Representing “why’s”

a proof language for IsaPlanner

Gudmund Grov (Edinburgh) & **Lucas Dixon** (Google)

Sponsored by: EPSRC funded AI4FM project, Bundy’s platform grants & Google

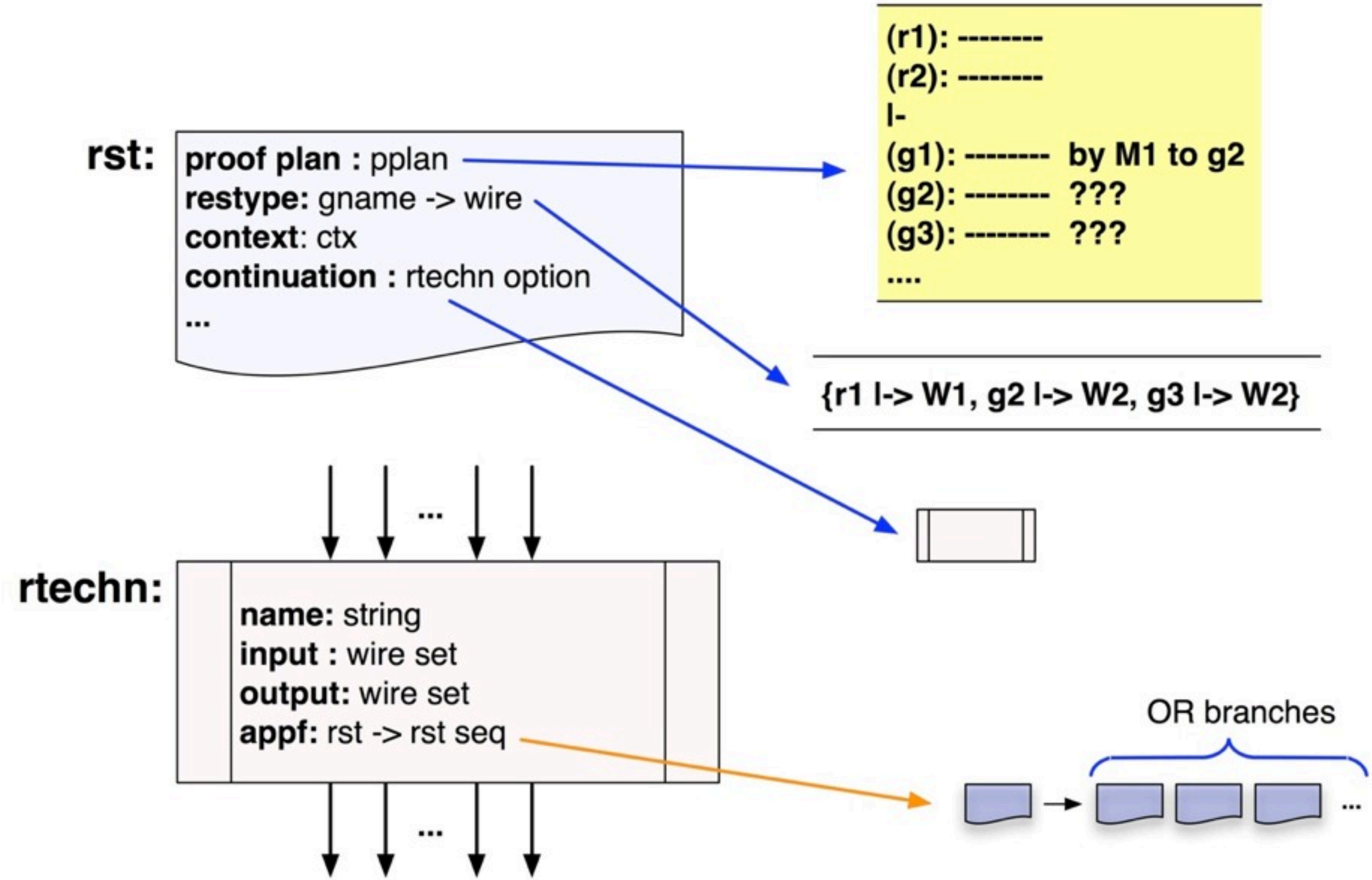
Motivation

- Difficult to combine reasoning techniques (tactics)
  - how to pass around and use goals & results?
- **LCF Tactics** - list of goals (reached by number in list)
  - hard to handle new goals (don't know result goals of tactic)
    - e.g. `apply a 1; apply b 2`
- **IsaPlanner-2** - all goals are named
  - techniques often represented as functions on a goal
  - keeps a list of open/current goals and results
  - not clear which should be open (& no "types of goals")
- **Our goal:** framework to represent "why's"
  - classification & handling of goals is the key
  - build on (refactoring of) existing work i.e. IsaPlanner/Isabelle

IsaPlanner-3

- ***Requirements***
  - clear & simple ~ uniform handling of goals
  - easy to classify goals
  - abstract/simple ~ machine learnable in longer term
- ***Approach***: **boxes** and **wires**
  - a box is a techniques
  - a wire is a goal/result type
    - some I/O wires may be "empty"
  - abstracts over actual goals/results
  - static checking of technique combinations

IsaPlanner-3



Wires

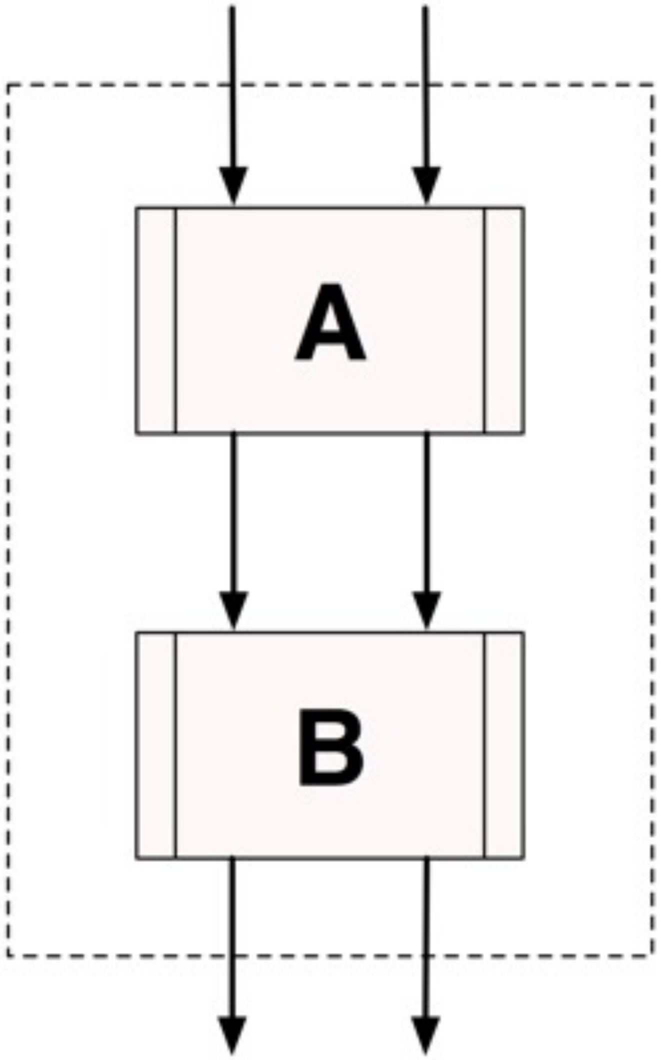
- A wire describes a "type of goal/result"
  - a nice way of classifying them
- Currently represented as strings
- Target has to be "more general" than source
  - partial order on wires as strings with . notation
    - e.g. "A.B" < "A"
- Separate BCK/FWD and AND/OR wires [more later]

FWD/BCK wires

- Techniques on goals are backwards
  - from goals to subgoals
  - linear: goal consumed & new goals created
  - input wire must be consumed
  - Q: what happens with discharged goals?
- Forward application from result to result
  - should be able to reason forwards from same result many times
  - input wire not consumed
- We separate forwards and backwards wires
  - "goal.x" vs "result.x"

Combinators

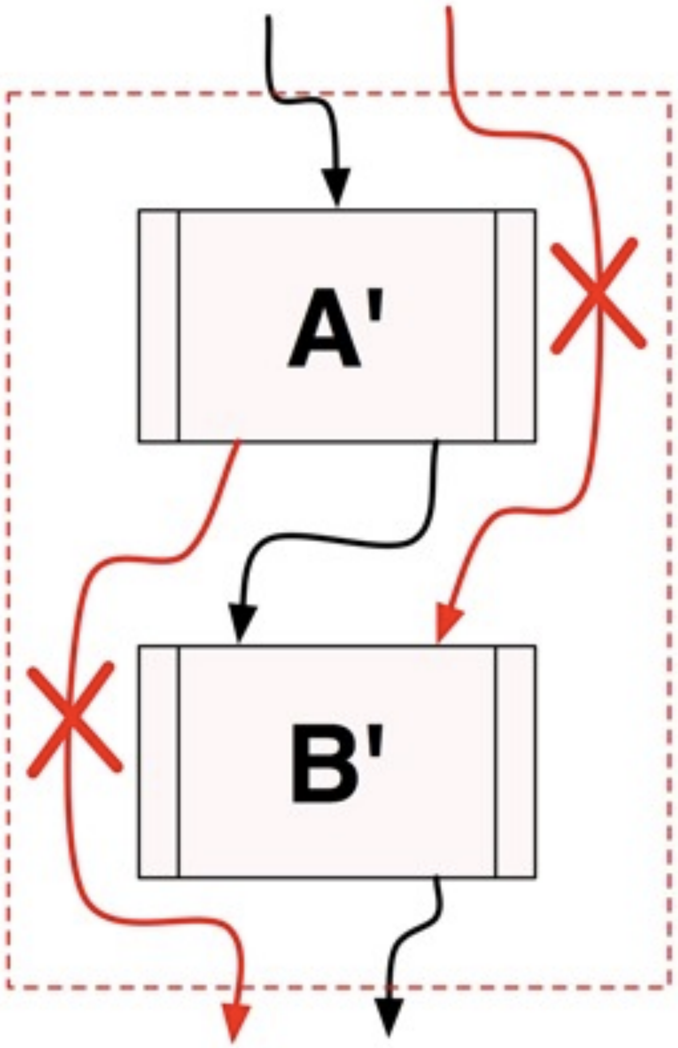
(`A then B`)



- Sequential composition
- I/O type ensured by combinator
  - `input(B) = output(A)` [almost]
  - `input(A then B) = input(A)`
  - `output(A then B) = output(B)`
- Composition/separation clear

Combinators

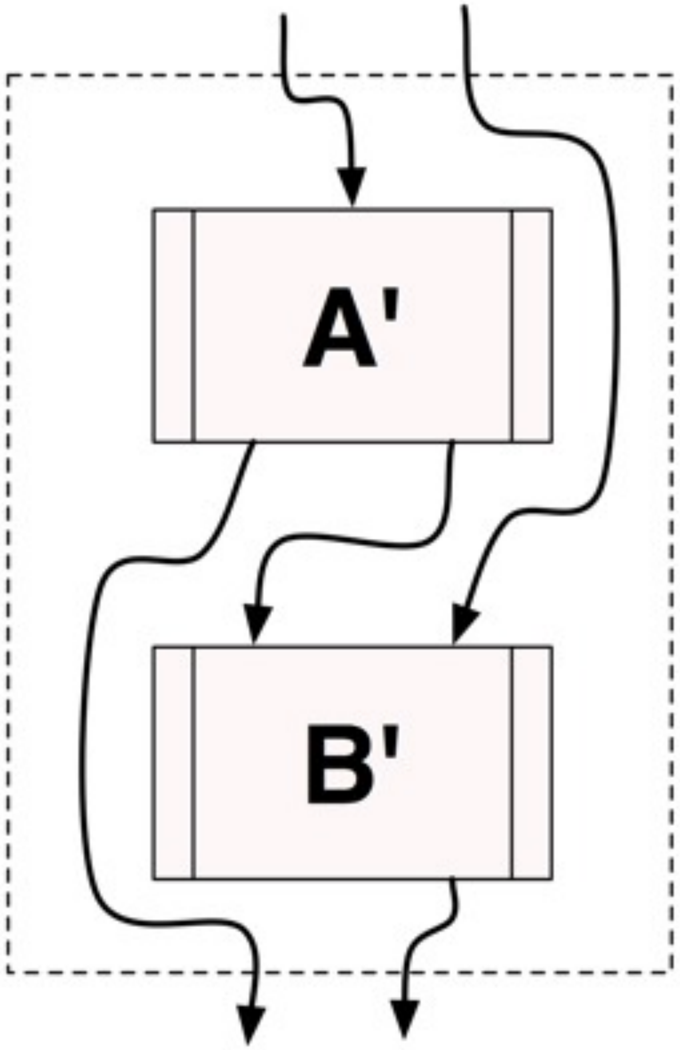
(A then B)



- Sequential composition
- I/O type ensured by combinator
  - `input(B) = output(A)` [almost]
  - `input(A then B) = input(A)`
  - `output(A then B) = output(B)`
- Composition/separation clear

Combinators

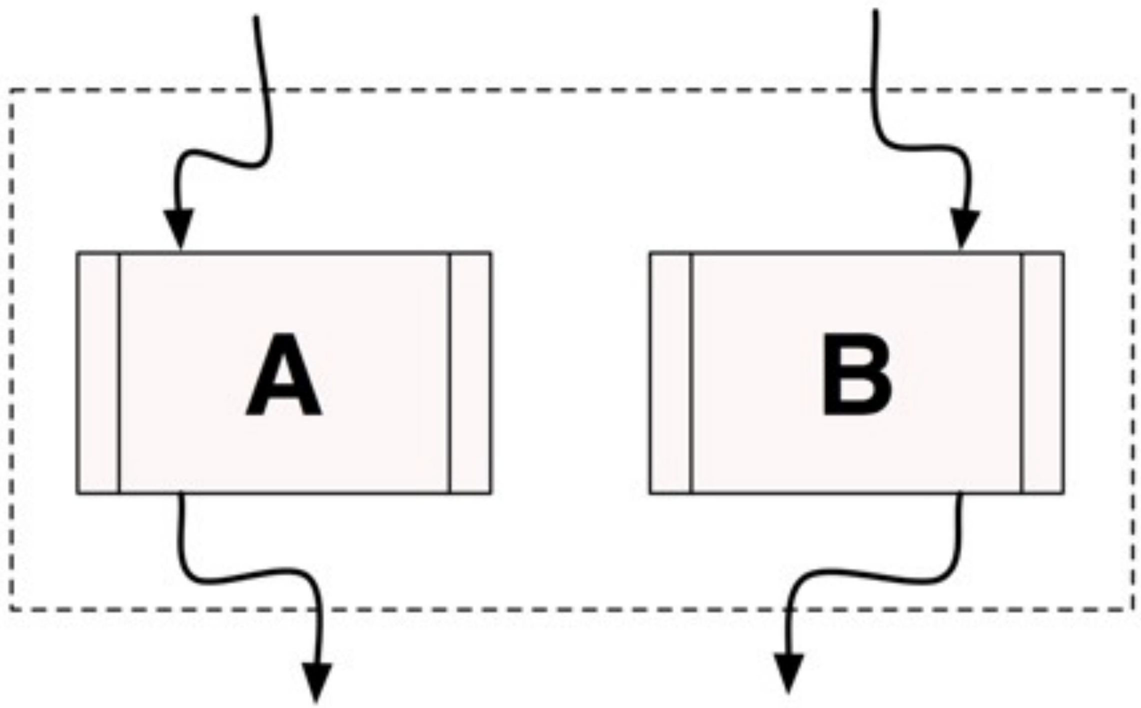
(A compose B)



- Generalises then
- Allows bypassing of wires
    - composition/separation less clear
- input(A’ compose B’) = input(A’) + (input(B’) - output(A’))
- output(A’ compose B’) = output(B’) + (output(A’) - input(B’))

Combinators

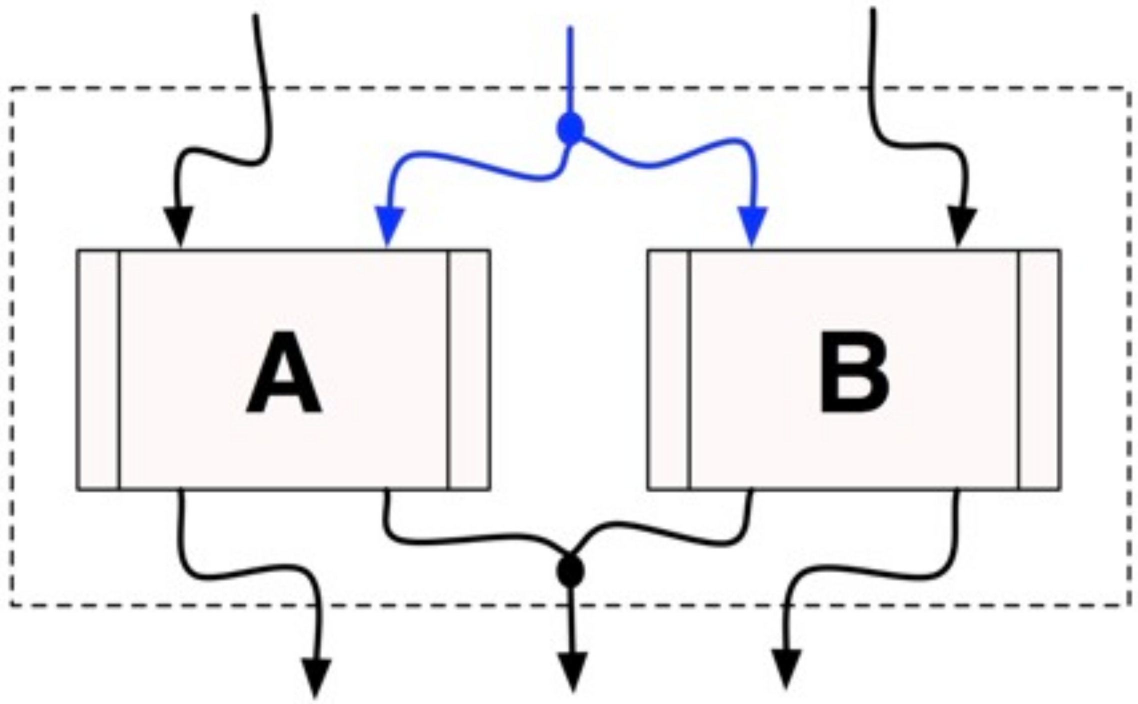
(`A tensor B`)



- Symmetric: `A tensor B = B tensor A`
  - can be parallelized
  - evaluation:
    - "run as in parallel (on same input) - combine results"

Combinators

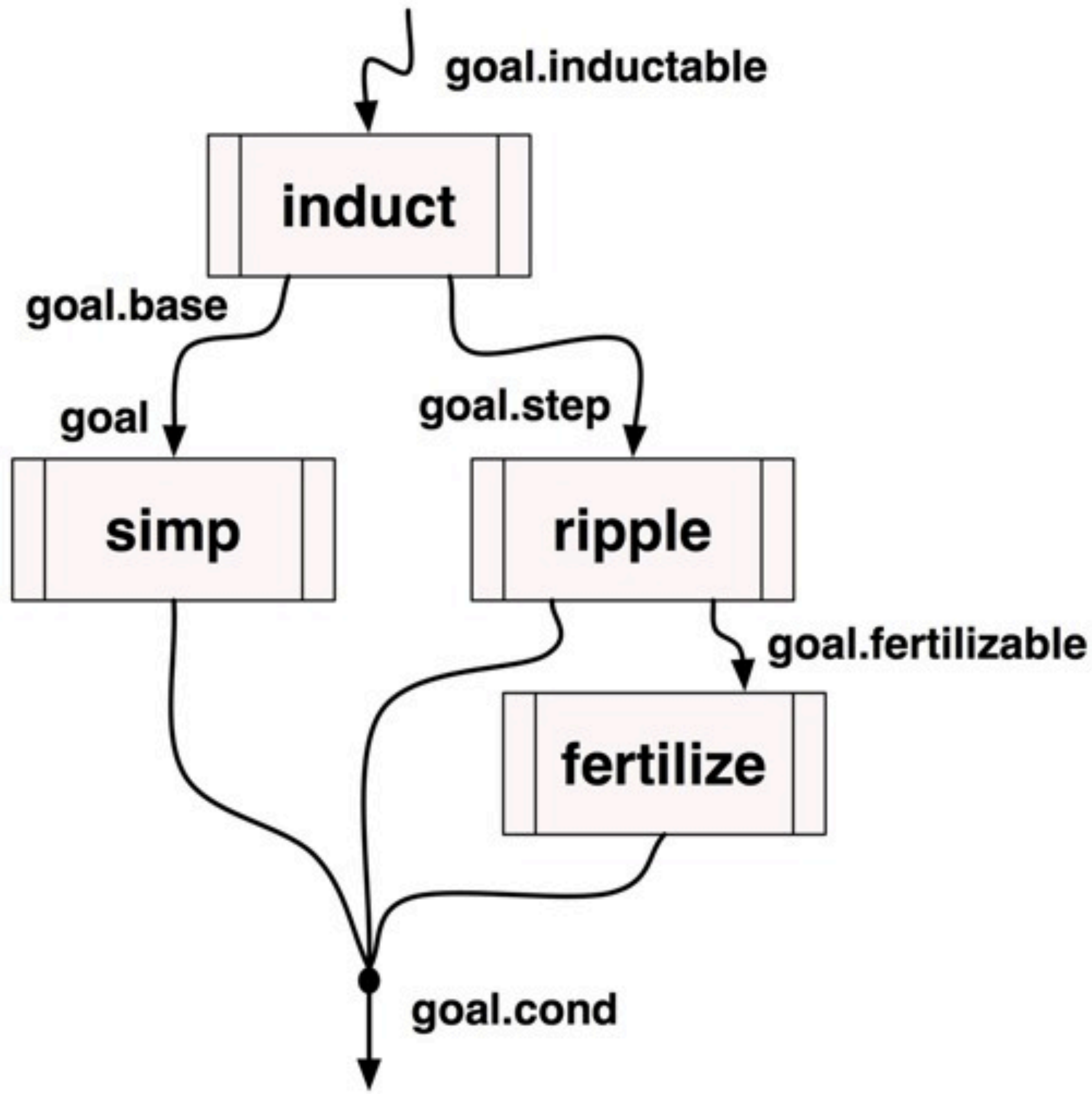
(A tensor B)



- Joint input (blue wire): only if fwd
  - or no meta variables
- Backward input wires must be disjoint
- no such requirement of output (due to eval)

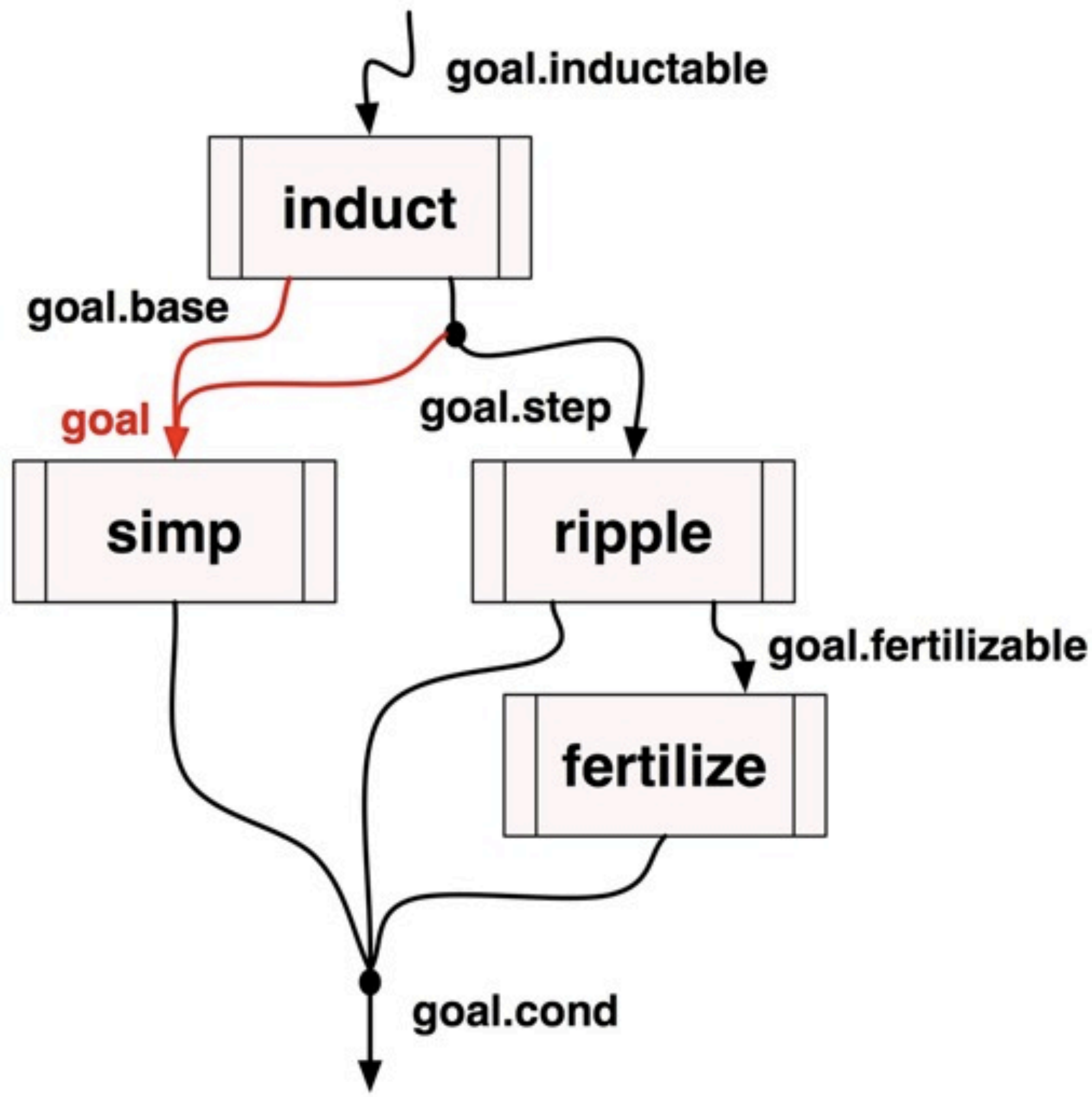
Example (BCK only)

induct <u>then</u> (**simp** <u>tensor</u> (**ripple** <u>compose</u> **fertilize**))



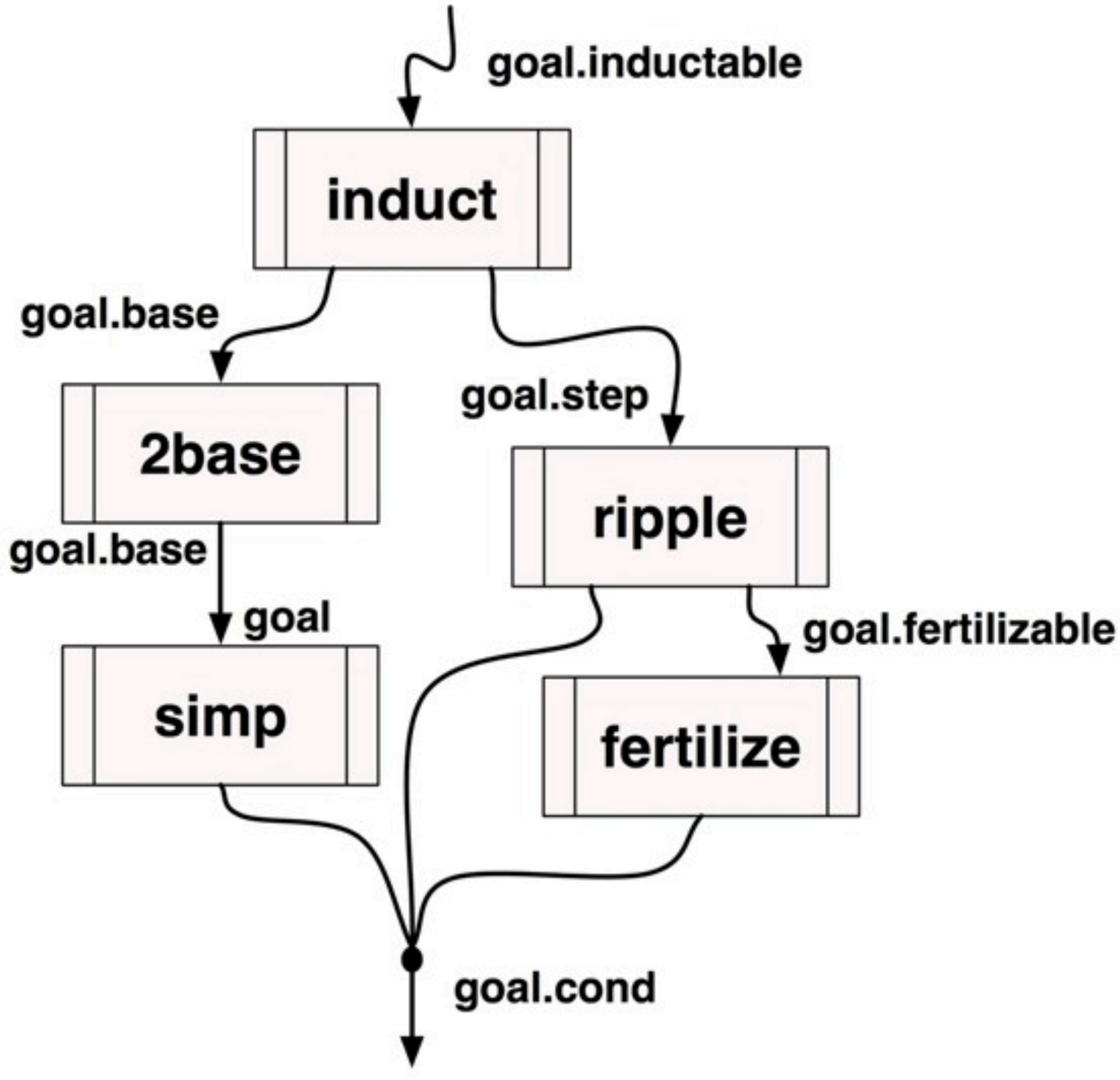
Example (BCK only)

induct then (**simp** tensor (**ripple** compose **fertilize**))



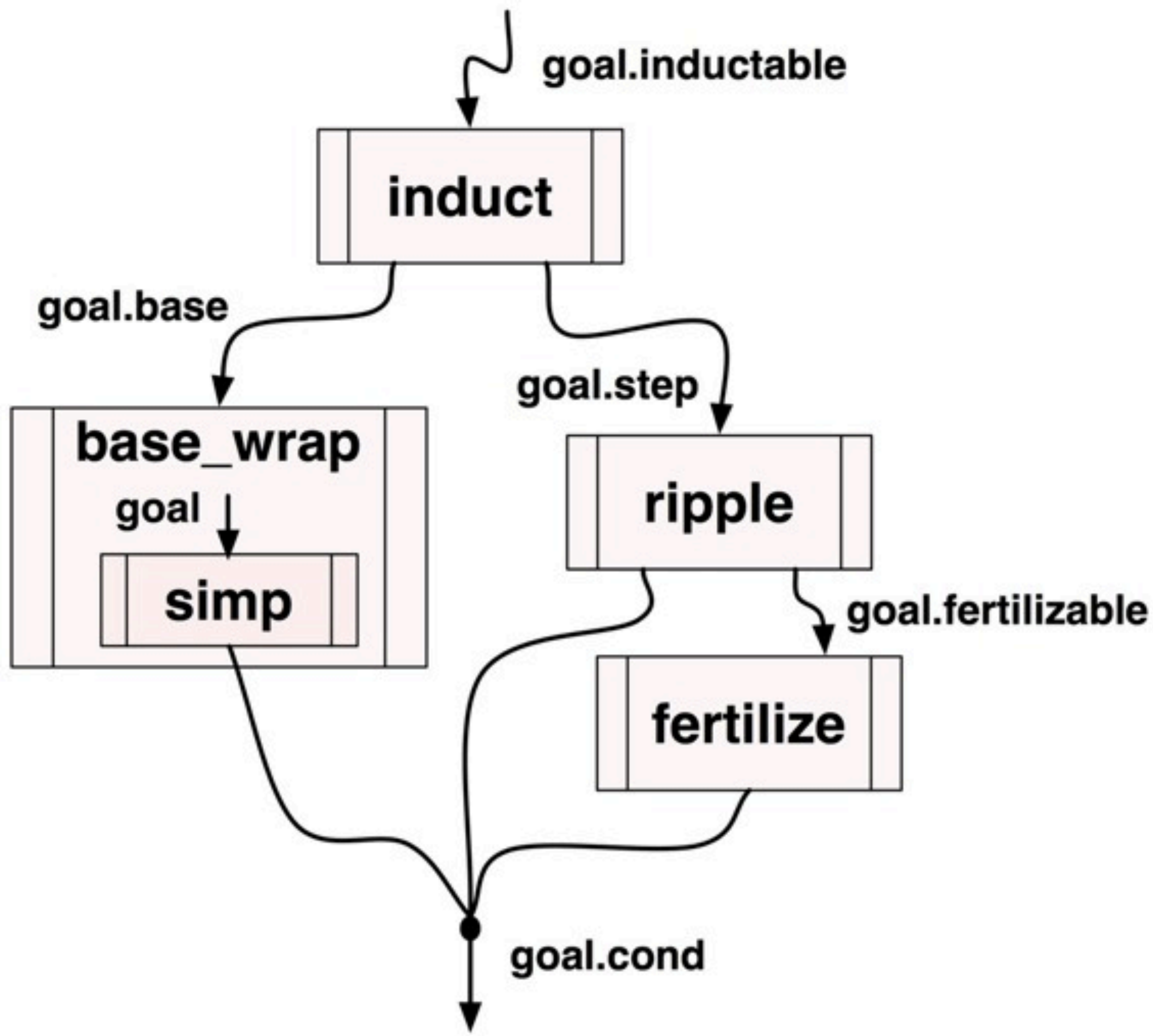
Example (BCK only)

induct <u>then</u> ((**2base** <u>then</u> **simp**)
<u>tensor</u> (**ripple** <u>compose</u> **fertilize**))

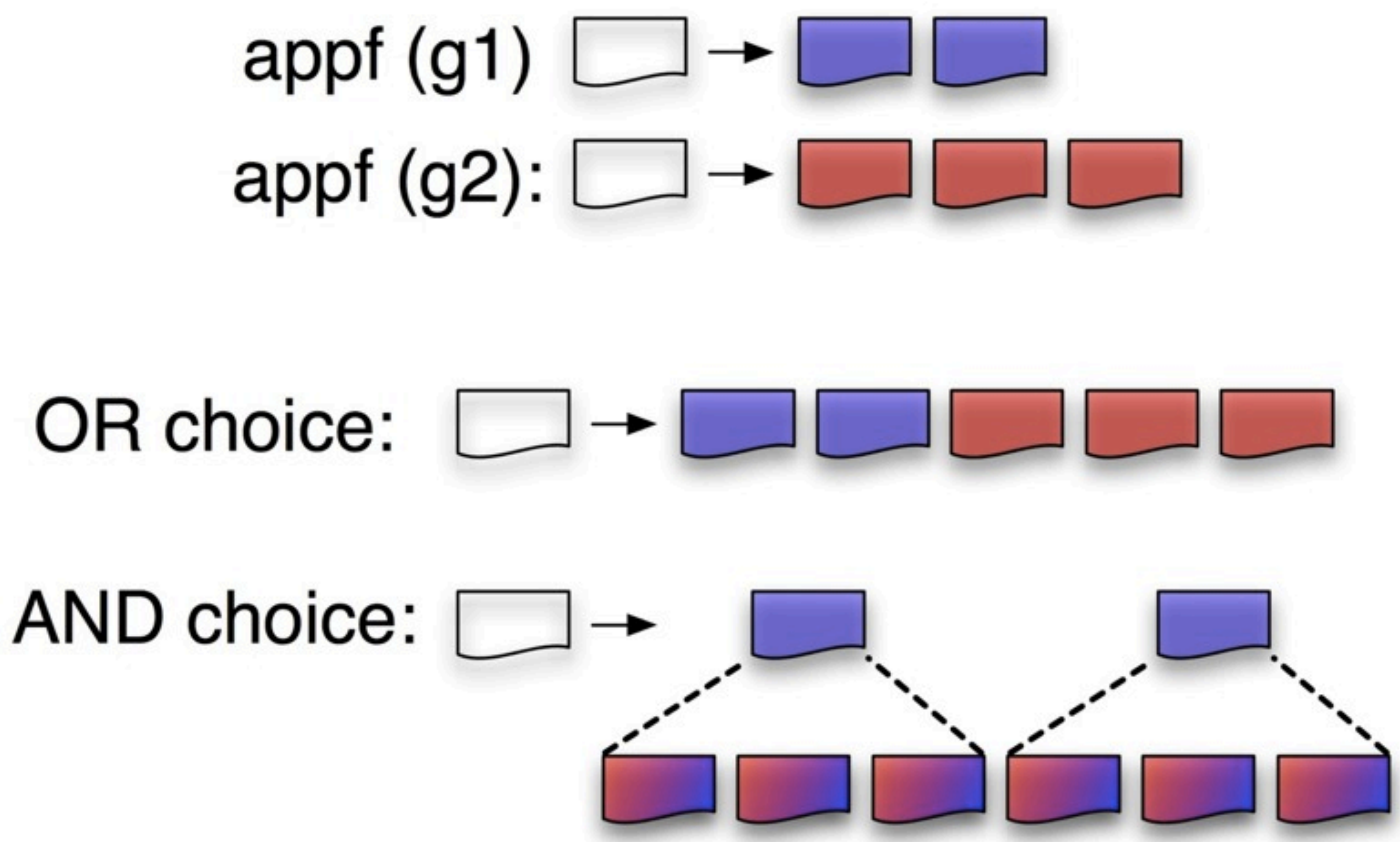


Example (BCK only)

induct then ((**base_wrap simp**)
tensor (**ripple** compose **fertilize**))

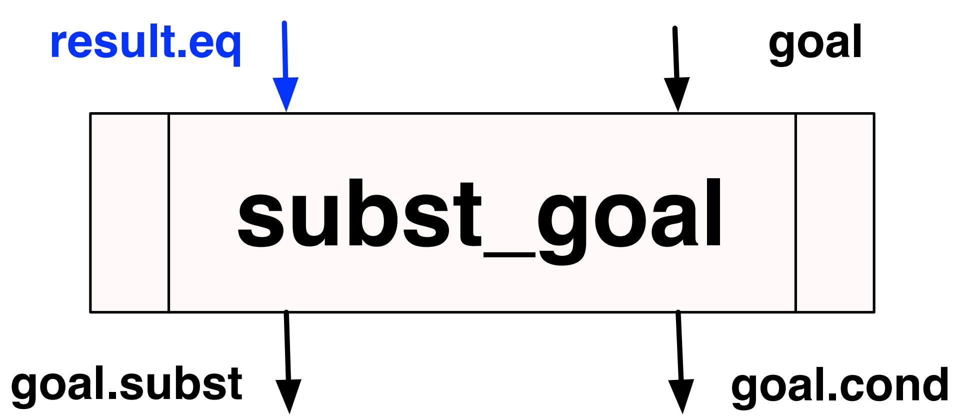


AND/OR wires



AND/OR wires

- No syntactic difference in in language between AND and OR
- In most cases wires are AND choices:
  - e.g. we simplify all base cases
- But, there are cases we want OR choices, e.g **substitution**:



- Problem: non-determinism - Example (with result.eq as AND)
  - **Input**: ({a=b,c=a, q => a=e},{P(c)})
  - **Output**: ({P(a)},{}) **or** ({P(b)},{}) **or** ({P(e)},{q}) **?**

Future work

- **Wires** - parameterize over them
    - more structure than names for better classification (reg-expr/1st order)
    - keep wire/type-checking decidable & static
- **Application function (appf)** - still a "black box" (cannot decompose)
    - loops: only low-level repetition
    - ```
      datatype appf = Comp of (rtechn * rtechn)
                    | Tensor of (rtechn * rtechn) ...
      ```
    - ```
      datatype appf = Nested of (rtechn HGraph)
                    | Atom of rst -> rst seq
      ```
- **Wire classification/learning** - sufficiently simple/abstract language to
    - recognise patterns where techniques succeeds/fails
    - automate classification & re-classify (specialise) goals/results
    - discover new combinations of techniques (or new techniques?) for given patterns