Le compilateur vérifié CompCert

Étude de l'impact d'une représentation par blocs

Odile Radet

sous la supervision de

David Pichardie et Jean-Christophe Léchenet

Équipe Celtique, Inria Rennes

ENS Rennes, Soutenance de stage de L3

30/08/2019

Pourquoi un compilateur vérifié ?

Traduttore, traditore.

Traduire, c'est trahir.

Dicton italien

Analogie courante

Compilateur $\sim$ Traducteur

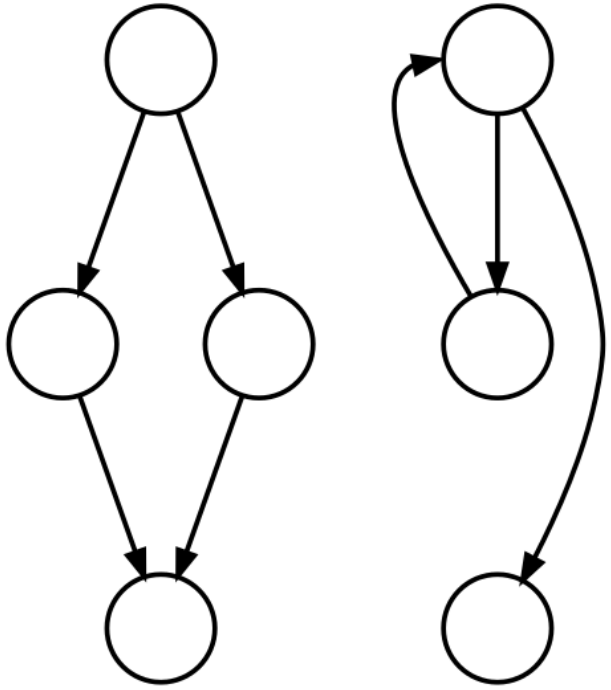
(code source $\rightarrow$ code objet)

Propriété souhaitée

Pas de trahison de la part du compilateur !

Objectif général

- CompCert :
  - compilateur vérifié
  - pour du C
  - écrit en Coq
- Design *inhabituel* pour les graphes de flot de contrôle



if then else while

Sommaire

Qu'est-ce que Coq ?

Un assistant de preuves interactif

- Écrire des programmes
- Écrire des preuves
- *Vérifier* formellement leur correction

Historique

- Logiciel développé par l'INRIA, initié par Gérard Pierre Huet et Thierry Coquand
- Première version : 1984
- Applications : 4 couleurs, Feit-Thompson, CompCert...

Qu'est-ce que Coq ?

En théorie

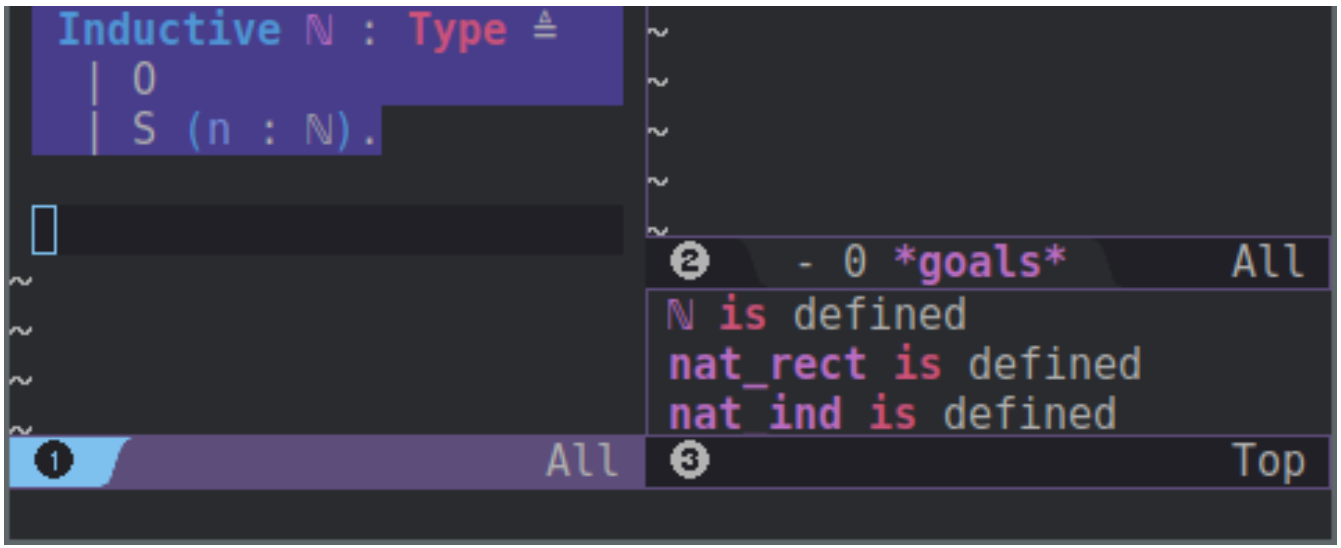
- Calcul des constructions (CoC)
- Isomorphisme de Curry-Howard : programme $\leftrightarrow$ preuve

En pratique

- Langage fonctionnel
- Succession de "tactiques"
- Environnement de développement dédié (CoqIDE)

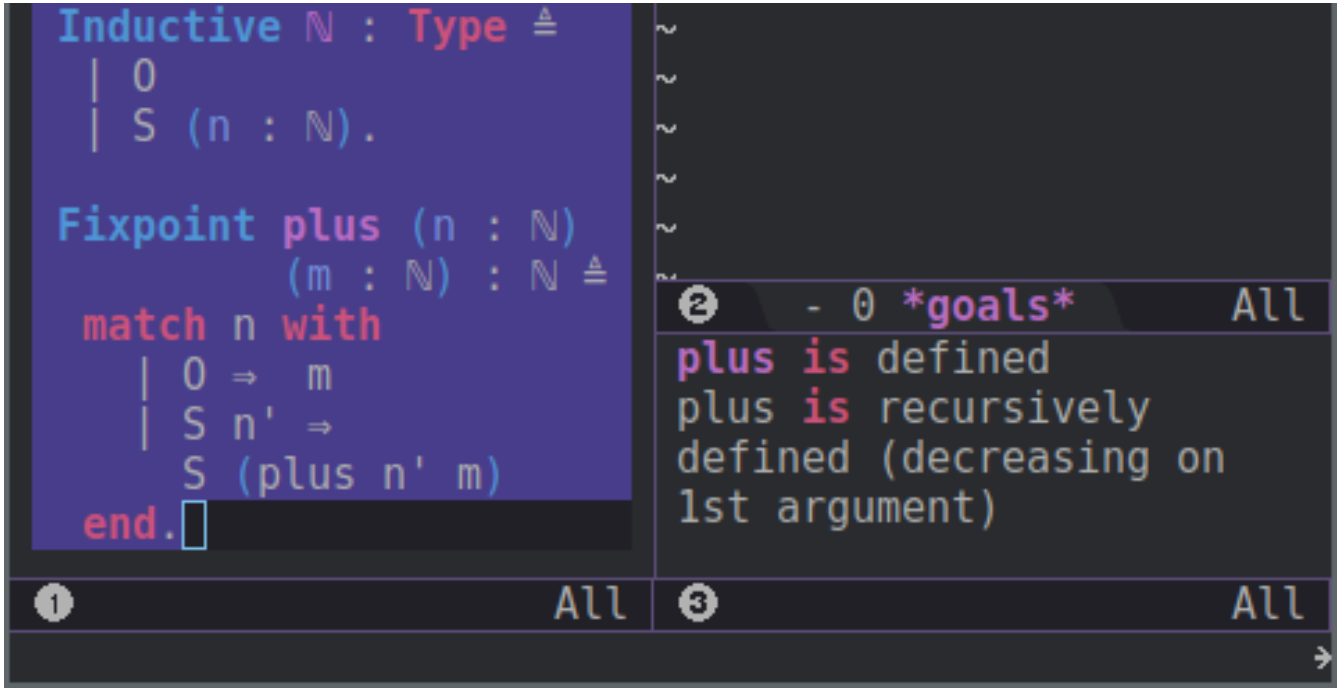
Un exemple : les entiers naturels

Définition du type "entier naturel"



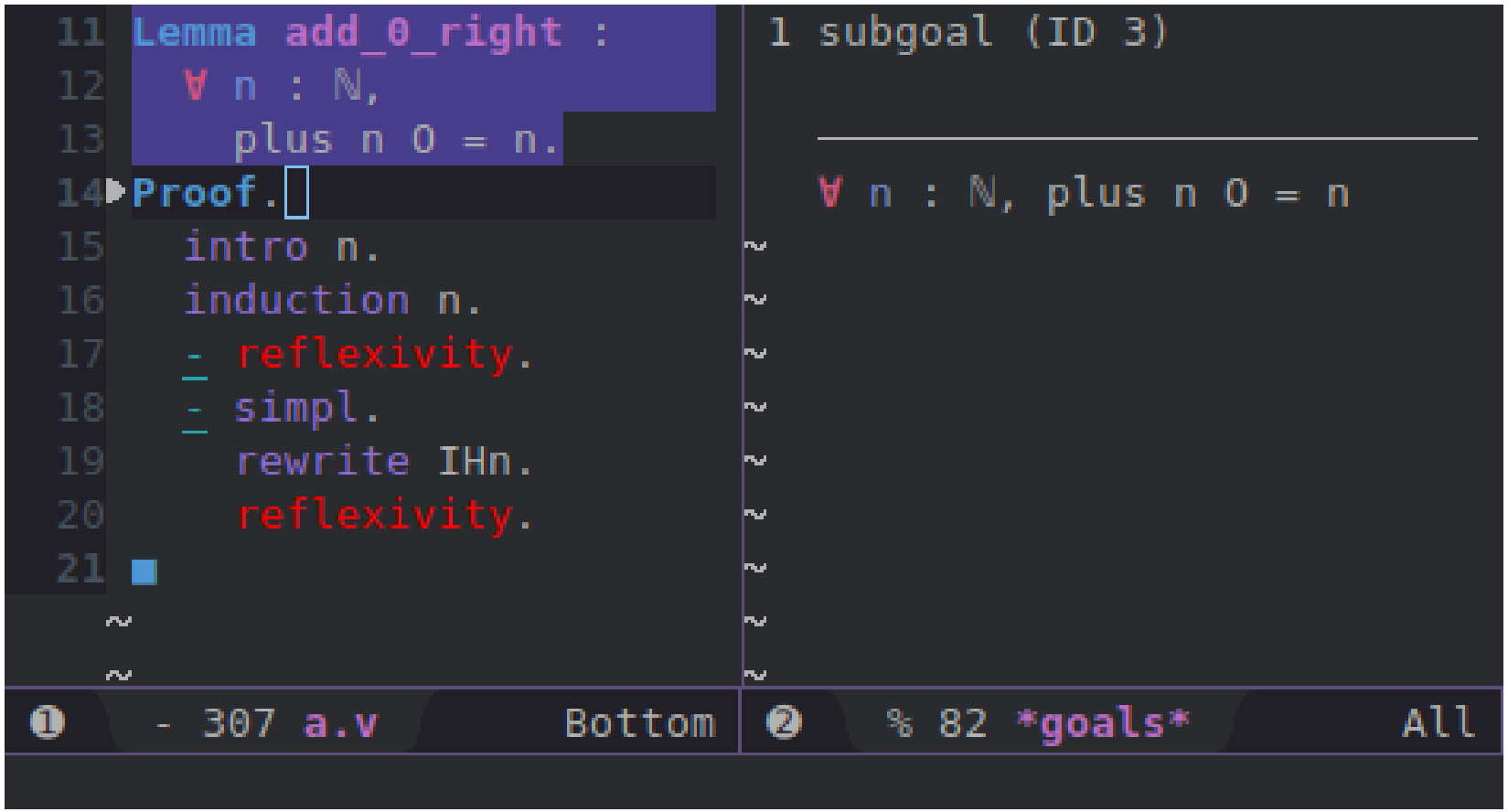
Un exemple : les entiers naturels

Définition de l'addition



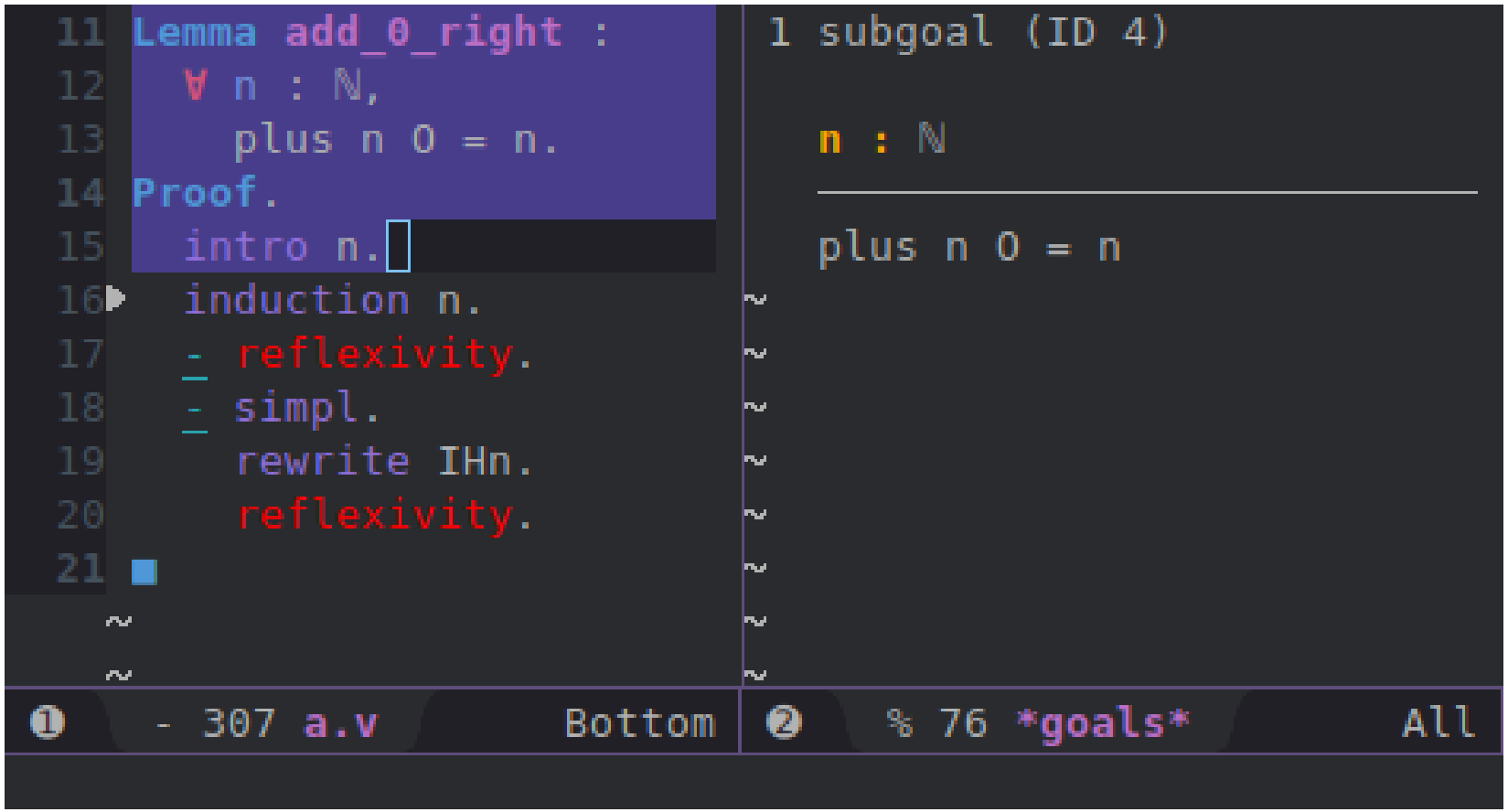
Un exemple : les entiers naturels

Preuve d'une propriété simple



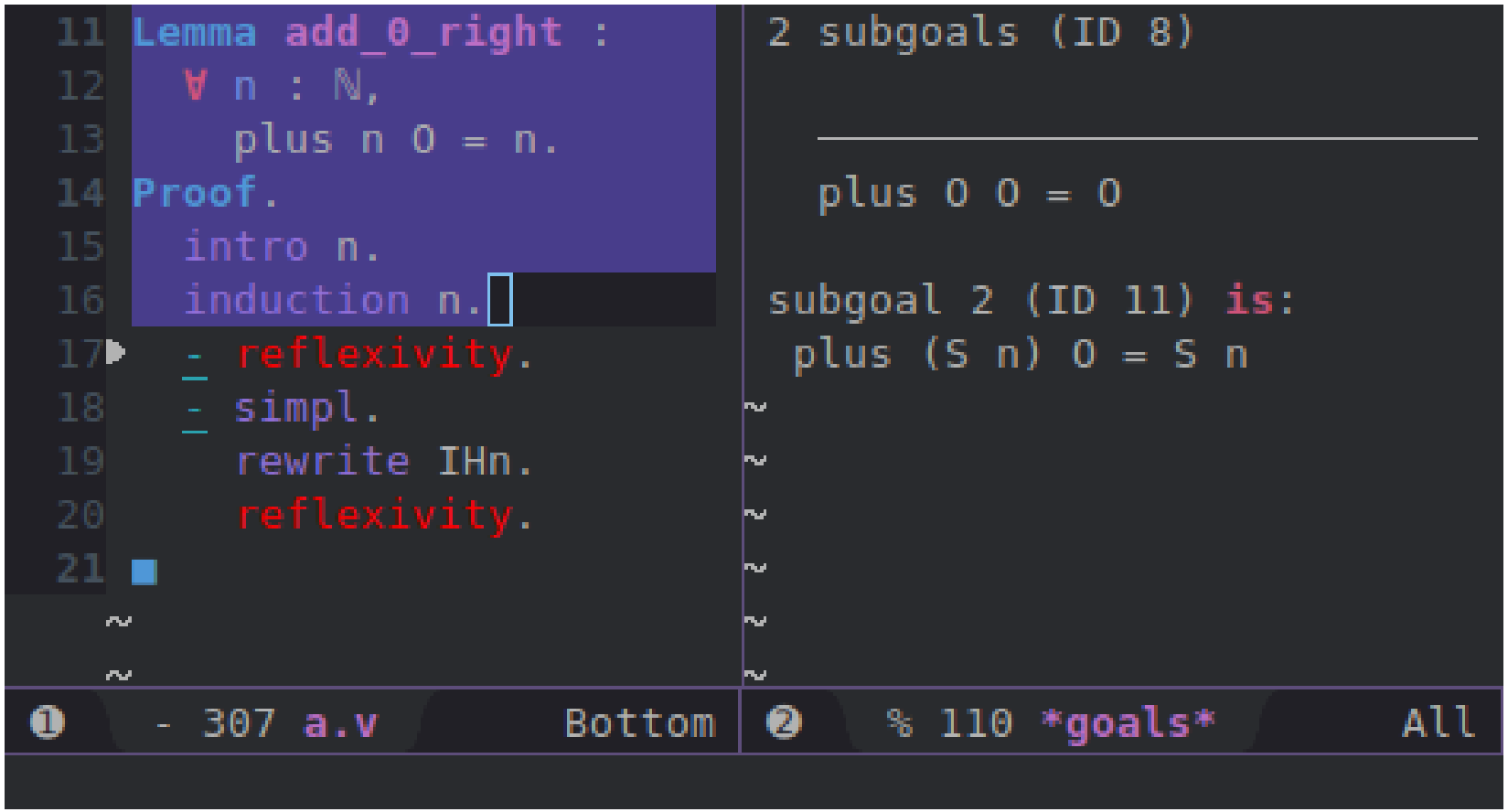
Un exemple : les entiers naturels

Preuve d'une propriété simple



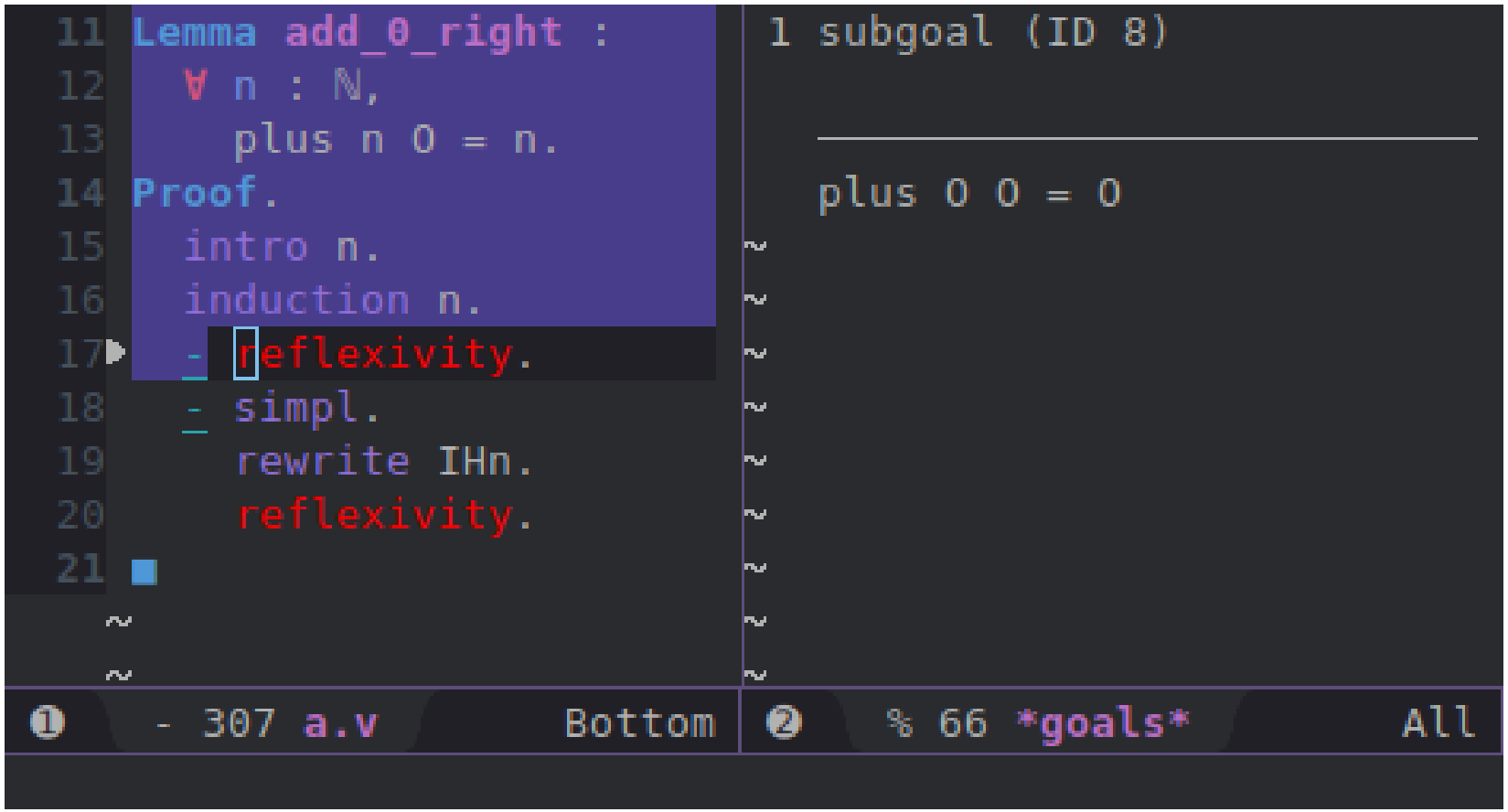
Un exemple : les entiers naturels

Preuve d'une propriété simple



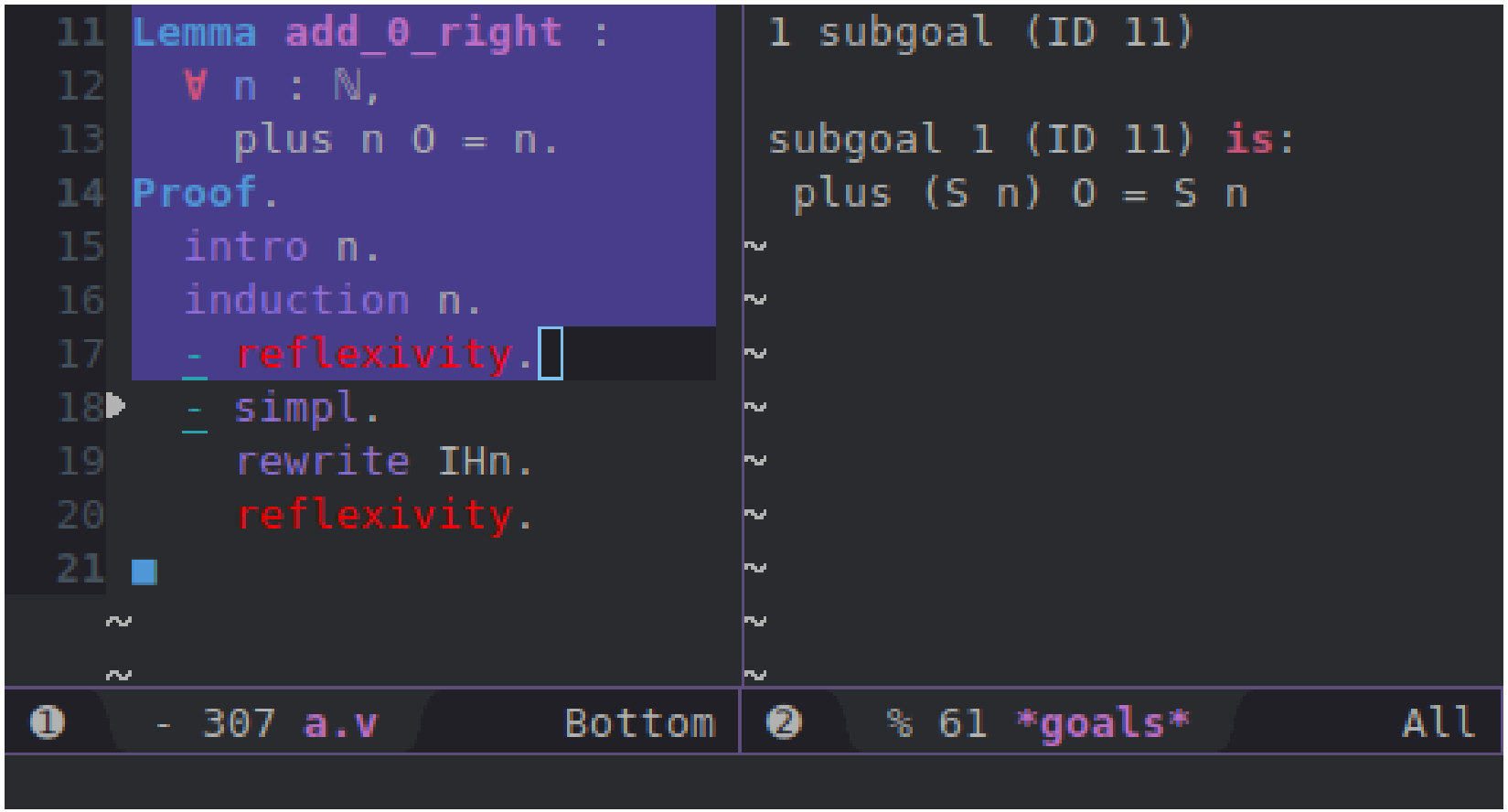
Un exemple : les entiers naturels

Preuve d'une propriété simple



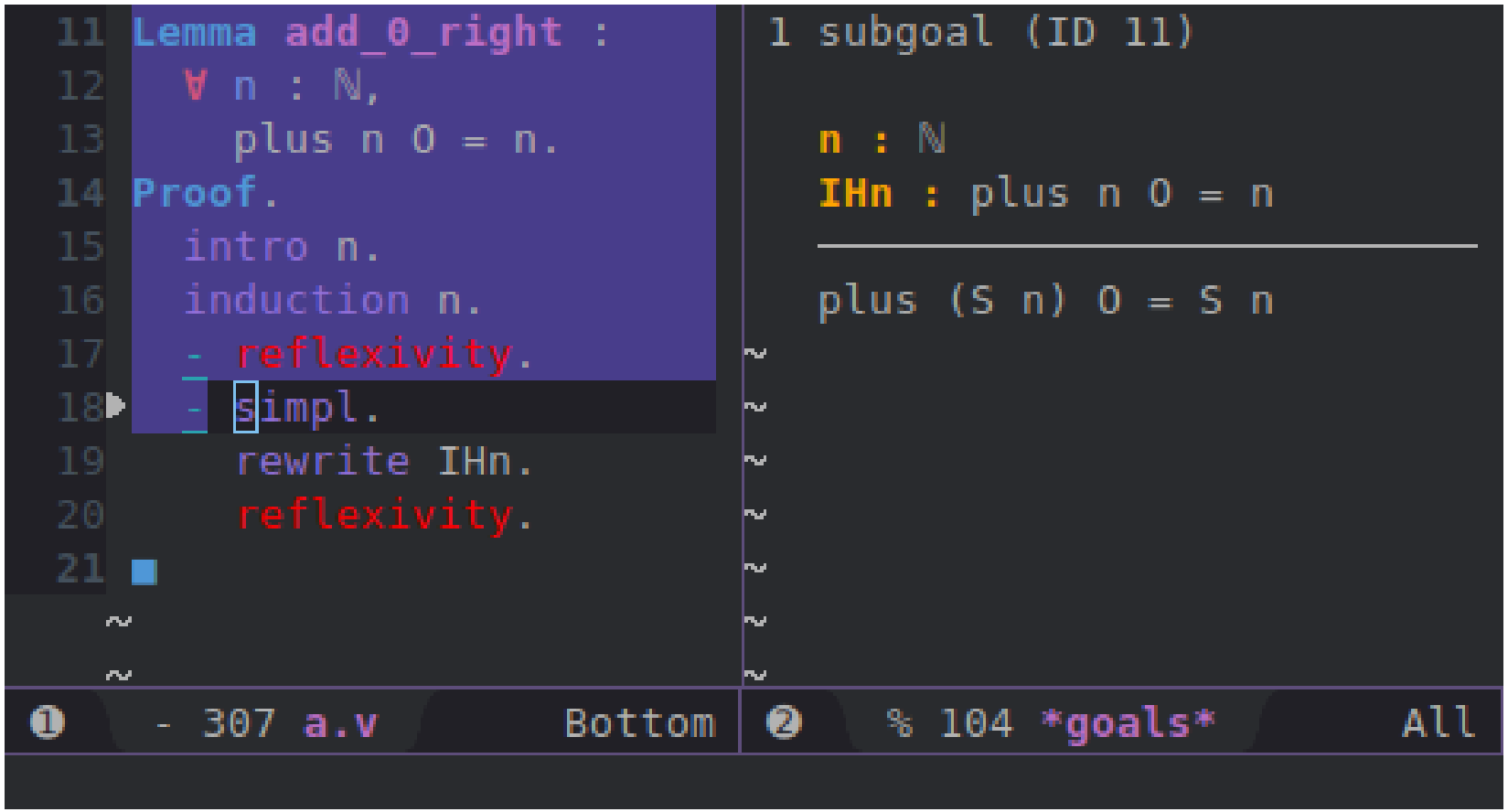
Un exemple : les entiers naturels

Preuve d'une propriété simple



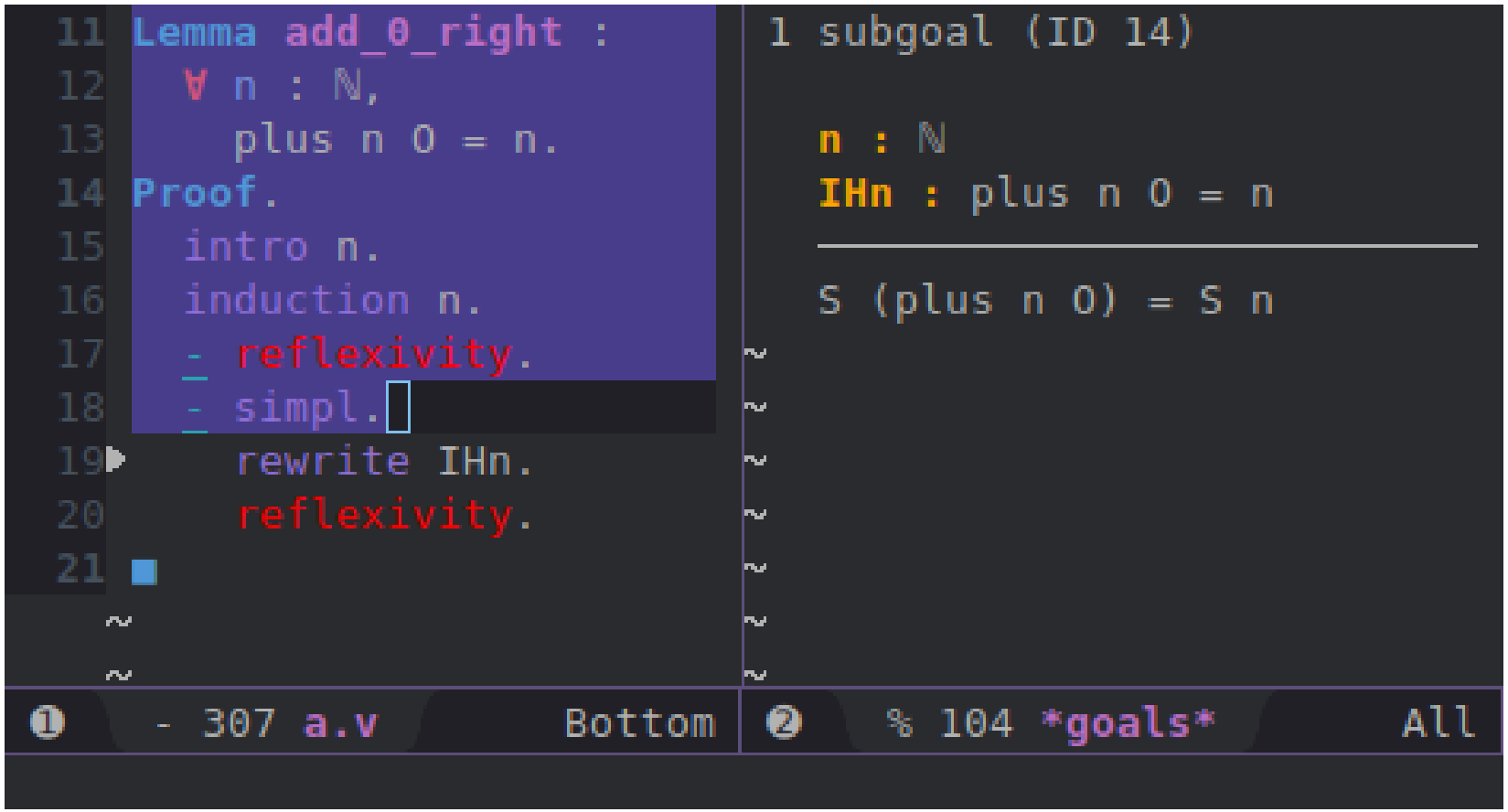
Un exemple : les entiers naturels

Preuve d'une propriété simple



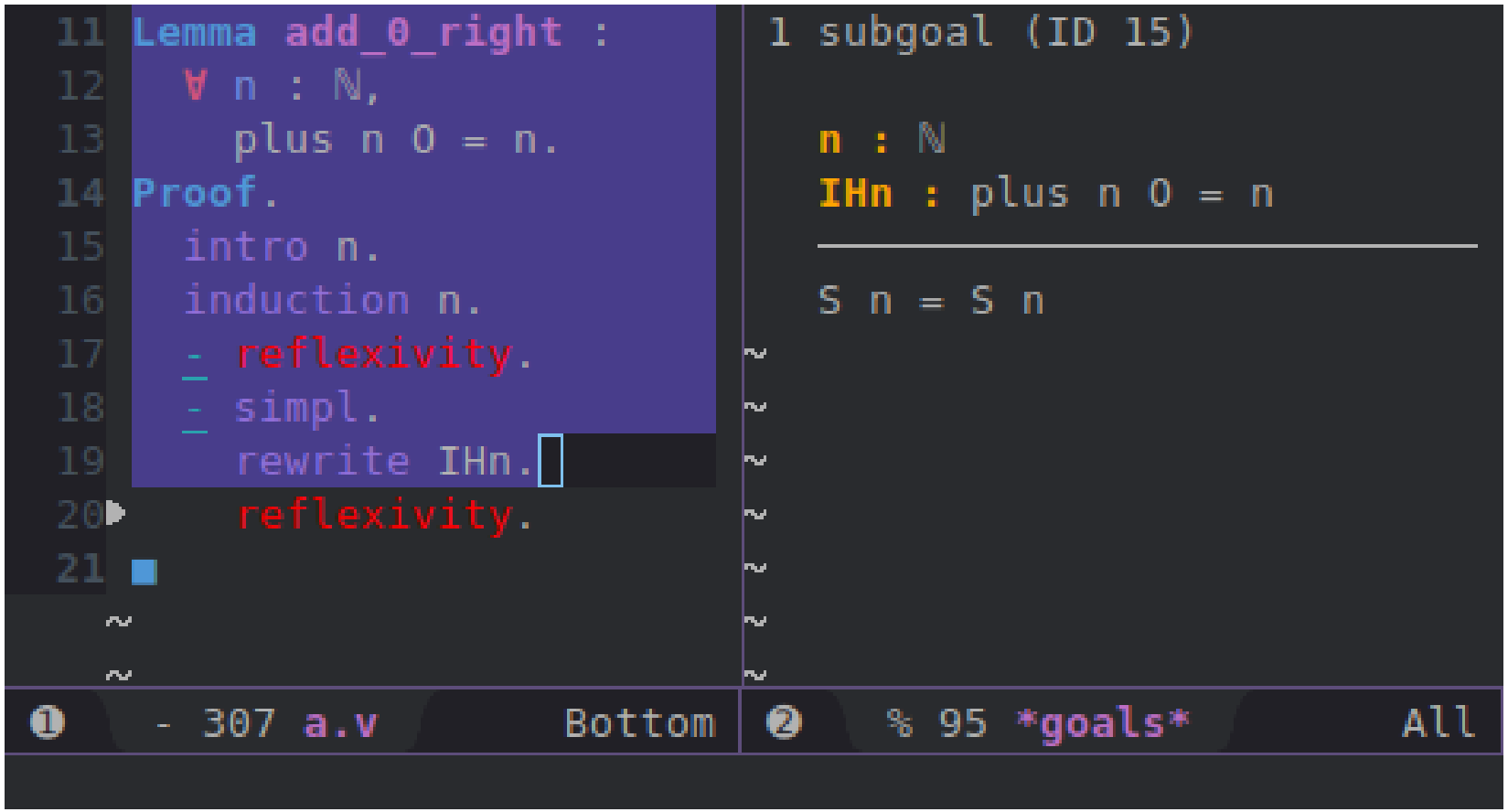
Un exemple : les entiers naturels

Preuve d'une propriété simple



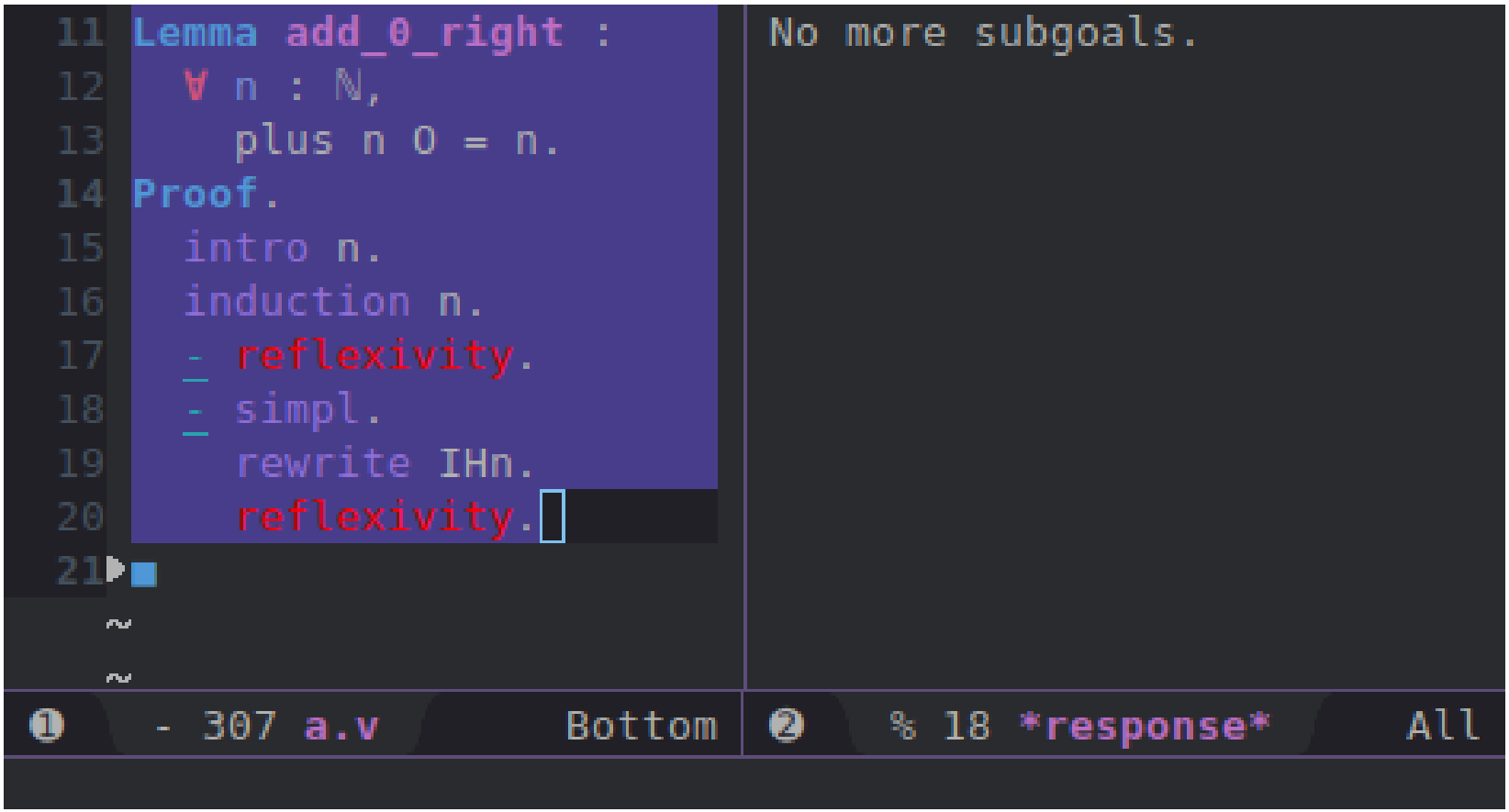
Un exemple : les entiers naturels

Preuve d'une propriété simple



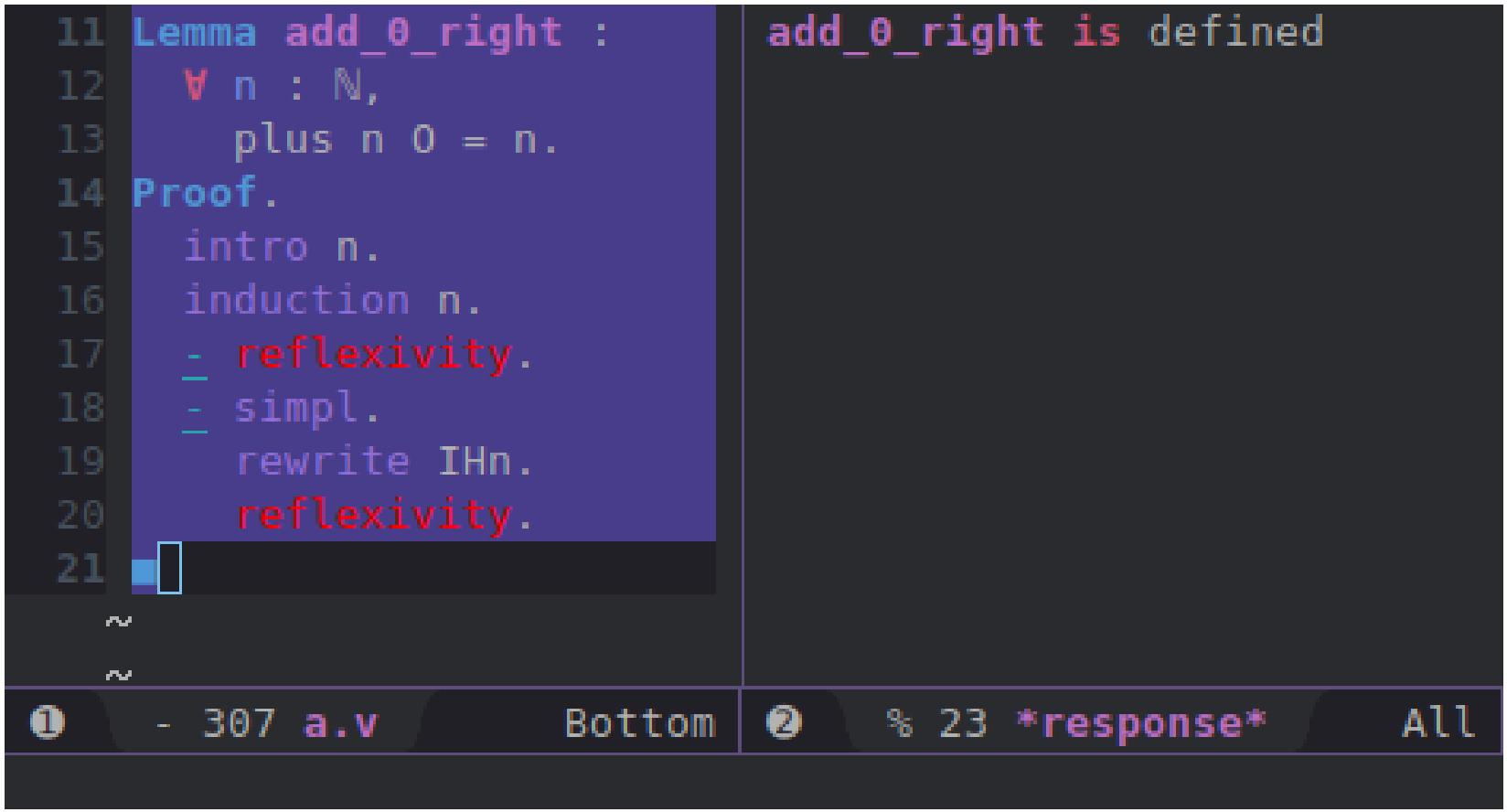
Un exemple : les entiers naturels

Preuve d'une propriété simple



Un exemple : les entiers naturels

Preuve d'une propriété simple



Sommaire

Préservation sémantique

Notations

- Programme source S
- Code objet C
- Comportement observable B (terminaison, erreurs...)
- $S \Downarrow B$: S s'éxécute avec le comportement observable B

Première idée de la définition

$$\forall B,\ S \Downarrow B \Leftrightarrow C \Downarrow B$$

Préservation sémantique

Une meilleure définition

Problème avec la définition précédente

Propriété trop forte :

- Cas d'un code source non-déterministe
- Cas de code mort menant à une erreur d'exécution

Meilleure définition

$$S \textit{ safe } \Rightarrow (\forall B, C \Downarrow B \Rightarrow S \Downarrow B)$$

(S est *safe* si aucun de ses comportements possibles ne mène à une erreur d'exécution)

Préservation sémantique

Une meilleure définition

Problème avec la définition précédente

Propriété trop forte :

- Cas d'un code source non-déterministe

Meilleure définition (si "tout est déterministe")

$$\forall B \notin \texttt{Wrong}, C \Downarrow B \Rightarrow S \Downarrow B$$

(`Wrong` est l'ensemble des comportements menant à une erreur)

Préservation sémantique

Conséquences

Correction du compilateur

$$S \models \mathrm{Spec} \Rightarrow C \models \mathrm{Spec}$$

($S \models \mathrm{Spec} \Leftrightarrow S$ est *safe* et tous les comportements de S satisfont la spécification Spec)

Si S ne rencontre pas d'erreur, alors C non plus :

$$S \text{ safe} \Rightarrow C \text{ safe}$$

Compilateur vérifié

Définition

Compilateur vérifié : on a préservation sémantique lorsque le compilateur produit du code

Composition

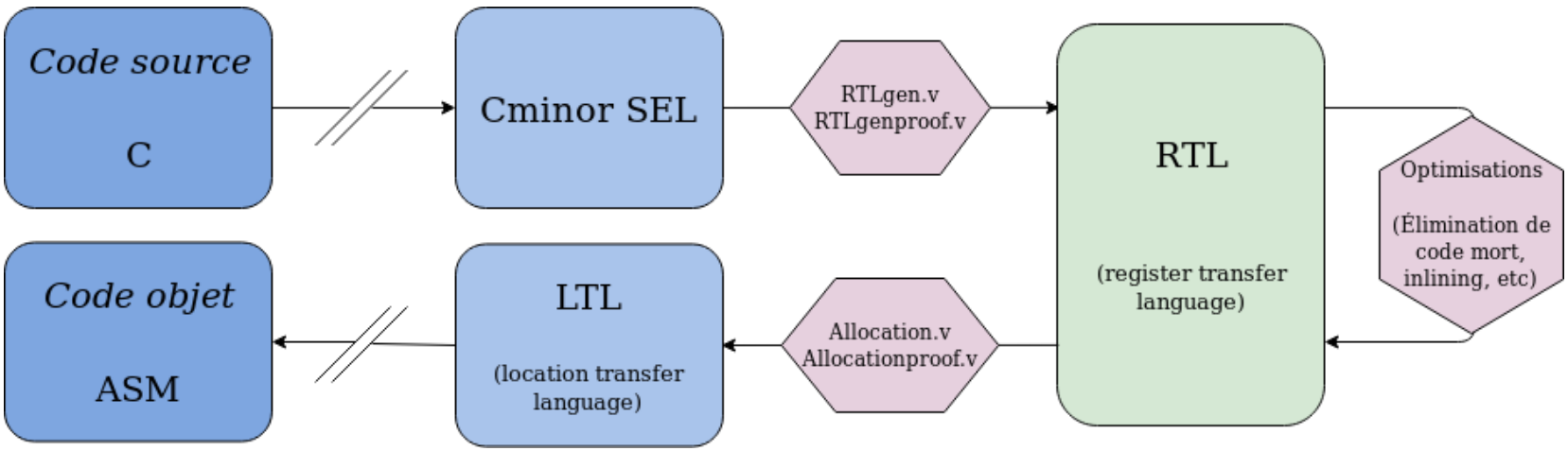
Vérifier un compilateur $\Leftrightarrow$ Vérifier chaque passe du compilateur

Sommaire

Qu'est-ce que CompCert ?

- Un compilateur C écrit et vérifié en Coq
- Un compilateur "réaliste"
- Projet initié par Xavier Leroy
- Environ 100 000 lignes de code
- 11 langages intermédiaires
- 20 passes

Structure générale



Sommaire

Présentation de RTL

Register Transfer Language

- Programme : Graphe de flot de contrôle
- Nœud : Instruction

Rôle dans CompCert

- Beaucoup d'optimisations !
- Choix d'une instruction par nœud non canonique
  $\Rightarrow$ étude d'une représentation par blocs

Représentation RTL

- Nœuds étiquetés par un entier naturel non nul
- Type inductif instruction :

```
Inductive instruction : Type :=
  | Inop (succ : node)
  | Iop (op : operation) (srcs : list reg) (dest : reg)
        (succ : node)
  | Iload (mem : memory_chunk) (mode : addressing)
        (srcs : list reg) (dst : reg) (succ : node)
  ...
  | Icond (cond : condition) (srcs : list reg)
        (iftrue : node) (iffalse : node)
  ...
  | Ireturn (oreg : option reg).
```

  contenant l'information du successeur
- Code : `Definition code : Type := PTree.tree instruction.`
- Sémantique : séquence de transitions (état initial $\rightarrow$ état final)

Nouvelle représentation : RTL_blocks

- Nouveau type instruction, sans le successeur en paramètre
  (sauf pour les instructions de branchement : Icond, Ijump, Ireturn)
- Introduction d'un nouveau type

```
Definition bblock := list instruction.
```

- Code : `Definition code : Type := PTree.tree bblock.`
- Sémantique : composition des sémantiques de RTL
- Preuves plus complexes (raisonnements par récurrence)
  $\Rightarrow$ difficulté : trouver la bonne hypothèse de récurrence

Sommaire

Une analyse : Liveness

- *Analyse de durée de vie des variables*
- Version simplifiée (indépendante des autres passes)
- But : optimisation de l'allocation de registres
- Application possible : élimination de code mort

Vocabulaire

- une variable r est *activée* par une instruction i si i lit dans r
- r est *tuée* par i si i écrit dans r
- r est *vivante immédiatement après* i si et seulement si elle est *vivante immédiatement avant* l'une des instructions succédant à i
- si r n'est ni activée ni tuée par i, alors r est *vivante immédiatement avant* i si et seulement si elle est *vivante immédiatement après* i

Principe de l'algorithme (liveness)

- Analyse "backwards" :
  - On initialise les variables vivantes avec l'ensemble vide
  - On remonte la liste des instructions en mettant à jour les nouvelles variables vivantes
- En pratique :
  - une fonction pour actualiser après chaque instruction ($\leftrightarrow$ RTL)
  - une fonction pour les blocs d'instructions
  - une fonction globale
  - preuve de la correction

Conclusion

Travail effectué

- Implémentation de RTL_blocks et Liveness_blocks ($\approx$ 1500 lignes de code)

Difficultés rencontrées

- Apprentissage de Coq
- Exploration et compréhension de CompCert

Pistes futures

- Preuve de l'équivalence sémantique RTL $\leftrightarrow$ RTL_blocks
- Comparaison des temps d'exécution pour des optimisations où le nombre de nœuds est important
- À terme : remplacer RTL par RTL_blocks ?