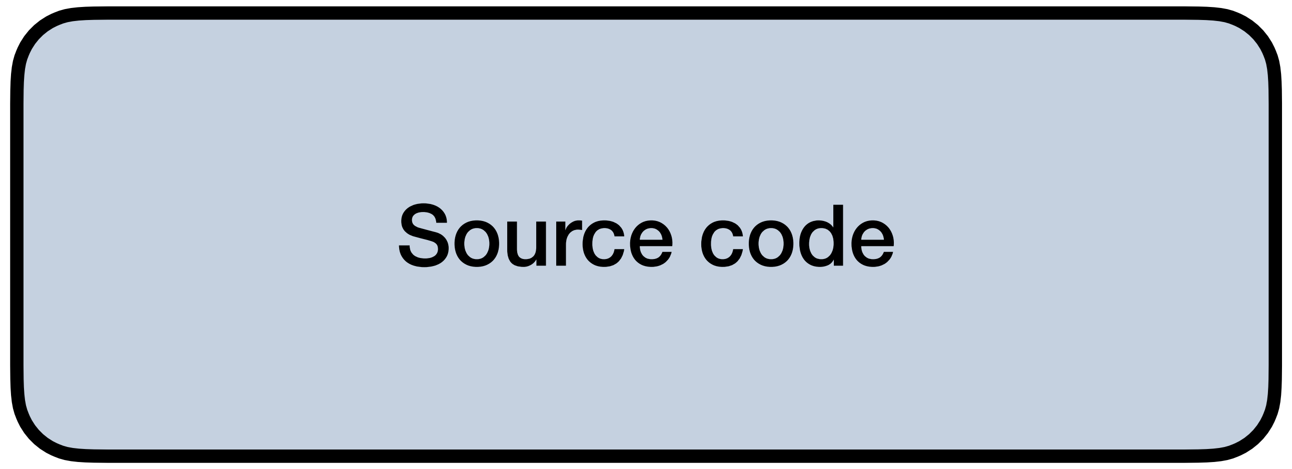
Nanopass Back-Translation of Call-Return Trees for Mechanized Secure Compilation Proofs

Jérémy Thibault, Joseph Lenormand, Catalin Hritcu

MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

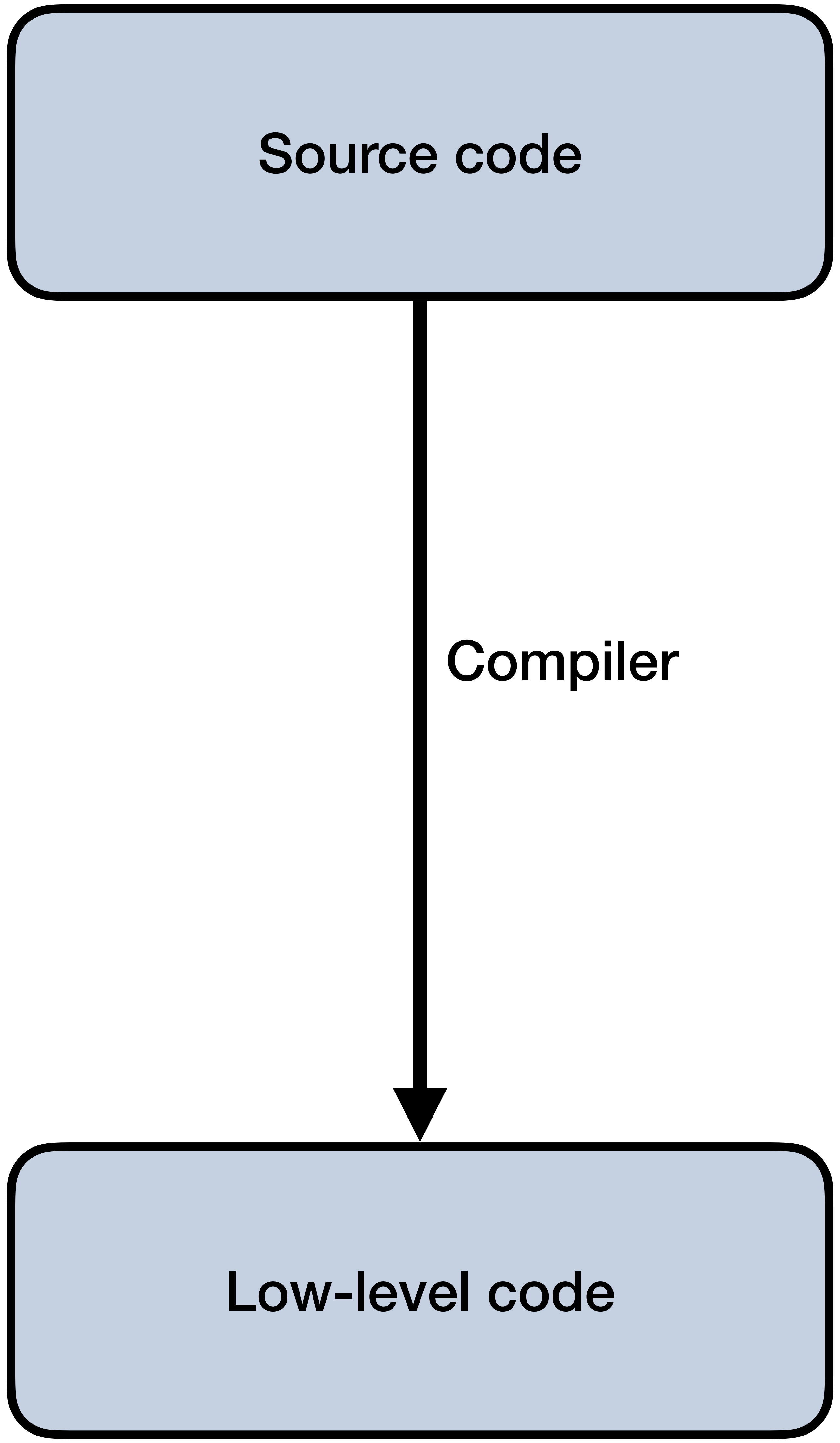
Source code



Source code

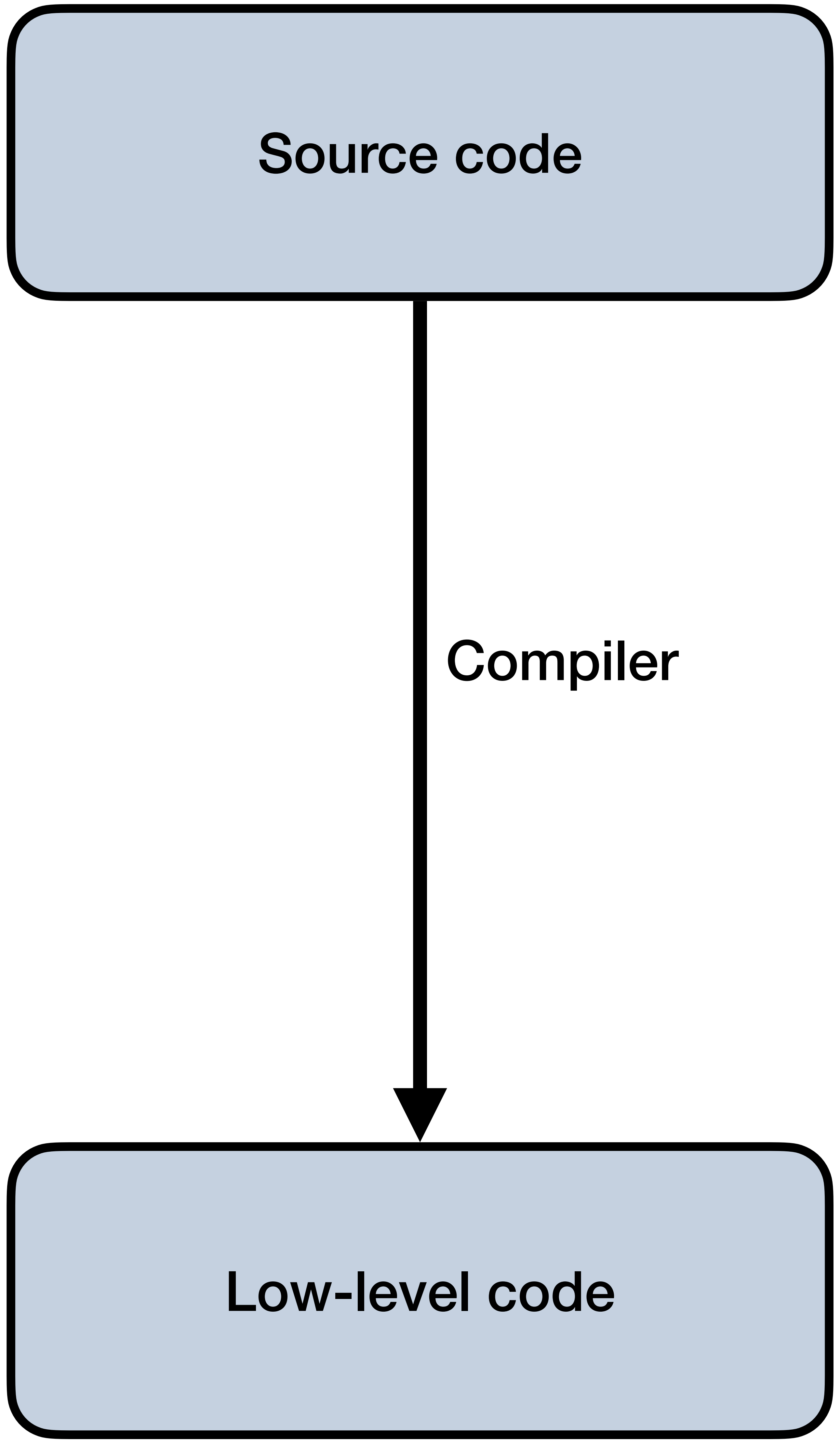
High-level abstractions:

- Types
- Structured control-flow
- Verification, static analysis, etc.



High-level abstractions:

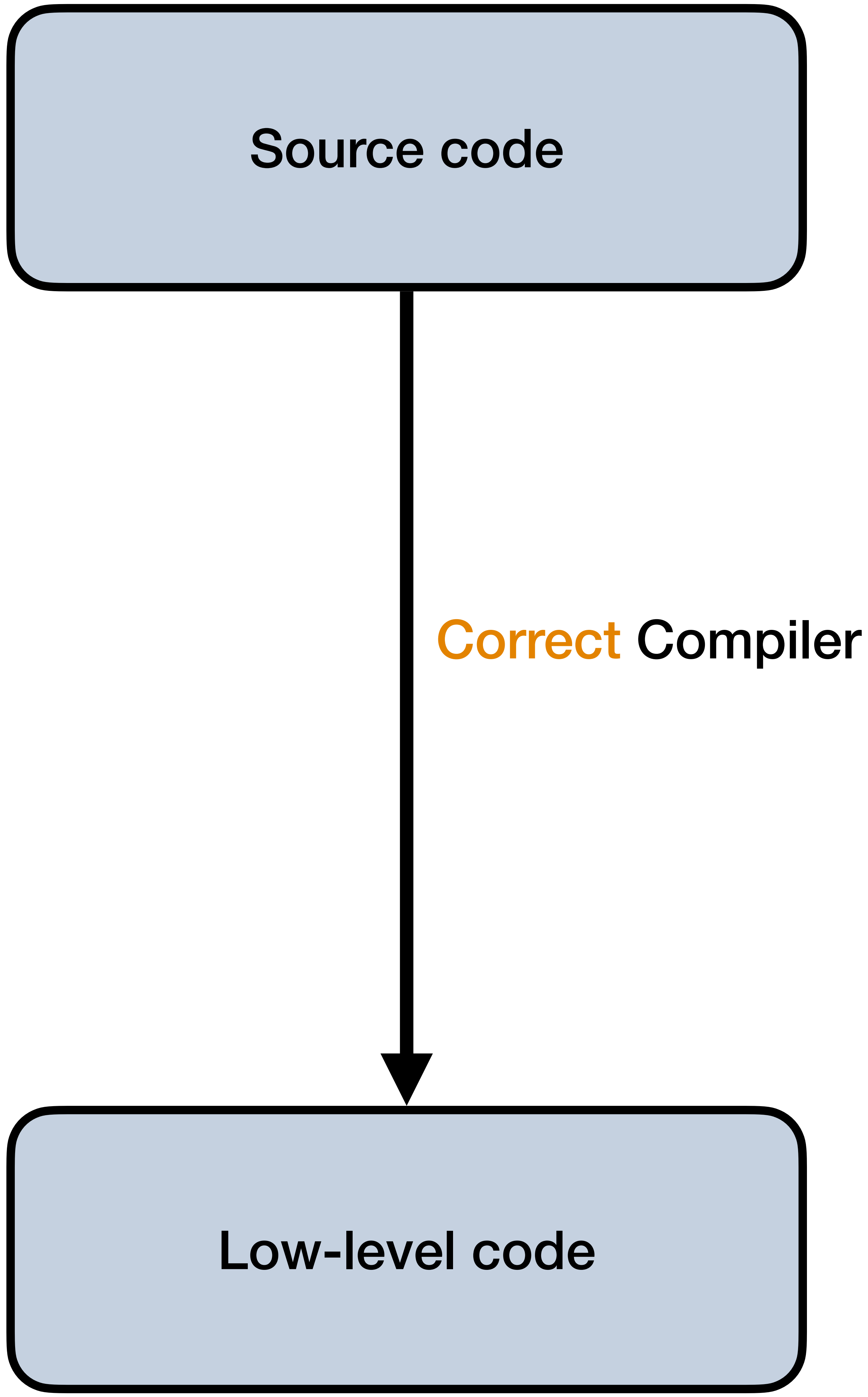
- Types
- Structured control-flow
- Verification, static analysis, etc.



High-level abstractions:

- Types
- Structured control-flow
- Verification, static analysis, etc.

No high-level abstraction anymore



Source code

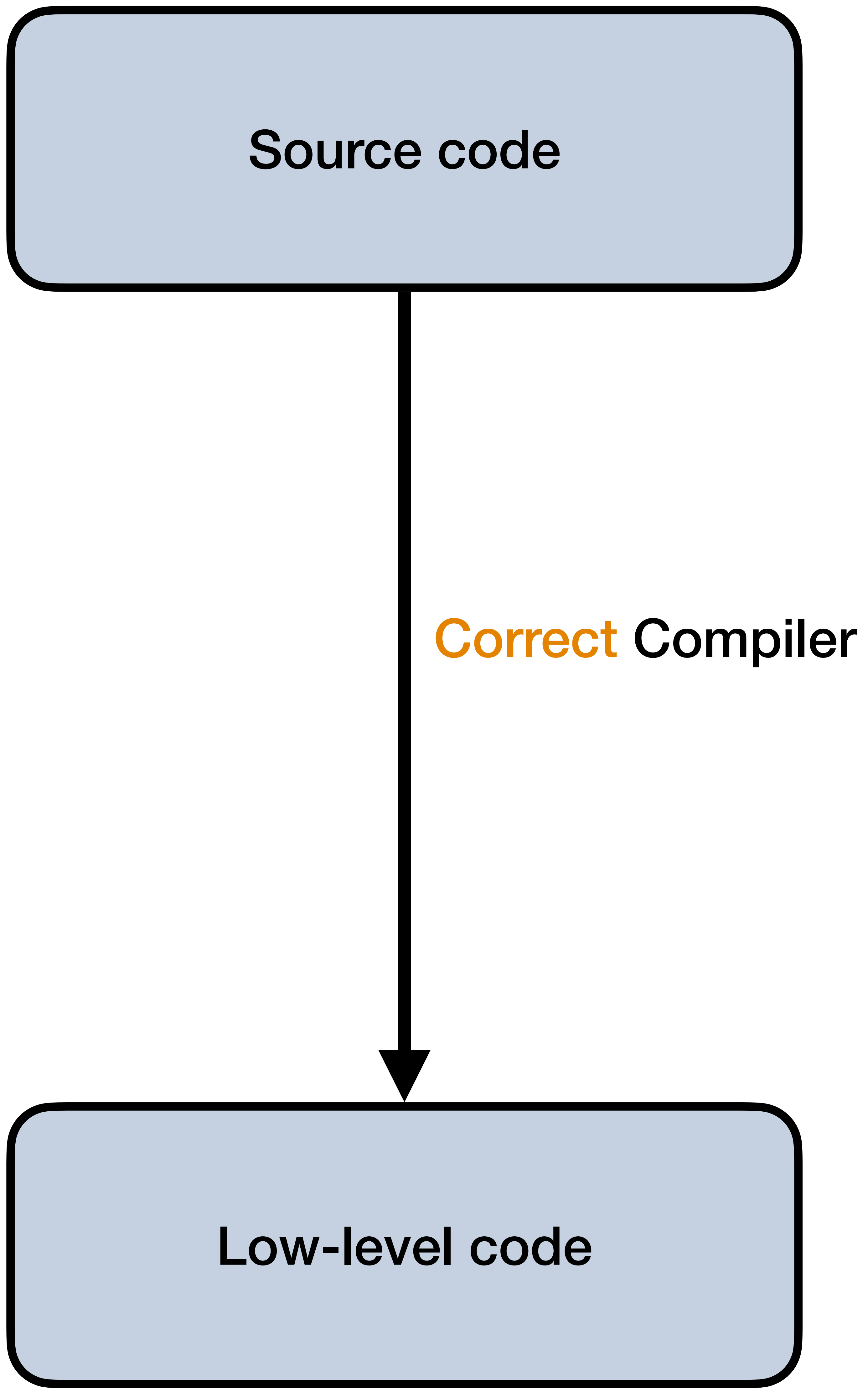
Correct Compiler

Low-level code

High-level abstractions:

- Types
- Structured control-flow
- Verification, static analysis, etc.

No high-level abstraction anymore



Source code

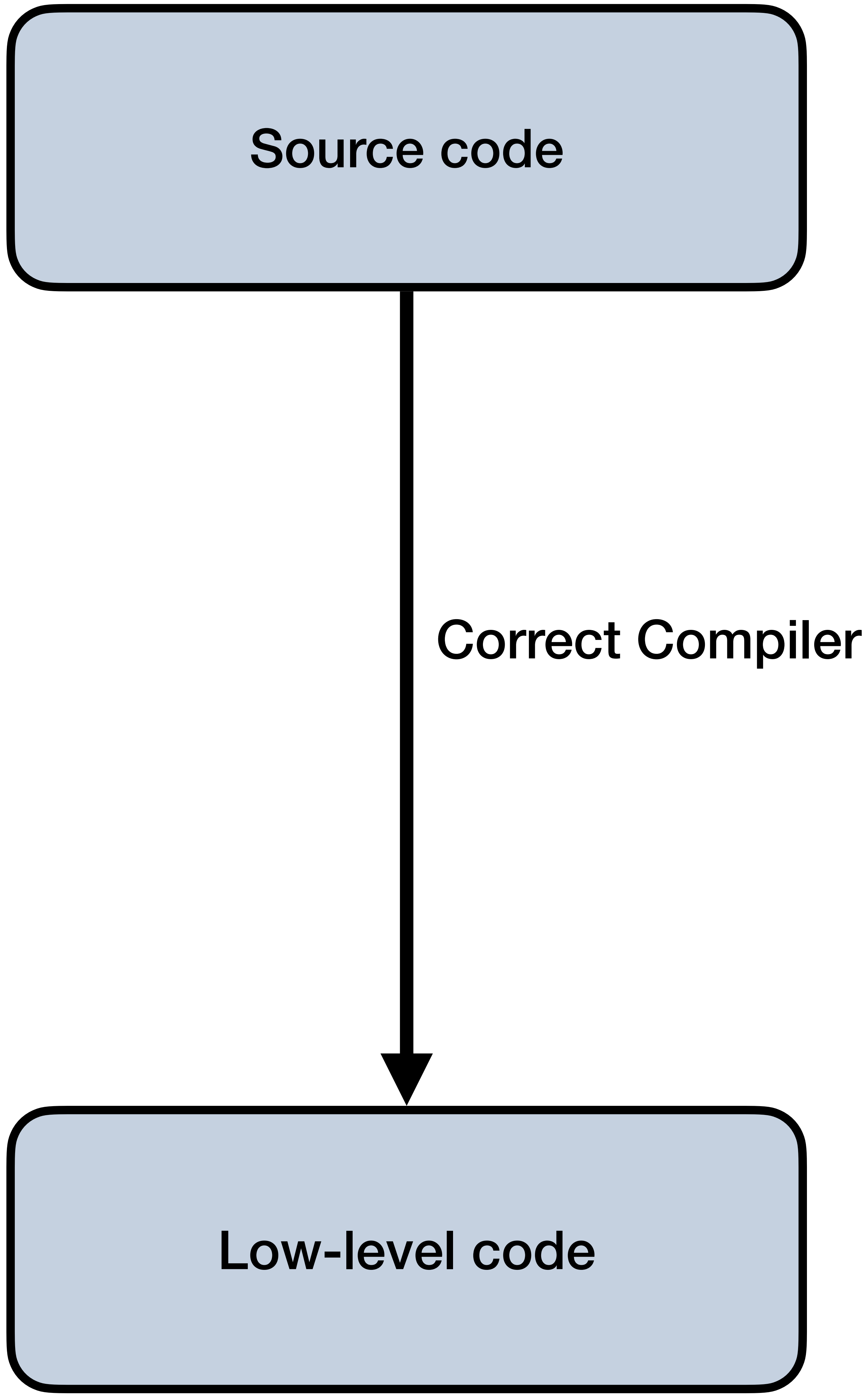
Correct Compiler

Low-level code

High-level abstractions:

- Types
- Structured control-flow
- Verification, static analysis, etc.

High-level abstractions preserved?

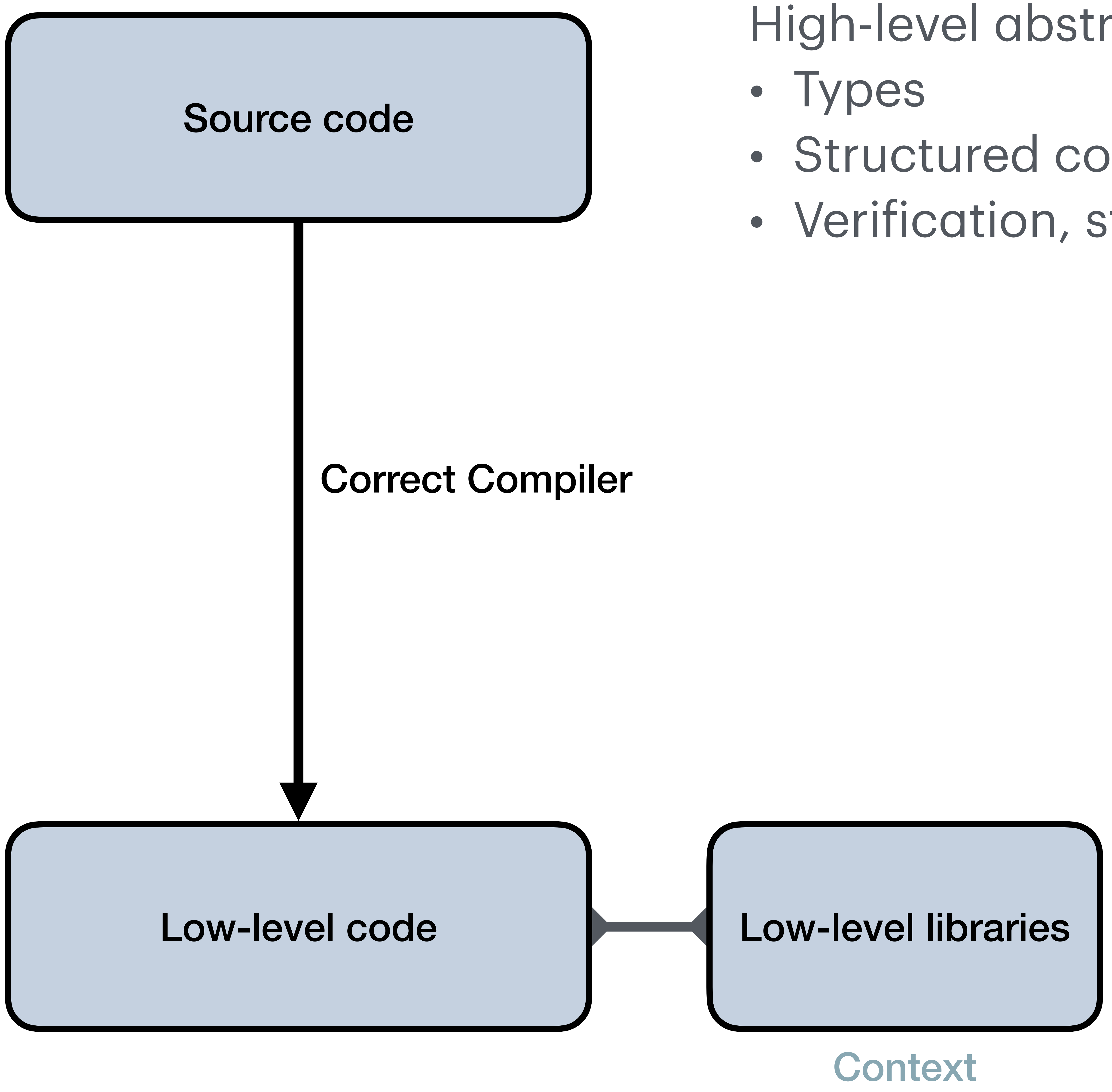


High-level abstractions:

- Types
- Structured control-flow
- Verification, static analysis, etc.

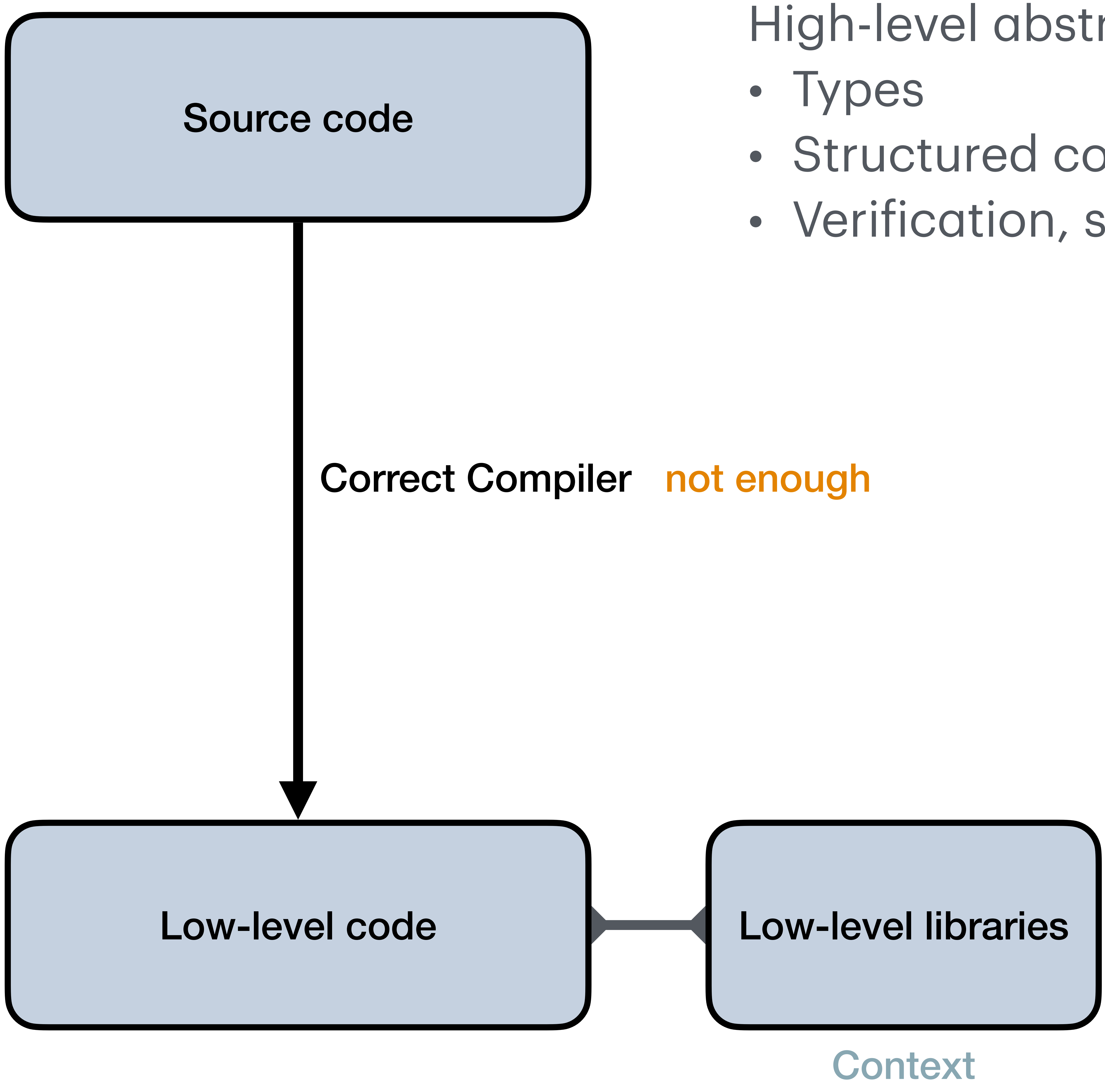
High-level abstractions:

- Types
- Structured control-flow
- Verification, static analysis, etc.



High-level abstractions:

- Types
- Structured control-flow
- Verification, static analysis, etc.



Source code

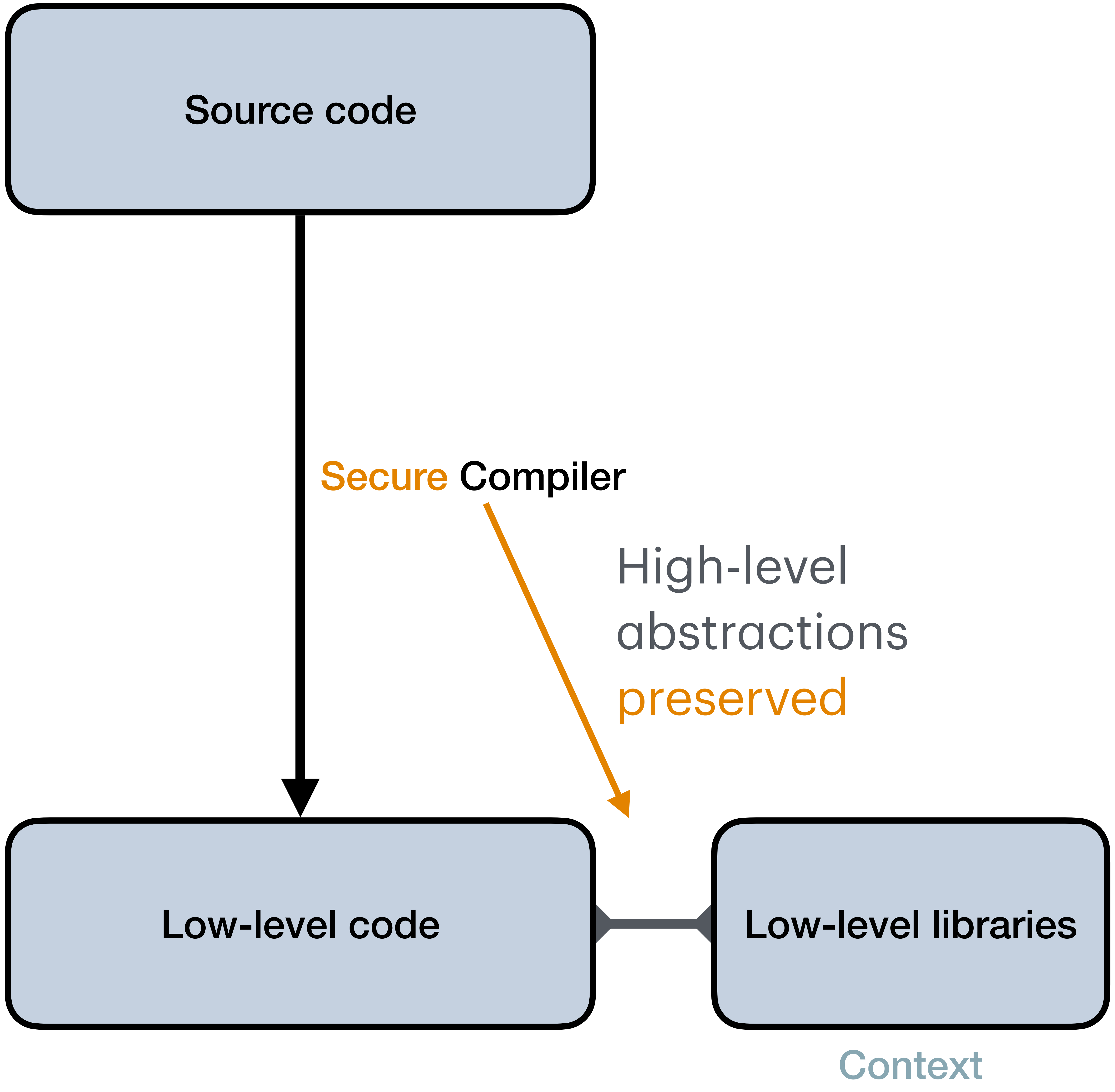
Correct Compiler not enough

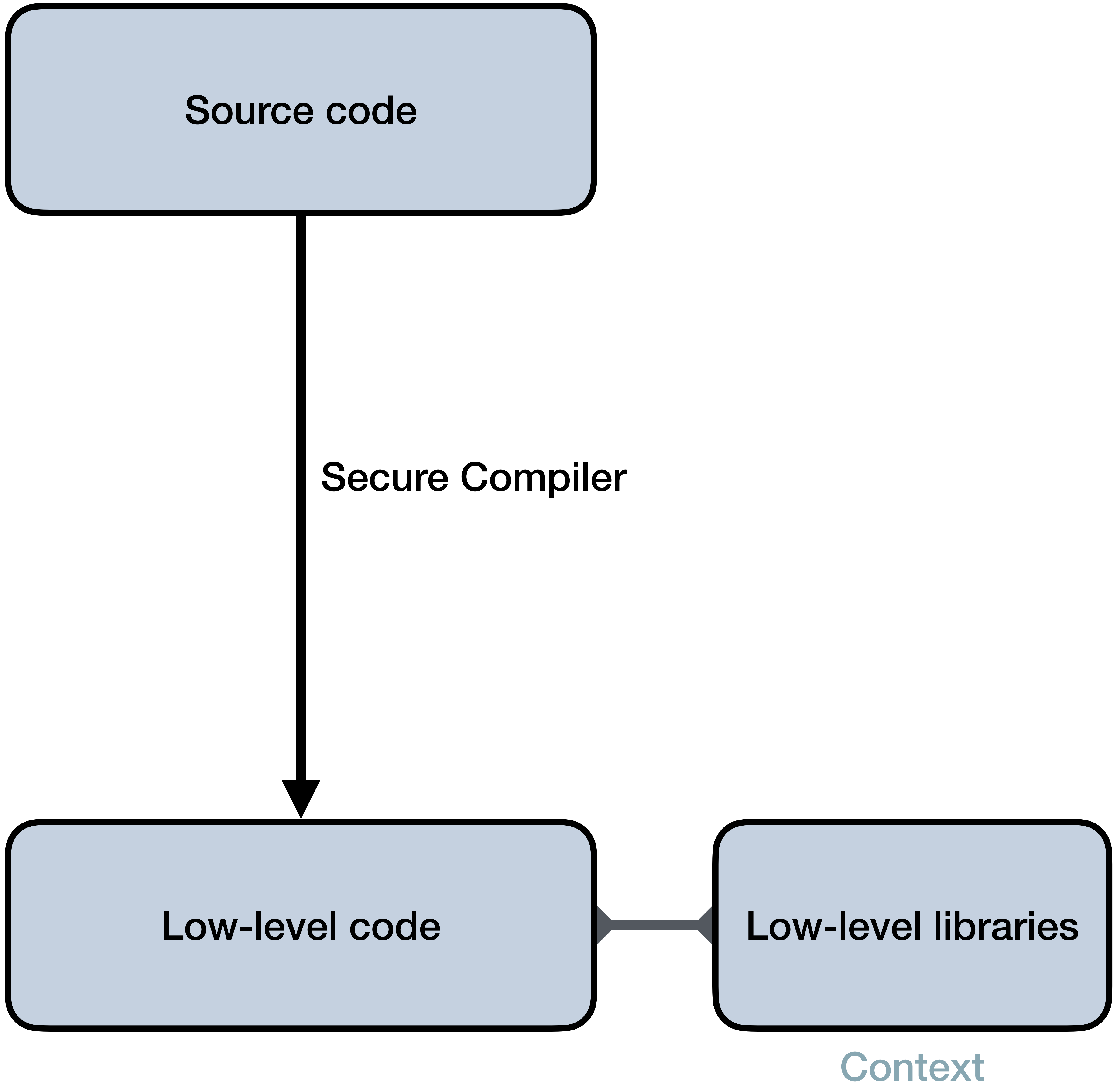
Low-level code

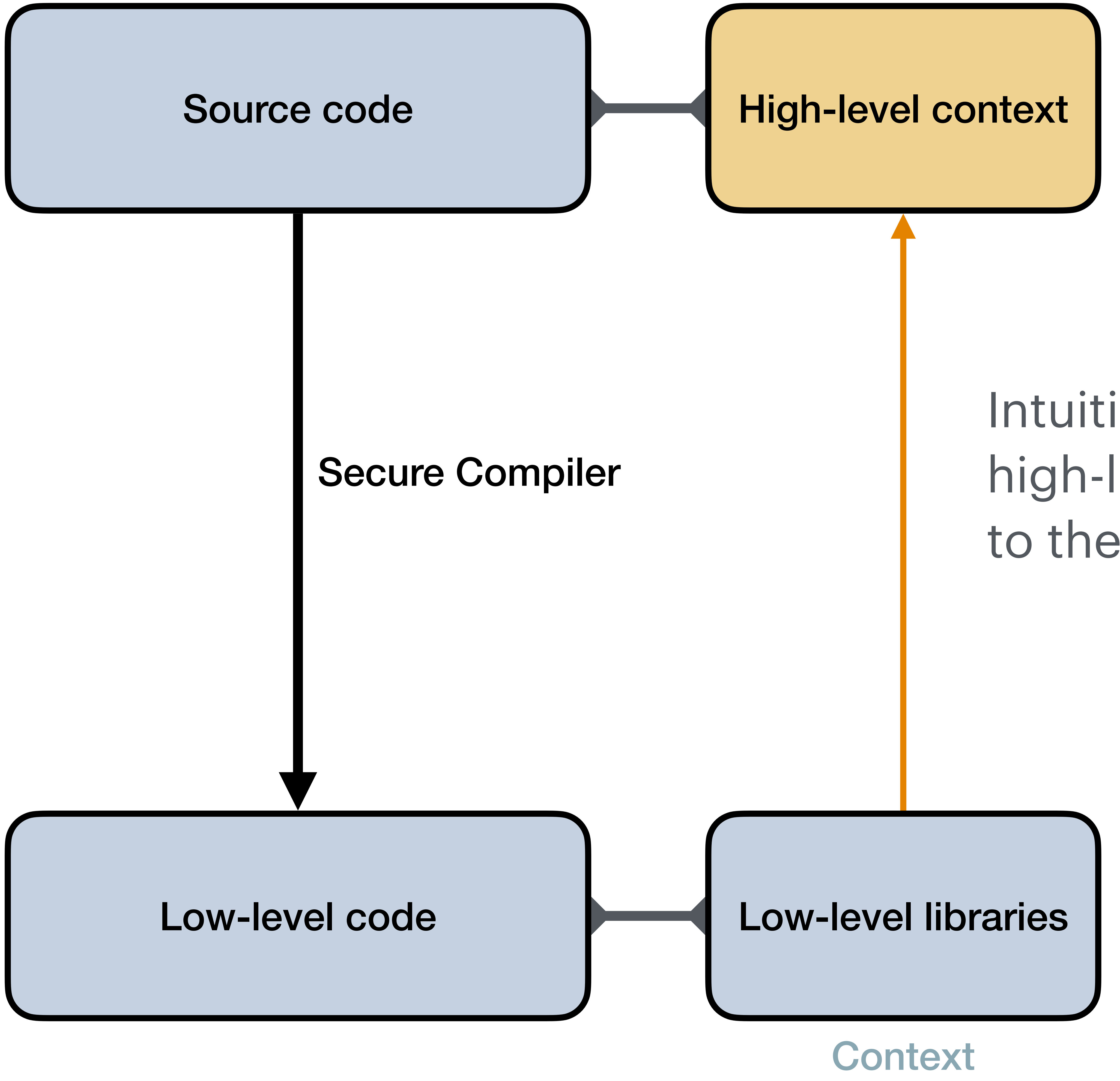
Low-level libraries

Context

Unverified, untrusted,
possibly full of bugs or
even attacker-controlled


Source code
Secure Compiler
High-level
abstractions
preserved
Low-level code
Low-level libraries
Context


Source code
Secure Compiler
Low-level code
Low-level libraries
Context

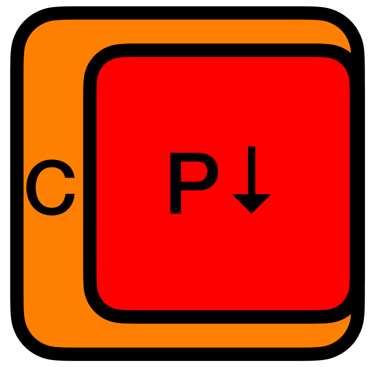


Intuition: existence of a high-level context “similar” to the low-level one

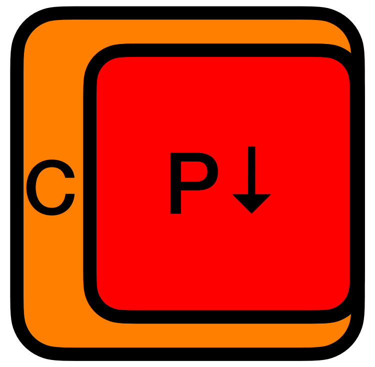
Example: Robust Safety Preservation (RSP)

Example: Robust Safety Preservation (RSP)

If

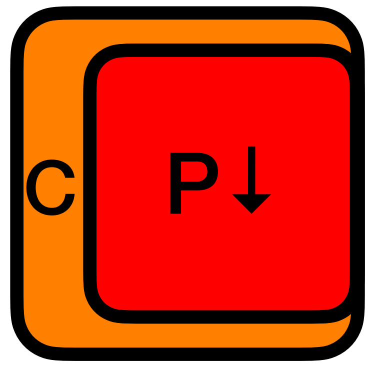


Example: Robust Safety Preservation (RSP)

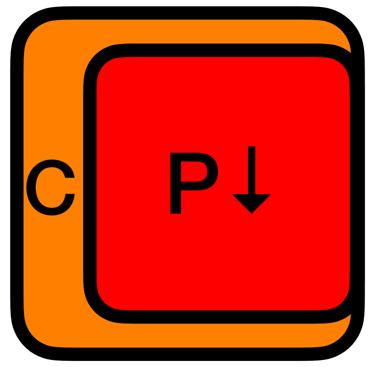
If 

produces finite trace **m**

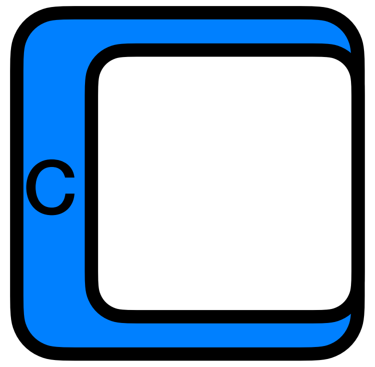
Example: Robust Safety Preservation (RSP)

If  produces finite trace **m** (witness of a violation of a certain safety property)

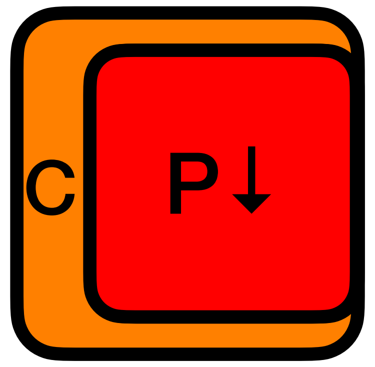
Example: Robust Safety Preservation (RSP)

If 

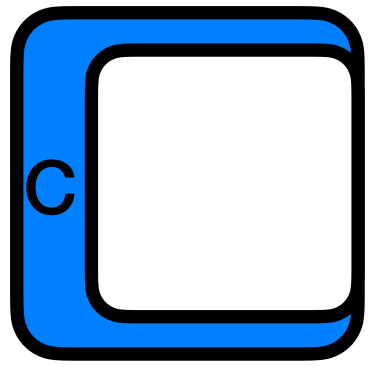
produces finite trace **m** (witness of a violation of a certain safety property)

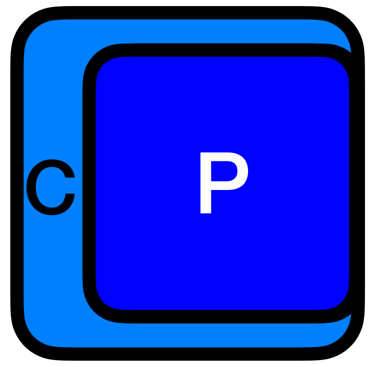
then there exists 

Example: Robust Safety Preservation (RSP)

If 

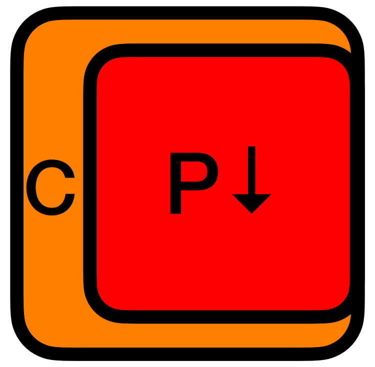
produces finite trace **m** (witness of a violation of a certain safety property)

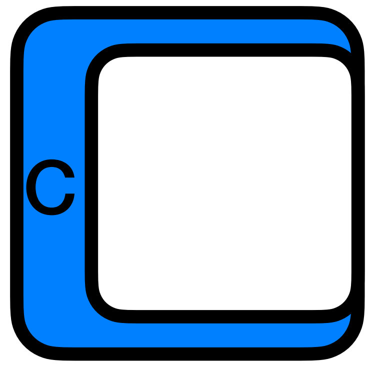
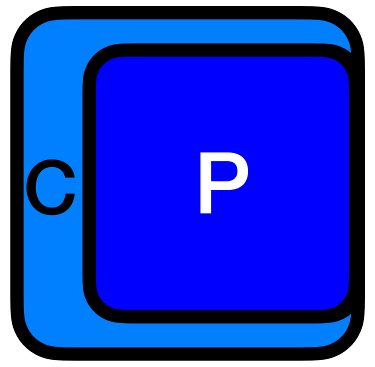
then there exists 

such that 

produces **m**

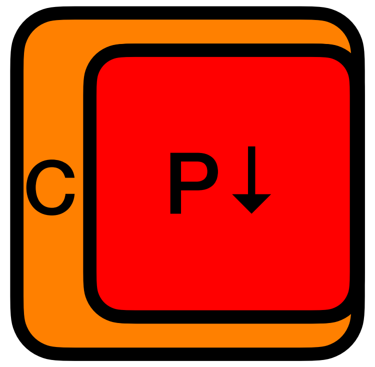
Example: Robust Safety Preservation (RSP)

If  produces finite trace **m** (witness of a violation of a certain safety property)

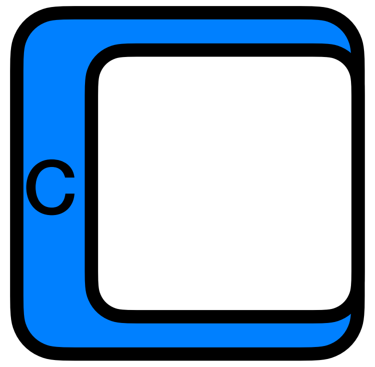
then there exists  such that  produces **m**

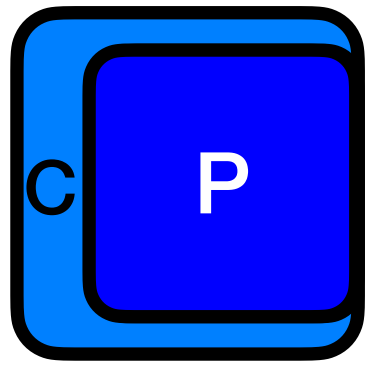
⇒ preservation of safety properties

Example: Robust Safety Preservation (RSP)

If 

produces finite trace **m** (witness of a violation of a certain safety property)

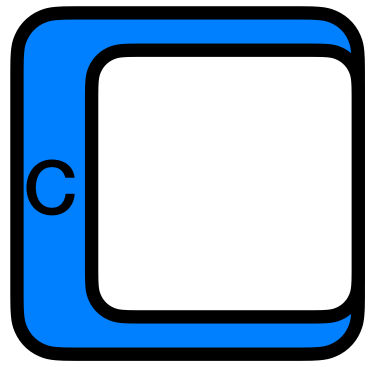
then there exists 

such that 

produces **m**

⇒ preservation of safety properties

Trace-based back-translation:

given **m**, build a context 

A Proof Technique for RSP

A Proof Technique for RSP

Prior work: proof technique for RSP designed to be mechanized

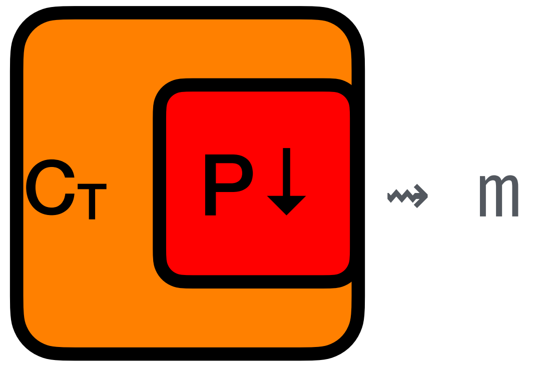
(Abate et al. (2018), When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise)

A Proof Technique for RSP

Prior work: proof technique for RSP designed to be mechanized

Source

Target

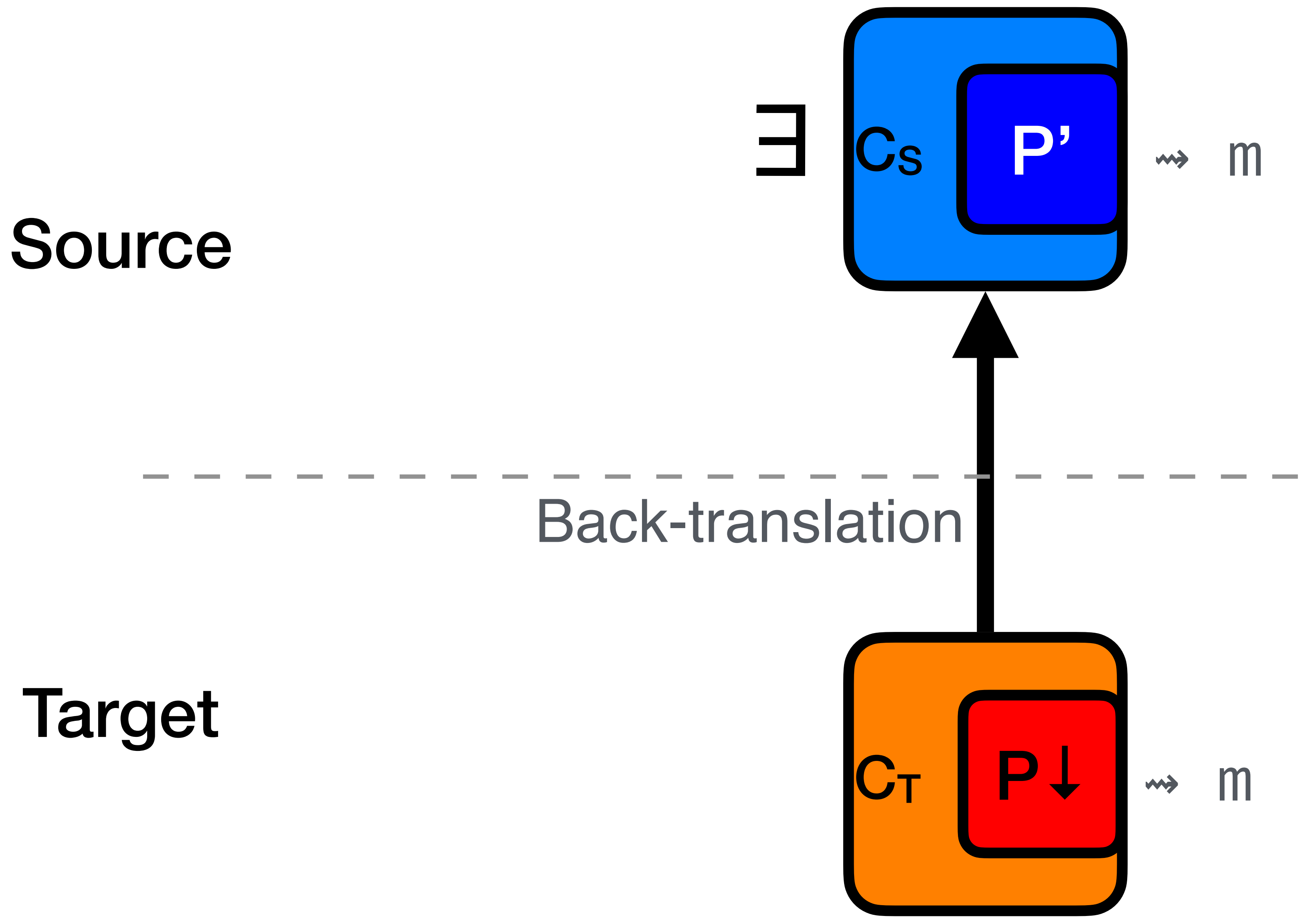


C_T $P\downarrow$ $\rightsquigarrow$ m

(Abate et al. (2018), When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise)

A Proof Technique for RSP

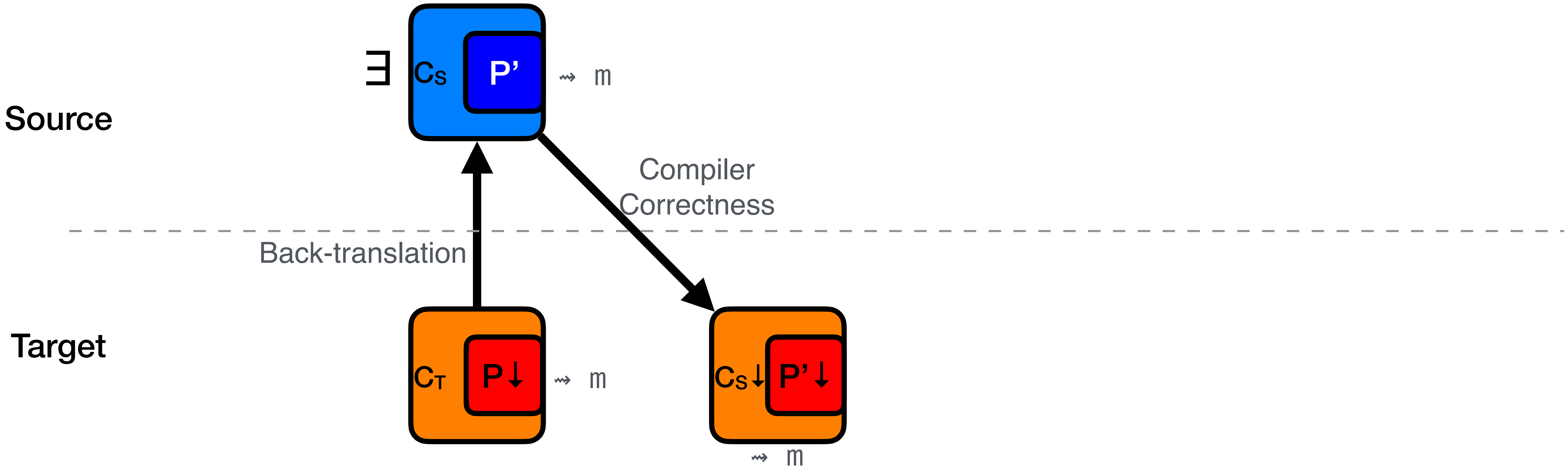
Prior work: proof technique for RSP designed to be mechanized



(Abate et al. (2018), When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise)

A Proof Technique for RSP

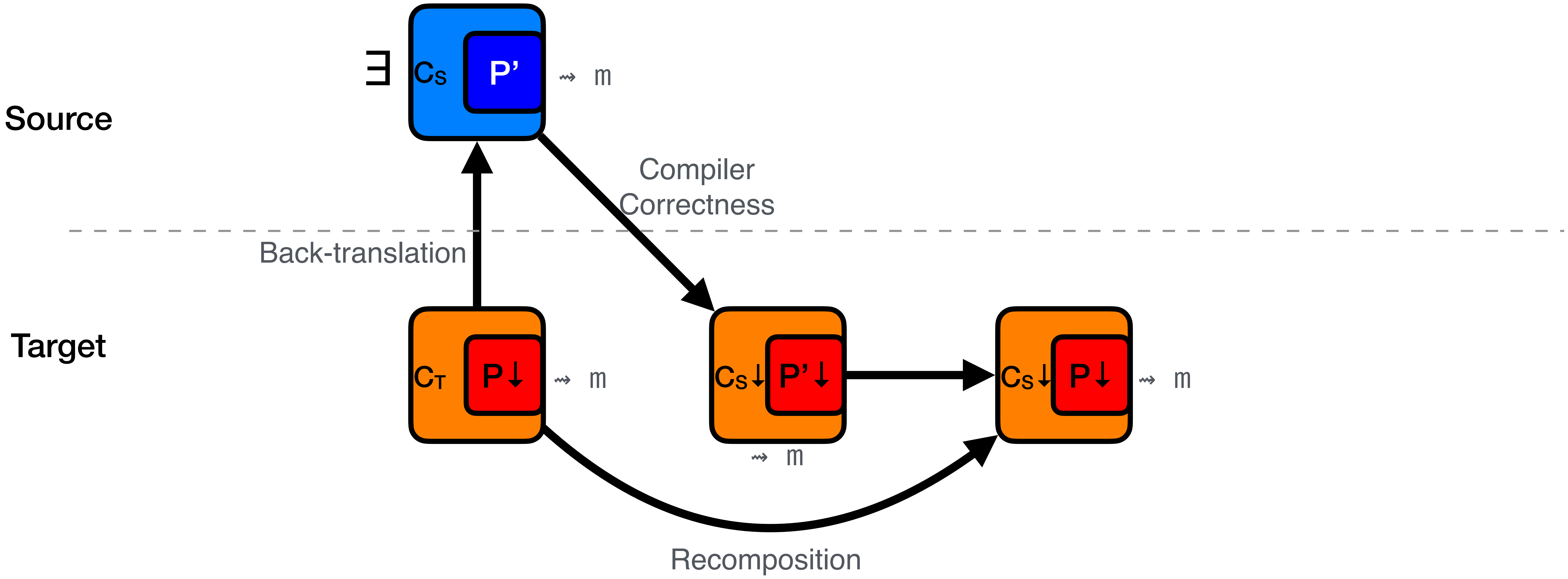
Prior work: proof technique for RSP designed to be mechanized



(Abate et al. (2018), When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise)

A Proof Technique for RSP

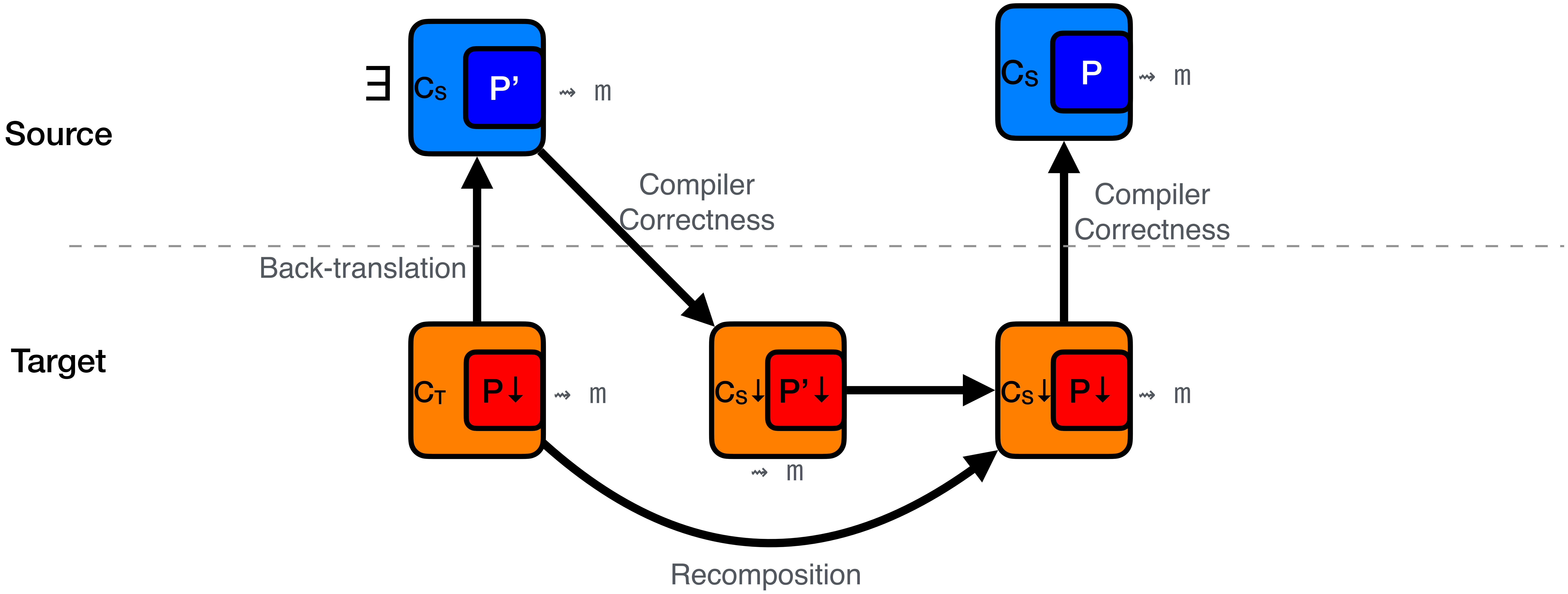
Prior work: proof technique for RSP designed to be mechanized



(Abate et al. (2018), When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise)

A Proof Technique for RSP

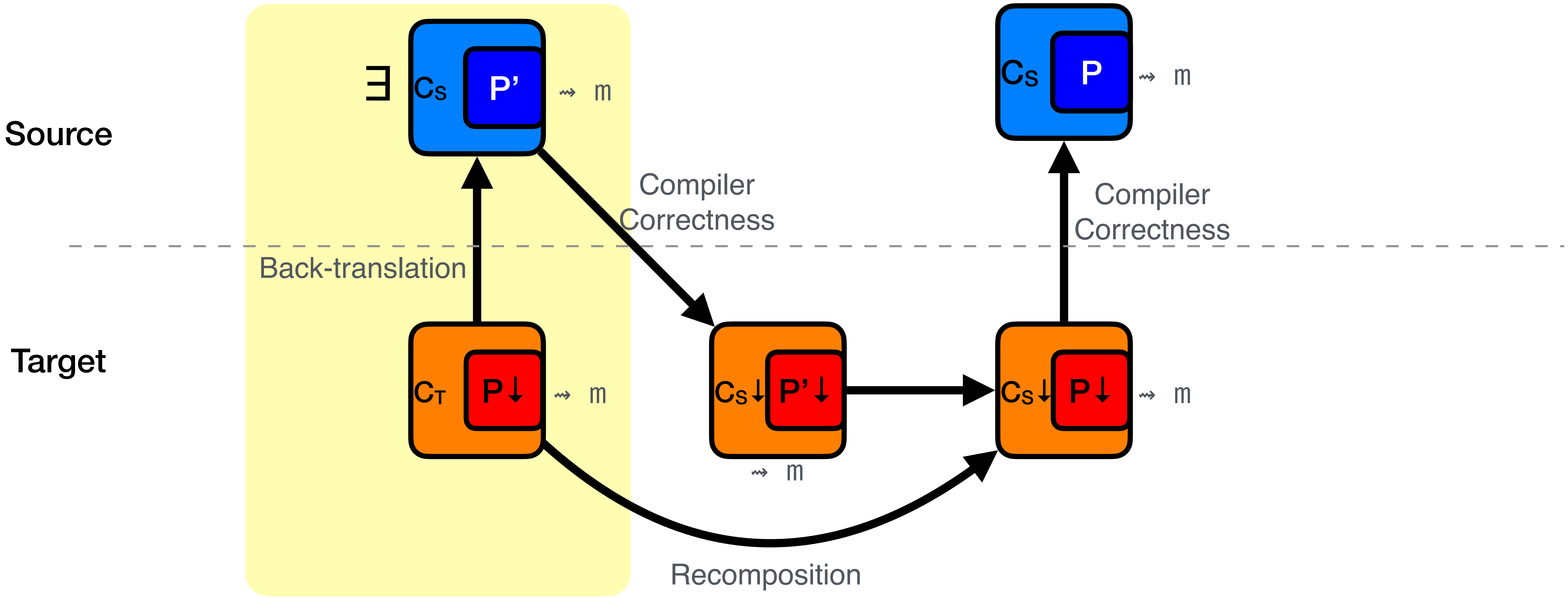
Prior work: proof technique for RSP designed to be mechanized



(Abate et al. (2018), When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise)

A Proof Technique for RSP

Prior work: proof technique for RSP designed to be mechanized



(Abate et al. (2018), When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise)

A Simple Back-Translation for RSP

```
Call C P 1;
Call P C 2;
Ret  C P 3;
Ret  P C 4;
```

A Simple Back-Translation for RSP

```
Call C P 1;
Call P C 2;
Ret  C P 3;
Ret  P C 4;
```

ctr ↦ 0

C

ctr ↦ 0

P

A Simple Back-Translation for RSP

```
c():
```

```
Call C P 1;
Call P C 2;
Ret  C P 3;
Ret  P C 4;
```

ctr ⟼ 0

C

ctr ⟼ 0

P

A Simple Back-Translation for RSP

```
Call C P 1;

Call P C 2;

Ret  C P 3;

Ret  P C 4;
```

```
C():
if (ctr = 0) {
```

ctr ⟼ 0

C

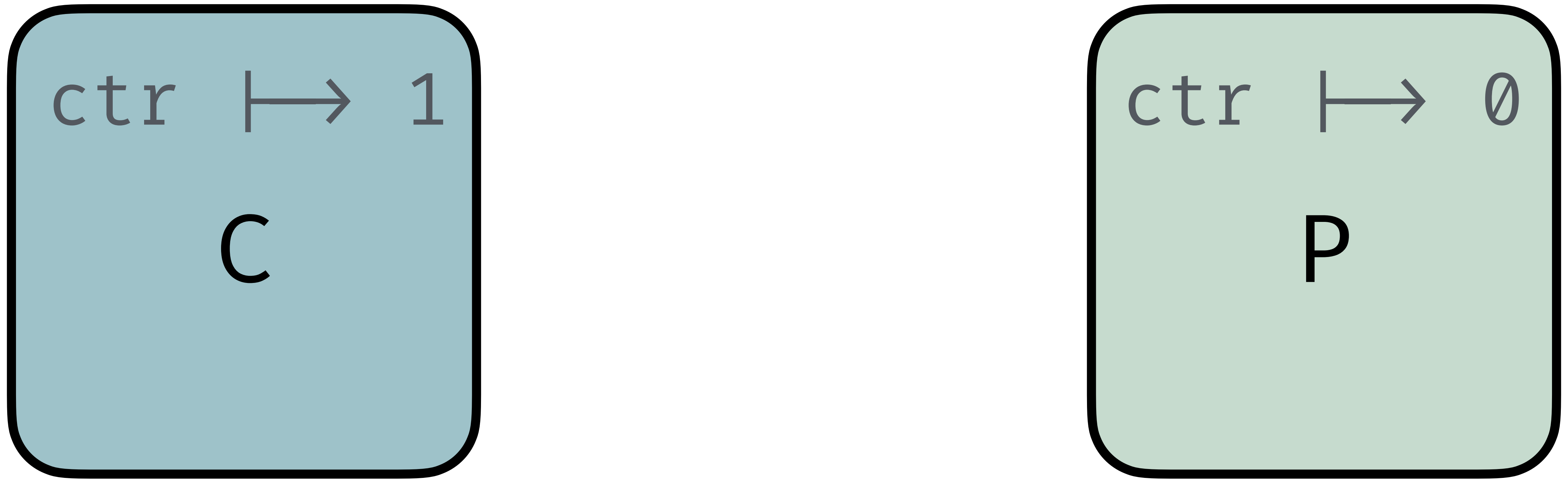
ctr ⟼ 0

P

A Simple Back-Translation for RSP

```
Call  C P 1;

Call  P C 2;

Ret   C P 3;

Ret   P C 4;
```

```
C():
if (ctr = 0) {
  ctr++;
```



ctr ⟼ 1

C

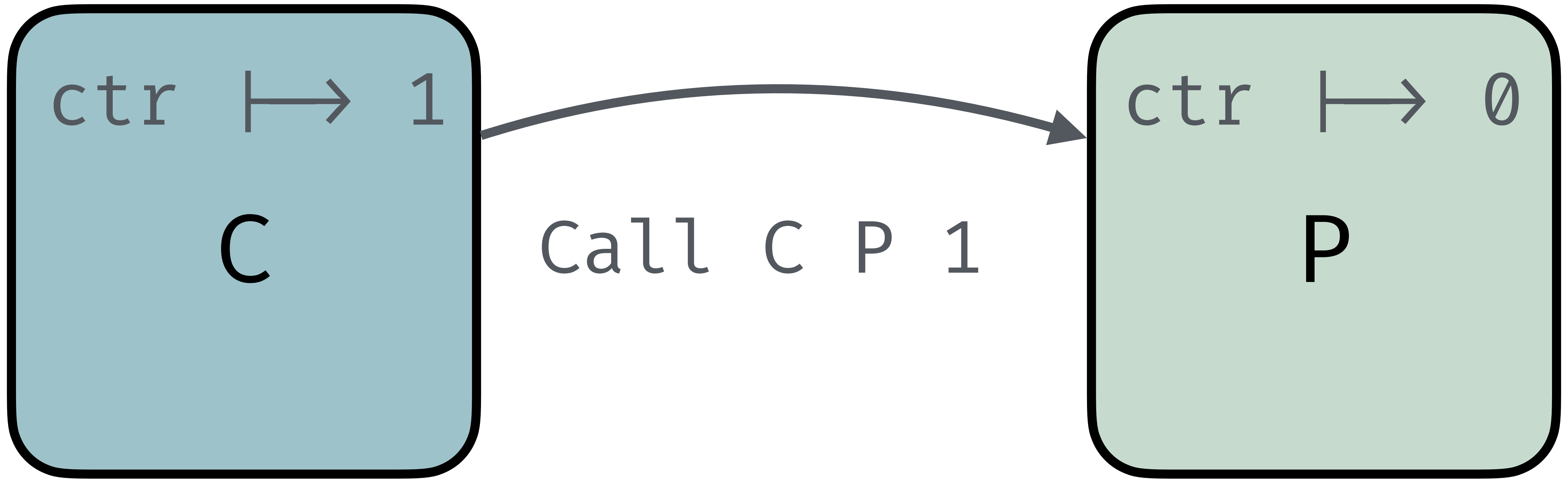
ctr ⟼ 0

P

A Simple Back-Translation for RSP

```
Call C P 1;

Call P C 2;

Ret  C P 3;

Ret  P C 4;
```

```
C():
if (ctr = 0) {
  ctr++;
  P(1);
```



A Simple Back-Translation for RSP

```
Call  C P 1;

Call  P C 2;

Ret   C P 3;

Ret   P C 4;
```

```
C():
if (ctr = 0) {
  ctr++;
  P(1);
```

ctr ⟼ 1

C

ctr ⟼ 0

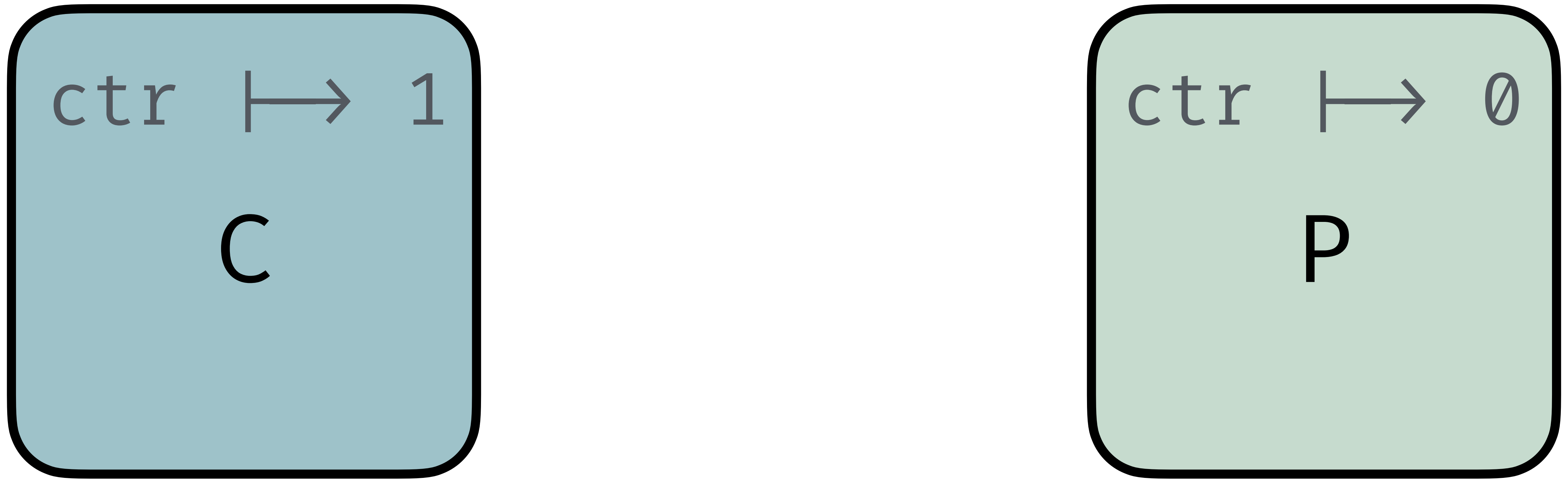
P

A Simple Back-Translation for RSP

```
Call C P 1;

Call P C 2;

Ret  C P 3;

Ret  P C 4;
```

```
C():
if (ctr = 0) {
  ctr++;
  P(1);
```

```
P():
```



ctr ↦ 1

C

ctr ↦ 0

P

A Simple Back-Translation for RSP

```
Call  C P 1;

Call  P C 2;

Ret   C P 3;

Ret   P C 4;
```

```
C():
if (ctr = 0) {
  ctr++;
  P(1);
```

```
P():
if (ctr = 0) {
```

ctr ↦ 1

C

ctr ↦ 0

P

A Simple Back-Translation for RSP

```
Call C P 1;

Call P C 2;

Ret  C P 3;

Ret  P C 4;
```

```
C():
if (ctr = 0) {
  ctr++;
  P(1);
```

```
P():
if (ctr = 0) {
  ctr++;
```

ctr ↦ 1

C

ctr ↦ 1

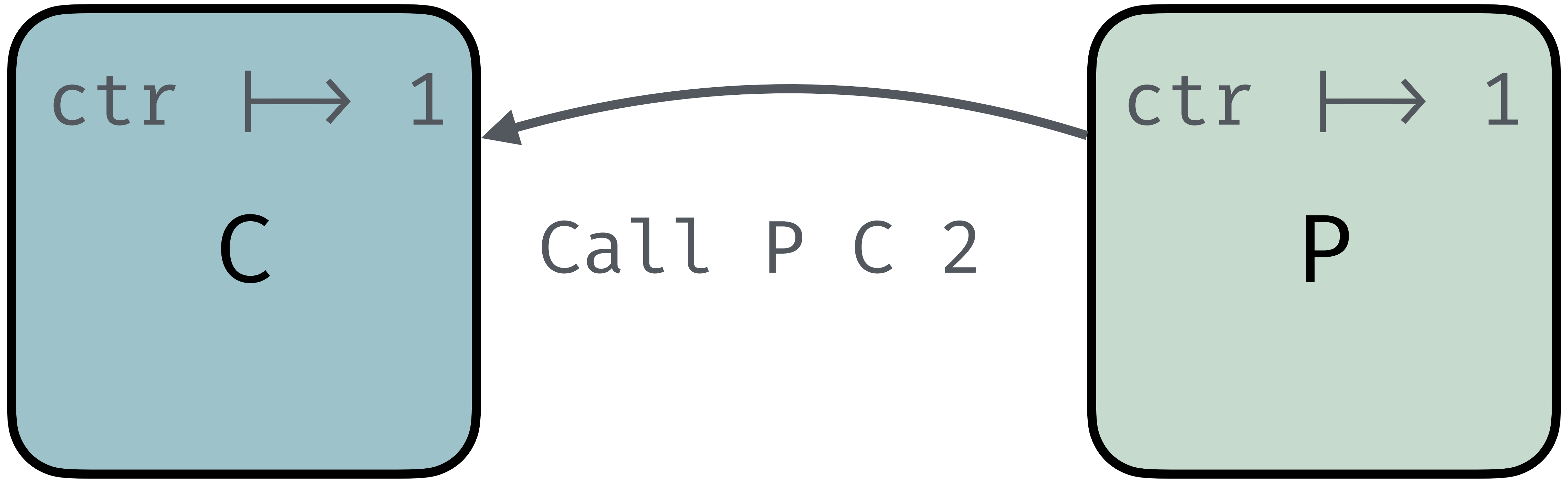
P

A Simple Back-Translation for RSP

```
Call C P 1;
Call P C 2;
Ret  C P 3;
Ret  P C 4;
```

```
C():
if (ctr = 0) {
  ctr++;
  P(1);
```

```
P():
if (ctr = 0) {
  ctr++;
  C(2);
```

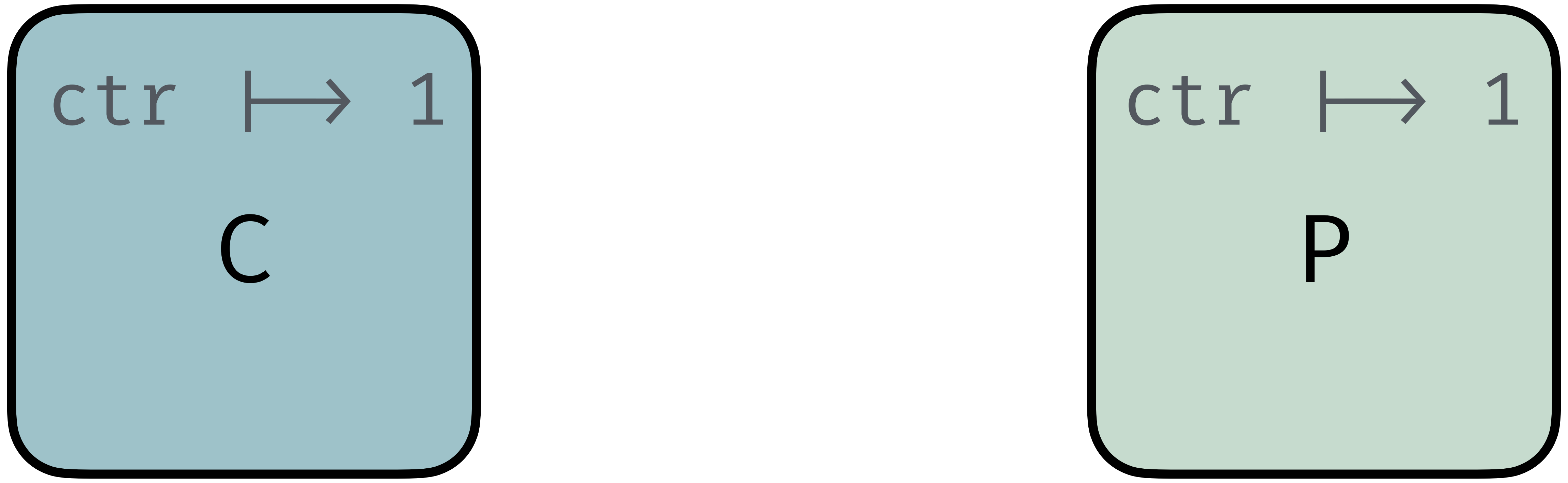


A Simple Back-Translation for RSP

```
Call C P 1;

Call P C 2;

Ret  C P 3;

Ret  P C 4;
```

```
C():
if (ctr = 0) {
  ctr++;
  P(1);
```

```
P():
if (ctr = 0) {
  ctr++;
  C(2);
```



A Simple Back-Translation for RSP

```
Call C P 1;

Call P C 2;

Ret  C P 3;

Ret  P C 4;
```

```
C():
if (ctr = 0) {
  ctr++;
  P(1);
  C();
```

```
P():
if (ctr = 0) {
  ctr++;
  C(2);
```

ctr ⟼ 1

C

ctr ⟼ 1

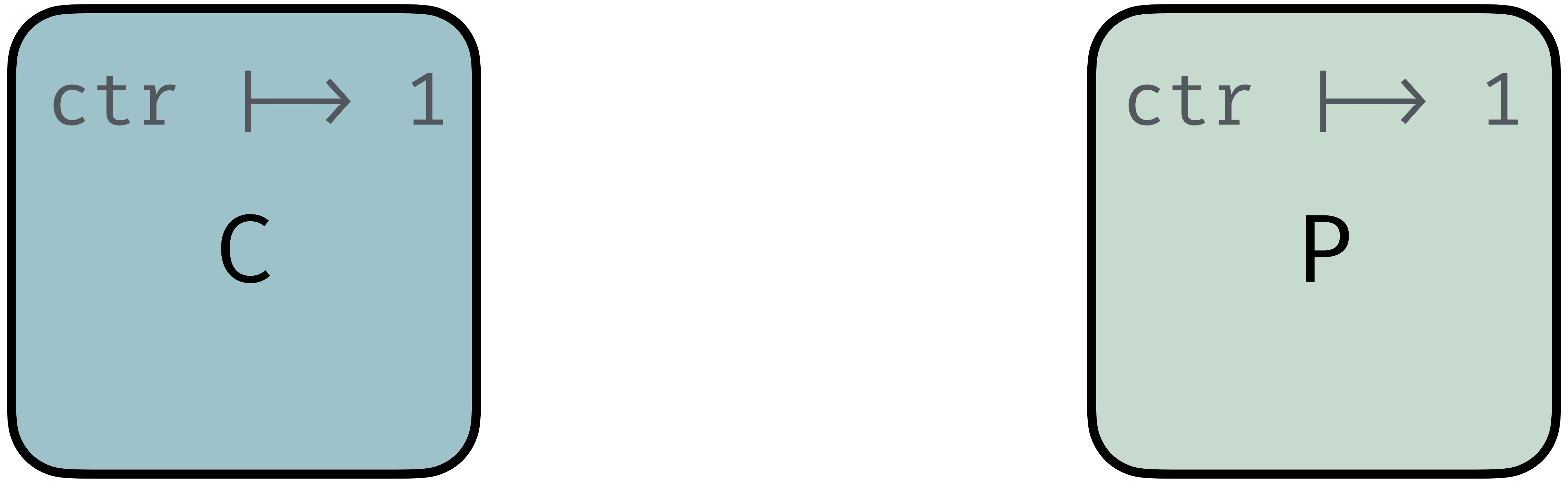
P

A Simple Back-Translation for RSP

```
Call C P 1;

Call P C 2;

Ret  C P 3;

Ret  P C 4;
```

```
C():
if (ctr = 0) {
  ctr++;
  P(1);
  C();
} else if (ctr = 1) {
```

```
P():
if (ctr = 0) {
  ctr++;
  C(2);
```



ctr ↦ 1

C

ctr ↦ 1

P

A Simple Back-Translation for RSP

```
Call C P 1;

Call P C 2;

Ret  C P 3;

Ret  P C 4;
```

```
C():
if (ctr = 0) {
  ctr++;
  P(1);
  C();
} else if (ctr = 1) {
  ctr++;
```

```
P():
if (ctr = 0) {
  ctr++;
  C(2);
```

ctr ⟼ 2

C

ctr ⟼ 1

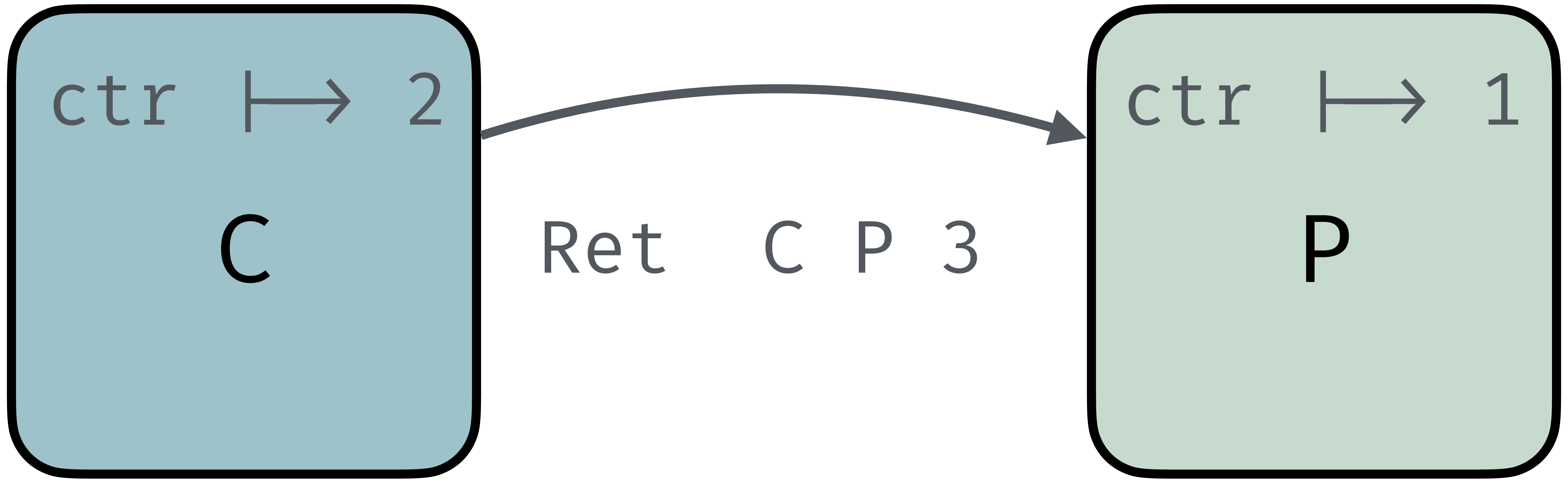
P

A Simple Back-Translation for RSP

```
Call C P 1;

Call P C 2;

Ret  C P 3;

Ret  P C 4;
```

```
C():
if (ctr = 0) {
  ctr++;
  P(1);
  C();
} else if (ctr = 1) {
  ctr++;
  return 3
}
```

```
P():
if (ctr = 0) {
  ctr++;
  C(2);
```

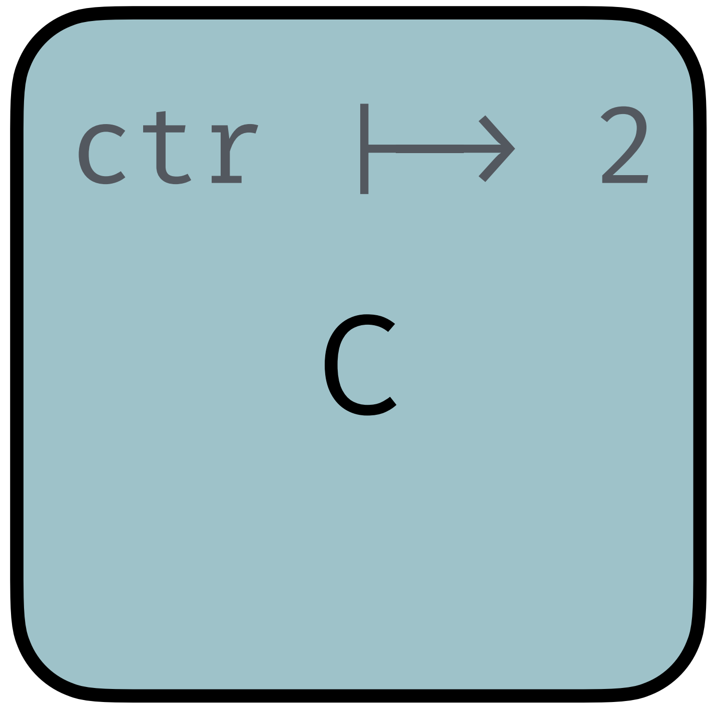


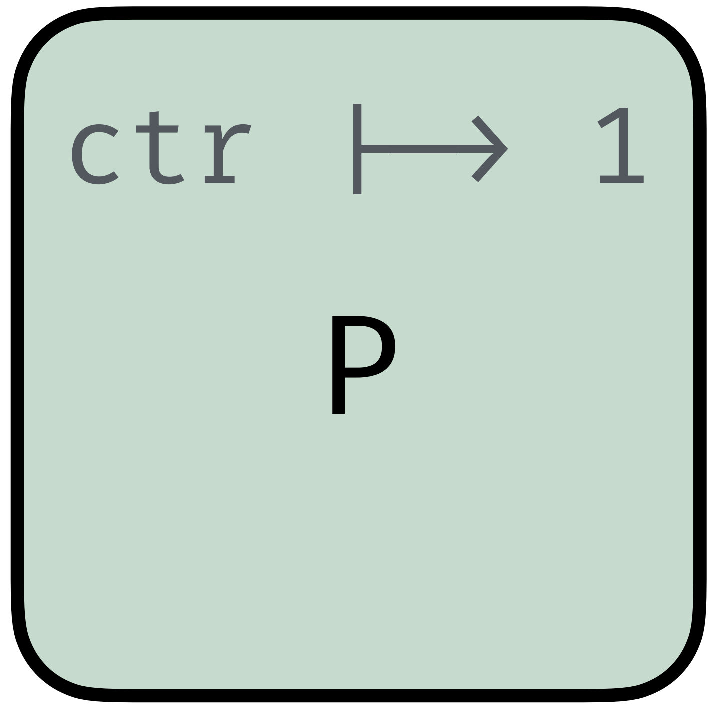
A Simple Back-Translation for RSP

```
Call C P 1;
Call P C 2;
Ret  C P 3;
Ret  P C 4;
```

```
C():
if (ctr = 0) {
  ctr++;
  P(1);
  C();
} else if (ctr = 1) {
  ctr++;
  return 3
}
```

```
P():
if (ctr = 0) {
  ctr++;
  C(2);
```



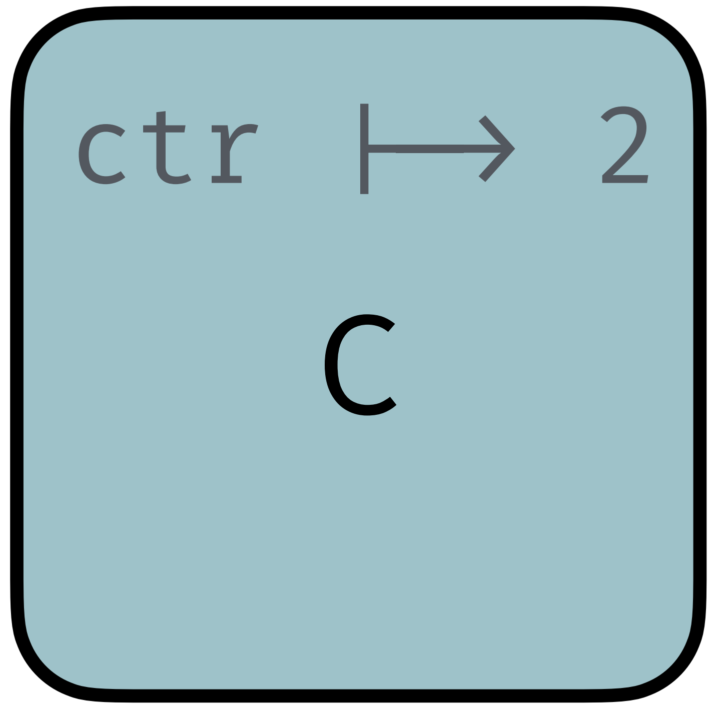


A Simple Back-Translation for RSP

```
Call  C P 1;

Call  P C 2;

Ret   C P 3;

Ret   P C 4;
```

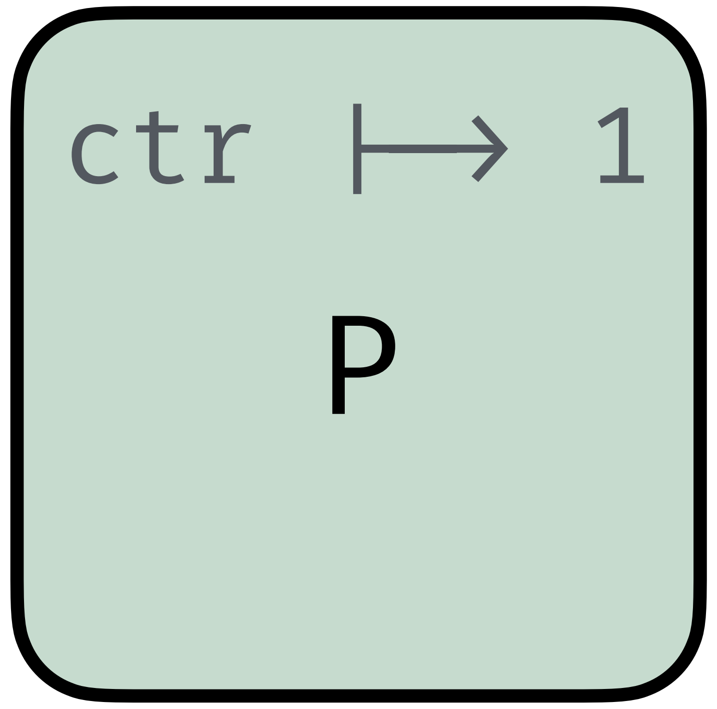
```
C():
if (ctr = 0) {
  ctr++;
  P(1);
  C();
} else if (ctr = 1) {
  ctr++;
  return 3
}
```

```
P():
if (ctr = 0) {
  ctr++;
  C(2);
  P();
```



ctr ↦ 2

C



ctr ↦ 1

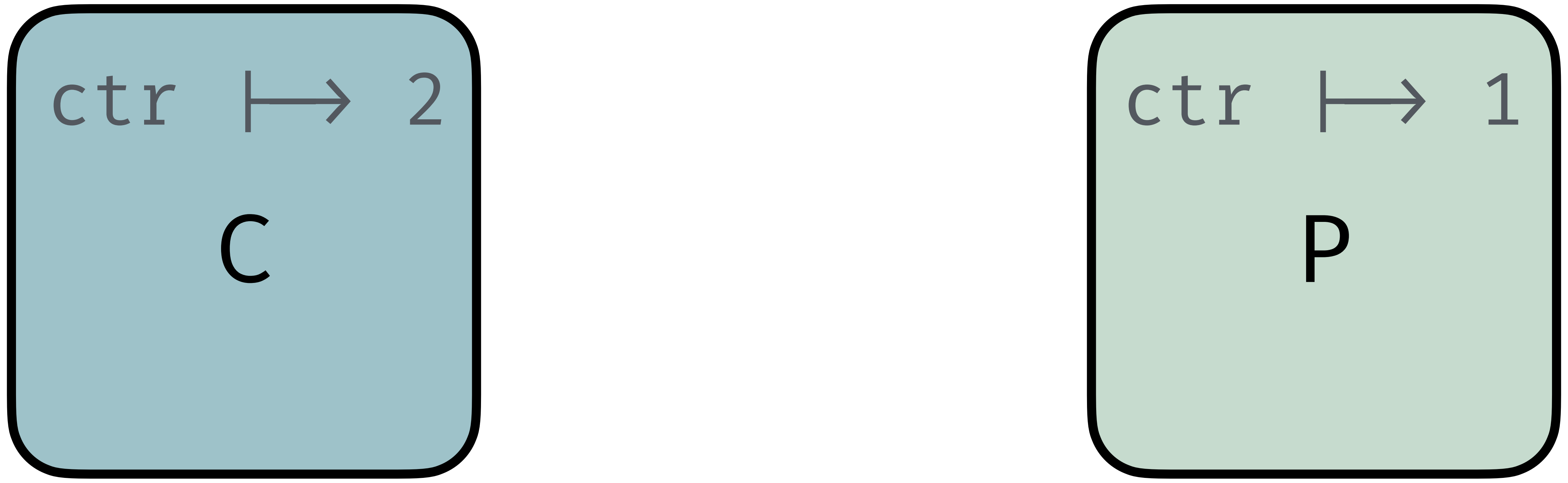
P

A Simple Back-Translation for RSP

```
Call C P 1;
Call P C 2;
Ret  C P 3;
Ret  P C 4;
```

```
C():
if (ctr = 0) {
  ctr++;
  P(1);
  C();
} else if (ctr = 1) {
  ctr++;
  return 3
}
```

```
P():
if (ctr = 0) {
  ctr++;
  C(2);
  P();
} else if (ctr = 1) {
```



ctr ⟼ 2

C

ctr ⟼ 1

P

A Simple Back-Translation for RSP

```
Call C P 1;
Call P C 2;
Ret  C P 3;
Ret  P C 4;
```

```
C():
if (ctr = 0) {
  ctr++;
  P(1);
  C();
} else if (ctr = 1) {
  ctr++;
  return 3
}
```

```
P():
if (ctr = 0) {
  ctr++;
  C(2);
  P();
} else if (ctr = 1) {
  ctr++;
```

ctr ⟼ 2

C

ctr ⟼ 2

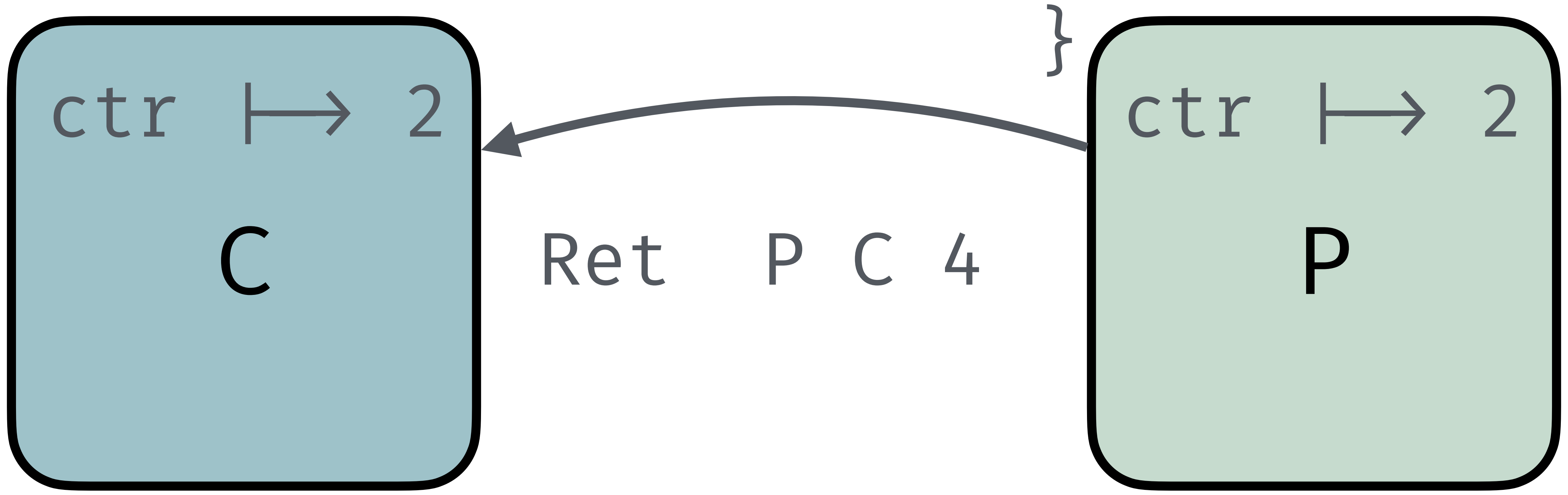
P

A Simple Back-Translation for RSP

```
Call C P 1;
Call P C 2;
Ret  C P 3;
Ret  P C 4;
```

```
C():
if (ctr = 0) {
  ctr++;
  P(1);
  C();
} else if (ctr = 1) {
  ctr++;
  return 3
}
```

```
P():
if (ctr = 0) {
  ctr++;
  C(2);
  P();
} else if (ctr = 1) {
  ctr++;
  return 4
}
```



Implementing and verifying this back-translation

(Abate et al. (2018), When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise)

Implementing and verifying this back-translation

Decently easy:

(Abate et al. (2018), When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise)

Implementing and verifying this back-translation

Decently easy:

- Manipulates one finite object (a trace)

(Abate et al. (2018), When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise)

Implementing and verifying this back-translation

Decently easy:

- Manipulates one finite object (a trace)
- Simple logic: just emit the events one by one

(Abate et al. (2018), When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise)

Implementing and verifying this back-translation

Decently easy:

- Manipulates one finite object (a trace)
- Simple logic: just emit the events one by one
- Proof by induction on the trace

(Abate et al. (2018), When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise)

Implementing and verifying this back-translation

Decently easy:

- Manipulates one finite object (a trace)
- Simple logic: just emit the events one by one
- Proof by induction on the trace
- Less than 600 LoC (including comments)

(Abate et al. (2018), When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise)

Can we get a stronger result?

Can we get a stronger result?

- So far: safety properties only
- Some security properties are more than safety: hypersafety (noninterference), relational hypersafety (observational equivalence)
- Can we adapt the proof technique to obtain a stronger criterion than RSP?

Can we get a stronger result?

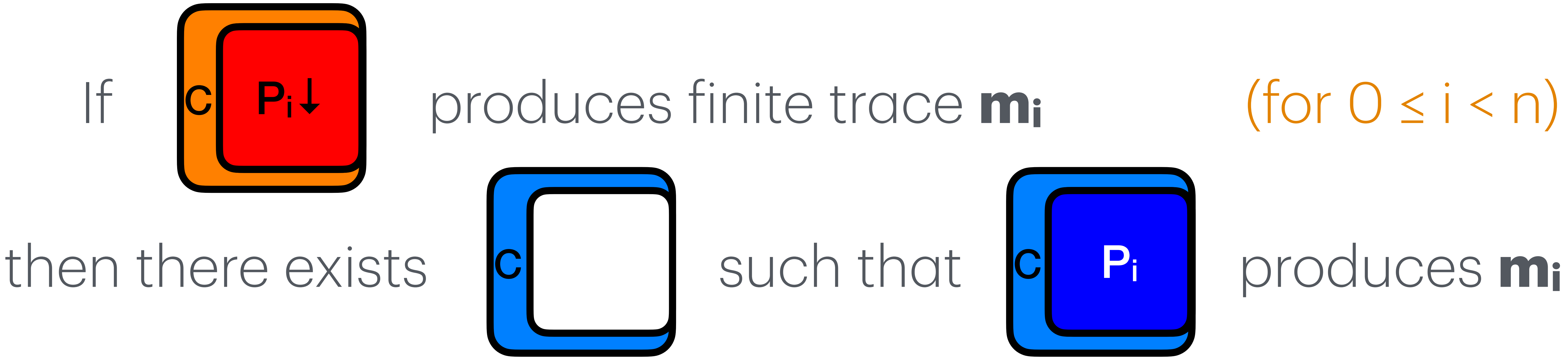
- So far: safety properties only
- Some security properties are more than safety: hypersafety (noninterference), relational hypersafety (observational equivalence)
- Can we adapt the proof technique to obtain a stronger criterion than RSP?

Yes!

Can we get a stronger result?

- So far: safety properties only
- Some security properties are more than safety: hypersafety (noninterference), relational hypersafety (observational equivalence)
- Can we adapt the proof technique to obtain a stronger criterion than RSP?

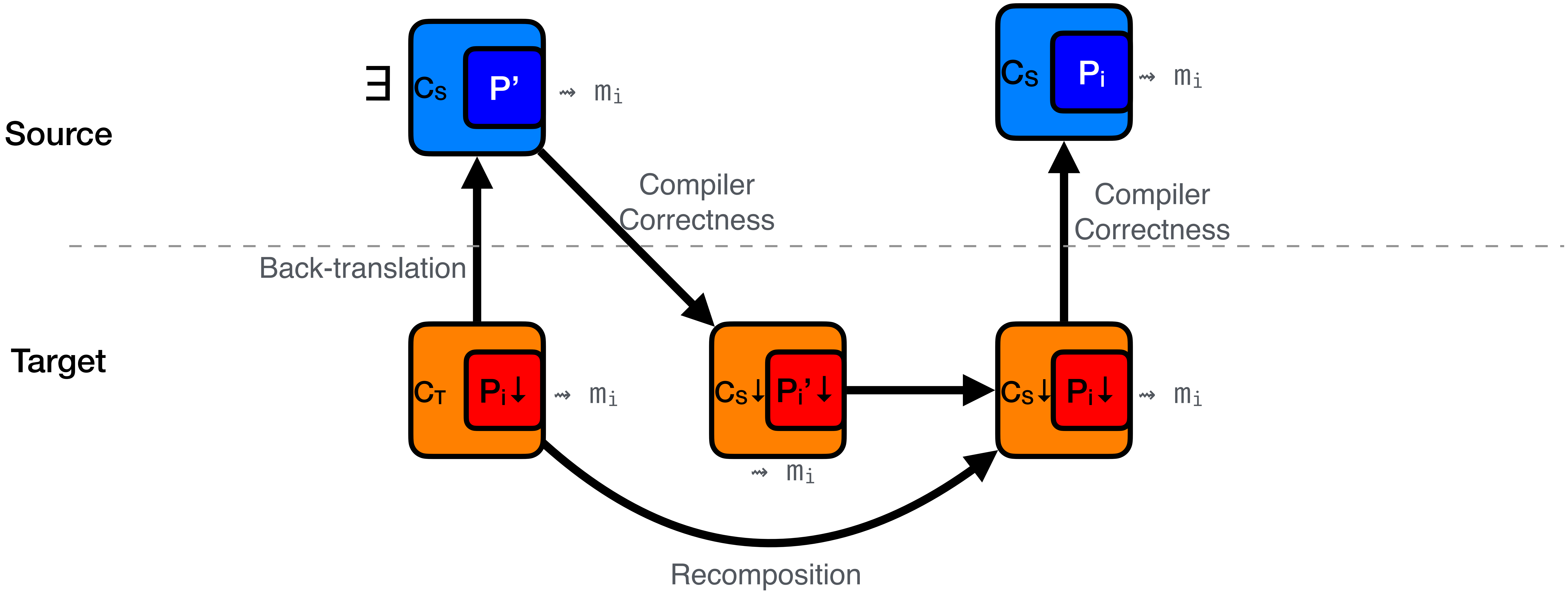
Yes!



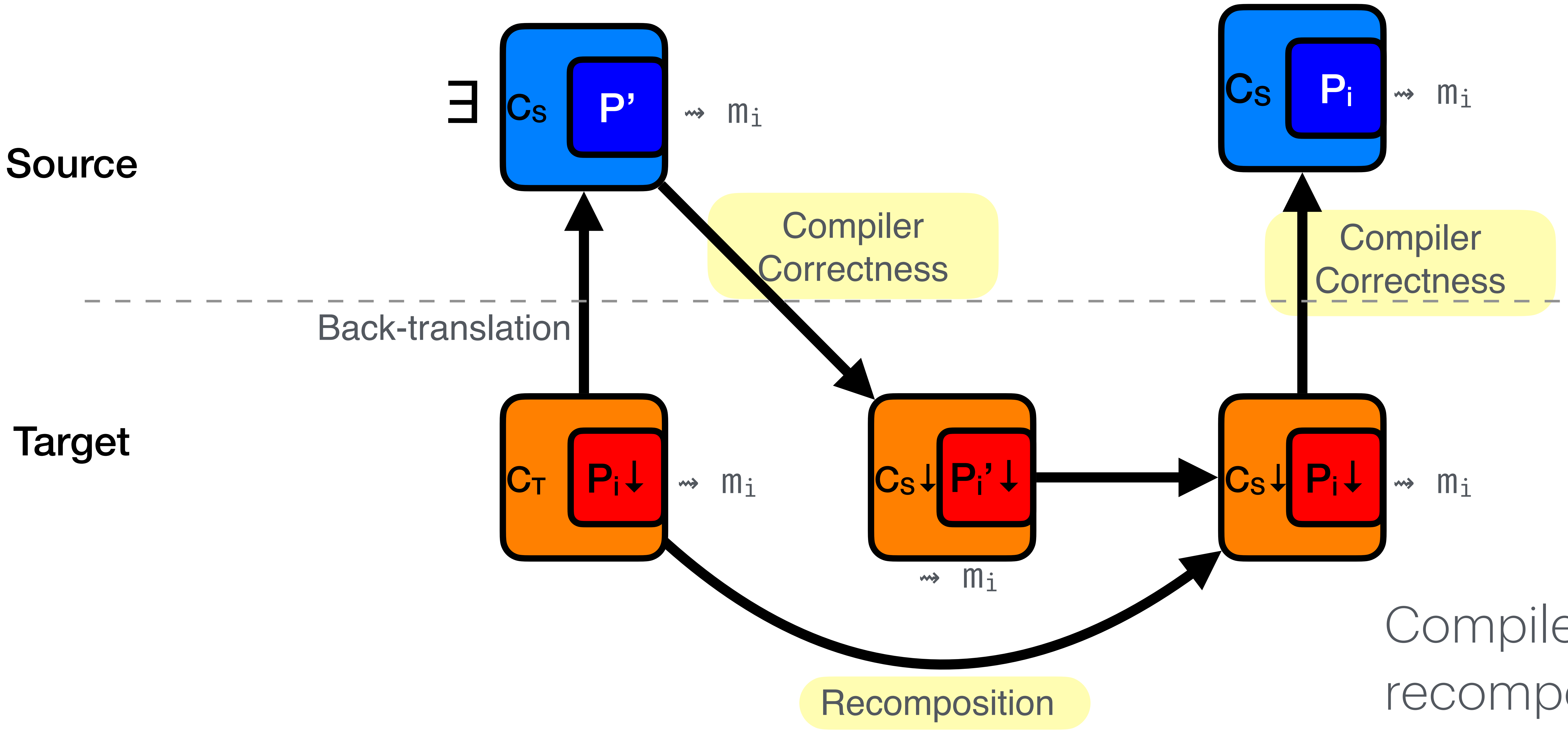
If produces finite trace $\mathbf{m_i}$ (for $0 \leq i < n$)

then there exists such that produces $\mathbf{m_i}$

Adapting the proof for multiple programs

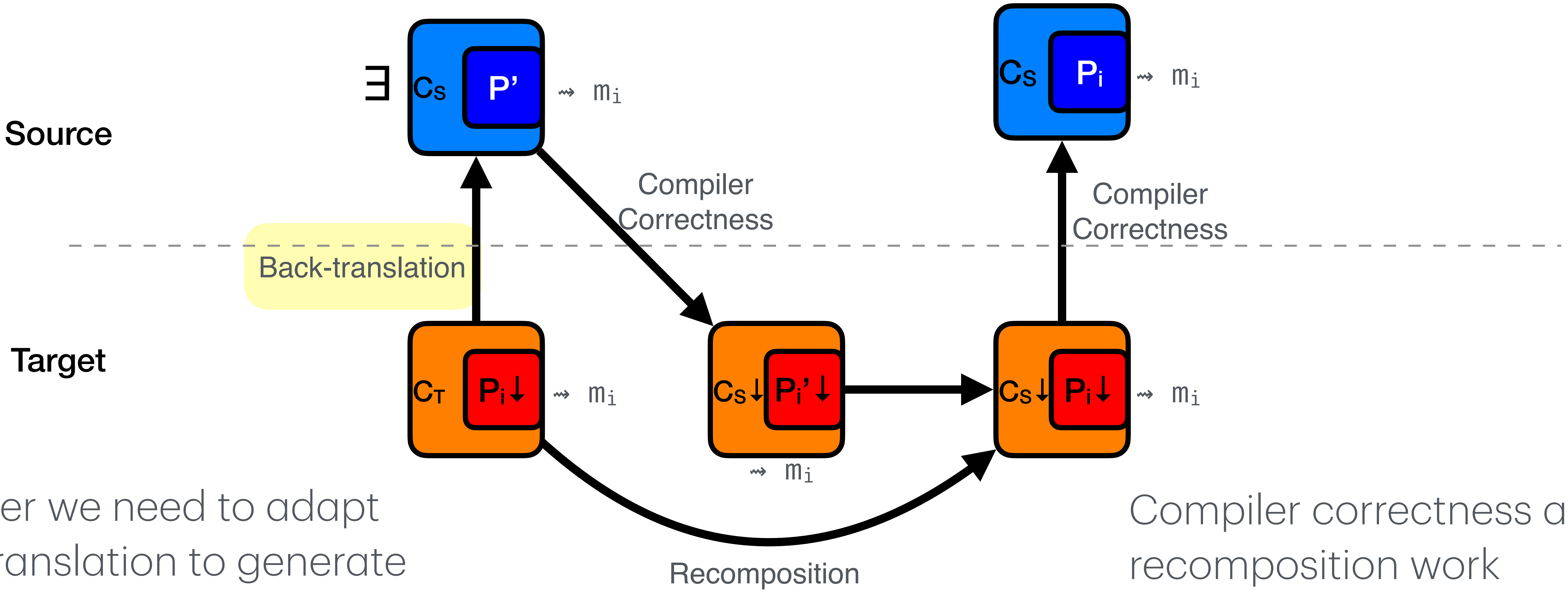


Adapting the proof for multiple programs



Compiler correctness and recomposition work pointwise, so we don't need to modify them

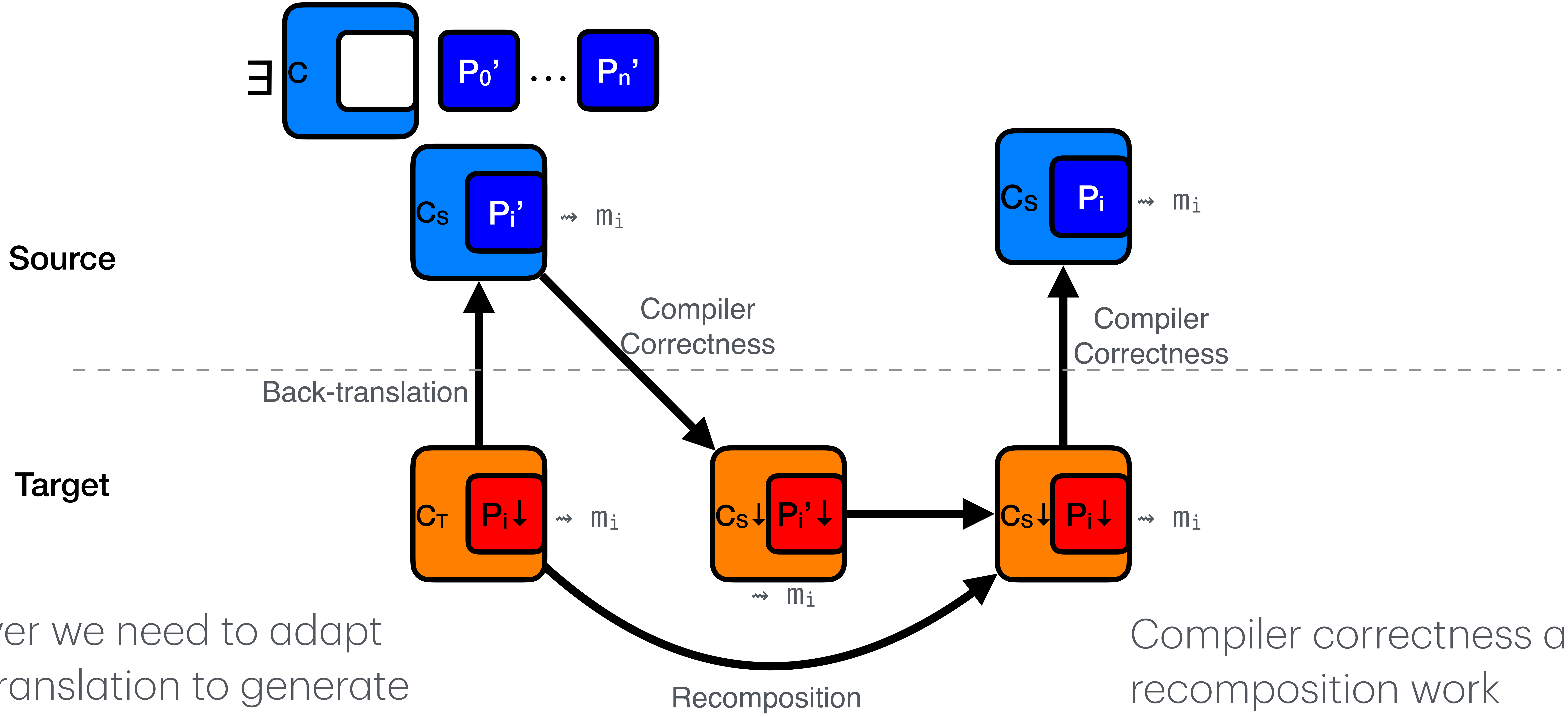
Adapting the proof for multiple programs



However we need to adapt back-translation to generate one context and several programs

Compiler correctness and recomposition work pointwise, so we don't need to modify them

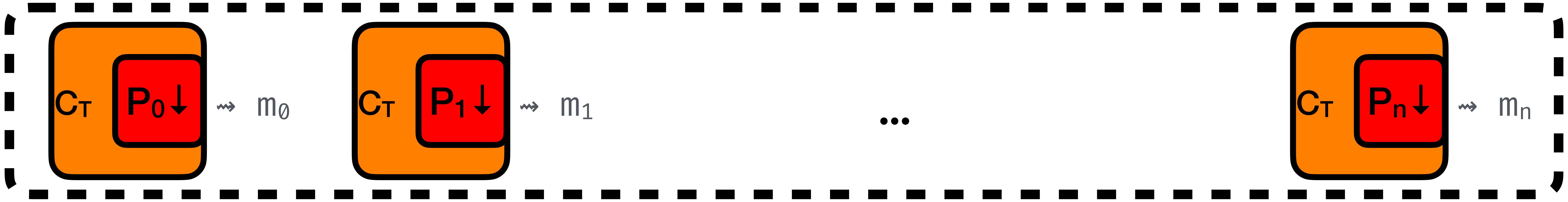
Adapting the proof for multiple programs



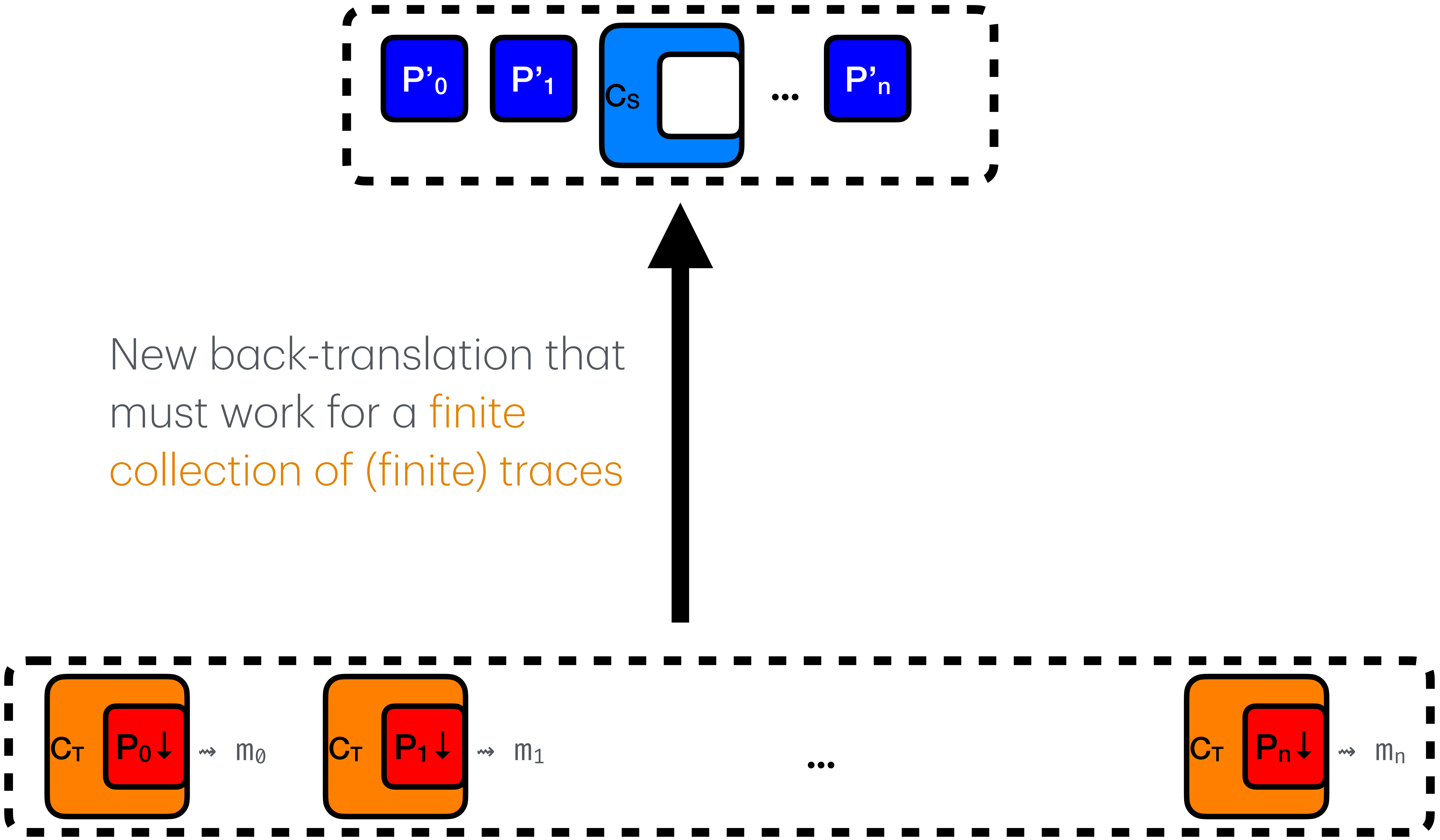
However we need to adapt back-translation to generate one context and several programs

Compiler correctness and recomposition work pointwise, so we don't need to modify them

Back-translation for several traces

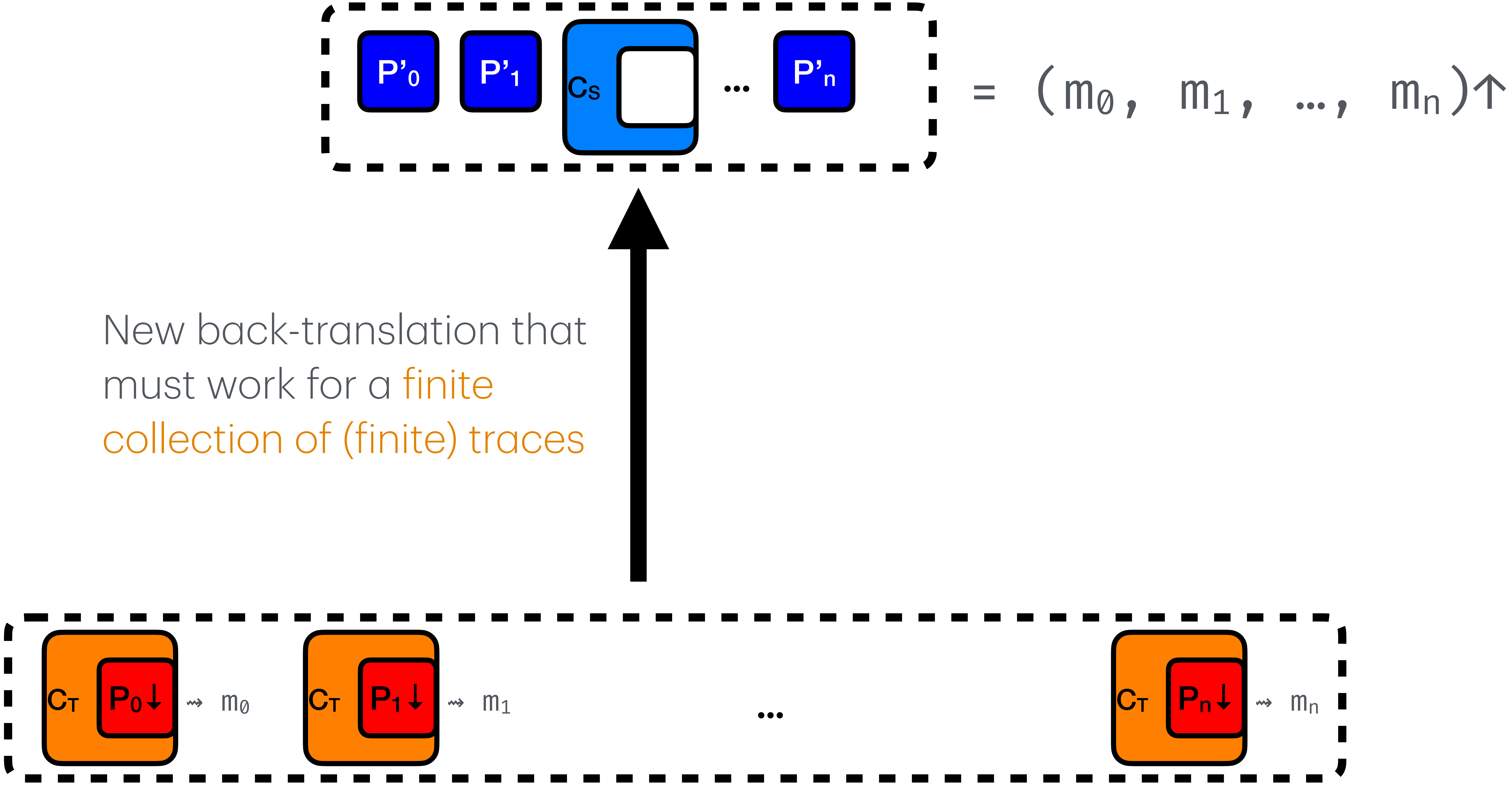


Back-translation for several traces



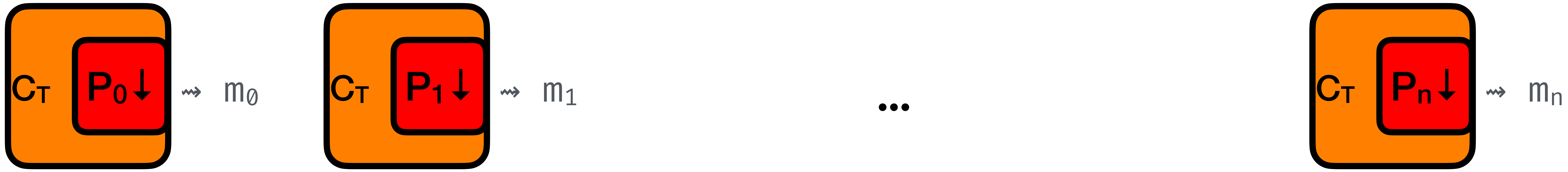
New back-translation that must work for a finite collection of (finite) traces

Back-translation for several traces



New back-translation that must work for a finite collection of (finite) traces

Structure of the traces



C_T $P_0\downarrow$ ⇝ m_0

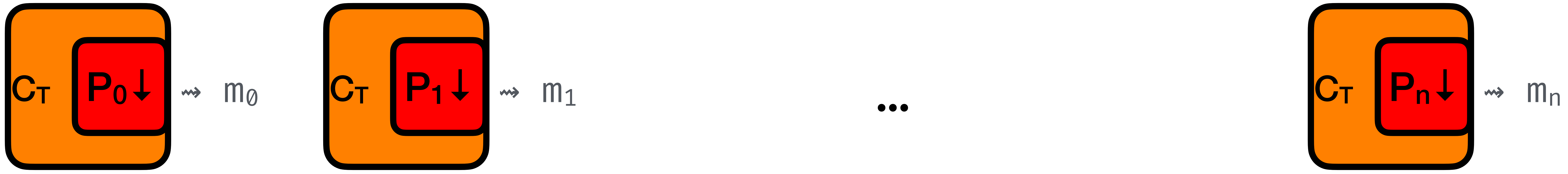
C_T $P_1\downarrow$ ⇝ m_1

...

C_T $P_n\downarrow$ ⇝ m_n

Structure of the traces

Traces are produced by the same context but different programs

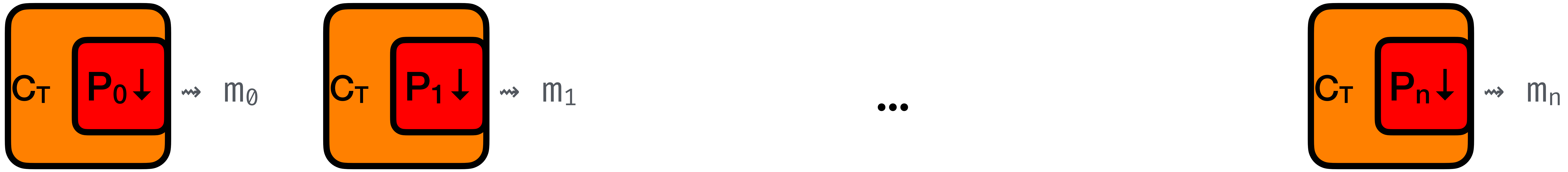


$C_T\ P_0{\downarrow} \rightsquigarrow m_0$ $C_T\ P_1{\downarrow} \rightsquigarrow m_1$ ... $C_T\ P_n{\downarrow} \rightsquigarrow m_n$

Structure of the traces

Traces are produced by the same context but different programs

Determinacy property: traces can only differ because the programs did something different

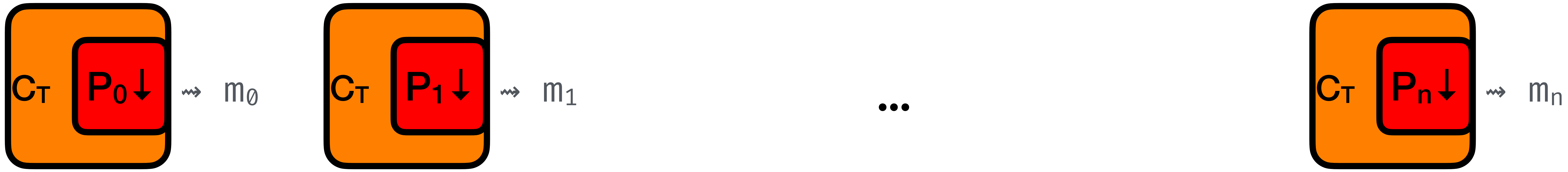


Structure of the traces

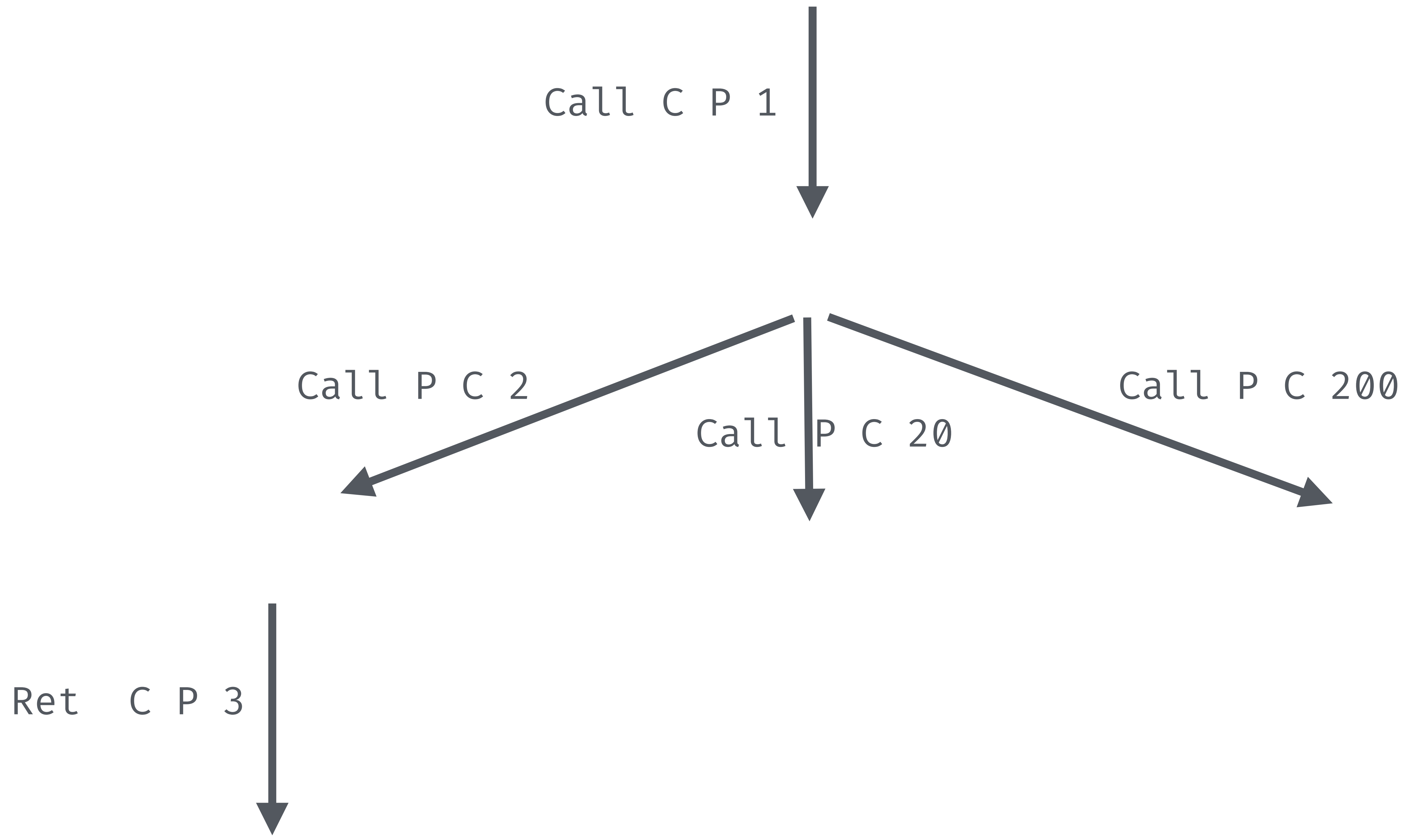
Traces are produced by the same context but different programs

Traces $m_0, m_1, \ldots, m_n$ can be represented by a tree T that branches on events from P

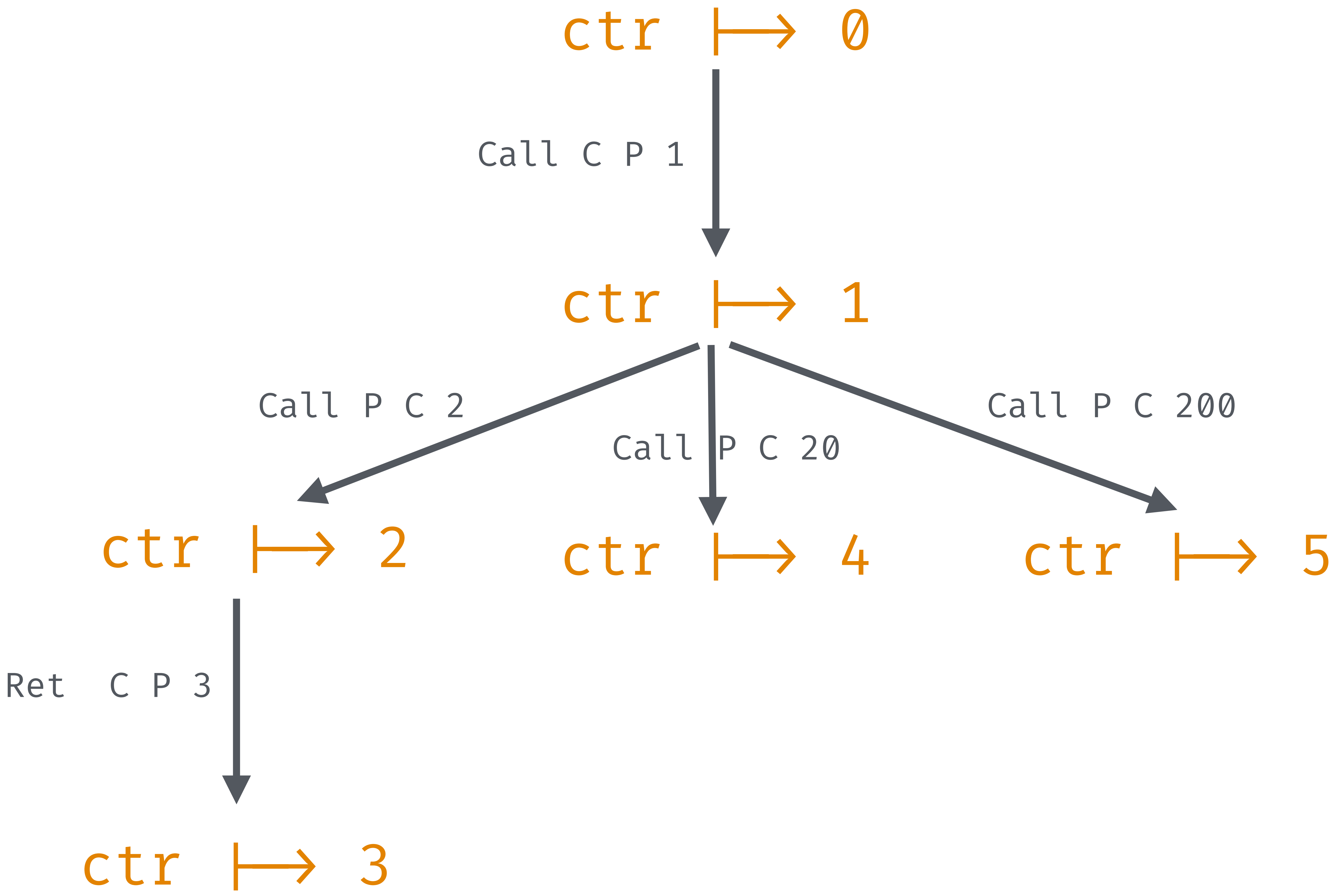
Determinacy property: traces can only differ because the programs did something different


C_T $P_0\downarrow$ $\rightsquigarrow m_0$
C_T $P_1\downarrow$ $\rightsquigarrow m_1$
…
C_T $P_n\downarrow$ $\rightsquigarrow m_n$

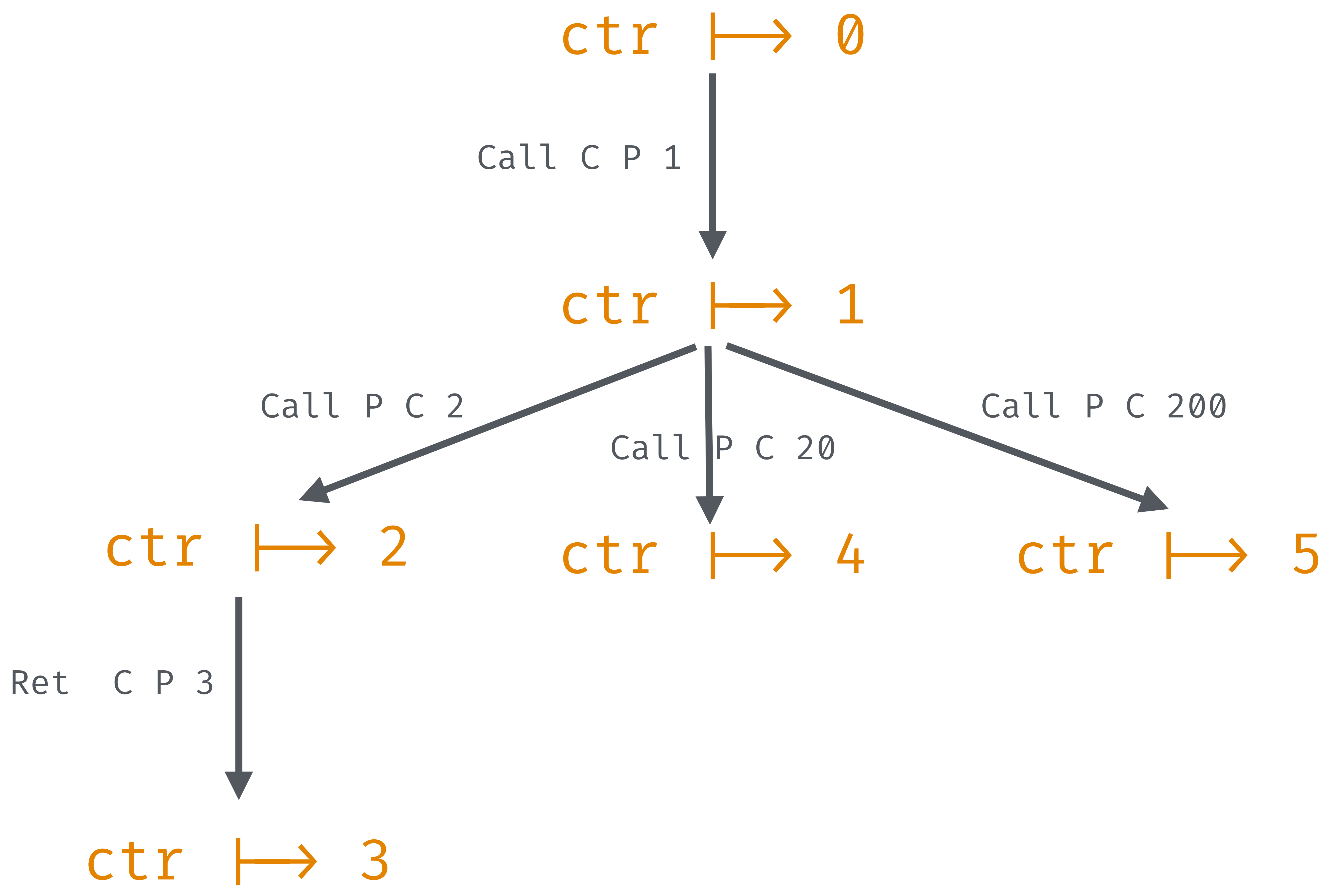
Back-translating trees is harder



Back-translating trees is harder



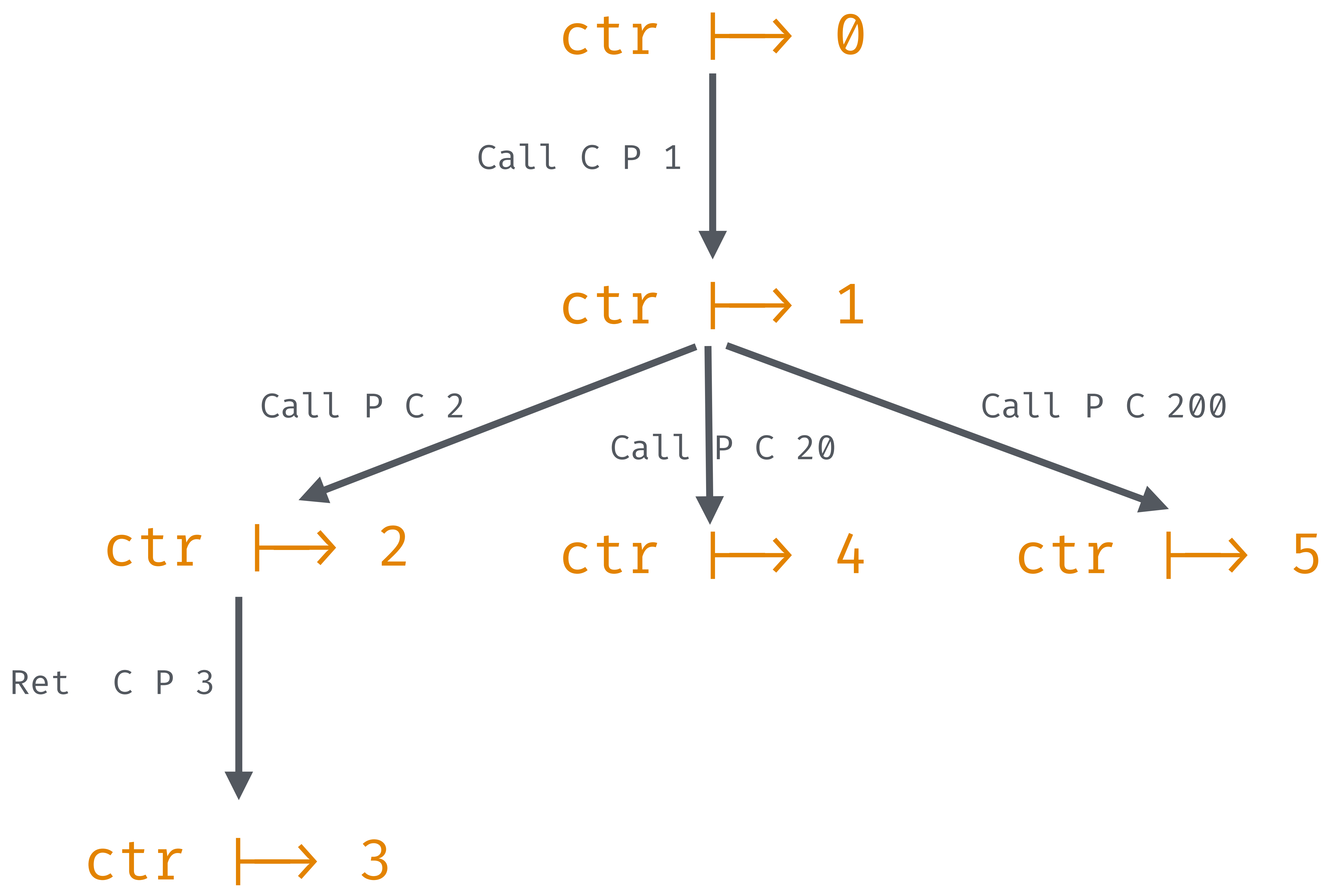
Back-translating trees is harder



To back-translate:

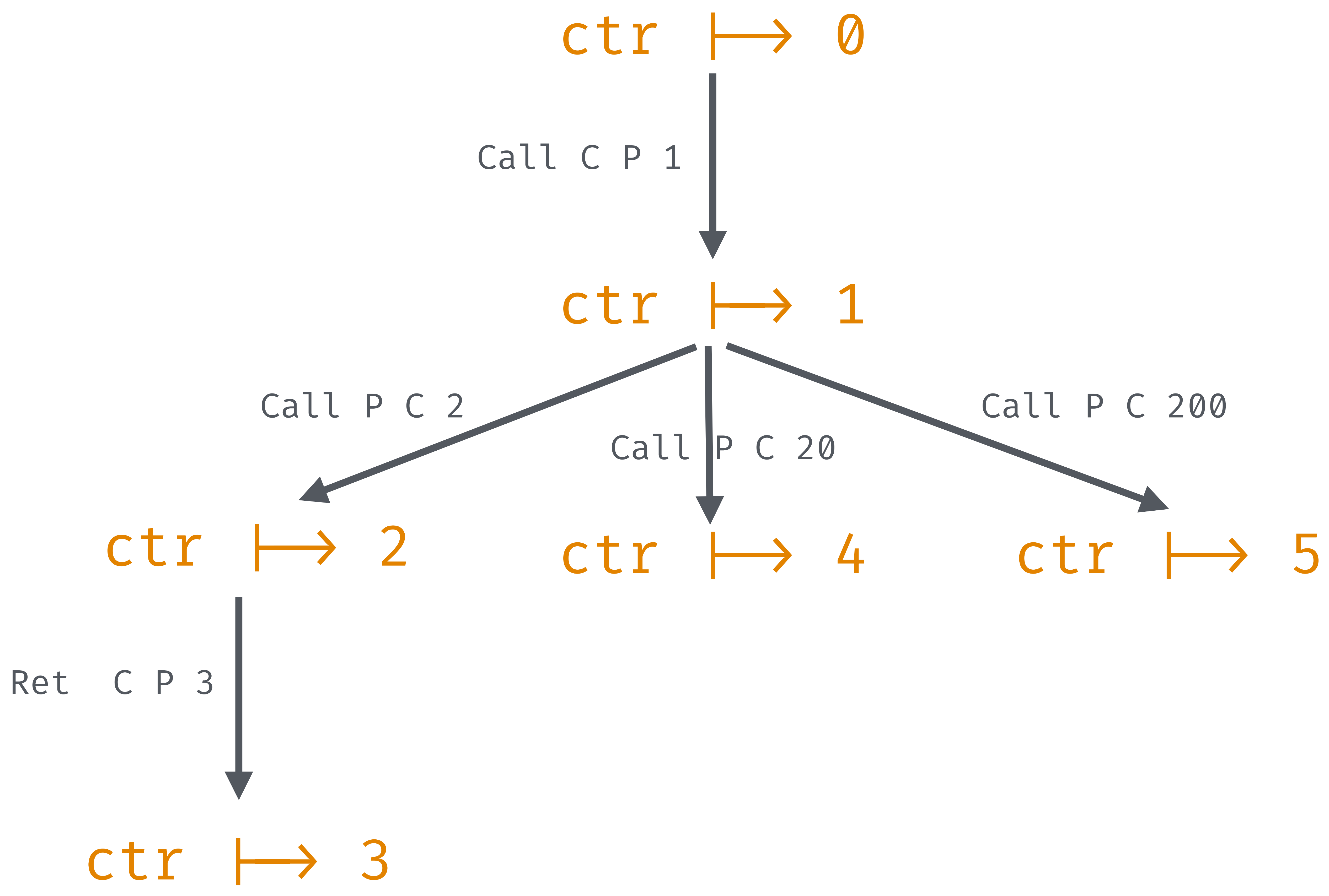
- Generalize the counter to record position in the tree
- Context needs to look at argument and return value before updating its local counter
- Also need to look at current local counter

Back-translating trees is harder



```
if (ctr = K && call_arg = V) {
  ctr := NEXT_CTR(K, V);
  DO_EVENT(K, V);
  C();
}
…
```

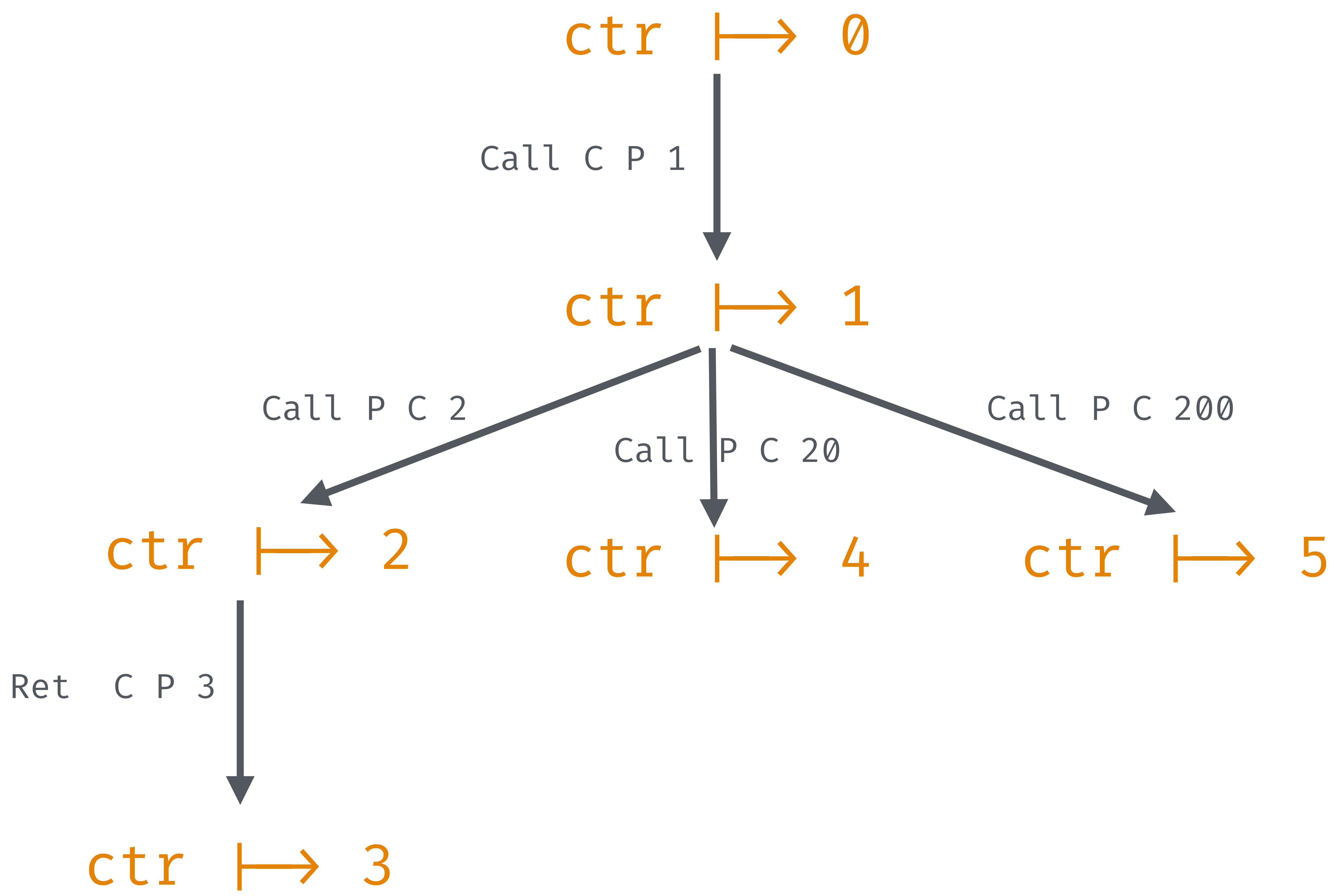
Back-translating trees is harder



```
if (ctr = K && call_arg = V) {
  ctr := NEXT_CTR(K, V);
  DO_EVENT(K, V);
  C();
}
…
```

Back-translation function is not trivial!

Proving the back-translation is also harder



ctr ⟼ 0

Call C P 1

ctr ⟼ 1

Call P C 2

Call P C 20

Call P C 200

ctr ⟼ 2

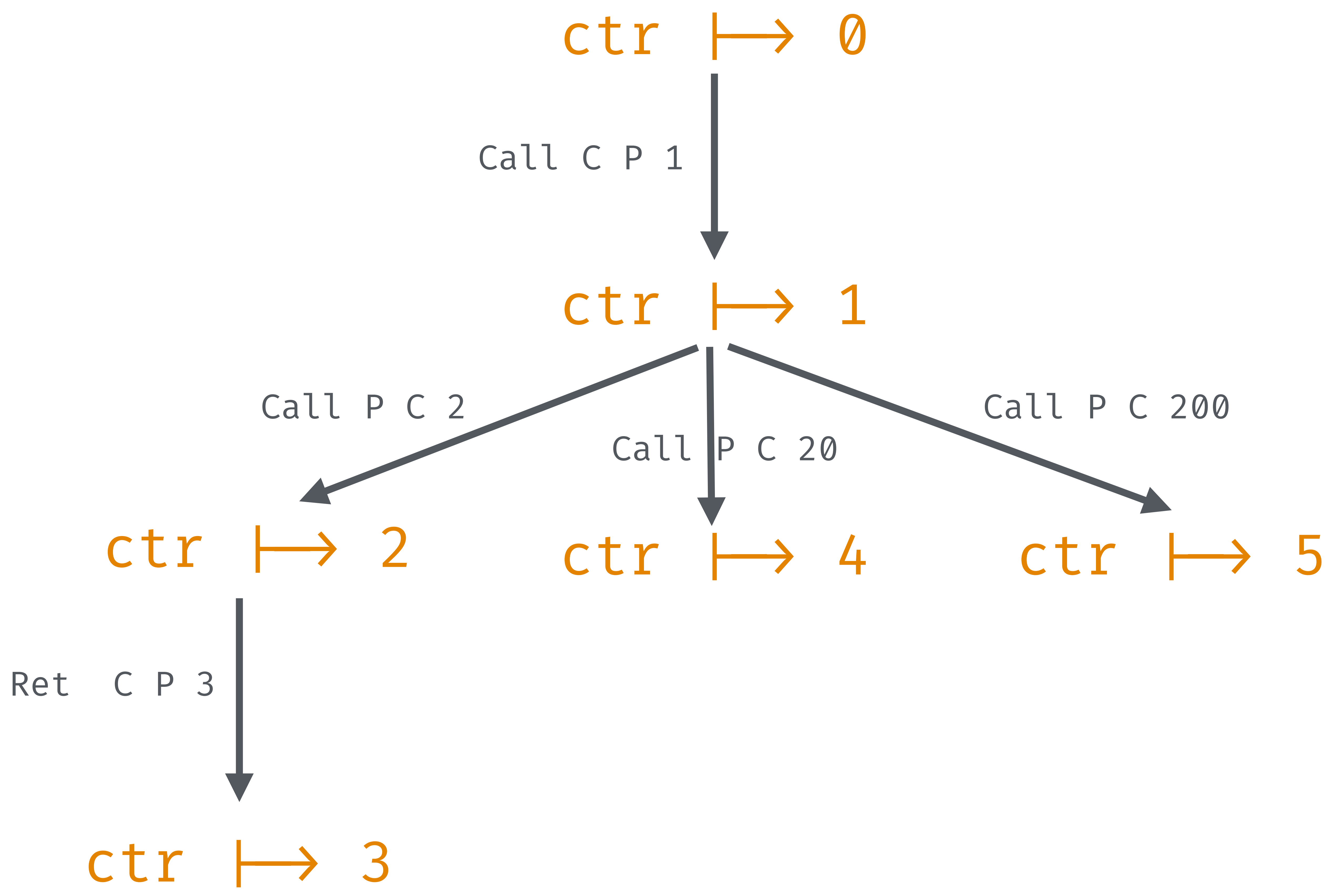
ctr ⟼ 4

ctr ⟼ 5

Ret C P 3

ctr ⟼ 3

Proving the back-translation is also harder



ctr ↦ 0

Call C P 1

ctr ↦ 1

Call P C 2

Call P C 20

Call P C 200

ctr ↦ 2

ctr ↦ 4

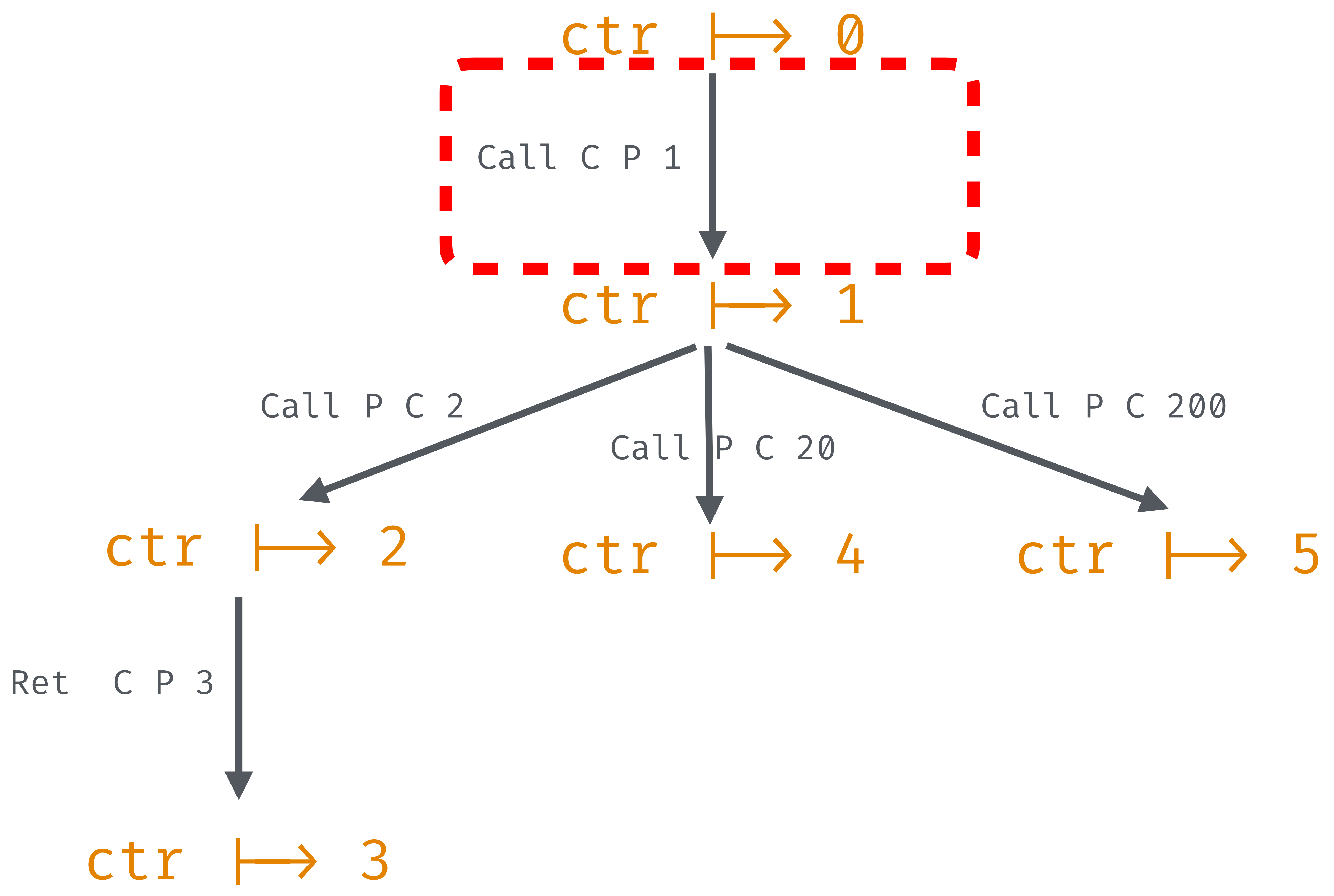
ctr ↦ 5

Ret C P 3

ctr ↦ 3

Non-trivial invariants!

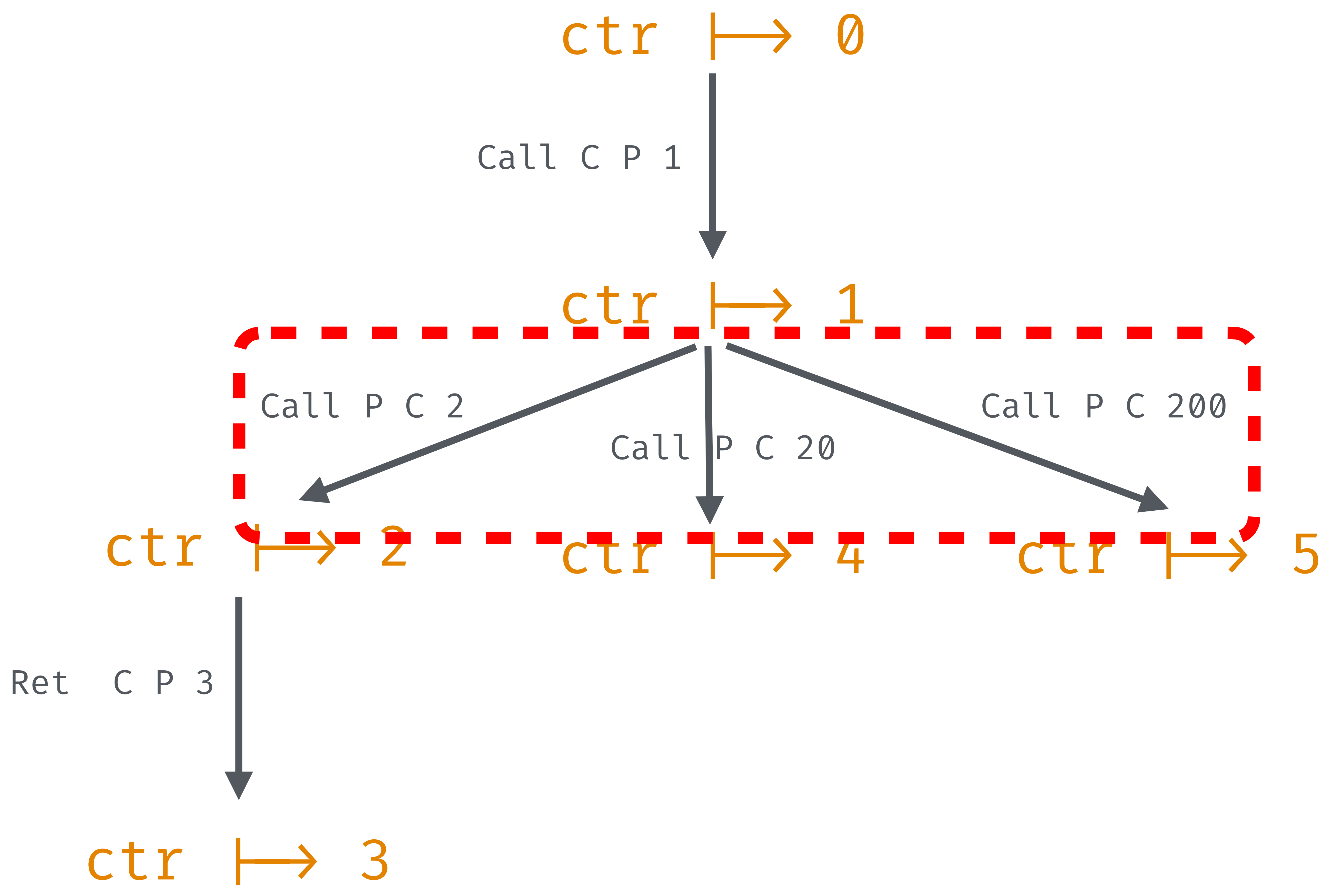
Proving the back-translation is also harder



Non-trivial invariants!

- Context can only do one event

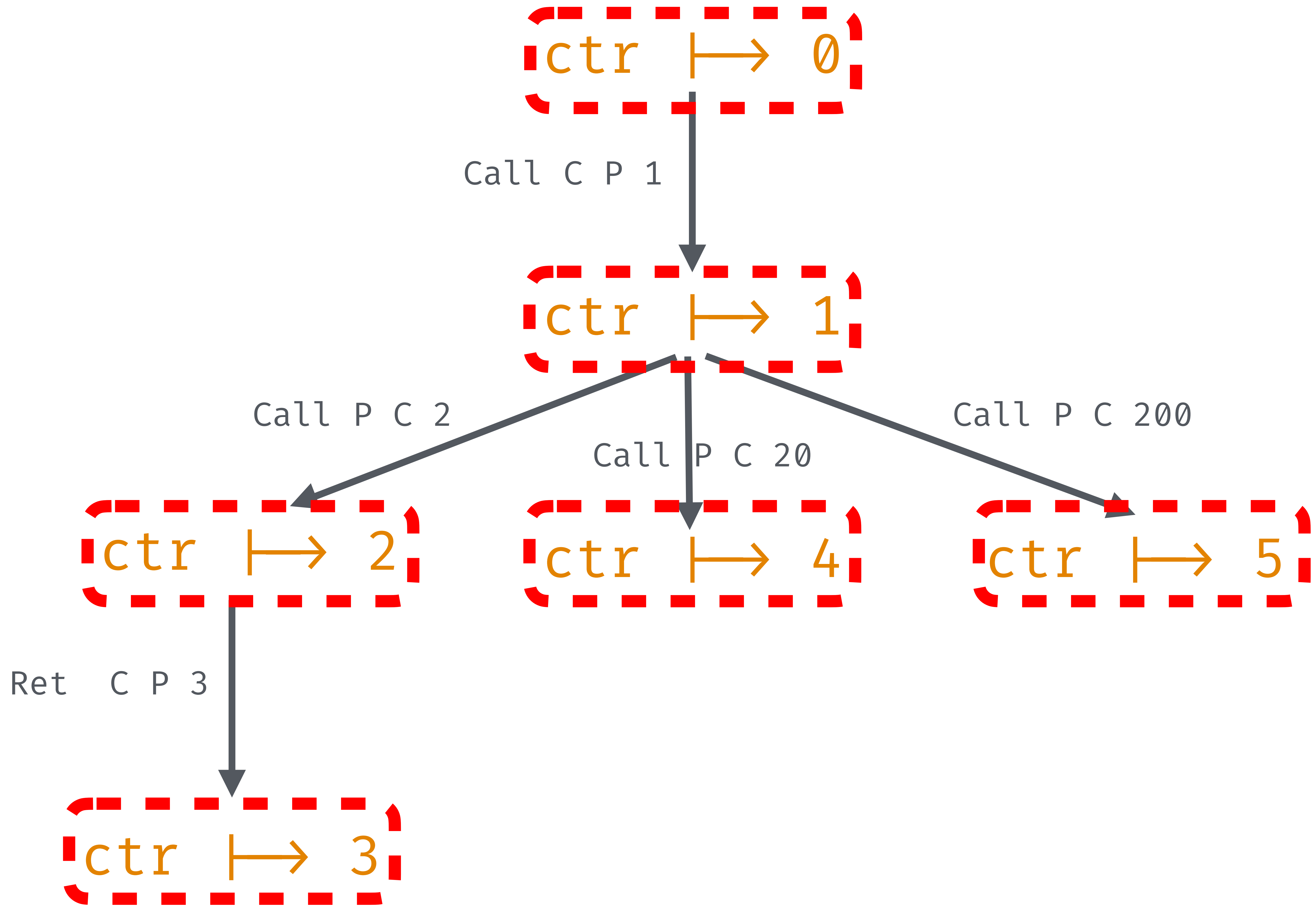
Proving the back-translation is also harder



Non-trivial invariants!

- Context can only do one event
- No duplicate events

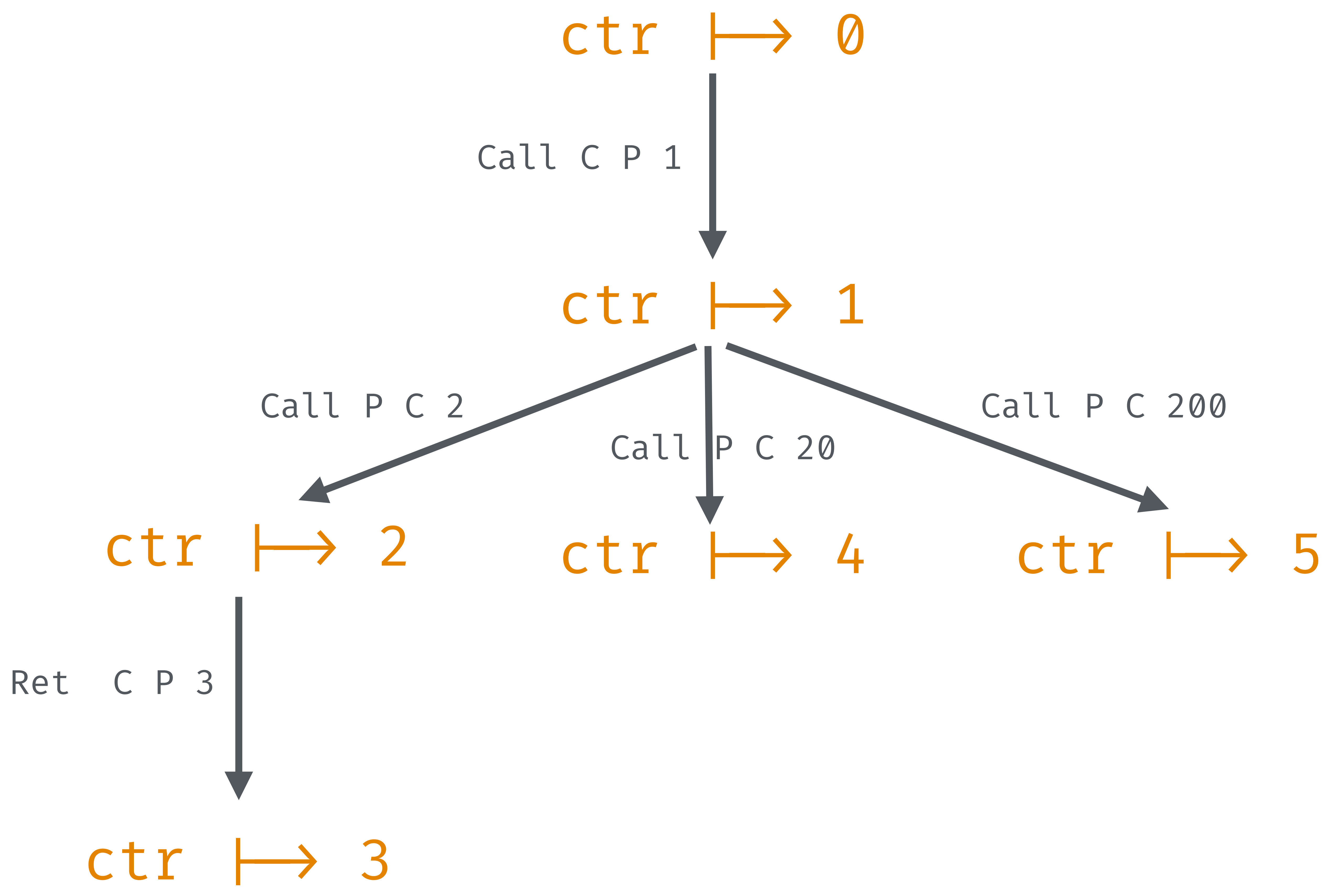
Proving the back-translation is also harder



Non-trivial invariants!

- Context can only do one event
- No duplicate events
- Unicity of `ctr`

Proving the back-translation is also harder

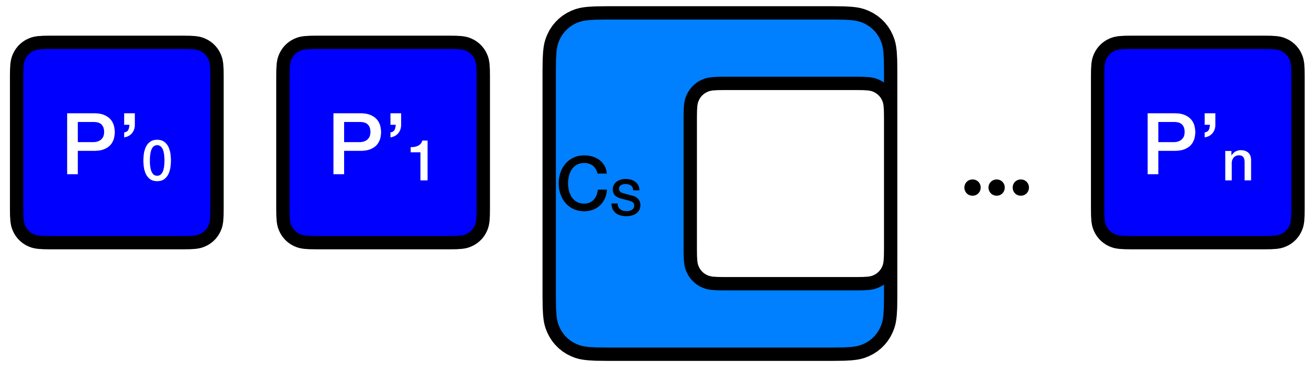


Non-trivial invariants!

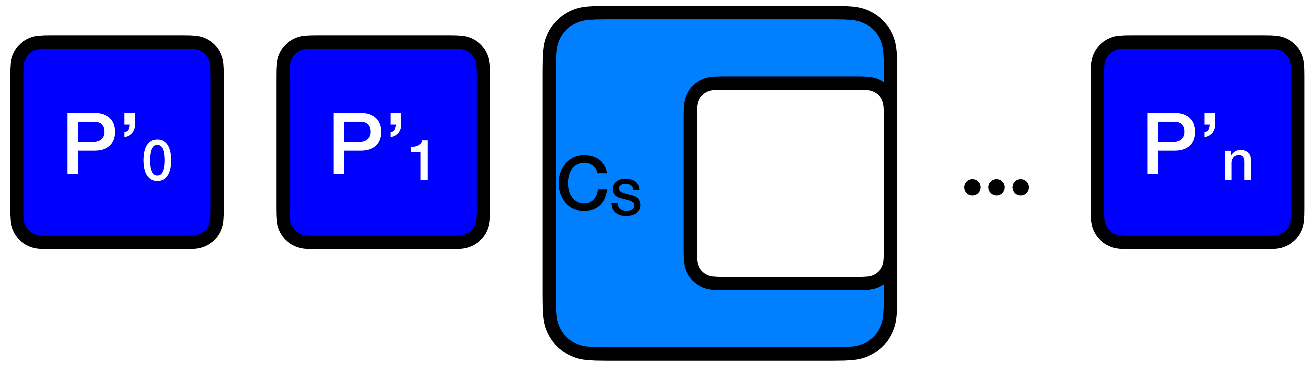
- Context can only do one event
- No duplicate events
- Unicity of `ctr`
- To handle returns: well-bracketedness

Back-translation in several steps

Back-translation in several steps

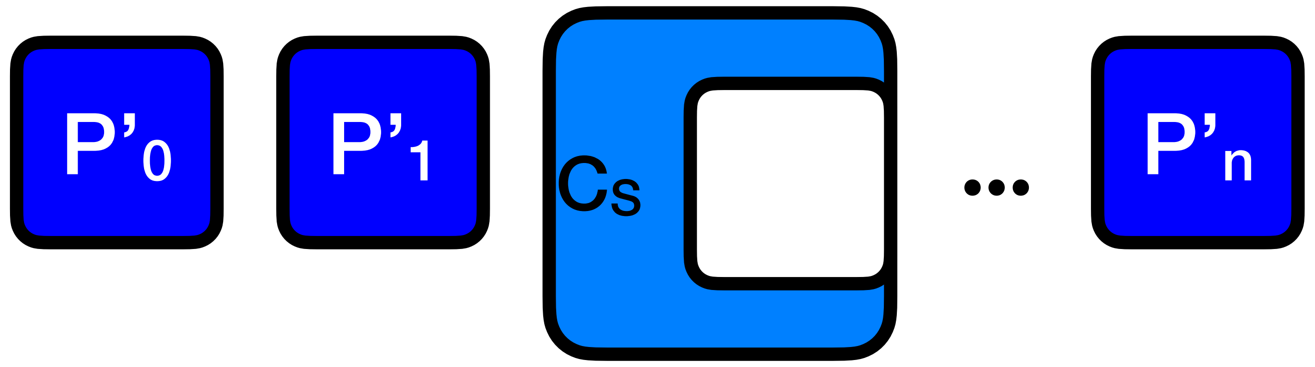


Back-translation in several steps



Tree T

Back-translation in several steps

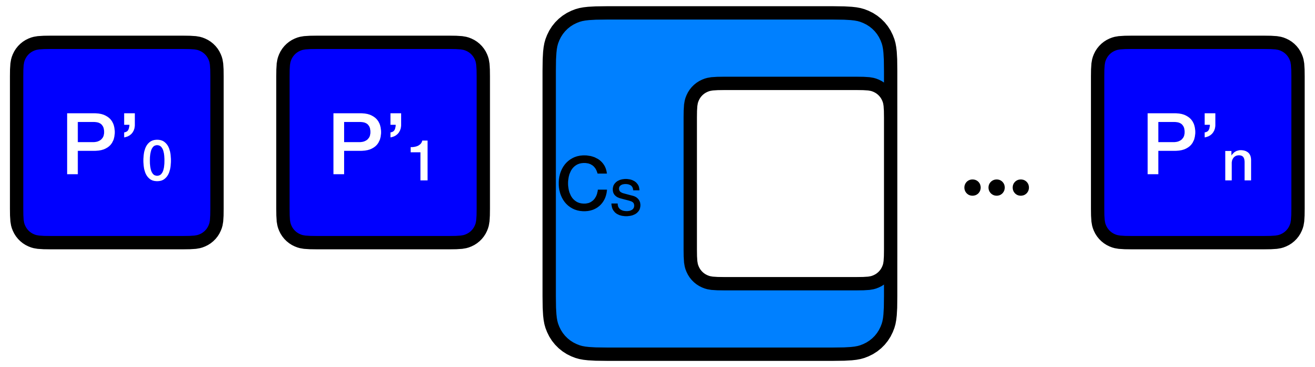


Tree T’ with ctr

↑

Tree T

Back-translation in several steps



Tree T' with stack record

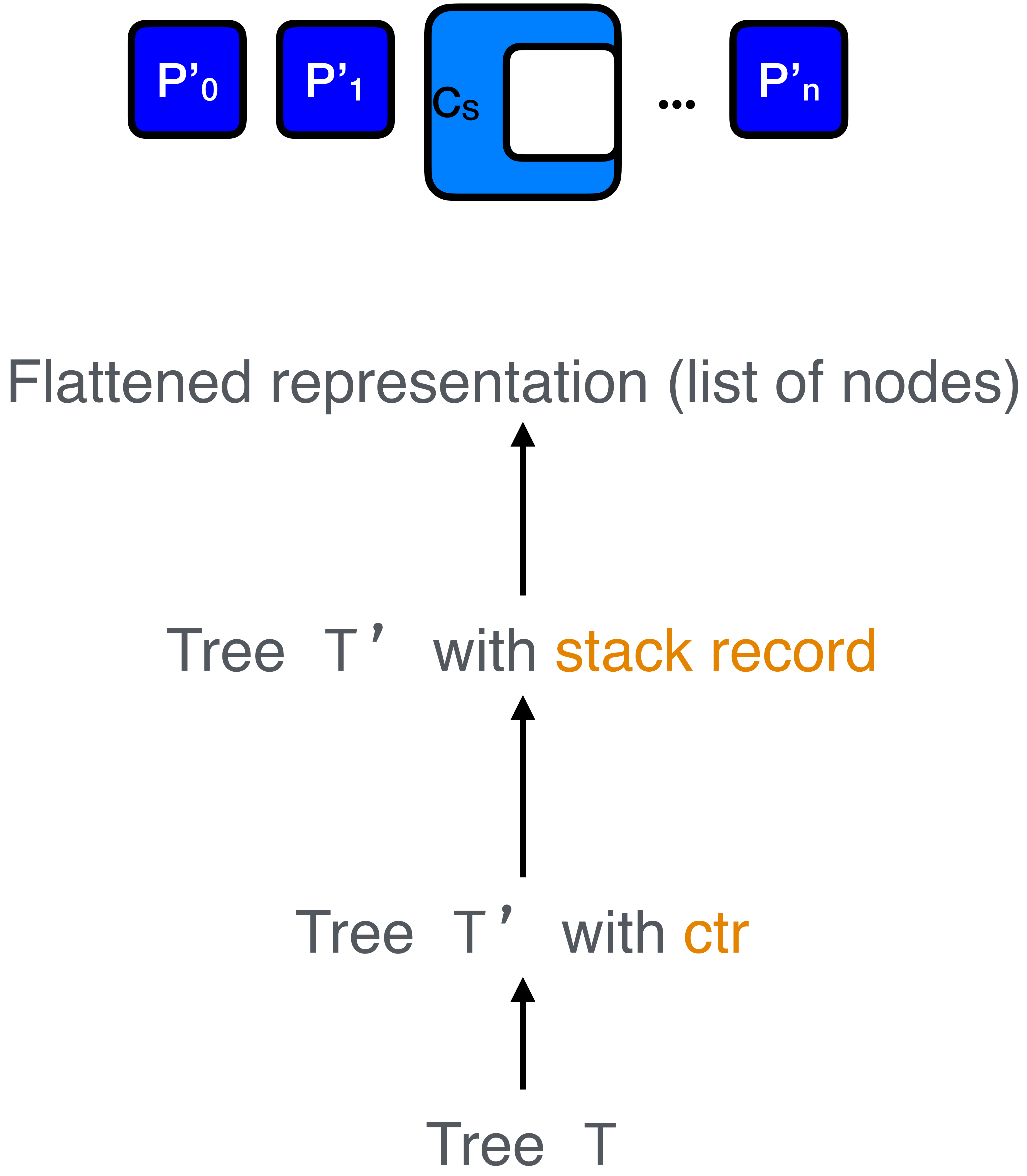
↑

Tree T' with ctr

↑

Tree T

Back-translation in several steps



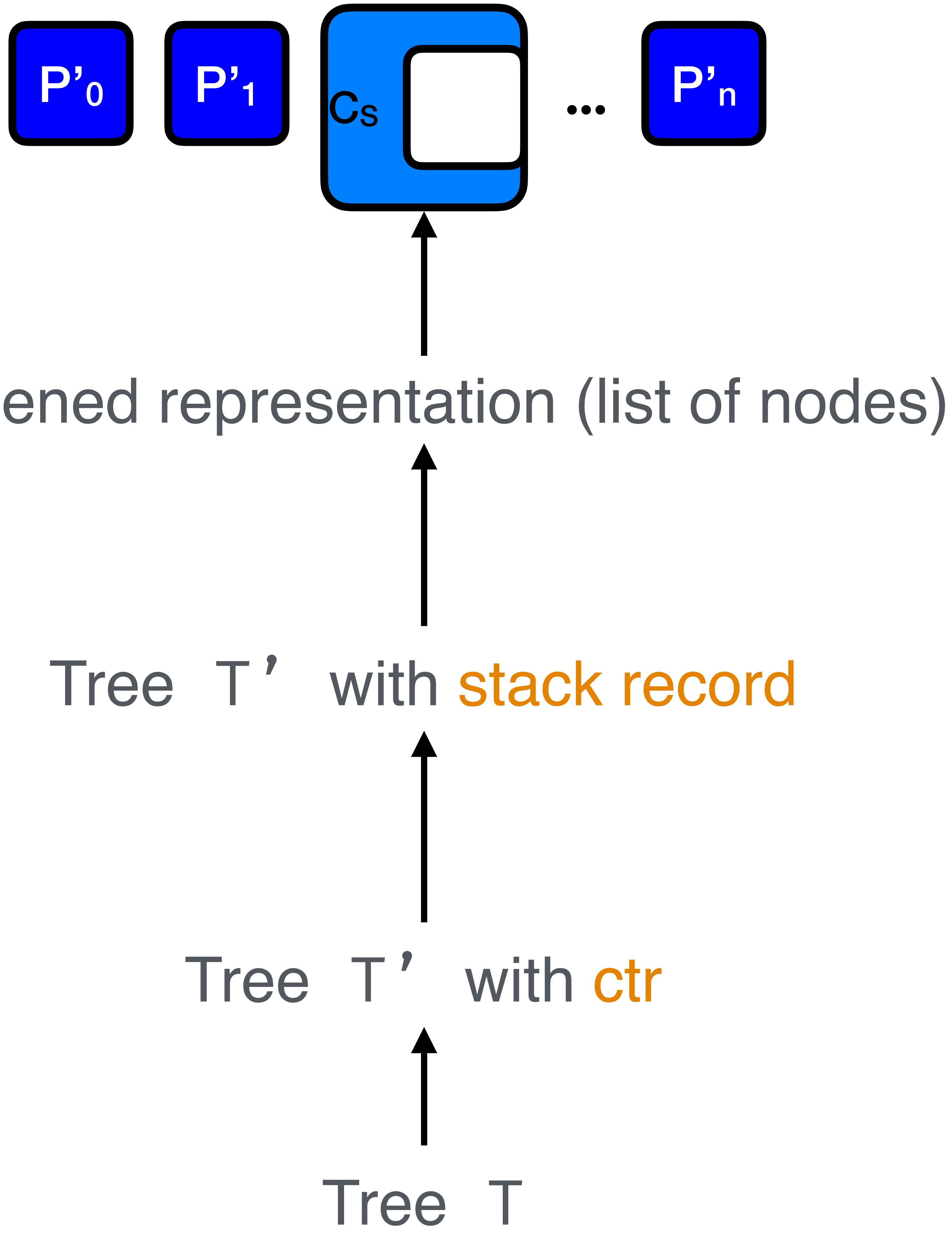
Flattened representation (list of nodes)

Tree T' with stack record

Tree T' with ctr

Tree T

Back-translation in several steps



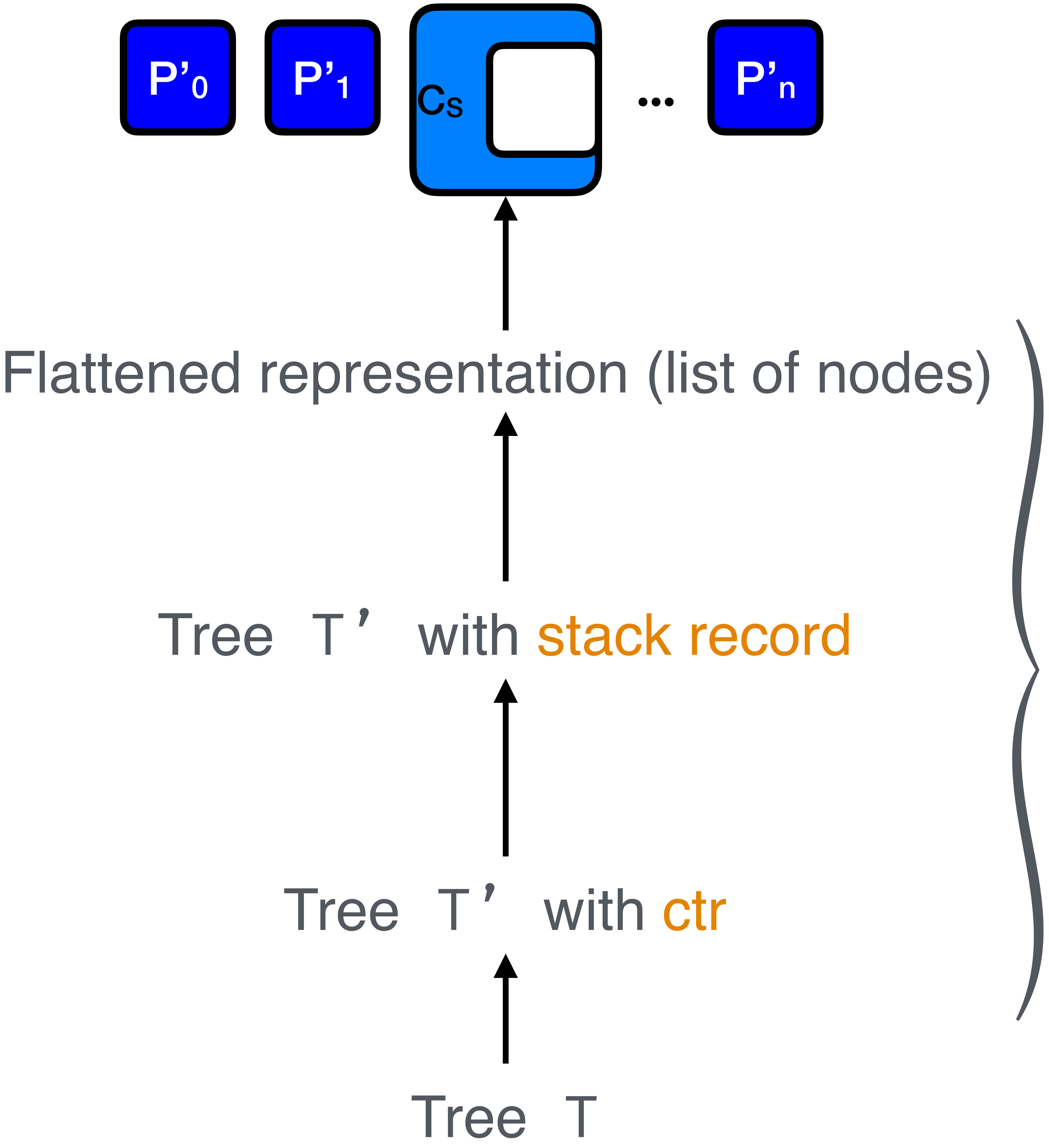
Flattened representation (list of nodes)

Tree T' with stack record

Tree T' with ctr

Tree T

Back-translation in several steps



Flattened representation (list of nodes)

Tree T' with stack record

Tree T' with ctr

Tree T

Passes as small as possible to simplify implementation and reasoning

Proving back-translation (2)

Proving back-translation (2)

- All intermediate languages equipped with small-step semantics

Proving back-translation (2)

- All intermediate languages equipped with small-step semantics
- Execution guided by trace: steps reduce the tree by emitting one event

$\mathtt{s} \rightarrow_{\mathtt{e}} \mathtt{s'}$ with trees part of $\mathtt{s}$ and $\mathtt{s'}$

Proving back-translation (2)

- All intermediate languages equipped with small-step semantics
- Execution guided by trace: steps reduce the tree by emitting one event

  $\mathtt{s} \rightarrow_{\mathtt{e}} \mathtt{s'}$ with trees part of $\mathtt{s}$ and $\mathtt{s'}$

- Languages add checks that correspond to (future) source-language conditions

Proving back-translation (2)

- All intermediate languages equipped with small-step semantics
- Execution guided by trace: steps reduce the tree by emitting one event

  $\texttt{s} \rightarrow_{\texttt{e}} \texttt{s'}$ with trees part of $\texttt{s}$ and $\texttt{s'}$

- Languages add checks that correspond to (future) source-language conditions
  - Only one new check per step (simplifies reasoning)

Proving back-translation (2)

- All intermediate languages equipped with small-step semantics
- Execution guided by trace: steps reduce the tree by emitting one event

$\texttt{s} \rightarrow_{\texttt{e}} \texttt{s'}$ with trees part of $\texttt{s}$ and $\texttt{s'}$

- Languages add checks that correspond to (future) source-language conditions
  - Only one new check per step (simplifies reasoning)
  - For instance, returns can only occur when the stack isn't empty and the top stack-frame records the caller/callee

Proving back-translation (2)

- All intermediate languages equipped with small-step semantics
- Execution guided by trace: steps reduce the tree by emitting one event

$s \rightarrow_e s'$ with trees part of s and s'

- Languages add checks that correspond to (future) source-language conditions
  - Only one new check per step (simplifies reasoning)
  - For instance, returns can only occur when the stack isn't empty and the top stack-frame records the caller/callee

$s, stack \rightarrow_e s', stack'$

Proving back-translation (2)

- All intermediate languages equipped with small-step semantics
- Execution guided by trace: steps reduce the tree by emitting one event

$\texttt{s} \rightarrow_{\texttt{e}} \texttt{s'}$ with trees part of $\texttt{s}$ and $\texttt{s'}$

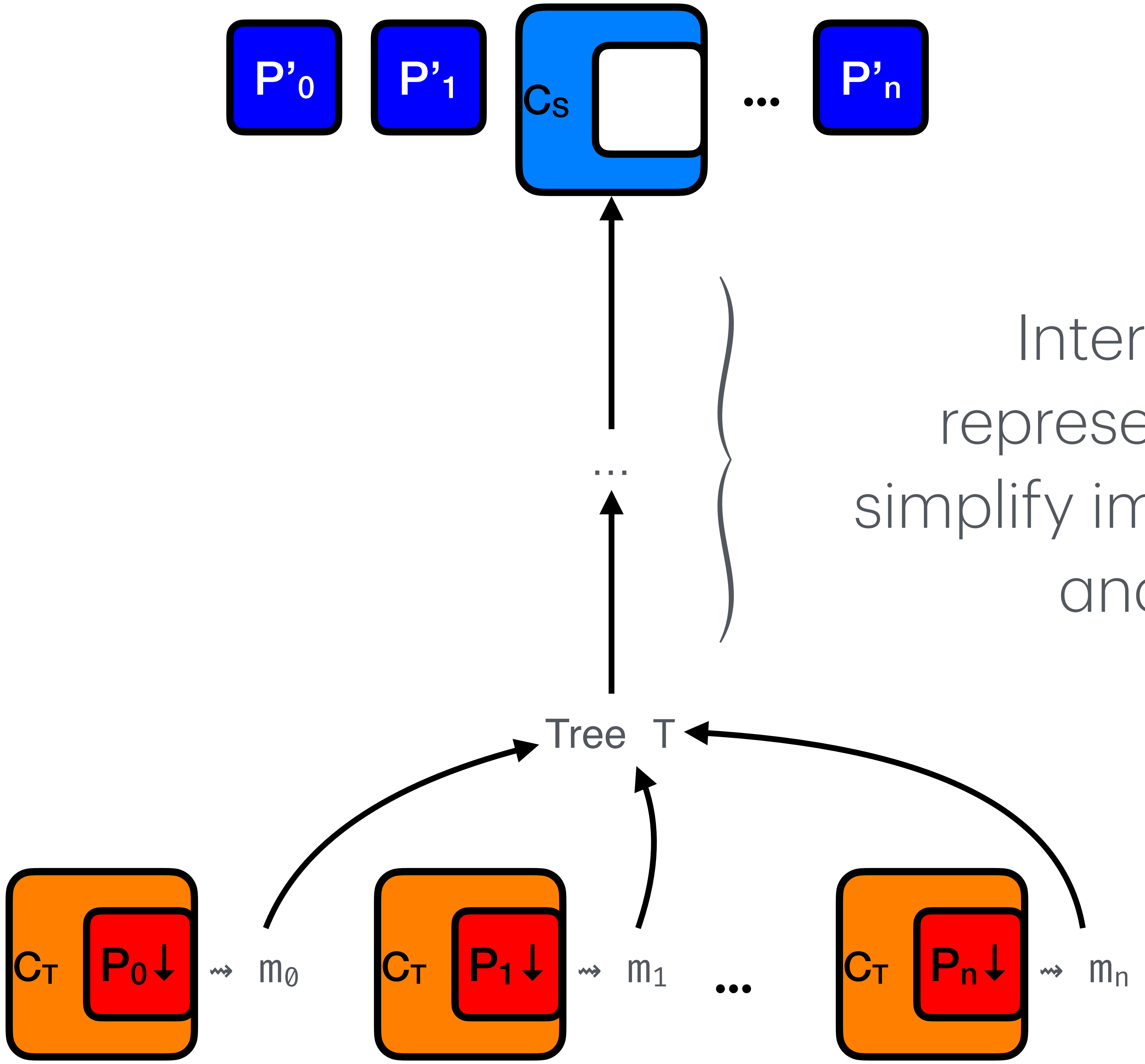
- Languages add checks that correspond to (future) source-language conditions
  - Only one new check per step (simplifies reasoning)
  - For instance, returns can only occur when the stack isn't empty and the top stack-frame records the caller/callee

$$\frac{\texttt{stack = (C,P) :: stack'} \qquad \texttt{e = Ret P C z} \qquad \dots}{\texttt{s, stack} \rightarrow_{\texttt{e}} \texttt{s', stack'}}$$

Proving back-translation (3)

- Each pass: CompCert-style forward simulation
  - Small passes means individual proofs are not so complicated
- What is difficult: “flattening”
  - Pass that goes from trees to list of nodes (closer to final code)
  - Unicity and determinacy conditions
- In the actual implementation: multiples compartments, not just two

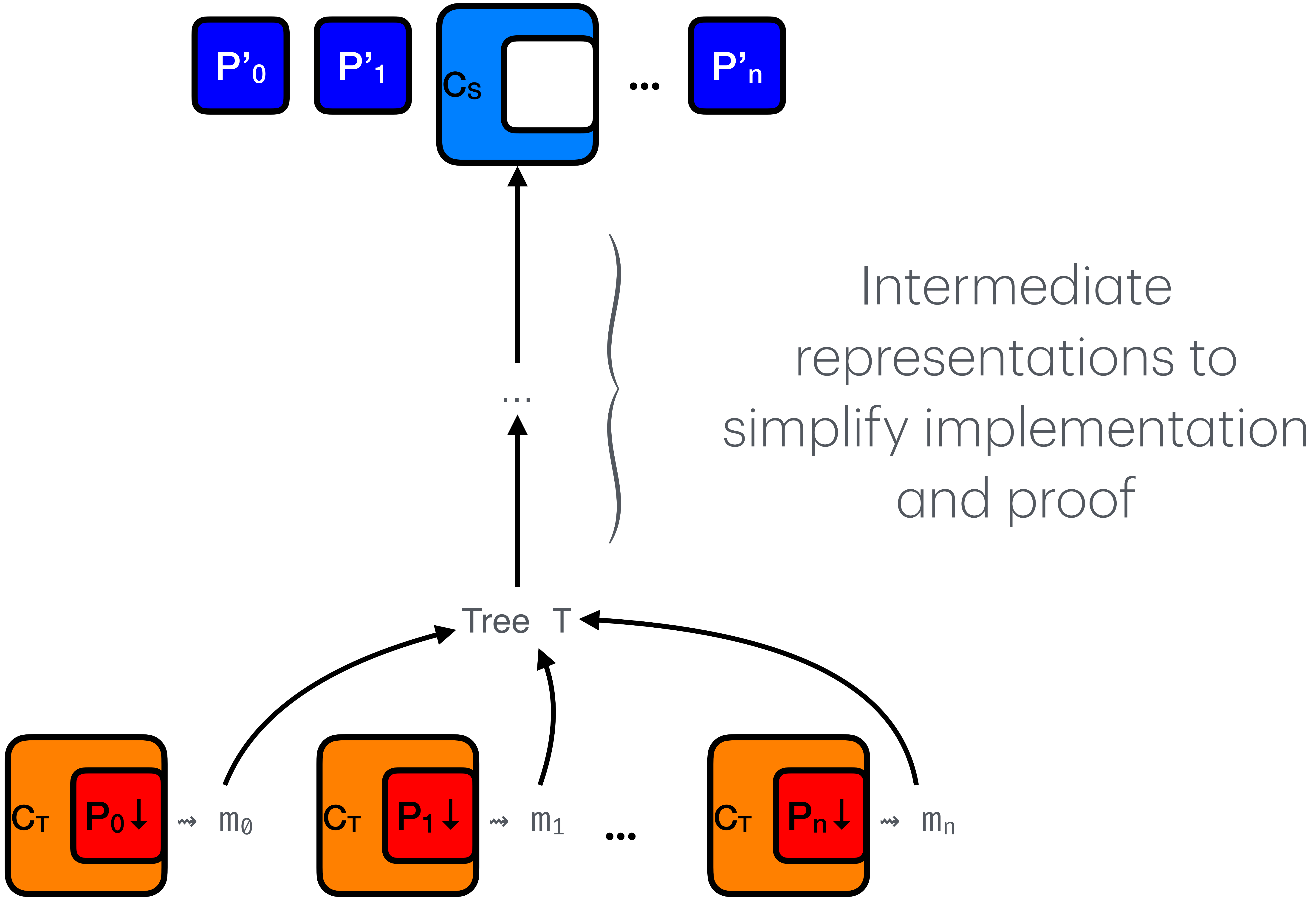
Conclusion



Intermediate representations to simplify implementation and proof

Conclusion

- Adapted a proof to obtain a much strong criterion

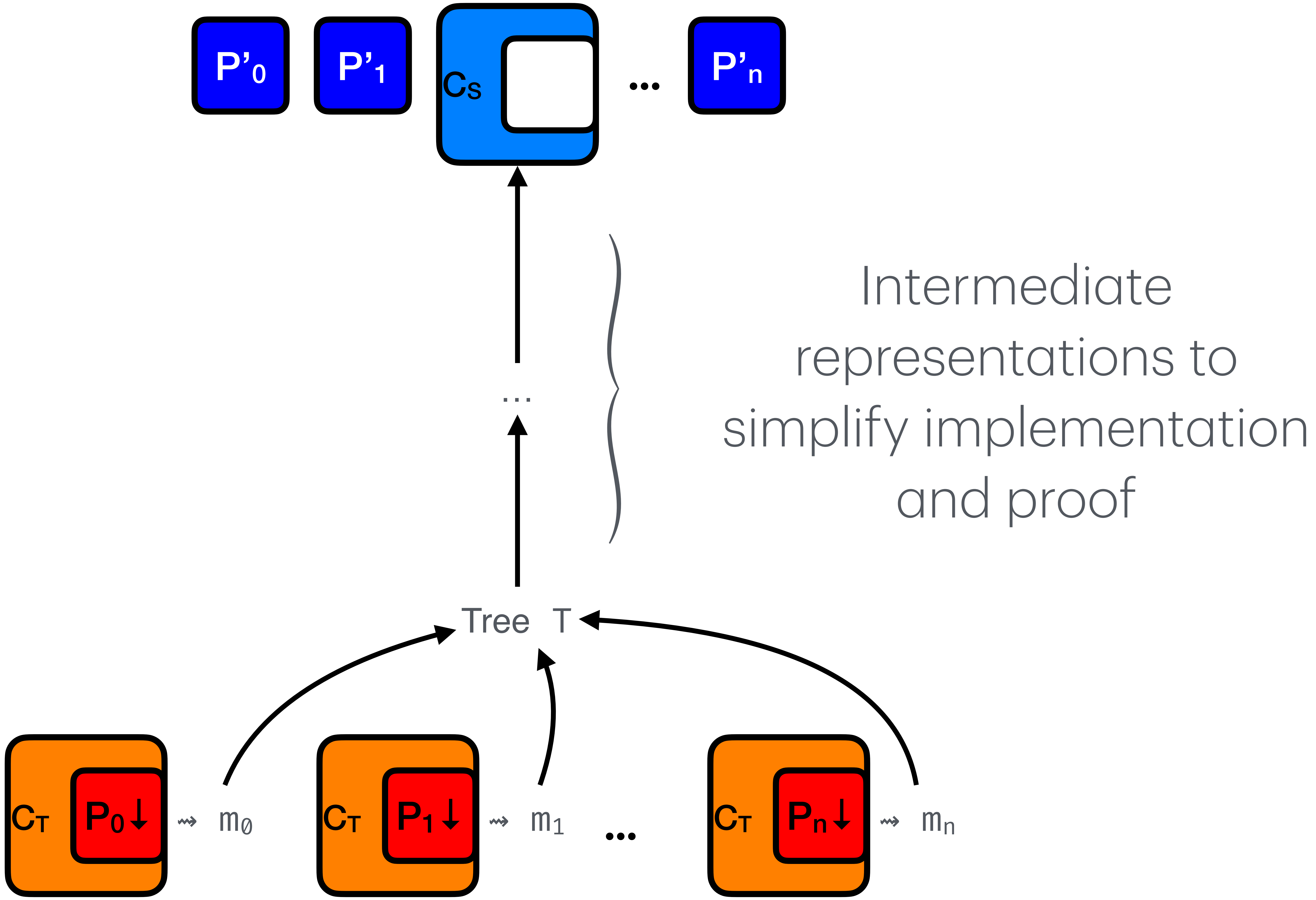


P'₀
P'₁
Cs
...
P'ₙ
Intermediate representations to simplify implementation and proof
...
Tree T
CT
P₀↓
⇝ m₀
CT
P₁↓
⇝ m₁
...
CT
Pₙ↓
⇝ mₙ

Conclusion

- Adapted a proof to obtain a much strong criterion

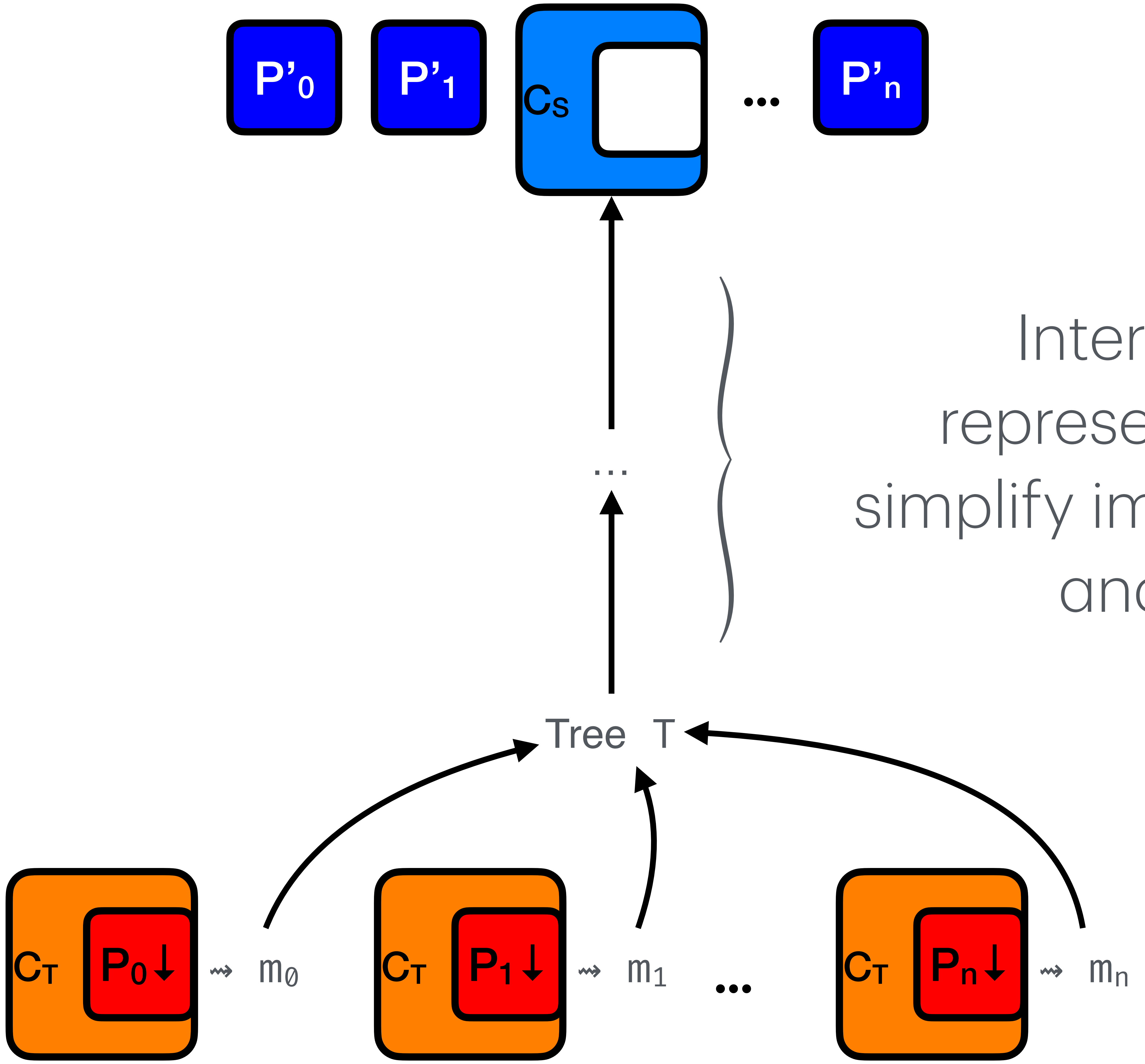
- Simplified implementation and proof by slicing the back-translation into multiple steps



P'₀
P'₁
Cs
...
P'ₙ
...
Intermediate representations to simplify implementation and proof
Tree T
CT
P₀↓
⇝ m₀
CT
P₁↓
⇝ m₁
...
CT
Pₙ↓
⇝ mₙ

Conclusion

- Adapted a proof to obtain a much strong criterion
- Simplified implementation and proof by slicing the back-translation into multiple steps
- Also helps with parallelizing the proof process



Intermediate representations to simplify implementation and proof