QuickChecking Confluence

Koen Claessen
Chalmers University of Technology
and Epic Games

Ulsan, 2026

ICFP 2023

The Verse Calculus: A Core Calculus for Deterministic Functional Logic Programming (Extended Version)

LENNART AUGUSTSSON, Epic Games, Sweden
JOACHIM BREITNER, Unaffiliated, Germany
KOEN CLAESSEN, Epic Games, Sweden
RANJIT JHALA, Epic Games, USA
SIMON PEYTON JONES, Epic Games, United Kingdom
OLIN SHIVERS, Epic Games, USA
GUY L. STEELE JR., Oracle Labs, USA
TIM SWEENEY, Epic Games, USA

203

Functional logic languages have a rich literature, but it is tricky to give them a satisfying semantics. In this paper we describe the Verse calculus, $\mathcal{VC}$, a new core calculus for deterministic functional logic programming. Our main contribution is to equip $\mathcal{VC}$ with a small-step rewrite semantics, so that we can reason about a
s one does with lambda calculus; that is, by applying successive rewrites to it.
ystem is confluent for well-behaved terms.
th appendices) of the paper in the Proceedings of the International Conference on

omputation → **Equational logic and rewriting**; *Proof theory*; **Rewrite**
xt-free languages; • **Software and its engineering** → **Syntax**; **Semantics**;
Functional languages; **Constraint and logic languages**; **Multiparadigm languages**.

Verse -
new programming
language for programming
the metaverse

```
e ::= v
    |  v=e
    |  v1(v2)
    |  e1;e2
    |  e1|e2
    |  fail
    |  one{e}
    |  all{e}
    |  ∃ x . e
```

```
v ::= x
    |  k
    |  <v1,..,vn>
    |  op
    |  \x. e
```

How to give semantics to this language?

small step operational semantics

big step operational semantics

rewrite semantics

denotational semantics

translational semantics

Application:

APP-ADD	$\mathbf{add}\langle k_1, k_2\rangle \longrightarrow k_3$	where $k_3 = k_1 + k_2$
APP-GT	$\mathbf{gt}\langle k_1, k_2\rangle \longrightarrow k_1$	if $k_1 > k_2$
APP-GT-FAIL	$\mathbf{gt}\langle k_1, k_2\rangle \longrightarrow \mathbf{fail}$	if $k_1 \leqslant k_2$
APP-LAM$^\alpha$	$(\lambda x.\, e)(v) \longrightarrow \exists x.\, x = v;\; e$	if $x \notin \text{fvs}(v)$
APP-TUP	$\langle v_1, \cdots, v_n\rangle(v) \longrightarrow (v = 1; v_1) \mid \cdots \mid (v = n; v_n)$	$n \geqslant 1$
APP-TUP-0	$\langle\rangle(v) \longrightarrow \mathbf{fail}$	

Unification:

U-LIT	$k = k \longrightarrow \langle\rangle$	
U-TUP-0	$\langle\rangle = \langle\rangle \longrightarrow \langle\rangle$	
U-TUP	$\langle v_1, \cdots, v_n\rangle = \langle v_1', \cdots, v_n'\rangle \longrightarrow v_1 = v_1'; \cdots; v_n = v_n'$	$n \geqslant 1$
U-FAIL	$hnf_1 = hnf_2 \longrightarrow \mathbf{fail}$	if U-LIT, U-TUP, U-OLAM do not match
U-OCCURS	$x = V[x] \longrightarrow \mathbf{fail}$	if $V \neq \Box$

Substitution:

SUBST-EXI	$S[x = v] \longrightarrow S\{v/x\}[x = v]$	$v \neq V[x]$

Normalization:

EXI-ELIM	$\exists x.\, e \longrightarrow e$	if $x \notin \text{fvs}(e)$
DEF-ELIM	$\exists x.\, E[x = v] \longrightarrow E[\langle\rangle]$	if $x \notin \text{fvs}(E) \cup \text{fvs}(v)$
EXI-FLOAT$^\alpha$	$C[\exists x.\, e] \longrightarrow \exists x.\, C[e]$	if $x \notin \text{fvs}(C) \cup \text{bvs}(C)$
SEQ-ASSOC	$(e_1; e_2);\; e_3 \longrightarrow e_1;\; (e_2;\; e_3)$	
SEQ-FLOAT	$v = (e_1;\; e_2) \longrightarrow e_1; v = e_2$	
SEQ-ELIM	$v;\; e \longrightarrow e$	
EQ-FLOAT	$v_1 = (v_2 = e) \longrightarrow v_2 = e;\; v_1 = \langle\rangle$	
EQ-SWAP	$v = x \longrightarrow x = v$	May apply infinitely for $x = y$
EQ-RESULT	$v = e;\; \langle\rangle \longrightarrow v = e$	

Choice:

CHOICE-ASSOC	$(e_1 \mid e_2) \mid e_3 \longrightarrow e_1 \mid (e_2 \mid e_3)$	
CHOICE-FAIL-L	$\mathbf{fail} \mid e \longrightarrow e$	
CHOICE-FAIL-R	$e \mid \mathbf{fail} \longrightarrow e$	
CHOICE	$C[e_1 \mid e_2] \longrightarrow C[e_1] \mid C[e_2]$	
CHOICE-FAIL	$C[\mathbf{fail}] \longrightarrow \mathbf{fail}$	

One and All:

ONE-FAIL	$\mathbf{one}\{\mathbf{fail}\} \longrightarrow \mathbf{fail}$	
ONE-VALUE	$\mathbf{one}\{v\} \longrightarrow v$	
ONE-CHOICE	$\mathbf{one}\{v \mid e\} \longrightarrow v$	
ALL-FAIL	$\mathbf{all}\{\mathbf{fail}\} \longrightarrow \langle\rangle$	
ALL-CHOICE	$\mathbf{all}\{v_1 \mid \cdots \mid v_n\} \longrightarrow \langle v_1, \cdots, v_n\rangle$	$n \geqslant 0$

```
SEQ-ASSOC (e1; e2); e3          ⟶ e1; (e2; e3)
SEQ-FLOAT v=(e1; e2)            ⟶ e1; v=e2
EQ-SWAP   v=x                   ⟶ x=v
```

```
APP-LAM  (\x.e)(v)              ⟶ ∃x. x=v; e       (x fresh)
```

```
UNI-TUP  <v1,..,vn>=<w1,..,wn>  ⟶ v1=w1;..;vn=wn
```

```
SUBST    S[x=v]                 ⟶ S{v/x}[x=v]      (x not in v)
```

```
ONE-FAIL   one{ fail }          ⟶ fail
ONE-VAL    one{ v }             ⟶ v
ONE-CHOICE one{ v | e }         ⟶ v
```

```
data Expr
  = Var Ident
  | Int Integer
  | Tuple [Expr]
  | Expr :=: Expr
  | Expr :>: Expr
  | Expr :|: Expr
  | …
```

Value and Expr the same type

run

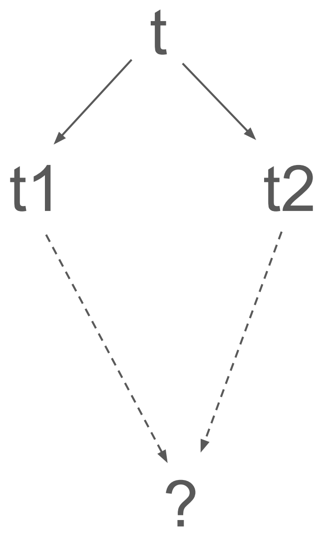
test

```
rules :: Rule Expr
rules =
  do Int i :=: Int j <- lhs
     guard (i==j)
     pure (Int i)
 <|>
  do Tuple vs :=: Tuple ws <- lhs
     pure (foldr (uncurry (:>:))
                 (Tuple vs)
                 (zip vs ws))
 <|>
  do Fail :>: e <- lhs
     pure Fail
 <|>
  do (e1 :>: e2) :>: e3 <- lhs
     pure (e1 :>: (e2 :>: e3))
 <|>
```

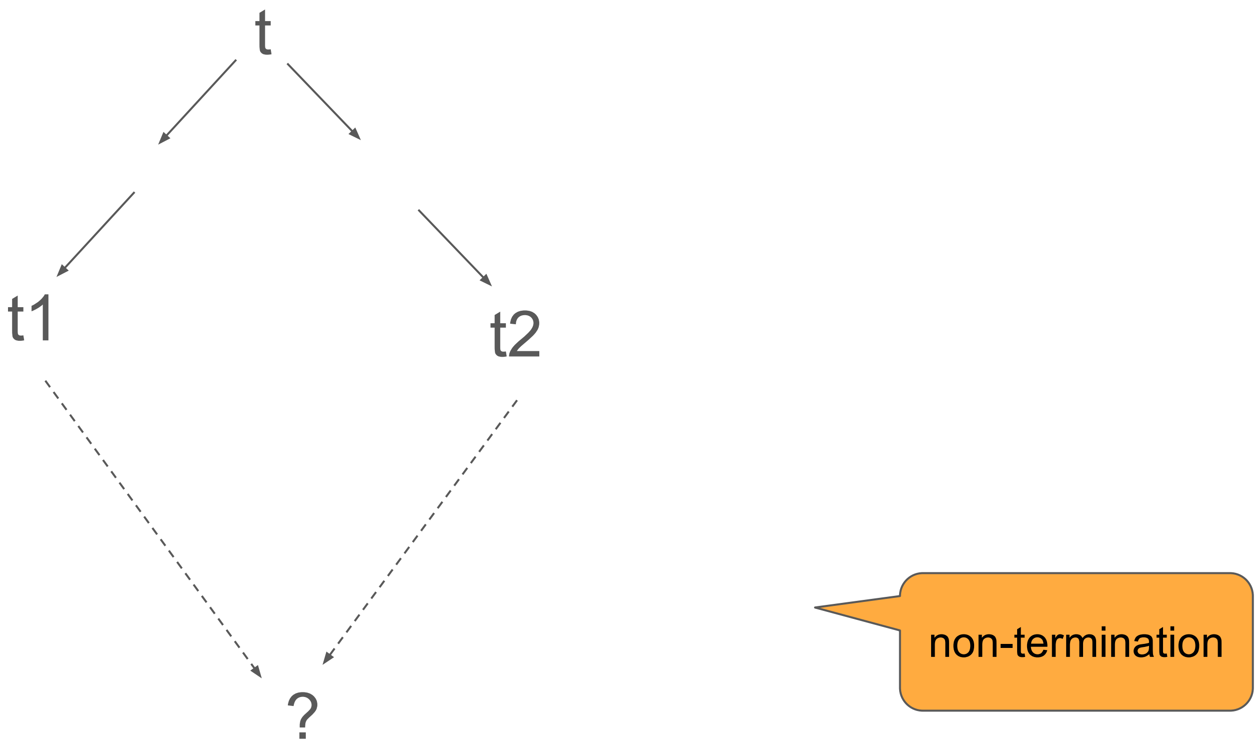
should look like "theory" as much as possible

```
exi x. 0(0); (x = (0(0) | fail))
  --CHOICE-->
exi x. 0(0); ((x = 0(0)) | (x = fail))
  --EXI-CHOICE-->
0(0); ((exi x. (x = 0(0))) | (ex x. (x = fail)))
  --FAIL-->
0(0); ((exi x. (x = 0(0))) | (ex x. fail))
  --EXI-ELIM-->
0(0); ((exi x. (x = 0(0))) | fail)
  --CHOICE-FAIL-R-->
0(0); (exi x. (x = 0(0)))
```

confluence



strong confluence



```
type Term = Expr
```

confluent?

one rewrite step

```
step :: Term -> [Term]
```

one computation step

assume terminating

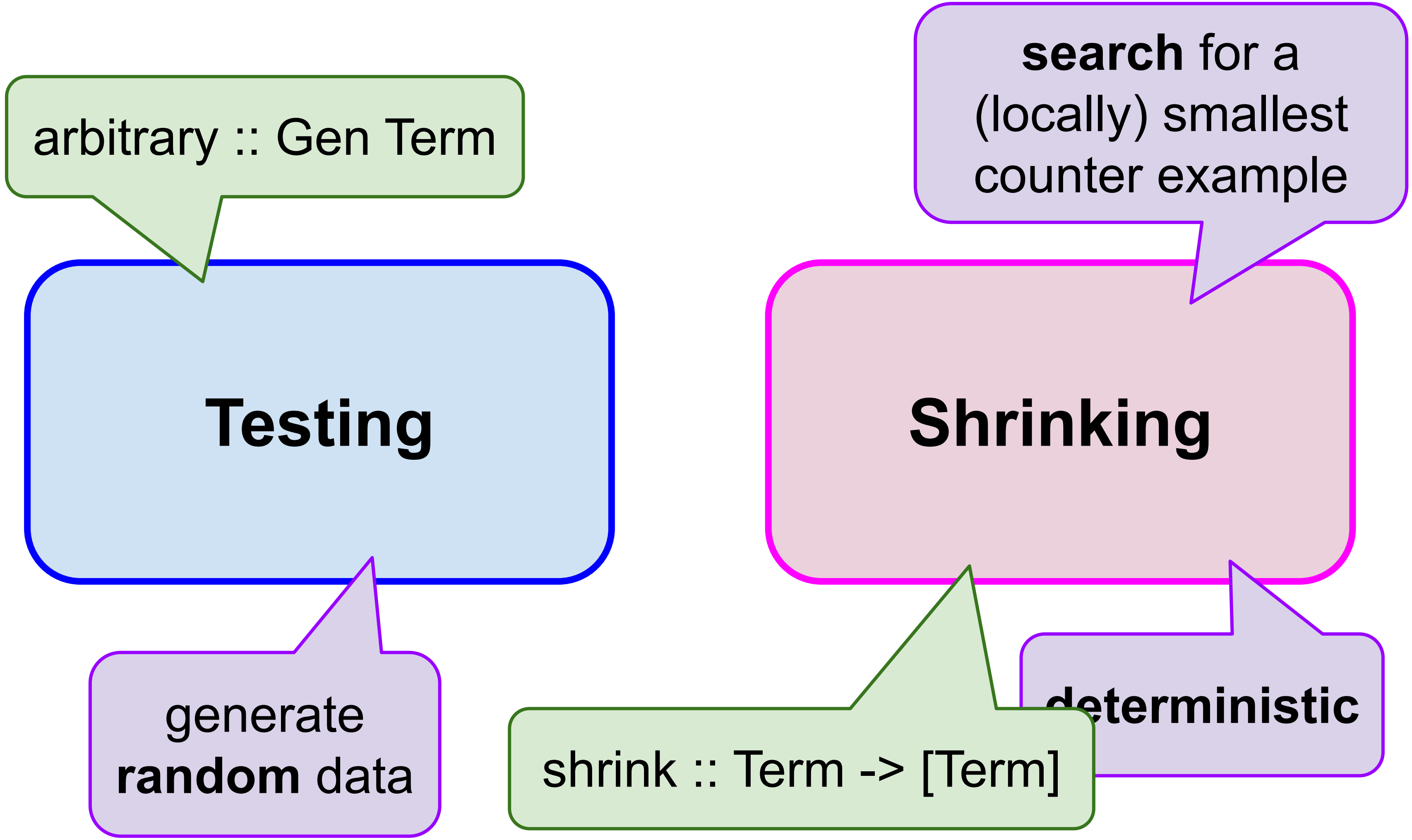
QuickCheck

generate **random** terms

property is **checked** for each term

```
prop_Confluence :: Term -> Property
prop_Confluence t = ...
```

counterexamples are reported


arbitrary :: Gen Term
search for a (locally) smallest counter example
Testing
Shrinking
generate random data
shrink :: Term -> [Term]
deterministic

Library
for writing 1-step shrinking functions

replace a part with an immediate sub-part

for free

custom rules

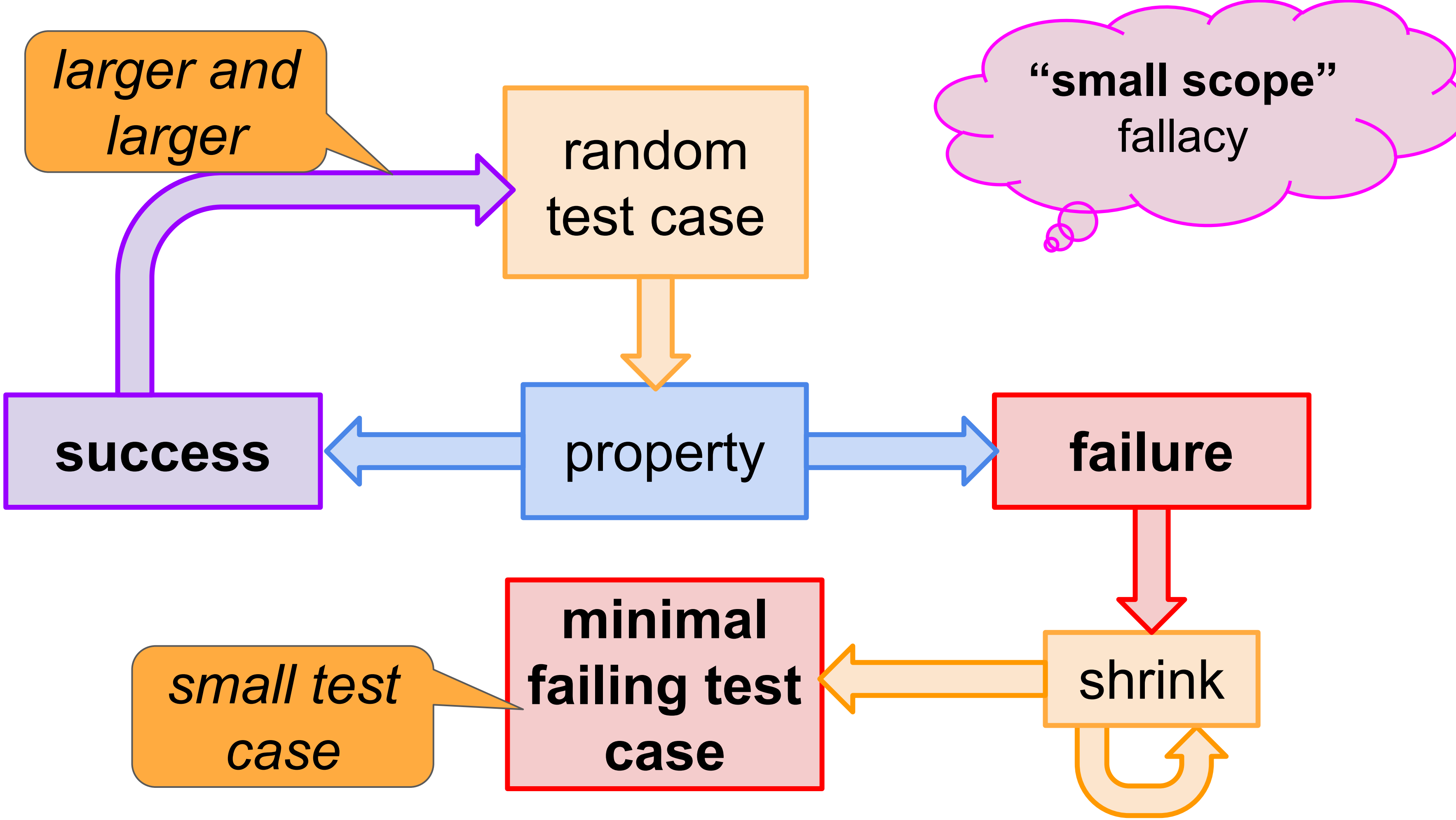
a + b ⟶ a, b

if e then p else q ⟶ p, q

C[var x in p] ⟶
var x in C[p]

while e do p ⟶
if e then p else skip

rules are applied repeatedly until a local minimum is found



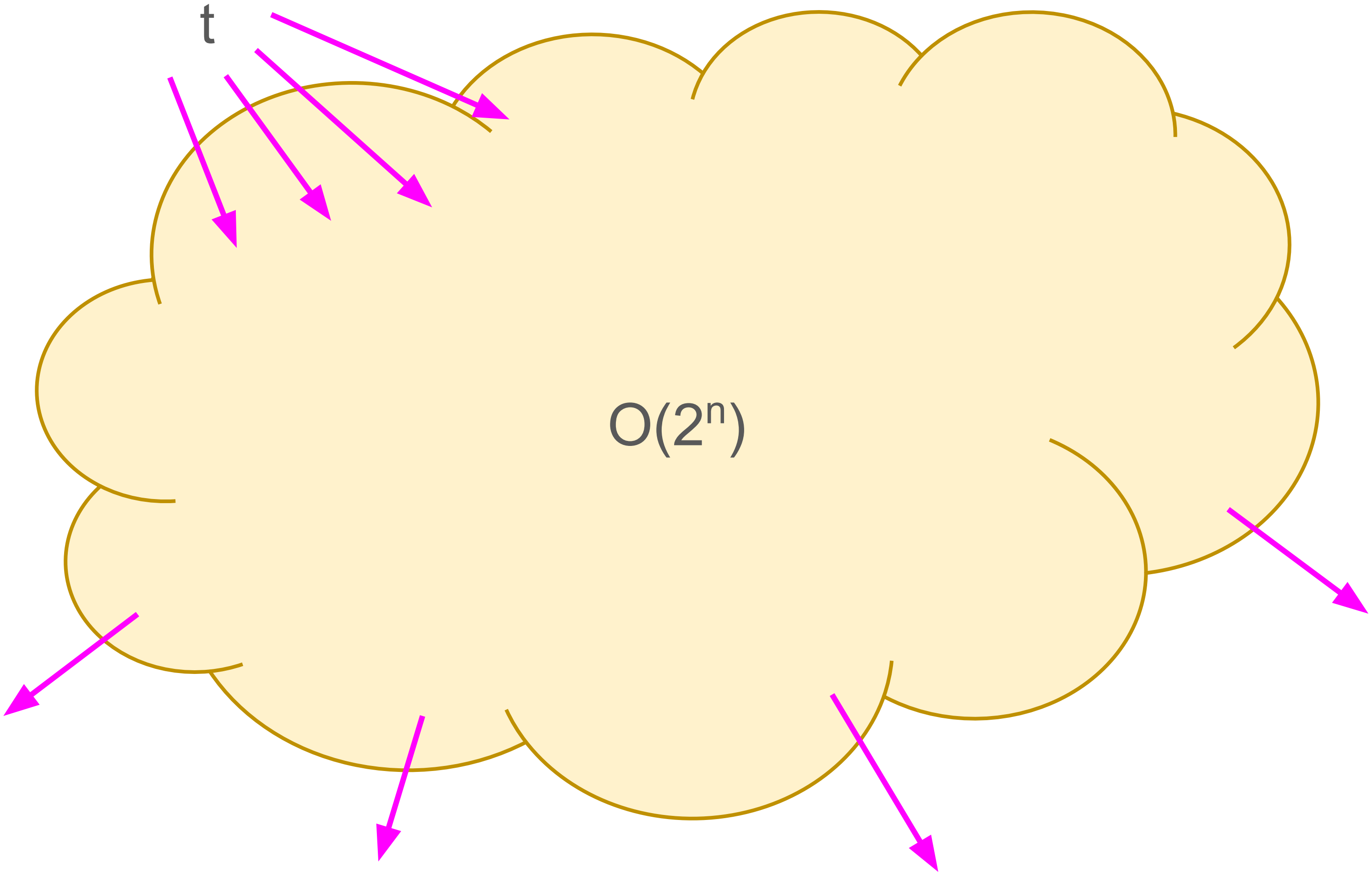
larger and larger
random test case
"small scope" fallacy
success
property
failure
small test case
minimal failing test case
shrink

compute all normal forms

```haskell
norms :: Term -> [Term]
norms t = go empty [t]
 where
  go seen []               = []
  go seen (t:ts)
    | t `member` seen = go seen ts
    | null ts'           = t : go seen ts
    | otherwise          = go (insert t seen)(ts'++ts)
   where
    ts' = step t
```

```haskell
prop_Confluence1 :: Term -> Bool
prop_Confluence1 t =
  case norms t of
    _t1 : _t2 : _ -> False
    _             -> True
```

very
expensive!


t
O(2^n)

compute arbitrary normal form

```haskell
arbNorm :: Term -> Gen Term
arbNorm t
  | null ts    = return t
  | otherwise = do t’ <- elements ts
                   arbNorm t’
 where
  ts = step t
```

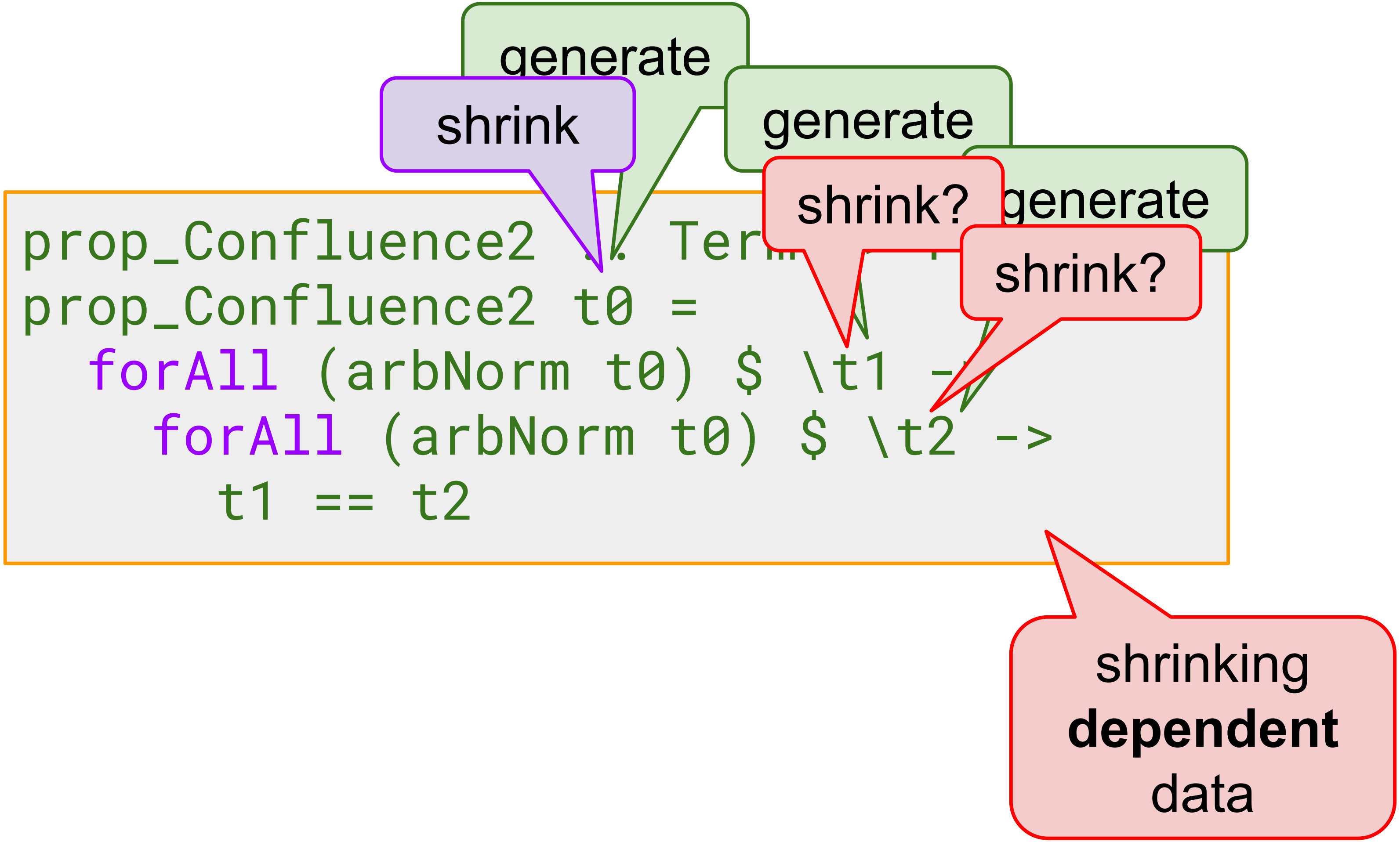
very cheap!

must run **more tests** to find bugs

```
prop_Confluence2 :: Term -> Property
prop_Confluence2 t0 =
  forAll (arbNorm t0) $ \t1 ->
    forAll (arbNorm t0) $ \t2 ->
      t1 == t2
```

bad shrinking

(no good feedback)


generate
shrink
generate
generate
shrink?
shrink?
prop_Confluence2 :: Ter
prop_Confluence2 t0 =
forAll (arbNorm t0) $ \t1 ->
forAll (arbNorm t0) $ \t2 ->
t1 == t2
shrinking **dependent** data

```haskell
data Fork = Fork Term Term Term
```

```haskell
arbFork :: Gen Fork
arbFork =
  do t0 <- arbTerm
     t1 <- arbNorm t0
     t2 <- arbNorm t0
     return (Fork t0 t1 t2)
```

```haskell
prop_Confluence3 :: Fork -> Bool
prop_Confluence3 (Fork _t0 t1 t2) =
  t1 == t2
```

```
Fork t0 t1 t2
```

```
Fork t0’ ??
```

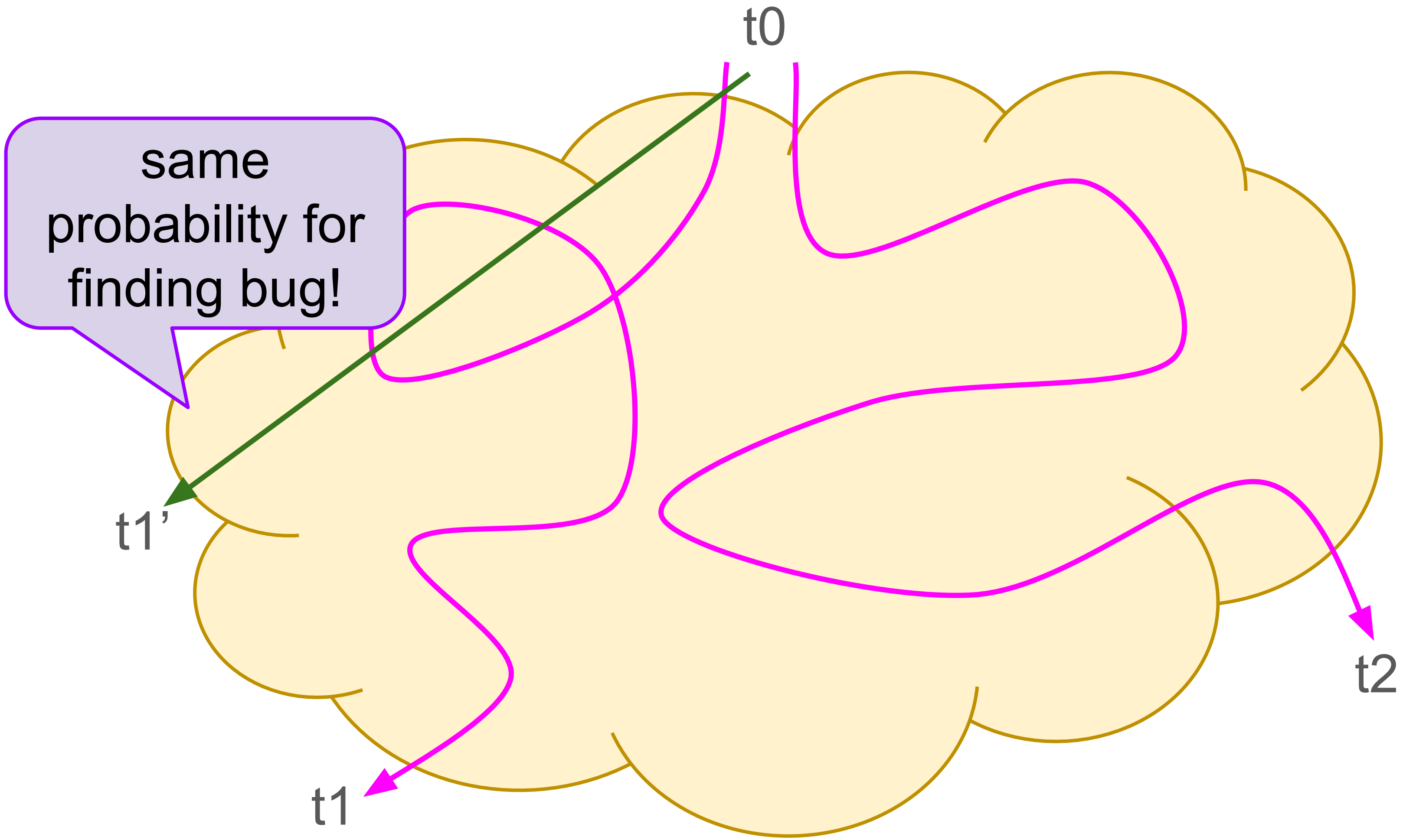
use (expensive) **norms**

Fork gives **fast testing**

```
instance Arbitrary Fork where
  ...

  shrink (Fork t0 _t1 _t2) =
    [ Fork t0’ t1’ t2’
    | t0’ <- shrink t0
    , t1’:t2’:_ <- [norms t0’]
    ]
```

but very very **slow shrinking**



t0
same probability for finding bug!
t1'
t1
t2

```
norm :: Term -> Term
norm t = case step t of
           []   -> t
           t':_ -> norm t'
```

always take leftmost step

```haskell
type Trace = [Term]
```

compute arbitrary *trace*

```haskell
arbTrace :: Term -> Gen Trace
arbTrace t
  | null ts    = return [t]
  | otherwise = do t' <- elements ts
                   (t:) `fmap` arbTrace t'
 where
  ts = step t
```

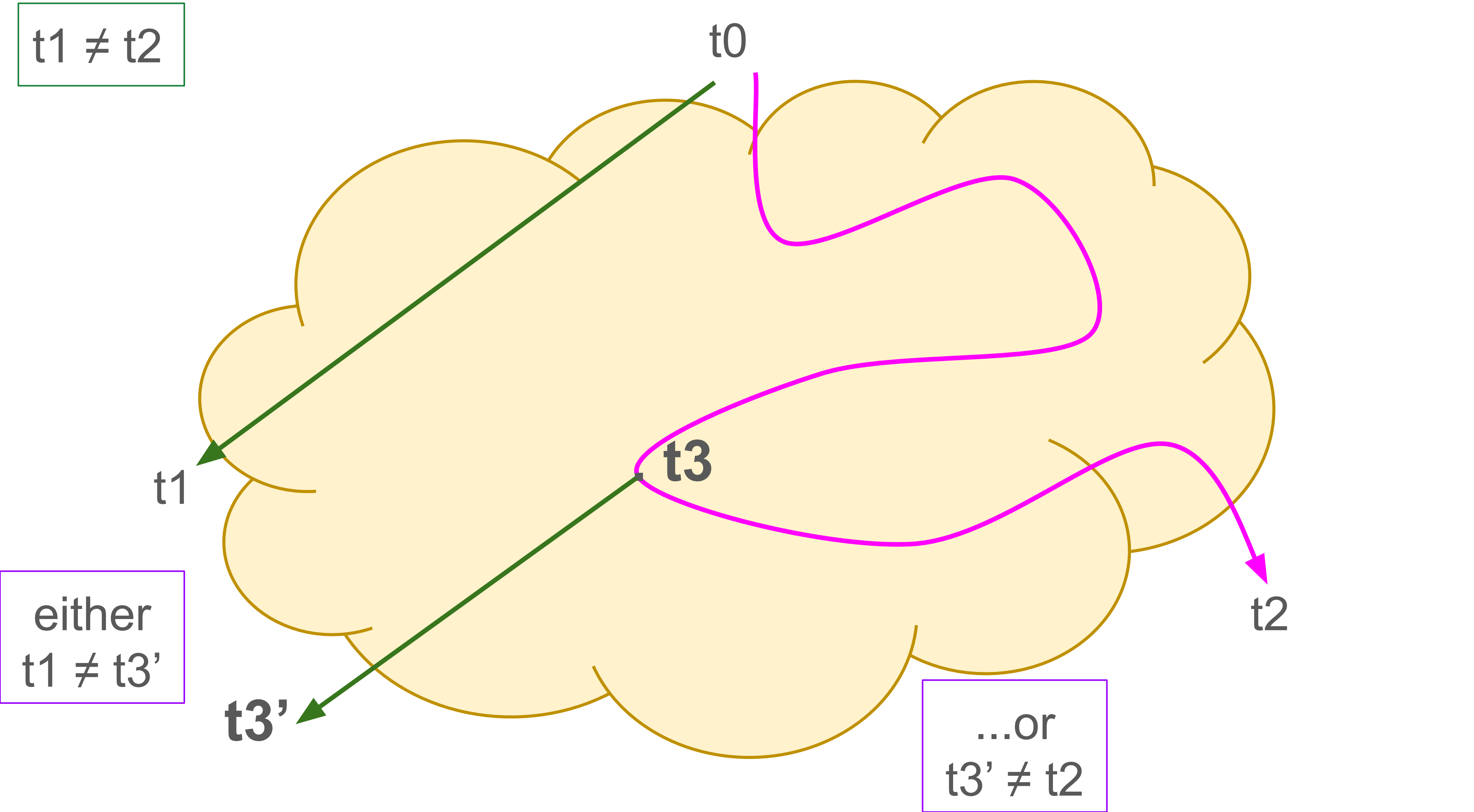
```haskell
type Trace = [Term]
```

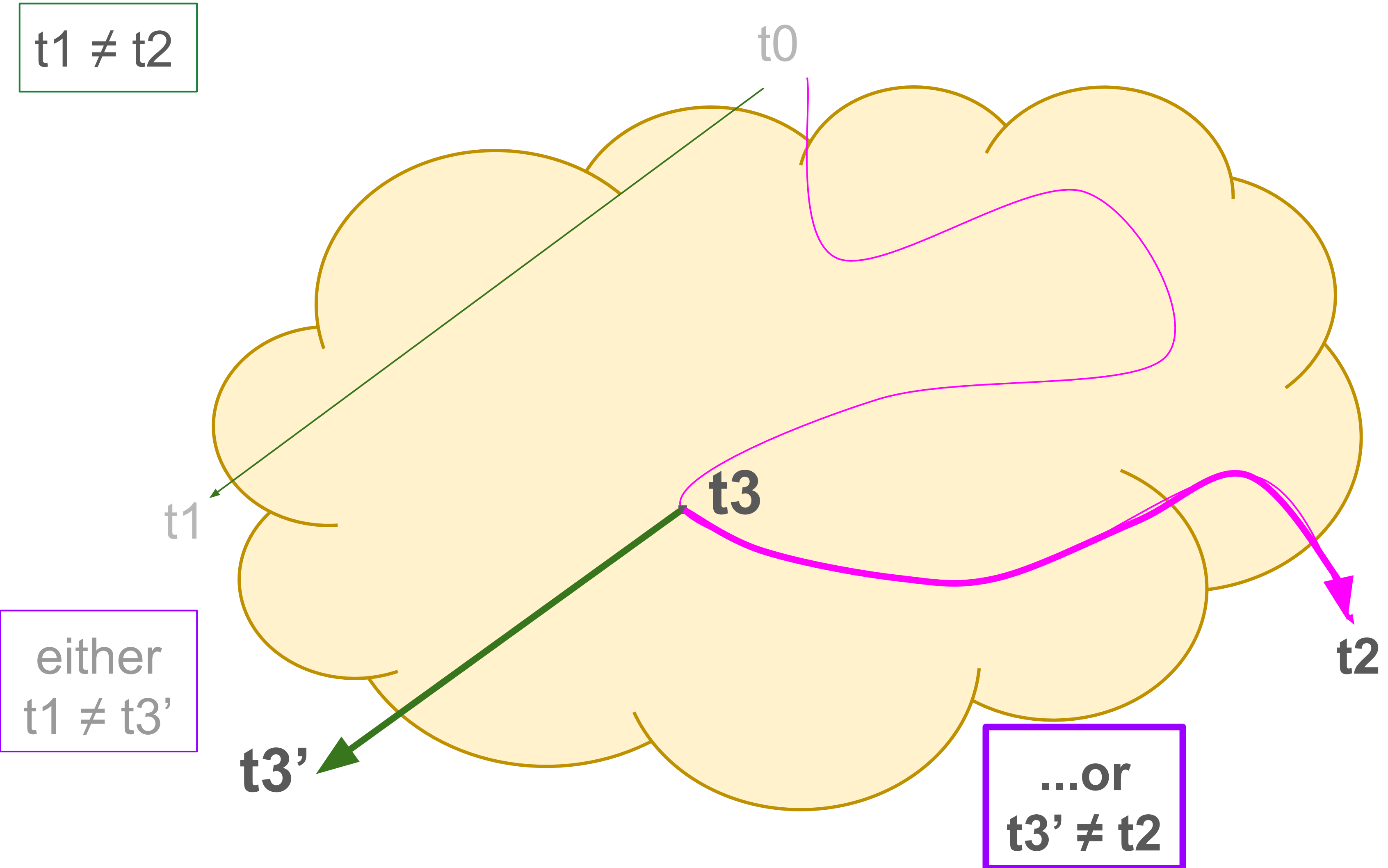
```haskell
data Fork = Fork Trace
```

```haskell
arbFork :: Gen Fork
arbFork =
  do t0 <- arbTerm
     tr <- arbTrace t0
     return (Fork tr)
```

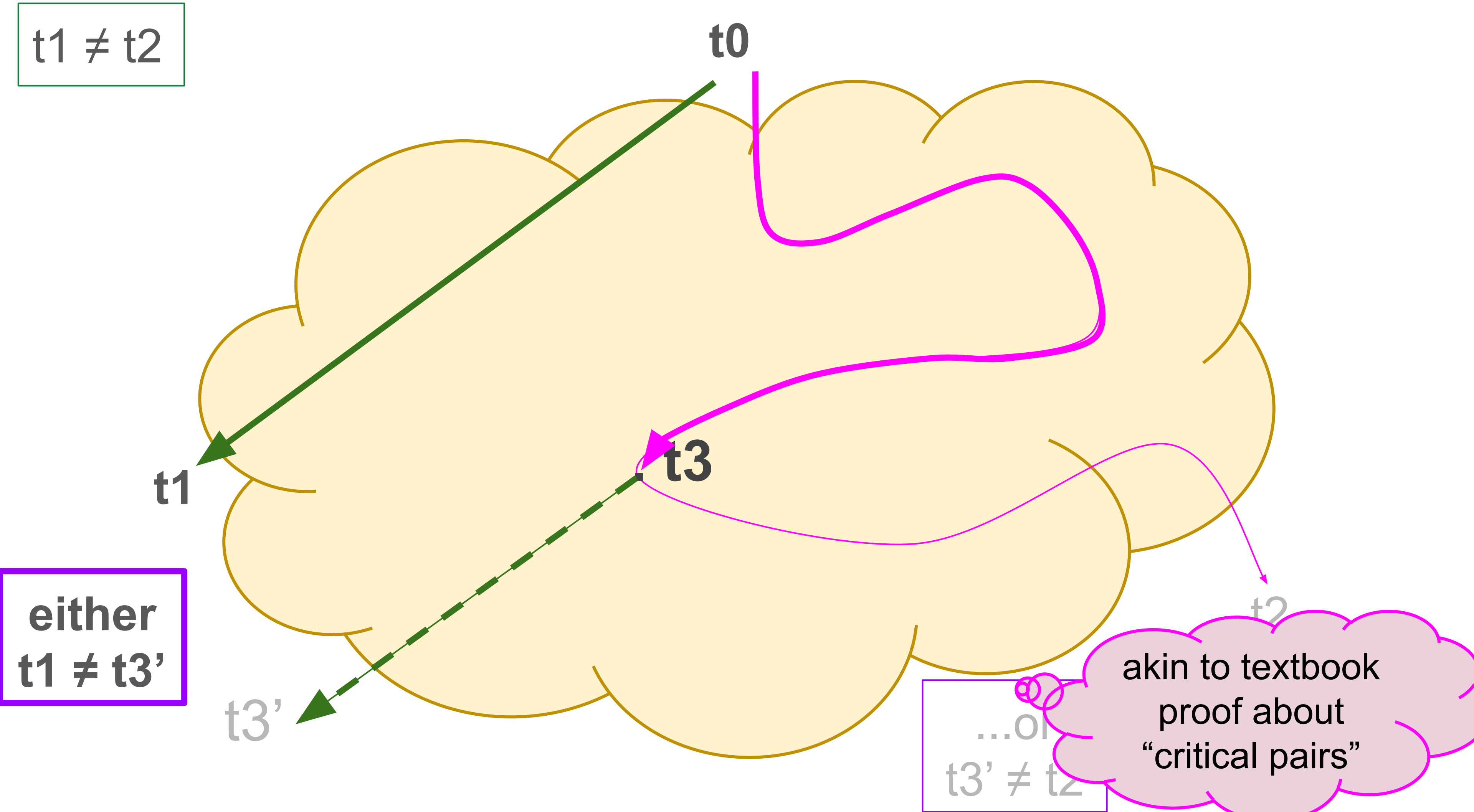
```haskell
prop_Confluence4 :: Fork -> Bool
prop_Confluence4 (Fork tr) =
  norm (head tr) == last tr
```

(not quite...)


t1 ≠ t2
t0
t1
t3
t2
either
t1 ≠ t3’
t3’
...or
t3’ ≠ t2



t1 ≠ t2
t0
t1
t3
t2
either
t1 ≠ t3’
t3’
...or
t3’ ≠ t2


t1 ≠ t2
t0
t1
t3
either
t1 ≠ t3’
t3’
t2
...or
t3’ ≠ t2
akin to textbook
proof about
“critical pairs”

```haskell
type Trace = [Term]
```

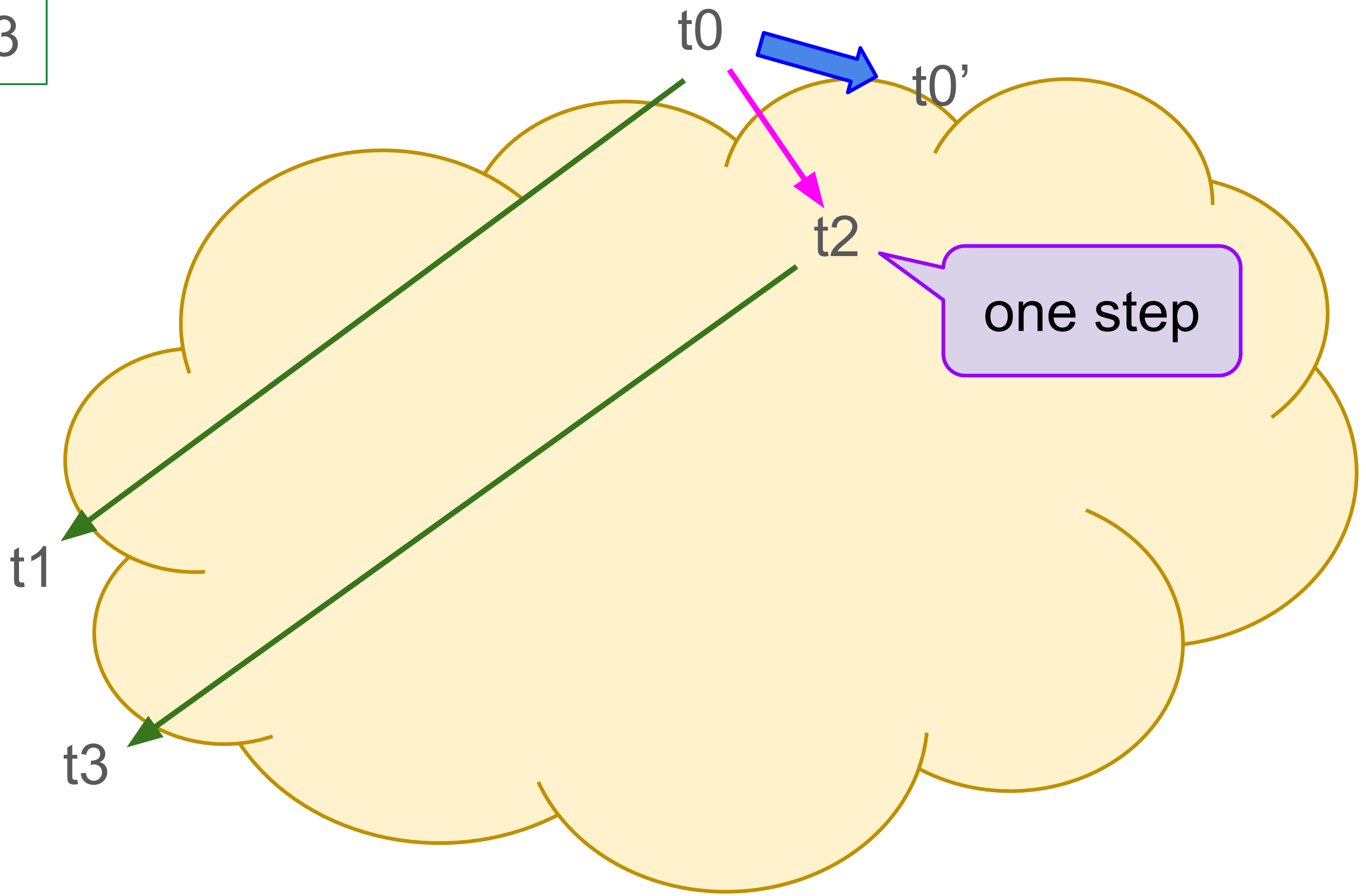
```haskell
data Fork = Fork Trace
```

```haskell
arbFork :: Gen Fork
arbFork =
  do t0 <- arbTerm
     tr <- arbTrace t0
     return (Fork tr)
```

```haskell
prop_Confluence4 :: Fork -> Bool
prop_Confluence4 (Fork tr) =
  norm (head tr) == norm (last tr)
```

(quite...)

$t1 \neq t3$



```haskell
instance Arbitrary Fork where
  ...
  shrink (Fork [t0,_t2]) =
    [ Fork [t0',t2']
    | t0' <- shrink t0
    , t2' <- step t0'
    ]

  shrink (Fork tr) =
    [ Fork (take (k+1) tr)
    , Fork (drop k tr)
    ]
   where
    k = length tr `div` 2
```

shrinking t0

shrinking the trace

use **specialized** property for bug shrinking

```
prop_Confluence5 :: Term -> Bool
prop_Confluence5 t =
  all (\t’ -> norm t == norm t’) (step t)
```

use **general** property for bug finding

has a different (worse) distribution!

only checks top-level step

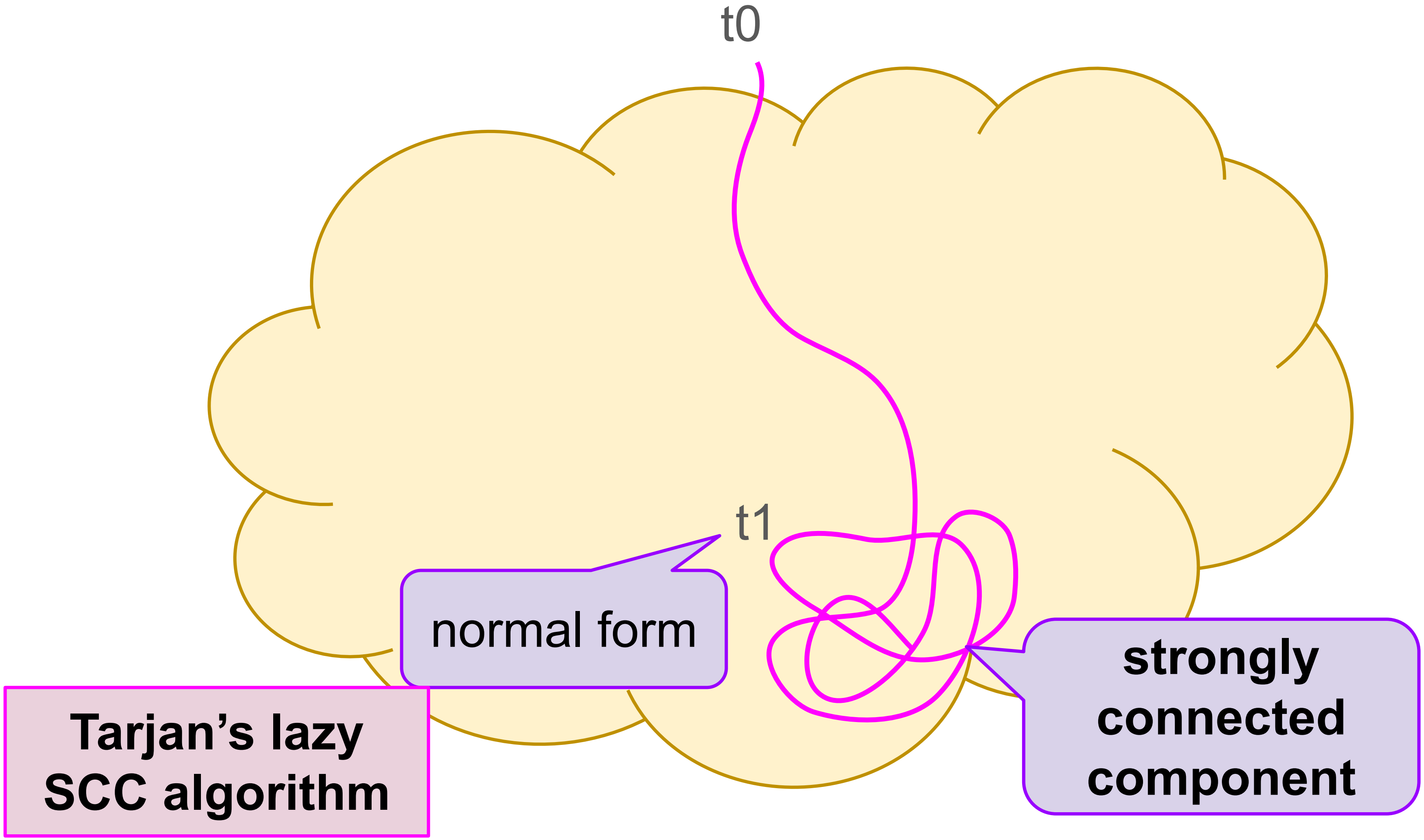
```
(VAR-SWAP) x = y ⟶ y = x
```

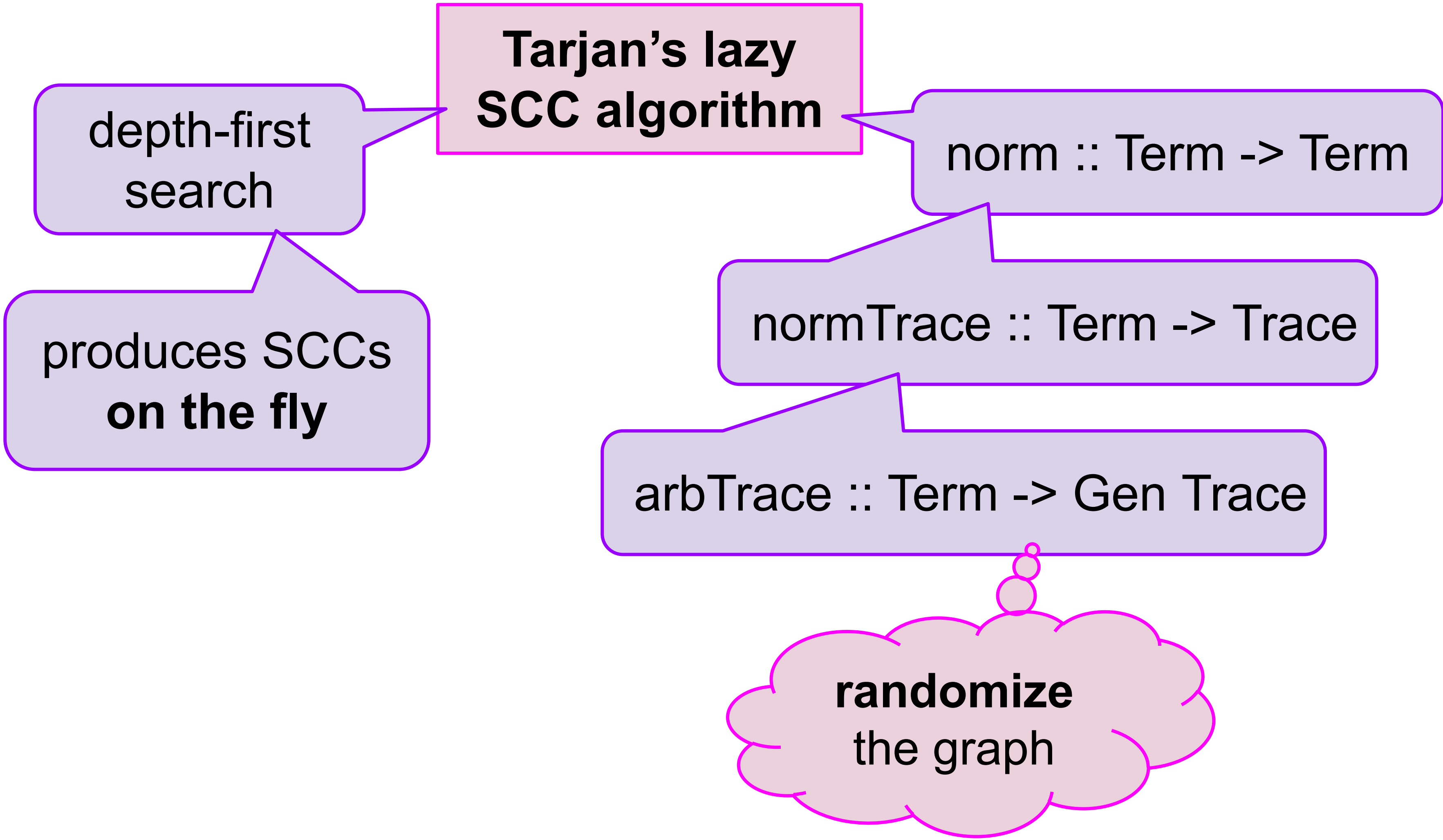
```
(EXI-SWAP) ∃x.(∃y.e) ⟶
∃y.(∃x.e)
```

don’t terminate!

“structural rules”

but they are **looping**


t0
t1
normal form
strongly connected component
Tarjan's lazy SCC algorithm


Tarjan's lazy SCC algorithm
depth-first search
produces SCCs on the fly
norm :: Term -> Term
normTrace :: Term -> Trace
arbTrace :: Term -> Gen Trace
randomize the graph

use **specialized** property for bug shrinking

```
prop_Confluence5 :: Term -> Bool
prop_Confluence5 t =
  all (\t' -> norm t == norm t') (step t)
```

use **general** property for bug finding

has a different (worse) distribution!

only checks top-level step

Summary

- Checking confluence:
  - Using random terms
  - Computing all normal forms: very slow
  - Left-most normal form (deterministic) == random normal form: very quick

- Finding small counterexamples:
  - Avoid data-dependency in quantifiers
  - Forall t . if t ⟶ t’ then norm(t)==norm(t’)
  - Shrink traces to get to the above property