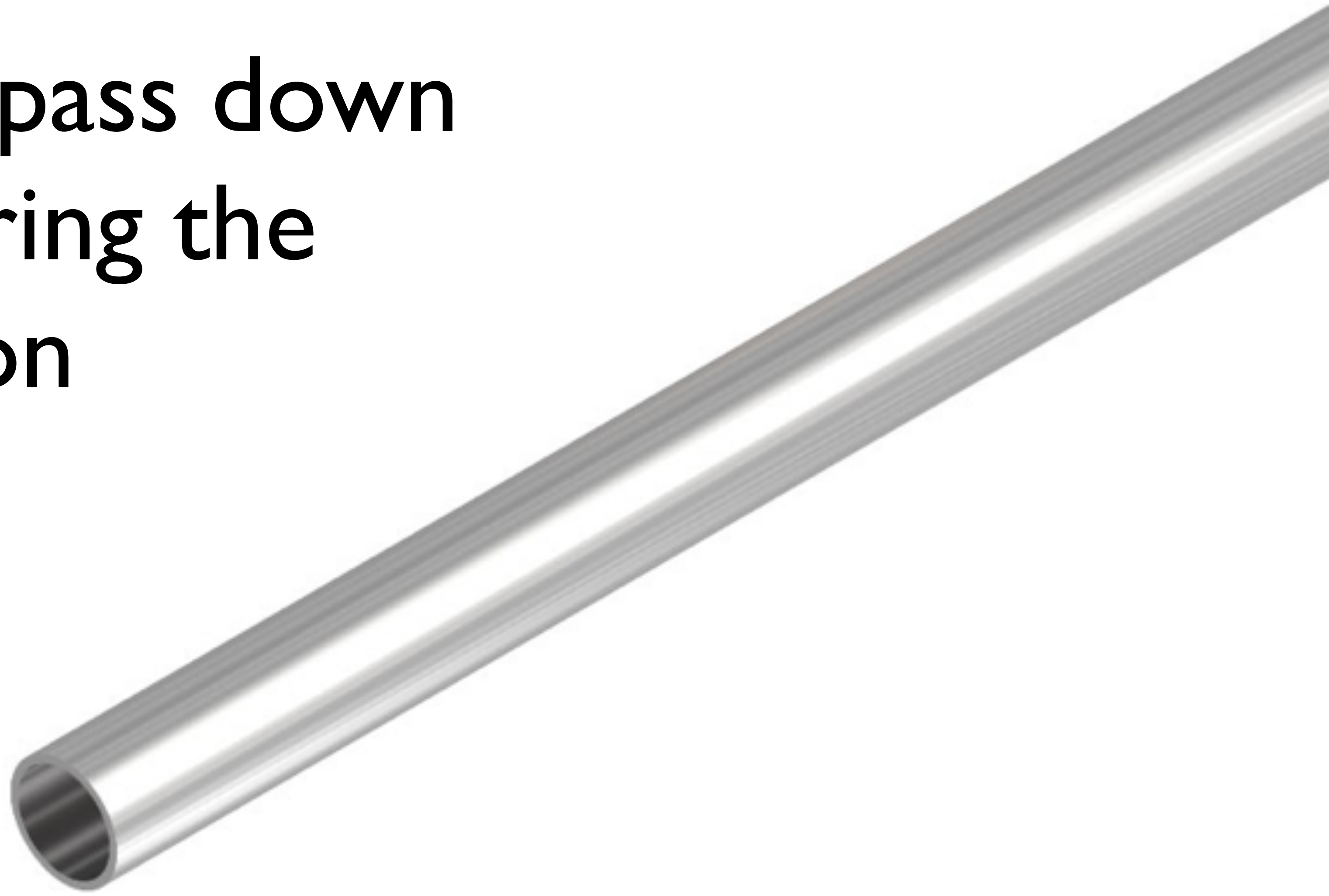
Passing goals

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

A goal must **fit** in the pipe it is in

Passing goals

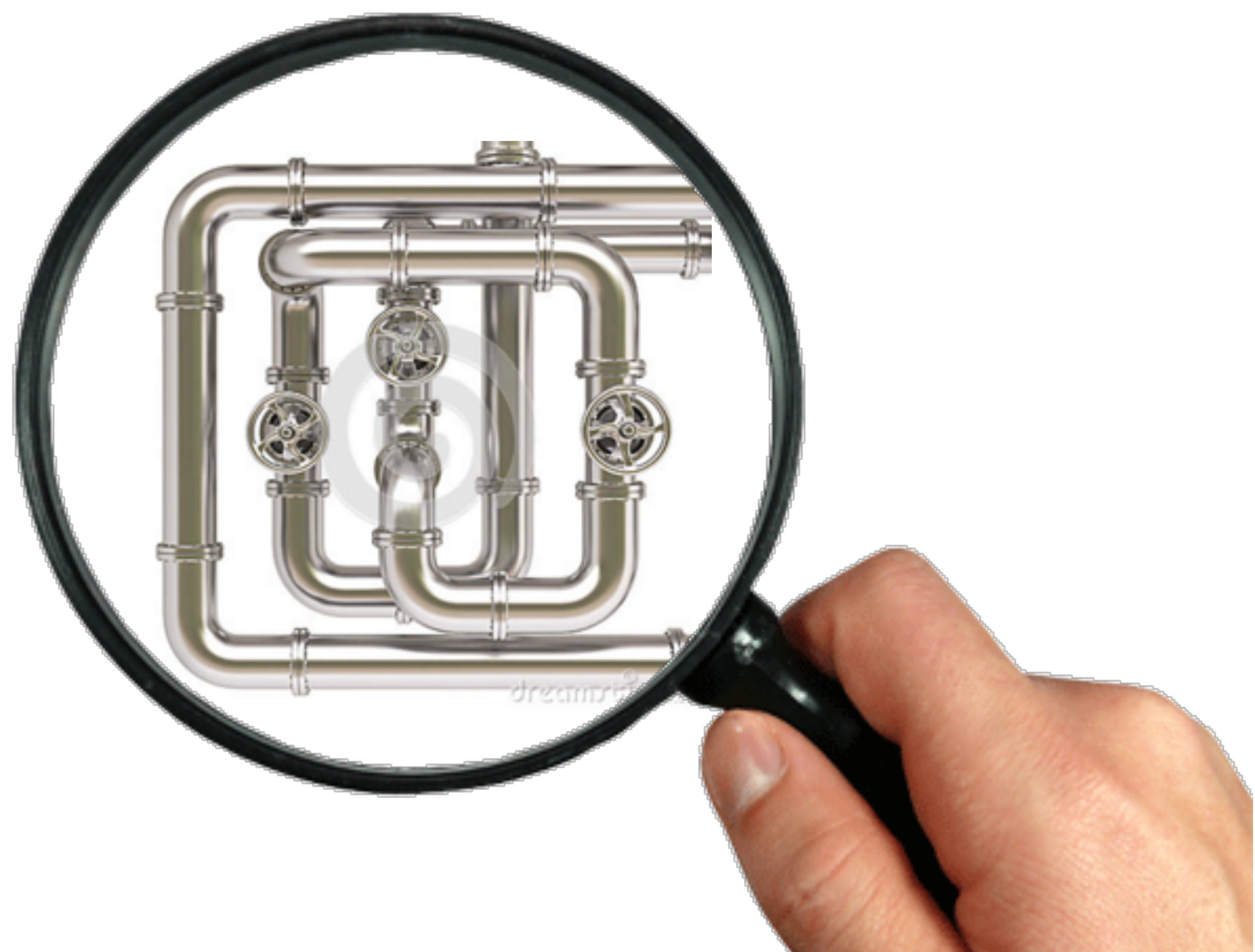
Multiple goals can pass down the **same pipe** during the course of evaluation



abstracts over goal **number** and **order**

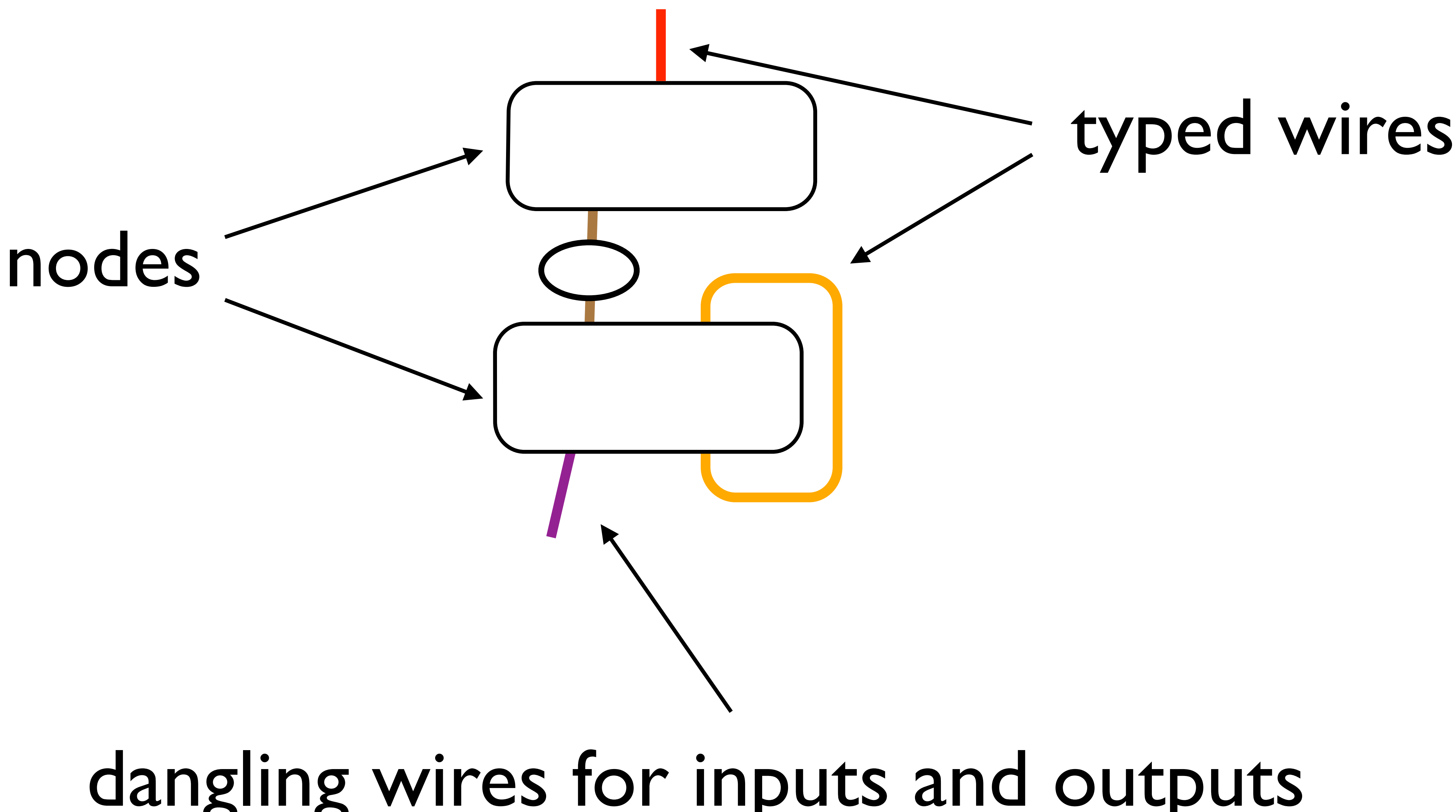
Hierarchies

Networks can be **structured** so a tactic can itself be a pipe network



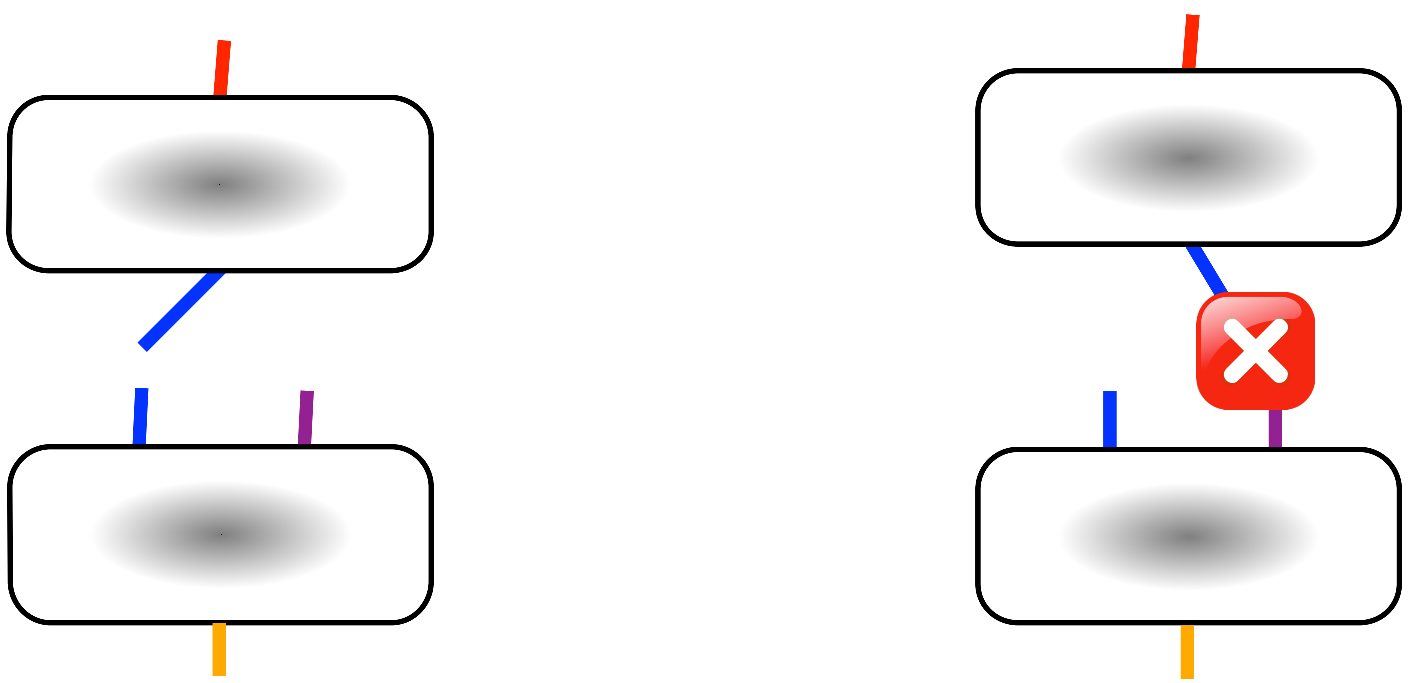
String diagrams

String diagrams give an abstract way to represent many kinds of processes. They consist of:



Composition

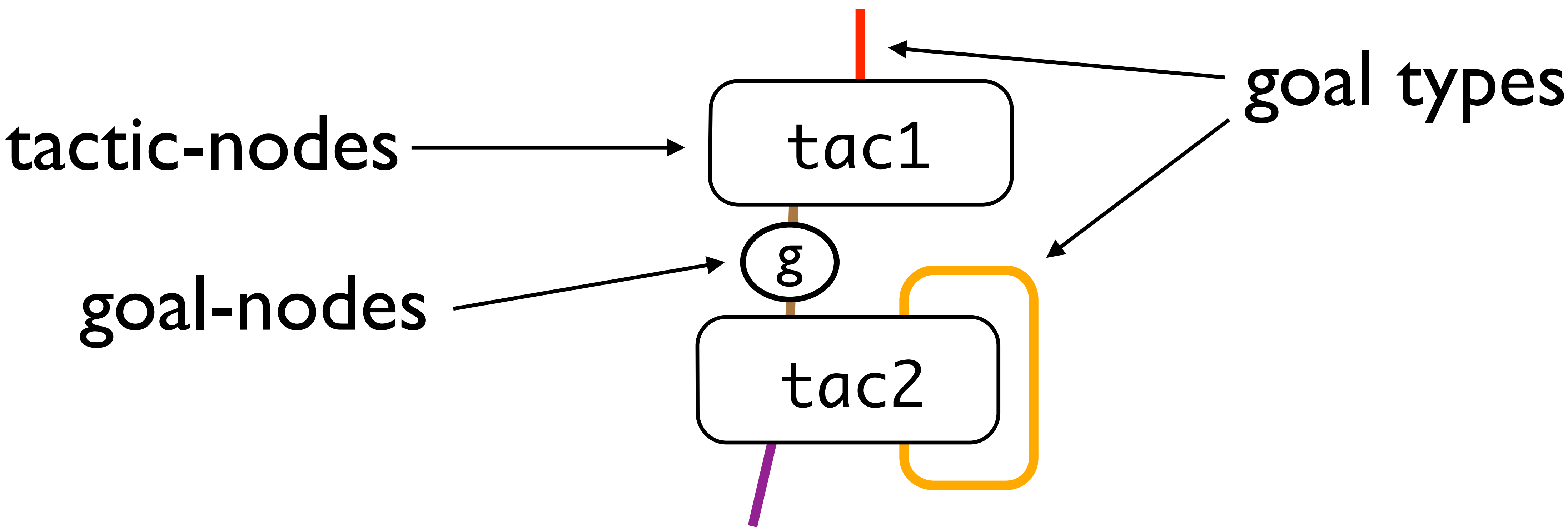
String diagrams are composed by **plugging** dangling output wires with dangling input wires



Connecting wires must have **same type**

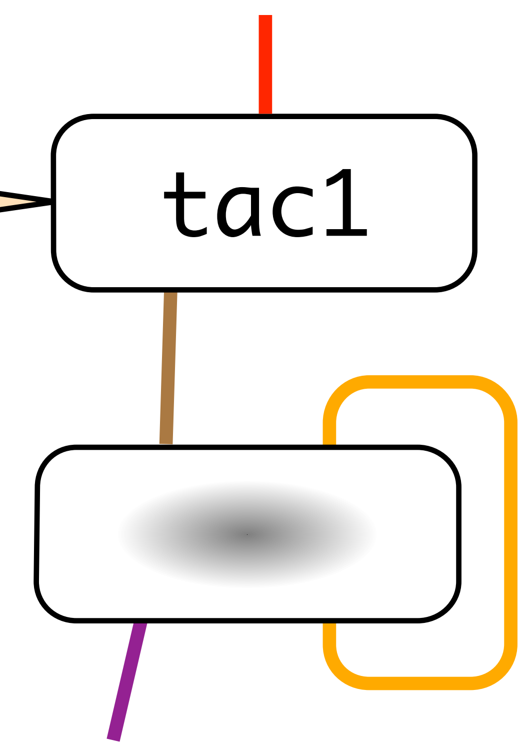
PSGraphs

Proof-strategory graphs (**PSGraphs**) are a type of **string diagram**, with:



PSGraph tactic nodes

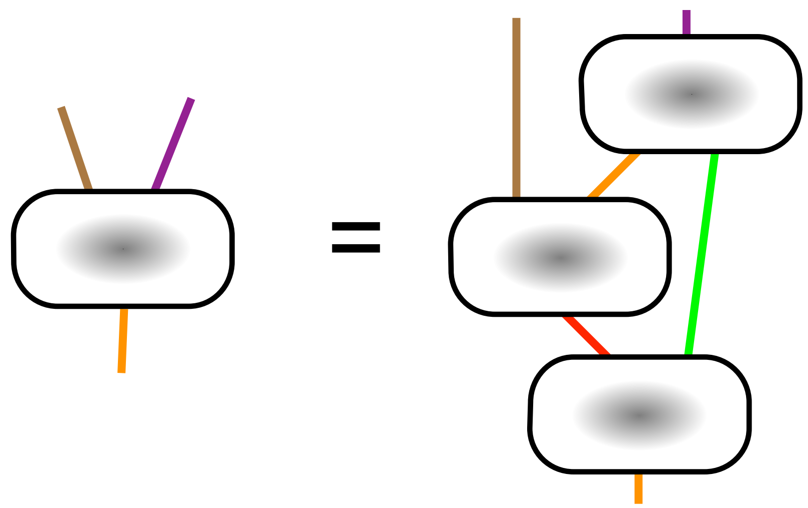
A tactic-node can be an **atomic tactic**, provided by the theorem prover



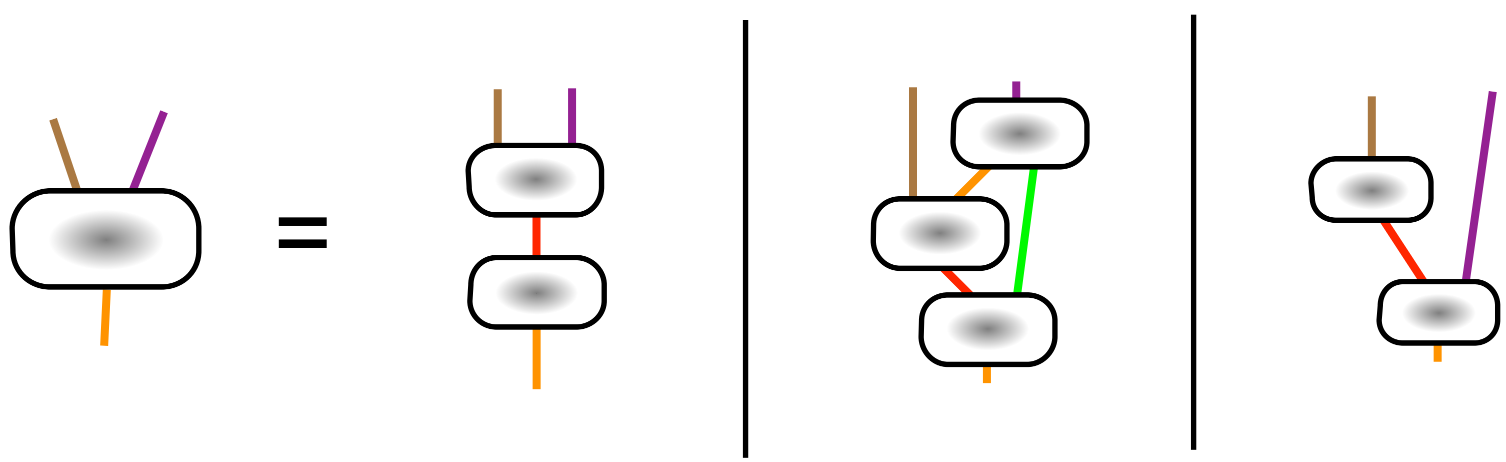
PSGraph tactic nodes

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

one graph
(hierarchical evaluation)

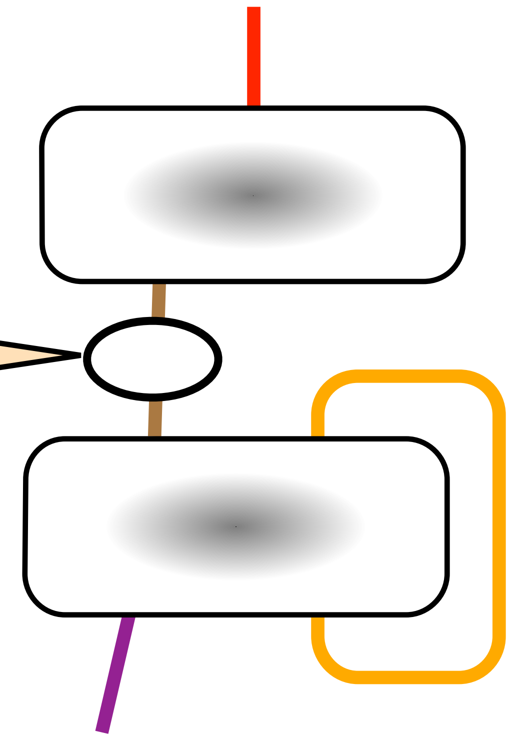


...or many graphs
(hierarchy + branching)



PSGraph goal nodes

Each open goal is represented by a **goal node** in the graph



Goal types

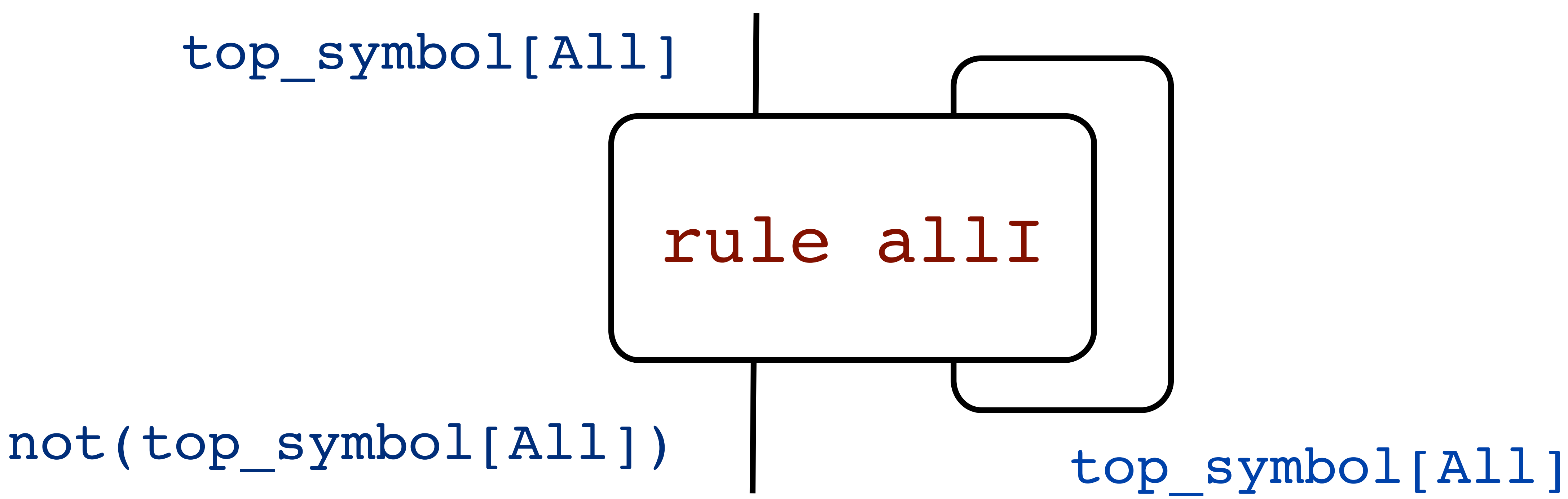
Wires are labelled by goal types, which are predicates over goals:

```
goaltype := top_symbol([string])
          | not(goaltype)
          | ...
```

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

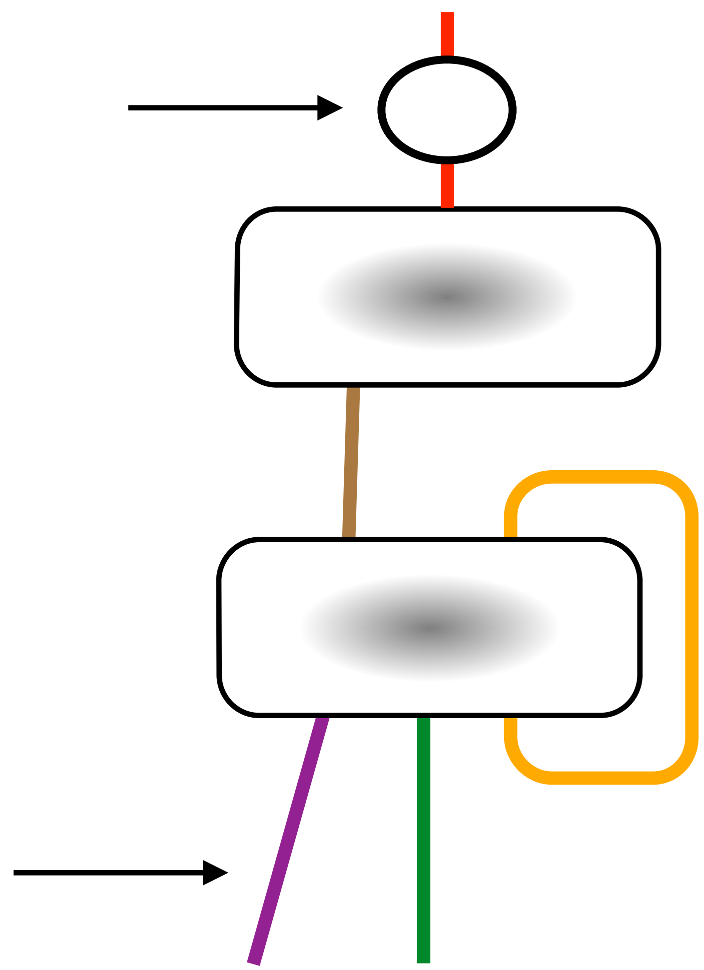
Example

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



Evaluation

Evaluation begins by placing a goal node on an input

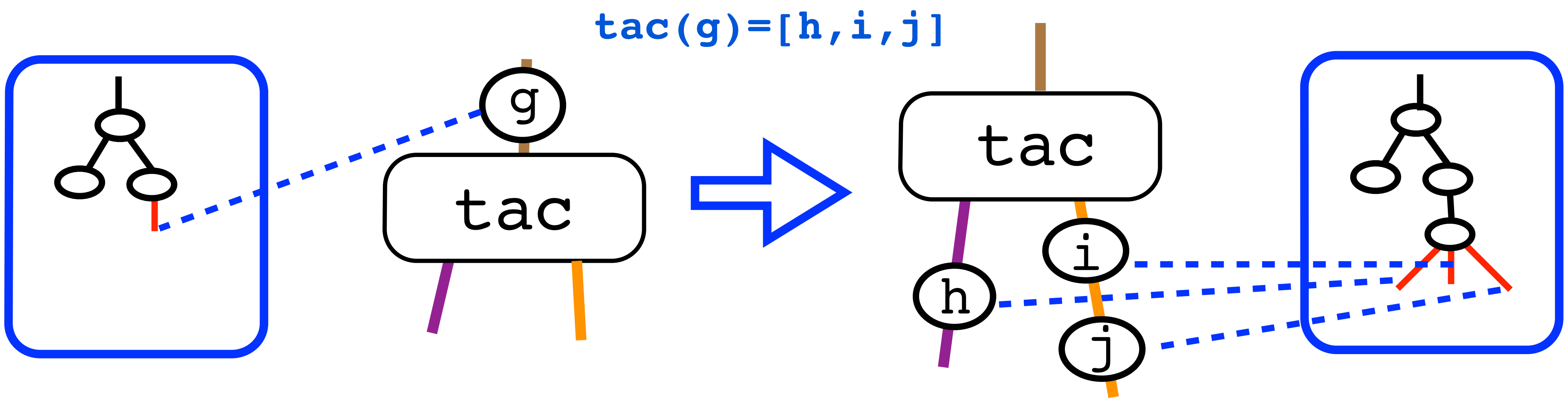


...and terminates when all remaining goals are on outputs.

Evaluation

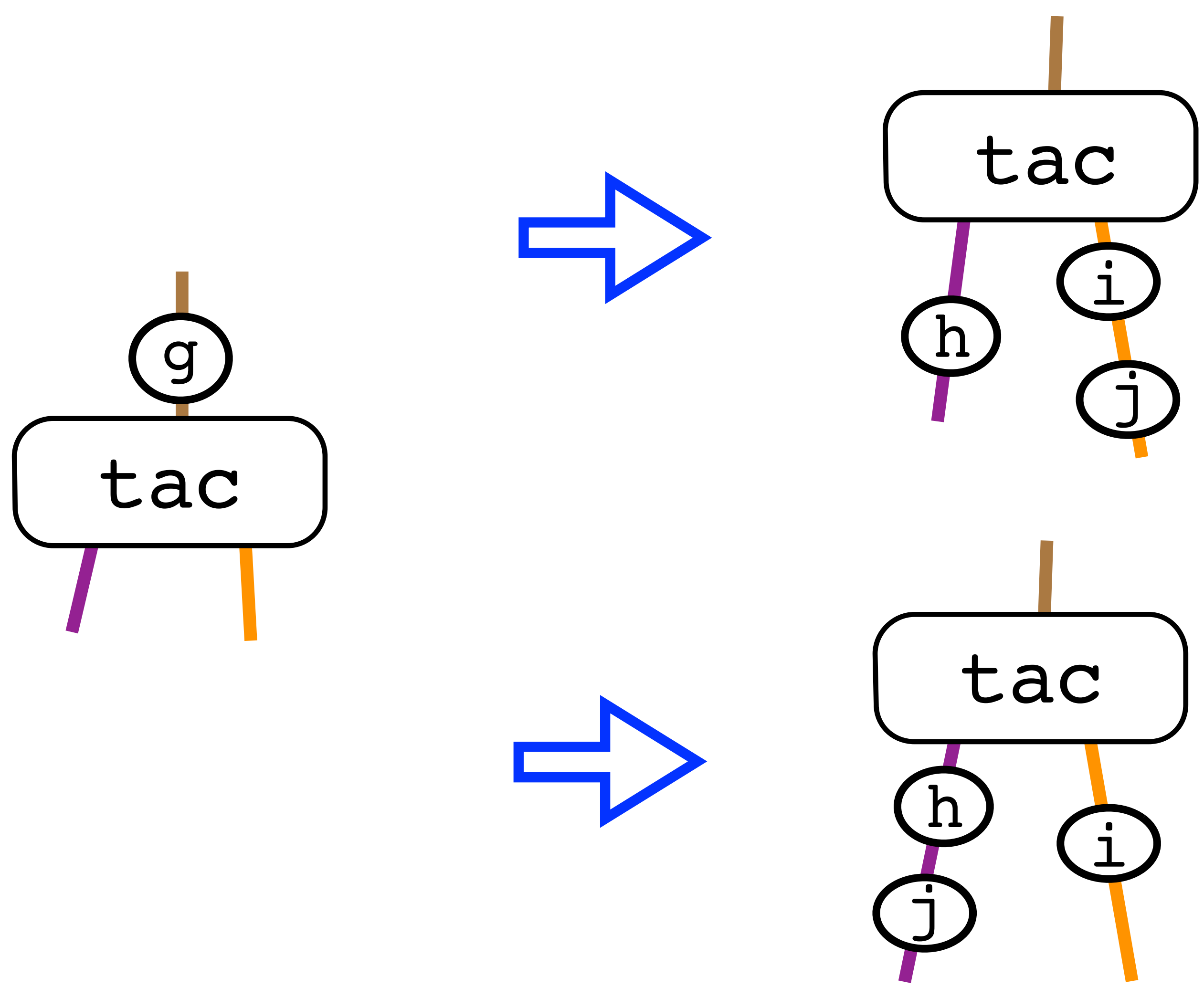
Goal-nodes are moved around via **graph rewrite rules**, which are generated on-the-fly by **tactic evaluation**:

consume one input goal node
produce new goal nodes on outputs



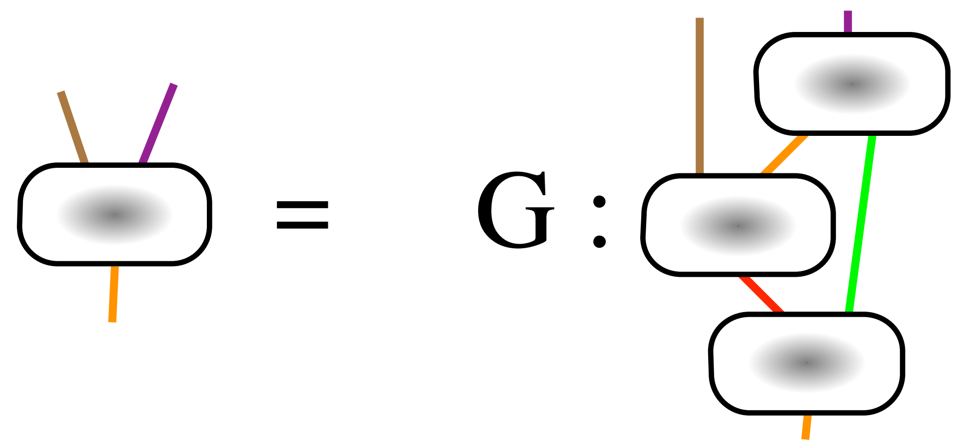
Branching

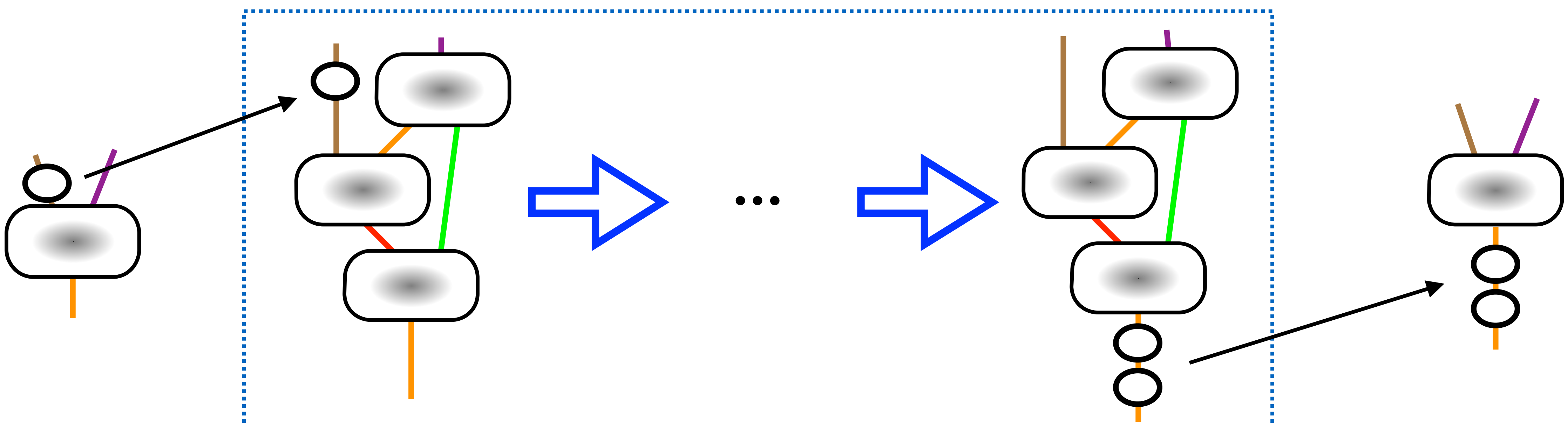
The output wire of a subgoal is chosen based on its goal type. If multiple wires match a single goal (or if `tac` is non-deterministic), evaluation can branch:



Hierarchies

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





one input goal-node added to the associated input of G

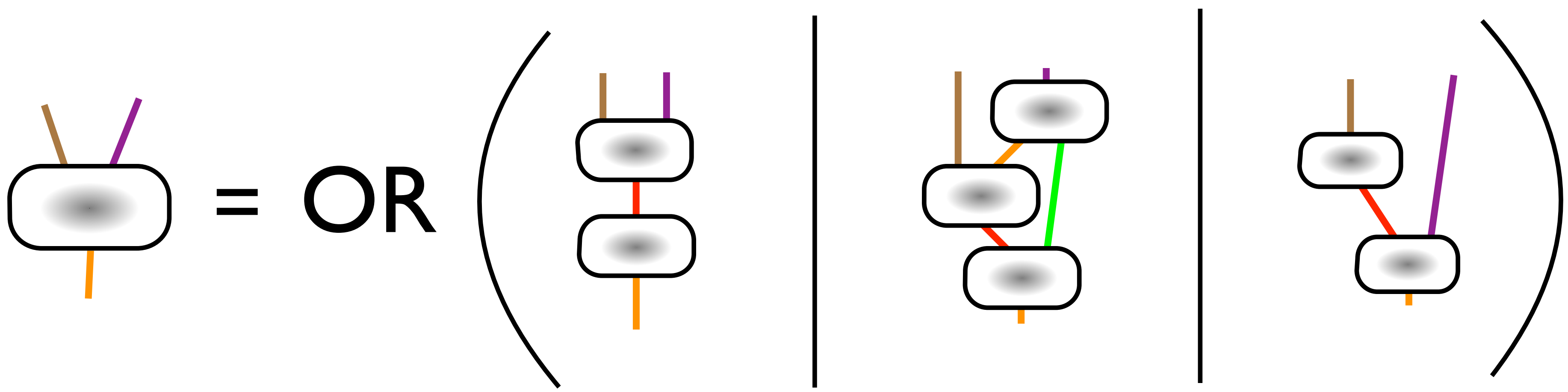
G evaluated recursively

output goals added as outputs to the graph tactic-node

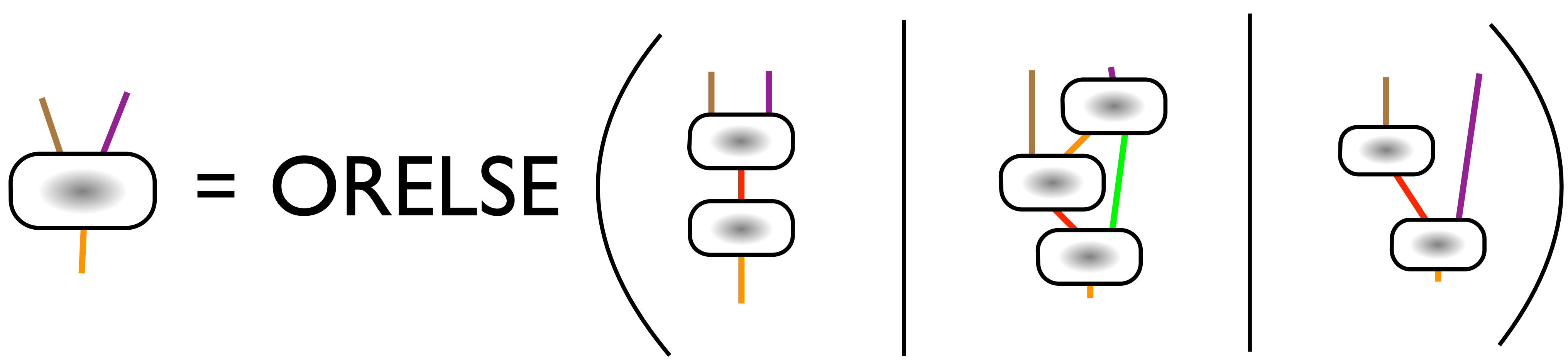
Hierarchies

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

non-deterministic evaluation:

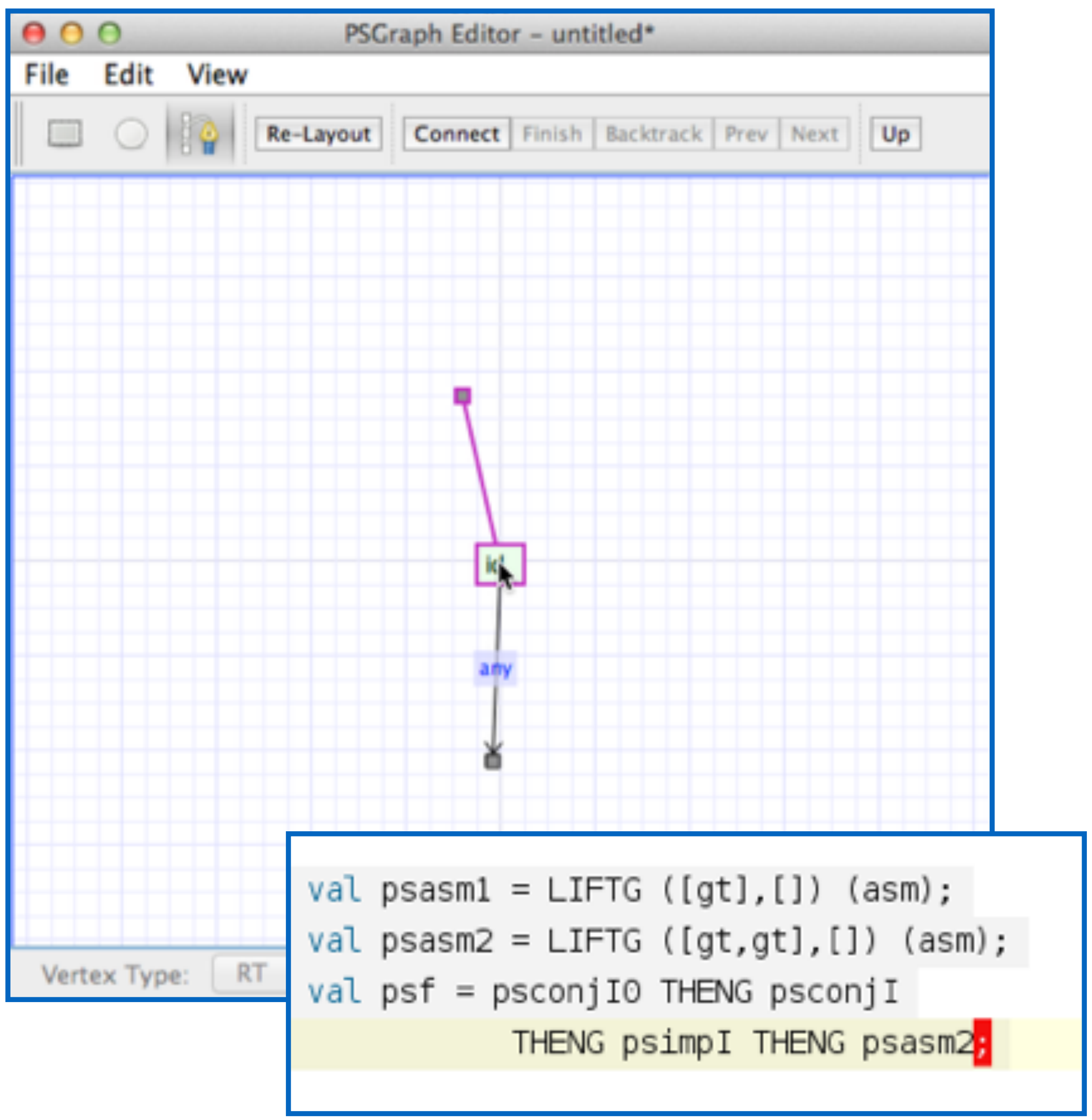


first successful evaluation:



Tinker

A tool for building PSGraphs...



...and evaluating them.