A Graphical Language for Proof Strategies

Gudmund Grov
Yuhui Lin

Aleks Kissinger

HERIOT WATT UNIVERSITY

UNIVERSITY OF OXFORD

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

Tactic based proving

LCF-based provers handle **soundness** by a **`thm`** type and a **kernel** of trusted axioms and inference rules

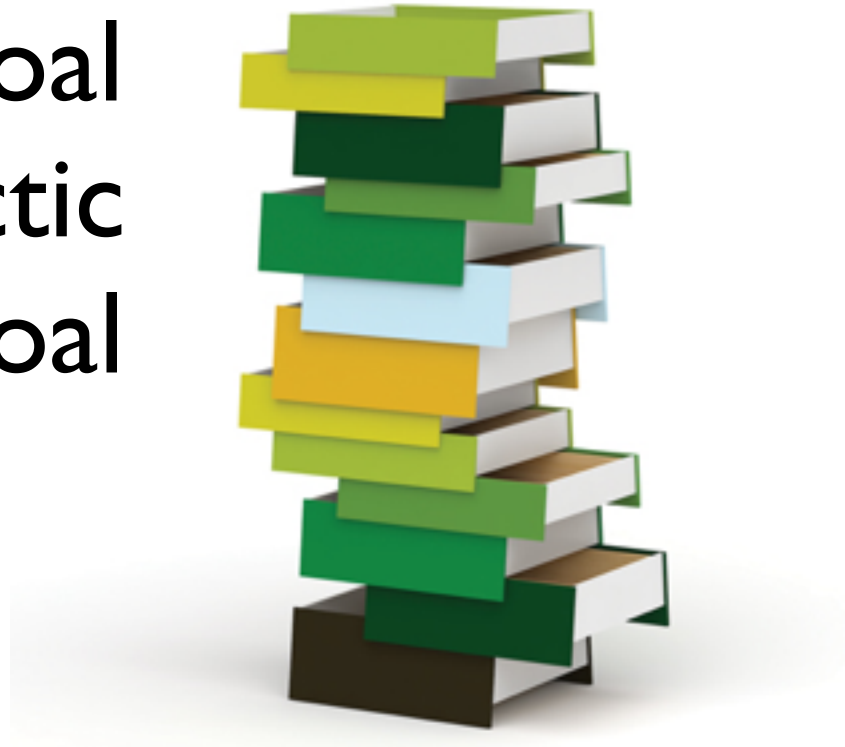
Proof automation by programs called **tactics**

```
goal -> [goal]
```

Tactic based proving

Stack based goal propagation

pop first goal
apply tactic
push new sub-goal



Tactic based proving

Proof strategies from existing tactics by **tactical** combinators

t_1 THEN t_2

t_1 OR t_2

REPEAT t

Tactic based proving

tac mytac := t_1 THEN t_2 THEN t_2 THEN t_3

Tactic based proving

tac mytac := t_1 THEN t_2 THEN t_2 THEN t_3

mytac(g) :=

Tactic based proving

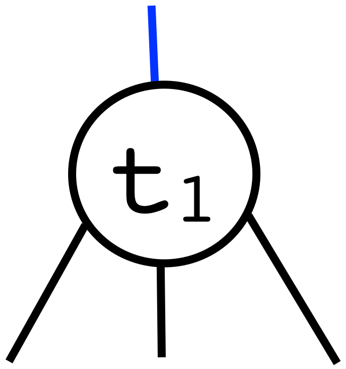
tac mytac := $\underline{t_1}$ THEN t_2 THEN t_2 THEN t_3

mytac(g) :=

Tactic based proving

tac mytac := t_1 THEN t_2 THEN t_2 THEN t_3

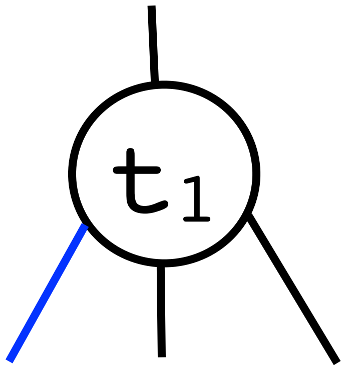
mytac(g) :=



Tactic based proving

tac mytac := t_1 THEN t_2 THEN t_2 THEN t_3

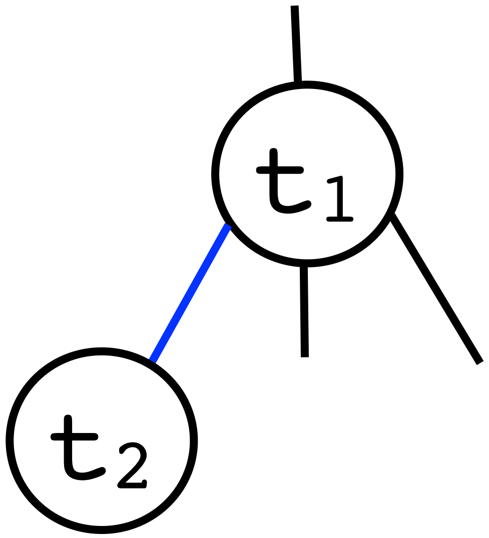
mytac(g) :=



Tactic based proving

tac mytac := t_1 THEN t_2 THEN t_2 THEN t_3

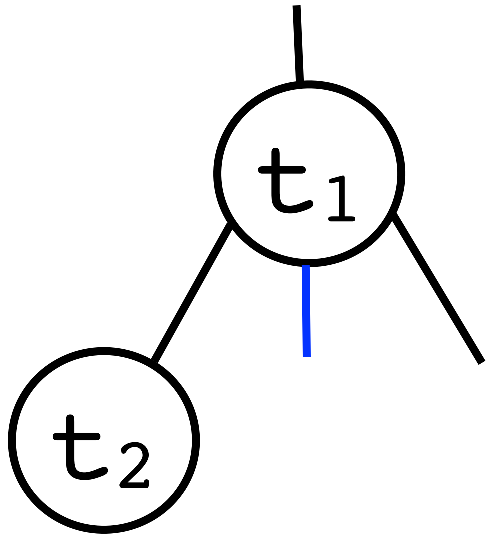
mytac(g) :=



Tactic based proving

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

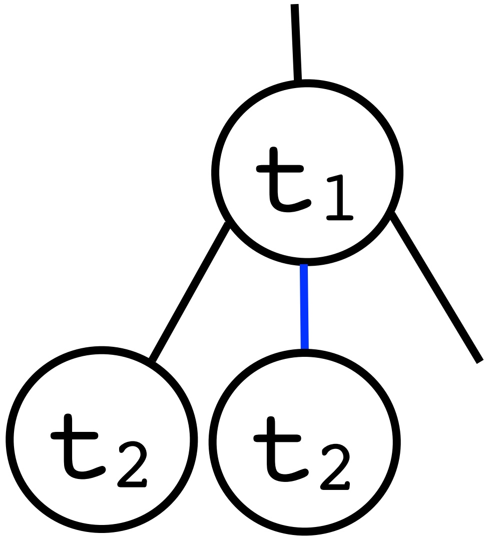
mytac(g) :=



Tactic based proving

tac mytac := t_1 THEN t_2 THEN t_2 THEN t_3

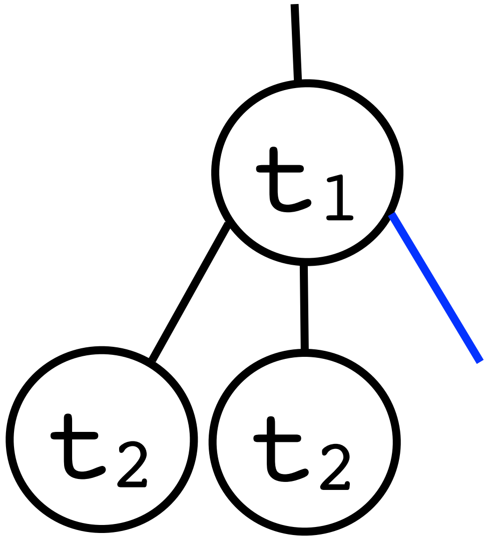
mytac(g) :=



Tactic based proving

tac mytac := t_1 THEN t_2 THEN t_2 THEN t_3

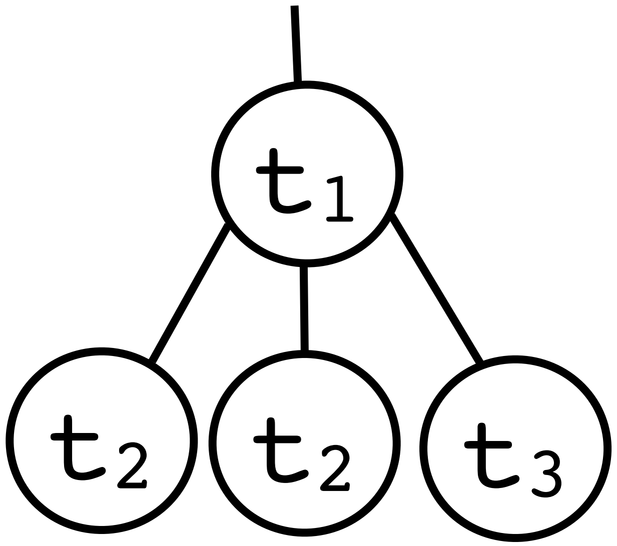
mytac(g) :=



Tactic based proving

tac mytac := t_1 THEN t_2 THEN t_2 THEN t_3

mytac(g) :=



Tactic based proving

Now, let us replace `t₁` with the "improved" `tₓ` tactic

Tactic based proving

tac mytac := t_x THEN t_2 THEN t_2 THEN t_3

mytac(g) :=

Tactic based proving

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

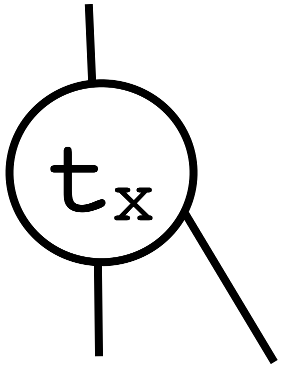
mytac(g) :=

Tactic based proving

tac mytac := t_x THEN t_2 THEN t_2 THEN t_3

↑

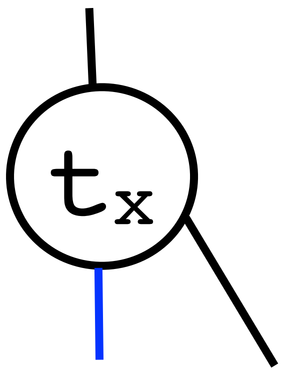
mytac(g) :=



Tactic based proving

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

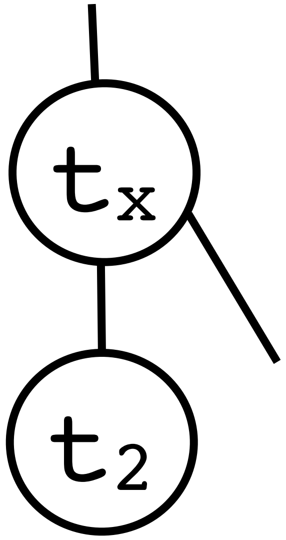
mytac(g) :=



Tactic based proving

tac mytac := t_x THEN t_2 THEN t_2 THEN t_3

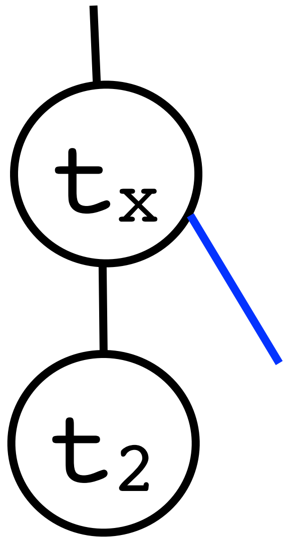
mytac(g) :=



Tactic based proving

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

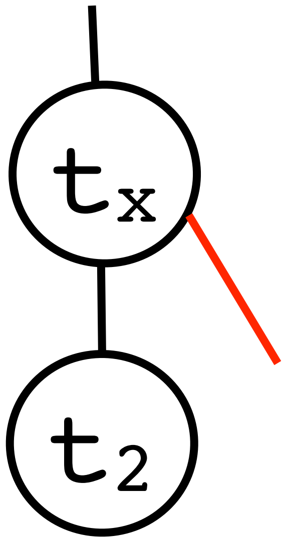
mytac(g) :=



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 t_2 THEN t_3

mytac(g) := ❎

Debugging

where did it go wrong?

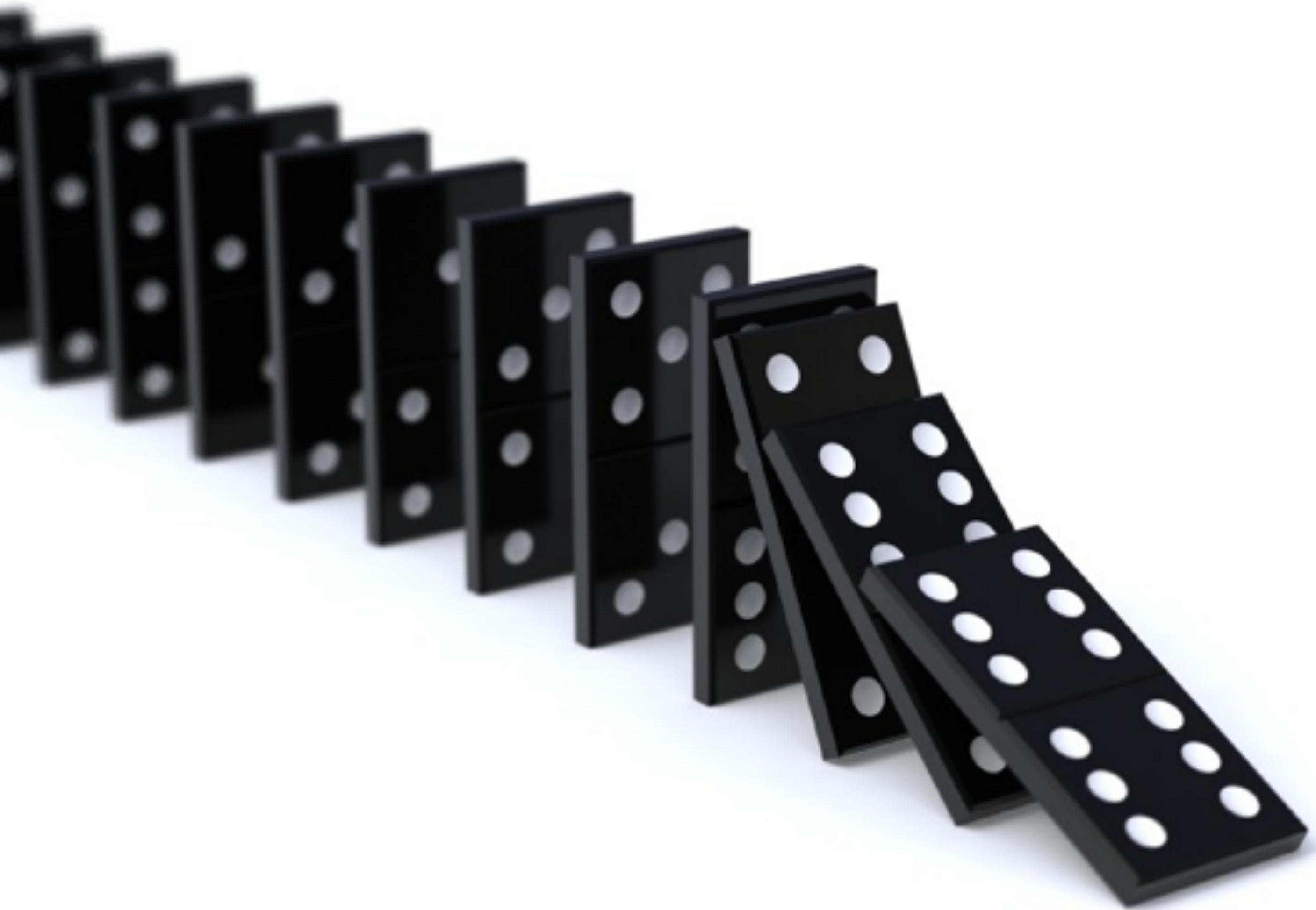
tac mytac := t_x THEN t_2 THEN t_2 THEN t_3

Debugging

where did it go wrong?

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

error



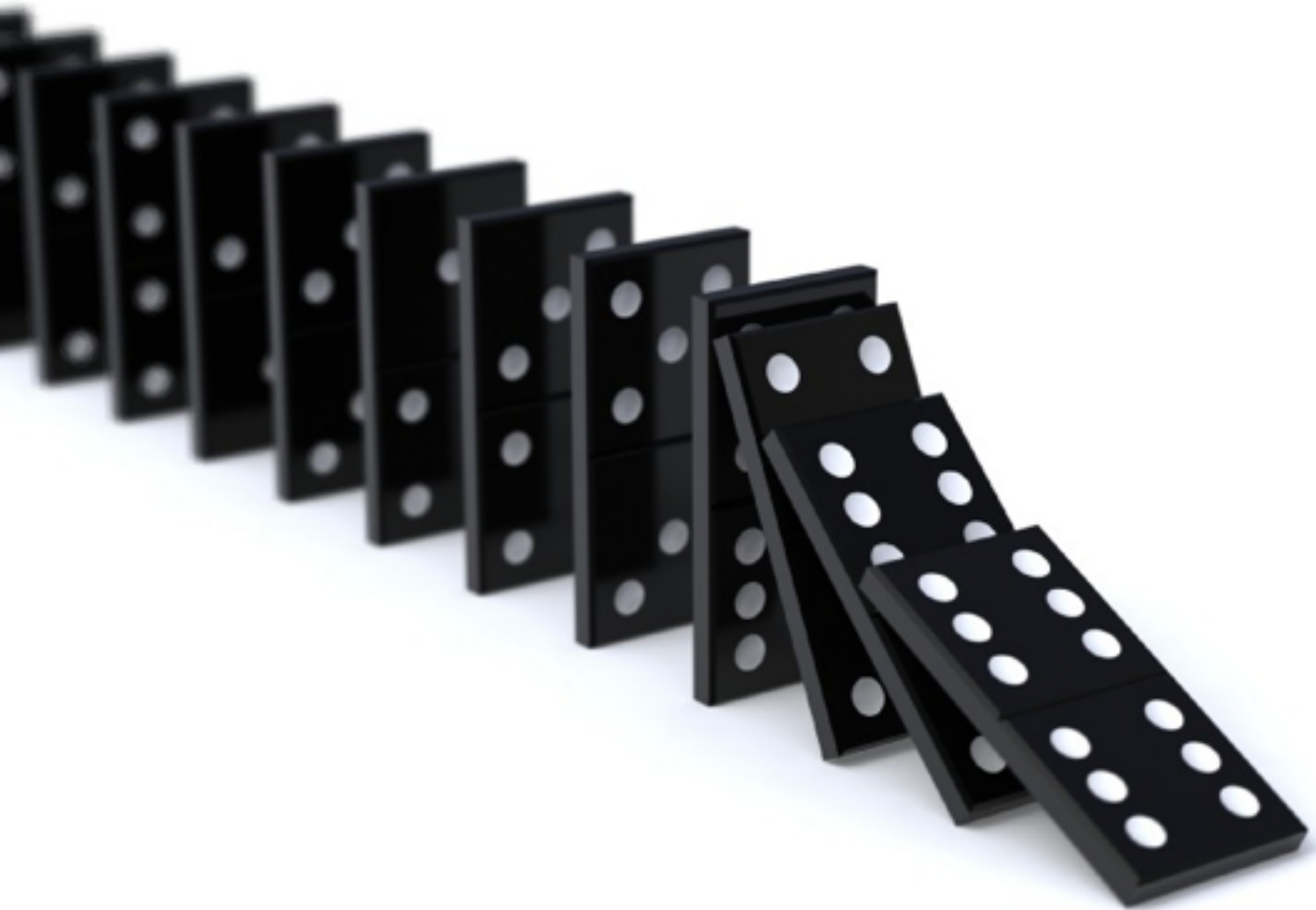
Debugging

where did it go wrong?

actual error

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

error



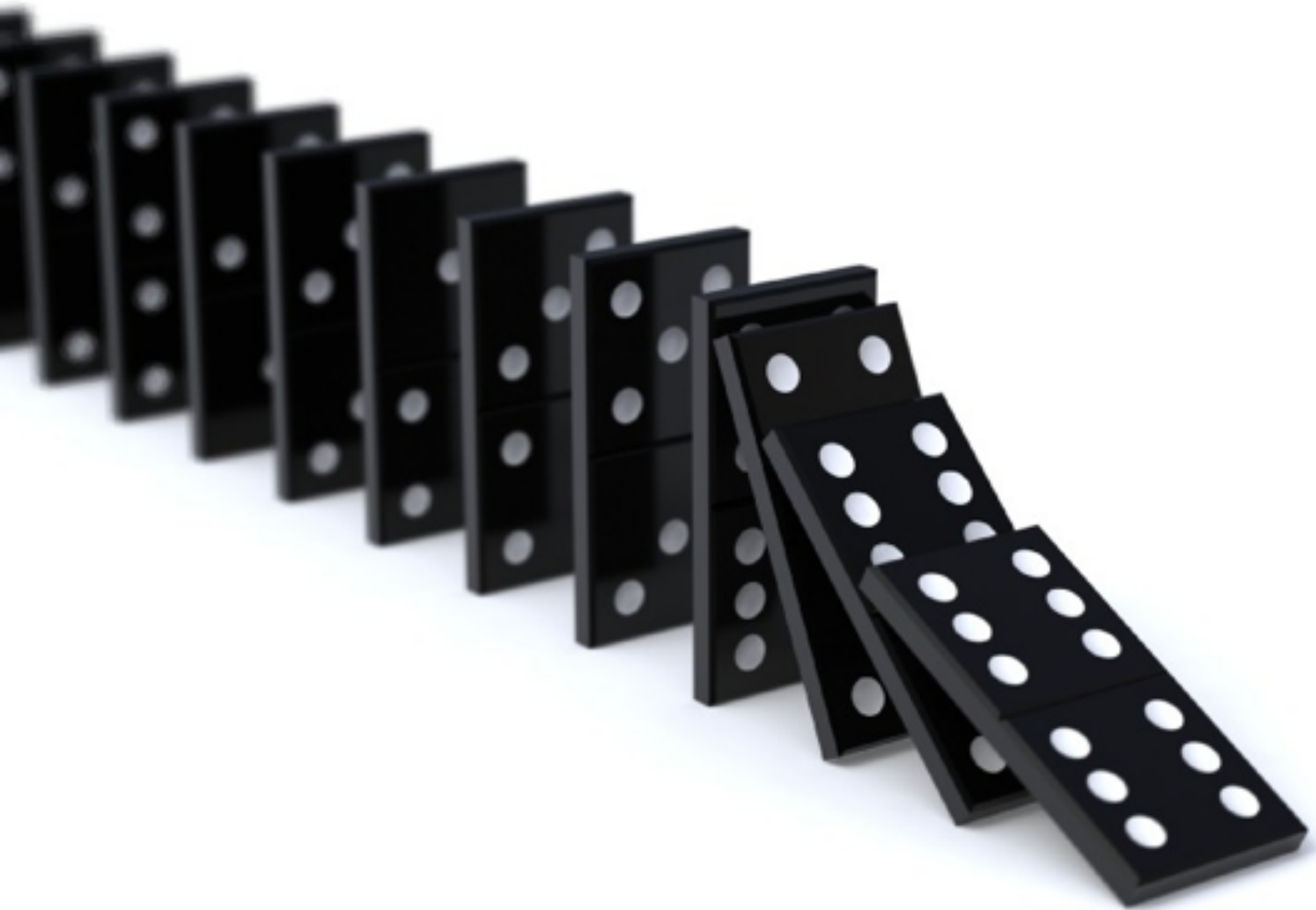
Debugging

where did it go wrong?

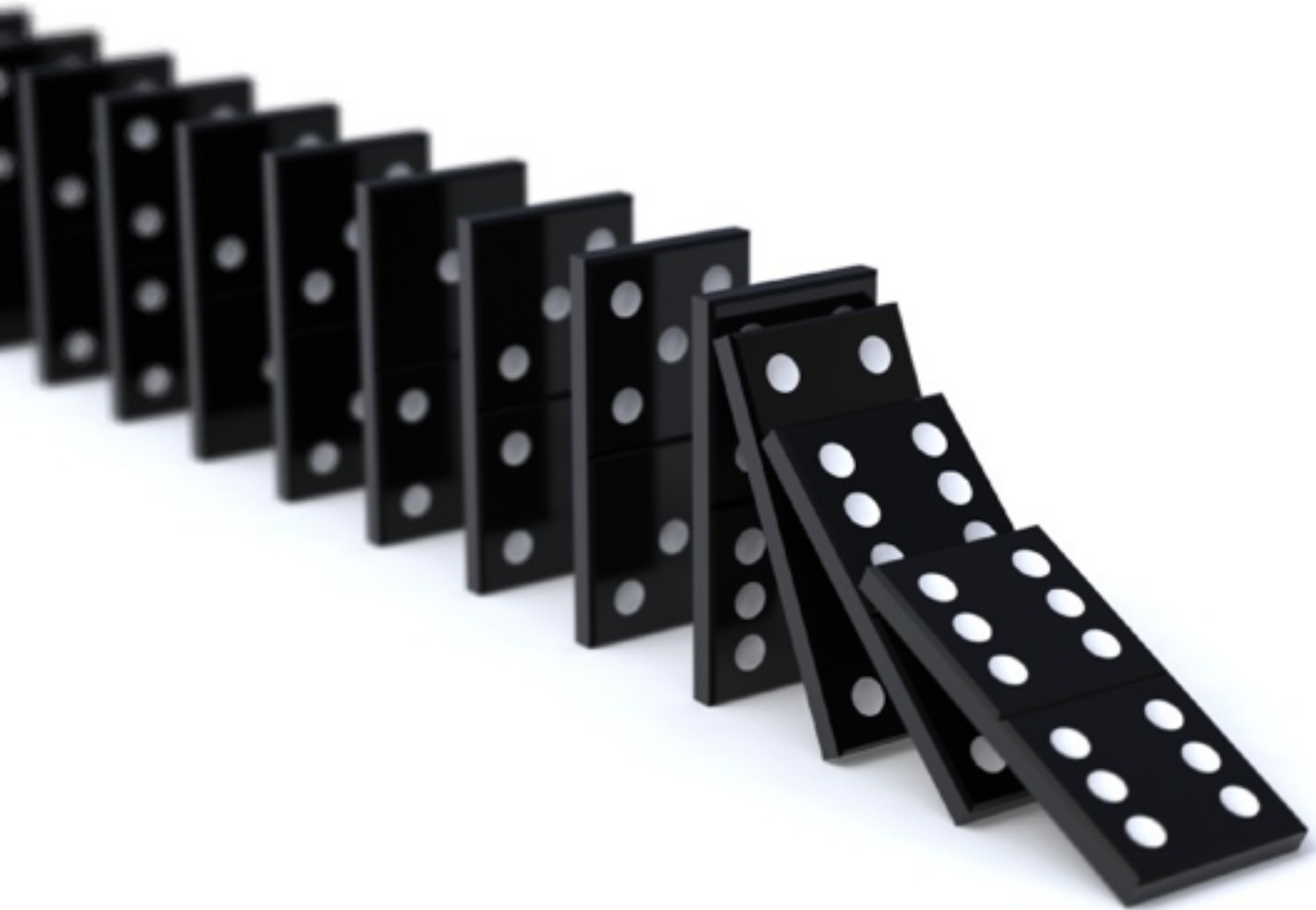
or here

tac mytac := t_x THEN t_2 THEN t_2 THEN t_3

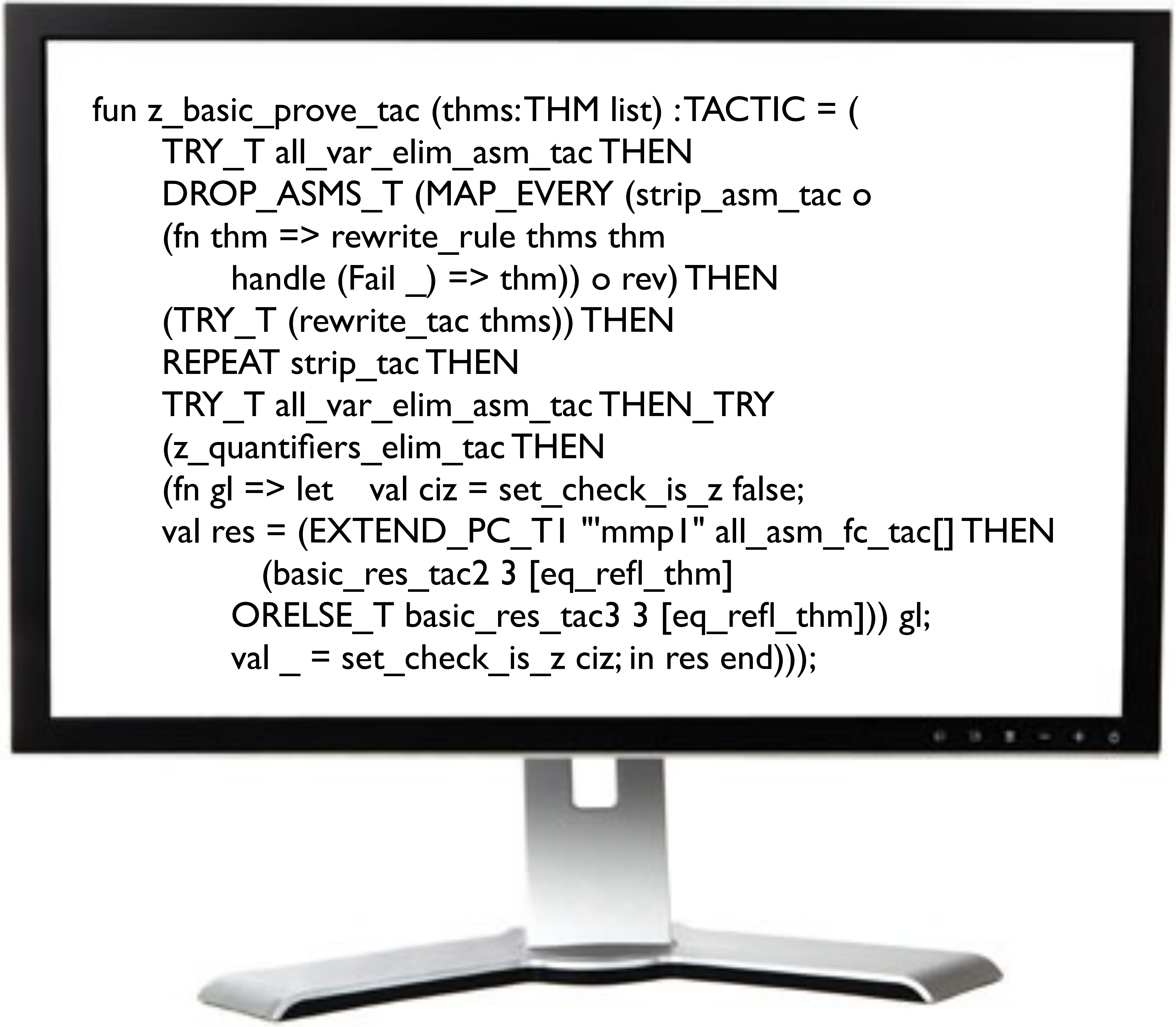
error



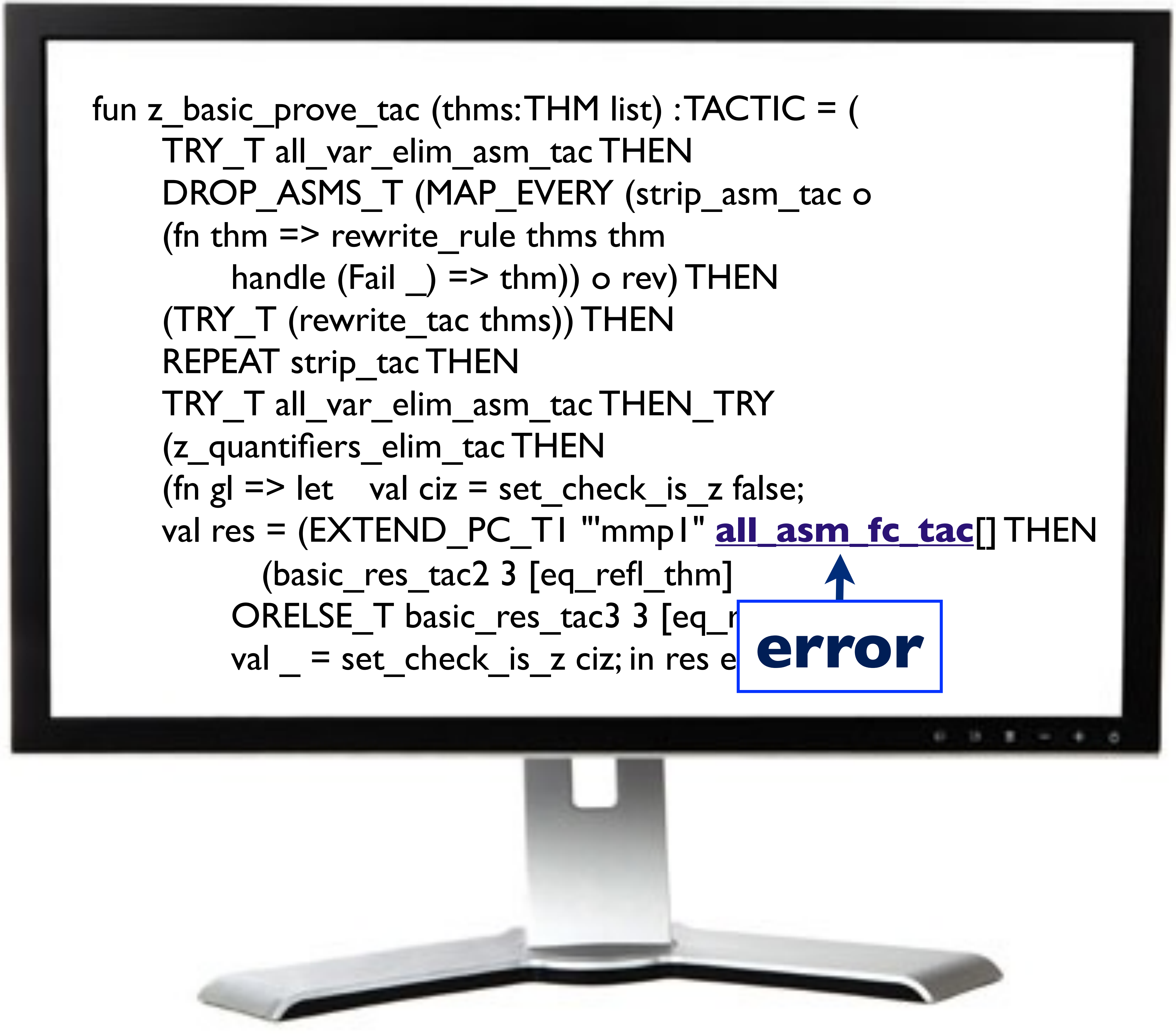
t_2 may also succeed here creating unexpected sub-goals



Bugs may be easy to spot for this example, but what if...

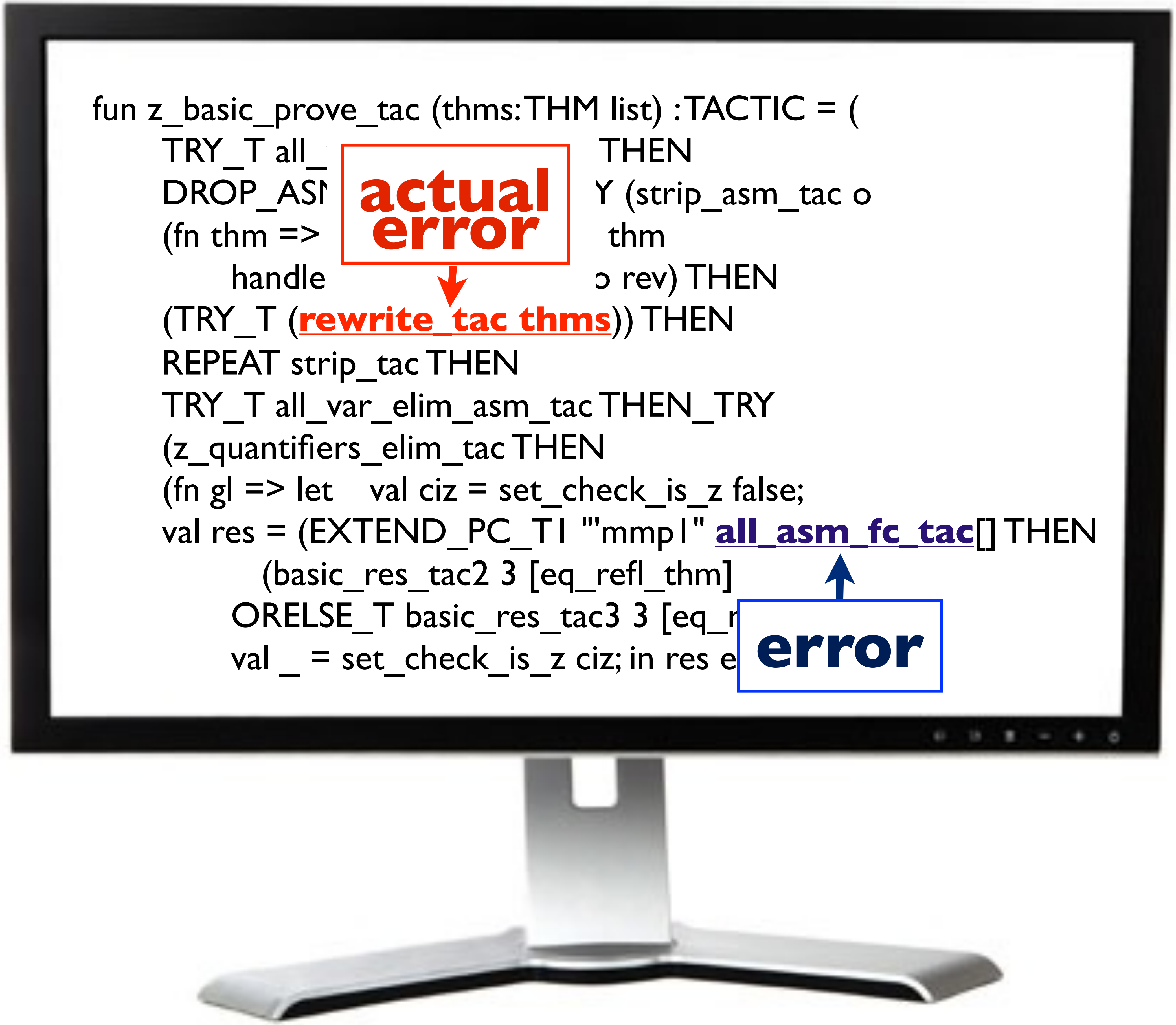


```
fun z_basic_prove_tac (thms:THM list) :TACTIC = (
        TRY_T all_var_elim_asm_tac THEN
        DROP_ASMS_T (MAP_EVERY (strip_asm_tac o
        (fn thm => rewrite_rule thms thm
                handle (Fail _) => thm)) o rev) THEN
        (TRY_T (rewrite_tac thms)) THEN
        REPEAT strip_tac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

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

Composing tactics

No (static) help to stop **plugging** together tactics that do not fit

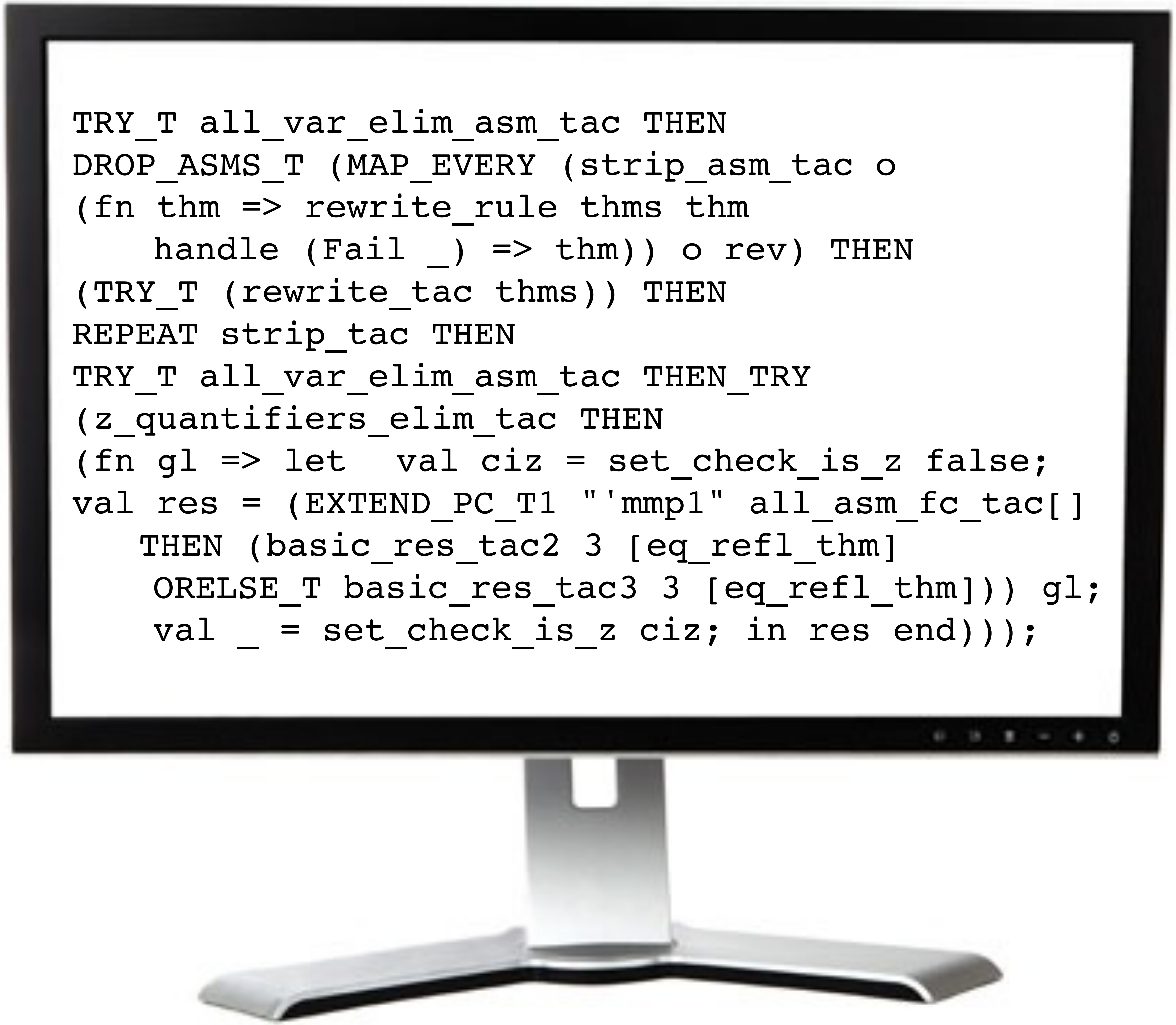
Composing tactics

Brittle since composition
relies on the **number** of goals

Composing tactics

Brittle since composition relies on goal **order**

Instead of...

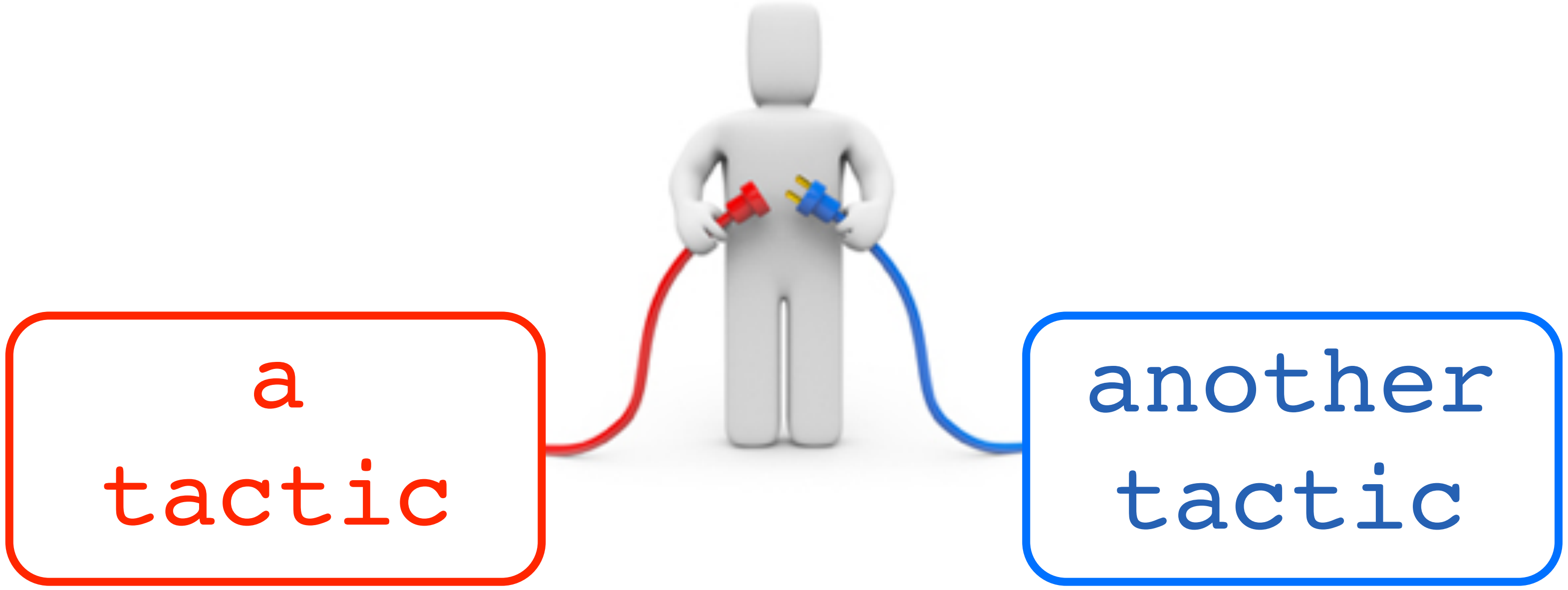


```
TRY_T all_var_elim_asm_tac THEN
DROP_ASMS_T (MAP_EVERY (strip_asm_tac o
(fn thm => rewrite_rule thms thm
    handle (Fail _) => thm)) o rev) THEN
(TRY_T (rewrite_tac thms)) THEN
REPEAT strip_tac THEN
TRY_T all_var_elim_asm_tac THEN_TRY
(z_quantifiers_elim_tac THEN
(fn gl => let  val ciz = set_check_is_z 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



a tactic

another tactic

Loops

Repetition is simply a **feedback** pipe

a looping tactic

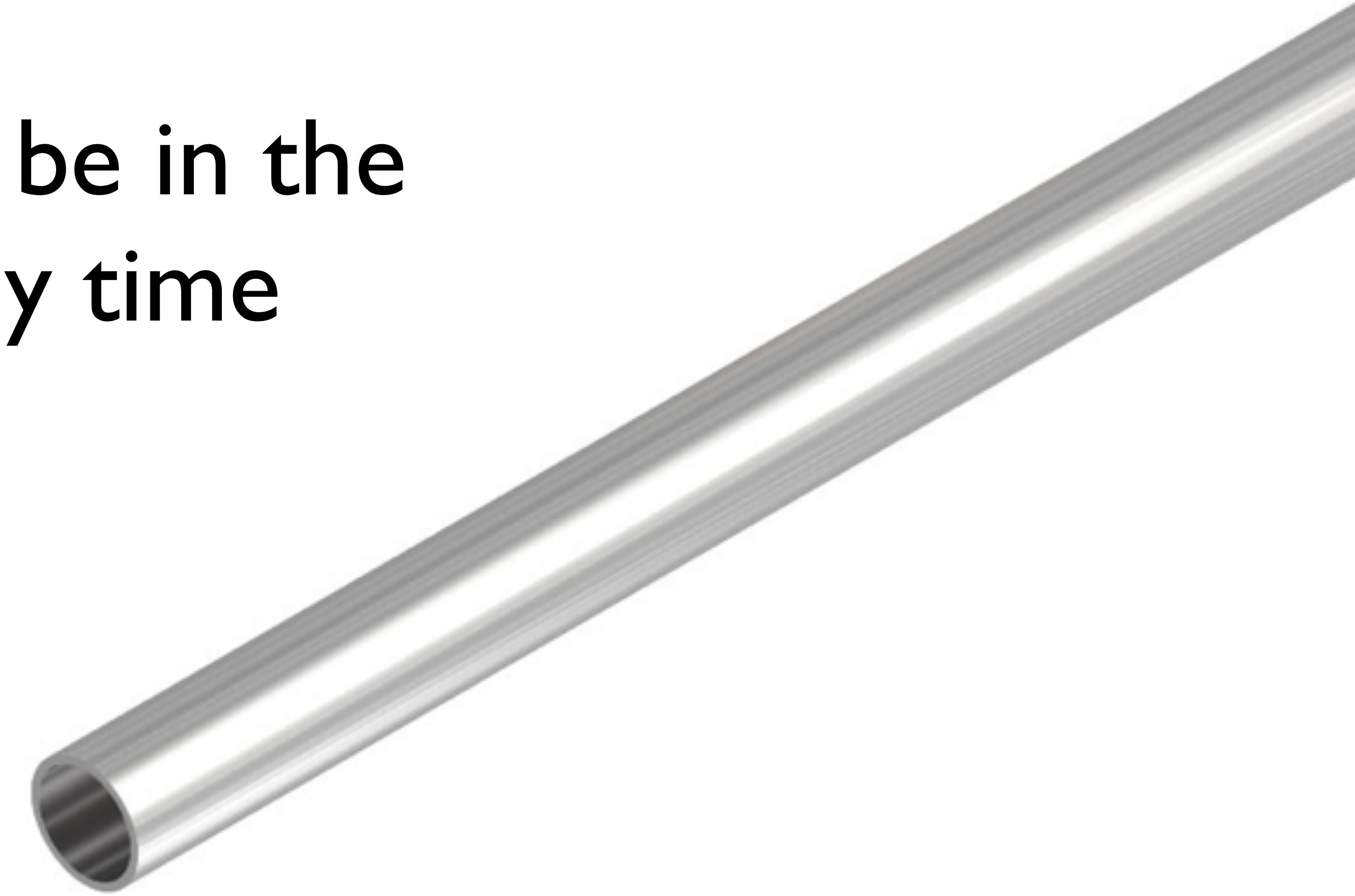
Passing goals

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

A goal must
fit in the pipe it is in

Passing goals

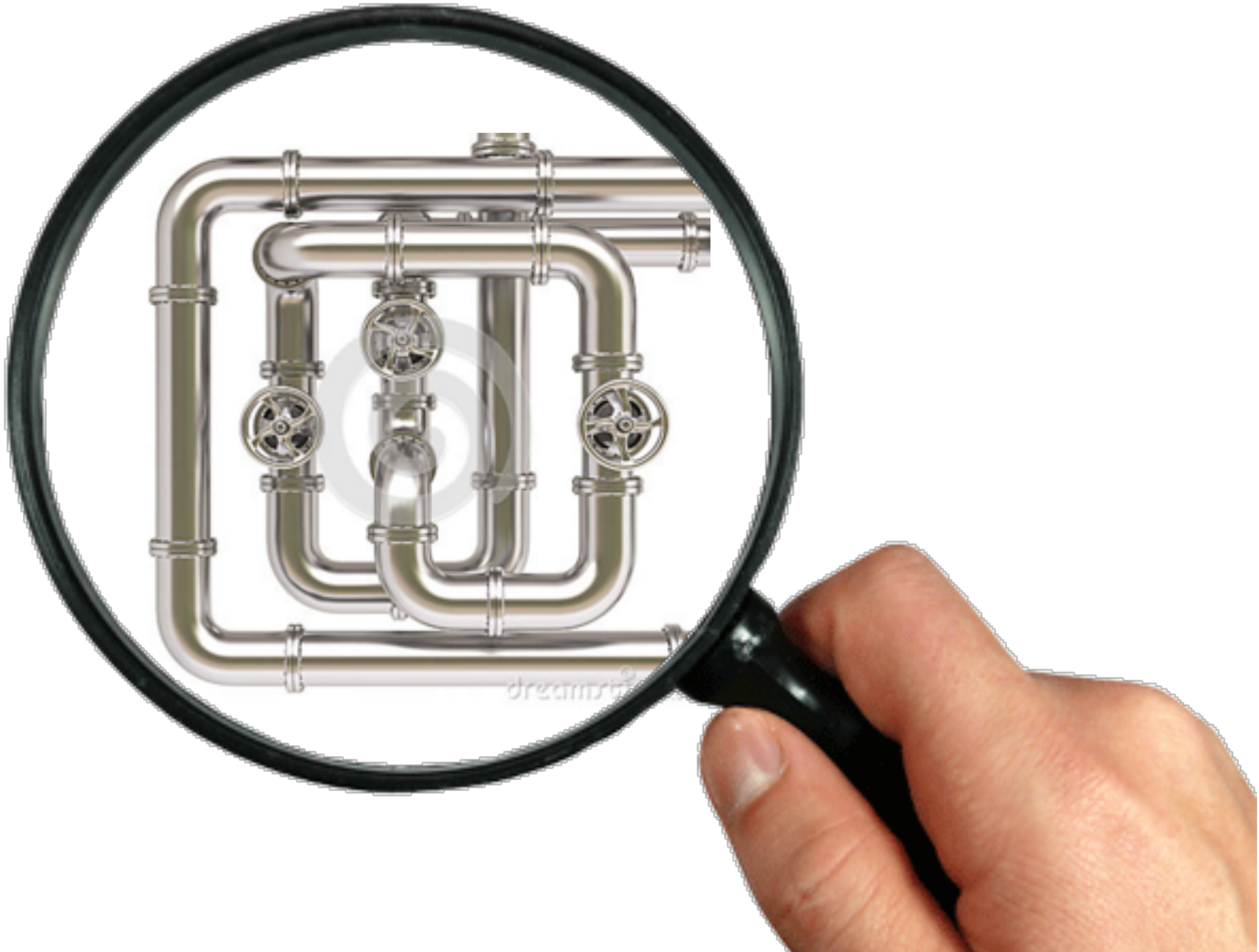
Multiple goals can be in the **same pipe** at any time



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

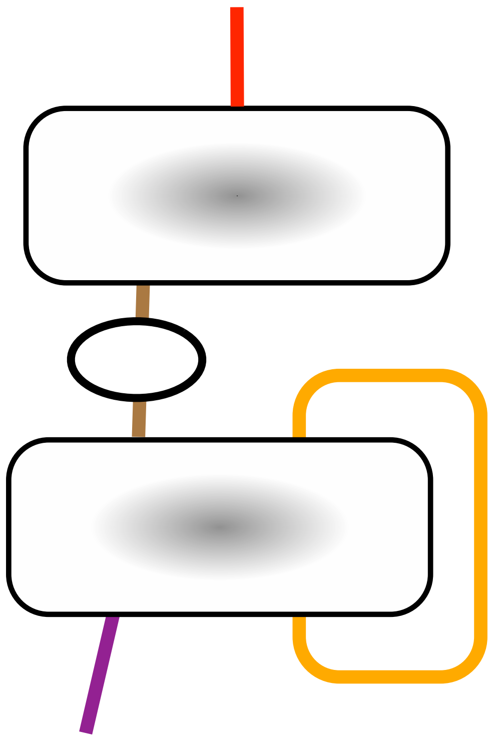
Hierarchies

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



PSGraph

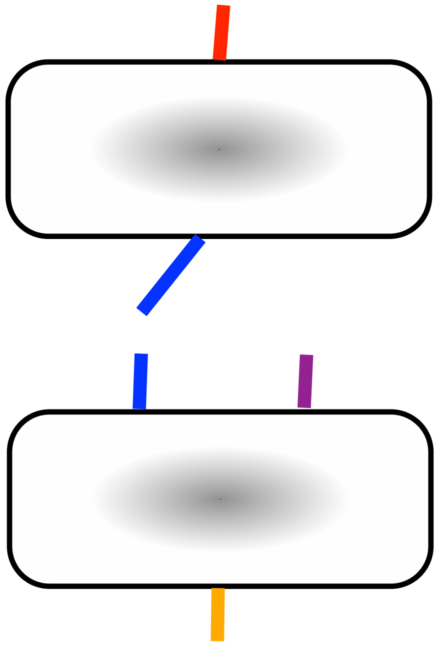
PSGraph formalises proof strategies as pipe networks using **string graphs**



typed graphs with dangling wires

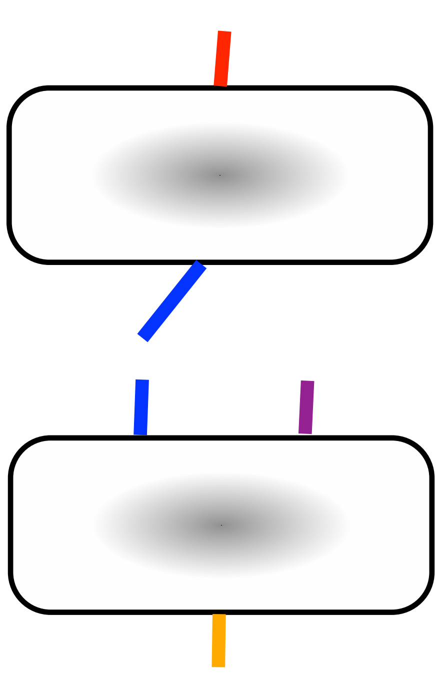
PSGraph composition

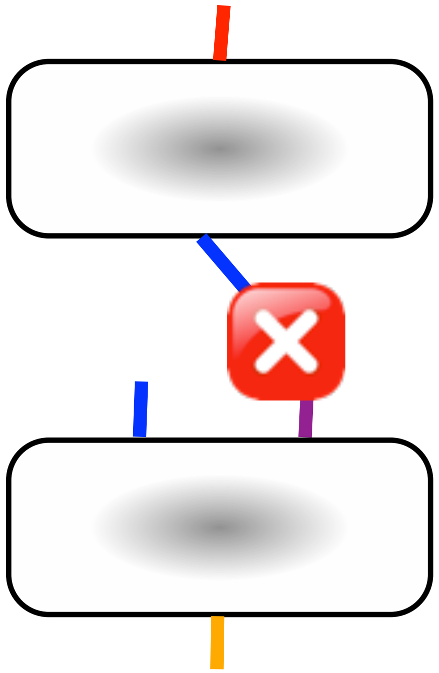
Graphs are composed by **plugging** dangling output wires with dangling input wires



PSGraph composition

Graphs are composed by **plugging** dangling output wires with dangling input wires



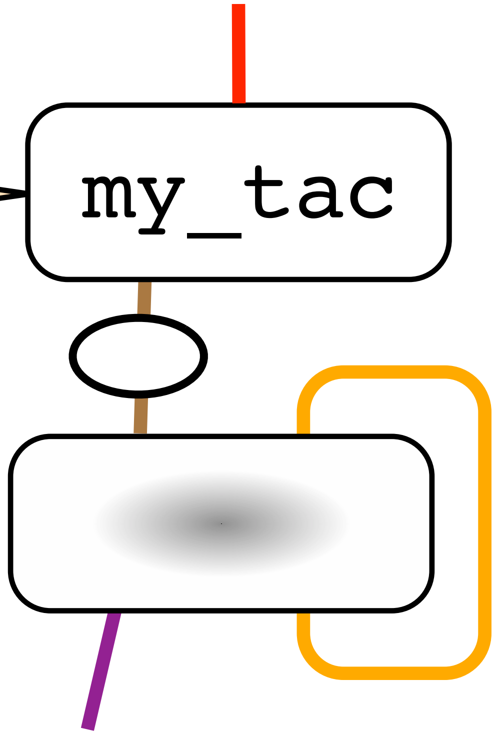


Connecting wires must have **same type**

PSGraph tactics

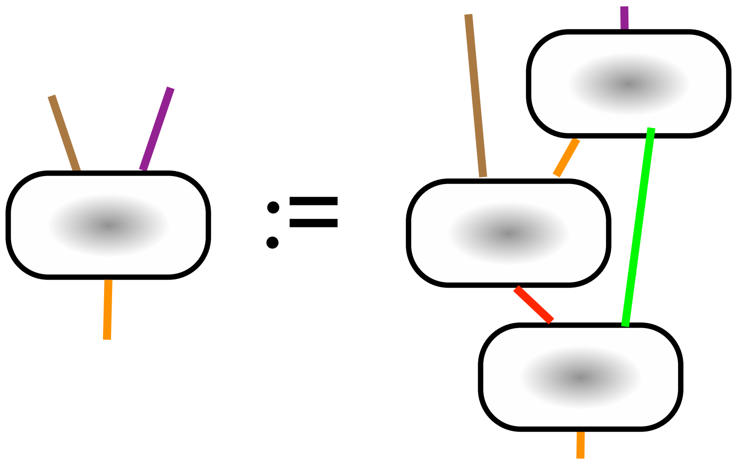
Generic with respect to underlying theorem prover

A node can be an **atomic tactic** of the theorem prover



PSGraph tactics

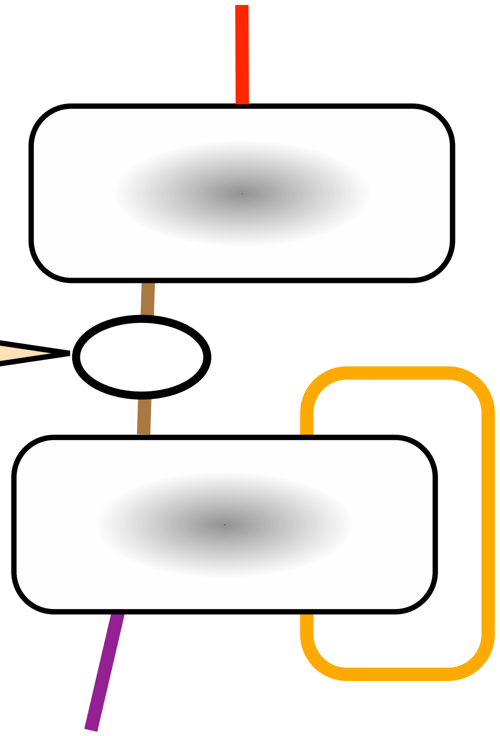
A node can also be a **graph tactic** containing one more graphs



PSGraph evaluation

Token style evaluation where goals are sent over the wires

Represented by a special **goal node** in the graph

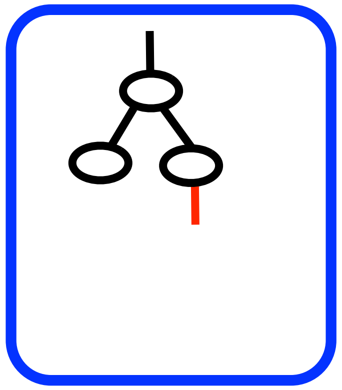


PSGraph evaluation

consume one input goal node
produce new goal nodes on outputs

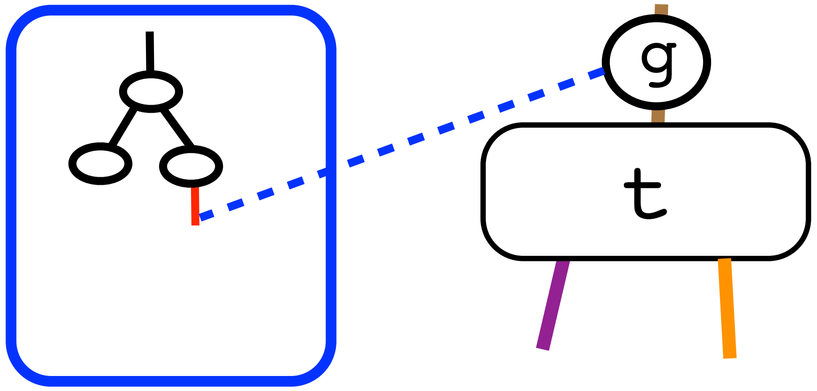
PSGraph evaluation

consume one input goal node
produce new goal nodes on outputs



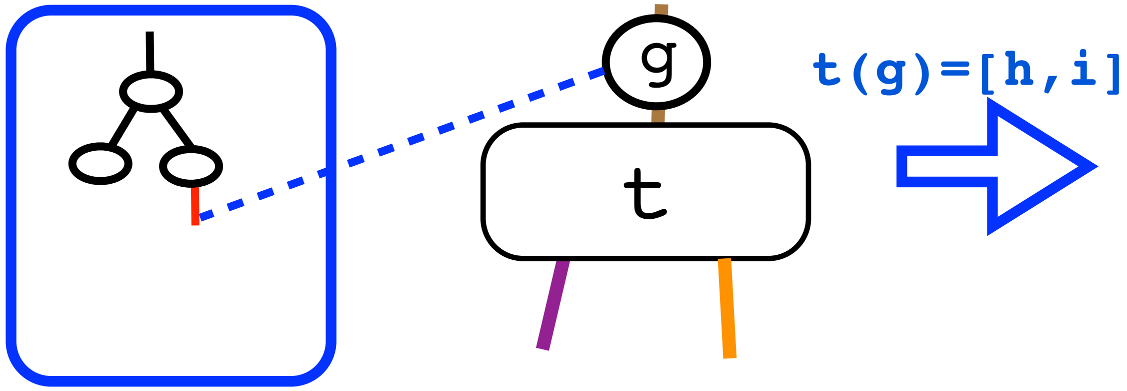
PSGraph evaluation

consume one input goal node
produce new goal nodes on outputs



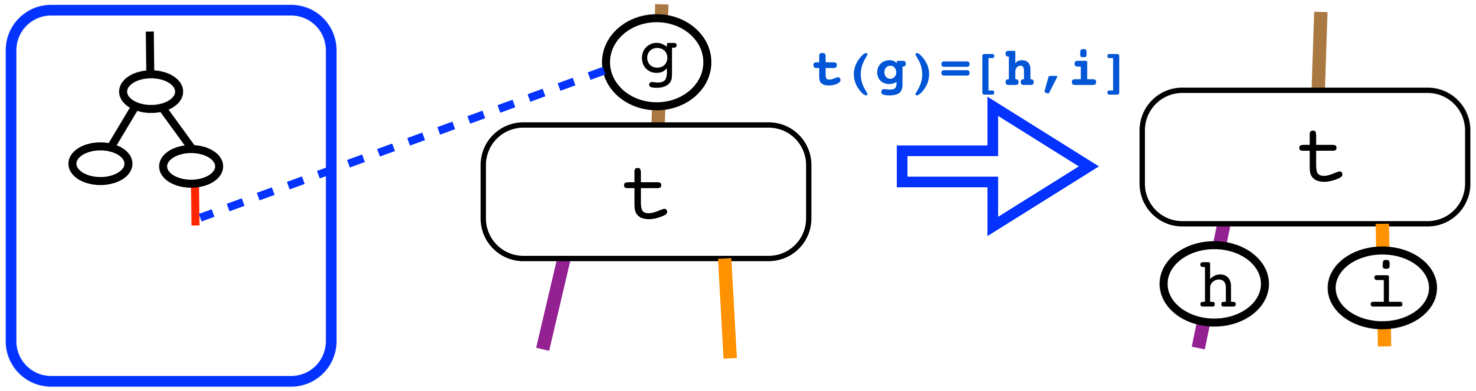
PSGraph evaluation

consume one input goal node
produce new goal nodes on outputs



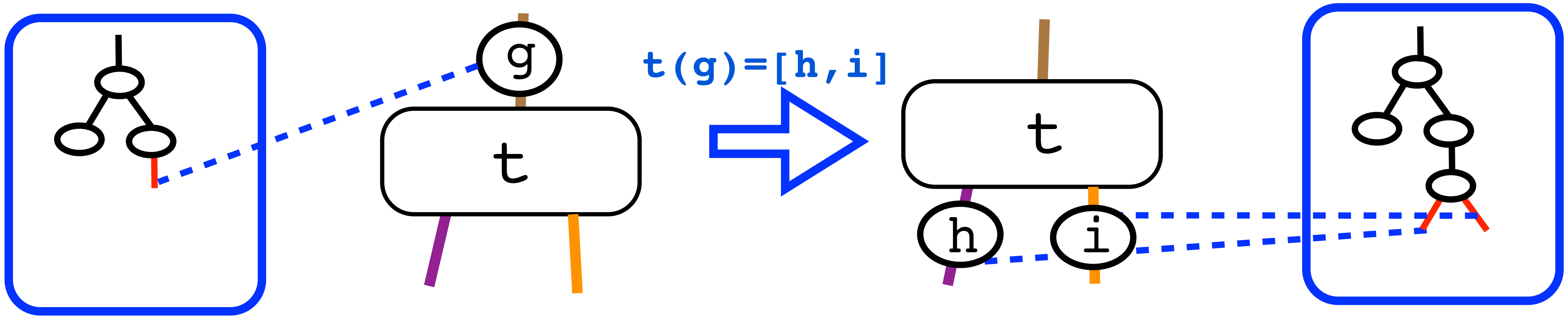
PSGraph evaluation

consume one input goal node
produce new goal nodes on outputs



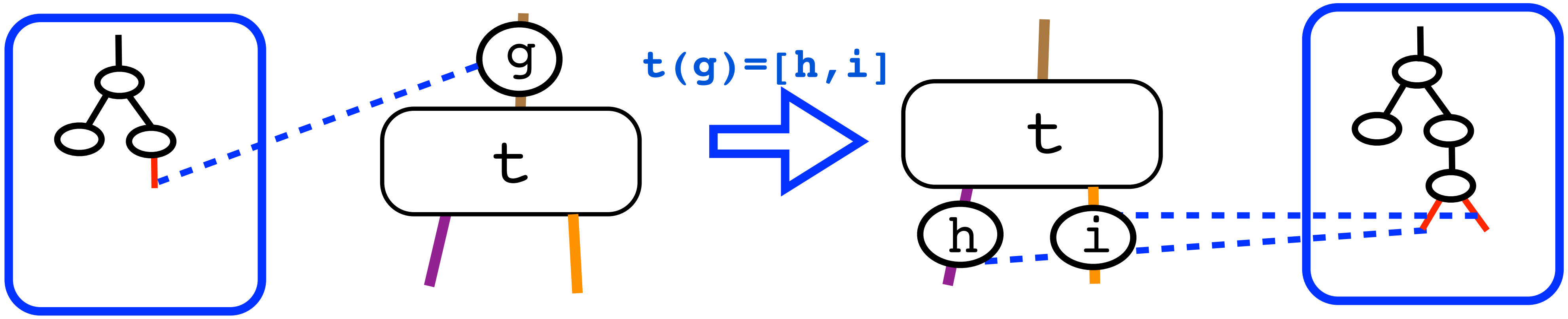
PSGraph evaluation

consume one input goal node
produce new goal nodes on outputs



PSGraph evaluation

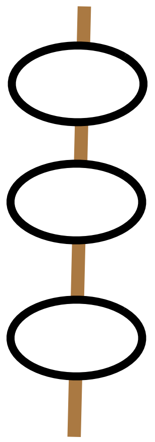
consume one input goal node
produce new goal nodes on outputs



formalised as **graph rewriting**

PSGraph evaluation

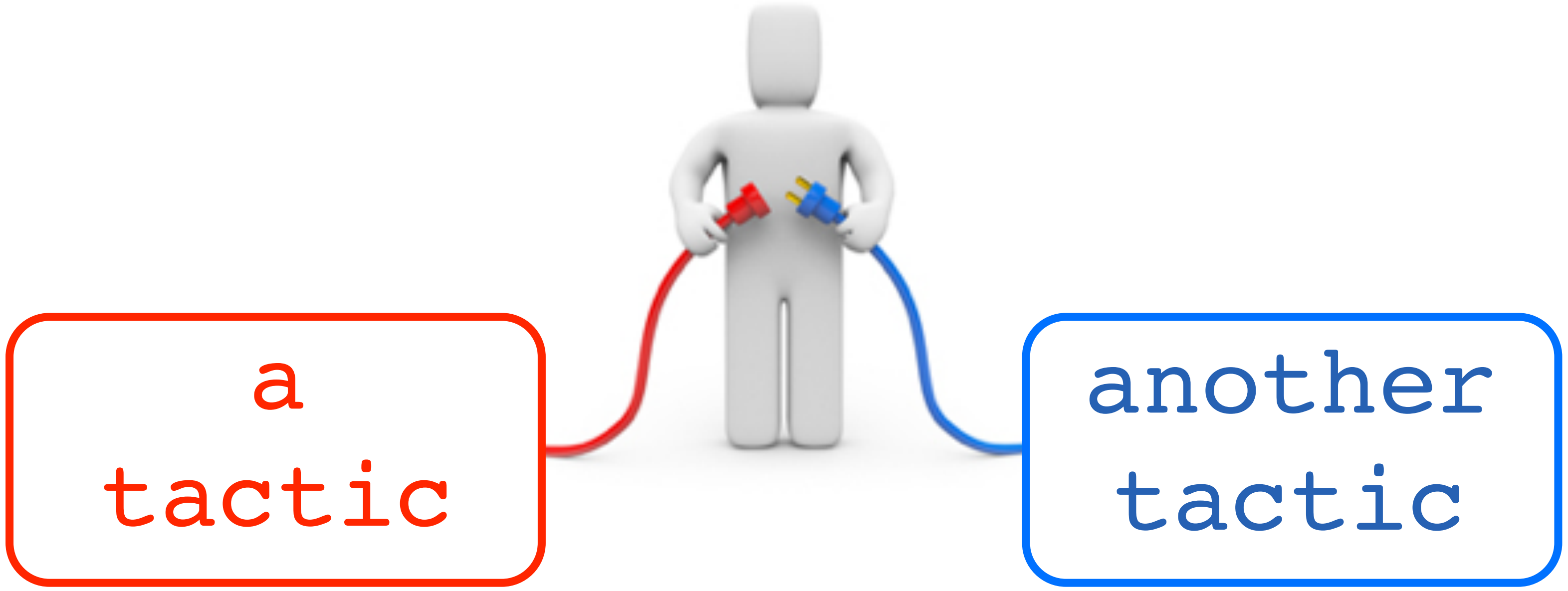
multiple goals may be produced on each output wire



but a goal node must satisfy the **goal type** on that particular wire

Goal types

Predicates on goal nodes to ensure correct **plugging** and **evaluation**



a tactic

another tactic

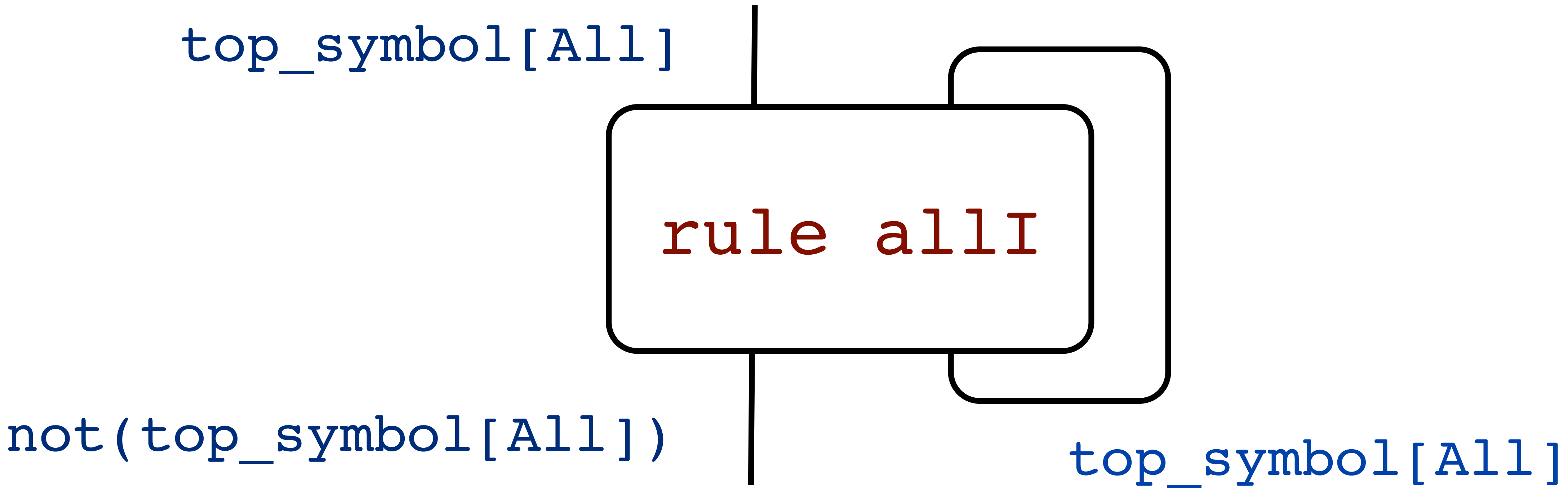
Goal types

PSGraph is **generic** w.r.t goal types. Here is one illustrative example:

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

Example

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



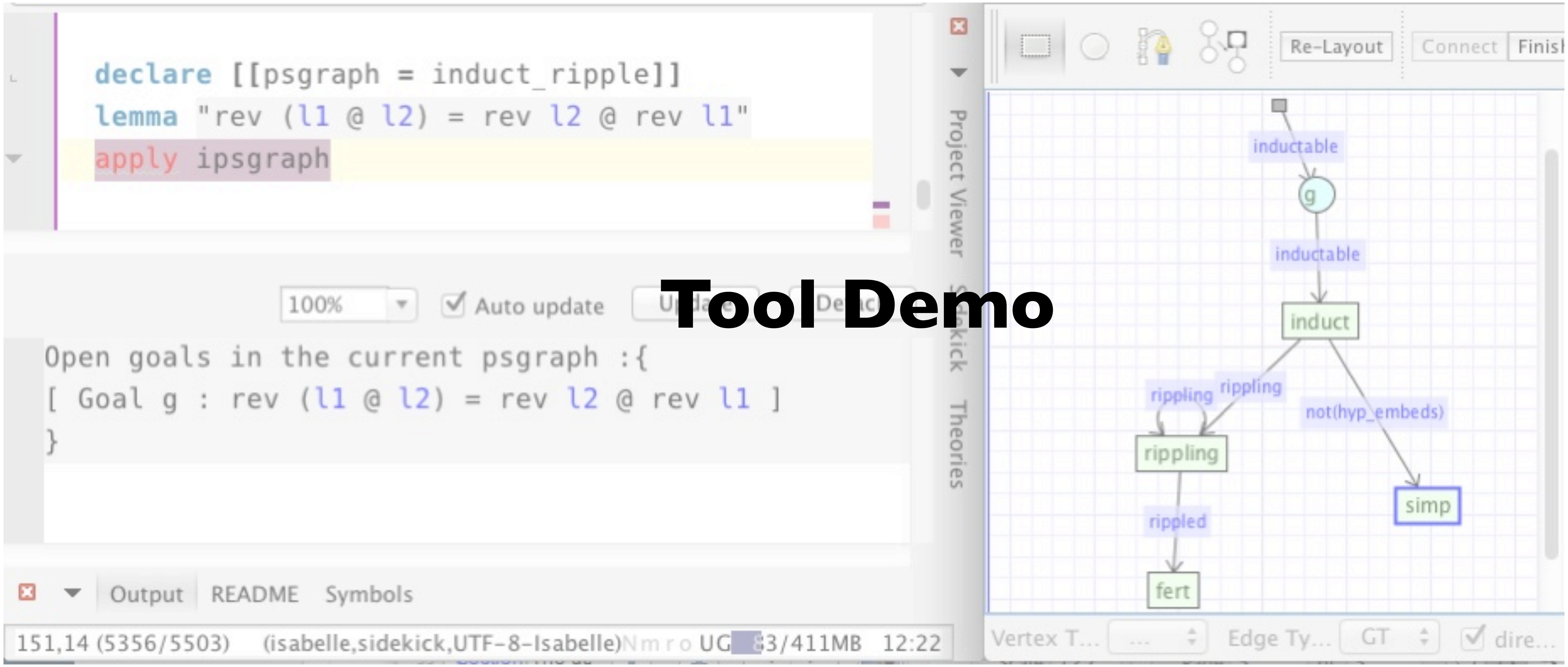
top_symbol[All]

rule allI

not(top_symbol[All])

top_symbol[All]

Tool Demo

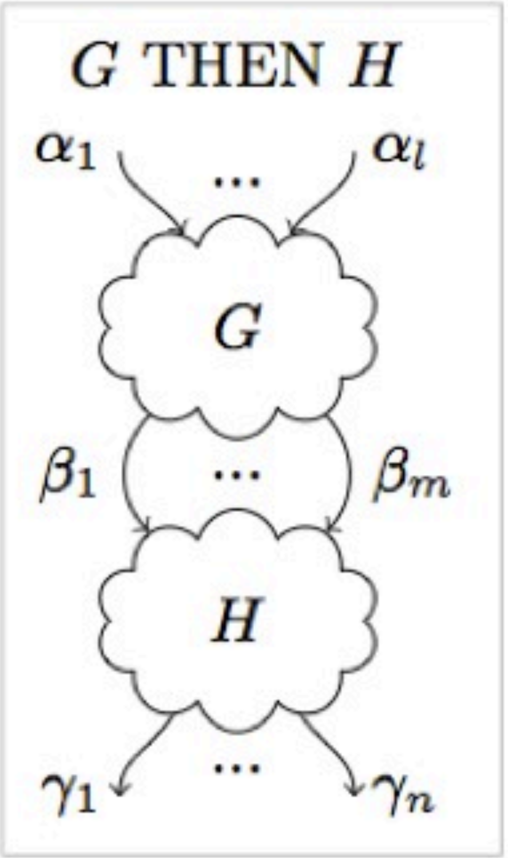


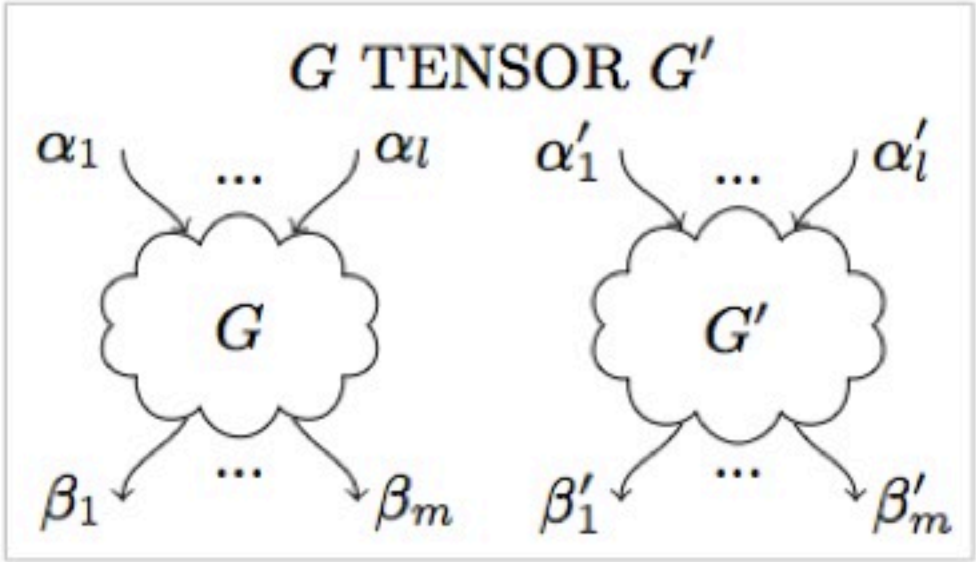
```
declare [[psgraph = induct_ripple]]
lemma "rev (l1 @ l2) = rev l2 @ rev l1"
apply ipsgraph
```

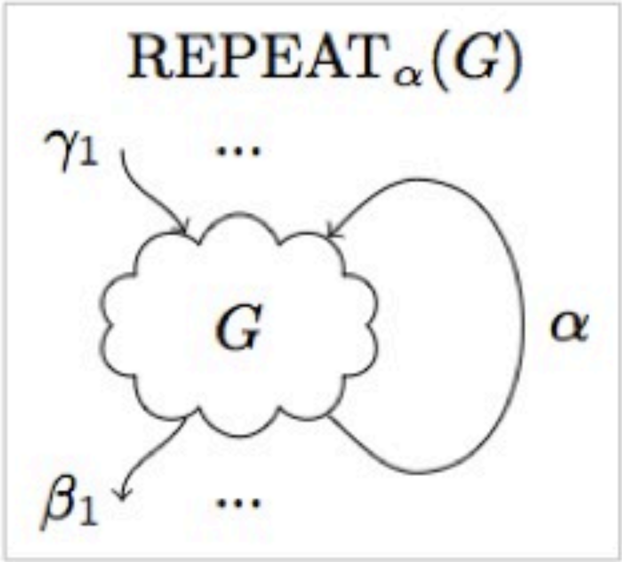
```
Open goals in the current psgraph :{
[ Goal g : rev (l1 @ l2) = rev l2 @ rev l1 ]
}
```

Combinators

Graphs can be **programmed** and combined using **graphical idioms**







Conclusion

PSGraph

proof strategies as **graphs**

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

abstracts over **evaluation order** and **search**

has static **composition** properties

⊢

easier to **debug**, **understand** & **maintain**