Certification de la génération et de la transformation d'obligations de preuve

Mi-parcours

10 juin 2020

Quentin Garchery

The `instantiate` example in Why3

Program with preconditions and a postcondition [1]:

```
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 preconditions and a postcondition [1]:

```
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 preconditions and a postcondition [1]:

```
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]
```

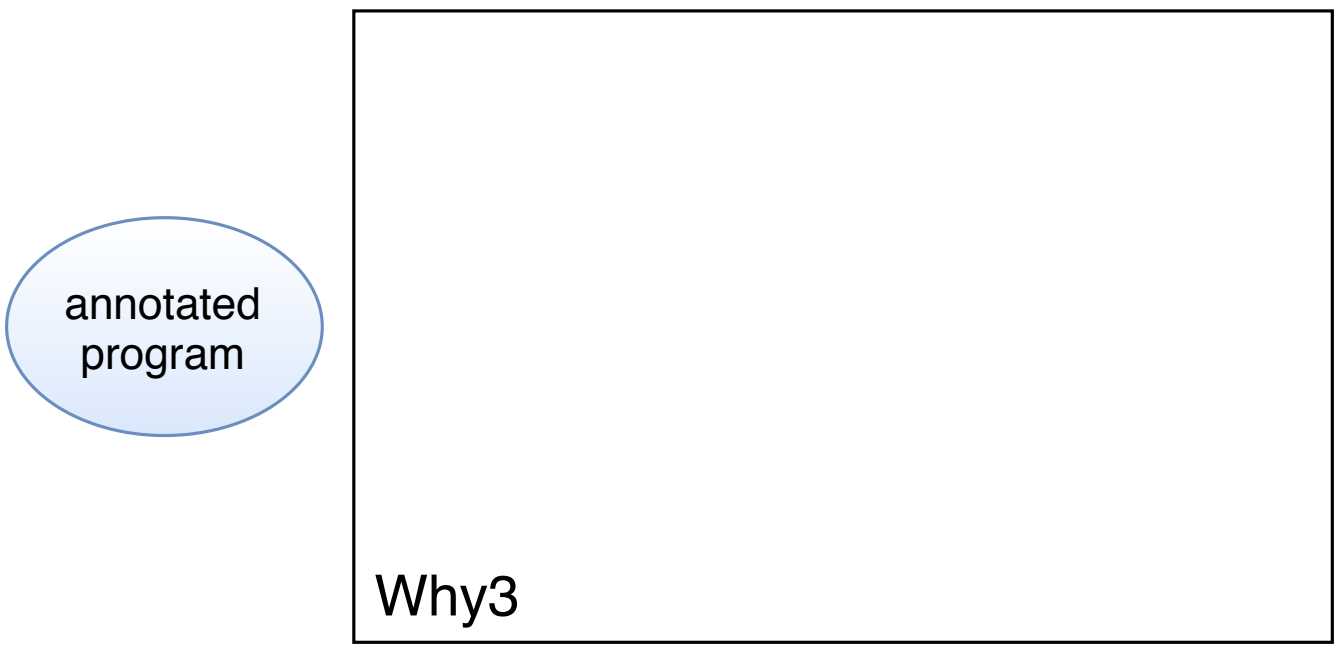
Transform postcondition (with `instantiate`) :

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

Overview of Why3

Why3 step by step :

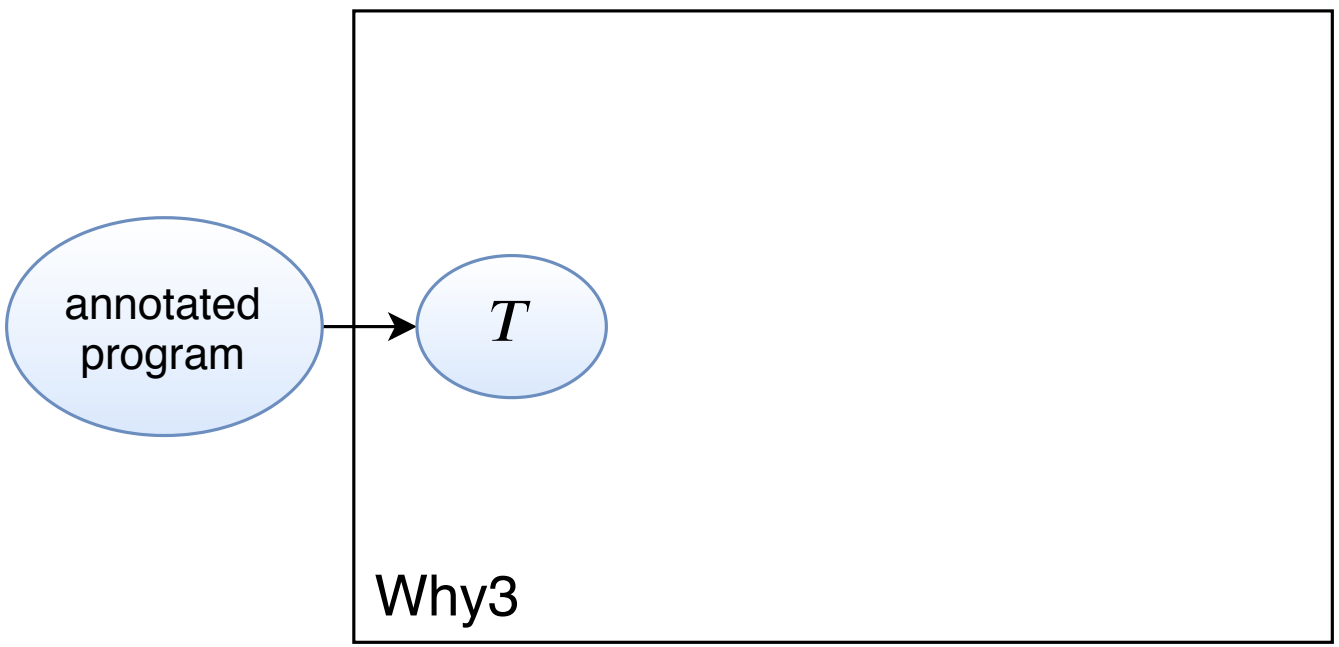
- initial task generation
- logical transformations
- call to automatic theorem provers



Overview of Why3

Why3 step by step :

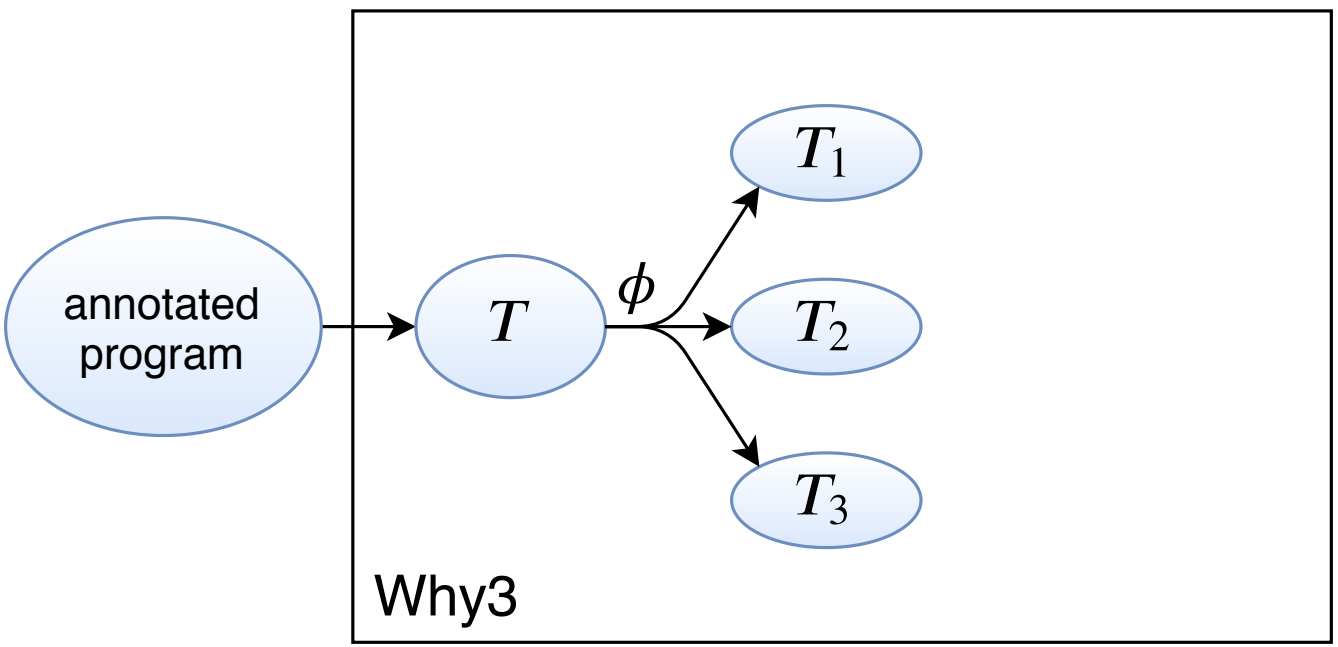
- initial task generation
- logical transformations
- call to automatic theorem provers



Overview of Why3

Why3 step by step :

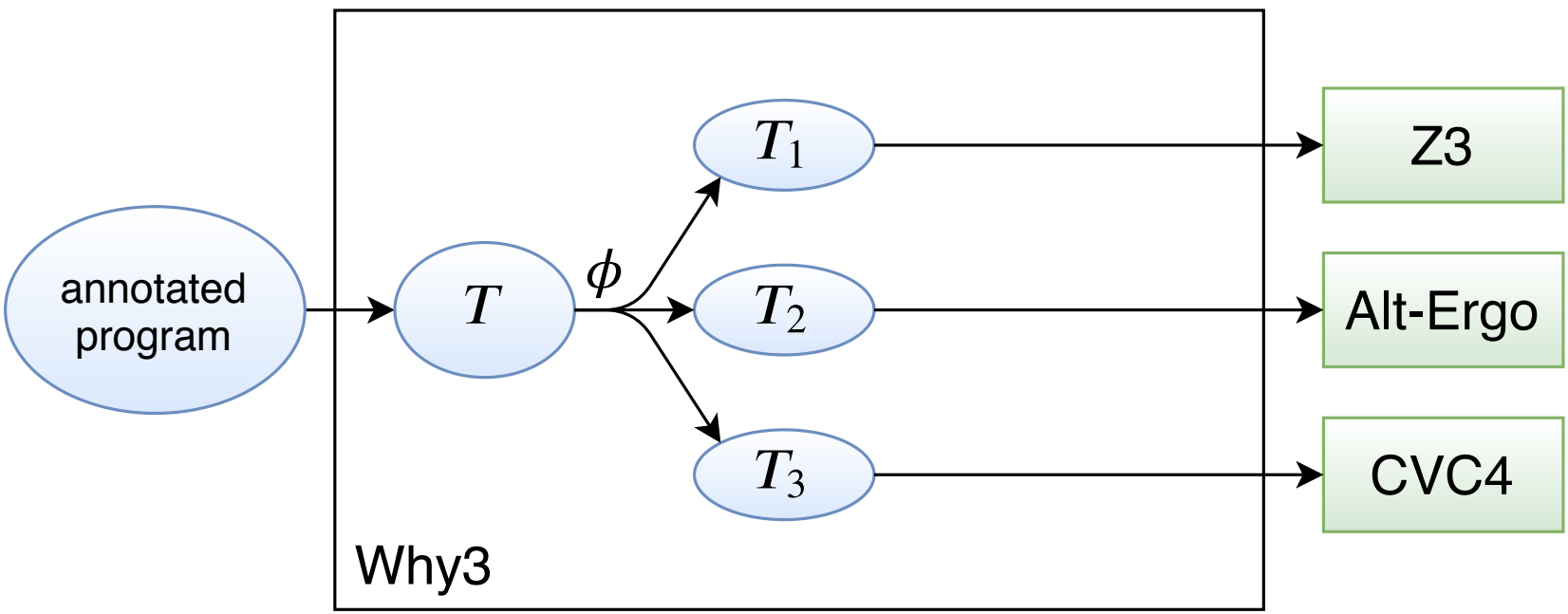
- initial task generation
- logical transformations
- call to automatic theorem provers



Overview of Why3

Why3 step by step :

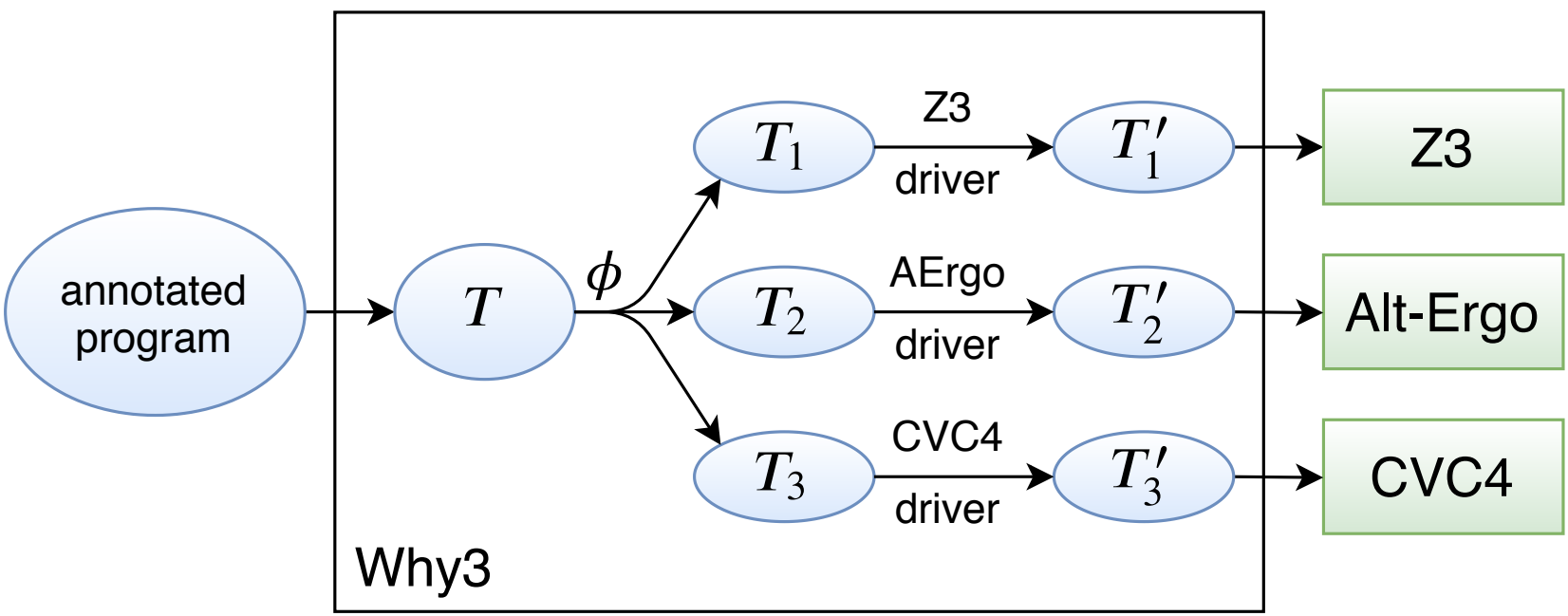
- initial task generation
- logical transformations
- call to automatic theorem provers



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

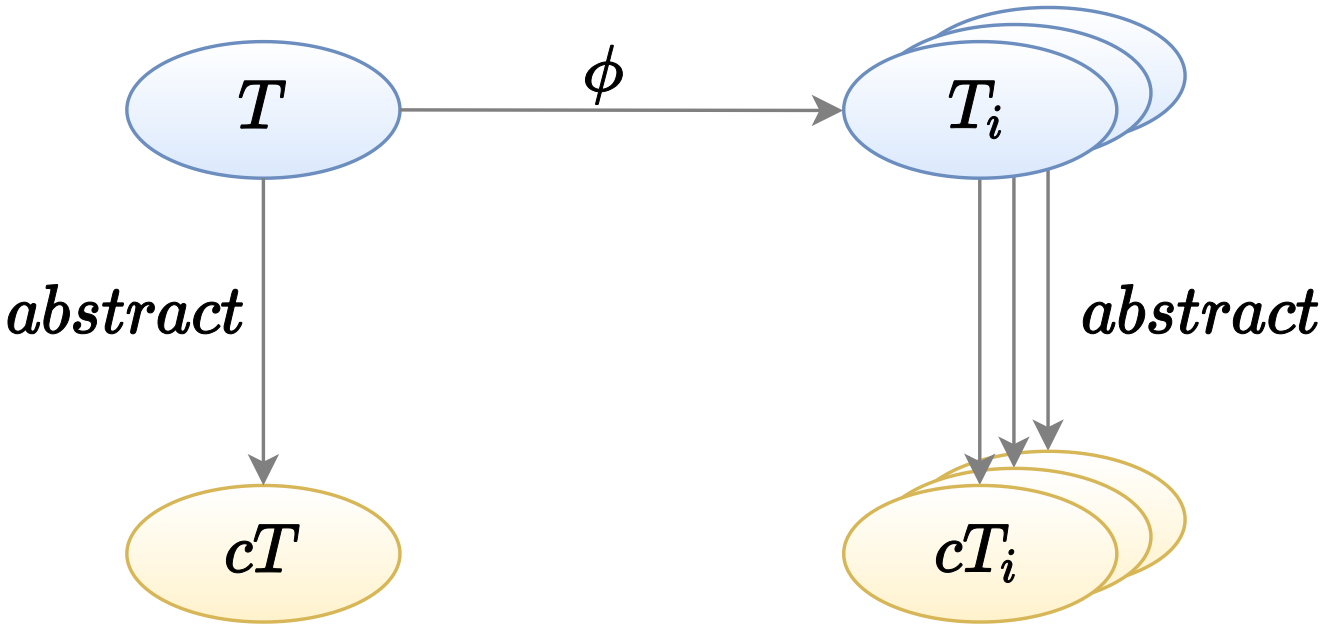
Plan

Task abstraction : motivation

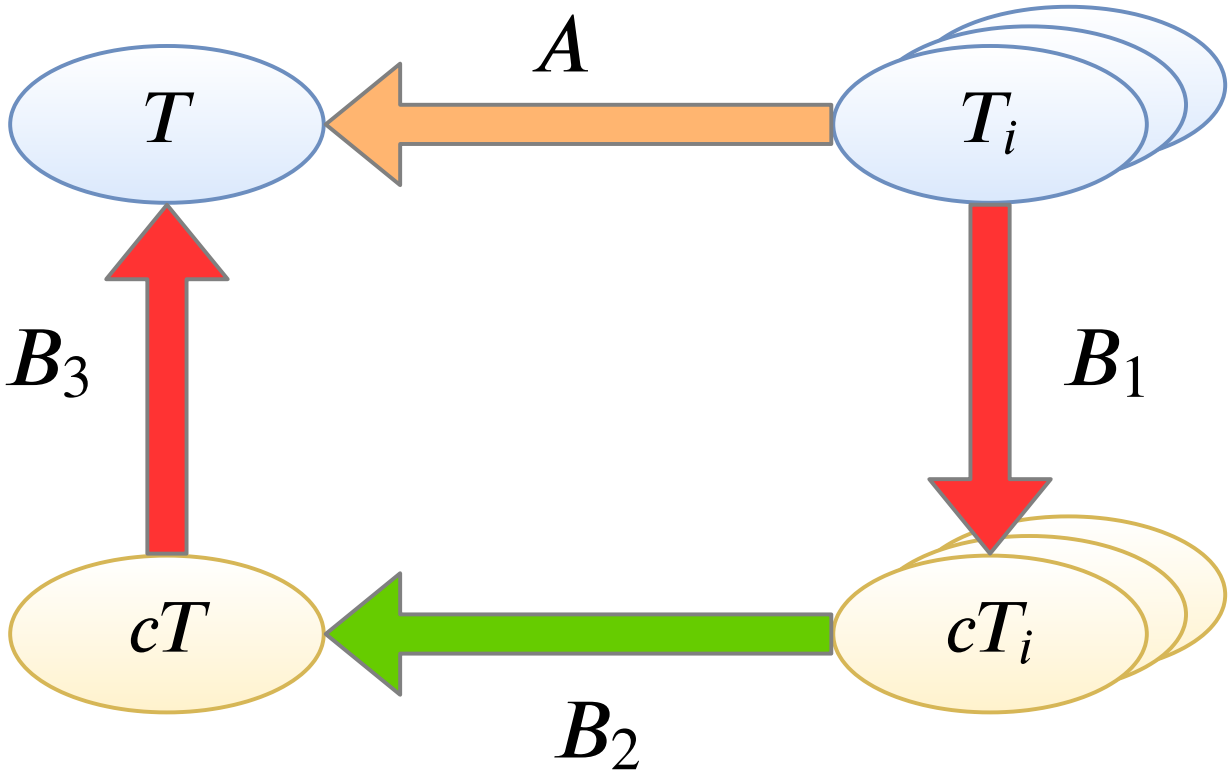
Proof tasks with meta-data, type and signature

$\hookrightarrow$ 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

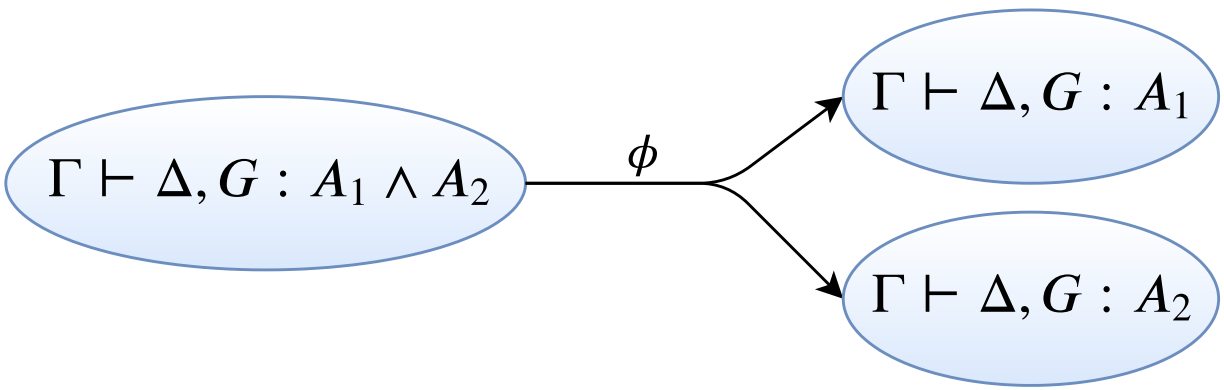


$\Longrightarrow$ we need equisatisfiability between a task and its abstraction

Certificates

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

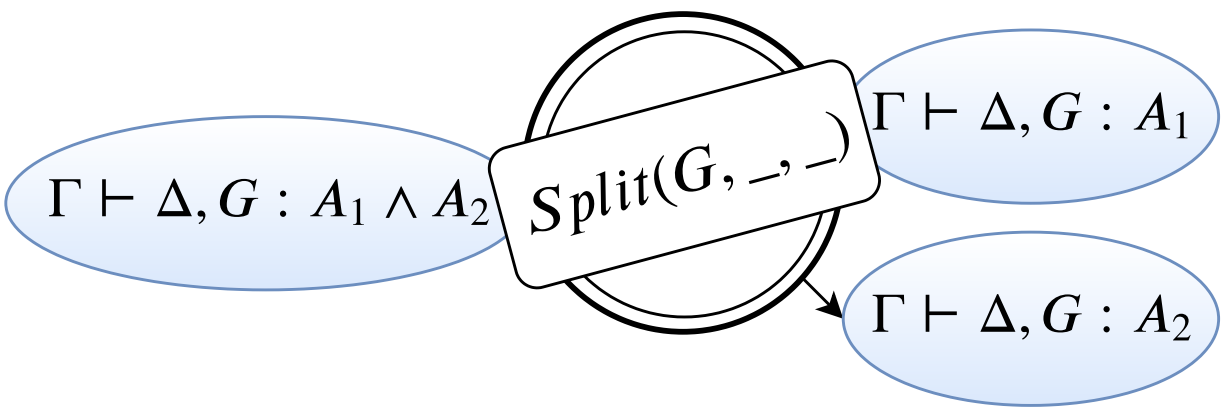
A constructor $\Leftrightarrow$ An elementary transformation



Certificates

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

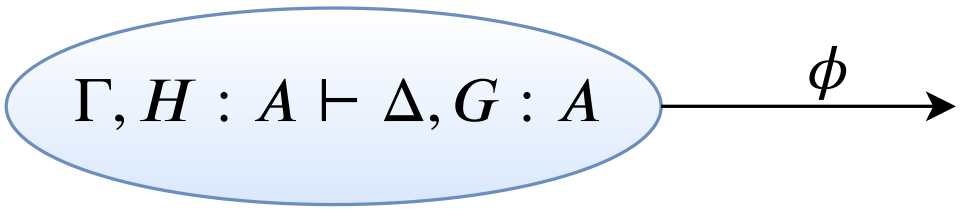
A constructor $\Leftrightarrow$ An elementary transformation



Certificates

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

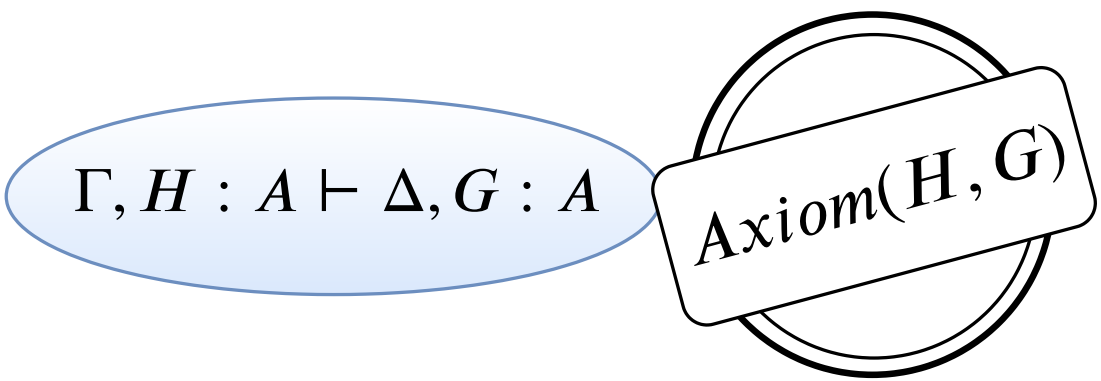
A constructor $\Leftrightarrow$ An elementary transformation



Certificates

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

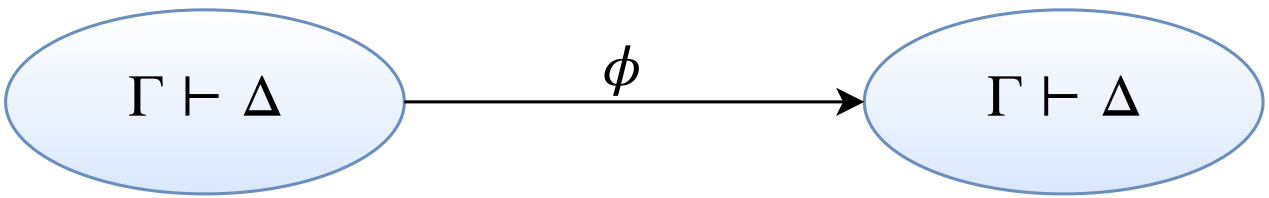
A constructor $\Leftrightarrow$ An elementary transformation



Certificates

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

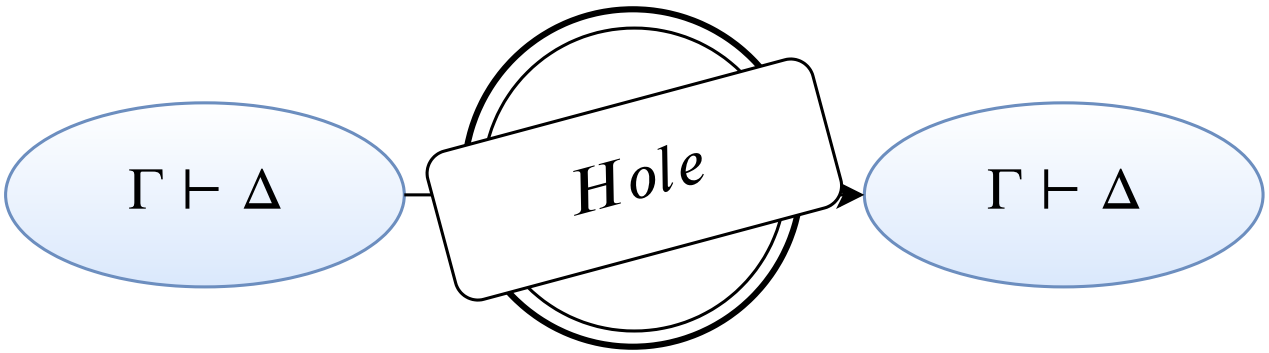
A constructor $\Leftrightarrow$ An elementary transformation



Certificates

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

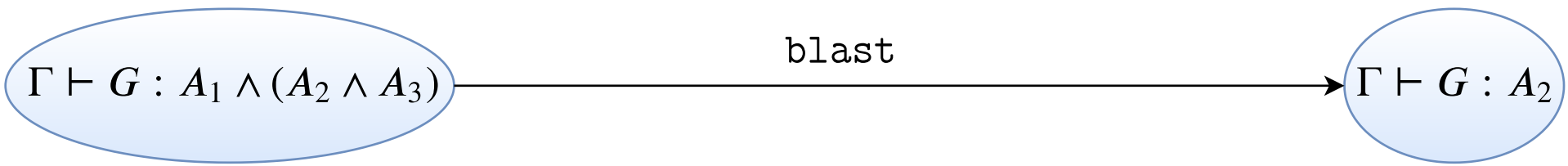
A constructor $\Leftrightarrow$ An elementary transformation



An example : the `blast` transformation

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

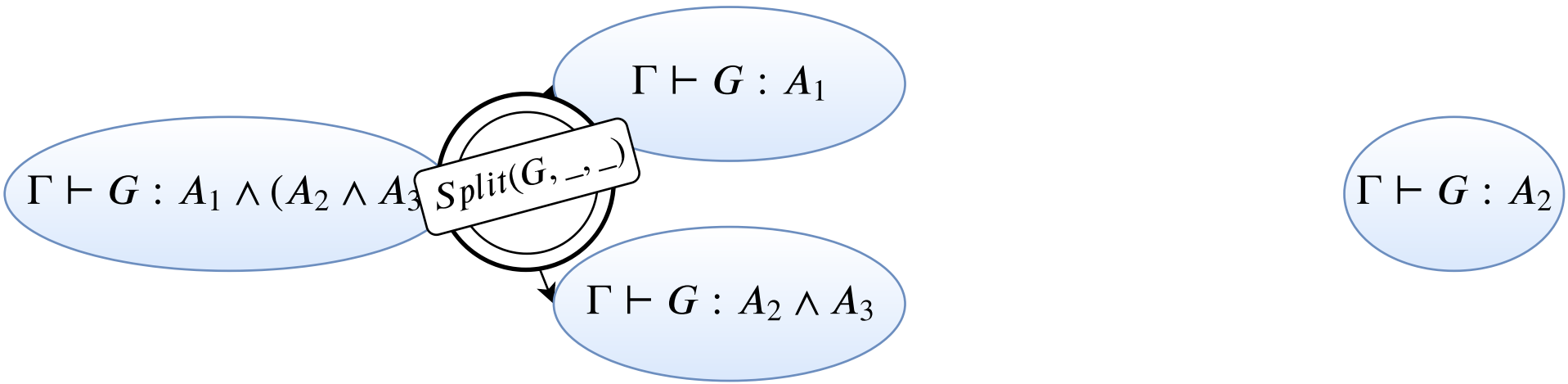
$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$



An example : the `blast` transformation

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

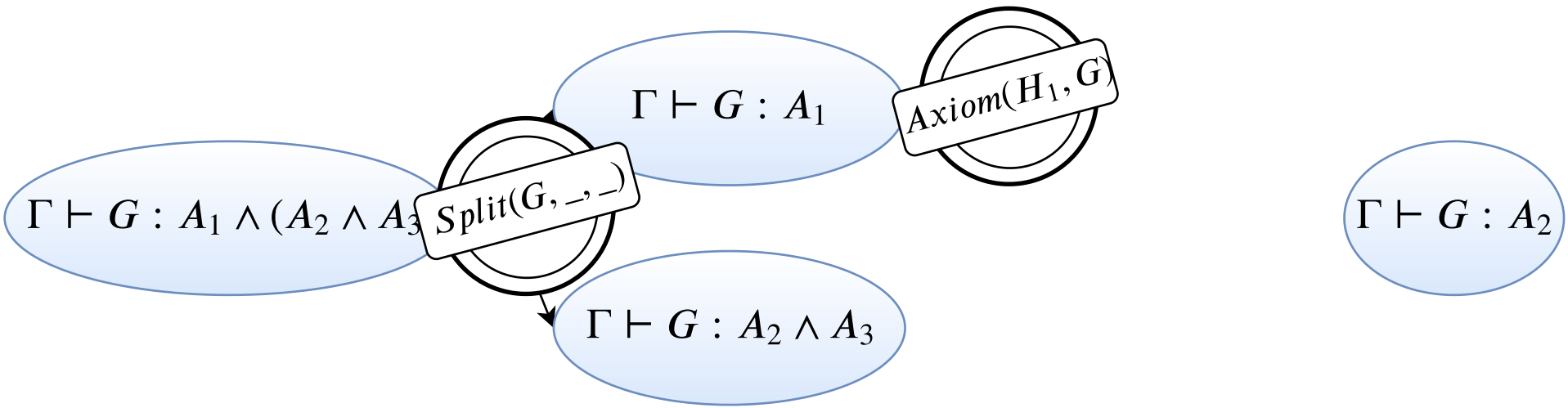
$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$



An example : the `blast` transformation

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

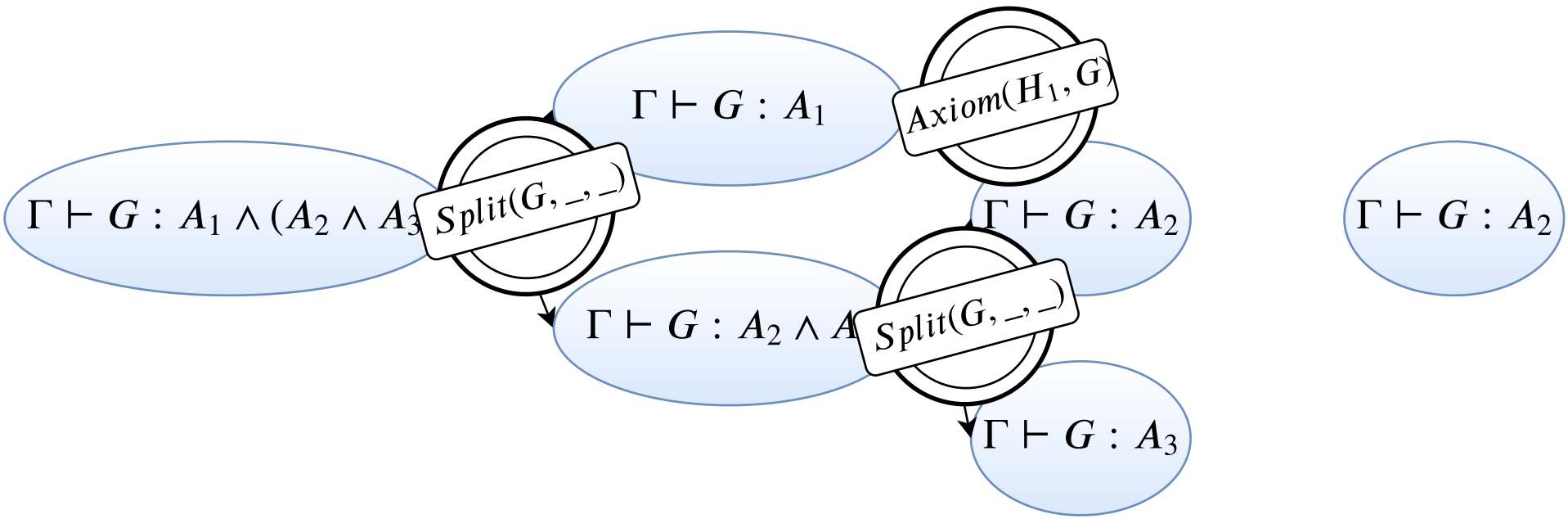
$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$



An example : the `blast` transformation

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

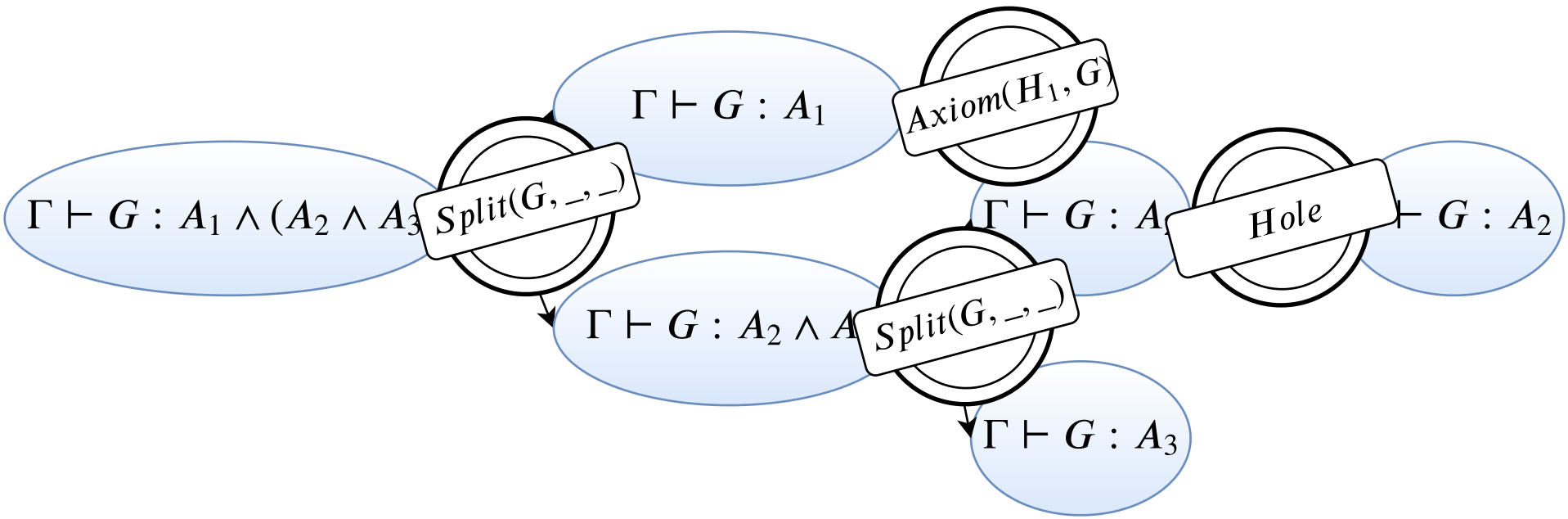
$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$



An example : the `blast` transformation

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

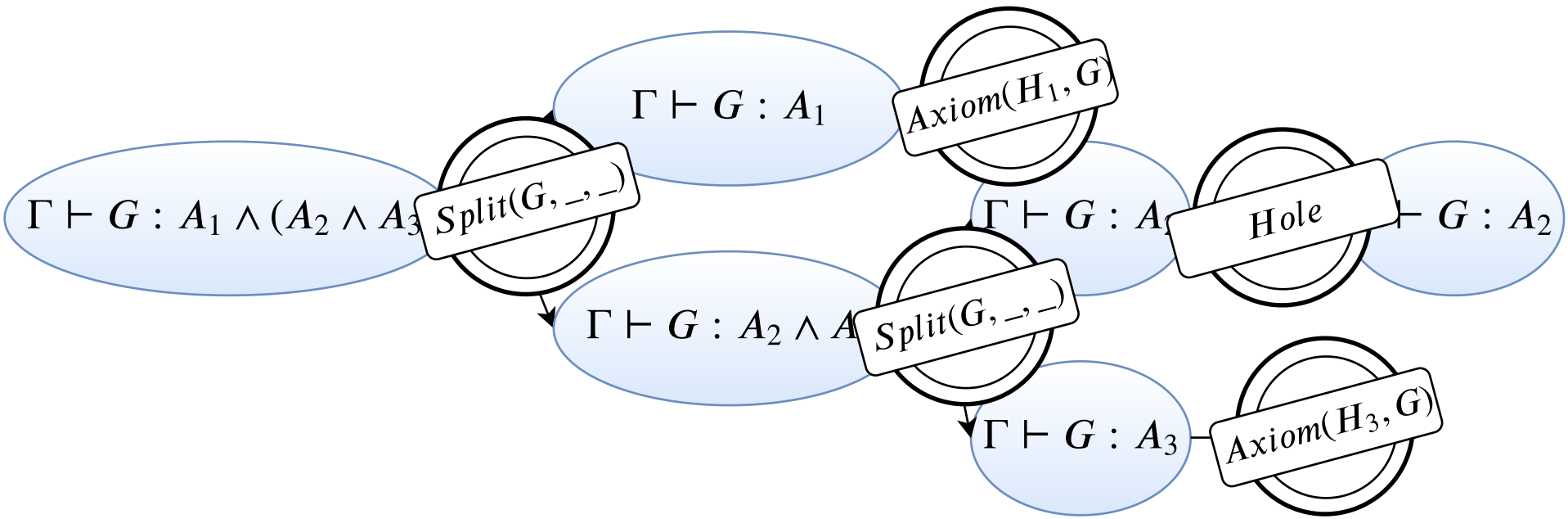
$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$



An example : the `blast` transformation

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

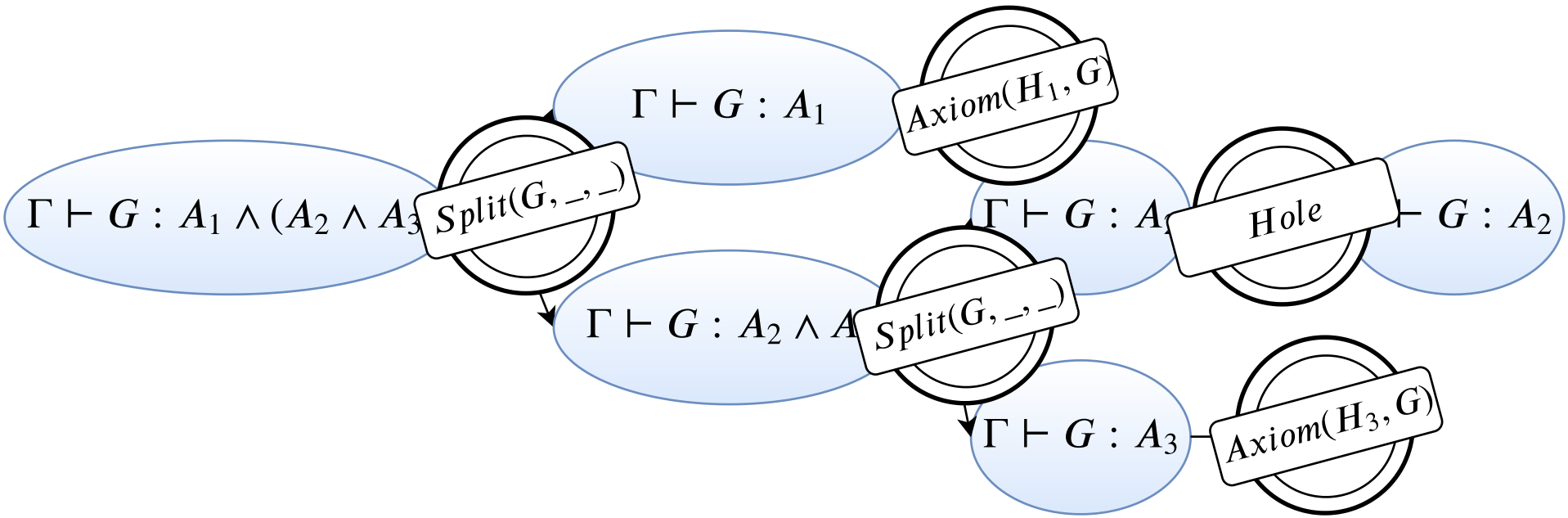
$T := \Gamma \vdash G : A_1 \wedge (A_2 \wedge A_3)$



An example : the `blast` transformation

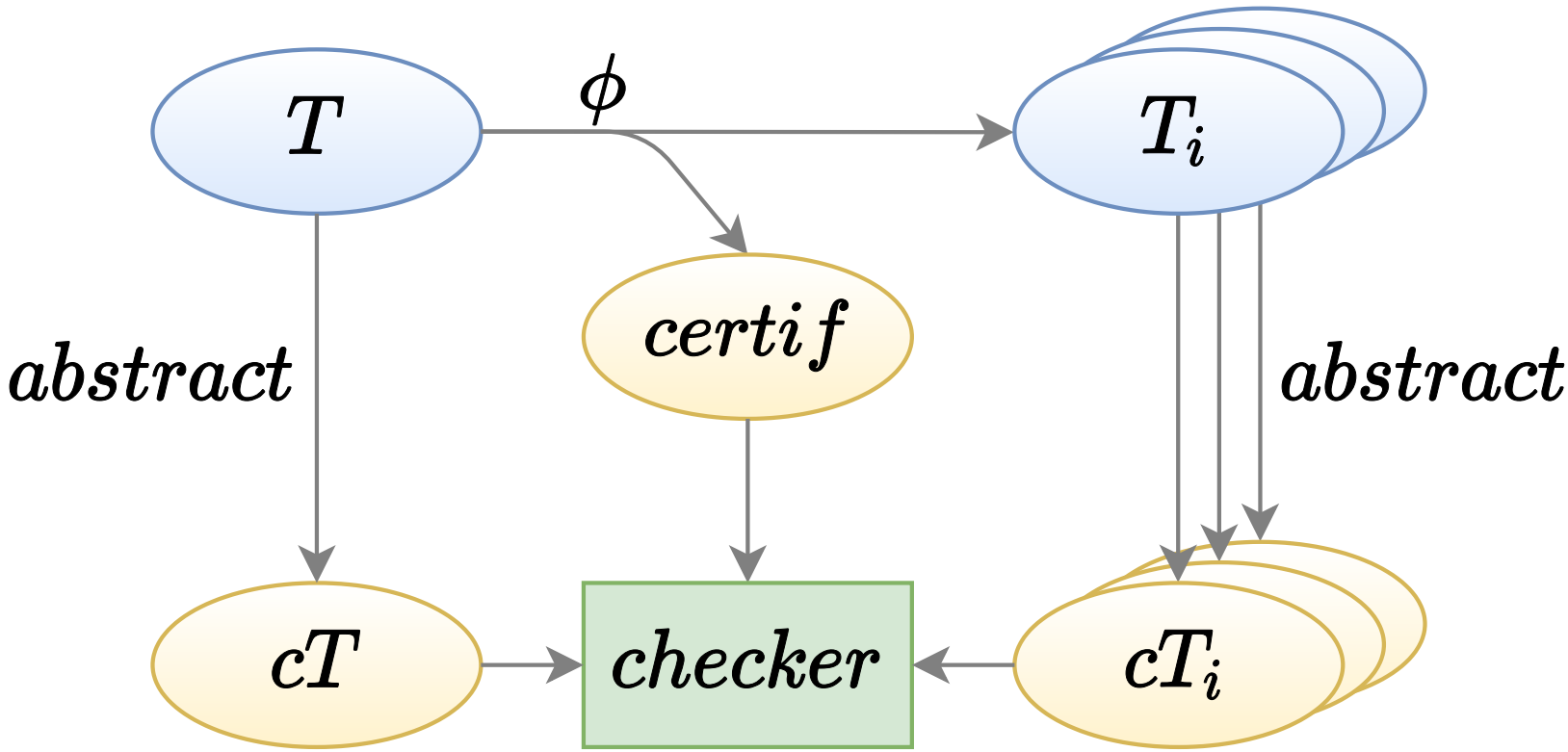
Pose : $\Gamma := H_1 : A_1, H_3 : A_3$

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



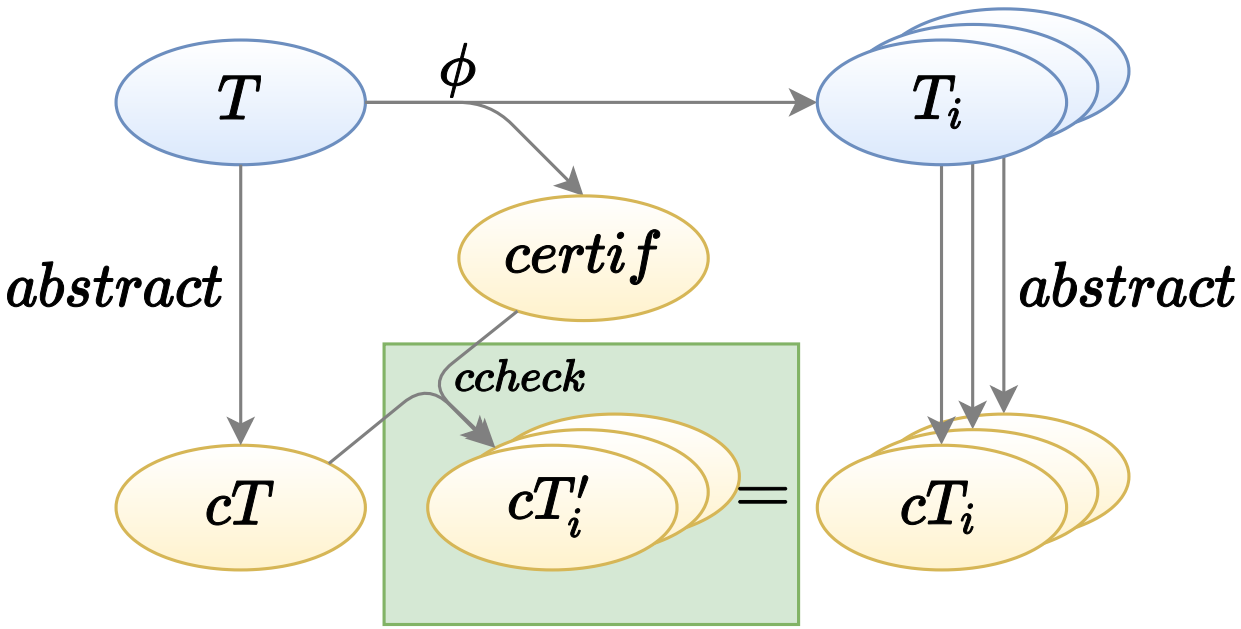
$Split\,(G, Axiom(H_1, G), Split(G, Hole, Axiom(H_3, G)))$

Framework for certificates



OCaml checker

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

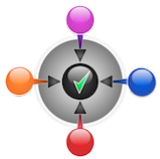


Correction of *ccheck*

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

Dedukti checker : overview

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



Dedukti [2] : universal proof checker, Curry-Howard, extensible conversion

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 :

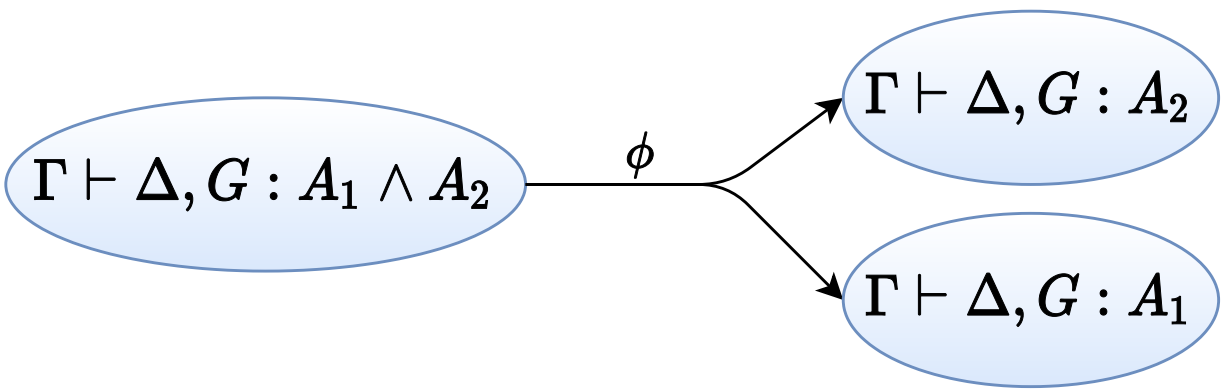
↳ find context of application, formulas employed ...

Compiling and improving certificates

Surface certificates → Kernel certificates :

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

Allow reuse and reordering of resulting tasks by naming them :



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

Resources

2 JFLA publications, more details about certifying transformation :
→ `https://hal.archives-ouvertes.fr/hal-02384946`

Available as a branch of Why3, latest work at :
→ `https://gitlab.inria.fr/why3/why3/tree/cert_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

Task generation : the alias problem

Why3's language has records with mutable fields and allows aliases

⚠ Hoare rules do not apply

```
val random unit : bool

type record = { mutable v : int }

let main : unit =
  let r1 = { v = 2 } in
  let r2 = r1 in
  if random () then r1.v ← r1.v + 1
  else r2.v ← r2.v + 1;
  assert { r1.v = 3 }
```

Task generation : towards a solution

Bibliography and future work :

Why3 solves the alias problem using a type system with regions [Jean-Christophe Filliâtre, Léon Gondelman, Andrei Paskevich. *A Pragmatic Type System for Deductive Verification.* 2016.]

First translate into another language
[Jean-Christophe Filliâtre. *Preuve de programmes impératifs en théorie des types.* 1999.]

Then certify task generation on this simpler language
[Andrei Paskevich. *Continuation passing as an abstract syntax for deductive verification.*]

Future work

Improve transformation certification :

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

↪ next few months

Certify task generation
↪ from now on