#28: Synthesizing Effectful Programs

Sankha Narayan Guria

EECS 700: Introduction to Program Synthesis

Effects

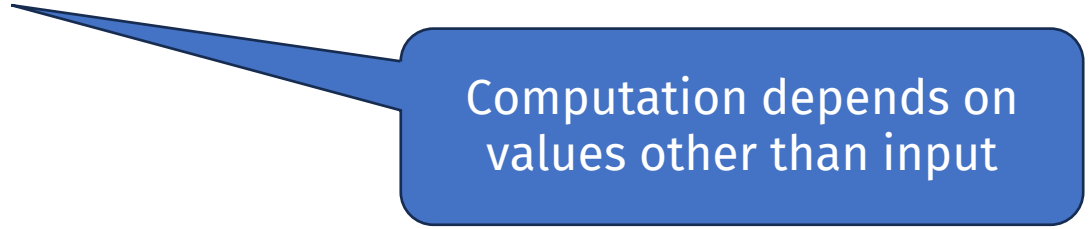
- Affects a program but does not show in input arguments or return values
- Global memory, exceptions, I/O, etc.

```
def divide(x, y)
  if y != 0
    x / y
  else
    raise RuntimeError, "y is 0"
  end
end
```

$$divide(x, y) = \frac{x}{y} \text{ if } y \neq 0$$

The problem with effects

```ruby
def secret_add(x)
  not_so_secret = File.read('secret.txt').to_i
  not_so_secret + x
end
```




```ruby
secret_add(5) # => 10
secret_add(5) # => 16
```

Effect Analysis

```
def secret_add(x)
  if x > 10
    not_so_secret = File.read('secret.txt').to_i
    not_so_secret + x
  else
    x + 42
  end
end
```

Int {}
Bool {}
String io
Int io
Int io
Int {}
Int io

Problem

- Goal: Synthesize Ruby programs, a popular language to write web apps
- Formal specs are difficult for Ruby programs
- Programs have side effects
- Tests and types are common for checking correctness

Our Insights

- Tests often fail because of side effects
- Infer failure inducing side effect from test execution
- Use type and effect information to build candidate expressions
- Use tests to verify correctness

Example: Blogging web app

- Method to update a blog post entry
- Method arguments: author, post's slug (a unique identifier), and fields to be updated
- Returns the updated post
- Type signature:

```
update_post :: (Str, Str, {author: ?Str, title: ?Str, slug: ?Str}) → Post
```

Test #1

If the author updates the post, update the title only

```
post = Post.create(author: 'author', slug:'hello-world', title: 'Hello World')
updated = update_post('author', 'hello-world', author: 'dummy',
                title: 'Foo Bar', slug: 'foobar')

assert { updated.id == post.id }          # the returned post should be same
assert { updated.author == "author" }     # the author field should not be updated
assert { updated.title == "Foo Bar" }     # the title should be updated
assert { updated.slug == 'hello-world' }
```

Database access methods

Classes and methods used in tests refer to tables and the methods used to access columns

Tables

User

```
username
name
```

Post

```
author
title
slug
```

Accessor Methods

Post class:

```
title  :: '() → Str',    read:  ['Post.title']
title= :: '(Str) → Str', write: ['Post.title']
```

User class:

```
username  :: '() → Str',    read:  ['User.username']
username= :: '(Str) → Str', write: ['User.username']
```

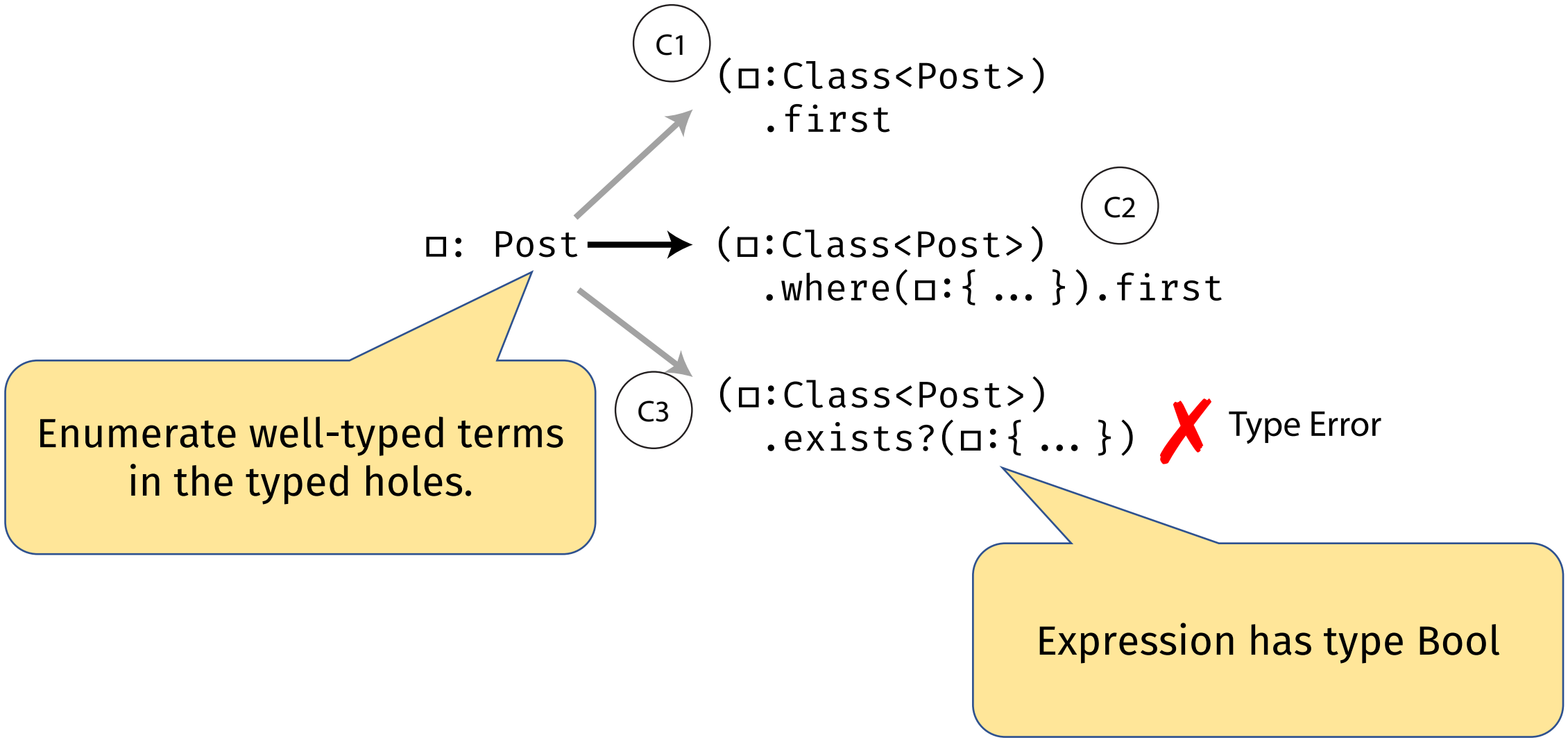
Initializing Synthesis

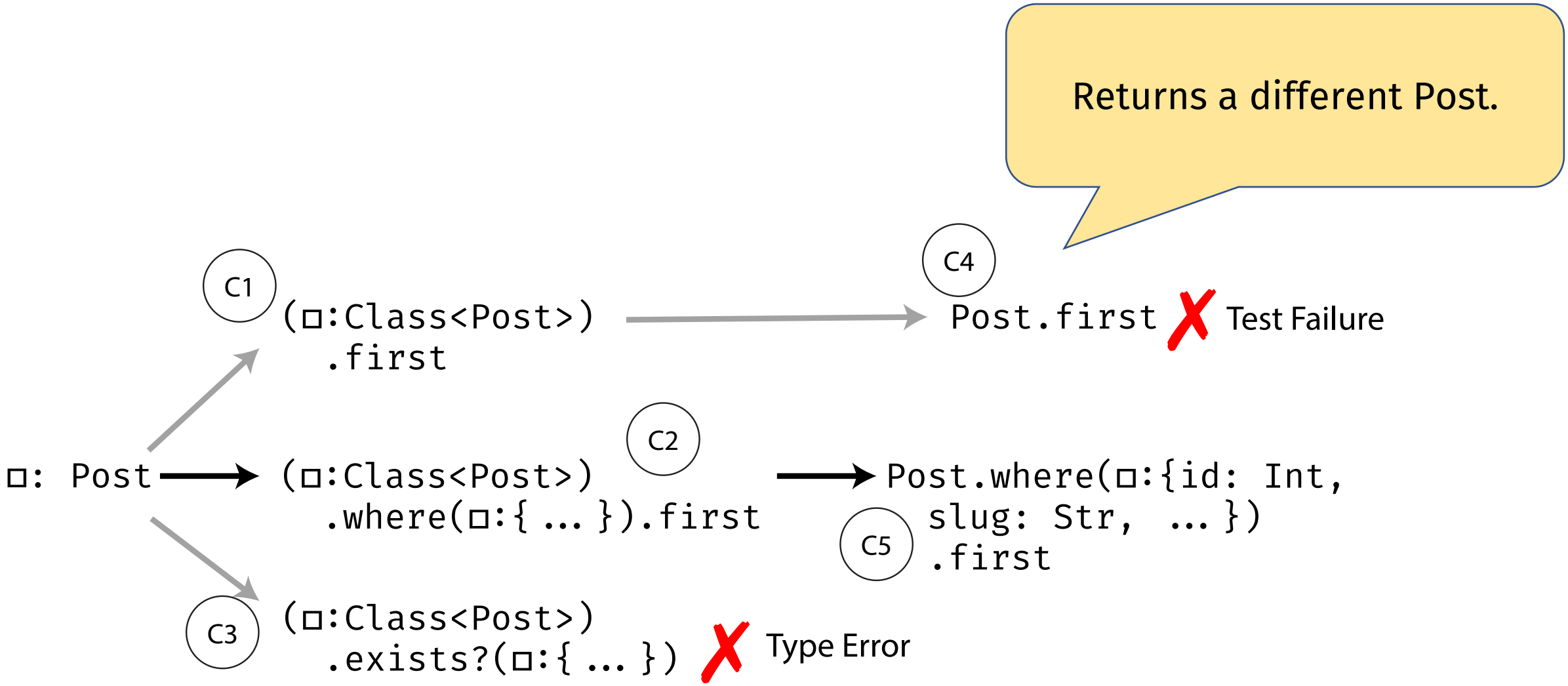
Type signature gives the template to start the candidate program search

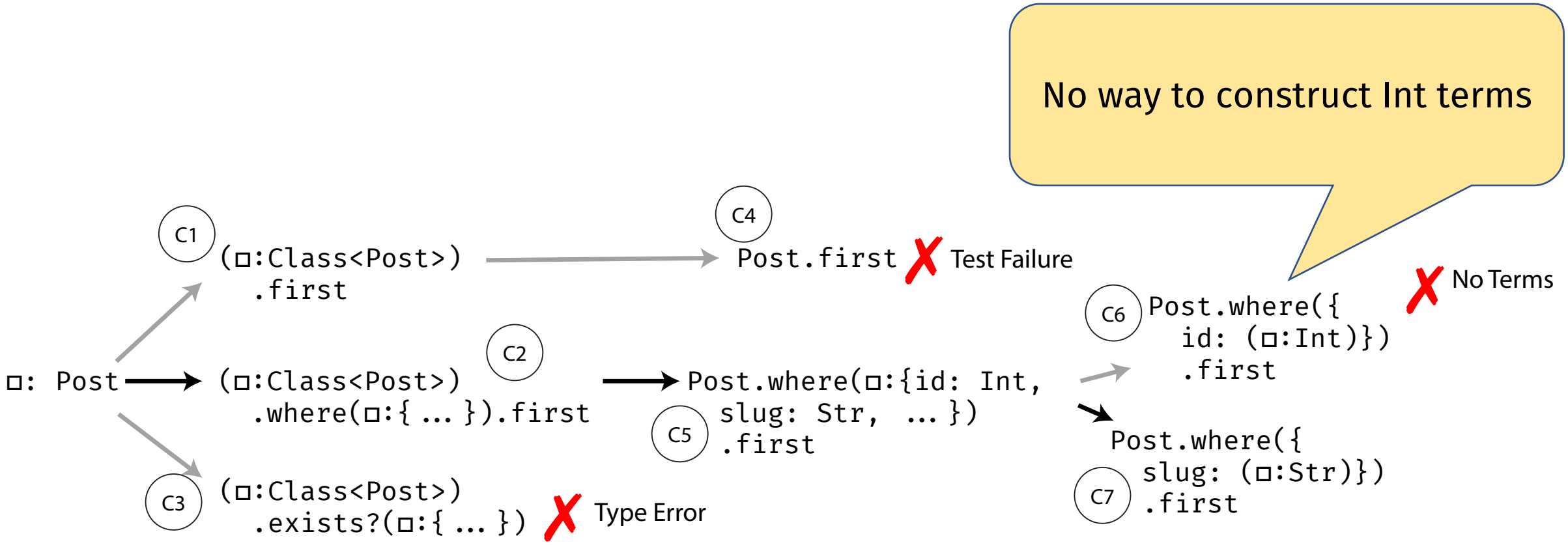
```
update_post :: (Str, Str, {author: ?Str, title: ?Str, slug: ?Str}) → Post
```

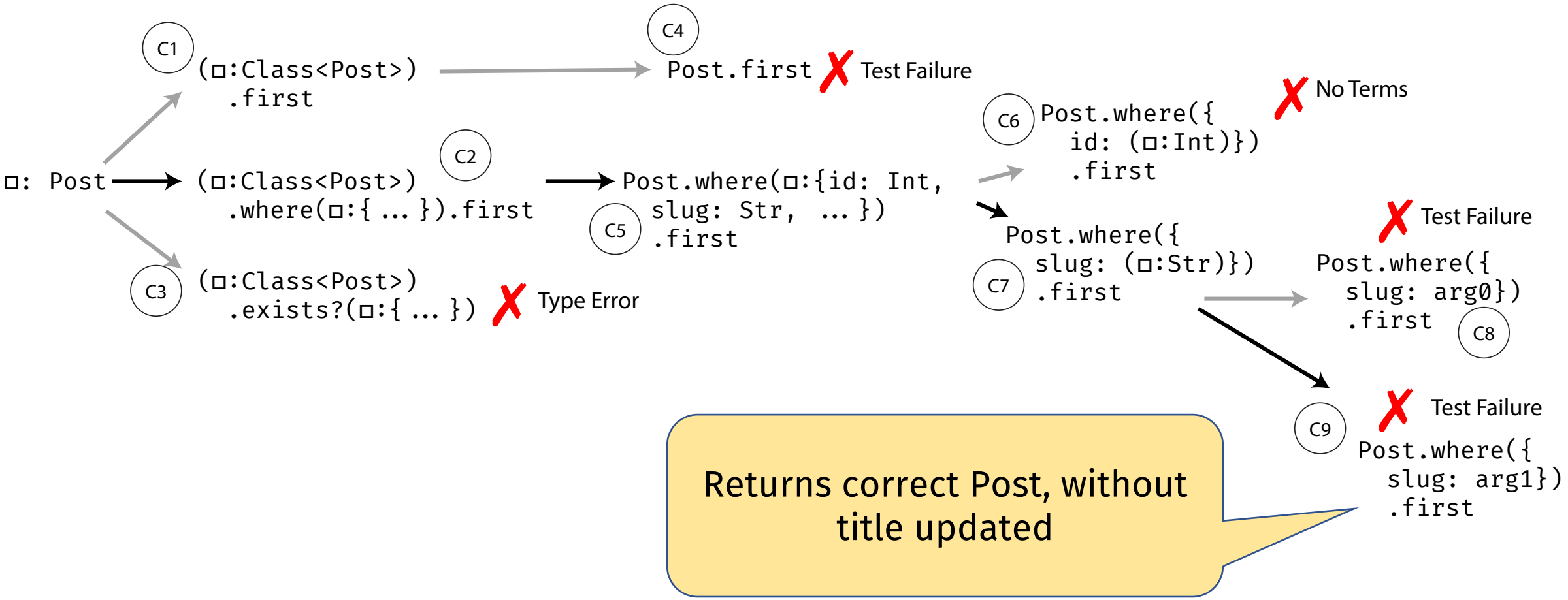
Synthesis Template

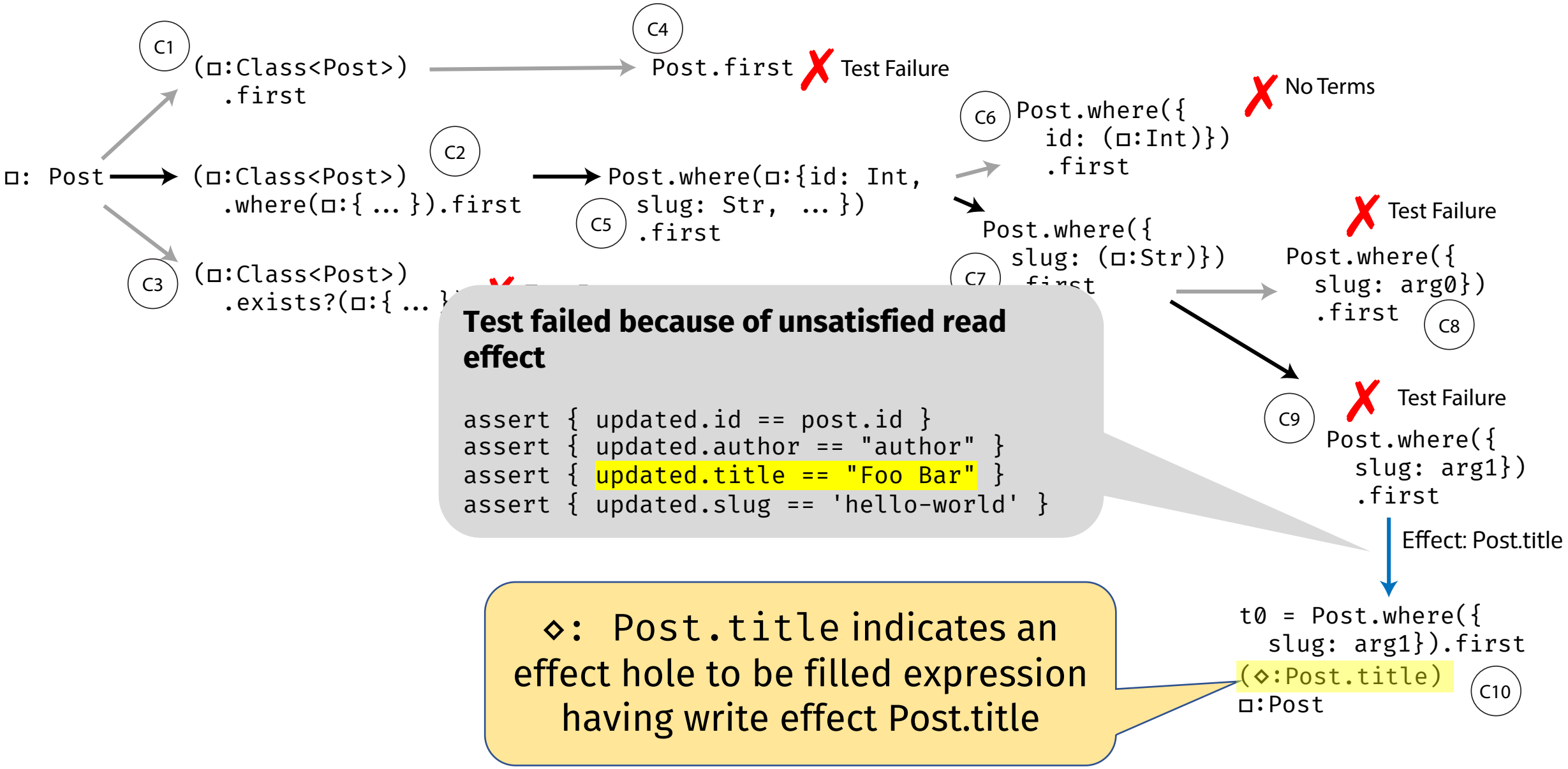
```
□: Post # □ is a placeholder for
        # expressions of type Post
```

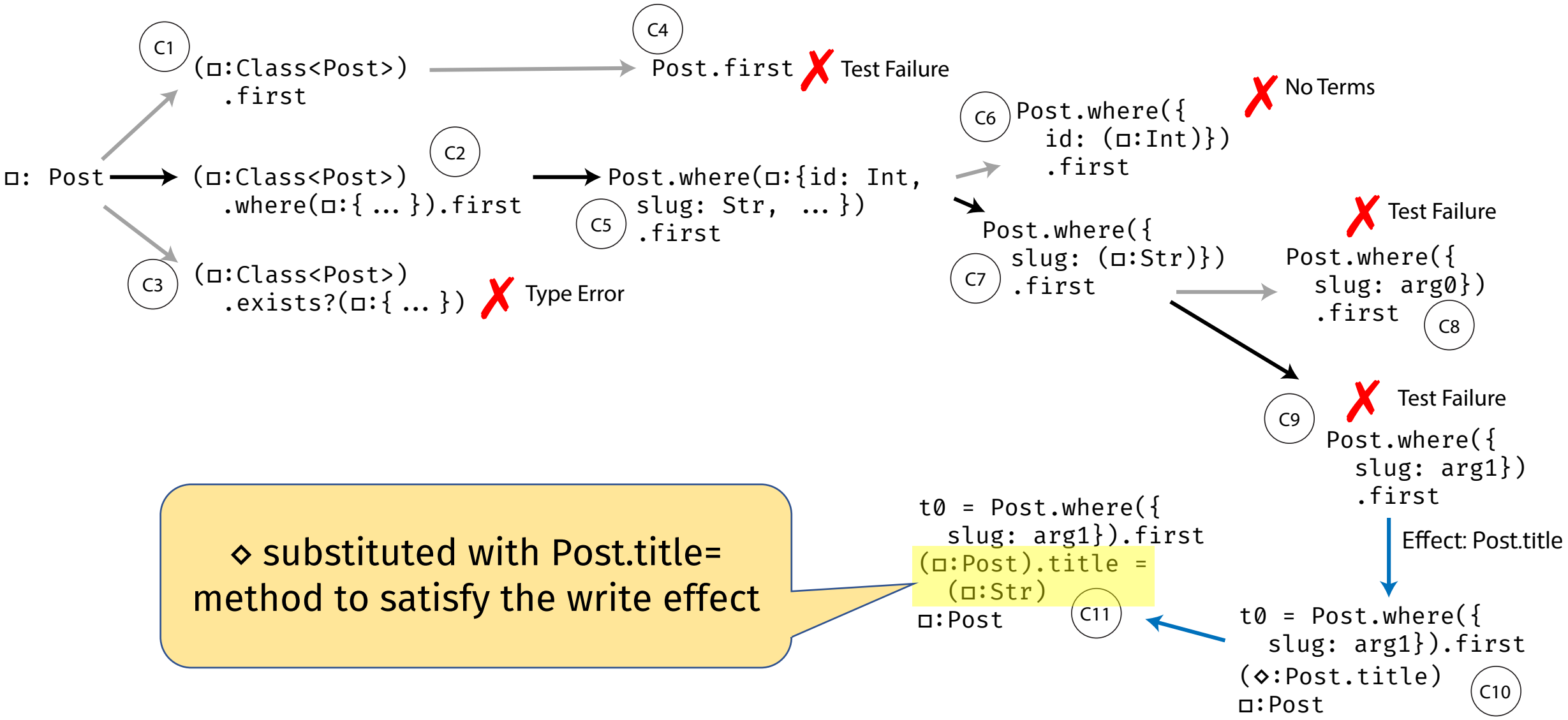

C1
(□:Class<Post>)
.first
□: Post
C2
(□:Class<Post>)
.where(□:{ ... }).first
C3
(□:Class<Post>)
.exists?(□:{ ... })
Type Error
Enumerate well-typed terms in the typed holes.
Expression has type Bool

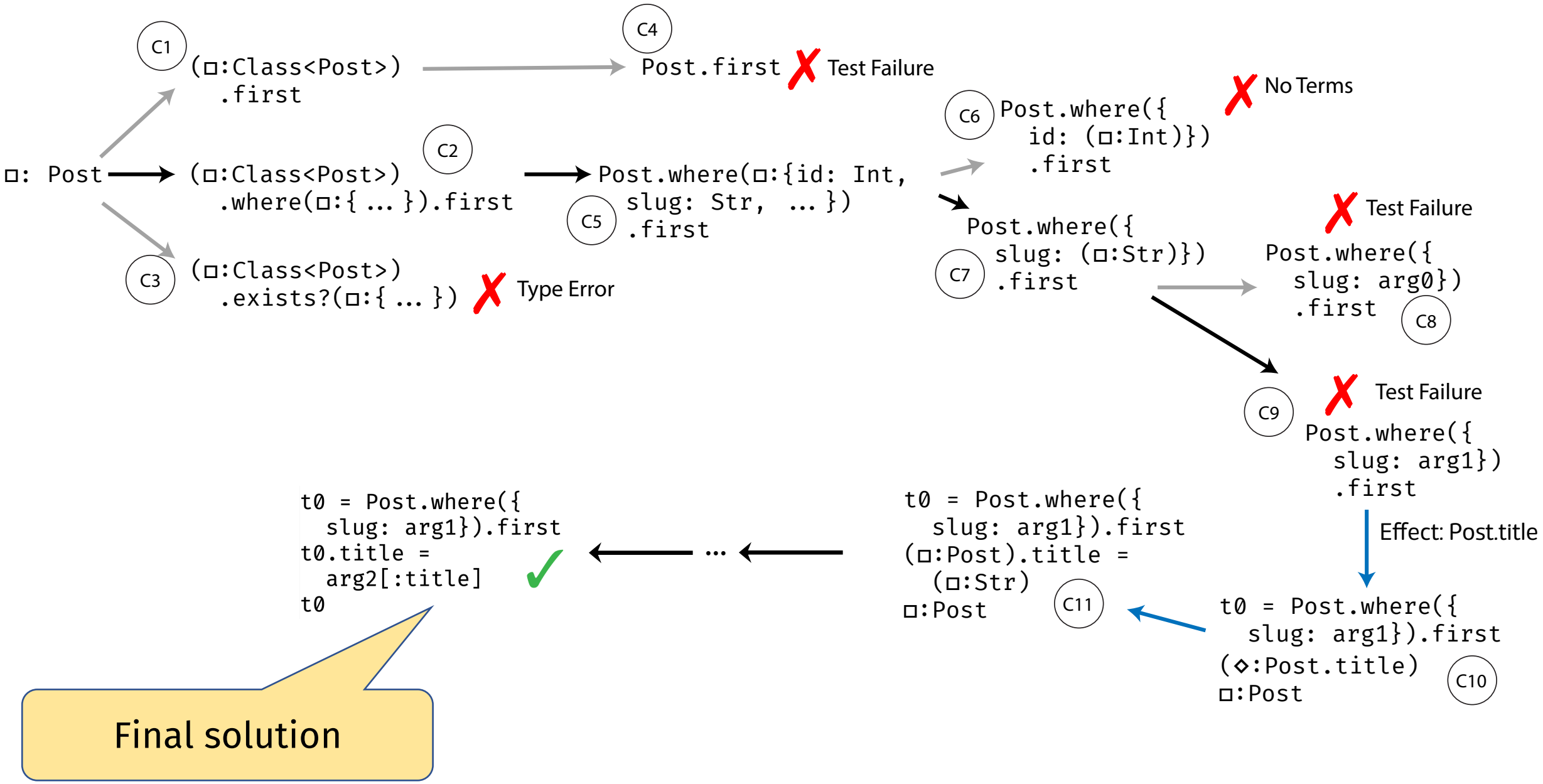

Returns a different Post.
C1
(□:Class<Post>)
.first
C4
Post.first
Test Failure
□: Post
C2
(□:Class<Post>)
.where(□:{ ... }).first
Post.where(□:{id: Int,
slug: Str, ... })
.first
C5
C3
(□:Class<Post>)
.exists?(□:{ ... })
Type Error


No way to construct Int terms
C1
(□:Class<Post>)
.first
C4
Post.first
Test Failure
C6
Post.where({
id: (□:Int)})
.first
No Terms
□: Post
C2
(□:Class<Post>)
.where(□:{ ... }).first
Post.where(□:{id: Int,
slug: Str, ...})
C5
.first
Post.where({
slug: (□:Str)})
C7
.first
C3
(□:Class<Post>)
.exists?(□:{ ... })
Type Error


C1
(□:Class<Post>)
.first
C4
Post.first
Test Failure
C6
Post.where({
id: (□:Int)})
.first
No Terms
□: Post
C2
(□:Class<Post>)
.where(□:{ ... }).first
Post.where(□:{id: Int,
slug: Str, ... })
C5
.first
Post.where({
slug: (□:Str)})
C7
.first
Test Failure
Post.where({
slug: arg0})
.first
C8
C3
(□:Class<Post>)
.exists?(□:{ ... })
Type Error
C9
Test Failure
Post.where({
slug: arg1})
.first
Returns correct Post, without title updated


C1
(□:Class<Post>)
.first
C4
Post.first
Test Failure
C6
Post.where({
id: (□:Int)})
.first
No Terms
□: Post
C2
(□:Class<Post>)
.where(□:{ ... }).first
Post.where(□:{id: Int,
slug: Str, ... })
C5
.first
C3
(□:Class<Post>)
.exists?(□:{ ...
Post.where({
slug: (□:Str)})
C7
Test Failure
Post.where({
slug: arg0})
.first
C8
Test Failure
C9
Post.where({
slug: arg1})
.first
Effect: Post.title
Test failed because of unsatisfied read effect
assert { updated.id == post.id }
assert { updated.author == "author" }
assert { updated.title == "Foo Bar" }
assert { updated.slug == 'hello-world' }
◇: Post.title indicates an effect hole to be filled expression having write effect Post.title
t0 = Post.where({
slug: arg1}).first
(◇:Post.title)
□:Post
C10


C1
(□:Class<Post>)
.first
C4
Post.first
Test Failure
C2
□: Post
(□:Class<Post>)
.where(□:{ ... }).first
Post.where(□:{id: Int,
slug: Str, ... })
.first
C5
C3
(□:Class<Post>)
.exists?(□:{ ... })
Type Error
C6
Post.where({
id: (□:Int)})
.first
No Terms
Post.where({
slug: (□:Str)})
.first
C7
Test Failure
Post.where({
slug: arg0})
.first
C8
C9
Test Failure
Post.where({
slug: arg1})
.first
Effect: Post.title
◇ substituted with Post.title=
method to satisfy the write effect
t0 = Post.where({
slug: arg1}).first
(□:Post).title =
(□:Str)
□:Post
C11
t0 = Post.where({
slug: arg1}).first
(◇:Post.title)
□:Post
C10


C1
(□:Class<Post>)
.first
C4
Post.first
Test Failure
□: Post
C2
(□:Class<Post>)
.where(□:{ ... }).first
Post.where(□:{id: Int,
slug: Str, ... })
.first
C5
C3
(□:Class<Post>)
.exists?(□:{ ... })
Type Error
C6
Post.where({
id: (□:Int)})
.first
No Terms
Post.where({
slug: (□:Str)})
.first
C7
Test Failure
Post.where({
slug: arg0})
.first
C8
C9
Test Failure
Post.where({
slug: arg1})
.first
Effect: Post.title
t0 = Post.where({
slug: arg1}).first
(◇:Post.title)
□:Post
C10
t0 = Post.where({
slug: arg1}).first
(□:Post).title =
(□:Str)
□:Post
C11
...
t0 = Post.where({
slug: arg1}).first
t0.title =
arg2[:title]
t0
Final solution

Test #2

- We still need to test when the post should not be updated
- If a user other than the author wants to update a post, return the same post without update.

```
post = Post.create(author: 'author', slug: 'hello-world', title: 'Hello World')
updated = update_post('dummy', 'hello-world', author: 'dummy',
                title: 'Foo Bar', slug: 'foobar')

assert { updated.id == post.id }           # the returned post should be same
assert { updated.author == "author" }      # the author field should not be updated
assert { updated.title == "Hello World" } # the title should not be updated
assert { updated.slug == 'hello-world' }
```

Synthesized Solutions

Updated by author

```
post = Post.create(author: 'author',
                   slug:'hello-world',
                   title: 'Hello World')
updated = update_post('author',
                  'hello-world',
                  author: 'dummy',
                  title: 'Foo Bar',
                  slug: 'foobar')
…
assert { updated.title == "Foo Bar" }
```

```
t0 = Post.where(slug:arg1).first
t0.title=arg2[:title]
t0
```

Updated by another user

```
post = Post.create(author: 'author',
                   slug:'hello-world',
                   title: 'Hello World')
updated = update_post('dummy',
                  'hello-world',
                  author: 'dummy',
                  title: 'Foo Bar',
                  slug: 'foobar')

…
assert { updated.title == "Hello World" }
```

```
Post.where(slug: arg1).first
```

Following the same steps yields this expression

Branching

- Combine separate solutions with if-then-else
- Need to synthesize conditional expressions for branches
- The conditional is an expression that evaluates to true in a scenario
- Reuse provided setup for each test and synthesize a conditional using the same algorithm

```
post = Post.create(author: 'author’, slug: 'hello-world’, title: 'Hello World’)
cond = branch('author’, 'hello-world’, author: 'dummy’,
                title: 'Foo Bar’, slug: 'foobar’)

assert { cond == true }
```

Synthesized Solutions

Updated by author		Updated by another user

`post = Post.create(author: 'author',`
`slug:'hello-world',`
`title: 'Hello World')`
`updated = update_post('author',`
`'hello-world',`
`author: 'dummy',`
`title: 'Foo Bar',`
`slug: 'foobar')`
`…`
`assert { updated.title == "Foo Bar" }`
	Test	`post = Post.create(author: 'author',`
		`slug:'hello-world',`
		`title: 'Hello World')`
		`updated = update_post('dummy',`
		`'hello-world',`
		`author: 'dummy',`
		`title: 'Foo Bar',`
		`slug: 'foobar')`
		`…`
		`assert { updated.title == "Hello World" }`

`t0 = Post.where(slug:arg1).first`
`t0.title=arg2[:title]`
`t0`
	Solution	`Post.where(slug: arg1).first`

`true`	Branch	`true`

Branching

- The same conditional holds for different programs, so cannot be used to branch between these
- Synthesize stronger conditional that is true for one test and false for another

```
post = Post.create(author: 'author', slug: 'hello-world', title: 'Hello World')
cond = branch('author', 'hello-world', author: 'dummy',
                title: 'Foo Bar', slug: 'foobar')

assert { cond == true } # first test

post = Post.create(author: 'author', slug: 'hello-world', title: 'Hello World')
cond = branch('dummy', 'hello-world', author: 'dummy',
                title: 'Foo Bar', slug: 'foobar')

assert { cond == false } # second test
```

Synthesized Solutions

Updated by author		Updated by another user

`post = Post.create(author: 'author', slug:'hello-world', title: 'Hello World')`
`updated = update_post('author', 'hello-world', author: 'dummy', title: 'Foo Bar', slug: 'foobar')`
`…`
`assert { updated.title == "Foo Bar" }`
	Test	`post = Post.create(author: 'author', slug:'hello-world', title: 'Hello World')`
		`updated = update_post('dummy', 'hello-world', author: 'dummy', title: 'Foo Bar', slug: 'foobar')`
		`…`
		`assert { updated.title == "Hello World" }`

`t0 = Post.where(slug:arg1).first`
`t0.title=arg2[:title]`
`t0`
	Solution	`Post.where(slug: arg1).first`

`Post.exists?(author: arg0, slug: arg1)`	Branch	`!Post.exists?(author: arg0, slug: arg1)`

Synthesized Method

Put everything together in an if-then-else to build final program:

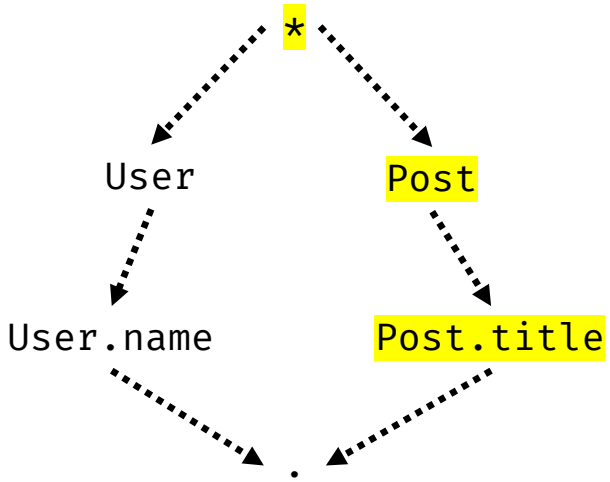
```
def update_post(arg0, arg1, arg2)
  if Post.exists?(author: arg0, slug: arg1)
    t0 = Post.where(slug:arg1).first
    t0.title=arg2[:title]
    t0
  else
    Post.where(slug: arg1).first
  end
end
```

Effect annotations are flexible

- Effects are abstract labels, can refer to abstract region in objects, or a class or just pure/impure
- Use lighter annotations to reduce annotation burden or when precise label is not possible

```
title= :: ‘(Str) → Str’, write: [‘Post.title’]
delete :: ‘() → Bool’  , write: [‘Post’]
```

```
t0 = Post.where({slug: arg1}).first
◇: Post.title
□: Post
```



- Effect labels determine the search order for candidate methods (first title=, then delete)