Modular certification of Why3

Deducteam seminar

July 2, 2020

Quentin Garchery

The `instantiate` example in Why3

Program with annotations [Dailler, Marché, FIDE 2018] :

```
let f (a:array int) (x:int) : int
  requires { a.length ≥ 1000 ∧ 0 ≤ x ≤ 10 }
  requires { forall i. 0 ≤ 4*i+1 < a.length → a[4*i+1] ≥ 0 }
  ensures { result ≥ 0 }
= let y = 2*x+1 in a[y*y]
```

To prove :

```
forall a:array int, x:int.
length a ≥ 1000 ∧ 0 ≤ x ≤ 10 →
(forall i:int. 0 ≤ 4 * i + 1 < length a → a[4 * i + 1] ≥ 0) →
let y = 2 * x + 1 in (0 ≤ y * y < length a) ∧ a[y * y] ≥ 0
```

The `instantiate` example in Why3

Program with annotations [Dailler, Marché, FIDE 2018] :

```
let f (a:array int) (x:int) : int
  requires { a.length ≥ 1000 ∧ 0 ≤ x ≤ 10 }
  requires { forall i. 0 ≤ 4*i+1 < a.length → a[4*i+1] ≥ 0 }
  ensures { result ≥ 0 }
= let y = 2*x+1 in a[y*y]
```

Transform what there is to prove (with `split_vc`) :

```
Req1 : length a ≥ 1000 ∧ 0 ≤ x ≤ 10
Req2 : forall i:int. 0 ≤ 4 * i + 1 < length a → a[4 * i + 1] ≥ 0
y : int = 2 * x + 1
------------------------
goal AccessInBounds : 0 ≤ (y * y) < length a
------------------------
goal Postcondition : a[y * y] ≥ 0
```

The `instantiate` example in Why3

Program with annotations [Dailler, Marché, FIDE 2018] :

```
let f (a:array int) (x:int) : int
  requires { a.length ≥ 1000 ∧ 0 ≤ x ≤ 10 }
  requires { forall i. 0 ≤ 4*i+1 < a.length → a[4*i+1] ≥ 0 }
  ensures { result ≥ 0 }
= let y = 2*x+1 in a[y*y]
```

Transform postcondition (with `instantiate`) :

Req1 : length a $\geq 1000 \wedge 0 \leq x \leq 10$

Req2 : forall i:int. $0 \leq 4 * i + 1 <$ length a $\rightarrow a[4 * i + 1] \geq 0$

y : int $= 2 * x + 1$

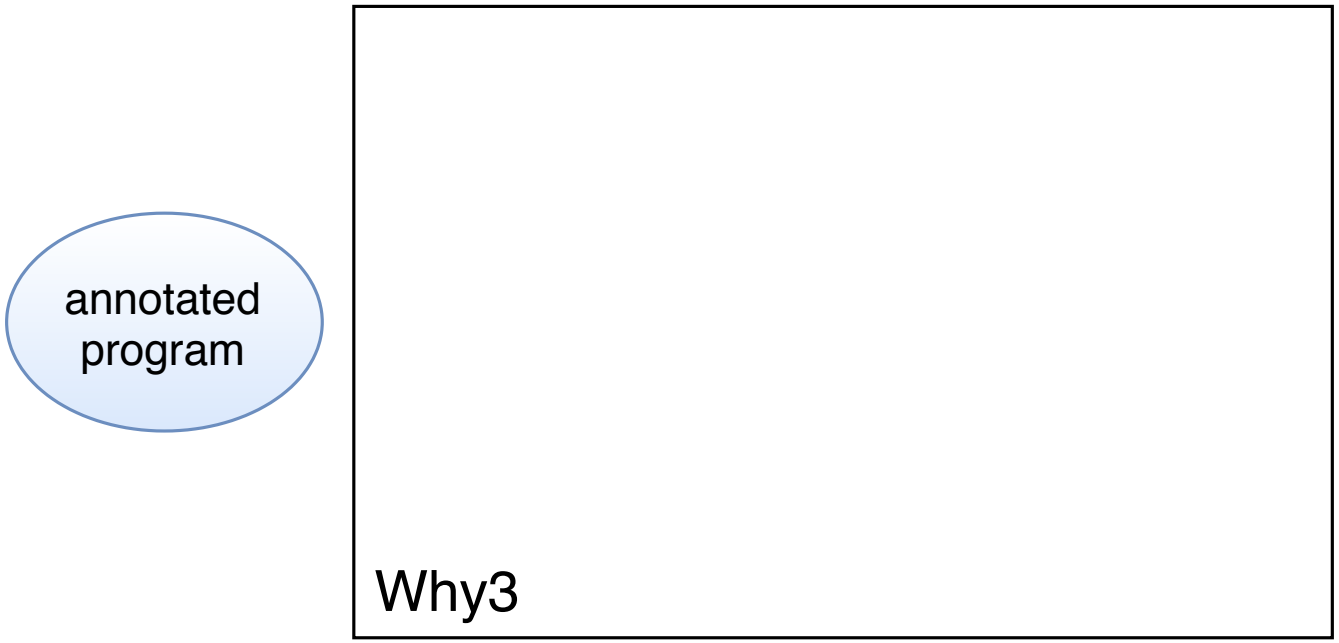
Hinst : $0 \leq 4 * (x * x + x) + 1 <$ length a $\rightarrow$
$a[4 * (x * x + x) + 1] \geq 0$

goal Postcondition : $a[y * y] \geq 0$

Overview of Why3

Why3 step by step :

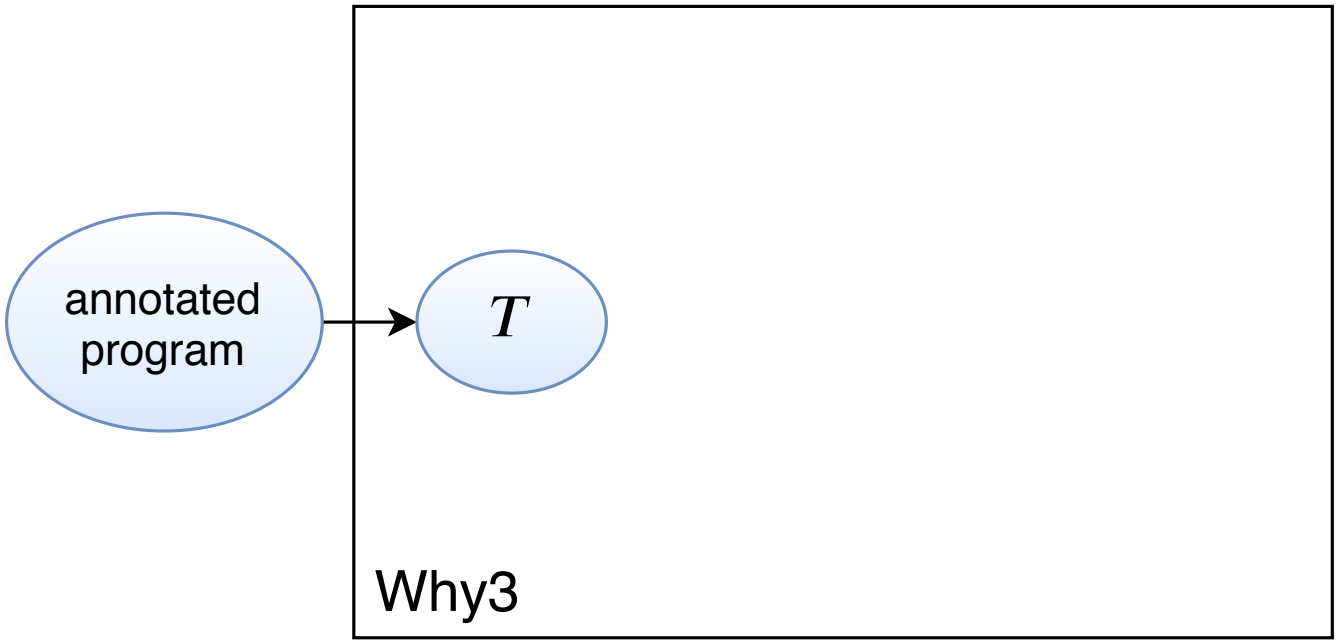
- initial task generation
- logical transformations
- call to automatic theorem provers



Overview of Why3

Why3 step by step :

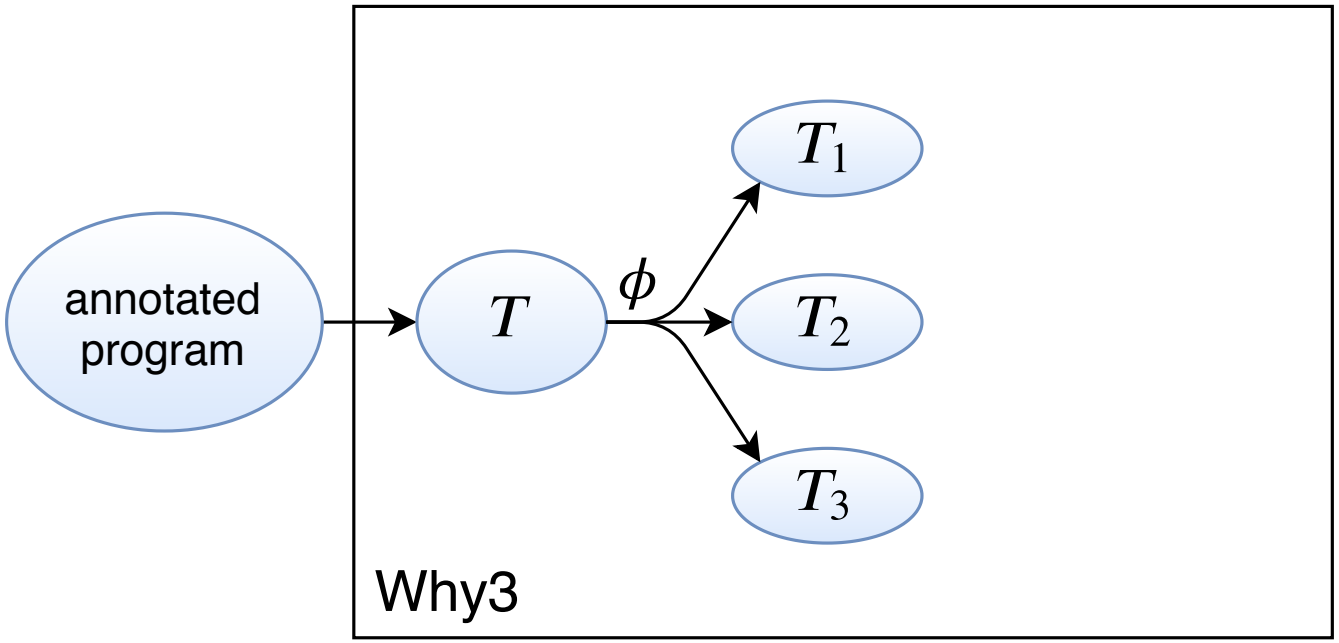
- initial task generation
- logical transformations
- call to automatic theorem provers



Overview of Why3

Why3 step by step :

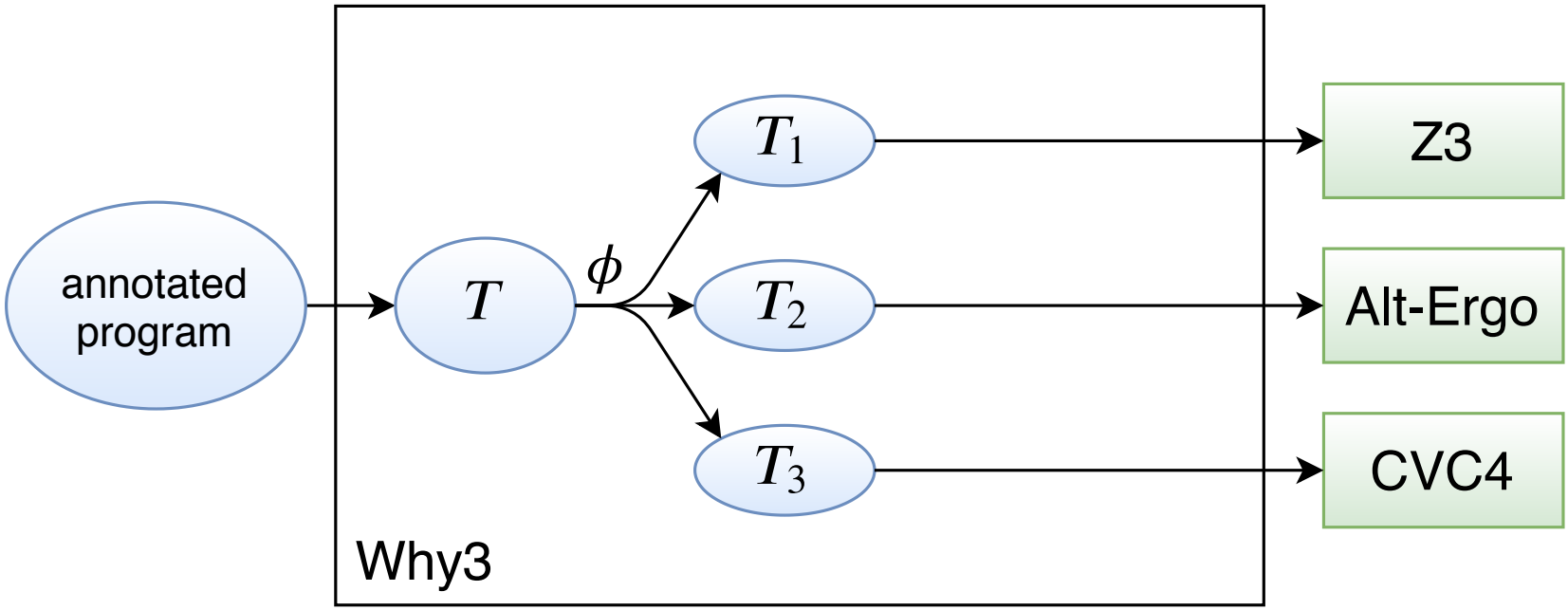
- initial task generation
- logical transformations
- call to automatic theorem provers



Overview of Why3

Why3 step by step :

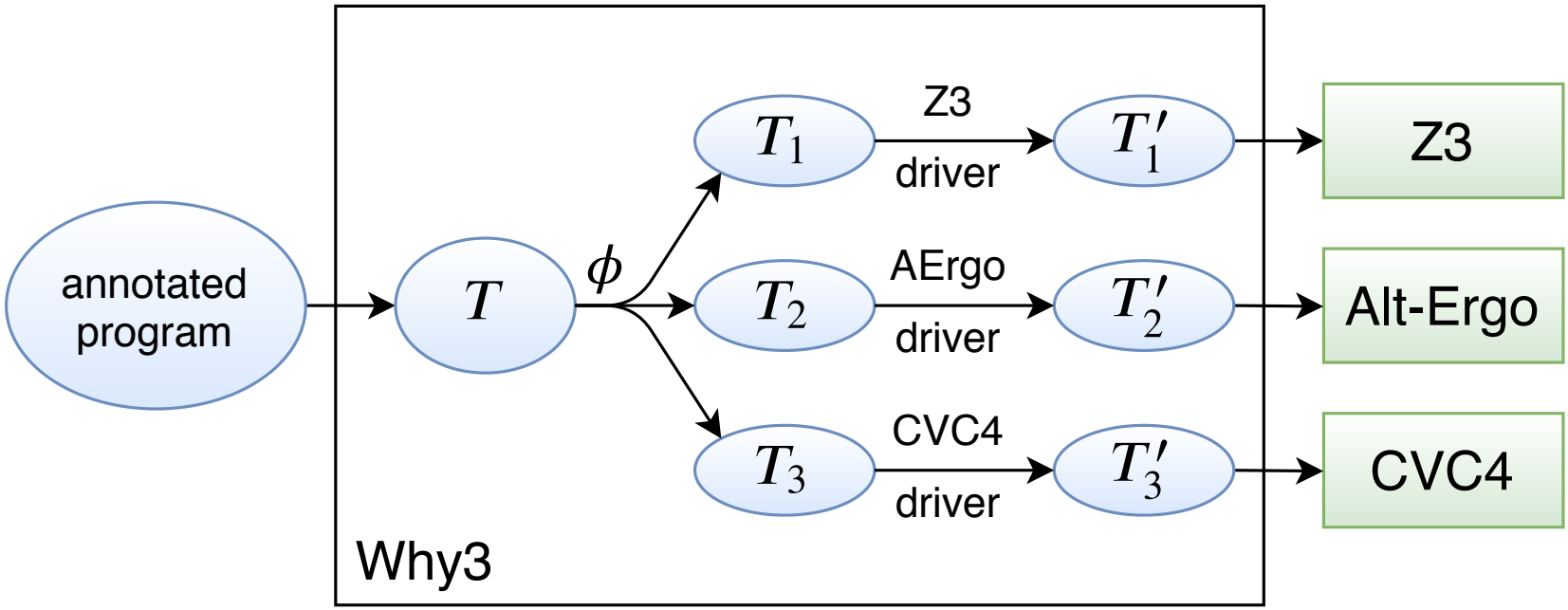
- initial task generation
- logical transformations
- call to automatic theorem provers



Overview of Why3

Why3 step by step :

- initial task generation
- logical transformations
- call to automatic theorem provers



Overview

How to improve trust in Why3

Reduce trust base by :

↳ isolating a kernel

↳ a posteriori certification

Emphasis on the modular and incremental approach

Plan

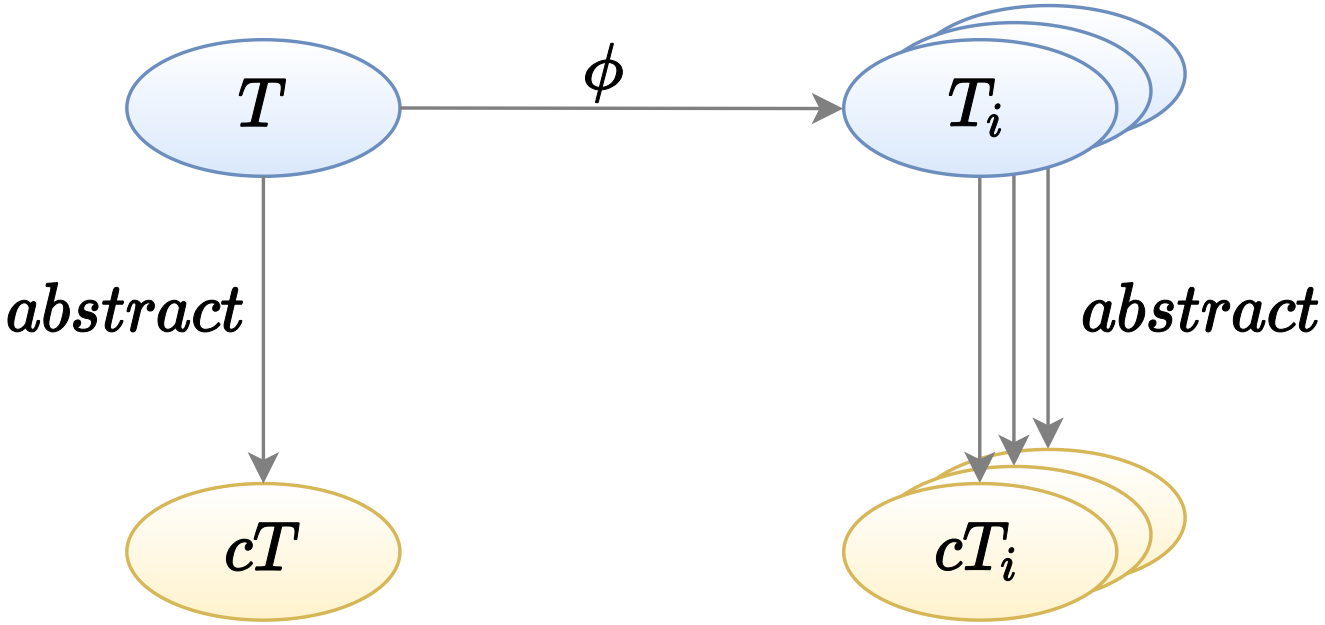
- Motivation
1. Certification framework
2. Checkers
3. High level API
4. Conclusion

Task abstraction : motivation

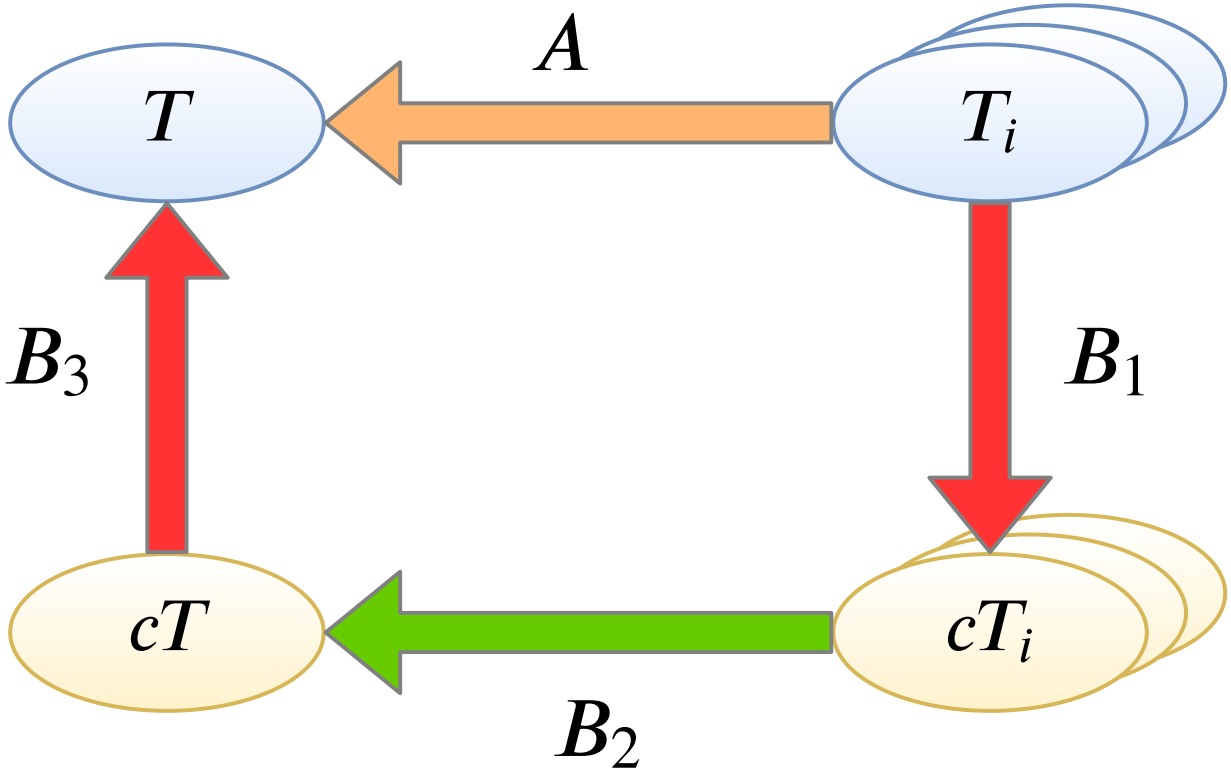
Proof tasks with meta-data, type and signature

$\hookrightarrow$ extract the core formula as a sequent in first-order untyped logic

$H_1 : A_1, ..., H_m : A_m \vdash G_1 : B_1, ..., G_n : B_n$



Task abstraction : correction

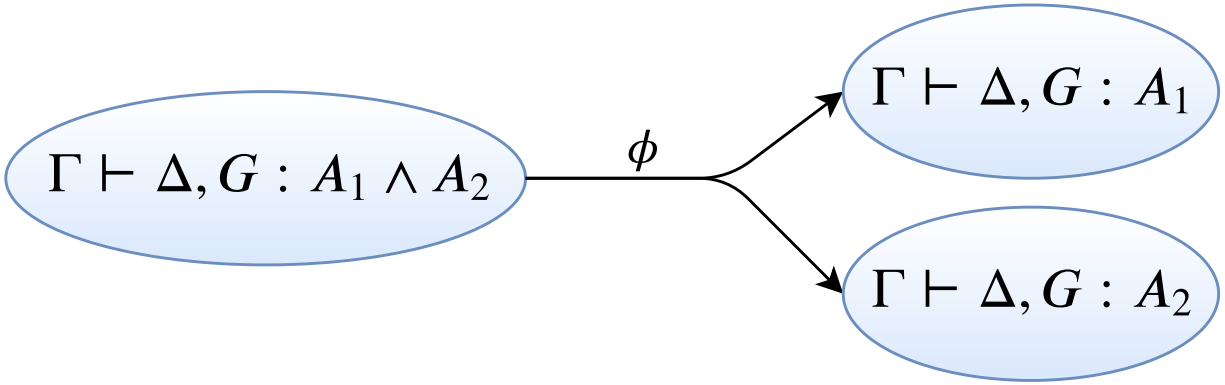


$\Longrightarrow$ we need equisatisfiability between a task and its abstraction

Certificates

```
type certif :=
 | Split of ident * certif * certif
 | Axiom of ident * ident
 | Hole
 | ...
```

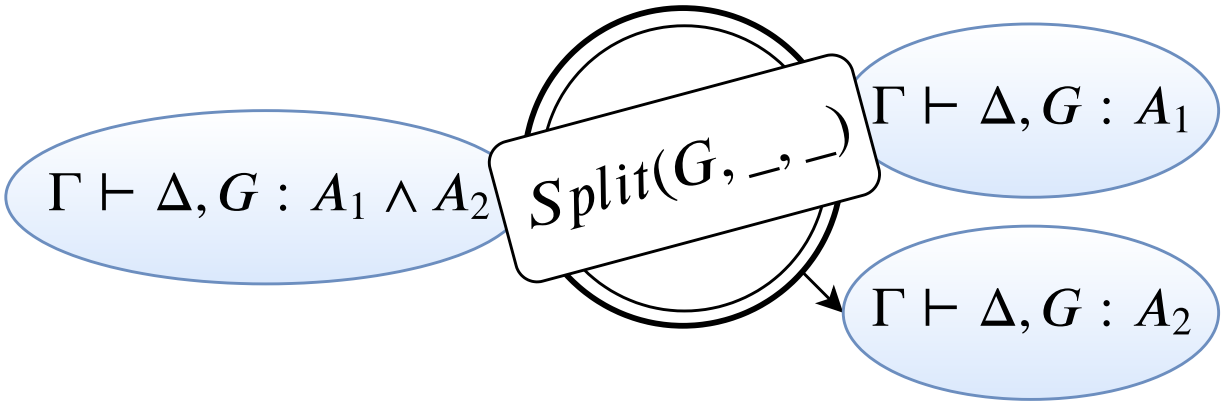
A constructor ⇔ An elementary transformation



Certificates

```
type certif :=
 | Split of ident * certif * certif
 | Axiom of ident * ident
 | Hole
 | ...
```

A constructor ⇔ An elementary transformation



Certificates

```
type certif :=
 | Split of ident * certif * certif
 | Axiom of ident * ident
 | Hole
 | ...
```

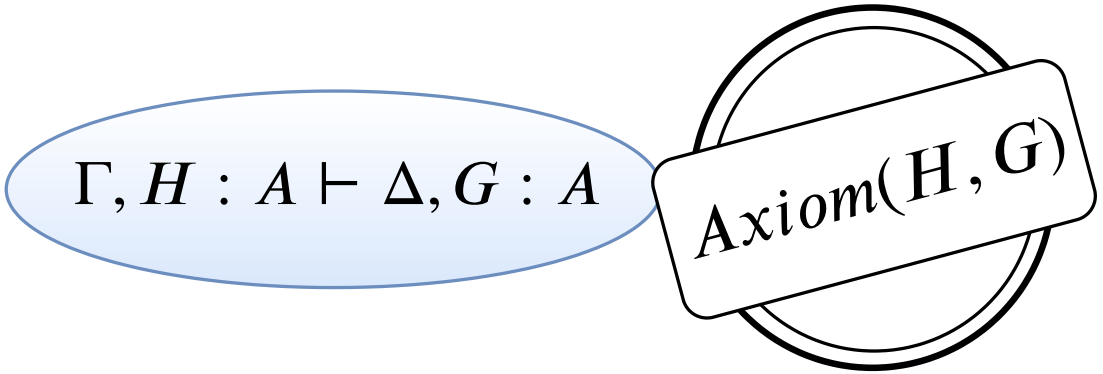
A constructor ⇔ An elementary transformation

$$\Gamma, H : A \vdash \Delta, G : A \xrightarrow{\phi}$$

Certificates

```
type certif :=
 | Split of ident * certif * certif
 | Axiom of ident * ident
 | Hole
 | ...
```

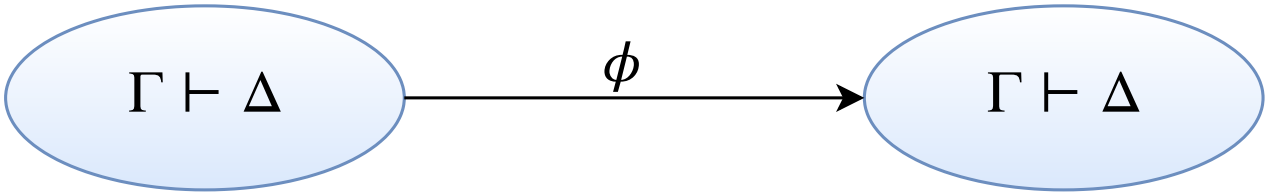
A constructor ⇔ An elementary transformation



Certificates

```
type certif :=
 | Split of ident * certif * certif
 | Axiom of ident * ident
 | Hole
 | ...
```

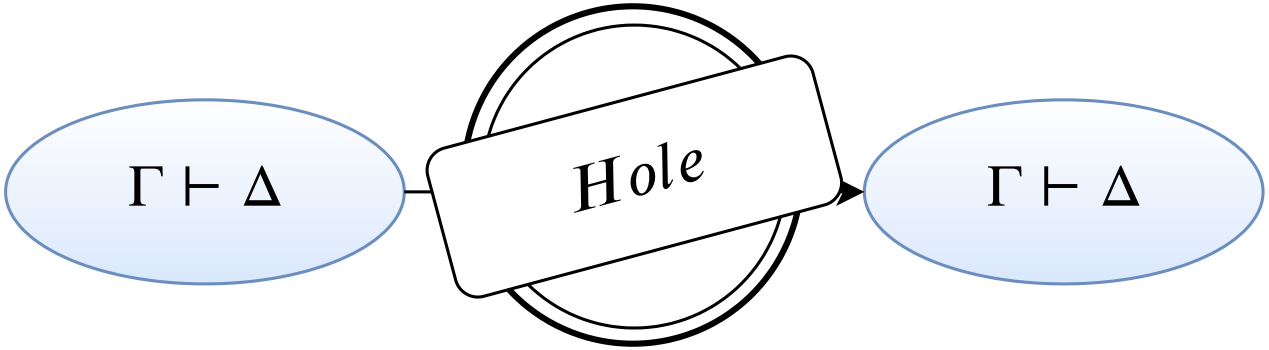
A constructor ⇔ An elementary transformation



Certificates

```
type certif :=
 | Split of ident * certif * certif
 | Axiom of ident * ident
 | Hole
 | ...
```

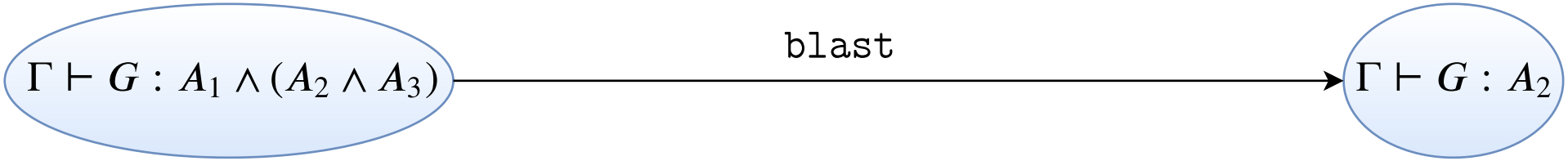
A constructor ⇔ An elementary transformation



An example : the `blast` transformation

Pose : $\Gamma := H_1 : A_1, H_3 : A_3$

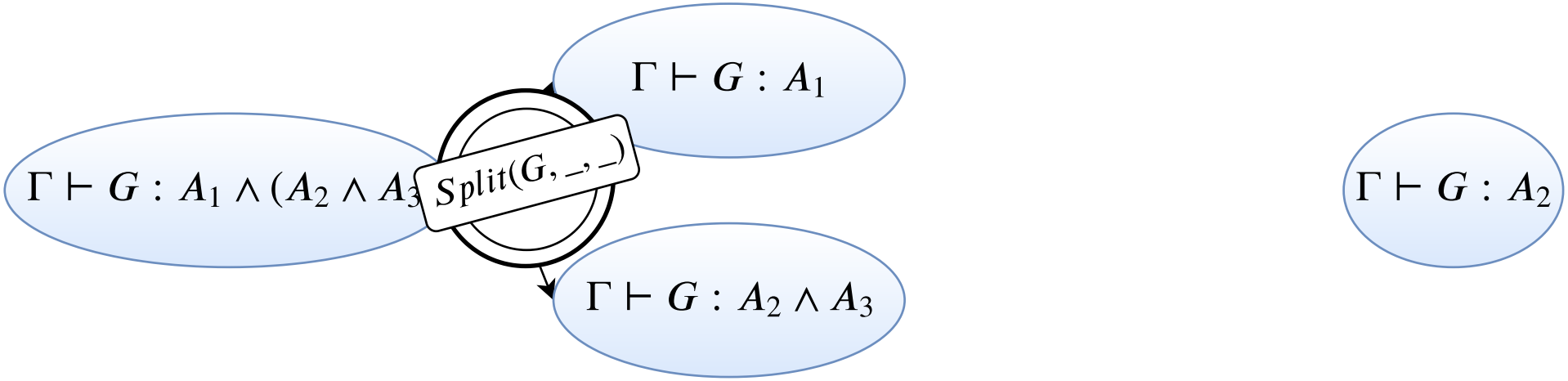
$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$



An example : the `blast` transformation

Pose : $\Gamma := H_1 : A_1, H_3 : A_3$

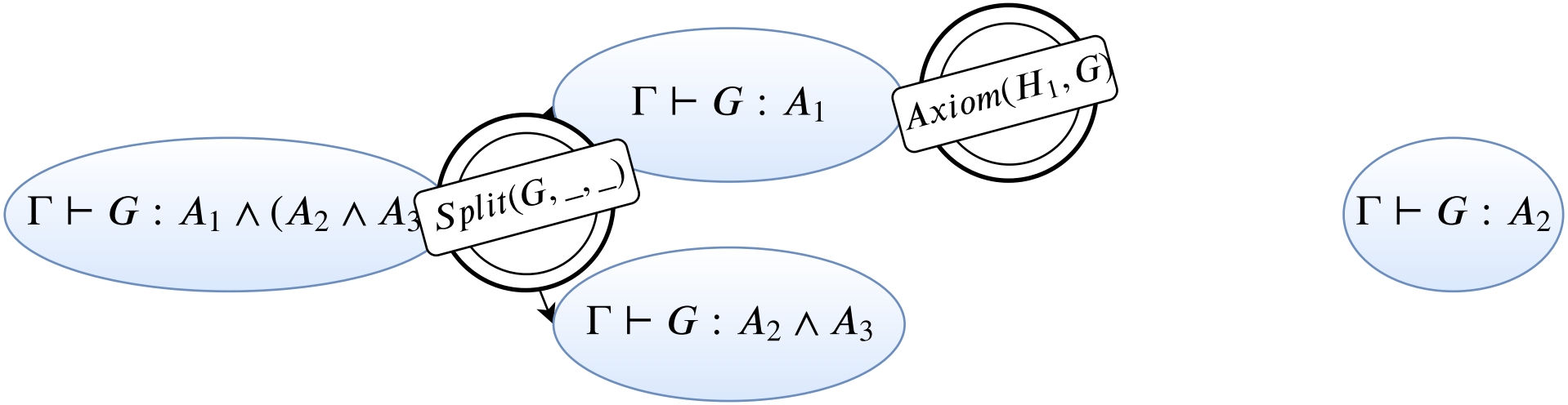
$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$



An example : the `blast` transformation

Pose : $\Gamma := H_1 : A_1, H_3 : A_3$

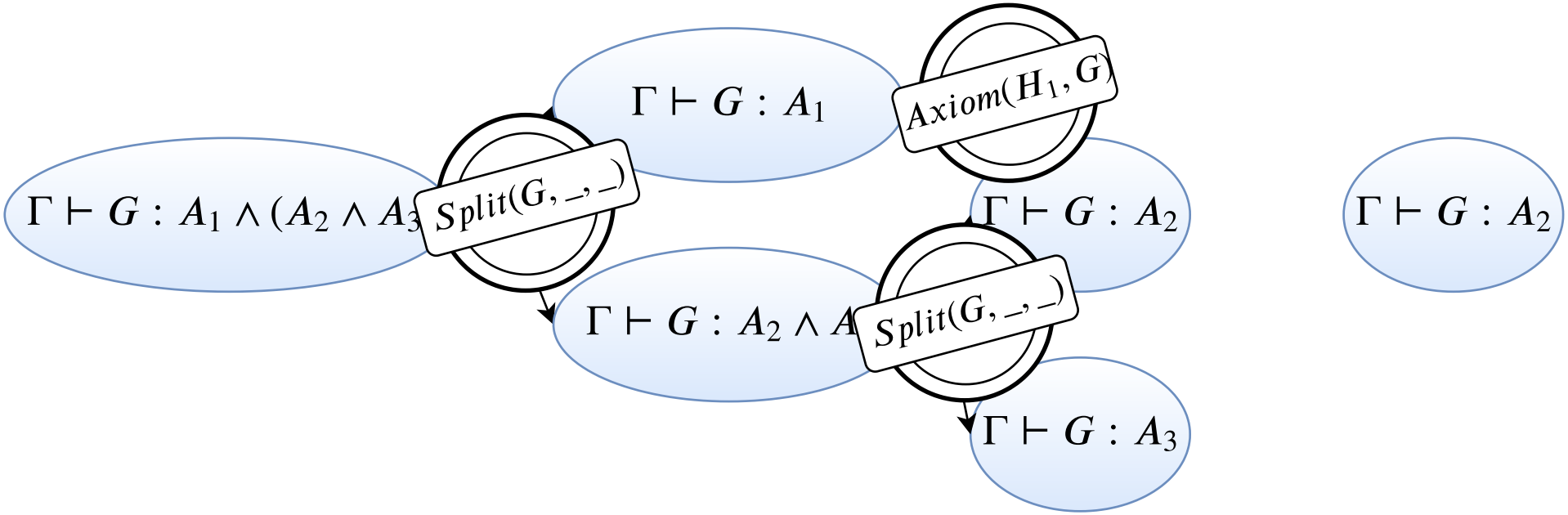
$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$



An example : the `blast` transformation

Pose : $\Gamma := H_1 : A_1, H_3 : A_3$

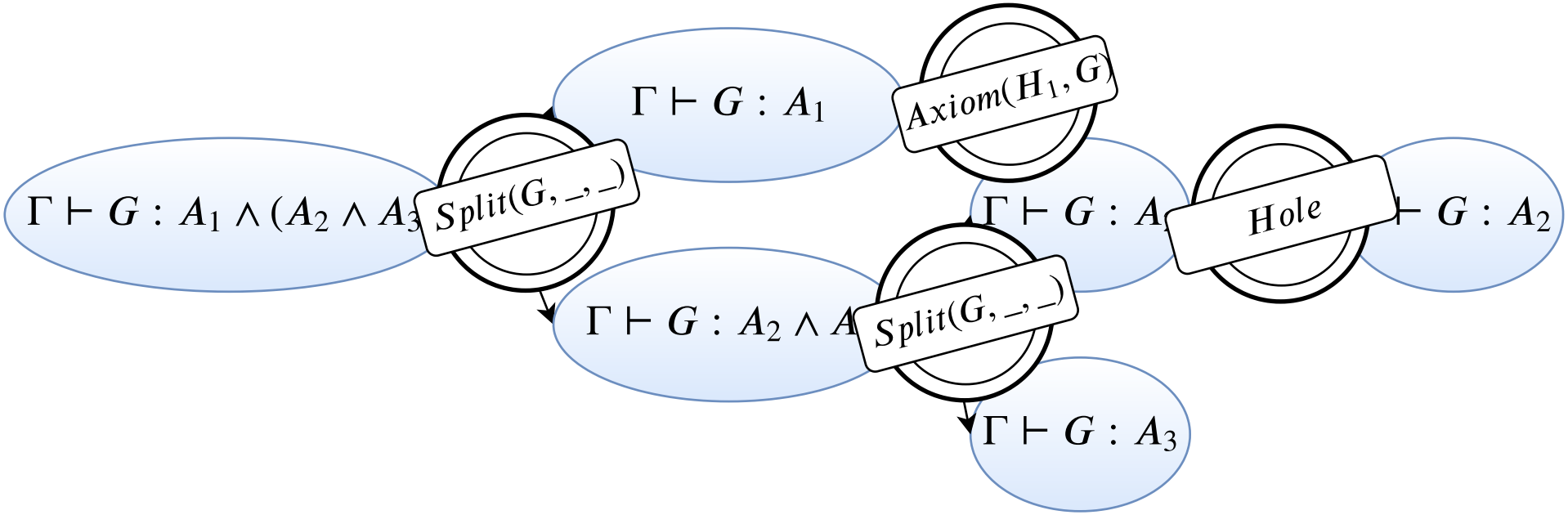
$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$



An example : the `blast` transformation

Pose : $\Gamma := H_1 : A_1, H_3 : A_3$

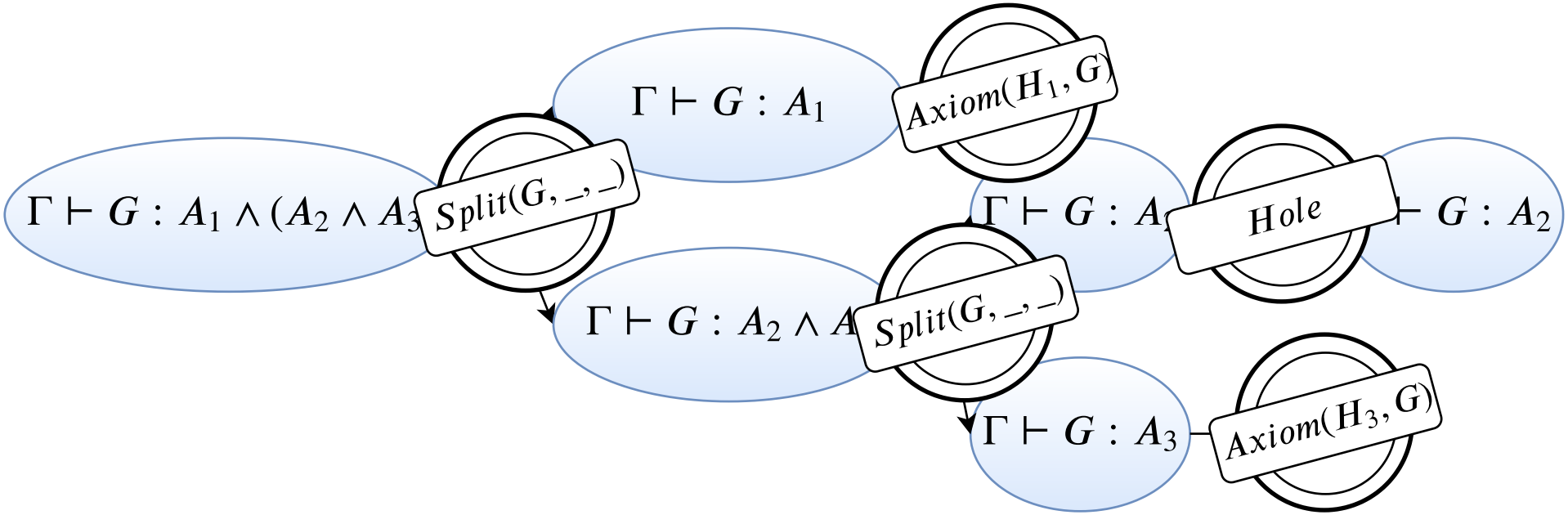
$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$



An example : the `blast` transformation

Pose : $\Gamma := H_1 : A_1, H_3 : A_3$

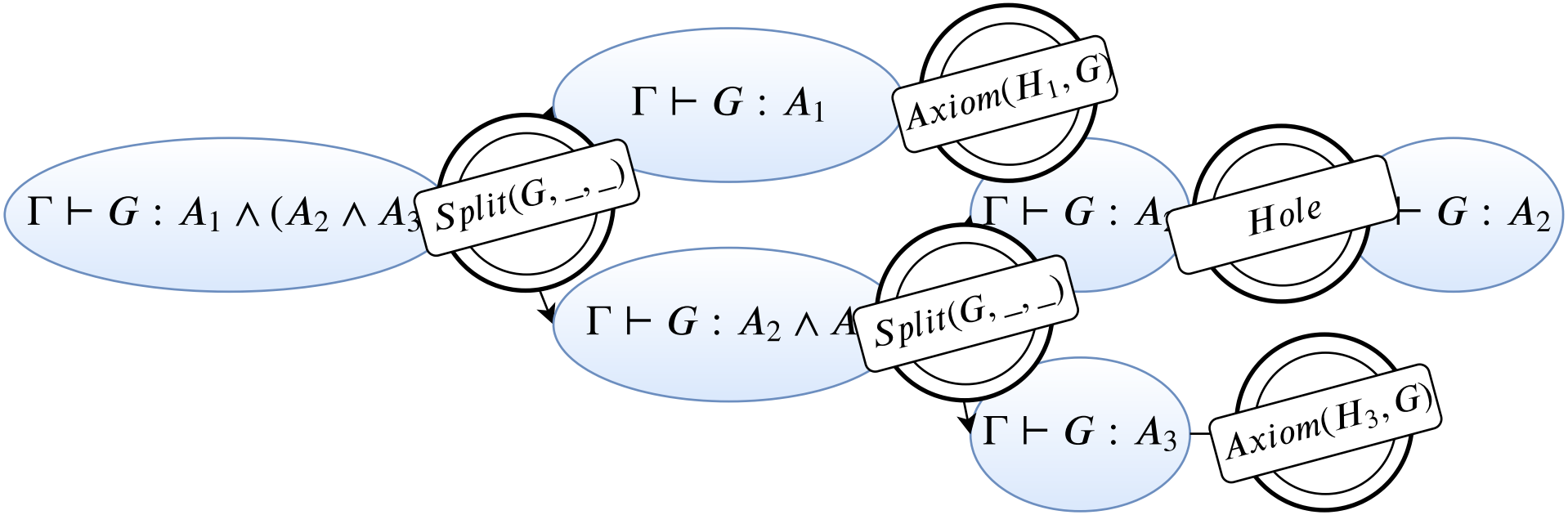
$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$



An example : the `blast` transformation

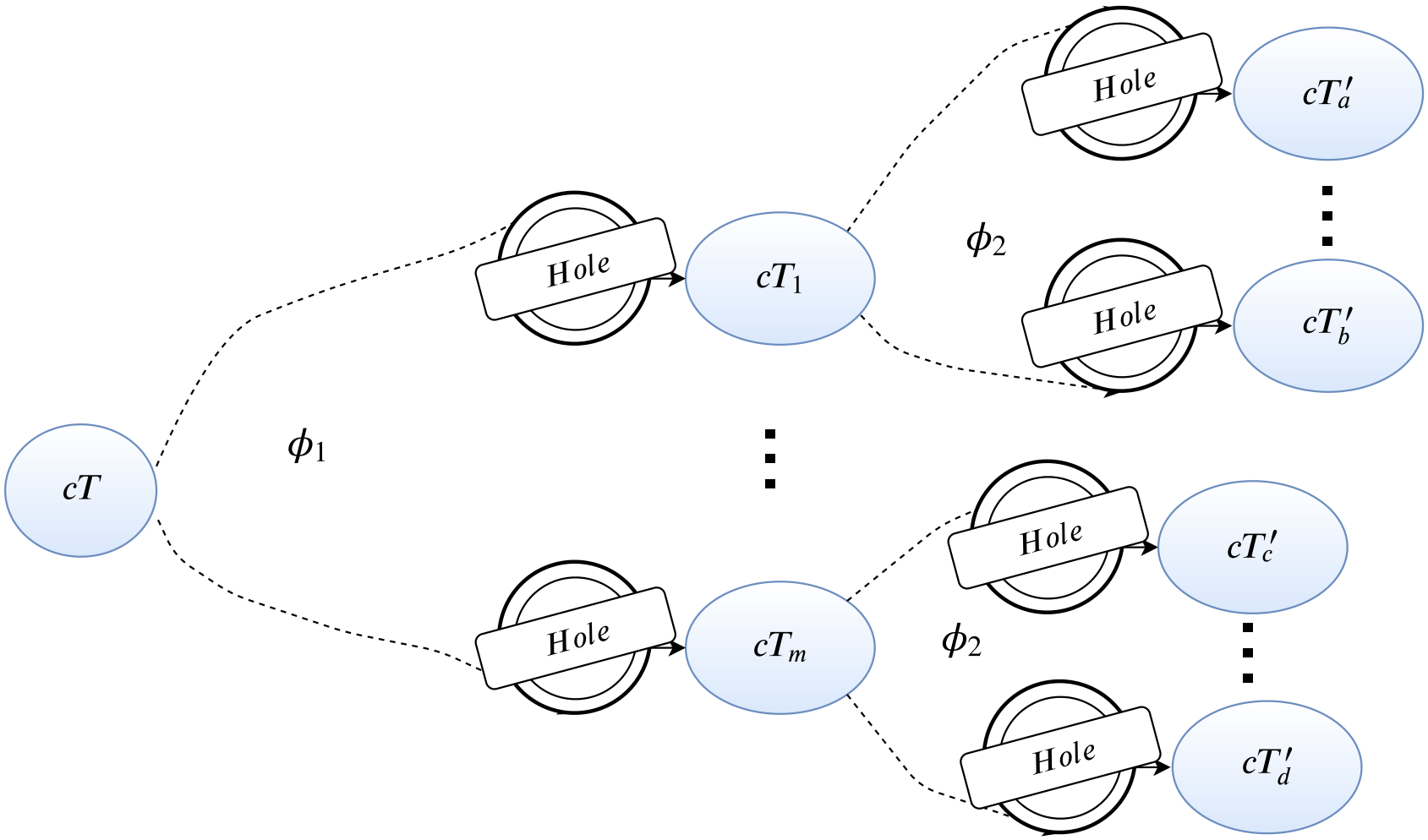
Pose : $\Gamma := H_1 : A_1, H_3 : A_3$

$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$

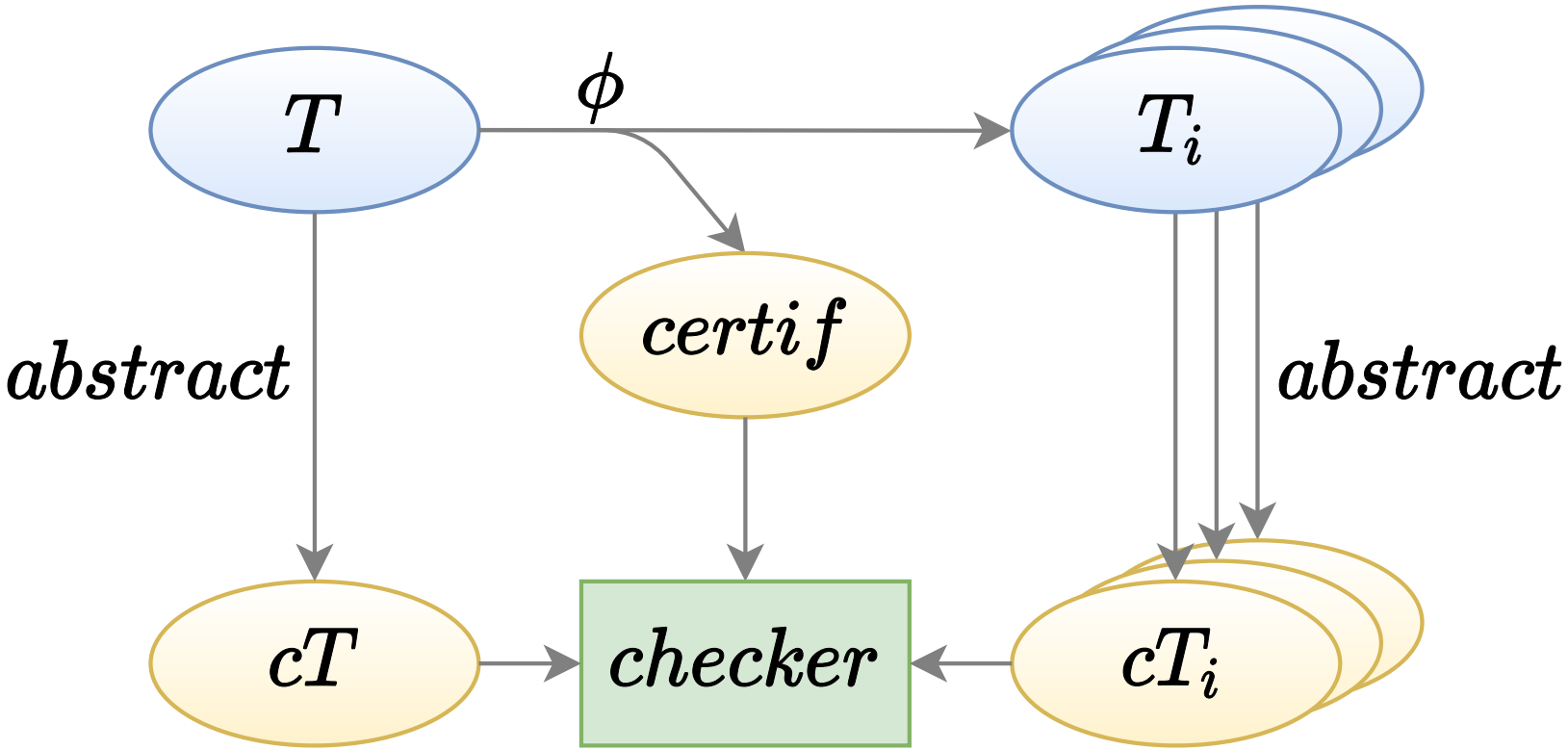


$$Split\,(G, Axiom(H_1, G), Split(G, Hole, Axiom(H_3, G)))$$

Composition of certifying transformations

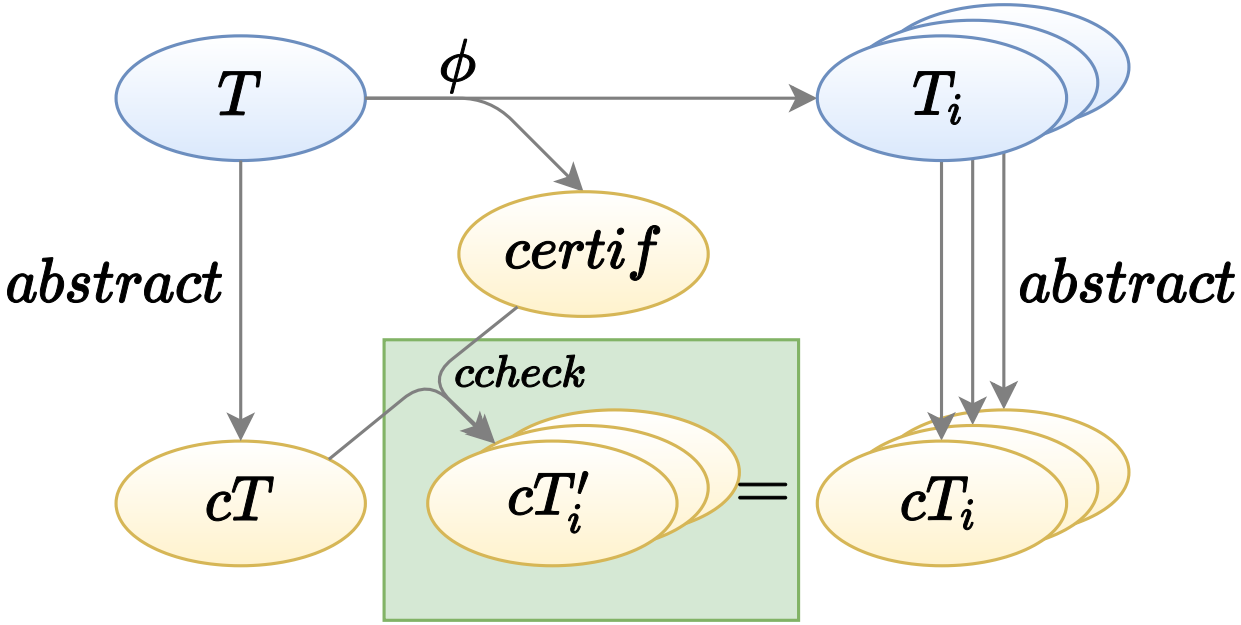


Framework for certificates



OCaml checker : overview

ccheck : `certif -> task -> task list`



Correction of *ccheck*

When *ccheck certif cT* computes into *cTi′*, the conjunction of all the *cTi′* implies *cT*.

OCaml checker : computing *ccheck*

The `blast` example :

$$\Gamma := H_1 : A_1, H_3 : A_3$$

$$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$$

$$certif := Split\,(G, Axiom(H_1, G), Split(G, Hole, Axiom(H_3, G)))$$

OCaml checker : computing *ccheck*

The `blast` example :

$$\Gamma := H_1 : A_1, H_3 : A_3$$
$$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$$
$$\mathit{certif} := \mathit{Split}\,(G, \mathit{Axiom}(H_1, G), \mathit{Split}(G, \mathit{Hole}, \mathit{Axiom}(H_3, G)))$$

Compute :

$$\begin{aligned} &\mathit{ccheck}\ \mathit{certif}\ T \\ &\quad \equiv \mathit{ccheck}\,(\mathit{Axiom}(H_1, G))\,(\Gamma \vdash G : A_1) \\ &\qquad @\ \mathit{ccheck}\,(\mathit{Split}(G, \mathit{Hole}, \mathit{Axiom}(H_3, G)))\,(\Gamma \vdash G : A_2 \wedge A_3) \end{aligned}$$

OCaml checker : computing *ccheck*

The `blast` example :

$$\Gamma := H_1 : A_1, H_3 : A_3$$
$$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$$
$$\mathit{certif} := \mathit{Split}\,(G, \mathit{Axiom}(H_1, G), \mathit{Split}(G, \mathit{Hole}, \mathit{Axiom}(H_3, G)))$$

Compute :

$$\begin{aligned}
&\mathit{ccheck}\ \mathit{certif}\ T \\
&\quad \equiv \mathit{ccheck}\,(\mathit{Axiom}(H_1, G))\,(\Gamma \vdash G : A_1) \\
&\qquad @\ \mathit{ccheck}\,(\mathit{Split}(G, \mathit{Hole}, \mathit{Axiom}(H_3, G)))\,(\Gamma \vdash G : A_2 \wedge A_3) \\
&\quad \equiv [] \\
&\qquad @\ \mathit{ccheck}\ \mathit{Hole}\,(\Gamma \vdash G : A_2) \\
&\qquad @\ \mathit{ccheck}\,(\mathit{Axiom}(H_3, G))\,(\Gamma \vdash G : A_3)
\end{aligned}$$

OCaml checker : computing *ccheck*

The `blast` example :

$$\Gamma := H_1 : A_1, H_3 : A_3$$
$$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$$
$$\mathit{certif} := \mathit{Split}\,(G, \mathit{Axiom}(H_1, G), \mathit{Split}(G, \mathit{Hole}, \mathit{Axiom}(H_3, G)))$$

Compute :

$$\begin{aligned}
&\mathit{ccheck}\ \mathit{certif}\ T \\
&\quad \equiv \mathit{ccheck}\,(\mathit{Axiom}(H_1, G))\,(\Gamma \vdash G : A_1) \\
&\qquad @\ \mathit{ccheck}\,(\mathit{Split}(G, \mathit{Hole}, \mathit{Axiom}(H_3, G)))\,(\Gamma \vdash G : A_2 \wedge A_3) \\
&\quad \equiv [] \\
&\qquad @\ \mathit{ccheck}\ \mathit{Hole}\,(\Gamma \vdash G : A_2) \\
&\qquad @\ \mathit{ccheck}\,(\mathit{Axiom}(H_3, G))\,(\Gamma \vdash G : A_3) \\
&\quad \equiv [\Gamma \vdash G : A_2]
\end{aligned}$$

Dedukti checker : overview

Aim : avoid a formal verification of *ccheck*'s code

⇒ use Dedukti !!

The checker step by step :

1. generate, from the tasks, a type *ty* in Dedukti
2. generate, from the certificate, a term *t* in Dedukti
3. check that $t : ty$ in Dedukti

The second step does not need to be trusted

Dedukti checker : generate type

First order logic encoding + excluded middle

Use the Dedukti arrow $\rightarrow$ to encode sequents :

$$trad(H_1 : A_1, ..., H_m : A_m \vdash G_1 : B_1, ..., G_n : B_n) = \\ A_1 \rightarrow ... \rightarrow A_m \rightarrow \neg B_1 \rightarrow ... \rightarrow \neg B_n \rightarrow \bot$$

With initial task cT and resulting tasks cT_i :

$$ty = trad\,(cT_1) \rightarrow ... \rightarrow trad\,(cT_n) \rightarrow trad\,(cT)$$

$\Rightarrow$ shallow embedding

Dedukti checker : generate term

`Split` :

$$\frac{\Gamma \vdash \Delta, G : A \qquad \Gamma \vdash \Delta, G : B}{\Gamma \vdash \Delta, G : A \wedge B}$$

We define

$$\texttt{split} \ : \ (\neg A \to \bot) \to (\neg B \to \bot) \to \neg(A \wedge B) \to \bot$$

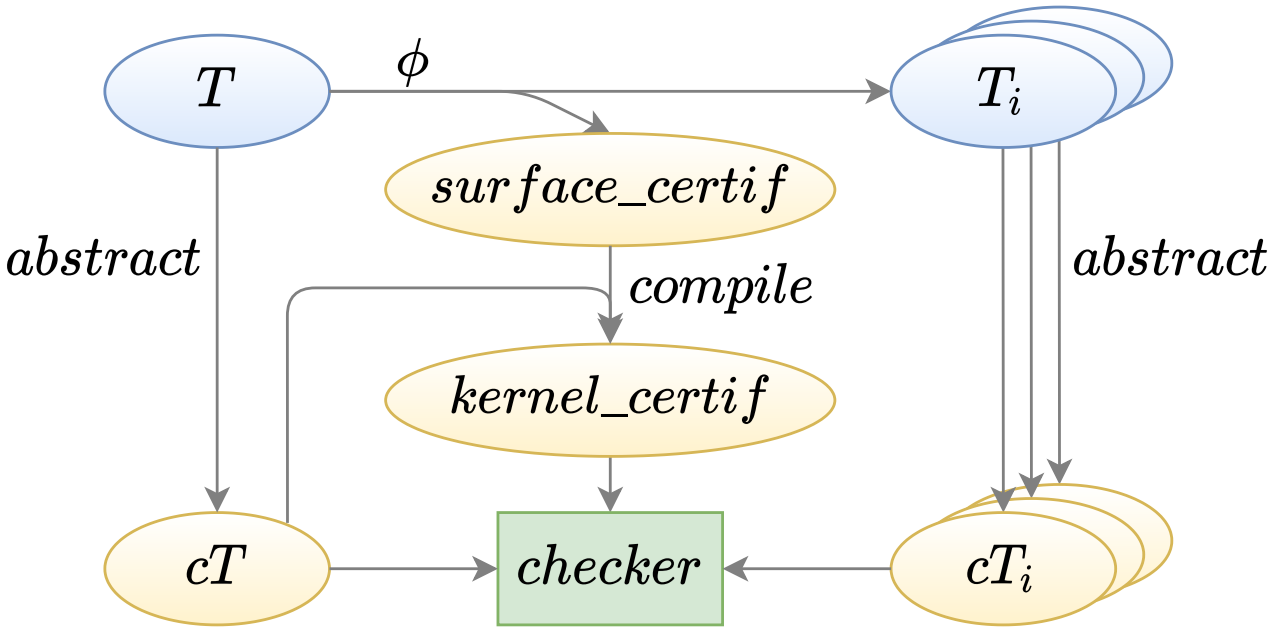
Need for elaboration :

↳ find context of application, formulas employed ...

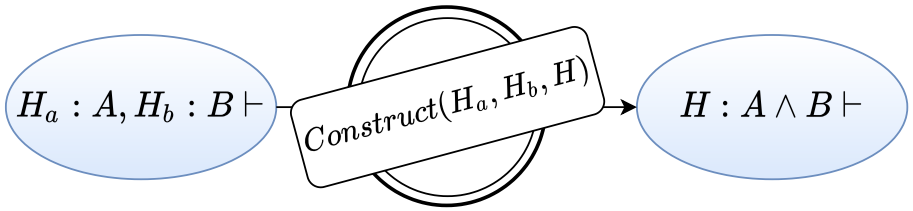
Compilation : surface and kernel certificates

Surface certificates → Kernel certificates :

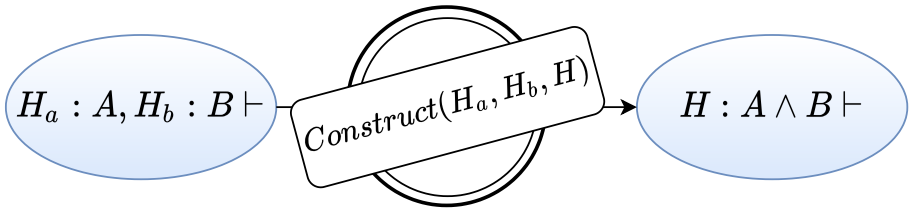
1. elaborate, go through sequents to find missing information
2. successively remove certificate constructors

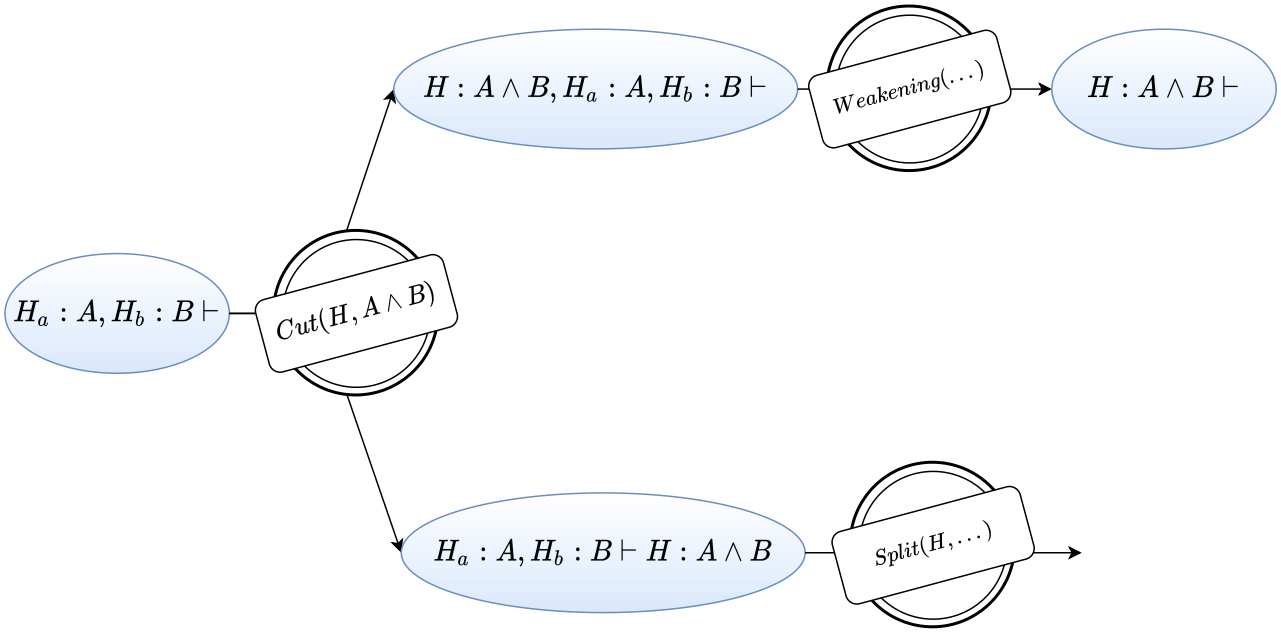


Compilation : remove `Construct` certificate



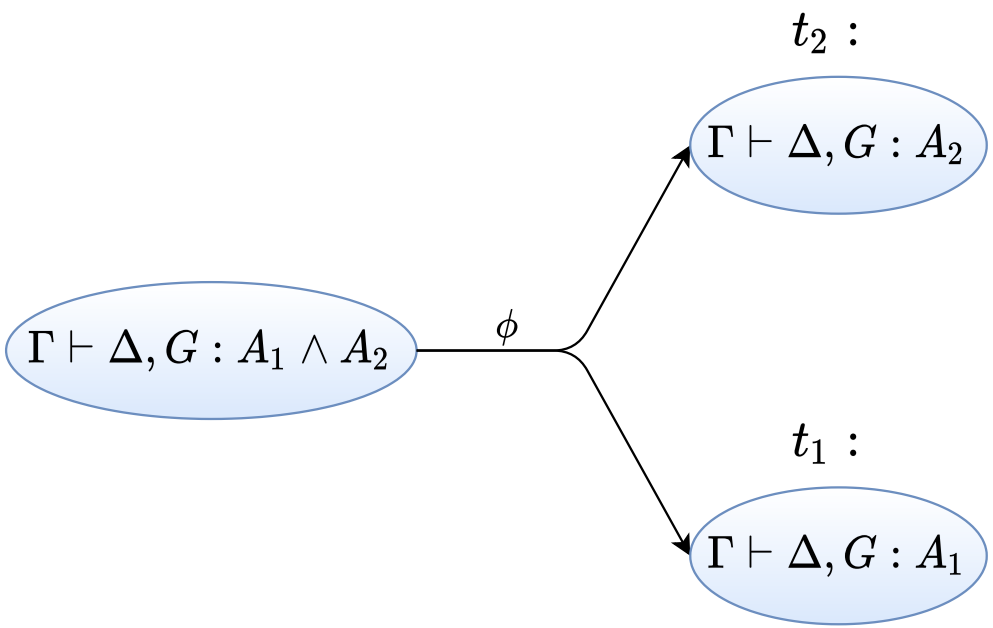
Compilation : remove `Construct` certificate





Naming tasks

Allow reuse and reordering of resulting tasks by naming them :



$[t_2; t_1], Split\ (G, Hole\ t_1, Hole\ t_2)$

Naming tasks : `destruct_disjunction`

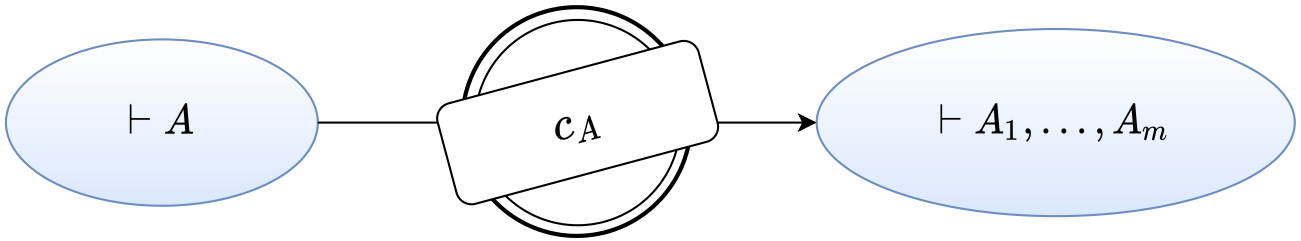
Apply `split_vc` transformation on $\Gamma \vdash \Delta, G : A \wedge B$

To simplify, assume the task is $\vdash A \wedge B$

Naming tasks : `destruct_disjunction`

Apply `split_vc` transformation on $\Gamma \vdash \Delta, G : A \wedge B$

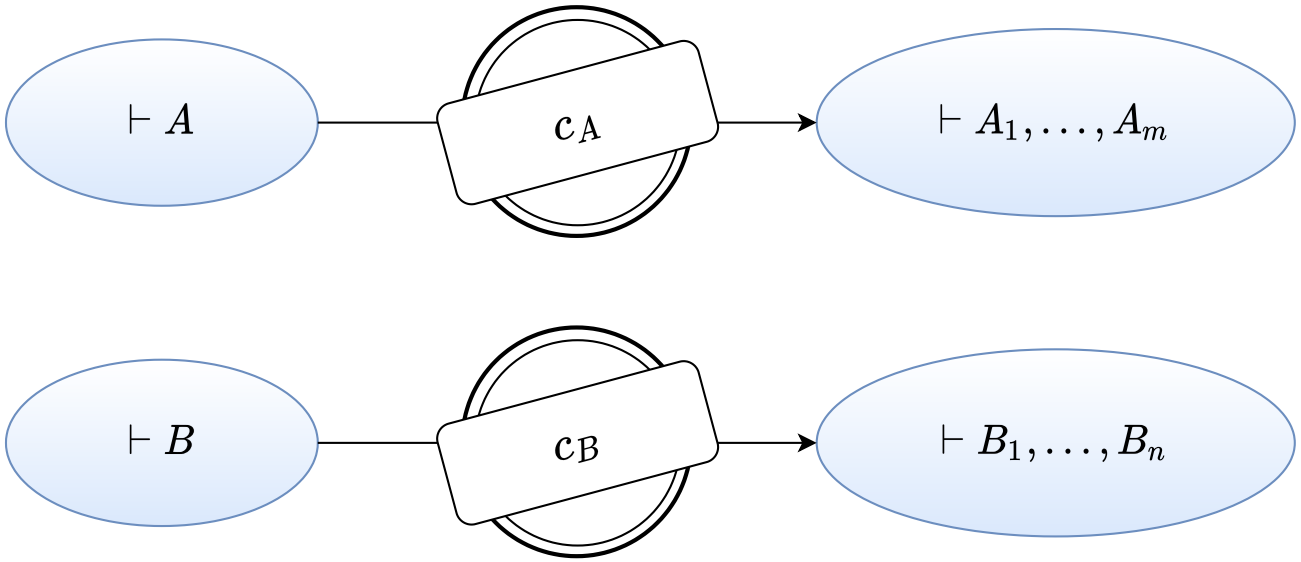
To simplify, assume the task is $\vdash A \wedge B$



Naming tasks : `destruct_disjunction`

Apply `split_vc` transformation on $\Gamma \vdash \Delta, G : A \wedge B$

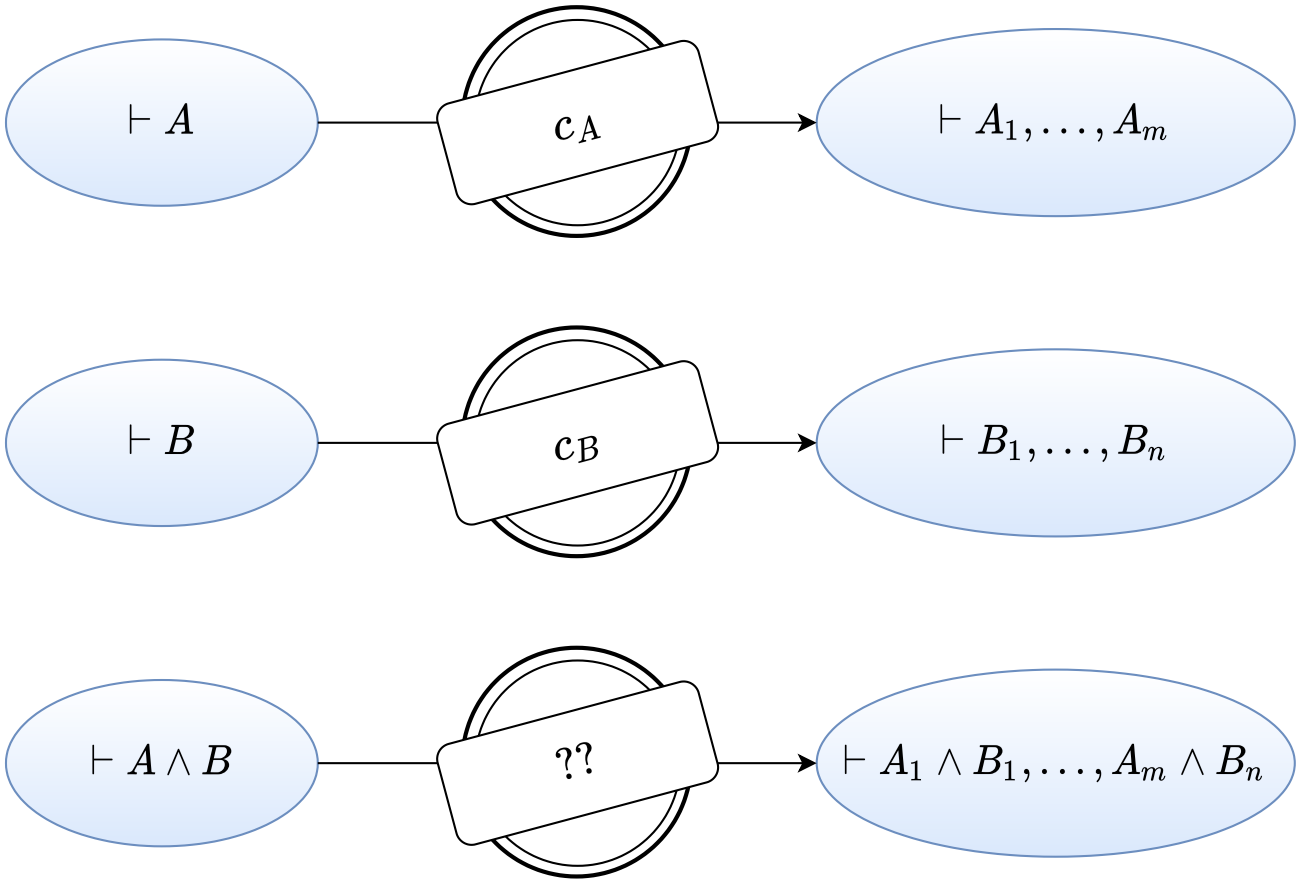
To simplify, assume the task is $\vdash A \wedge B$



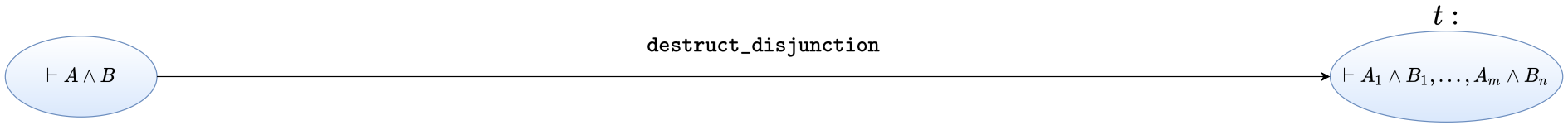
Naming tasks : `destruct_disjunction`

Apply `split_vc` transformation on $\Gamma \vdash \Delta, G : A \wedge B$

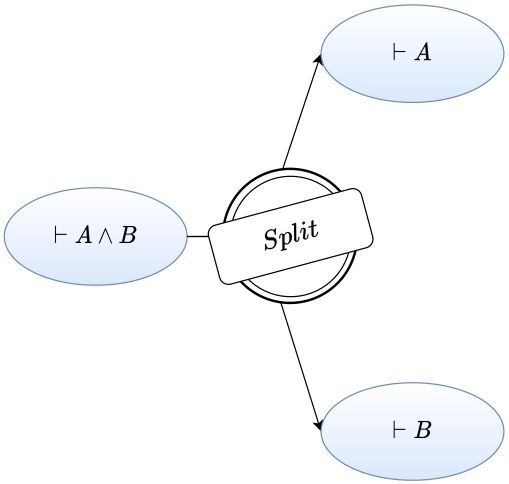
To simplify, assume the task is $\vdash A \wedge B$



Naming tasks : destruct_disjunction



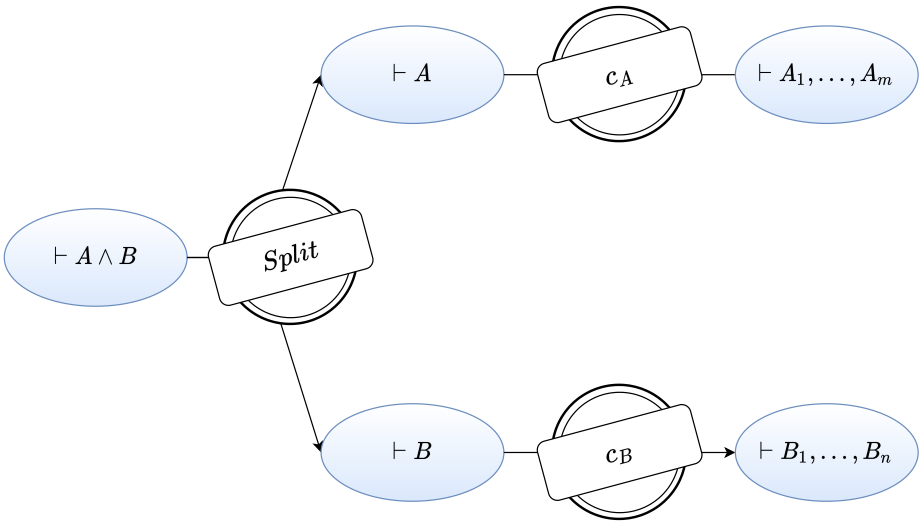
Naming tasks : destruct_disjunction



t :

$\vdash A_1 \wedge B_1, \ldots, A_m \wedge B_n$

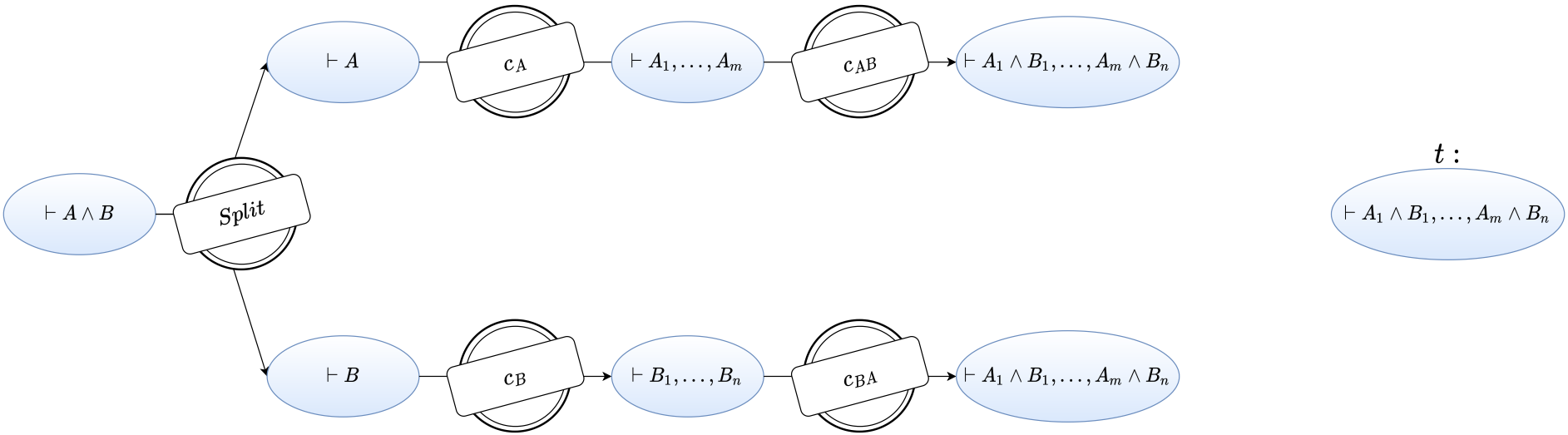
Naming tasks : destruct_disjunction



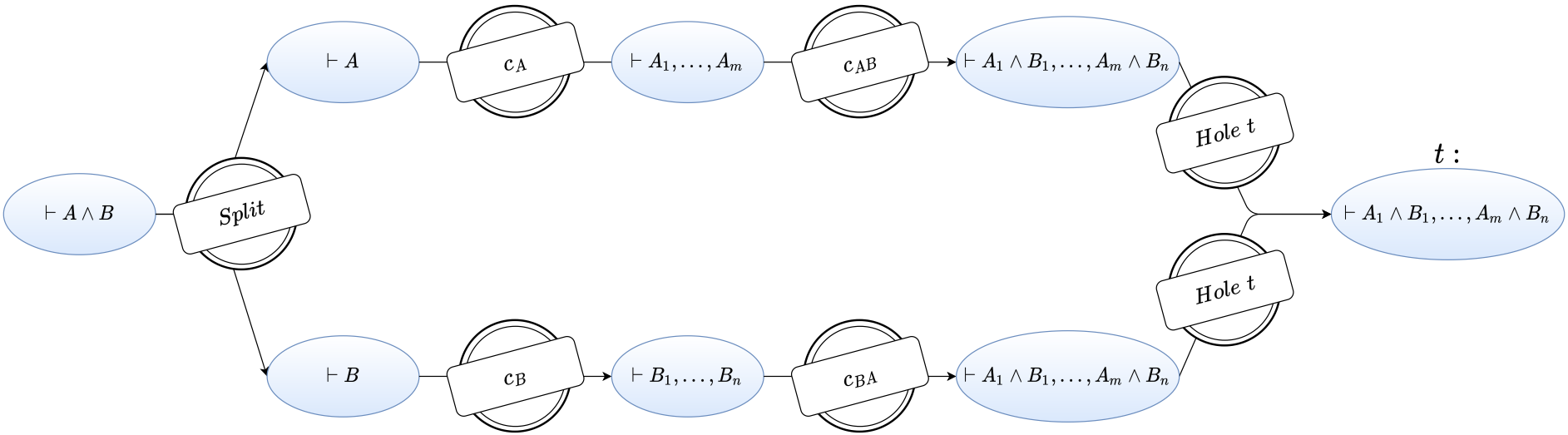
t :

$\vdash A_1 \wedge B_1, \ldots, A_m \wedge B_n$

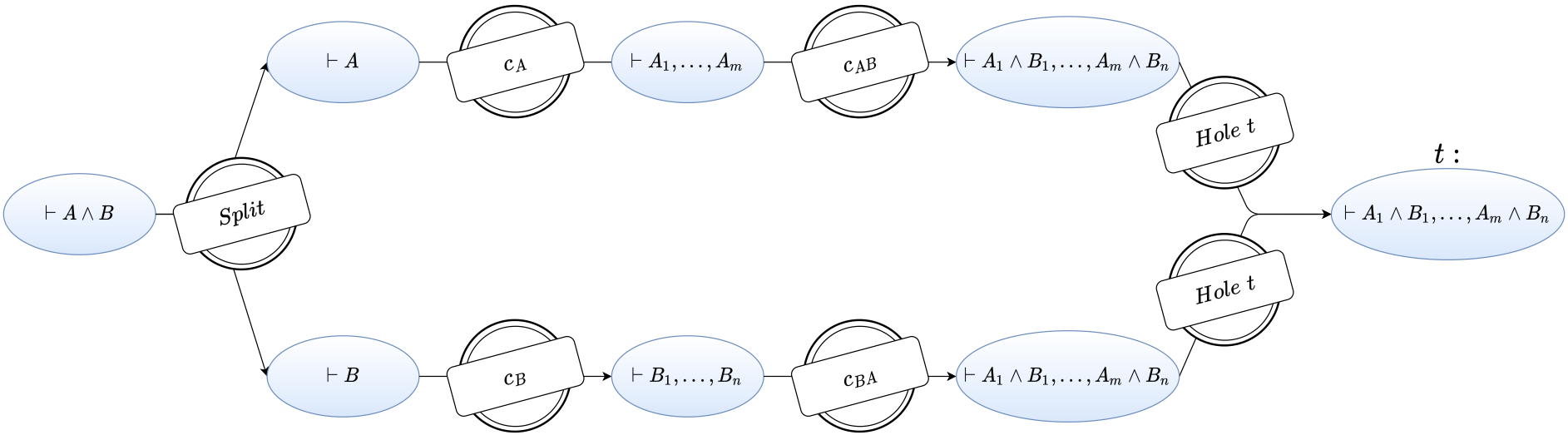
Naming tasks : destruct_disjunction



Naming tasks : destruct_disjunction



Naming tasks : destruct_disjunction



$$\dfrac{\dfrac{\dfrac{\vdash A_1 \wedge B_1, ..., A_m \wedge B_n}{\vdash A_1, ..., A_m}\, c_{AB}}{\vdash A}\, c_A \qquad \dfrac{\dfrac{\vdash A_1 \wedge B_1, ..., A_m \wedge B_n}{\vdash B_1, ..., B_n}\, c_{BA}}{\vdash B}\, c_B}{\vdash A \wedge B}\, Split$$

Contributions

A *generic, fine-grained* method to make transformations certifying

Software :

- modular verification framework, including compilation of surface certificates
- 2 independant checkers
- transformation composition

Applications :

- $\sim$ 15 simple certifying transformations
- `blast`
- `rewrite` and `split_vc`
- validation of transformations by Dedukti
- pigeonhole principle
- validation of the `instantiate` example in Why3

Future work and ressources

Improve transformation certification :

- extend certification of `split_vc` and `rewrite`
- improve the certificate format with types, higher order
- compile and optimize surface certificates

Certify generation of verification conditions for imperative programs with aliases

Find more details in JFLA publication
→ `https://hal.archives-ouvertes.fr/hal-02384946`

Find latest work as a branch of Why3 at
→ `https://gitlab.inria.fr/why3/why3/tree/cert_split`