Certification de la transformation de tâches de preuve

Quentin Garchery

université
PARIS-SACLAY

Certification de la transformation de tâches de preuve

Certification of the transformation of proof tasks

Thèse de doctorat de l'Université Paris-Saclay

École doctorale n° 580, sciences et technologies de l'information et de la communication (STIC)

Spécialité de doctorat : **Informatique**

Unité de recherche : Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France
Référent : Faculté des sciences d'Orsay

Thèse présentée et soutenue à Paris-Saclay, le 25 janvier 2022, par

Quentin GARCHERY

Composition du jury

Mihaela Sighireanu Professeure, ENS Paris-Saclay	Présidente
Sandrine Blazy Professeure, Université de Rennes 1	Rapporteur & Examinatrice
Sylvain Boulmé Maître de conférences, INP Grenoble 1	Rapporteur & Examinateur
Stephan Merz Directeur de recherche, Inria Nancy	Examinateur
Jorge Sousa Pinto Maître de conférences, Université de Minho	Examinateur

Direction de la thèse

Claude Marché Directeur de recherche, Inria, Université Paris-Saclay	Directeur de thèse
Chantal Keller Maître de conférences, Université Paris-Saclay	Co-encadrante
Andrei Paskevich Maître de conférences, Université Paris-Saclay	Co-encadrant

Thèse de doctorat

NNT : 2022UPASG006

Remerciements

En premier lieu, je tiens à remercier mon directeur de thèse, Claude Marché, et mes encadrants de thèse, Chantal Keller et Andrei Paskevich, pour leur dévouement sans faille et pour leur volonté de partager leurs nombreuses connaissances en informatique et dans d'autres domaines. Je me suis inspiré continuellement de la patience de Chantal, de la précision d'Andrei, et du recul de Claude. Merci à Andrei de m'avoir fait partager son goût du beau et une certaine désinvolture nécessaire à la recherche. Merci à Claude de s'être assuré que ma thèse suive une trajectoire cohérente. Merci à Chantal de m'avoir donné la chance, à la suite du stage de master, de continuer, en thèse, à m'imprégner de son expertise scientifique.

Merci à mes rapporteurs de thèse, Sandrine Blazy et Sylvain Boulmé, pour leurs remarques qui ont été précieuses afin d'améliorer le manuscrit. Je suis très heureux d'avoir eu leurs retours qui ont donné un éclairage nouveau à ma thèse. Merci à la présidente du jury, Mihaela Sighireanu, et merci aux examinateurs de ma thèse, Stephan Merz et Jorge Sousa Pinto, d'avoir accepté de faire partie du jury.

Je remercie également tous les membres du LMF d'avoir instauré une ambiance chaleureuse, propice au travail. Merci à Guillaume Melquiond pour l'enthousiasme avec lequel il a partagé ses connaissances, merci à Jean-Christophe Filliâtre pour ses convictions qui motivent à comprendre. Merci à Safouan Taha pour son énergie et sa bienveillance. Merci aussi à Thibaut Balabonski, à Kim Nguyen, à Sylvain Conchon et à Pablo Arrighi de m'avoir permis d'enseigner dans les meilleures conditions possibles.

Je tiens à remercier, pour leur exemple, Raphaël Rieu-Helft, Diane Gallois-Wong, Mattias Roux et l'ancienne génération de doctorants (tous ceux qui ont soutenu leur thèse avant moi). Merci à Antoine Lanco, à Rébecca Zucchini, à Alexandrina Korneva, à Xavier Denis et à la nouvelle génération de doctorants (tous ceux qui vont soutenir leur thèse après moi) qui maintiennent et améliorent les choses.

Merci à Yohann Le Faou, à Côme Huré et à Pierre Ménard de m'avoir montré la voie du thésard. Merci à Dimitri Zawada, à Aurélien Maillocheau, à Anthony Mette, à Nicolas Nalpon et à Paul Berdin de m'avoir amené à me dépasser. J'espère leur rendre la pareille.

Merci à toute ma famille, à mes parents, Sophie et Pascal, à mes frères et sœurs, Christophe, Guillaume, Lorraine, Clément, Martial, Constance et Blandine, je leur dois tant.

Table des matières

Table des figures

Chapitre 1

Introduction

La citation latine, attribuée au poète Juvenal,

Quis custodiet ipsos custodes ?

peut être traduite par “Mais qui gardera ces gardiens ?”. Dans le contexte des méthodes formelles en informatique, sa version serait “Qui vérifiera ces vérificateurs ?”. Elle nous invite à réfléchir à la question de la confiance quand on sait que cette dernière dépend pour le moins de la confiance accordée à celui qui vérifie. On peut alors en déduire qu'il n'est pas possible de tout vérifier, ou encore que notre confiance doit être accordée au moins à un programme ou à une personne vérifiant un programme.

Cette thèse s'inscrit dans le cadre de la vérification de programmes. Un des enjeux de ce domaine résulte de l'utilisation d'outils informatiques dans des systèmes critiques tels que les transports, l'aéronautique et la médecine, domaines dans lesquels il est nécessaire d'évaluer les risques encourus. Pour vérifier un programme, il s'agit d'abord de lui donner une spécification, c'est-à-dire d'expliciter le comportement qu'on attend de lui ; un programme est dit correct lorsqu'il respecte sa spécification. La spécification d'un programme peut être vue comme un contrat entre ce programme et le contexte dans lequel il est utilisé. Un tel contrat peut porter sur des propriétés fonctionnelles du programme mais aussi, par exemple, sur sa complexité.

Exemple 1.1. *La fonction* 91 *de McCarthy peut être implémentée en OCaml de la façon suivante :*

```
let rec m (n : int) : int =
  if n > 100 then n - 10
  else m (m (n + 11))
```

Nous devons déterminer la spécification de ce programme avant de pouvoir dire qu'il est correct. On peut choisir une telle spécification, ce choix pouvant être, par exemple, d'affirmer que la valeur de retour est un entier positif. Si nous souhaitons spécifier ce programme davantage en disant qu'il termine toujours, nous verrons qu'il est nécessaire de renforcer la spécification sur la valeur de retour.

Dans tous les cas, établir la spécification d'un programme ne peut pas être un procédé mécanique. La spécification d'un programme est typiquement donnée en langage naturel, ce qui peut poser des problèmes d'interprétation. Pour les résoudre, la spécification d'un programme peut s'exprimer à l'aide de formules mathématiques : il s'agit par exemple d'annotations du programme ou de théorèmes décrivant son exécution. Si, de plus, la sémantique des

programmes est définie formellement, alors la vérification de la correction d'un programme peut être mécanisée. Il s'agit en effet de faire correspondre la sémantique du programme avec le comportement attendu. Les outils de vérification de programmes s'appuient sur cette idée mais, puisque ceux-ci sont également des outils informatiques, il se pose alors encore la question de leur correction.

1.1 Méthodes formelles

Nous souhaitons accroître la confiance accordée aux outils de vérification de programmes. Nous supposons donnée, dans ce paragraphe, une représentation formelle de ce problème, et nous expliquons alors notre démarche.

1.1.1 Formules mathématiques et transformations logiques

Les formules mathématiques permettent d'énoncer un fait de manière concise et sans ambiguïté. Le langage dans lequel sont écrites ces formules ainsi que le sens qu'on leur donne dépendent tous les deux du domaine d'application dans lequel on se place. Par exemple, en mécanique classique, on peut écrire la formule

$$\vec{a} = \frac{1}{m} \sum_i \vec{F_i}$$

et celle-ci donne un lien direct entre l'accélération d'un corps et la somme des forces qui lui sont appliquées.

Dans cette thèse, les formules mathématiques sont vues comme des objets d'étude. Plus précisément, nous étudions comment ces formules sont transformées, comment elles sont simplifiées, scindées, adaptées, etc. Ces modifications ont généralement pour but de prouver la validité des formules en question. Nous définissons alors les *transformations logiques* comme des fonctions prenant en argument une formule et renvoyant une liste de formules.

Une transformation logique appliquée à une formule produit de nouvelles formules. L'intérêt des transformations logiques est de passer de la vérification de la validité de la formule initiale à la vérification de celle de chacune des formules résultantes. On applique une transformation logique dans l'espoir qu'il soit plus simple de vérifier chaque formule résultante que de vérifier la formule initiale. De l'application d'une transformation logique résulte donc de nouvelles formules qu'il suffira de vérifier pour s'assurer que la formule initiale est vraie. Nous nous posons la question de la correction des transformations logiques, ce qui, on le verra par la suite, peut encore être vu comme une formule mathématique à prouver.

1.1.2 Preuve formelle

La preuve formelle, ou preuve assistée par ordinateur, permet de donner des garanties sur la correction des preuves mathématiques. Nous distinguons deux familles d'outils de preuve formelle : les prouveurs interactifs et les prouveurs automatiques. Afin d'expliquer notre approche de la vérification formelle des transformations logiques, nous détaillons comment les prouveurs se différencient les uns des autres, notamment en fonction de la famille à laquelle ils appartiennent.

Les garanties apportées par la preuve formelle reposent sur la base de confiance de l'outil de vérification utilisé, c'est-à-dire sur le code auquel on doit faire confiance afin de garantir que les formules sont validées.

Prouveurs interactifs. Le rôle d'un prouveur interactif est simplement de vérifier une preuve donnée. D'un prouveur interactif à l'autre, le formalisme utilisé peut varier mais il permet, dans la plupart des cas, d'exprimer de nombreuses propriétés ainsi que leurs preuves. Par exemple, le prouveur interactif Coq [72] se fonde sur le calcul des constructions inductives et le prouveur Isabelle [70] sur la logique d'ordre supérieur.

Exemple 1.2. *Le fragment de code Coq suivant*

```
Lemma eq_nth { A } : forall x : A, forall l1 l2 : list A,
  length l1 = length l2 ->
  (forall k, 0 <= k < length l1 ->
             nth k l1 x = nth k l2 x) ->
  l1 = l2.
```

énonce le fait que deux listes sont égales dès qu'elles sont de même longueur et que la fonction d'accès au n-ième élément renvoie des valeurs identiques sur les deux listes.

La preuve est donnée de manière interactive, le prouveur devant décider si chaque pas de preuve est correct. Pour éviter à l'utilisateur de donner tous les détails minutieux de chaque étape du raisonnement, les prouveurs interactifs mettent à disposition des *tactiques* [39] permettant de donner des pas de preuve plus grands où certains détails de raisonnement sont complétés par la tactique elle-même. Ces tactiques peuvent même effectuer de la recherche de preuve, ce qui permet d'automatiser davantage le travail de vérification.

Exemple 1.3. *Nous complétons l'exemple précédent avec la preuve de la propriété énoncée.*

```
Proof.
intros x l1.
  induction l1 as [| h1 t1 Hrec]; intros l2 Hlen Hnth.
  - destruct l2. trivial. discriminate Hlen.
  - destruct l2 as [| h2 t2]. discriminate Hlen.
    pose (H0 := Hnth 0). simpl in H0.
    rewrite H0; [| lia]. f_equal.
    apply Hrec. simpl in Hlen. lia.
    intros k Hk. pose (Hsk := Hnth (S k)).
    simpl in Hsk. apply Hsk. lia.
Qed.
```

La preuve de cette propriété passe par de nombreuses étapes et les tactiques permettent de décrire comment passer aux étapes suivantes. Notons que même si nous n'avons pas utilisé de lemmes sur les entiers ou les listes, nous avons appliqué des tactiques automatiques telles que `lia` *(procédure de décision sur l'arithmétique entière linéaire), ce qui a grandement raccourci la preuve, notamment au niveau des détails du contexte d'application de ces tactiques.*

Les tactiques ne font pas nécessairement partie de la base de confiance des prouveurs interactifs. En effet, certains d'entre eux tels que Coq utilisent les tactiques pour obtenir une preuve complète et vérifient ensuite cette preuve indépendamment de la manière dont elle a été construite.

Prouveurs automatiques. Le rôle d'un prouveur automatique est de résoudre, sans intervention de la part de l'utilisateur, les problèmes qui lui sont donnés en entrée. Puisque ceux-ci peuvent être arbitrairement compliqués, le prouveur pourra renvoyer une réponse

non significative ou ne pas terminer dans le temps imparti, mais ne doit pas renvoyer de réponses erronées. Des exemples de prouveurs automatiques sont donnés par CVC4 [7], Z3 [38], Alt-Ergo [30], veriT [25], Vampire [64], SPASS [82], etc.

Exemple 1.4. *Nous donnons le fichier d'entrée (format SMT-LIB [9]) suivant :*

```
1 (set-logic QF_LIA)
2 (declare-fun x () Int)
3 (declare-fun y () Int)
4 (assert (and (or (and (<= (+ x y) (- 3))
5                       (>= y 0))
6                  (<= x (- 3)))
7              (>= x 0)))
8 (check-sat)
9 (exit)
```

Ce problème est donné dans la logique propositionnelle (`QF` *signifie* Quantifier Free*) avec la théorie de l'arithmétique entière linéaire (*`LIA` *signifie* Linear Integer Arithmetic*), ligne 1. Deux variables entières x et y sont déclarées (lignes 2 et 3) et on pose comme hypothèse (lignes 4 à 7) la formule*

$$(((x + y \leq -3) \wedge (y \geq 0)) \vee (x \leq -3)) \wedge (x \geq 0)$$

La commande de la ligne 8 interroge le prouveur pour savoir si les conditions sont satisfiables, c'est-à-dire s'il existe des instances de ces variables qui vérifient la propriété. Un prouveur SMT tel que veriT renvoie alors que ce problème n'est pas satisfiable.

Même si les prouveurs automatiques résolvent des problèmes de satisfaisabilité, soumettre la négation d'une formule permet de trancher le problème de la validité de cette formule. Les prouveurs automatiques se distinguent par l'étendue des problèmes qu'ils peuvent résoudre et par leur efficacité. Dans cette thèse nous voyons les prouveurs comme des boîtes noires permettant de résoudre certains problèmes. La question de la confiance accordée aux prouveurs automatiques peut être posée, une solution a été développée par Keller [62] et implémentée dans l'outil SMTCoq [43] qui est un greffon pour Coq permettant de faire appel à différents prouveurs automatiques de manière sûre. De notre côté, puisque nous ne les utilisons pas pour vérifier les transformations logiques, nous n'avons pas à nous interroger sur leur correction.

Transformations logiques. Les transformations logiques sont utiles à la fois pour la vérification interactive et pour la vérification automatique. En effet, les transformations peuvent être comparées à des tactiques dans les prouveurs interactifs, mais font partie de la base de confiance de l'outil qui les utilise. Par ailleurs, une transformation logique peut également être utilisée afin de traduire une formule dans un langage plus réduit. Cela rend alors possible l'appel de prouveurs automatiques. Par exemple, puisque les types polymorphes ne sont pas directement disponibles dans Z3, nous pouvons appliquer une transformation qui les encode avant de lui envoyer un problème donné. Ainsi, les transformations logiques peuvent être appliquées dans le cadre de l'interaction avec des prouveurs automatiques.

Les transformations logiques sont donc des outils génériques développés pour manipuler des formules mathématiques, comme, par exemple, une transformation qui élimine les types algébriques. Ces transformations peuvent être complexes selon différents critères : elles peuvent ajouter de nouveaux symboles (par exemple, la transformation de skolémisation), faire des calculs, etc. Enfin, la correction des outils de vérification faisant appel à des

transformations logiques dépend de la correction de ces transformations. Toutes ces raisons expliquent l'intérêt que l'on a à vérifier la correction des transformations.

1.2 Vérification de programmes

1.2.1 Vérification déductive

Le cadre des transformations logiques — celui de la preuve de propriétés mathématiques — est volontairement général. Nous le voyons à travers le prisme des méthodes formelles. Les formules ont une représentation dans un langage de programmation, les transformations sont alors des fonctions sur cette structure de données et nous cherchons à les vérifier.

Différentes méthodes existent afin de pouvoir accorder une plus grande confiance à un programme. On peut simplement relire avec attention ce programme, faire des tests, utiliser des outils de *model checking*, etc. Ces méthodes ne sont pas complètes dans le cas général. Il n'est par exemple pas possible d'écrire suffisamment de tests pour s'assurer de la correction d'une fonction prenant un entier non borné en argument.

Exemple 1.5. *On s'intéresse au cas particulier $n = 4$ de la conjecture d'Euler. Le problème consiste à s'assurer qu'il n'existe pas d'entiers naturels non nuls a_1, a_2, a_3 et b tels que :*

$$a_1^4 + a_2^4 + a_3^4 = b^4$$

Il est possible de tester les nombres b un à un : pour b donné, le domaine à chercher est fini, il est inclus dans l'ensemble des triplets d'entiers naturels non nuls inférieurs à b. Cependant, si l'on ne trouve pas de contre-exemple en temps raisonnable, cela ne signifie pas que la propriété est vraie. En effet, l'identité

$$95800^4 + 217519^4 + 414560^4 = 422481^4$$

nous donne le plus petit contre-exemple [47] pour $b = 422481$.

Nous nous intéressons à la vérification déductive de programmes, approche qui s'appuie sur les méthodes formelles (prouveur automatique et prouveur interactif). C'est une méthode qui permet d'énoncer et de prouver des propriétés sur toutes les exécutions possibles d'un programme, au prix d'une difficulté de vérification accrue.

Il existe différents outils de vérification déductive de programmes qui permettent de définir un contrat dans un premier temps et de le vérifier dans un deuxième temps. Ils se distinguent par l'expressivité de leurs contrats, le paradigme choisi pour les établir et les valider, mais aussi par l'étendue de la base de confiance. Un objectif général de tels outils est de faire en sorte que la vérification soit aussi automatisée que possible. Dans de nombreux cas, cette automatisation se fait à l'aide de code venant s'ajouter à la base de confiance.

1.2.2 Présentation de Why3

Dans ce contexte, mon objectif est de renforcer la confiance que l'on accorde au logiciel de vérification déductive de programmes Why3 [19]. En effet, le fait que Why3 puisse faire appel à des dizaines de prouveurs différents permet d'améliorer l'automatisation de la vérification mais augmente d'autant la base de confiance, ensemble du code sur lequel repose la vérification.

Why3 s'appuie sur le langage WhyML, un langage de programmation annoté. Ces annotations permettent au programmeur d'expliciter la spécification d'un programme.

Exemple 1.6. *Le programme WhyML suivant définit la valeur absolue sur les entiers et a simplement comme spécification que son résultat est positif.*

```
let abs (x : int)
  ensures { result >= 0 }
=
  if x >= 0 then x else -x
```

La vérification d'un programme en Why3 se fait selon le paradigme suivant :

1. à partir du programme et de ses annotations, l'outil génère une *tâche de preuve*, énoncé logique dont la validité entraîne la correction du programme vis-à-vis du contrat ;
2. l'outil cherche à valider cette tâche de preuve, de manière automatique ou interactive.

La génération de la tâche de preuve est fondée sur l'algorithme de calcul de plus faible précondition de Dijkstra [41], où seul le contrat d'une fonction est utilisé lors de chacun de ses appels.

Exemple 1.7. *Sur l'exemple 1.6, la tâche de preuve générée est de la forme :*

```
[...] (* les déclarations précédentes sont omises  *)
goal abs'vc :
  forall x:int. forall result:int.
  (if x >= 0 then result = x else result = (- x)) ->
  result >= 0
```

Cette tâche définit un but nommé `abs'vc` *qui énonce que pour toute valeur d'entrée* `x`*, et toute valeur de sortie* `result` *qui est obtenue par application de* `abs` *sur* `x`*, c'est-à-dire telle que*

```
if x >= 0 then result = x else result = (- x)
```

alors la valeur de sortie est positive. Cette tâche peut être vérifiée par des prouveurs automatiques qui supportent l'arithmétique entière tels que Z3 ou Alt-Ergo.

Lorsque l'annotation `variant` apparaît dans la spécification d'une fonction récursive, Why3 génère une tâche de preuve qui, une fois validée, nous garantit que cette fonction termine.

Exemple 1.8. *Implémentons l'exemple 1.1 dans Why3 et donnons à la fonction obtenue une spécification qui porte non seulement sur sa terminaison mais aussi sur la valeur de retour.*

```
1 let rec m (n : int) : int
2   variant { 100 - n }
3   ensures { n > 100 -> result = n - 10 }
4   ensures { n <= 100 -> result = 91 }
5 = if n > 100 then n - 10
6   else m (m (n + 11))
```

Ainsi annoté ce code est prouvé automatiquement par des prouveurs SMT, mais ce n'est pas le cas si la ligne 3, ou la ligne 4, est omise.

Dans le cas général, la tâche de preuve générée est complexe et ne peut pas être directement validée par les prouveurs automatiques disponibles. Les annotations permettent de donner des indices aidant à sa vérification.

Exemple 1.9. *Le programme suivant calcule la somme des n premiers entiers en itérant sur n et en mettant à jour une référence. Sa spécification est que si l'argument donné est positif ou nul alors la valeur renvoyée est égale à* $\frac{n(n+1)}{2}$.

```
let sum n
  requires { n >= 0 }
  ensures { result = div (n * (n+1)) 2 }
=
  let ref s = 0 in
  for i = 1 to n do
      invariant { s = div ((i-1) * i) 2 }
      s <- s + i
  done;
  s
```

Ce programme a également une indication qui est sous la forme d'un invariant de boucle et qui permet d'obtenir automatiquement une preuve de sa correction.

Exemple 1.10. *On définit la fonction qui ajoute à un accumulateur la somme des nombres de Fibonacci d'une liste d'entiers naturels et on veut montrer qu'elle termine. On écrit*

```
let rec sum_fib (acc:int) (l:list int) : int
  requires { forall n. mem n l -> n >= 0 }
  variant { sum_pow l }
= match l with
| Nil -> acc
| Cons n r ->
    if n <= 1 then sum_fib (acc + n) r
    else sum_fib acc (Cons (n-1) (Cons (n-2) r))
end
```

où `mem` *est la fonction indiquant si un élément appartient à une liste et* `sum_pow` *est la fonction qui calcule la somme des puissances de deux de la liste donnée en argument. Notons que ces deux fonctions n'interviennent pas dans le code de* `sum_fib` *et servent uniquement à l'annoter. Les prouveurs SMT n'arrivent pas à valider directement la tâche de preuve générée, mais, en faisant appel à des transformations logiques afin de décomposer cette tâche, nous rendons la correction du programme donné accessible à certains prouveurs automatiques tels que Z3 et CVC4.*

L'utilisation de transformations permet d'obtenir de nouvelles tâches de preuve, en espérant que celles-ci soient plus simples à valider. Ces transformations logiques peuvent être utilisées de manière interactive, ce qui permet à l'utilisateur de décomposer le problème, ou plus généralement de le modifier à sa convenance pourvu que l'environnement de preuve garantisse la correction de ces modifications.

Exemple 1.11. *La tâche de preuve de l'exemple 1.9 regroupe plusieurs vérifications à effectuer conditionnant la correction du programme : il faut s'assurer que l'invariant de boucle est vrai au début de la boucle, qu'il est préservé à chaque tour de boucle, et que la valeur*

du résultat est celle indiquée par l'annotation `ensures`*. Appliquer la transformation* split *produit une tâche de preuve pour chacune des vérifications à effectuer. Pour la préservation de l'invariant, on obtient une tâche de preuve de la forme*

```
[...] (* les déclarations précédentes sont omises *)
LoopInvariant : s1 = div ((i - 1) * i) 2
H : s = (s1 + i)
goal sum'vc : s = div (((i + 1) - 1) * (i + 1)) 2
```

et nous montrons qu'elle est valide en établissant l'égalité mathématique suivante

$$\frac{(i-1)i}{2} + i = \frac{i(i+1)}{2}$$

Les annotations peuvent aussi être placées dans le corps d'un programme sous la forme d'instructions `assert`. Cela permet de diriger l'algorithme de calcul de plus faible précondition et peut être mis à profit pour la preuve de propriétés mathématiques.

Exemple 1.12. *La formalisation de l'exemple 1.2 en WhyML peut être donnée par le fragment de code suivant.*

```
let rec lemma eq_nth (l1 l2 : list 'a)
  requires { length l1 = length l2 }
  requires { forall k. 0 <= k < length l1 -> nth k l1 = nth k l2 }
  ensures { l1 = l2 }
= match l1, l2 with
| Cons a1 t1, Cons a2 t2 ->
    assert { a1 = a2 && length t1 = length t2 };
    assert { forall k. 0 <= k < length t1 ->
             nth k t1 = nth k t2
             by nth (k+1) l1 = nth (k+1) l2 };
    eq_nth t1 t2
| Nil, Nil -> ()
| _ -> absurd
end
```

Cet exemple est tiré de ma solution proposée à VerifyThis 2019 [42] et propose une fonction au contenu calculatoire sans intérêt (elle n'a pas de valeur de retour ni d'effets de bord), mais qui sert à établir la propriété exprimée par son contrat. L'application de la transformation `split` *permet de décomposer la tâche de preuve générée en des tâches qui peuvent être alors envoyées aux prouveurs automatiques pour vérification.*

Les transformations logiques sont aussi utilisées automatiquement par Why3 lorsqu'une tâche de preuve est envoyée à un prouveur automatique. En fonction du prouveur automatique considéré, différentes transformations sont appelées afin de traduire la tâche de preuve dans la logique du prouveur en question. Par exemple, Bobot et Paskevich [20] ont mis en place une famille de transformations Why3 qui éliminent le polymorphisme, ce qui rend possible d'envoyer des tâches de preuve polymorphes aux prouveurs SMT.

Certaines propriétés ou raisonnements ne sont pas réalisés par les prouveurs automatiques et les transformations logiques permettent de combler ce manque. Un exemple type est le raisonnement par récurrence, typiquement difficile à automatiser efficacement. L'instanciation de lemmes quantifiés peut aussi poser problème pour les prouveurs automatiques dans certains cas.

Exemple 1.13. *Considérons l'exemple suivant proposé par Dailler, Marché et Moy [37] :*

```
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]
```

Après application de la transformation split *sur la tâche de preuve générée, une des tâches à valider est donnée par :*

```
[...] (* les déclarations précédentes sont omises *)
Requires1 : length a >= 1000 /\ 0 <= x <= 10
Requires2 : forall i:int. 0 <= 4 * i + 1 < length a ->
            a[4 * i + 1] >= 0
H : y = 2 * x + 1
goal VC2 : a[y * y] >= 0
```

Le but `VC2` *ci-dessus n'est prouvé par aucun prouveur SMT, à cause de la difficulté d'instancier l'hypothèse* `Requires2` *avec la bonne valeur de* `i`*. Un appel manuel de la transformation* `instantiate Requires2 x*x+x` *produit la tâche*

```
[...] (* les déclarations précédentes sont omises *)
Hinst : 0 <= 4 * (x * x + x) + 1 < length a ->
        a[4 * (x * x + x) + 1] >= 0
goal VC2 : a[y * y] >= 0
```

qui est maintenant validée automatiquement par les prouveurs pouvant traiter les buts arithmétiques.

1.3 Choix d'une approche sceptique

Mes travaux de thèse ont pour objectif de renforcer la confiance que l'on peut accorder aux transformations logiques. Nous nous intéressons aux transformations logiques des outils de vérification déductive, et nous appliquons notre méthode aux transformations logiques de Why3. L'intérêt que l'on porte à la correction de ces transformations s'explique par deux observations. D'une part les transformations logiques, nécessaires à chaque appel de prouveur automatique, sont omniprésentes dans les développements en Why3. D'autre part, leur implémentation comporte déjà plus de 17000 lignes de code OCaml, code qui est dans la base de confiance de Why3 et que nous souhaitons vérifier.

Mon approche est fondée sur l'utilisation de certificats vérifiables par une tierce partie, selon l'approche dénommée *sceptique* par Harrison [61]. L'idée est d'instrumenter le code des transformations afin qu'il génère un *certificat* de preuve, celui-ci permettant d'assurer a posteriori la correction de chaque exécution de l'outil. Ce certificat peut être vérifié par un outil externe qui, lui, peut avoir une base de confiance réduite. L'approche sceptique est particulièrement indiquée lorsque l'on souhaite instrumenter petit à petit le code à vérifier ou lorsque l'on veut pouvoir prouver plus de propriétés par la suite, comme par exemple dans les travaux sur SMTCoq [3, 17]. Je mettrai donc l'accent sur l'aspect incrémental et modulaire de l'utilisation des certificats. On peut distinguer, au sein de l'approche sceptique, le travail de production de certificats du travail de vérification de ces certificats. Premièrement, celui qui instrumente la transformation doit expliciter, pour chaque cas d'application et sous la

forme d'un certificat, la raison pour laquelle la transformation est correcte. Deuxièmement, la vérification de l'application d'une transformation peut s'appuyer sur un certificat, ce qui facilite grandement cette vérification. Similairement à l'approche FVDP de Boulmé [24], nous pouvons donc argumenter en faveur de la facilité de la vérification.

À l'inverse, une approche *autarcique*, selon la formulation de Barendregt et Barendsen [6], aurait cherché à prouver le code fourni. Par exemple, l'approche autarcique appliquée par Lescuyer à la vérification du prouveur Alt-Ergo [68] a mené à une version formalisée en Coq du noyau de ce prouveur. Cette approche permet de vérifier des propriétés du code autres que les propriétés fonctionnelles : il est par exemple possible de vérifier la terminaison ou de borner sa complexité. Un autre avantage est donné par le fait que la vérification d'une fonction est effectuée une fois lors de sa définition, et non à chacun de ses appels. Cette deuxième approche est cependant plus rigide car modifier le programme oblige également à adapter sa preuve de correction.

Faisons aussi la distinction entre les outils de vérification de programmes qui se fondent sur un assistant de preuve et ceux pour lesquels ce n'est pas le cas. Dans le premier cas, le langage de programmation et sa sémantique sont tous deux définis dans l'assistant de preuve considéré et le développement du langage se fait à travers l'implémentation de bibliothèques dédiées. C'est le cas notamment de la bibliothèque Iris [65] implémentée en Coq et développée pour la vérification de programmes impératifs et concurrents. Un autre exemple est donné par la bibliothèque AutoCorres [56, 57] qui permet de vérifier des programmes C en Isabelle. Dans ce contexte, la correction de la méthode repose sur la définition formelle de la sémantique des programmes et sur les règles de déduction qui doivent être établies une fois pour toutes. Cela demande de fournir un effort de preuve important, ce qui limite la flexibilité du langage. Dans le deuxième cas, la vérification ne s'appuie plus sur les prouveurs interactifs mais sur les tâches de preuve (aussi appelées obligations de preuve) dérivées à partir des annotations des programmes, et qui peuvent ensuite être validées par des prouveurs automatiques. Nous pouvons donner des exemples de tels outils de vérification de programmes en citant Why3 mais aussi Dafny [66], Viper [69], Frama-C [63], etc. Bien que la correction de ces outils repose sur des fondements théoriques solides, l'implémentation en elle-même de chacun d'entre eux, et en particulier l'encodage des tâches de preuve dans la logique des prouveurs automatiques appelés, n'est pas garantie. Une exception est donnée par F* [79], dont la correction de l'encodage dans la logique d'un prouveur SMT a été partiellement établie en Coq par Aguirre [2].

1.4 Encodage profond et encodage superficiel

Nous faisons mention, à divers endroits dans ce manuscrit, d'*encodages profonds* et d'*encodages superficiels*. En effet, au moment de vérifier nos certificats, nous avons le choix de l'outil de vérification que l'on utilise, de l'encodage des certificats dans cet outil et donc de la façon dont leur sémantique y est reflétée. D'un côté, l'encodage profond d'un langage (resp. d'une logique) L_{source} dans un langage (resp. dans une logique) consiste à déclarer de nouveaux symboles dans L_{cible} permettant de traduire les symboles de L_{source}, et à donner la sémantique de ces nouveaux symboles, par exemple à l'aide de nouveaux axiomes. Un encodage profond permet de raisonner sur L_{source} vu comme des objets de L_{cible}. Par exemple, dans la lignée du compilateur certifié CompCert [67], Blazy [16] propose un encodage profond d'un sous-ensemble de C en Coq, ce qui permet d'énoncer et de prouver la correction de sa traduction vers un langage intermédiaire. D'un autre côté, l'encodage superficiel de L_{source} dans L_{cible} consiste en une traduction syntaxique des symboles de L_{source} comme une combinaison de symboles de L_{cible}. La sémantique de l'encodage de L_{source} dans L_{cible}

découle de la sémantique donnée à L_{cible}. De même, comme le remarque Cauderlier [26], les propriétés et les fonctionnalités de L_{cible} peuvent être directement mises à profit, facilitant ainsi les développements s'appuyant sur un encodage superficiel.

Dans cette thèse, nous utilisons les deux techniques d'encodage pour la vérification des certificats (chapitre 5). Nous avons également défini deux types de certificat, le premier étant encodé de manière superficielle dans le second (voir le 6.4), ceux-ci pouvant être encodés de manière profonde. Cela représente donc un moyen de combiner les deux types d'encodage en définissant des *derived constructs* comme le montre Svenningsson [78]. Par ailleurs, les certificats de surface font usage de la *syntaxe abstraite d'ordre supérieur* proposée par Pfenning [73] afin de réutiliser les lieurs du langage ambiant pour décrire ceux du langage que l'on définit. Puisque les autres constructions du langage sont définies de manière profonde, cela représente aussi un travail à mi-chemin entre l'encodage profond et l'encodage superficiel.

1.5 Contributions

Nous commençons, dans le chapitre 2, par définir le cadre logique dans lequel nous nous plaçons : les transformations logiques sont définies comme des fonctions partielles sur les tâches de preuve, elles-mêmes définies à partir de termes typés. Les transformations sont ensuite instrumentées, dans le chapitre 3, afin qu'elles renvoient un certificat à chacune de leurs applications. Le format de certificat choisi, celui des certificats de noyau, facilite la vérification et permet de certifier des transformations dans une logique d'ordre supérieur avec polymorphisme. Nous étendons notre formalisme dans le chapitre 4 en permettant l'ajout de théories interprétées telles que l'égalité et l'arithmétique entière. Nous expliquons au passage le mécanisme générique pour ajouter une théorie à notre formalisme. La vérification de certificats du chapitre 5 est modulaire. En effet, nous définissons deux vérificateurs. Le premier suit une approche calculatoire efficace de la vérification de certificats qui manipule l'encodage profond des tâches de preuve. Le second vérificateur se fonde sur un prouveur interactif dans lequel nous encodons notre formalisme de façon superficielle. Afin de simplifier l'instrumentation des transformations, nous définissons un deuxième format de certificat dans le chapitre 6, celui des certificats de surface. Ces derniers sont de plus haut niveau que les certificats de noyau, ce qui facilite leur génération et améliore leur modularité. Les certificats de surface sont traduits en des certificats de noyau avant leur vérification. Nous appliquons ensuite notre méthode aux transformations de Why3 dans le chapitre 7. Cela nécessite de traduire le formalisme de Why3 dans le nôtre. Nous donnons ensuite des garanties du bien-fondé et de l'efficacité de notre méthode. En conclusion de cette thèse, dans le chapitre 8, nous rappelons nos contributions, nous les comparons avec des travaux connexes et nous donnons les améliorations que nous envisageons.

Les développements logiciels de cette thèse sont accessibles à l'adresse

`https://gitlab.inria.fr/why3/why3/tree/cert`

Les détails pour compiler, utiliser et tester ces travaux sont donnés dans le fichier `README_CERT.md` à la racine du dépôt.

Chapitre 2

Cadre logique

Nous souhaitons décrire les transformations logiques, nous nous plaçons pour cela dans la logique d'ordre supérieur classique [10] à l'exception du fait que les types peuvent être polymorphes, la quantification de type étant en forme prénexe. Ce formalisme est suffisamment général pour permettre la certification des transformations logiques que nous considérons.

Dans ce chapitre, nous commençons, au paragraphe 2.1, par introduire les notations utilisées tout au long du manuscrit. Ces notations visent à faciliter l'écriture des termes et des tâches de preuve qui apparaissent à de nombreuses reprises. Nous explicitons ensuite, dans les paragraphes 2.2 à 2.4, la grammaire des termes, celle de leurs types et les substitutions que nous considérons. Cela nous permet, dans les paragraphes 2.5 à 2.7, d'expliciter le typage des termes, de définir les tâches de preuve et de leur donner une sémantique. Enfin, dans le paragraphe 2.8, nous donnons la définition des transformations logiques ainsi que celle de la correction de leurs applications.

2.1 Définitions générales et notations

Le symbole := permet de donner une définition.

Booléens et entiers. Les littéraux des booléens sont notés $true$ et $false$. On utilise les notations usuelles pour les entiers. Cela inclut la notation des symboles arithmétiques usuels ainsi que la précédence et l'associativité de ces symboles.

Listes. Nous notons les listes entre crochets et séparons les éléments par des points-virgules. L'élément d'indice k d'une liste L est noté $L(k)$, sachant que les indices commencent à 1. La concaténation de la liste L_1 avec la liste L_2 est notée $L_1 \mathbin{+\!\!+} L_2$ et on écrit $L_1 \subseteq L_2$ pour signifier que chaque élément de L_1 est un élément de L_2.

Exemple 2.1. *On a* $[1;\ 5;\ 5] \mathbin{+\!\!+} [3] = [1;\ 5;\ 5;\ 3]$. *Si* L *est égale à la liste* $[1;\ 5;\ 5;\ 3]$, *alors l'élément* $L(4)$ *est* 3 *et* $L(i)$ *n'est défini que pour* i *compris entre* 1 *et* 4.

Identifiants. Nous supposons donné un ensemble infini de noms, aussi appelés des *identifiants*, que l'on désigne par *ident*. Cet ensemble nous permet de définir les variables. Nous supposons aussi qu'il existe un ordre total sur les identifiants, noté $<_{ident}$.

Dictionnaires. Un *dictionnaire* est un ensemble de paires avec un identifiant pour élément de gauche telles que les identifiants sont deux à deux distincts. Ces paires sont appelées des *déclarations*. La notation $i : v$ désigne la déclaration composée de i et de v, on dit alors que v est associé à i. Un dictionnaire est noté en séparant ses déclarations par des virgules. Nous notons $M(i)$ l'élément associé à i dans le dictionnaire M, s'il existe. Par abus de notation, on écrit parfois $i \in M$ pour signifier qu'il existe une paire dans M avec i pour élément de gauche.

Exemple 2.2.
- *La notation* $C : 1,\ D : 2$ *désigne le même dictionnaire que* $D : 2,\ C : 1$*, c'est le dictionnaire associant* 1 *à* C *et* 2 *à* D*.*
- $(M, i : v)(i) = v$ *pour tout* M*,* i *et* v*.*
- *Si* $i' \in M$ *et* $i' \neq i$ *alors* $(M, i : v)(i') = M(i')$*.*

Lieurs, variables libres et substitutions. Dans la suite nous définissons des termes avec des variables qui sont explicitement liées. De tels termes utilisent donc des lieurs, comme en λ-calcul, et sont considérés à α-renommage près, la variable qui est liée n'ayant pas de sens au-delà du lieur. Cette considération nous permet de donner une définition récursive de la *substitution* dans de tels termes et de définir les *variables libres* comme les variables qui apparaissent dans un terme.

Afin d'alléger les notations, lorsque cela ne peut pas prêter à confusion, on se permet d'omettre les lieurs additionnels et même les indications de type. Par exemple, en lambda-calcul simplement typé, au lieu de $\lambda x : int.\ \lambda y : int.\ x + y$, on peut écrire $\lambda x\ y.\ x + y$.

2.2 Syntaxe des types

Les termes sont typés par des types polymorphes et définis par la grammaire de la figure 2.1.

$type$	$::=$	α	variable de type
	$\mid$	$prop$	type des formules
	$\mid$	$type \rightsquigarrow type$	type flèche
	$\mid$	$\iota(type, \ldots, type)$	application d'un symbole de type

FIGURE 2.1 – Grammaire des types

Le type *prop* est le type des *formules* et le type flèche permet de donner un type aux fonctions et aux prédicats. On note $V_\alpha(\tau)$ l'ensemble des variables de type apparaissant dans le type τ. La quantification sur les variables de type est implicite dans le sens où l'on ne définit pas de lieur dans la grammaire. Cette quantification est prénexe et, dans la suite, nous explicitons à quel moment il faut comprendre un type comme étant quantifié universellement. La flèche de type, notée $\rightsquigarrow$, est associative à droite.

Exemple 2.3. *Dans cet exemple, les types sont à comprendre comme quantifiés universellement et de façon prénexe sur toutes leurs variables de type. La fonction identité polymorphe a alors pour type* $\alpha \rightsquigarrow \alpha$*. La fonction de composition polymorphe a pour type* $(\beta \rightsquigarrow \gamma) \rightsquigarrow (\alpha \rightsquigarrow \beta) \rightsquigarrow \alpha \rightsquigarrow \gamma$ *tandis que l'égalité polymorphe a pour type* $\alpha \rightsquigarrow \alpha \rightsquigarrow prop$*.*

La grammaire des types contraint les symboles de type à être complètement appliqués. Ces symboles sont donnés par une *signature de type*, dictionnaire associant, à un symbole de type, un entier naturel représentant son *arité*, c'est-à-dire son nombre d'arguments de type. Ce dictionnaire est donc constitué de paires de la forme '$\iota : n$' où ι est un symbole de type et n est un entier, et sont appelées des *déclarations de symbole de type*. Pour simplifier les notations, l'application d'un symbole d'arité 0 est notée sans parenthèses.

Définition 2.4 (Type bien formé)**.** *Étant donné une signature de type I, un type τ est bien formé dans I si chaque application d'un symbole de type ι de τ respecte l'arité $I(\iota)$.*

Dans la suite on suppose que tous les types sont bien formés dans leurs signatures de type respectives.

Exemple 2.5. *Considérons la signature de type*

$$I := \ color : 0,\ set : 1$$

définissant un symbole de type d'arité 0 *pour les couleurs et un symbole de type d'arité* 1 *pour les ensembles polymorphes. Les types suivants sont bien formés dans I :*

- *color pour les couleurs, par exemple le rouge*
- *set(color) pour les ensembles de couleurs*
- *$\alpha \rightsquigarrow set(\alpha) \rightsquigarrow set(\alpha)$ pour la fonction polymorphe qui ajoute un élément à un ensemble*
- *$\alpha \rightsquigarrow set(\alpha) \rightsquigarrow prop$ pour le prédicat d'appartenance à un ensemble*

2.3 Syntaxe des termes

Le langage des termes (et des formules) est donné par la grammaire de la figure 2.2. Nous définissons des formules polymorphes en utilisant un quantificateur de type explicite Π, cette quantification de type étant nécessairement en forme prénexe. La notation $(t)_\tau$ décrit un terme t annoté avec son type τ, nous disons alors que t est *accompagné* de τ. Cette annotation de type fait partie de la syntaxe des termes. La plupart du temps, afin de simplifier les notations, nous omettons les types des termes. Par exemple, nous écrivons $\top \wedge \bot$ à la place de $((\top)_{prop} \wedge (\bot)_{prop})_{prop}$.

Les variables sont aussi appelées des *symboles de terme* et sont données par une *signature*, dictionnaire associant un type à un symbole de terme, ce type étant à comprendre comme quantifié universellement et de façon prénexe sur toutes ses variables de type. Ce dictionnaire est donc constitué de paires de la forme '$x : \tau$' où x est un symbole de terme et τ est un type ; une telle paire est appelée une *déclaration de symbole de terme*. L'application utilise la notation à la Curry : c'est une opération binaire et associative à gauche.

Exemple 2.6. *Reprenons l'exemple 2.3 en commençant par définir une signature déclarant ces fonctions :*

$$\begin{aligned}\Sigma := \ & id : \alpha \rightsquigarrow \alpha,\\ & comp : (\beta \rightsquigarrow \gamma) \rightsquigarrow (\alpha \rightsquigarrow \beta) \rightsquigarrow \alpha \rightsquigarrow \gamma,\\ & eq : \alpha \rightsquigarrow \alpha \rightsquigarrow prop\end{aligned}$$

On peut également donner les formules suivantes :

- *la réflexivité de l'égalité*

$$\Pi\alpha.\ \forall x : \alpha.\ eq\ x\ x$$

formula	::=	*term*	
	|	$(\Pi\alpha.\ formula)_{type}$	quantificateur de type
term	::=	$(termNode)_{type}$	
termNode	::=	x	variable
	|	$\top$	formule vraie
	|	$\bot$	formule fausse
	|	$\neg\ term$	négation logique
	|	*term op term*	connecteur logique binaire
	|	*term term*	application
	|	$\lambda x : type.\ term$	fonction anonyme
	|	$\exists x : type.\ term$	quantificateur existentiel
	|	$\forall x : type.\ term$	quantificateur universel
op	::=	$\wedge$	conjonction
	|	$\vee$	disjonction
	|	$\Rightarrow$	implication
	|	$\Leftrightarrow$	équivalence

FIGURE 2.2 – Grammaire des termes

— *une formule définissant la fonction identité*

$$\Pi\alpha.\ \forall x : \alpha.\ eq\ (id\ x)\ x$$

— *une formule définissant la fonction de composition*

$$\Pi\alpha\ \beta\ \gamma.\ \forall g : \beta \rightsquigarrow \gamma.\ \forall f : \alpha \rightsquigarrow \beta.\ \forall x : \alpha.\ eq\ (comp\ g\ f\ x)\ (g\ (f\ x))$$

2.4 Substitutions

On rappelle que les termes et les types sont définis à α-renommage près, comme expliqué dans le paragraphe 2.1. Cela nous permet de donner la définition suivante sans risque de capturer les variables liées.

Définition 2.7 (Substitution)**.** *La substitution $t_1[(x)_\tau \mapsto (t_2)_\tau]$ est définie récursivement en remplaçant les occurrences de $(x)_\tau$ dans t_1 par $(t_2)_\tau$. Cette substitution est aussi notée plus simplement $t_1[x \mapsto t_2]$ lorsqu'il n'y a pas d'ambiguïté sur le type τ. Les définitions et notations sont similaires dans le cas de la substitution d'un type dans un terme $t[\alpha \mapsto \tau]$ et d'un type dans un type $\tau_1[\alpha \mapsto \tau_2]$. Plus généralement, une substitution $t\sigma$ pourra désigner t dans lequel plusieurs remplacements ont été effectués en parallèle.*

Exemple 2.8. *Soit $\sigma := [a \mapsto x, x \mapsto y]$, le terme $(x \vee (\forall x.\ a \Rightarrow x))\sigma$ est défini par :*

$$y \vee (\forall z.\ x \Rightarrow z)$$

La substitution s'applique aussi aux types accompagnant les termes.

Exemple 2.9. *La notation $(id)_{\alpha \rightsquigarrow \alpha}[\alpha \mapsto \iota]$ désigne $(id)_{\iota \rightsquigarrow \iota}$.*

$$\frac{I, \iota : 0 \mid \Sigma \Vdash f[\alpha \mapsto \iota] : prop \qquad \iota \notin I}{I \mid \Sigma \Vdash \Pi\alpha.\ f : prop} \qquad \frac{\tau = \Sigma(x)\sigma \qquad V_\alpha(\tau) = \emptyset}{\Sigma \Vdash x : \tau}$$

$$\frac{}{\Sigma \Vdash \top : prop} \qquad \frac{}{\Sigma \Vdash \bot : prop} \qquad \frac{\Sigma \Vdash t : prop}{\Sigma \Vdash \neg t : prop} \qquad \frac{\Sigma \Vdash t_1 : prop \qquad \Sigma \Vdash t_2 : prop}{\Sigma \Vdash t_1\ op\ t_2 : prop}$$

$$\frac{\Sigma \Vdash t_1 : \tau \rightsquigarrow \tau' \qquad \Sigma \Vdash t_2 : \tau}{\Sigma \Vdash t_1\ t_2 : \tau'} \qquad \frac{\Sigma, x : \tau' \Vdash t : \tau \qquad V_\alpha(\tau') = \emptyset \qquad x \notin \Sigma}{\Sigma \Vdash (\lambda x : \tau'.\ t) : \tau' \rightsquigarrow \tau}$$

$$\frac{\Sigma, x : \tau \Vdash t : prop \qquad V_\alpha(\tau) = \emptyset \qquad x \notin \Sigma}{\Sigma \Vdash (\forall x : \tau.\ t) : prop} \qquad \frac{\Sigma, x : \tau \Vdash t : prop \qquad V_\alpha(\tau) = \emptyset \qquad x \notin \Sigma}{\Sigma \Vdash (\exists x : \tau.\ t) : prop}$$

FIGURE 2.3 – Typage des termes

Étant donné le terme C défini par $\lambda x : \tau.\ t$, on note $C[u]$ la substitution $t[(x)_\tau \mapsto (u)_\tau]$. Notons bien que $C[u]$ est une substitution et non l'application de la fonction C au terme u qui est, quant à elle, notée $(C\ u)$.

Définition 2.10 (β-réduction)**.** *La* β-réduction *est la relation qui associe* $C[u]$ *à* $(C\ u)$ *pour toute fonction* C *et tout terme* u*, et qui est stable par passage au contexte.*

Notons $\sim_\beta$ la clôture réflexive, symétrique et transitive de la β-réduction.

Exemple 2.11. *On a*

$$\lambda s.\ c_6\ (c_7\ s) \sim_\beta c_{42}$$

où c_n *désigne le* n*-ème entier de Church, c'est-à-dire*

$$\lambda s.\ \lambda z.\ \underbrace{s\ (\ldots s\ (s\, z)\ldots)}_{n\ fois}$$

Définissons aussi l'*η-expansion* comme la relation qui associe $\lambda x : \tau.\ t\ x$ à t lorsque x n'est pas libre dans t et qui est stable par passage au contexte. L'*η-réduction* est la relation inverse.

2.5 Typage des termes

Définition 2.12 (Typage)**.** *On note* $I \mid \Sigma \Vdash (t)_\tau$ *le prédicat affirmant que le terme* $(t)_\tau$ *est bien typé dans la signature* Σ *dont les types sont bien formés dans* I*. Formellement, ce prédicat est défini par les règles d'inférence données dans la figure 2.3. On simplifie les notations en écrivant ce prédicat* $I \mid \Sigma \Vdash t : \tau$*, ou même* $\Sigma \Vdash t : \tau$ *lorsque l'on peut facilement déduire* I *du contexte.*

Notons que tout terme ou formule bien typé ne peut pas contenir de variables de type libres. En effet, dans la figure 2.3, la règle de typage des variables exige que le type $\tau = \Sigma(x)\sigma$ accompagnant une occurrence donnée de la variable x soit une instance monomorphe du type de x dans la signature Σ : la substitution σ est telle que $\Sigma(x)\sigma$ n'a pas de variables libres. De plus, toute variable liée (par un quantificateur ou une abstraction) doit avoir un type monomorphe. Ces restrictions ne nous empêchent pas d'écrire des formules polymorphes : la généralisation est possible grâce à la règle pour $\Pi\alpha.\ f$.

Exemple 2.13. *La formule* $t := \forall x : \alpha.\ p\ x$ *n'est bien typée dans aucune signature, quels que soient les types accompagnant* t *et ses sous-termes. Par contre, la formule* $\Pi\alpha.\ \forall x : \alpha.\ p\ x$ *est bien typée dans la signature* $p : \beta \rightsquigarrow prop$.

Exemple 2.14. *Soit*

$$\begin{aligned} I &:= color : 0, list : 1 \\ \Sigma &:= red : color, \\ &\quad\ nil : list(\alpha), \\ &\quad\ cons : \alpha \rightsquigarrow list(\alpha) \rightsquigarrow list(\alpha), \\ &\quad\ is\_nil : list(\alpha) \rightsquigarrow prop \\ t_1 &:= \neg\ (is\_nil\ (cons\ red\ nil)) \\ t_2 &:= \Pi\alpha.\ \forall x : \alpha.\ \neg\ (is\_nil\ (cons\ x\ nil)) \end{aligned}$$

On a $\Sigma \Vdash t_1 : prop$ *si et seulement si* t_1 *s'écrit, en abrégeant* $list(color)$ *par* lc,

$$(\neg\ ((is\_nil)_{lc \rightsquigarrow prop}(((cons)_{color \rightsquigarrow lc \rightsquigarrow lc}\ (red)_{color})_{lc \rightsquigarrow lc}\ (nil)_{lc})_{lc})_{prop})_{prop}$$

où l'occurrence de is_nil *est accompagnée de* $list(color) \rightsquigarrow prop$ *et l'occurrence de* nil *est accompagnée de* $list(color)$. *La dérivation de typage de* t_1 *a la forme suivante*

$$\dfrac{\dfrac{\dfrac{\tau = (list(\alpha) \rightsquigarrow color)[\alpha \mapsto prop] \qquad V_\alpha(\tau) = \emptyset}{\Sigma \Vdash is\_nil : \tau} \qquad \dfrac{\vdots}{\Sigma \Vdash cons\ red\ nil : list(color)}}{\Sigma \Vdash is\_nil\ (cons\ red\ nil) : prop}}{\Sigma \Vdash \neg\ (is\_nil\ (cons\ red\ nil)) : prop}$$

en posant $\tau := list(color) \rightsquigarrow prop$. *On a* $I \mid \Sigma \Vdash t_2 : prop$ *seulement si l'occurrence de* is_nil *est accompagnée de* $list(\alpha) \rightsquigarrow prop$ *et l'occurrence de* nil *est accompagnée de* $list(\alpha)$.

Insistons sur le fait que deux termes accompagnés de types différents ne sont pas syntaxiquement égaux. Nous verrons dans le paragraphe 2.7 que l'interprétation d'un terme dépend des types qui l'accompagnent et montrons dans l'exemple 2.28 que deux termes, qui sont égaux si ce n'est pour les types qui les accompagnent, peuvent être interprétés différemment.

Exemple 2.15. *Reprenons l'exemple précédent et posons*

$$\begin{aligned} u_1 &:= (is\_nil)_{list(color) \rightsquigarrow prop}\ (nil)_{list(color)} \\ u_1 &:= (is\_nil)_{list(list(color)) \rightsquigarrow prop}\ (nil)_{list(list(color))} \end{aligned}$$

On a $\Sigma \Vdash u_1 : prop$ *et* $\Sigma \Vdash u_2 : prop$.

Insistons aussi sur le fait que la quantification de type introduit toutes les variables de type d'une formule.

Exemple 2.16. *On a*

$$\iota : 0 \mid x : \alpha \rightsquigarrow \alpha \Vdash (x)_{(\iota \rightsquigarrow \iota) \rightsquigarrow (\iota \rightsquigarrow \iota)}\ (x)_{\iota \rightsquigarrow \iota} : \iota \rightsquigarrow \iota$$

mais $t := \Pi\alpha.\ \forall x : \alpha \rightsquigarrow \alpha.\ x\ x$ *n'est bien typé dans aucune signature, quels que soient les types qui accompagnent* t *lui-même et ses sous-termes.*

Les propositions suivantes permettent de nous assurer que le typage est préservé par des opérations sur les termes, telles que celles effectuées par application de transformations logiques. Notons $FV(t)$ l'ensemble des variables libres d'un terme t et $FV(t) \subseteq \Sigma$ lorsque, pour toute variable libre x de t, on a $x \in \Sigma$.

Proposition 2.17. *Si $\Sigma \Vdash t : \tau$, alors $FV(t) \subseteq \Sigma$.*

Démonstration. Par récurrence sur la dérivation $\Sigma \Vdash t : \tau$. Nous présentons le cas de l'abstraction, de l'application et de la variable, les autres cas pouvant se déduire aisément à partir de ceux-ci.

— Cas $t = x$. On doit avoir $\tau = \Sigma(x)\sigma$, donc $x \in \Sigma$.

— Cas $t = t_1\ t_2$. Par hypothèse de récurrence, on a $FV(t_1) \subseteq \Sigma$ et $FV(t_2) \subseteq \Sigma$. On a alors :

$$FV(t) = FV(t_1) \cup FV(t_2) \subseteq \Sigma$$

— Cas $t = \lambda x : \tau'.\ u$. Par hypothèse de récurrence, on a $FV(u) \subseteq \Sigma, x : \tau'$. Puisque x n'appartient pas à Σ, on en déduit que l'ensemble $FV(u)$ privé de x est inclus dans Σ, ce que l'on voulait obtenir.

□

Les prémisses $x \notin \Sigma$ des règles de typage de chacun des quantificateurs nous assurent donc que les variables liées ne sont pas capturées lors de l'ouverture du lieur.

Proposition 2.18. *Soit une signature Σ, un terme t et une variable x fraîche par rapport à Σ et t. Pour tous types τ_1 et τ_2, si $\Sigma, x : \tau_1 \Vdash t : \tau_2$ alors $\Sigma \Vdash t : \tau_2$.*

Démonstration. Par récurrence sur t. Nous présentons le cas de l'abstraction, de l'application et de la variable, les autres cas pouvant se déduire aisément à partir de ceux-ci.

— Cas $t = y$ avec $y \neq x$. Il suffit de remarquer que $\Sigma(y)\sigma = (\Sigma, x : \tau_1)(y)\sigma$.

— Cas $t = t_1\ t_2$. On a nécessairement $\Sigma, x : \tau_1 \Vdash t_1 : \tau_2' \rightsquigarrow \tau_2$ et $\Sigma, x : \tau_1 \Vdash t_2 : \tau_2'$ pour un certain type τ_2'. On conclut par la règle de typage de l'application en utilisant les hypothèses de récurrence pour en prouver les prémisses. On a en effet que :

$$\Sigma, x : \tau_1 \Vdash t_1 : \tau_2' \rightsquigarrow \tau_2 \text{ implique } \Sigma \Vdash t_1 : \tau_2' \rightsquigarrow \tau_2$$
$$\Sigma, x : \tau_1 \Vdash t_2 : \tau_2' \text{ implique } \Sigma \Vdash t_2 : \tau_2'$$

— Cas $t = \lambda z : \tau'.\ u$. On doit avoir τ_2 de la forme $\tau' \rightsquigarrow \tau$ et $\Sigma, x : \tau_1, z : \tau' \Vdash u : \tau$ avec $V_\alpha(\tau') = \emptyset$ et $z \notin \Sigma, x : \tau_1$. Donc, par hypothèse de récurrence, $\Sigma, z : \tau' \Vdash u : \tau$. On peut appliquer la règle de typage de l'abstraction pour en déduire que $\Sigma \Vdash t : \tau_2$.

□

Cette dernière proposition nous permet notamment de prouver les propriétés de typage des substitutions suivantes. Ces propriétés sont nécessaires pour nous assurer de la bonne formation des certificats, voir par exemple la proposition 3.15.

Proposition 2.19. *Si $\Sigma \Vdash t : \tau_1$ et $\Sigma \Vdash u : \tau_2$, alors $\Sigma \Vdash t[(x)_{\tau_2} \mapsto (u)_{\tau_2}] : \tau_1$ pour toute variable x.*

Démonstration. Par récurrence sur t. Dans le cas de la variable $(x)_{\tau_2}$, on doit avoir $\tau_1 = \tau_2$ et on conclut par l'hypothèse $\Sigma \Vdash u : \tau_2$. Pour le cas de l'abstraction, il s'agit de remarquer que l'on peut écrire $(\lambda z.\ v)[x \mapsto u] = \lambda z.\ v[x \mapsto u]$, les lieurs étant définis à α-renommage près. □

Corollaire 2.20. *Si $\Sigma, x : \tau_2 \Vdash t : \tau_1$ et $\Sigma \Vdash u : \tau_2$ alors $\Sigma \Vdash t[(x)_{\tau_2} \mapsto (u)_{\tau_2}] : \tau_1$.*

2.6 Définition des tâches de preuve

Une tâche de preuve représente un séquent dans la logique d'ordre supérieur, nous explicitons la sémantique d'un tel séquent dans le paragraphe 2.7. Une tâche de preuve est donc un énoncé logique donné sous la forme de buts où il s'agit de prouver l'un d'entre eux à partir d'hypothèses, chaque but ou hypothèse étant une formule. Une *déclaration de formule* est une paire de la forme '$P : t$' où P est un identifiant et t est une formule. Les hypothèses et les buts forment des dictionnaires associant une formule à un identifiant, ce sont donc des ensembles de déclarations de formules. Une *tâche de preuve* regroupe une signature de type, une signature, des hypothèses et des buts.

Définition 2.21 (Tâche de preuve)**.** *Soit I une signature de type, Σ une signature, Γ et Δ des ensembles de déclarations de formules. La* tâche de preuve $I \mid \Sigma \mid \Gamma \vdash \Delta$ *représente le séquent où il s'agit de prouver la disjonction des formules de Δ à partir de la conjonction des formules de Γ dans la signature Σ et dans la signature de type I.*

Par ailleurs, on suppose dans la suite que pour deux déclarations d'une tâche donnée, les noms de ces déclarations sont distincts.

Exemple 2.22. *Si on se donne la tâche*

$$t_1 : prop, t_2 : prop \mid H : t_1 \vee t_2 \vdash G : t_1$$

alors on suppose que les identifiants t_1, t_2, H et G sont deux à deux distincts.

Définition 2.23 (Tâche de preuve bien typée)**.** *On dit qu'une tâche $I \mid \Sigma \mid \Gamma \vdash \Delta$ est bien typée lorsque pour toute déclaration de formule $P : t$ de Γ ou de Δ, on a $\Sigma \Vdash t : prop$.*

On notera parfois simplement $\Sigma \mid \Gamma \vdash \Delta$ (resp. $\Gamma \vdash \Delta$) lorsque l'on peut facilement déduire I (resp. et Σ) du contexte. On se permettra aussi d'omettre Γ (resp. Δ) lorsque ce dictionnaire est vide. Les identifiants des déclarations des formules d'une tâche de preuve utilisent généralement les lettres P, H quand c'est une hypothèse et G (pour *Goal*) quand c'est un but. Une tâche associe donc, à chacun de ces identifiants-ci, une formule et sa *position*, booléen indiquant si la formule est un but (booléen $true$) ou une hypothèse (booléen $false$).

Exemple 2.24. *La notation $\vdash G : t_1 \vee t_2$ désigne la tâche*

$$\emptyset \mid t_1 : prop, t_2 : prop \mid \emptyset \vdash G : t_1 \vee t_2$$

Dans cette tâche, la formule G a pour position $true$ et pour terme $t_1 \vee t_2$.

2.7 Sémantique des tâches de preuve

Nous définissons l'interprétation d'une signature de type, d'une signature et d'un ensemble de formules. Cela nous permet ensuite de définir la validité d'une tâche de preuve.

Notre formalisme s'inspire de THF0 [11], un langage pour la logique d'ordre supérieur, et de TFF1 [13], un langage pour la logique du premier ordre avec polymorphisme. En effet, nous nous plaçons dans une logique d'ordre supérieur classique où il est possible de définir des symboles de fonction polymorphes et des formules polymorphes. En revanche, dans un terme bien typé donné, on impose que les variables de terme qui sont liées soient monomorphes (condition $V_\alpha(\tau) = \emptyset$). Nous avons donc une hiérarchie de types implicite : les types monomorphes et, au-dessus, les types où les variables de type sont implicitement

quantifiées universellement. Ainsi, notre formalisme, tel un système avec polymorphisme prédicatif [33], ne permet pas d'exprimer le paradoxe de Girard.

Notre formalisme peut aussi être vu comme une application du système STT∀ [81]. Au lieu de définir une logique minimale, nous cherchons à faciliter la définition de tâches de preuve, notamment en rendant accessibles les opérateurs propositionnels habituels et en étendant notre formalisme avec des théories interprétées (voir le chapitre 4). Une autre différence notable est que dans STT∀ la spécialisation des symboles polymorphes se fait par application à un type alors que cette application est implicite dans notre formalisme.

Notre sémantique cible est donc la sémantique standard de la logique d'ordre supérieur [10], où les fonctions peuvent également prendre des arguments de type. Les modèles considérés admettent l'extensionnalité de fonctions. De cette façon deux fonctions sont égales lorsqu'elles sont égales point à point. De même, les modèles admettent l'extensionnalité booléenne où l'on ne peut pas distinguer une formule vraie d'une autre. Il est donc possible de remplacer une formule par une formule équivalente.

Nous explicitons les modèles pour une signature de type I, une signature Σ, et un ensemble S de formules bien typées dans I et Σ. Un tel modèle $\mathcal{M}$ définit une collection d'ensembles non vides $\mathbb{D}$ appelée collection de *domaines*. Supposons qu'il existe un domaine particulier à deux éléments $D_o := \{\mathcal{T}, \mathcal{F}\}$ qui interprète les formules et que, pour tous domaines D_1 et D_2, l'ensemble des fonctions (totales) de D_1 dans D_2, noté $D_1 \to D_2$, soit aussi un domaine. On dit alors que la collection de domaines $\mathbb{D}$ est *fonctionnelle*. De plus, pour chaque symbole ι d'arité n, le modèle définit une famille de domaines $\iota^{\mathcal{M}}\ D_1\ \ldots\ D_n$ indexée par n domaines $D_1, \ldots, D_n$.

La figure 2.4 définit l'interprétation $[\![\tau]\!]_\theta$ d'un type τ où θ est une *valuation de type*, c'est-à-dire une fonction qui associe un domaine à chaque variable de type.

$$
\begin{aligned}
[\![\alpha]\!]_\theta &:= \theta(\alpha) \\
[\![prop]\!]_\theta &:= D_o \\
[\![\tau_1 \rightsquigarrow \tau_2]\!]_\theta &:= [\![\tau_1]\!]_\theta \to [\![\tau_2]\!]_\theta \\
[\![\iota(\tau_1, \ldots, \tau_n)]\!]_\theta &:= \iota^{\mathcal{M}}\ [\![\tau_1]\!]_\theta\ \ldots\ [\![\tau_n]\!]_\theta
\end{aligned}
$$

FIGURE 2.4 – Interprétation des types

Une *valuation* est une fonction ξ qui, à une variable x et un domaine D, associe un élément $\xi(x, D)$ appartenant à D. Par analogie avec les substitutions, on note $f[x \mapsto v]$ la fonction qui est égale à f en tout point sauf en x où elle vaut v.

La figure 2.5 définit l'interprétation $[\![t]\!]_{\theta,\xi}$ d'un terme t où ξ est une valuation et θ est une valuation de type. Remarquons que l'interprétation $[\![t]\!]_{\theta,\xi}$ d'une formule t bien typée ne dépend pas de la valeur initiale de θ. En effet, la formule t étant bien typée, elle n'a pas de variable de type libre. On note alors cette interprétation par $[\![t]\!]_\xi$.

Définition 2.25 (Modèle)**.** *Un modèle $\mathcal{M}$ d'une signature de type I, d'une signature Σ, et d'un ensemble S de formules bien typées dans I et Σ est la donnée d'une collection fonctionnelle de domaines $\mathbb{D}$, de fonctions $\iota^{\mathcal{M}}$ pour chaque symbole ι de I, et d'une valuation ξ telles que pour toute formule t de S, on a :*

$$[\![t]\!]_\xi = \mathcal{T}$$

$$
\begin{array}{rcl}
[\![(f)_\tau]\!]_{\theta,\xi} & := & [\![f]\!]^\tau_{\theta,\xi} \\
\\
[\![\Pi\alpha.\ f]\!]^{prop}_{\theta,\xi} & := & \mathcal{T} \text{ ssi pour tout domaine } D, \text{ on a } [\![f]\!]_{\theta[\alpha\mapsto D],\xi} = \mathcal{T} \\
\\
[\![x]\!]^\tau_{\theta,\xi} & := & \xi(x, [\![\tau]\!]_\theta) \\
[\![\top]\!]^{prop}_{\theta,\xi} & := & \mathcal{T} \\
[\![\bot]\!]^{prop}_{\theta,\xi} & := & \mathcal{F} \\
[\![\neg t]\!]^{prop}_{\theta,\xi} & := & \mathcal{T} \text{ ssi } [\![t]\!]_{\theta,\xi} = \mathcal{F} \\
[\![t_1 \wedge t_2]\!]^{prop}_{\theta,\xi} & := & \mathcal{T} \text{ ssi } [\![t_1]\!]_{\theta,\xi} = \mathcal{T} \text{ et } [\![t_2]\!]_{\theta,\xi} = \mathcal{T} \\
[\![t_1 \vee t_2]\!]^{prop}_{\theta,\xi} & := & \mathcal{T} \text{ ssi } [\![t_1]\!]_{\theta,\xi} = \mathcal{T} \text{ ou } [\![t_2]\!]_{\theta,\xi} = \mathcal{T} \\
[\![t_1 \Rightarrow t_2]\!]^{prop}_{\theta,\xi} & := & \mathcal{T} \text{ ssi } [\![t_1]\!]_{\theta,\xi} = \mathcal{F} \text{ ou } [\![t_2]\!]_{\theta,\xi} = \mathcal{T} \\
[\![t_1 \Leftrightarrow t_2]\!]^{prop}_{\theta,\xi} & := & \mathcal{T} \text{ ssi } [\![t_1]\!]_{\theta,\xi} = [\![t_2]\!]_{\theta,\xi} \\
[\![t_1\ t_2]\!]^\tau_{\theta,\xi} & := & [\![t_1]\!]_{\theta,\xi}([\![t_2]\!]_{\theta,\xi}) \\
[\![\lambda x : \tau_1.\ t]\!]^{\tau_1\rightsquigarrow\tau_2}_{\theta,\xi} & := & \begin{pmatrix} D_1 \to D_2 \\ a \mapsto [\![t]\!]_{\theta,\xi[(x,D_1)\mapsto a]} \end{pmatrix} \text{ où } D_1 := [\![\tau_1]\!]_\theta \text{ et } D_2 := [\![\tau_2]\!]_\theta \\
[\![\forall x : \tau.\ t]\!]^{prop}_{\theta,\xi} & := & \mathcal{T} \text{ ssi pour tout } a \text{ dans } D := [\![\tau]\!]_\theta, \text{ on a } [\![t]\!]_{\theta,\xi[(x,D)\mapsto a]} = \mathcal{T} \\
[\![\exists x : \tau.\ t]\!]^{prop}_{\theta,\xi} & := & \mathcal{T} \text{ ssi il existe } a \text{ dans } D := [\![\tau]\!]_\theta \text{ tel que } [\![t]\!]_{\theta,\xi[(x,D)\mapsto a]} = \mathcal{T}
\end{array}
$$

FIGURE 2.5 – Interprétation des termes

Définition 2.26 (Validité d'une tâche et contre-modèle)**.** *Une tâche $T := I \mid \Sigma \mid \Gamma \vdash \Delta$ bien typée est dite* valide *quand tout modèle de I, de Σ et de chaque formule de Γ est aussi un modèle (de I, de Σ et) d'une formule de Δ. On appelle* contre-modèle *de T un modèle de I, de Σ et des formules de Γ dans lequel on a $[\![t]\!]_\xi = \mathcal{F}$ pour chaque formule t de Δ. Une tâche T est valide si et seulement si elle n'a pas de contre-modèle.*

Exemple 2.27. *On complète l'exemple 2.5 en définissant la tâche $I \mid \Sigma \mid \Gamma \vdash \Delta$ avec*

$$
\begin{array}{rl}
I := & color : 0,\ set : 1 \\
\Sigma := & red : color,\ green : color,\ blue : color, \\
& empty : set(\alpha), \\
& add : \alpha \rightsquigarrow set(\alpha) \rightsquigarrow set(\alpha), \\
& mem : \alpha \rightsquigarrow set(\alpha) \rightsquigarrow prop \\
\Gamma := & H_1 : \Pi\alpha.\ \forall x : \alpha.\ \forall y : \alpha.\ \forall s : set(\alpha).\ mem\ x\ s \Rightarrow mem\ x\ (add\ y\ s), \\
& H_2 : \Pi\alpha.\ \forall x : \alpha.\ \forall s : set(\alpha).\ mem\ x\ (add\ x\ s) \\
\Delta := & G : mem\ green\ (add\ red\ (add\ green\ empty))
\end{array}
$$

Cette tâche définit les types color et set et les symboles de terme red, green, blue, empty et add correspondants. En particulier, la déclaration de symbole de type set : 1 définit un symbole de type set d'arité 1 pour les ensembles polymorphes et la déclaration de signature $add : \alpha \rightsquigarrow set(\alpha) \rightsquigarrow set(\alpha)$ nous permet de déclarer une fonction add pouvant être utilisée pour ajouter un élément de n'importe quel type à un ensemble contenant des éléments du même type. Cette tâche définit également un prédicat mem et des hypothèses telles que mem soit vrai si son premier argument est un élément de son deuxième argument. Notamment, l'hypothèse H_2, applicable à des ensembles de n'importe quel type, exprime le fait qu'un ensemble contient l'élément qui vient de lui être ajouté. Avec le but G donné, cette tâche est valide.

Exemple 2.28. *La tâche* $\iota_1 : 0,\ \iota_2 : 0 \mid p : \alpha \rightsquigarrow prop,\ x : \alpha \mid H : p\ (x)_{\iota_1} \vdash G : p\ (x)_{\iota_2}$ *n'est pas valide. Un contre-modèle* $\mathcal{M}$ *est donné par les domaines* $\iota_1^{\mathcal{M}} = \{0\}$ *et* $\iota_2^{\mathcal{M}} = \{1\}$*, et par la valuation* ξ *telle que* $\xi(p, \iota_1^{\mathcal{M}} \to D_o)$ *est la fonction constante égale à* $\mathcal{T}$ *et que* $\xi(p, \iota_2^{\mathcal{M}} \to D_o)$ *est la fonction constante égale à* $\mathcal{F}$*.*

Notre formalisme s'attache à prouver des propriétés fonctionnelles des symboles définis dans les tâches. Pour aller dans ce sens, les symboles sont directement interprétés comme des fonctions (ou des constantes). Dans les modèles que nous considérons, la β-réduction, l'η-réduction et l'η-expansion n'associent que des termes de même interprétation. Le cas de la β-réduction découle du théorème suivant.

Théorème 2.29. *Soit* $\mathcal{M}$ *un modèle de* I, Σ *et* S*. Pour toute valuation* ξ*, toute valuation de type* θ*, tous termes* t *et* u *bien typés, toute variable* x*, si* x *et* u *ont un même type* τ *qui n'a pas de variable de type, alors :*

$$[\![t[x \mapsto u]]\!]_{\theta,\xi} = [\![t]\!]_{\theta,\xi[(x,[\![\tau]\!]_\theta)\mapsto[\![u]\!]_{\theta,\xi}]}$$

Démonstration. Par récurrence sur t. □

2.8 Transformations logiques et correction

2.8.1 Correction d'une application d'une transformation logique

Définition 2.30 (Transformation logique)**.** *Une* transformation logique *est une fonction partielle qui prend une tâche en argument et renvoie une liste de tâches. On dit qu'une transformation logique s'applique à une* tâche initiale *et produit une liste de* tâches résultantes*.*

Informellement, la liste des tâches résultantes contient les tâches qu'il nous reste à valider pour nous assurer que la tâche initiale est valide. Nous n'avons plus d'autre preuve à faire lorsque la liste est vide, on dit dans ce cas que la transformation *clôt* la tâche initiale. Une transformation logique peut ne pas être définie sur une tâche initiale et ne pas renvoyer de tâches résultantes. Dans ce cas on dit que l'application de la transformation échoue. Cette application ne nous donne pas plus d'informations sur les conditions de validité de la tâche initiale et il ne faut pas confondre ce cas avec le cas où la liste de tâches résultantes est vide.

Définition 2.31 (Correction d'une application d'une transformation logique)**.** *Une application d'une transformation à une tâche initiale* T *est dite* correcte *si, lorsqu'elle produit la liste de tâches résultantes* L *(et donc lorsqu'elle n'échoue pas), alors la validité de toutes les tâches de* L *implique la validité de* T*.*

Une définition équivalente en terme de contre-modèle est la suivante : s'il existe un contre-modèle à la tâche T, alors il existe un contre-modèle (pas nécessairement le même, comme nous le verrons dans le paragraphe suivant) à au moins une des tâches de L.

Exemple 2.32. *Soit une transformation* ϕ *qui permet de clore des buts propositionnels telle que l'application de* ϕ *à* $\vdash G : \bot$ *échoue et l'application de* ϕ *à* $t : prop \mid H : t \vdash G : t$ *renvoie la liste vide. Ces deux applications de* ϕ *sont correctes.*

2.8.2 Discussion sur la correction des transformations

Nous souhaitons insister sur la définition choisie pour la correction d'une application d'une transformation. Il est important de se rappeler que les signatures sont locales à chaque

tâche, cet aspect peut passer inaperçu au premier abord si l'on considère des transformations qui se limitent à de simples manipulations propositionnelles. Plaçons-nous dans un tel cas d'une application d'une transformation qui, appliquée à la tâche initiale T de signature Σ, produit la tâche T' avec la même signature Σ et imaginons que l'on définisse la correction de cette application par : tout contre-modèle de T (qui est donc défini sur Σ) est un contre-modèle de T'.

Par exemple si on considère la tâche

$$T := t : prop,\ u : prop \mid H : t \vdash G : t \vee u$$

que l'on transforme en éliminant la disjonction en la tâche

$$T' := t : prop,\ u : prop \mid H : t \vdash G : u$$

qui semble avoir la même signature, cette application est correcte dans ce nouveau sens. En effet si on a un contre-modèle de T alors ce même contre-modèle convient pour T'. Par contre, si on considère la tâche

$$T := t : prop,\ u : prop \mid H : t \vee u \vdash G : t$$

que l'on transforme, en remplaçant toutes les occurrences de t par sa négation, en la tâche

$$T' := t : prop,\ u : prop \mid H : \neg t \vee u \vdash G : \neg t$$

alors cette application n'est pas correcte dans ce nouveau sens. En effet, le contre-modèle de T qui interprète t en $\mathcal{F}$ et u en $\mathcal{T}$ n'est pas un contre-modèle de T'. Montrons toutefois que cette application est bien correcte au sens de la définition 2.31 : s'il existe un contre-modèle $\mathcal{M}$ de T alors il existe un contre-modèle $\mathcal{M}'$ de T'. Il suffit pour cela d'interpréter t dans $\mathcal{M}'$ en la négation de l'interprétation de t dans $\mathcal{M}$ et d'interpréter u dans $\mathcal{M}'$ en la même interprétation que celle de u dans $\mathcal{M}$.

En résumé : les signatures sont propres à une tâche de preuve et lient les symboles de cette tâche. Nous reviendrons sur cette discussion au chapitre 4 où la présence de théories interprétées conduira à une précision sur la définition de la correction des applications des transformations.

2.9 Conclusion

Ce chapitre a donné un cadre logique permettant de définir des transformations logiques et d'énoncer la propriété de correction de leurs applications. Nous souhaitons maintenant nous donner les moyens de prouver qu'une application d'une transformation est correcte. À cette fin, nous instrumentons cette dernière pour qu'elle produise également un certificat. À chaque application de la transformation, le certificat produit permet de nous assurer que l'application est correcte, c'est l'approche sceptique. Nous décrivons cette approche dans le chapitre suivant.

Chapitre 3

Certificats de noyau

Afin de certifier le résultat des transformations logiques, celles-ci sont instrumentées de sorte qu'elles renvoient également un certificat. Lorsqu'une telle transformation est appliquée, le certificat produit est vérifié, ce qui permet de nous assurer que l'application est correcte. Dans ce chapitre nous définissons les certificats de noyau de sorte qu'ils soient aussi élémentaires que possible, tout en nous assurant qu'ils conservent leur expressivité (voir le paragraphe 3.5). De plus, nous avons fait en sorte que les certificats de noyau explicitent très précisément le contexte dans lequel ils sont appliqués. Pour ces raisons, les certificats de noyau sont adéquats pour la vérification (voir le chapitre 5) et cela explique l'appellation de ces certificats : ils sont au cœur du procédé de vérification ; la preuve de la correction d'une application d'une transformation logique repose sur eux.

La verbosité des certificats de noyau, nécessaire à leur précision, pose des problèmes de robustesse et de modularité des certificats, en partie parce que ceux-ci contiennent alors des informations spécifiques aux tâches de preuve d'une application d'une transformation (voir le paragraphe 3.7). Ces informations, qui peuvent sembler redondantes au premier abord, sont fournies afin de simplifier la définition de nouveaux vérificateurs. En particulier, le second vérificateur que nous définissons (paragraphe 5.5) s'appuie sur un encodage superficiel des tâches de preuve dans Lambdapi/Dedukti [4]. Selon cet encodage, les déclarations des tâches de preuve sont implicites et il n'est alors pas aisé de retrouver les informations fournies par les certificats de noyau dans ce contexte. Notons par ailleurs que la limitation due à la verbosité des certificats de noyau a été surmontée et que notre solution est présentée au chapitre 6.

3.1 Langage des certificats de noyau

Le langage des certificats est défini par la grammaire donnée dans la figure 3.1 où les booléens sont désignés par *bool*, les formules et les termes sont désignés par *formula* et *term*, les symboles et les noms des formules sont désignés par *ident*, les types sont désignés par *type* et les tâches sont désignées par *task*. Certains constructeurs de certificat ont, pour premier champ, un booléen indiquant la position de la formule à laquelle s'applique le certificat, c'est-à-dire qu'il indique si cette formule est une hypothèse ou un but.

Une des particularités de ce format de certificat est la possibilité de laisser un "trou" dans une preuve. Cela se manifeste par le constructeur KHole, celui-ci contenant une tâche de preuve. Pour un certificat de noyau c, on appelle *feuilles* de c et on note $\mathsf{leaf}(c)$ la liste des tâches contenues dans les constructeurs KHole de c, cette liste étant obtenue en faisant

$$
\begin{array}{rcl}
\mathit{cert} & ::= & \mathsf{KHole}(\mathit{task}) \\
& | & \mathsf{KClear}(\mathit{bool}, \mathit{formula}, \mathit{ident}, \mathit{cert}) \\
& | & \mathsf{KForget}(\mathit{ident}, \mathit{cert}) \\
& | & \mathsf{KAssert}(\mathit{ident}, \mathit{formula}, \mathit{cert}, \mathit{cert}) \\
& | & \mathsf{KAxiom}(\mathit{formula}, \mathit{ident}, \mathit{ident}) \\
& | & \mathsf{KConv}(\mathit{bool}, \mathit{formula}, \mathit{ident}, \mathit{formula}, \mathit{cert}) \\
& | & \mathsf{KTrivial}(\mathit{bool}, \mathit{ident}) \\
& | & \mathsf{KSwap}(\mathit{bool}, \mathit{term}, \mathit{ident}, \mathit{cert}) \\
& | & \mathsf{KUnfoldIff}(\mathit{bool}, \mathit{term}, \mathit{term}, \mathit{ident}, \mathit{cert}) \\
& | & \mathsf{KUnfoldArr}(\mathit{bool}, \mathit{term}, \mathit{term}, \mathit{ident}, \mathit{cert}) \\
& | & \mathsf{KSlice}(\mathit{bool}, \mathit{term}, \mathit{term}, \mathit{ident}, \mathit{cert}, \mathit{cert}) \\
& | & \mathsf{KDestruct}(\mathit{bool}, \mathit{term}, \mathit{term}, \mathit{ident}, \mathit{ident}, \mathit{ident}, \mathit{cert}) \\
& | & \mathsf{KIntroQuant}(\mathit{bool}, \mathit{type}, \mathit{term}, \mathit{ident}, \mathit{ident}, \mathit{cert}) \\
& | & \mathsf{KInstQuant}(\mathit{bool}, \mathit{type}, \mathit{term}, \mathit{ident}, \mathit{ident}, \mathit{term}, \mathit{cert}) \\
& | & \mathsf{KIntroType}(\mathit{formula}, \mathit{ident}, \mathit{ident}, \mathit{cert}) \\
& | & \mathsf{KInstType}(\mathit{formula}, \mathit{type}, \mathit{ident}, \mathit{ident}, \mathit{cert})
\end{array}
$$

FIGURE 3.1 – Grammaire des certificats

un parcours infixe de c. Les feuilles d'un certificat sont censées être, à terme, des tâches résultantes de l'application de la transformation que l'on cherche à certifier. Ainsi, un certificat de noyau est destiné à représenter une preuve de la correction de cette application. La correction d'une application d'une transformation logique peut donc être certifiée sans avoir besoin de compléter la preuve par celle des tâches résultantes. Cela permet de considérer les transformations logiques indépendamment les unes des autres et offre donc la possibilité de développer les transformations certifiantes de façon modulaire. Cet aspect rend nos certificats originaux : traditionnellement les systèmes de preuve attendent que la preuve soit complète avant de la vérifier.

3.2 Sémantique des certificats

La sémantique des certificats est définie par un prédicat binaire $T \downarrow c$, reliant une tâche initiale T à un certificat c. Informellement, ce prédicat affirme que le certificat c *garantit* que la validité des feuilles de c entraîne celle de T, on dira qu'il garantit la correction de l'application de transformation correspondante. On donne la liste complète des règles d'inférence qui définissent le jugement $T \downarrow c$ dans les figures 3.2 à 3.5. Dans la figure 3.2 on trouve les règles d'identité (KHole et KAxiom) et les règles structurelles (KClear et KForget). La figure 3.3 donne les règles de la conversion tandis que la figure 3.4 donne celles de la logique propositionnelle. Enfin, la figure 3.5 donne les règles pour les quantificateurs de terme et le quantificateur de type. Les certificats et leur sémantique sont inspirés du système **LK** [54]. Puisque les formules sont contenues dans des ensembles, les règles de contraction et d'échange de celles-ci ne sont pas nécessaires. On retrouve les règles d'affaiblissement avec KClear, d'identité avec KAxiom, KTrivial et de coupure avec KAssert. En plus des règles structurelles, on ajoute KIntroType et KInstType afin de pouvoir traiter les formules polymorphes.

Remarquons que les règles qui introduisent de nouveaux noms de formules dans une tâche (certificats KAssert, KDestruct, KInstQuant et KInstType) ont une prémisse nous assurant que ces noms ne sont pas déjà présents dans les hypothèses ou les buts de cette tâche. Par

$$\frac{}{\Gamma \vdash \Delta \downarrow \mathsf{KHole}(\Gamma \vdash \Delta)}$$

$$\frac{\Gamma \vdash \Delta \downarrow c}{\Gamma, H : f \vdash \Delta \downarrow \mathsf{KClear}(false, f, H, c)} \qquad \frac{\Gamma \vdash \Delta \downarrow c}{\Gamma \vdash \Delta, G : f \downarrow \mathsf{KClear}(true, f, G, c)}$$

$$\frac{\Sigma \mid \Gamma \vdash \Delta \downarrow c \qquad x \text{ n'apparaît pas dans } \Gamma, \Delta}{\Sigma, x : \tau \mid \Gamma \vdash \Delta \downarrow \mathsf{KForget}(x, c)}$$

$$\frac{\Sigma \mid \Gamma \vdash \Delta, P : f \downarrow c_1 \qquad \Sigma \mid \Gamma, P : f \vdash \Delta \downarrow c_2 \qquad \Sigma \Vdash f : prop \qquad P \notin \Gamma \cup \Delta}{\Sigma \mid \Gamma \vdash \Delta \downarrow \mathsf{KAssert}(P, f, c_1, c_2)}$$

$$\frac{}{\Gamma, H : f \vdash \Delta, G : f \downarrow \mathsf{KAxiom}(f, H, G)}$$

FIGURE 3.2 – Sémantique des certificats (groupe identité et groupe structurel)

exemple, la règle pour KAssert a $P \notin \Gamma \cup \Delta$ pour prémisse, ce qui signifie que P ne fait pas partie des identifiants du dictionnaire obtenu par union des ensembles Γ et Δ.

Exemple 3.1. *Un certificat de la forme*

$$c := \mathsf{KDestruct}(\_, \_, \_, \_, P_1, P_2, \_)$$

ne permet jamais de certifier une application d'une transformation dont la tâche initiale contient une formule nommée P_1 ou P_2, en hypothèse ou en but.

Certains certificats ont des règles duales, une qui traite le cas d'une formule dans le cas où c'est une hypothèse, et une qui traite le cas où cette formule est un but. Par exemple, le certificat KSlice avec l'identifiant P est utilisé pour garantir la validité d'une transformation qui commence par scinder une formule P et a deux règles duales. On remarque aussi que les règles KAssert, KIntroQuant et KInstQuant font apparaître la signature Σ parce qu'elle est soit modifiée, soit utilisée. Implicitement les autres règles gardent la même signature dans les prémisses et la conclusion. Nous décrivons maintenant les différentes règles, certificat par certificat.

3.2.1 Groupe identité et groupe structurel

Nous donnons ici les règles des certificats du groupe identité et du groupe structurel décrites par la figure 3.2.

Certificat KAxiom. Le certificat $\mathsf{KAxiom}(f, H, G)$ garantit la correction d'une application d'une transformation qui clôt une tâche ayant la formule f à la fois comme but G et comme hypothèse H.

Certificats KClear et KForget. Le certificat KClear permet d'enlever une formule de Γ ou de Δ et le certificat KForget permet d'enlever un symbole de la signature qui n'est pas utilisé dans la tâche.

Exemple 3.2. *Considérons la transformation qui enlève les symboles non utilisés. En posant* $\Gamma := H_{wrong} : \forall c.\ isRed\ c,\ H_{green} : \neg(isRed\ green)$*, l'application de cette transformation à la tâche*

$$color : 0 \mid red : color,\ green : color,\ isRed : color \rightsquigarrow prop \mid \Gamma \vdash$$

produit la tâche

$$T' := color : 0 \mid green : color,\ isRed : color \rightsquigarrow prop \mid \Gamma \vdash$$

Le certificat $\mathsf{KForget}(red, \mathsf{KHole}(T'))$ *garantit que cette application est correcte.*

Certificat KHole. Le certificat KHole est utilisé afin de garantir la validité des applications des transformations qui ont des tâches résultantes.

Exemple 3.3. *Le certificat* $\mathsf{KHole}(T)$ *garantit la correction de l'application de la transformation identité à la tâche* T*. Autrement dit, pour toute tâche* T*, l'application de la transformation identité produit la liste* $[T]$ *et la correction de cette application est garantie par* $\mathsf{KHole}(T)$ *car* $T \downarrow \mathsf{KHole}(T)$*.*

Certificat KAssert. Le certificat KAssert introduit une coupure sur une formule potentiellement polymorphe.

Exemple 3.4. *Posons :*

$$\begin{aligned} I &:= \iota : 0 \\ \Sigma &:= p : \iota \rightsquigarrow prop,\ g : \iota \rightsquigarrow \iota \rightsquigarrow prop \\ \Gamma &:= H : \forall x\ y.\ g\ x\ y \Rightarrow g\ y\ x \end{aligned}$$

Considérons une application d'une transformation qui, à partir de la tâche

$$T := I \mid \Sigma \mid \Gamma \vdash G : \exists a\ b.\ p\ a \wedge g\ a\ b$$

produit les tâches

$$\begin{aligned} T_1 &:= I \mid \Sigma \mid \Gamma \vdash P : \exists a\ b.\ p\ b \wedge g\ a\ b \\ T_2 &:= I \mid \Sigma \mid \Gamma, P : \exists a\ b.\ p\ b \wedge g\ a\ b \vdash G : \exists a\ b.\ p\ a \wedge g\ a\ b \end{aligned}$$

On a $T \downarrow c$ *pour le certificat* c *défini par :*

$$\mathsf{KAssert}(P, (\exists a\ b.\ p\ b \wedge g\ a\ b), \mathsf{KClear}(true, (\exists a\ b.\ p\ a \wedge g\ a\ b), G, \mathsf{KHole}(T_1)), \mathsf{KHole}(T_2))$$

3.2.2 Certificat pour la conversion

Les règles du certificat KConv sont décrites dans la figure 3.3. Certains termes peuvent avoir la même sémantique tout en étant syntaxiquement différents. Nous souhaitons pouvoir identifier certains termes par le calcul, ce qui, dans certains cas, simplifierait grandement la certification. D'une manière générale, le problème de savoir si deux termes ont la même sémantique est un problème difficile. Pour cette raison, nous devons indiquer sous quelles conditions nous identifions les termes.

Étant donné un ensemble de règles de réécriture [40] prédéfinies, nous considérons la relation de réécriture des termes sous contexte. Nous disons que deux termes t_1 et t_2 sont *convertibles*, noté $t_1 \equiv t_2$, lorsqu'ils sont équivalents par la clôture symétrique, réflexive et

$$\frac{\Gamma, H : f_2 \vdash \Delta \downarrow c \qquad f_1 \equiv f_2}{\Gamma, H : f_1 \vdash \Delta \downarrow \mathsf{KConv}(false, f_1, H, f_2, c)}$$

$$\frac{\Gamma \vdash \Delta, G : f_2 \downarrow c \qquad f_1 \equiv f_2}{\Gamma \vdash \Delta, G : f_1 \downarrow \mathsf{KConv}(true, f_1, G, f_2, c)}$$

FIGURE 3.3 – Sémantique des certificats (conversion)

transitive de cette relation. Le certificat KConv permet donc de changer un terme d'une hypothèse ou d'un but en un terme qui lui est convertible.

Pour une signature donnée, et deux termes t et u de même type, une règle réécrivant t en u est dite *valide* lorsque pour toute interprétation de la signature, les interprétations de t et de u coïncident. On doit donc avoir :

$$[\![t]\!]_\xi = [\![u]\!]_\xi$$

Notons que les termes t et u peuvent avoir des variables libres et que leur interprétation est alors fixée par la fonction ξ.

Le certificat KConv offre un moyen pratique d'étendre notre formalisme : il est possible d'ajouter des règles de réécriture comme nous le verrons dans le paragraphe 4.3.1. L'ajout de règles devra préserver la correction de notre méthode de vérification des transformations. **Toutes les règles de réécriture de notre formalisme doivent être valides.** Dans ce cas, par passage au contexte, deux formules convertibles d'une tâche de preuve ont la même interprétation. Si nous ne considérons pour l'instant que la β-réduction, il est possible par la suite d'étendre notre formalisme avec l'η-réduction et l'η-expansion. Rappelons que le fait que la règle de β-réduction soit valide repose sur le théorème 2.29. On peut dire que notre *framework* de certification est paramétré par l'ensemble des règles de réécriture.

Exemple 3.5. *Soit*

$$t := (\lambda f : prop \rightsquigarrow prop.\ \lambda x : prop.\ f\ (f\ x))\ (\lambda x : prop.\ \bot)\ \top$$

La validité de la tâche $H : t \vdash$ est garantie par le certificat

$$\mathsf{KConv}(false, t, H, \bot, \mathsf{KTrivial}(false, H))$$

3.2.3 Groupe propositionnel

Nous décrivons ici les règles des certificats propositionnels de la figure 3.4.

Certificat KTrivial. Le certificat $\mathsf{KTrivial}(b, P)$ garantit la correction d'une application d'une transformation qui clôt une tâche ayant une formule P triviale, c'est-à-dire lorsque P est le but $\top$ (quand sa position b est $true$), ou lorsque P est l'hypothèse $\bot$ (quand sa position b est $false$).

Certificat KSwap. Le certificat KSwap traite le cas de la négation en permettant de faire passer dans les buts la négation d'une formule qui était en hypothèse (et inversement des buts vers les hypothèses).

$$\frac{}{\Gamma, H : \bot \vdash \Delta \downarrow \mathsf{KTrivial}(false, H)} \qquad \frac{}{\Gamma \vdash \Delta, G : \top \downarrow \mathsf{KTrivial}(true, G)}$$

$$\frac{\Gamma \vdash \Delta, P : t \downarrow c}{\Gamma, P : \neg t \vdash \Delta \downarrow \mathsf{KSwap}(false, t, P, c)} \qquad \frac{\Gamma, P : t \vdash \Delta \downarrow c}{\Gamma \vdash \Delta, P : \neg t \downarrow \mathsf{KSwap}(true, t, P, c)}$$

$$\frac{\Gamma, H : (t_1 \Rightarrow t_2) \wedge (t_2 \Rightarrow t_1) \vdash \Delta \downarrow c}{\Gamma, H : t_1 \Leftrightarrow t_2 \vdash \Delta \downarrow \mathsf{KUnfoldIff}(false, t_1, t_2, H, c)}$$

$$\frac{\Gamma \vdash \Delta, G : (t_1 \Rightarrow t_2) \wedge (t_2 \Rightarrow t_1) \downarrow c}{\Gamma \vdash \Delta, G : t_1 \Leftrightarrow t_2 \downarrow \mathsf{KUnfoldIff}(true, t_1, t_2, G, c)}$$

$$\frac{\Gamma, H : \neg t_1 \vee t_2 \vdash \Delta \downarrow c}{\Gamma, H : t_1 \Rightarrow t_2 \vdash \Delta \downarrow \mathsf{KUnfoldArr}(false, t_1, t_2, H, c)}$$

$$\frac{\Gamma \vdash \Delta, G : \neg t_1 \vee t_2 \downarrow c}{\Gamma \vdash \Delta, G : t_1 \Rightarrow t_2 \downarrow \mathsf{KUnfoldArr}(true, t_1, t_2, G, c)}$$

$$\frac{\Gamma, H : t_1 \vdash \Delta \downarrow c_1 \qquad \Gamma, H : t_2 \vdash \Delta \downarrow c_2}{\Gamma, H : t_1 \vee t_2 \vdash \Delta \downarrow \mathsf{KSlice}(false, t_1, t_2, H, c_1, c_2)}$$

$$\frac{\Gamma \vdash \Delta, G : t_1 \downarrow c_1 \qquad \Gamma \vdash \Delta, G : t_2 \downarrow c_2}{\Gamma \vdash \Delta, G : t_1 \wedge t_2 \downarrow \mathsf{KSlice}(true, t_1, t_2, G, c_1, c_2)}$$

$$\frac{\Gamma, H_1 : t_1, H_2 : t_2 \vdash \Delta \downarrow c \qquad H_1, H_2 \notin \Gamma \cup \Delta}{\Gamma, H : t_1 \wedge t_2 \vdash \Delta \downarrow \mathsf{KDestruct}(false, t_1, t_2, H, H_1, H_2, c)}$$

$$\frac{\Gamma \vdash \Delta, G_1 : t_1, G_2 : t_2 \downarrow c \qquad G_1, G_2 \notin \Gamma \cup \Delta}{\Gamma \vdash \Delta, G : t_1 \vee t_2 \downarrow \mathsf{KDestruct}(true, t_1, t_2, G, G_1, G_2, c)}$$

FIGURE 3.4 – Sémantique des certificats (propositionnels)

Exemple 3.6. *Pour un terme t donné, le certificat*

$$c := \mathsf{KSwap}(false, \neg t, P, \mathsf{KSwap}(true, t, P, \mathsf{KAxiom}(t, P, G)))$$

garantit que la tâche suivante est valide :

$$T := P : \neg(\neg t) \vdash G : t$$

En effet, on a $T \downarrow c$ et c n'a pas de feuille.

Certificats KDestruct et KSlice. Les certificats KDestruct et KSlice permettent de traiter le cas de la conjonction et de la disjonction.

Exemple 3.7. *Étant donné deux termes t_1 et t_2, le certificat*

$$\begin{array}{l}\mathsf{KDestruct}(false, t_1, t_2, H, H_1, H_2,\\ \quad \mathsf{KSlice}(true, t_2, t_1, G,\\ \quad\quad \mathsf{KAxiom}(t_2, H_2, G),\\ \quad\quad \mathsf{KAxiom}(t_1, H_1, G)))\end{array}$$

garantit que la tâche $H : t_1 \wedge t_2 \vdash G : t_2 \wedge t_1$ est valide

Certificats KUnfoldIff et KUnfoldArr. Le certificat KUnfoldIff gère l'équivalence en permettant de voir $t_1 \Leftrightarrow t_2$ comme $(t_1 \Rightarrow t_2) \wedge (t_2 \Rightarrow t_1)$. Le certificat KUnfoldArr gère l'implication en permettant de voir $t_1 \Rightarrow t_2$ comme $\neg t_1 \vee t_2$.

Exemple 3.8. *On peut garantir que la tâche suivante est valide*

$$P : t \Rightarrow u, H : t \vdash G : u$$

grâce au certificat :

$$\begin{array}{l}\mathsf{KUnfoldArr}(false, t, u, P,\\ \mathsf{KSlice}(false, \neg t, u, P,\\ \quad \mathsf{KSwap}(t, P, \mathsf{KAxiom}(t, H, P)),\\ \quad \mathsf{KAxiom}(u, P, G)))\end{array}$$

3.2.4 Certificats pour les quantificateurs

Nous décrivons ici les règles des certificats pour les quantificateurs de la figure 3.5.

Certificats KIntroQuant et KInstQuant. Les certificats KIntroQuant et KInstQuant permettent de garantir la correction de transformations qui manipulent des formules ayant une quantification de terme.

Exemple 3.9. *Nous reprenons l'exemple 3.4, et nous posons :*

$$T_3 := I \mid \Sigma,\ a : \iota,\ b : \iota \mid \Gamma,\ P : p\ b \wedge g\ a\ b \vdash G_2 : p\ b \wedge g\ b\ a$$

$$\frac{\Sigma, y : \tau \mid \Gamma, H : t[x \mapsto y] \vdash \Delta \downarrow c \qquad y \notin \Sigma}{\Sigma \mid \Gamma, H : \exists x : \tau.\ t \vdash \Delta \downarrow \mathsf{KIntroQuant}(false, \tau, \lambda x : \tau.\ t, H, y, c)}$$

$$\frac{\Sigma, y : \tau \mid \Gamma \vdash \Delta, G : t[x \mapsto y] \downarrow c \qquad y \notin \Sigma}{\Sigma \mid \Gamma \vdash \Delta, G : \forall x : \tau.\ t \downarrow \mathsf{KIntroQuant}(true, \tau, \lambda x : \tau.\ t, G, y, c)}$$

$$\frac{\Sigma \mid \Gamma, H_1 : \forall x : \tau.\ t, H_2 : t[x \mapsto u] \vdash \Delta \downarrow c \qquad \Sigma \Vdash u : \tau \qquad H_2 \notin (\Gamma, H_1 : \forall x : \tau.\ t) \cup \Delta}{\Sigma \mid \Gamma, H_1 : \forall x : \tau.\ t \vdash \Delta \downarrow \mathsf{KInstQuant}(false, \tau, \lambda x : \tau.\ t, H_1, H_2, u, c)}$$

$$\frac{\Sigma \mid \Gamma \vdash \Delta, G_1 : \exists x : \tau.\ t, G_2 : t[x \mapsto u] \downarrow c \qquad \Sigma \Vdash u : \tau \qquad G_2 \notin \Gamma \cup (\Delta, G_1 : \exists x : \tau.\ t)}{\Sigma \mid \Gamma \vdash \Delta, G_1 : \exists x : \tau.\ t \downarrow \mathsf{KInstQuant}(true, \tau, \lambda x : \tau.\ t, G_1, G_2, u, c)}$$

$$\frac{I, \iota : 0 \mid \Sigma \mid \Gamma \vdash \Delta, G : f[\alpha \mapsto \iota] \downarrow c \qquad \iota \notin I}{I \mid \Sigma \mid \Gamma \vdash \Delta, G : \Pi\alpha.\ f \downarrow \mathsf{KIntroType}(\Pi\alpha.\ f, G, \iota, c)}$$

$$\frac{\Gamma, H_1 : \Pi\alpha.\ f, H_2 : f[\alpha \mapsto \tau] \vdash \Delta \downarrow c \qquad V_\alpha(\tau) = \emptyset \qquad H_2 \notin (\Gamma, H_1 : \Pi\alpha.\ f) \cup \Delta}{\Gamma, H_1 : \Pi\alpha.\ f \vdash \Delta \downarrow \mathsf{KInstType}(\Pi\alpha.\ f, \tau, H_1, H_2, c)}$$

FIGURE 3.5 – Sémantique des certificats (quantificateurs)

Une application d'une transformation qui produit T_3 à partir de T_2 peut être garantie correcte grâce au certificat suivant.

$$\begin{aligned}
&\mathsf{KIntroQuant}(false, \iota, (\lambda x.\ \exists y.\ p\ y \wedge g\ x\ y), H_1, a,\\
&\mathsf{KIntroQuant}(false, \iota, (\lambda y.\ p\ y \wedge g\ a\ y), H_1, b,\\
&\mathsf{KInstQuant}(true, \iota, (\lambda x.\ \exists y.\ p\ x \wedge g\ x\ y), G, G_1, b,\\
&\mathsf{KInstQuant}(true, \iota, (\lambda y.\ p\ b \wedge g\ b\ y), G_1, G_2, a,\\
&\mathsf{KClear}(true, (\exists x\ y.\ p\ x \wedge g\ x\ y), G,\\
&\mathsf{KClear}(true, (\exists y.\ p\ b \wedge g\ b\ y), G_1,\\
&\mathsf{KHole}(T_3)))))))
\end{aligned}$$

Par ailleurs, on peut garantir que la tâche T_3 est valide à l'aide de certificats propositionnels. On en déduit que l'on peut garantir que l'application d'une transformation produisant T_1 à partir de T est correcte.

Certificats KIntroType et KInstType. Les certificats KIntroType et KInstType permettent de garantir la correction de transformations qui manipulent des formules quantifiées par des variables de type.

Exemple 3.10. *Nous reprenons l'exemple 2.14 en posant :*

$$\begin{aligned}
I &:= color : 0,\ list : 1\\
\Sigma &:= red : color,\ green : color,\\
&\quad\ nil : list(\alpha),\\
&\quad\ cons : \alpha \rightsquigarrow list(\alpha) \rightsquigarrow list(\alpha),\\
&\quad\ is_nil : list(\alpha) \rightsquigarrow prop\\
\Gamma &:= H : \Pi\alpha.\ \forall x : \alpha.\ \forall l : list(\alpha).\ \neg(is_nil\ (cons\ x\ l))\\
\Delta &:= G : \neg(is_nil\ (cons\ red\ nil))
\end{aligned}$$

On peut donner un certificat qui garantit que la tâche $T := I \mid \Sigma \mid \Gamma \vdash \Delta$ est valide en commençant par instancier l'hypothèse H. Autrement dit, en posant

$$T' := I \mid \Sigma \mid \Gamma,\ H' : \forall x : color.\ \forall l : list(color).\ \neg(is\_nil\ (cons\ x\ l)) \vdash \Delta$$

on peut garantir que T' est valide et que l'application d'une transformation qui produit T' à partir de T est correcte. Cette dernière étape peut se faire en fournissant le certificat

$$\mathsf{KInstType}((\Pi\alpha.\ \forall x.\ \forall l.\ \neg(is\_nil\ (cons\ x\ l))), color, H, H', \mathsf{KHole}(T'))$$

Le fait que la quantification de type soit explicite dans les termes nous permet de cantonner la gestion du polymorphisme aux règles KIntroType et KInstType.

Exemple 3.11. *La règle pour la disjonction dans les hypothèses*

$$\frac{\Gamma, H : t_1 \vdash \Delta \downarrow c_1 \qquad \Gamma, H : t_2 \vdash \Delta \downarrow c_2}{\Gamma, H : t_1 \vee t_2 \vdash \Delta \downarrow \mathsf{KSlice}(false, t_1, t_2, H, c_1, c_2)}$$

ne s'applique qu'à des termes t_1, t_2 qui sont monomorphes (d'après la grammaire des termes). Cette restriction est nécessaire à la correction de cette règle, de la même manière que l'on ne peut pas prouver que $\forall x.\ f\ x \vee g\ x$ implique $(\forall x.\ f\ x) \vee (\forall x.\ g\ x)$.

3.3 Notation et nommage des certificats

Les certificats font apparaître toutes les informations nécessaires pour identifier la forme des tâches auxquelles ils s'appliquent. Cela implique qu'ils sont verbeux et, par souci de concision, nous nous permettons d'omettre certains de leurs champs lorsqu'il est possible de les retrouver aisément ; en particulier le booléen donnant la position de la formule à laquelle s'applique le certificat est souvent omis.

Exemple 3.12. *Considérons une application d'une transformation qui, à partir de la tâche initiale $H : t_1 \vdash G : t_1 \wedge t_2$, produit la tâche résultante $H : t_1 \vdash G : t_2$. Lorsque le certificat*

$$\mathsf{KSlice}(true, t_1, t_2, G, \mathsf{KAxiom}(t_1, H, G), \mathsf{KHole}(H : t_1 \vdash G : t_2))$$

est associé à cette application, on s'autorise à le noter plus simplement par :

$$\mathsf{KSlice}(\_, \_, \_, G, \mathsf{KAxiom}(\_, H, G), \mathsf{KHole}(\_))$$

Dans les arbres de dérivation des certificats, le certificat de la conclusion d'une règle pourra contenir des symboles _ pour désigner les certificats des prémisses de cette règle dans leur ordre d'apparition. Cette notation nous permet d'écrire des dérivations sans avoir à répéter le certificat complet à chaque prémisse.

Exemple 3.13. *On montre que* $\vdash G : \forall q.\ q \vee \neg q \downarrow c$ *avec*

$$\begin{aligned} c := \mathsf{KIntroQuant}\big(&true, prop, (\lambda q.\ q \vee \neg q), G, q, \\ &\mathsf{KDestruct}\big(true, q, \neg q, G, P_1, P_2, \\ &\quad \mathsf{KSwap}\big(true, q, P_2, \\ &\quad\quad \mathsf{KAxiom}\big(q, P_2, P_1\big)\big)\big)\big) \end{aligned}$$

par la dérivation :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P_2 : q \vdash P_1 : q \downarrow \mathsf{KAxiom}(q, P_2, P_1)}}{\vdash P_1 : q, P_2 : \neg q \downarrow \mathsf{KSwap}(true, q, P_2, \_)}}{\vdash G : q \vee \neg q \downarrow \mathsf{KDestruct}(true, q, \neg q, G, P_1, P_2, \_)}}{\vdash G : \forall q.\ q \vee \neg q \downarrow \mathsf{KIntroQuant}(true, prop, (\lambda q.\ q \vee \neg q), G, q, \_)}$$

Les noms des constructeurs de certificat ont été choisis afin de faciliter leur compréhension dans le contexte des transformations qui les définissent. En effet, puisque les transformations partent de la tâche initiale et produisent des tâches résultantes qui impliquent la tâche initiale, il faut lire un arbre de preuve en partant de la racine et en remontant vers les feuilles. Cet arbre de preuve est matérialisé par le certificat ; nous choisissons donc les noms des différentes étapes de celui-ci en conséquence.

Exemple 3.14. *Soit une application d'une transformation qui, à partir de la tâche initiale* $\Gamma, H : t_1 \wedge t_2 \vdash \Delta$*, produit la tâche* $\Gamma, H_1 : t_1, H_2 : t_2 \vdash \Delta$*. On donne le certificat* KDestruct *à une telle application et cela qualifie bien ce que la transformation a effectué. Au contraire, il aurait été inadéquat ici de dire que la transformation a effectué l'introduction de la conjonction.*

Pour pouvoir lire un certificat (vu comme un arbre de preuve), en partant de la racine et en remontant vers les feuilles, les constructeurs de certificat qui introduisent de nouveaux noms de variables ou de types ont ces noms pour paramètres. De cette manière, les certificats décrivent précisément l'étape effectuée par l'application de la transformation qu'ils certifient, y compris les noms introduits par la transformation. Les règles qui modifient la signature ou la signature de type sont les règles de KIntroQuant et de KIntroType. Les règles correspondant à ces certificats ont été modifiées par rapport au système **LK** afin d'expliciter, dans le certificat de la conclusion, les nouveaux noms qu'elles introduisent.

3.4 Correction des certificats

Proposition 3.15. *Pour tout certificat* c *et toute tâche* T*, si* $T \downarrow c$ *et si la tâche* T *est bien typée, alors toutes les tâches apparaissant dans la dérivation de* $T \downarrow c$ *sont bien typées.*

Démonstration. Par récurrence sur l'arbre de dérivation de $T \downarrow c$, les différents cas à prouver découlent des prémisses de typage. Nous donnons la preuve pour KDestruct, KAssert et KInstQuant, les autres cas étant similaires à l'un de ces trois cas.

- Cas KAssert. La prémisse de typage $\Sigma \Vdash f : prop$ nous assure que le terme ajouté dans Γ ou dans Δ est bien une formule.
- Cas KDestruct. On sait que $\Sigma \Vdash t_1\ op\ t_2 : prop$ pour $op = \wedge$ ou $op = \vee$. On doit donc avoir $\Sigma \Vdash t_1 : prop$ et $\Sigma \Vdash t_2 : prop$, ce que l'on voulait obtenir.
- Cas KInstQuant. On doit avoir $\Sigma, x : \tau \Vdash t : prop$. La prémisse de typage $\Sigma \Vdash u : \tau$ nous permet d'appliquer le corollaire 2.20 pour conclure.

□

Cette proposition nous assure que toutes les tâches intermédiaires dans un jugement $T \downarrow c$ sont bien typées. On impose également que les tâches originales, c'est-à-dire la tâche initiale et les tâches résultantes des applications des transformations, soient bien typées. Nous supposons donc à partir de maintenant que toutes les tâches sont bien typées.

Lemme 3.16. *Soit $\mathcal{M}$ un modèle de I, Σ et S. On a, pour toute valuation de type θ, tout terme t et tout type τ :*

$$[\![t[\alpha \mapsto \tau]]\!]_{\theta,\xi} = [\![t]\!]_{\theta[\alpha \mapsto [\![\tau]\!]_\theta],\xi}$$

Démonstration. Par récurrence sur t. □

Nous avons vu que la sémantique des certificats est donnée à travers un prédicat binaire $T \downarrow c$. Nous disons alors que c garantit que la validité des feuilles de c implique la validité de T, et cette locution est justifiée par le théorème suivant.

Théorème 3.17 (Correction du prédicat $T \downarrow c$)**.** *Pour tout certificat c et toute tâche T, si $T \downarrow c$, alors la validité de toutes les feuilles de c entraîne la validité de T.*

Démonstration. La preuve se fait par récurrence sur l'arbre de dérivation de $T \downarrow c$. Comme nous le verrons dans le paragraphe 5.5.4.1, la preuve de chacun des cas de récurrence a été vérifiée par ordinateur. Nous détaillons néanmoins les différents cas à prouver qui ne découlent pas directement de la correction des règles du système **LK**.

— Cas KHole. Immédiat.

— Cas KForget. Un contre-modèle de $\Sigma, x : \tau \mid \Gamma \vdash \Delta$ est un contre-modèle de $\Sigma \mid \Gamma \vdash \Delta$.

— Cas KConv. Cela découle du fait que les règles de réécriture sont valides.

— Cas KUnfoldIff et KUnfoldArr. Ces règles sont obtenues par application de règles de **LK** successives. Par exemple, la règle de KUnfoldArr dans les buts est obtenue par la dérivation :

$$\dfrac{\dfrac{\vdash \neg t_1 \vee t_2}{\vdash t_1 \Rightarrow t_2, \neg t_1 \vee t_2}\,\text{Weak} \qquad \dfrac{\dfrac{\dfrac{\overline{t_1 \vdash t_2, t_1}}{\neg t_1, t_1 \vdash t_2}\,\neg L \qquad \overline{t_2, t_1 \vdash t_2}}{\neg t_1 \vee t_2, t_1 \vdash t_2}\,\vee \text{L}}{\neg t_1 \vee t_2 \vdash t_1 \Rightarrow t_2}\,\Rightarrow R}{\vdash t_1 \Rightarrow t_2}\,\text{Cut}$$

— Cas KIntroType. Donnons-nous un contre-modèle $\mathcal{M}$ de

$$I \mid \Sigma \mid \Gamma \vdash \Delta, G : \Pi\alpha.\ f$$

et montrons qu'il existe un contre-modèle de

$$I, \iota : 0 \mid \Sigma \mid \Gamma \vdash \Delta, G : f[\alpha \mapsto \iota]$$

Soit $\mathbb{D}$ la collection de domaines et ξ la valuation de ce contre-modèle. On sait que $[\![\Pi\alpha.\ f]\!]_{\theta,\xi} = \mathcal{F}$ pour tout θ, il existe donc un domaine D dans lequel $[\![f]\!]_{\theta[\alpha \mapsto D],\xi} = \mathcal{F}$ pour tout θ.
Définissons le modèle $\mathcal{M}'$ qui est identique au modèle $\mathcal{M}$ si ce n'est que l'on a la définition supplémentaire :

$$\iota^{\mathcal{M}'} := D$$

L'interprétation des types et des termes dans $\mathcal{M}'$, notée avec une apostrophe, coïncide avec l'interprétation dans $\mathcal{M}$ pour des types et termes dans lesquels ι n'apparaît pas. On a donc, en utilisant le lemme 3.16, pour tout θ et tout terme u :

$$\begin{aligned}\llbracket u[\alpha \mapsto \iota] \rrbracket'_{\theta,\xi} &= \llbracket u \rrbracket'_{\theta[\alpha \mapsto D],\xi} \\ &= \llbracket u \rrbracket_{\theta[\alpha \mapsto D],\xi}\end{aligned}$$

En particulier, $\llbracket f[\alpha \mapsto \iota] \rrbracket'_\xi = \mathcal{F}$, et pour toute formule v de Γ ou de Δ, $\llbracket v \rrbracket'_\xi = \llbracket v \rrbracket_\xi$, ce que l'on voulait montrer.

— Cas KInstType. Supposons que $\llbracket \Pi\alpha.\ f \rrbracket_\xi = \mathcal{T}$ et montrons que $\llbracket f[\alpha \mapsto \tau] \rrbracket_\xi = \mathcal{T}$ pour un type τ sans variable de type. Par le lemme 3.16, il s'agit de montrer que $\llbracket f \rrbracket_{\theta[\alpha \mapsto \llbracket \tau \rrbracket_\theta],\xi} = \mathcal{T}$ pour tout θ. Or, par hypothèse, on a, pour tout domaine D et tout θ, $\llbracket f \rrbracket_{\theta[\alpha \mapsto D],\xi} = \mathcal{T}$. Il suffit donc de choisir $D = \llbracket \tau \rrbracket_\theta$.

□

3.5 Expressivité des certificats

La définition des certificats de noyau vise à définir des constructeurs de certificat les plus élémentaires possible. Nous veillons également à ne pas définir de certificats de noyau qui seraient dérivables à partir d'autres certificats. L'intérêt d'une telle approche est de limiter la base de confiance ou de faciliter la vérification, comme cela est décrit plus précisément dans le chapitre 5.

Nous souhaitons également pouvoir certifier le plus d'applications de transformation possible. Nous définissons donc des certificats pour chacun des objectifs suivants :

— pour gérer la structure des séquents, les certificats KClear, KForget et KAssert
— pour clore la dérivation, si la tâche reste à prouver, le constructeur KHole, sinon KAxiom
— pour traiter les conversion entre les termes, le certificat KConv
— pour gérer $\top$ et $\bot$, le certificat KTrivial
— pour gérer la négation, le certificat KSwap
— pour gérer l'implication, le certificat KUnfoldArr et pour l'équivalence KUnfoldIff
— pour gérer la conjonction et la disjonction, KSlice et KDestruct
— pour gérer les quantificateurs, les certificats KInstQuant et KIntroQuant et pour les quantificateurs de type, les certificats KInstType et KIntroType

Le fait que les certificats de noyau soient élémentaires nous demande de les enchaîner pour pouvoir les utiliser dans différentes situations.

Exemple 3.18. *L'instance d'une des lois de De Morgan*

$$(\neg t_1 \vee \neg t_2) \Rightarrow \neg(t_1 \wedge t_2)$$

peut se démontrer dans notre formalisme en montrant par exemple que $P : \neg t_1 \vee \neg t_2 \vdash Q : \neg(t_1 \wedge t_2) \downarrow c$ *avec un certificat* c *sans feuille. Définissons*

$$\begin{aligned} c := \mathsf{KSwap}\big(&\_, \_, Q, \\ &\mathsf{KDestruct}\big(\_, \_, \_, Q, H_1, H_2, \\ &\mathsf{KSlice}\big(\_, \_, \_, P, \\ &\qquad \mathsf{KSwap}\big(\_, \_, P, \mathsf{KAxiom}(\_, H_1, P)\big) \\ &\qquad \mathsf{KSwap}\big(\_, \_, P, \mathsf{KAxiom}(\_, H_2, P)\big)\big)\big)\big) \end{aligned}$$

et montrons que $P : \neg t_1 \vee \neg t_2 \vdash Q : \neg(t_1 \wedge t_2) \downarrow c$ *:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash P : t_1 \downarrow \mathsf{KAxiom}(\_, H_1, P)}}{\Gamma, P : \neg t_1 \vdash\downarrow \mathsf{KSwap}(\_, \_, P, \_)} \quad \dfrac{\dfrac{}{\Gamma \vdash P : t_2 \downarrow \mathsf{KAxiom}(\_, H_2, P)}}{\Gamma,\ P : \neg t_2 \vdash\downarrow \mathsf{KSwap}(\_, \_, P, \_)}}{\Gamma,\ P : \neg t_1 \vee \neg t_2 \vdash\downarrow \mathsf{KSlice}(\_, \_, \_, P, \_, \_)}}{P : \neg t_1 \vee \neg t_2,\ Q : t_1 \wedge t_2 \vdash\downarrow \mathsf{Destruct}(\_, \_, \_, Q, H_1, H_2, \_)}}{P : \neg t_1 \vee \neg t_2 \vdash Q : \neg(t_1 \wedge t_2) \downarrow \mathsf{KSwap}(\_, \_, Q, \_)}$$

où l'on a posé $\Gamma := H_1 : t_1,\ H_2 : t_2$*. Une autre façon de montrer cette instance des lois de De Morgan est de montrer que* $\vdash Q : \neg(t_1 \wedge t_2) \downarrow c'$ *où* c' *est un certificat qui a pour seule feuille la tâche* T' *définie par :*

$$T' := \vdash P : \neg t_1 \vee \neg t_2$$

Il est possible de définir ce certificat c' *par coupure sur la formule* $\neg t_1 \vee \neg t_2$*, en réutilisant le certificat* c *précédent :*

$$\begin{aligned} c' := \mathsf{KAssert}(&P, \neg t_1 \vee \neg t_2, \\ &\mathsf{KClear}(\_, \_, Q, \mathsf{KHole}(T')), \\ &c) \end{aligned}$$

On peut ainsi définir de nouveaux certificats pour des transformations plus complexes tout en gardant un nombre réduit de certificats élémentaires sur lesquels ils sont construits. Le certificat KAssert est très utile dans ces constructions, il permet en particulier d'introduire la formule sous la forme voulue.

Exemple 3.19. *Il est possible d'obtenir un certificat pour renommer une formule. Étant donné un certificat* c*, il existe un certificat* c' *qui permet d'obtenir une dérivation de la forme*

$$\dfrac{\dfrac{\Sigma \mid \Gamma \vdash \Delta, G_2 : f \downarrow c}{\vdots}}{\Sigma \mid \Gamma \vdash \Delta, G_1 : f \downarrow c'}$$

Dans la suite, on notera ce certificat $\mathsf{krenameGoal}(f, G_1, G_2, c)$*. En omettant* Σ*,* Γ *et* Δ *dans les séquents, on le définit en remarquant qu'on a la dérivation suivante*

$$\dfrac{\dfrac{\vdash G_2 : f \downarrow c}{\vdash G_1 : f, G_2 : f \downarrow \mathsf{KClear}(\_, \_, G_1, c)} \quad \dfrac{}{G_2 : f \vdash G_1 : f \downarrow \mathsf{KAxiom}(\_, G_2, G_1)} \quad \Sigma \Vdash f : \mathit{prop}}{\vdash G_1 : f \downarrow \mathsf{KAssert}(G_2, f, \_, \_)}$$

3.6 Transformations certifiantes

Définition 3.20 (Transformation certifiante)**.** *Une transformation* certifiante ϕ *est une transformation logique qui, à partir d'une tâche initiale* T*, produit un certificat* c *en plus de la liste de tâches résultantes* L*.*

On rappelle ici que vérifier une application d'une transformation logique ϕ à une tâche T renvoyant une liste de tâches L revient à s'assurer que si toutes les tâches de L sont valides, alors T est valide également. En toute généralité c'est un problème indécidable : étant donné une formule t, en prenant $T := \vdash G : t$ et $L := []$, il s'agit de décider de la validité de t.

Définition 3.21 (Certificat adapté)**.** *Soit une application d'une transformation ϕ à la tâche T produisant la liste de tâches L. On dit que c est* adapté *à cette application lorsque* $\mathsf{leaf}(c) \subseteq L$ *et* $T \downarrow c$*. On dit aussi que c* certifie *cette application.*

Si on a obtenu un certificat adapté pour une application d'une transformation, alors, d'après le théorème 3.17, cette application est correcte. Vérifier ce prédicat consiste à appliquer les règles de la sémantique des certificats. On a donc limité la recherche de la preuve à l'application de ces règles, ce qui représente un problème algorithmiquement plus simple.

Exemple 3.22. *Soit une application d'une transformation qui, à partir de la tâche initiale* $T := H_1 : t_1, H_2 : t_2 \vdash \Delta$*, produit la tâche* $T' := H : t_1 \wedge t_2 \vdash \Delta$ *et le certificat :*

$$\begin{aligned} c := \mathsf{KAssert}\big(&H, t_1 \wedge t_2, \\ &\mathsf{KSlice}\big(\_, \_, \_, H, \mathsf{KAxiom}\big(\_, H_1, H\big), \mathsf{KAxiom}\big(\_, H_2, H\big)\big) \\ &\mathsf{KClear}\big(\_, \_, H_1, \mathsf{KClear}\big(\_, \_, H_2, \mathsf{KHole}\big(T'\big)\big)\big)\big) \end{aligned}$$

Il suffit donc de vérifier qu'on a $T \downarrow c$ pour s'assurer que cette application est correcte, ce qui ne demande pas de faire une recherche de preuve.

On a donc déplacé le problème : au lieu d'effectuer une recherche de preuve, il s'agit à la place de définir des transformations qui produisent des certificats. Cette approche ne contraint pas la vérification à être complète ; si un certificat est inadapté pour une application de ϕ, on ne peut pas conclure que cette application est incorrecte. En pratique cela permet une certaine incrémentalité dans l'implémentation car il est possible de commencer par définir des transformations certifiantes produisant des certificats inadaptés dans certains cas, puis de les affiner lorsqu'il devient intéressant de couvrir plus de cas.

Exemple 3.23. *Supposons que l'on veuille rendre certifiante la transformation* case_simple *qui produit*

- *la liste* $L := [\Gamma, H : t_1 \vdash \Delta \ ; \ \Gamma, H : t_2 \vdash \Delta]$ *lorsqu'elle est appliquée à une tâche T de la forme* $\Gamma, H : t_1 \vee t_2 \vdash \Delta$
- *la liste* $L := [\Gamma, H_1 : t_1, H_2 : t_2 \vdash \Delta]$ *lorsqu'elle est appliquée à une tâche T de la forme* $\Gamma, H : t_1 \wedge t_2 \vdash \Delta$*.*

Une première approche est de renvoyer le certificat

$$\mathsf{KSlice}(\_, \_, \_, H, \mathsf{KHole}(\_), \mathsf{KHole}(\_))$$

dans tous les cas. On obtient alors une transformation certifiante et ce certificat est adapté lorsque T est de la première forme. Pour pouvoir couvrir tous les cas il faudrait, à la place du certificat précédent, renvoyer

$$\mathsf{KDestruct}(\_, \_, \_, H, H_1, H_2, \mathsf{KHole}(\_))$$

lorsque T est de la deuxième forme.

3.7 Limitations des certificats de noyau

Les certificats de noyau sont conçus pour être théoriquement simples afin de faciliter leur vérification (voir le chapitre 5). En dépit de leur expressivité, cela limite, en pratique, l'instrumentation des transformation logiques. Nous présentons ces limitations dans ce paragraphe mais tenons à préciser qu'elles ont été surmontées en introduisant un nouveau type de certificat comme décrit dans le chapitre 6.

Verbosité. Les certificats de noyau sont volontairement très précis : l'objectif est de décrire précisément la forme de la tâche à laquelle ils s'appliquent, ce qui les rend adéquats pour la vérification. En outre, les certificats de noyau sont élémentaires, ce qui accentue leur verbosité. Cela explique le fait que la production de ces certificats soit laborieuse et c'est pourquoi, dans ce chapitre, nous avons omis certains arguments des certificats de noyau.

Modularité. Les certificats de noyau sont spécifiques à la tâche initiale et aux tâches résultantes auxquelles ils s'appliquent. Il est donc difficile de définir des certificats indépendamment les uns des autres et indépendamment des transformations et, en pratique, cela limite leur modularité.

Exemple 3.24. *Le certificat*

$$\mathsf{KSlice}(false, t_1, t_2, H, \mathsf{KHole}(\Gamma, H : t_1 \vdash \Delta), \mathsf{KHole}(\Gamma, H : t_2 \vdash \Delta))$$

est adapté à une application d'une transformation qui, à partir de la tâche

$$\Gamma, H : t_1 \vee t_2 \vdash \Delta \text{ renvoie la liste de tâches } [(\Gamma, H : t_1 \vdash \Delta); (\Gamma, H : t_2 \vdash \Delta)]$$

mais pas à une application d'une transformation qui, à partir de la tâche

$$\Gamma, H : t_1' \vee t_2 \vdash \Delta \text{ renvoie la liste de tâches } [(\Gamma, H : t_1' \vdash \Delta); (\Gamma, H : t_2 \vdash \Delta)]$$

avec $t_1' \neq t_1$. *Ce certificat n'est pas non plus adapté à une application d'une transformation qui, à partir de la tâche*

$$\Gamma', H : t_1 \vee t_2 \vdash \Delta \text{ renvoie la liste de tâches } [(\Gamma', H : t_1 \vdash \Delta); (\Gamma', H : t_2 \vdash \Delta)]$$

avec $\Gamma' \neq \Gamma$.

3.8 Conclusion

Malgré l'expressivité des certificats de noyau, certaines transformations logiques ne sont pas aisément instrumentées. Cette limitation, surmontée dans la suite dans le chapitre 6, est due au fait qu'ils ont été conçus de sorte à être très détaillés et précis, facilitant ainsi leur vérification. Avant de présenter notre méthode de vérification dans le chapitre 5, nous décrivons les extensions que nous apportons à notre logique : les théories interprétées.

Chapitre 4

Théories interprétées

Nous avons décrit le formalisme dans lequel nous nous plaçons pour certifier les transformations logiques. Ce formalisme comprend la logique utilisée : la logique d'ordre supérieur avec polymorphisme explicite en forme prénexe. Dans ce cadre nous souhaitons pouvoir définir les théories sur lesquelles s'appuient les transformations. Nous commençons par donner notre méthode pour étendre notre formalisme avec des théories interprétées puis nous l'appliquons, dans le paragraphe 4.2, à la théorie de l'égalité, et, dans le paragraphe 4.3, à l'arithmétique entière des entiers relatifs.

4.1 Définition de nouvelles théories

4.1.1 Axiomatisation d'une théorie

Il est possible de définir une nouvelle théorie dans une tâche en déclarant les symboles de terme, les symboles de type et les axiomes de cette théorie.

Exemple 4.1. *On a vu dans l'exemple 2.27 qu'il est possible de déclarer de nouveaux types et constantes de ces types. Le type color est ainsi déclaré dans la signature de type et les constantes red, green et blue sont déclarées dans la signature. Il est aussi possible de déclarer des fonctions et de les axiomatiser, comme c'est fait dans ce même exemple pour le type set et les fonctions add et mem avec les axiomes H_1 et H_2.*

4.1.2 Intérêt des théories interprétées

Dans certaines théories il existe des procédures de décision plus efficaces qu'une recherche générique de preuve dans une théorie axiomatique. De telles procédures peuvent être mises en place aussi bien par des prouveurs automatiques externes que par des transformations logiques. Afin de valoriser les raisonnements dédiés à ces théories, nous souhaitons également proposer des certificats dédiés. Autrement dit, il s'agit d'éviter d'obliger les certificats à exprimer une preuve uniquement à partir d'axiomes de la théorie.

Exemple 4.2. *Plaçons-nous dans une théorie des entiers naturels pour laquelle nous axiomatisons la comparaison stricte. Pour simplifier la lecture, la fonction de comparaison $<$*

utilise la notation infixe.

$$\begin{aligned} I &:= int : 0 \\ \Sigma &:= z : int,\ s : int \rightsquigarrow int,\ < : int \rightsquigarrow int \rightsquigarrow prop \\ \Gamma &:= H_{zs} : \forall\ y.\ z < s\ y, \\ &\quad\ H_{ss} : \forall\ x\ y.\ x < y \Rightarrow s\ x < s\ y \end{aligned}$$

On note $\widetilde{n}$ l'application de la n-ième itération de s à z, c'est-à-dire la représentation de l'entier n dans cette théorie. Dans ce contexte, la preuve de validité de certaines tâches, telle que celle qui a pour but $\widetilde{17} < \widetilde{42}$, peut devenir rapidement fastidieuse. On souhaite pouvoir résoudre efficacement des buts arithmétiques, quitte à devoir fixer les constantes et les opérateurs de cette théorie.

4.1.3 Ajustement de la notion de correction

Comme discuté dans le paragraphe 2.8.2, il est important de considérer les signatures comme locales aux tâches pour définir la correction d'une application d'une transformation. La situation change en présence de théories interprétées. Par exemple, supposons que l'on transforme la tâche $T := \vdash G : \bot$ trivialement fausse en la tâche T' suivante

$$int : 0 \mid x : int,\ \leq : int \rightsquigarrow int \rightsquigarrow prop \mid\ \vdash G : x \leq x$$

avec l'intention d'utiliser une théorie des entiers. Au sens de la définition 2.31, cette application est correcte. En effet, il existe un contre-modèle de T' : on peut interpréter le symbole $\leq$ par une relation qui n'est pas réflexive. L'erreur est de considérer tous les modèles possibles là où on voulait fixer l'interprétation de $\leq$.

La situation est donc différente en présence de théories interprétées : les symboles de ces théories ne doivent pas être locaux aux tâches mais prédéfinis. Au niveau de la sémantique des tâches, nous fixons l'interprétation de ces symboles prédéfinis une fois pour toutes. Tous les modèles considérés doivent interpréter ces symboles de cette même façon. Dans l'exemple ci-dessus, l'application n'est alors plus considérée comme correcte.

4.1.4 Définition d'une théorie interprétée

Afin de pouvoir distinguer les symboles interprétés des variables, nous réservons certains identifiants. Cela permet d'éviter que des symboles interprétés soient manipulés par les transformations et les certificats. Dans certains cas, il sera aussi plus pratique d'étendre la grammaire pour décrire ces symboles interprétés. Nous étendons également le typage pour prendre en compte les symboles interprétés lorsque nous en définissons de nouveaux. Ces symboles n'étant pas locaux aux tâches, ils n'apparaissent pas comme des éléments de la signature. Pourtant, pour pouvoir typer leurs occurrences dans notre système de type, nous étendons sur les symboles interprétés la définition des types bien formés (définition 2.4) ainsi que la notation $\Sigma(x)$ dans les règles de typage (figure 2.3).

Ces considérations nous permettent de certifier des transformations dans lesquelles apparaissent des symboles interprétés sans utiliser leurs propriétés spécifiques. La preuve de correction ainsi obtenue est composable avec une preuve de validité des tâches résultantes pour une interprétation standard de cette théorie (voir le paragraphe 4.1.3 précédent). Pour certifier une transformation qui utilise les propriétés de ces symboles interprétés, nous donnons de nouveaux certificats spécifiques à ces symboles. Ces symboles obtiennent leurs propriétés par extension des certificats de noyau et bénéficient alors de règles dédiées.

4.2 Théorie de l'égalité polymorphe

Nous souhaitons voir l'égalité comme un symbole interprété, et nous définissons cette théorie de façon à simplifier la certification des transformations s'appliquant à des tâches dans lesquelles apparaît ce symbole. Nous utilisons la notation infixe habituelle pour l'égalité, de sorte qu'on écrit $x = y$ pour désigner l'application de l'égalité à x, elle-même appliquée à y. Le symbole $=$ est donc un identifiant réservé qui n'apparaît pas dans la signature Σ, ayant le type $\alpha \rightsquigarrow \alpha \rightsquigarrow \mathit{prop}$. Rappelons que ce type est néanmoins donné par $\Sigma(=)$ pour l'application des règles de typage.

Les modèles que nous considérons pour les tâches interprètent ce symbole comme l'égalité mathématique. Plus précisément, pour tout modèle ayant pour valuation ξ et tout domaine D de ce modèle, on impose que $\xi(=, D \to D \to D_o)$ soit la relation d'égalité.

Exemple 4.3. *La tâche* $\iota : 0 \mid x : \iota \mid \vdash G : x = x$ *est valide.*

Pour refléter les propriétés habituelles de l'égalité, nous augmentons nos certificats de noyau de deux constructeurs :

$$\mathsf{KEqRefl}(\mathit{term}, \mathit{ident})$$
$$\mathsf{KRewrite}(\mathit{bool}, \mathit{term}, \mathit{term}, \mathit{term}, \mathit{ident}, \mathit{ident}, \mathit{cert})$$

Les règles d'inférence de la sémantique des certificats qui sont associées à ces constructeurs sont les suivantes :

$$\frac{}{\Gamma \vdash \Delta, G : t = t \downarrow \mathsf{KEqRefl}(t, G)}$$

$$\frac{\Gamma, H : t_1 = t_2, H' : t[t_2] \vdash \Delta \downarrow c}{\Gamma, H : t_1 = t_2, H' : t[t_1] \vdash \Delta \downarrow \mathsf{KRewrite}(\mathit{false}, t_1, t_2, t, H, H', c)}$$

$$\frac{\Gamma, H : t_1 = t_2 \vdash \Delta, G : t[t_2] \downarrow c}{\Gamma, H : t_1 = t_2 \vdash \Delta, G : t[t_1] \downarrow \mathsf{KRewrite}(\mathit{true}, t_1, t_2, t, H, G, c)}$$

Le certificat KEqRefl garantit donc la validité d'une tâche dont un des buts est obtenu par réflexivité de l'égalité. Le certificat KRewrite garantit la validité d'une transformation qui commence par réécrire dans une formule une égalité qui se trouve en hypothèse de la tâche initiale. On vérifie que ces nouvelles règles préservent le bon typage des tâches et on complète aisément la preuve du théorème 3.17 avec le cas du certificat KEqRefl.

Montrons maintenant le cas de récurrence KRewrite du théorème 3.17. Étant donné que le terme $t_1 = t_2$ est bien typé, on a nécessairement un type τ monomorphe permettant de typer à la fois t_1 et t_2. Le terme $t[t_1]$ n'est défini que lorsque t est une abstraction et la condition de typage de cette substitution nous donne la forme de t : c'est $\lambda x : \tau.\ u$ pour un certain terme u. Supposons donné un contre-modèle de la conclusion et prouvons que c'est un contre-modèle de la prémisse. Il suffit de montrer que, pour toute valuation de type θ

$$[\![t[t_1]]\!]_{\theta,\xi} = [\![t[t_2]]\!]_{\theta,\xi}$$

On a, par le théorème 2.29 :

$$\begin{aligned} [\![t[t_1]]\!]_{\theta,\xi} &= [\![u[x \mapsto t_1]]\!]_{\theta,\xi} \\ &= [\![u]\!]_{\theta,\xi[(x,[\![\tau]\!]_\theta) \mapsto [\![t_1]\!]_{\theta,\xi}]} \end{aligned}$$

De même, on a :

$$[\![t[t_2]]\!]_{\theta,\xi} = [\![u]\!]_{\theta,\xi[(x,[\![\tau]\!]_\theta)\mapsto[\![t_2]\!]_{\theta,\xi}]}$$

On conclut en remarquant que t_1 et t_2 ont la même interprétation et que l'on peut donc remplacer $[\![t_1]\!]_{\theta,\xi}$ par $[\![t_2]\!]_{\theta,\xi}$.

Ces nouveaux constructeurs nous permettent de définir des certificats exprimant d'autres propriétés attendues de l'égalité : symétrie, transitivité et congruence (voir l'exemple 6.29).

Exemple 4.4. *On axiomatise la théorie des groupes comme extension de la théorie de l'égalité et on pose :*

$$\begin{aligned}
I &:= \iota : 0\\
\Sigma &:= e : \iota,\\
&\quad\ op : \iota \rightsquigarrow \iota \rightsquigarrow \iota,\\
&\quad\ inv : \iota \rightsquigarrow \iota\\
\Gamma &:= H_1 : \forall a.\ a = op\ e\ a,\\
&\quad\ H_2 : \forall a.\ a = op\ a\ e,\\
&\quad\ H_3 : \forall a.\ e = op\ a\ (inv\ a),\\
&\quad\ H_4 : \forall a.\ e = op\ (inv\ a)\ a\\
\Delta &:= G : inv\ e = e\\
\Delta_1 &:= G : inv\ e = op\ e\ (inv\ e)\\
\Delta_2 &:= G : op\ e\ (inv\ e) = op\ e\ (inv\ e)
\end{aligned}$$

On peut montrer que la tâche $T := I \mid \Sigma \mid \Gamma \vdash \Delta$ est valide

1. *en montrant que si la tâche $T_1 := \Gamma \vdash \Delta_1$ est valide, alors la tâche T est valide,*
2. *en montrant que si la tâche $T_2 := \Gamma \vdash \Delta_2$ est valide, alors la tâche T_1 est valide,*
3. *et en montrant que la tâche T_2 est valide.*

Pour la première étape, on donne le certificat

$$\begin{aligned}
&\mathsf{KInstQuant}(false, \iota, (\lambda a.\ e = op\ a\ (inv\ a)), H_3, H, e,\\
&\qquad \mathsf{KRewrite}(true, e, op\ e\ (inv\ e), (\lambda x.\ inv\ e = x), H, G,\\
&\qquad \mathsf{KClear}(false, e = op\ e\ (inv\ e), H,\\
&\qquad \mathsf{KHole}(T_1))))
\end{aligned}$$

La deuxième étape est similaire à la première et la dernière étape est simplement donnée par le certificat KEqRefl.

4.3 Théorie de l'arithmétique entière des entiers relatifs

Nous définissons un nouveau symbole de type nommé *int* d'arité 0. On réserve également les identifiants des opérateurs habituels sur les entiers avec le type attendu. On écrit $\Sigma(+) = int \rightsquigarrow int \rightsquigarrow int$ pour l'application des règles de typage et on type les opérateurs $*$ et $-$ par $int \rightsquigarrow int \rightsquigarrow int$, le moins unaire (que l'on note également $-$ par abus de notation) par $int \rightsquigarrow int$ et les opérateurs $<$, $\leq$, $>$ et $\geq$ par $int \rightsquigarrow int \rightsquigarrow prop$. On utilise les notations usuelles pour ces opérateurs arithmétiques.

Au lieu de réserver un identifiant pour chaque littéral entier, nous complétons la grammaire des termes :

$$\begin{array}{rcll} \mathit{termNode} & ::= & \ldots & \\ & | & i & \text{littéral entier} \end{array}$$

Le système de typage des termes doit alors prendre en compte cette nouvelle construction de la grammaire et on type chaque littéral entier par *int*.

Les modèles que nous considérons interprètent le type *int* par l'ensemble $\mathbb{Z}$, ils interprètent les symboles $+, *, -, -$ (unaire), $<, \leq, >, \geq$ par les opérateurs sur $\mathbb{Z}$ correspondants et interprètent chaque littéral entier par l'entier relatif correspondant.

4.3.1 Transformations arithmétiques et conversions

Afin de simplifier la certification de transformations effectuant des calculs sur les entiers, nous définissons de nouvelles règles de réécriture. Il est ainsi possible de certifier directement certaines de ces manipulations par le certificat KConv, là où il aurait fallu donner un certificat en plusieurs étapes si on avait dû instancier puis appliquer différents axiomes sur les opérateurs entiers.

Nous ajoutons la règle réécrivant $i_1 + i_2$ en i lorsque la somme des entiers représentés par les littéraux i_1 et i_2 donne l'entier représenté par le littéral i. Nous ajoutons aussi, de façon similaire, des règles de réécriture pour les autres opérateurs arithmétiques ($*$, $-$, $-$ unaire, $<, \leq, >$ et $\geq$) lorsqu'ils sont appliqués à des littéraux entiers. Nous ajoutons également les règles de réécriture suivantes :

— $0 + x$ se réécrit en x
— $x + 0$ se réécrit en x
— $x - 0$ se réécrit en x
— $0 * x$ se réécrit en 0
— $x * 0$ se réécrit en 0
— $1 * x$ se réécrit en x
— $x * 1$ se réécrit en x

Il est clair que toutes ces règles sont valides.

Exemple 4.5. *L'application de la transformation qui clôt la tâche initiale* $\vdash G : t$ *où* t *est défini par* $(\lambda x : int.\ x + 2)\ 3 < 7$ *est certifiée par* $\mathsf{KConv}(true, t, G, \top, \mathsf{KTrivial}(true, G))$ *car on a* $t \equiv (3 + 2 < 7) \equiv (5 < 7) \equiv \top$.

Remarquons que le choix des règles de réécriture est arbitraire dans le sens où il est tout à fait possible d'ajouter des règles de réécriture valides et de permettre de certifier des transformations qu'il était alors impossible de certifier. Dans tous les cas, certaines applications de transformations arithmétiques resteront inaccessibles à la certification.

Exemple 4.6. *Il n'est pas possible de certifier l'application de la transformation qui clôt la tâche* $\vdash G : x - 2 = (x - 1) - 1$ *mais cela devient possible en ajoutant la règle réécrivant* $x - (y + z)$ *en* $(x - y) - z$.

D'une manière générale, nous cherchons à améliorer l'expressivité des certificats : nous souhaitons que le plus possible d'applications de transformations soient accessibles à la certification. Pour des raisons pratiques, il n'est pas satisfaisant de définir un nouveau constructeur de certificat pour chaque application que nous souhaiterions rendre accessible. En revanche, en ce qui concerne le raisonnement par récurrence, nous allons voir que l'on améliore grandement l'expressivité des certificats et que nous n'avons à ajouter qu'un seul constructeur.

4.3.2 Transformations d'induction sur les entiers relatifs

Supposons donnée une transformation permettant de faire un raisonnement par induction sur les entiers. Une telle transformation est difficilement applicable automatiquement et représente donc un exemple type d'une transformation interactive, ce qui la rend intéressante à certifier. Afin de pouvoir faire appel à la version forte du principe de récurrence sur les entiers, nous ajoutons aux certificats de noyau le constructeur :

$$\mathsf{KInduction}(\mathit{ident}, \mathit{ident}, \mathit{ident}, \mathit{term}, \mathit{term}, \mathit{formula}, \mathit{cert}, \mathit{cert})$$

ainsi que la règle :

$$\frac{\begin{array}{c}\Sigma,\ i : \mathit{int} \mid \Gamma,\ H_i : i \leq a \vdash \Delta, G : t[i] \downarrow c_{base} \qquad i \text{ n'apparaît pas dans } \Gamma \text{ ou } \Delta \\ \Sigma \Vdash a : \mathit{int} \qquad \Sigma,\ i : \mathit{int} \mid \Gamma,\ H_i : a < i, H_{rec} : \forall n : \mathit{int}.\ n < i \Rightarrow t[n] \vdash \Delta,\ G : t[i] \downarrow c_{rec}\end{array}}{\Sigma,\ i : \mathit{int} \mid \Gamma \vdash \Delta,\ G : t[i] \downarrow \mathsf{KInduction}(G, H_i, H_{rec}, i, a, t, c_{base}, c_{rec})}$$

Le certificat KInduction permet de certifier une application d'une transformation qui commence par faire une induction forte sur un entier à partir d'une borne donnée (a dans la règle). On suppose que, mis à part le but donné, la tâche initiale n'a pas d'occurrence de l'entier sur lequel l'induction est faite. Cette restriction ne nuit pas à l'expressivité des certificats : il est toujours possible de généraliser les déclarations faisant intervenir l'entier sur lequel est faite l'induction comme expliqué dans le paragraphe 7.5. En ce sens, cette restriction a pour objectif de définir des certificats de noyau élémentaires.

4.4 Théories interprétées et formalisme logique

La définition d'une nouvelle théorie interprétée répond à un besoin d'efficacité pour résoudre une catégorie de problèmes. Pour répondre à ce besoin, nous avons décrit comment étendre notre formalisme, et nous avons appliqué cette méthode aux entiers relatifs et à l'égalité. Notre formalisme étant maintenant fixé, nous nous intéressons à la vérification d'applications de transformations logiques certifiantes.

Chapitre 5

Vérification de certificats

La vérification d'une application d'une transformation certifiante se fait à l'aide d'un vérificateur, procédure qui indique si la correction d'une application donnée est garantie. La confiance apportée par cette approche (l'approche sceptique) peut être renforcée lorsque le vérificateur est lui-même formellement vérifié ou suffisamment réduit et simple pour que l'inspection de son code soit envisageable.

L'utilisation de certificats offre une grande flexibilité par rapport au code des transformations car ce code n'est pas directement lié à une preuve de sa correction et peut changer sans que cela ne requiert nécessairement de modifier également cette preuve. Par ailleurs, cette approche offre également une flexibilité au niveau de la vérification effectuée. En effet, les certificats sont à voir comme des indications pour la vérification d'une application d'une transformation, indication que les vérificateurs peuvent suivre de la manière qui leur convient. Nous utilisons à profit, dans ce chapitre, ce deuxième aspect de la flexibilité offerte par les certificats.

Nous commençons, dans le paragraphe 5.1, par donner le cadre général des vérificateurs puis nous expliquons comment ils s'intègrent à notre travail dans le paragraphe 5.2. Nous détaillons ensuite, dans les paragraphes 5.3 et 5.5, les deux vérificateurs que nous avons définis, le deuxième s'appuyant sur l'encodage présenté au paragraphe 5.4.

5.1 Cadre général des vérificateurs

Soit une application d'une transformation certifiante ϕ qui, à partir de la tâche initiale T, produit la liste de tâches résultantes $L := [T_1;\ \ldots;\ T_n]$ et le certificat c. Nous souhaitons vérifier que cette application est correcte. Pour ce faire, nous implémentons un *vérificateur*, procédure qui prend en entrée T, L et c et qui renvoie un booléen. L'intention est que si le résultat est vrai et que toutes les tâches $T_1, \ldots, T_n$ sont valides, alors T est valide également.

Définition 5.1 (Vérificateur correct)**.** *Un vérificateur est dit* correct *si, à chaque fois qu'il renvoie true sur T, L et c, alors la validité des tâches de L implique la validité de T.*

Remarquons que le certificat n'intervient pas dans la définition de la correction du vérificateur. Le certificat fournit des indications au vérificateur mais celui-ci est libre de s'en servir à sa convenance. Une première façon d'implémenter un vérificateur est de chercher à vérifier que $T \downarrow c$ et qu'on a bien $\mathsf{leaf}(c) \subseteq L$, ce qui nous permet de conclure par le théorème 3.17. Cette approche, bien que justifiée, offre peu de flexibilité dans la définition d'un vérificateur.

Dans ce chapitre, les vérificateurs ne vont pas établir directement $T \downarrow c$ mais plutôt se servir de la structure de c pour justifier que la validité des tâches de L entraîne celle de T.

Nous nous intéressons également à la complétude des vérificateurs dans le sens suivant : nous souhaitons avoir le moins de faux négatifs possible, c'est-à-dire d'applications correctes telles que le vérificateur renvoie $false$. Pour aller dans ce sens, nous souhaitons que les vérificateurs que nous définissons renvoient $true$ lorsque le certificat renvoyé est adapté. Cela nous mène à la définition suivante.

Définition 5.2 (Vérificateur adapté)**.** *Un vérificateur est dit* adapté *lorsque : pour toute tâche T, liste L et certificat c, si $T \downarrow c$ et si* $\mathsf{leaf}(c) \subseteq L$*, alors le vérificateur renvoie true.*

Considérons une application d'une transformation certifiante qui renvoie c et L à partir d'une tâche T. La figure 5.1 résume les différents cas :

— si l'application n'est pas correcte, un vérificateur correct doit renvoyer $false$
— si l'application est correcte alors on distingue deux cas.
  — Lorsque $T \downarrow c$ et que $\mathsf{leaf}(c) \subseteq L$, c'est-à-dire lorsque le certificat est adapté, un vérificateur adapté renvoie $true$
  — Dans le cas contraire, un vérificateur pourra renvoyer n'importe quelle valeur booléenne

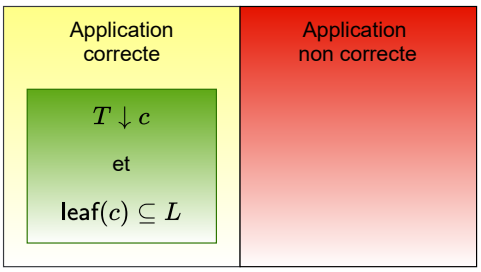

FIGURE 5.1 – Différents types d'applications de transformations

Nous définissons dans ce chapitre deux vérificateurs corrects et adaptés (à l'exception du vérificateur OCaml sur le certificat KConv) : l'un est implémenté en OCaml et s'appuie sur une approche calculatoire alors que l'autre repose sur un encodage superficiel dans Lambda-pi/Dedukti [4], assistant de preuve qui s'appuie sur un vérificateur de type pour le $\lambda\Pi$-calcul modulo réécriture.

5.2 Intégration des vérificateurs

Nous souhaitons définir des transformations pour lesquelles nous sommes assurés que chacune de leurs applications est correcte. Notre méthode, qui consiste en l'intégration des vérificateurs à notre travail, nous permet de nous en assurer. Étant donné une transformation certifiante ϕ et un vérificateur V, nous définissons une nouvelle transformation Φ définie comme la transformation qui :

— à partir d'une tâche initiale T, applique ϕ
— si ϕ échoue, alors Φ échoue également
— sinon, ϕ produit une liste de tâches résultantes L et un certificat c
  — si le vérificateur V sur T, L et c renvoie $true$, alors Φ renvoie L
  — sinon Φ échoue.

Si le vérificateur V est correct alors toutes les applications de Φ sont correctes. Puisque nous définissons, pour chaque transformation certifiante ϕ, une nouvelle transformation Φ de cette manière, l'appellation *transformation certifiante* pourra aussi désigner Φ, en précisant ou non quel vérificateur a été utilisé.

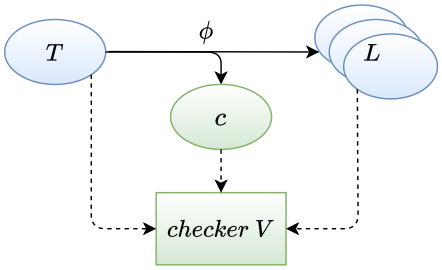

FIGURE 5.2 – Intégration d'un vérificateur

La correction des applications d'une transformation certifiante dépend donc uniquement de la correction du vérificateur utilisé.

5.3 Approche calculatoire de la vérification de certificats

Le premier vérificateur proposé est implémenté en OCaml. Dans cette première approche, j'ai implémenté une version modifiée et exécutable du jugement $T \downarrow c$ sous la forme d'une fonction OCaml ccheck qui, à partir d'une tâche T et d'un certificat c, calcule une liste de tâches telle que la validité de toutes ces tâches implique la validité de la tâche initiale.

Une transformation certifiante produit un certificat, ce qui permet de vérifier son résultat. Dans le cas du vérificateur en OCaml cette vérification se fait en appelant la fonction ccheck selon la figure 5.3, puis en vérifiant que la liste des tâches calculée est incluse dans à la liste des tâches résultantes.

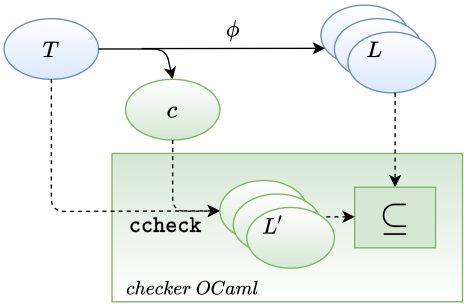

FIGURE 5.3 – Vérificateur OCaml

À partir d'une tâche T, la transformation ϕ considérée donne c et une liste de tâches L. Lorsque la fonction ccheck est appelée sur c et T, celle-ci inspecte le constructeur de tête du certificat, ce qui permet de déterminer une règle de certificat et les prémisses de cette règle. La fonction ccheck s'appelle ensuite récursivement sur ces tâches et les sous-certificats et concatène les listes obtenues. Lorsque les feuilles sont atteintes, la fonction renvoie la liste singleton constituée de la tâche courante et ignore la tâche contenue dans le certificat KHole.

Les tâches intermédiaires créées par la fonction ccheck sont les mêmes que les tâches intermédiaires dans une dérivation d'un jugement $T \downarrow c$. Pour cette raison, les tâches considérées restent des tâches bien typées. La fonction ccheck et la vérification de l'inclusion des listes constituent la base de confiance de cette approche.

Au niveau de l'implémentation, la vérification d'inclusion des listes peut se faire dans la fonction ccheck, lorsqu'une feuille est atteinte, en vérifiant que la tâche courante fait partie de la liste des tâches résultantes.

5.3.1 Réalisation et correction de la fonction ccheck

L'implémentation de ccheck procède naturellement par récurrence sur le certificat. Ce calcul vérifie les conditions d'application des règles, et lève des exceptions quand elles ne sont pas vérifiées, le vérificateur OCaml renvoyant alors $false$.

```
1  let rec ccheck c cta =
2    match c with
3      | KSlice (_, _, _, p, c1, c2) ->
4          let t, pos = find_formula p cta in
5          begin match t, pos with
6            | Binop (And, t1, t2), true | Binop (Or, t1, t2), false ->
7                let cta1 = add p (t1, pos) cta in
8                let cta2 = add p (t2, pos) cta in
9                union (ccheck c1 cta1) (ccheck c2 cta2)
10           | _ -> verif_failed "Not splittable"
11         end
12   ...
```

FIGURE 5.4 – Fragment de code KSlice du vérificateur OCaml

Détaillons l'implémentation de ccheck dans le cas où le certificat a la forme

$$\mathsf{KSlice}(b, t_1, t_2, P, c_1, c_2)$$

Le fragment de code correspondant est donné dans la figure 5.4 et a été légèrement simplifié afin d'en faciliter l'explication. Les lignes 1 et 2 donnent le début de la fonction ccheck, où il apparaît que celle-ci est définie récursivement sur le certificat pris en argument. Nous ne détaillons pas les autres constructeurs de certificat (ligne 12). La fonction ccheck vérifie que la formule `t` en position `pos` (ligne 4) qui est désignée par l'identifiant `p` est scindable (ligne 6), c'est-à-dire que c'est une conjonction si c'est un but ou une disjonction si c'est une hypothèse. Si c'est le cas, ccheck s'appelle récursivement sur les certificats `c1` et `c2` (ligne 9), le résultat final étant obtenu par concaténation. Par exemple, dans le cas d'un but, les appels récursifs se font sur la tâche initiale à laquelle on ajoute la déclaration qui associe `t1` au but `p` (ligne 7) et sur la tâche initiale à laquelle on ajoute la déclaration qui associe `t2` au but `p` (ligne 8). Si la formule n'est pas scindable, ccheck lève une exception (ligne 10).

On remarque que les premiers champs de ce certificat ne coïncident pas nécessairement avec la tâche. En effet, la fonction ccheck inspecte aussi la tâche et peut en déduire quelle est la règle à appliquer et quels sont ses paramètres. Cette flexibilité permet au vérificateur de ne pas avoir à vérifier que ces champs additionnels coïncident. De manière similaire, lorsqu'elle atteint une feuille, la fonction ccheck renvoie la liste singleton contenant la tâche courante et ne fait pas usage de la tâche contenue dans le constructeur KHole. Cela représente donc

soit une optimisation de la méthode soit un moyen de faciliter la définition du vérificateur. Nous verrons en effet, dans le paragraphe 5.5, que ces champs et les feuilles d'un certificat peuvent être utiles pour la définition d'autres vérificateurs. La proposition suivante formalise ces considérations.

Proposition 5.3. *Pour tout certificat c et toute tâche T, si* ccheck c T *renvoie L, alors il existe un certificat $\tilde{c}$ qui a pour feuilles L (et la même structure que c) et tel que $T \downarrow \tilde{c}$ est dérivable.*

Exemple 5.4. *Le certificat*

$$c := \mathsf{KSlice}(false, t_2, t_1, G, \mathsf{KHole}(\vdash G_{wrong} : \bot), \mathsf{KHole}(\vdash G_{alsowrong} : \top))$$

n'est pas adapté à l'application d'une transformation qui, à partir de la tâche $\vdash G : t_1 \wedge t_2$, produit les tâches $T_1 := \vdash G : t_1$ et $T_2 := \vdash G : t_2$. Le certificat c permet néanmoins de vérifier cette application grâce au vérificateur OCaml. En effet, puisque ccheck *ne tient pas compte des trois premiers arguments de ce certificat, il n'échoue pas et renvoie bien la liste $[T_1;\ T_2]$. On a $T \downarrow \tilde{c}$ en posant*

$$\tilde{c} := \mathsf{KSlice}(true, t_1, t_2, G, \mathsf{KHole}(T_1), \mathsf{KHole}(T_2))$$

Nous ne donnons pas de preuve formelle de la proposition précédente. Cependant, étant donné la sémantique d'OCaml, la preuve peut se faire par récurrence sur le certificat en analysant les différents cas. Le code de la fonction ccheck restant relativement simple, nous pouvons l'inspecter pour nous convaincre de sa correction. Le théorème suivant énonce la correction du vérificateur OCaml et sa preuve repose sur cette proposition : le théorème n'est donc pas non plus prouvé formellement.

Théorème 5.5 (Correction de ccheck)**.** *Pour tout certificat c et toute tâche T, la validité des tâches renvoyées par* ccheck c T *implique la validité de T.*

Démonstration. C'est un corollaire de la proposition 5.3 précédente et du théorème 3.17. □

5.3.2 Réécriture et certificat Conv en OCaml

Le vérificateur OCaml renvoie une exception dans le cas où le certificat contient le constructeur KConv. On aurait pu implémenter une fonction OCaml spécifique à la β-réduction et à l'ensemble de règles de réécriture du paragraphe 4.3, et qui indique si deux termes sont convertibles. Ces règles étant particulièrement simples, un *pattern-matching* d'OCaml suffirait. Afin de pouvoir étendre l'ensemble de règles on pourrait envisager d'utiliser un moteur de réécriture générique. Cela élargirait de façon significative le nombre de lignes de code OCaml auxquelles il faudrait faire confiance. Nous avons préféré déléguer la vérification de la convertibilité au vérificateur que nous définissons dans la suite de ce chapitre (voir 5.5.5.2).

5.3.3 Vers une preuve vérifiée par ordinateur

La propriété de correction de la fonction ccheck repose sur la proposition 5.3 dont la preuve pourrait être vérifiée par ordinateur. Pour ce faire, une méthode possible est d'encoder les tâches et les certificats dans un assistant de preuve, de leur donner une sémantique, de définir la fonction ccheck dans cet assistant de preuve et de prouver sa propriété de correction par récurrence. Une telle méthode peut être délicate et nécessite de choisir avec précaution les

définitions prises. En particulier, faire la preuve de correction dans un assistant de preuve nécessite de définir des termes avec lieurs, de prouver certaines de leurs propriétés et de formaliser les dictionnaires (pour les formules dans une tâche de preuve) ce qui peut poser des problèmes d'efficacité en fonction de la représentation choisie. Dans un assistant de preuve bien choisi, il est alors possible d'extraire le code de la fonction de vérification et d'obtenir ainsi une version exécutable. Plutôt que cette approche, j'ai choisi de garder ce vérificateur dans sa version efficace et de définir un deuxième vérificateur qui nous donne des garanties fortes de sa correction.

5.4 Encodage superficiel dans un système de type pur

La deuxième approche de la réalisation d'un vérificateur vise à se passer d'une preuve formalisée d'un théorème de correction comme le théorème 5.5. Mon implémentation d'un tel vérificateur utilise, en son cœur, le *framework* logique Lambdapi [4] avec un encodage d'un système de type pur [5, 35]. Par ailleurs, dans ce paragraphe, nous proposons un encodage superficiel de notre formalisme dans un système de type pur et nous montrons que, dans un tel système, nous avons besoin de toutes les quantifications qui caractérisent le Calcul des Constructions (CoC) [34]. Nous verrons dans le paragraphe 5.5 suivant comment nous définissons le plongement profond de CoC dans Lambdapi et comment nous pouvons le combiner avec l'encodage proposé ici afin de définir un vérificateur de certificats en Lambdapi.

5.4.1 Choix d'un système de type pur

Nous nous plaçons dans un système de type pur fonctionnel [35] et montrons dans ce paragraphe que nous avons besoin de toutes les quantifications offertes par le Calcul des Constructions afin de pouvoir définir la sémantique des tâches de preuve par un encodage superficiel.

Le système de type pur fonctionnel que nous choisissons possède exactement deux sortes :
— $\boldsymbol{Type}$ qui permet de typer les formules des tâches de preuve et les types des termes
— $\boldsymbol{Kind}$ qui permet de typer $\boldsymbol{Type}$

On note $\forall \boldsymbol{x : t_1},\ \boldsymbol{t_2}$ le produit dans ce système. La notation $\boldsymbol{t_1 \Rightarrow t_2}$ est utilisée lorsque ce produit n'est pas dépendant, c'est à dire quand $\boldsymbol{x}$ n'apparaît pas comme une variable libre dans $\boldsymbol{t_2}$. On considère que ce symbole est associatif à droite et que l'application est associative à gauche. L'abstraction est notée $\boldsymbol{\lambda x : t_1,\ t_2}$.

La notation $\boldsymbol{Type}^n$ désigne $\underbrace{\boldsymbol{Type} \Rightarrow \ldots \Rightarrow \boldsymbol{Type}}_{n \text{ fois}} \Rightarrow \boldsymbol{Type}$, c'est-à-dire le type des fonctions d'arité n sur $\boldsymbol{Type}$. Par exemple, si $\boldsymbol{map}$ dénote le type des dictionnaires, alors $\boldsymbol{map}$ est de type $\boldsymbol{Type}^2 := \boldsymbol{Type} \Rightarrow \boldsymbol{Type} \Rightarrow \boldsymbol{Type}$. La fonction de traduction des termes, notée tr, et la fonction de traduction des types, notée tr_τ, seront décrites dans le paragraphe 5.4.2.

Traduisons une tâche de preuve dans un tel système. Soit T la tâche de preuve suivante :

$$\iota_1 : i_1, \ldots, \iota_k : i_k \mid x_1 : \tau_1, \ldots, x_l : \tau_l \mid H_1 : t_1, \ldots, H_m : t_m \vdash G_1 : u_1, \ldots, G_n : u_n$$

Rappelons que l'on suppose qu'il existe un ordre total sur les identifiants noté $<_{ident}$. Supposons également que les identifiants de la tâche T soient triés selon cet ordre, c'est-à-dire que l'on a : $\iota_1 <_{ident} \ldots <_{ident} \iota_k$ et $x_1 <_{ident} \ldots <_{ident} x_l$ et $H_1 <_{ident} \ldots <_{ident} H_m$ et $G_1 <_{ident} \ldots <_{ident} G_n$.

Définition 5.6 (Formule exprimant la validité d'une tâche)**.** *La* formule exprimant la validité *de la tâche T dans un système de type pur est notée $tr(T)$ et est définie par :*

$$\begin{aligned} tr(T) := & \forall \boldsymbol{\iota_1} : \boldsymbol{Type}^{i_1}, \ldots, \forall \boldsymbol{\iota_k} : \boldsymbol{Type}^{i_k}, \\ & \forall \boldsymbol{x_1} : tr_\tau(\tau_1), \ldots, \forall \boldsymbol{x_l} : tr_\tau(\tau_l), \\ & tr(t_1) \Rightarrow \ldots \Rightarrow tr(t_m) \Rightarrow \\ & (tr(u_1) \Rightarrow tr(\bot)) \Rightarrow \ldots \Rightarrow (tr(u_n) \Rightarrow tr(\bot)) \Rightarrow \\ & tr(\bot) \end{aligned}$$

Remarquons que nous avons utilisé le fait que notre logique soit classique pour encoder un but u en $tr(u) \Rightarrow tr(\bot)$.

Un système de type pur vient avec la règle axiome suivante

$$\frac{}{\boldsymbol{\Gamma \vdash Type : Kind}}$$

ainsi que la règle suivante pour typer le produit

$$\frac{\boldsymbol{\Gamma \vdash t_1 : s_1} \qquad \boldsymbol{\Gamma, x : t_1 \vdash t_2 : s_2}}{\boldsymbol{\Gamma \vdash (\forall x : t_1,\ t_2) : s_2}}$$

où il reste à choisir les sortes $\boldsymbol{s_1}$ et $\boldsymbol{s_2}$ pour lesquelles cette règle est définie. Nous montrons que nous avons besoin de toutes les paires de sortes $(\boldsymbol{s_1}, \boldsymbol{s_2})$ et donc que, dans ce cadre, la logique considérée est celle du Calcul des Constructions.

En effet, nous avons besoin de :

— $(\boldsymbol{Type}, \boldsymbol{Type})$ pour former le type d'une fonction monomorphe telle que la fonction de type $\boldsymbol{int \Rightarrow int}$ qui incrémente un entier,

— $(\boldsymbol{Kind}, \boldsymbol{Type})$ pour former le type d'une fonction polymorphe telle que la fonction identité de type $\forall \boldsymbol{a : Type},\ \boldsymbol{a \Rightarrow a}$,

— $(\boldsymbol{Type}, \boldsymbol{Kind})$ pour former le type d'un prédicat monomorphe tel que le prédicat de type $\boldsymbol{int \Rightarrow Type}$ qui indique si un entier est pair,

— $(\boldsymbol{Kind}, \boldsymbol{Kind})$ pour former le type d'un prédicat polymorphe tel que l'égalité qui a pour type $\forall \boldsymbol{a : Type},\ \boldsymbol{a \Rightarrow a \Rightarrow Type}$.

La propriété de correction de la traduction d'une tâche de preuve en CoC s'exprime (selon le principe de la correspondance de Curry-Howard) par le fait que la validité d'une tâche T est équivalente à l'existence d'un terme de type $tr(T)$.

Nous avons vu que pour pouvoir définir la formule exprimant la validité d'une tâche de preuve, nous avons besoin de quantifier sur des variables polymorphes. Nous ne pouvons pas exprimer une telle formule comme une formule de STT$\forall$ [81], et nous n'utiliserons donc pas l'encodage de cette logique en Lambdapi [80].

5.4.2 Encodage dans le Calcul des Constructions

Nous réutilisons les notations des systèmes de type purs, et précisons la fonction de traduction des termes et la fonction de traduction des types dans le Calcul des Constructions. La figure 5.5 décrit l'encodage des types où l'on introduit, pour chaque symbole de type ι d'arité n, un symbole $\boldsymbol{\iota}$ de type $\boldsymbol{Type}^n$ dans CoC. La figure 5.6 décrit l'encodage des termes. Notons que la hiérarchie de types implicite dont nous parlions au paragraphe 2.7 se manifeste ici par la définition de deux fonctions de traduction des types : tr_τ et tr'_τ.

$$
\begin{array}{rcl}
tr_\tau(\tau) & := & \forall \boldsymbol{\alpha_1 : Type},\ \ldots\ \forall \boldsymbol{\alpha_n : Type},\ tr'_\tau(\tau) \\
 & & \text{où } V_\alpha(\tau) = \{\alpha_1,\ \ldots,\ \alpha_n\} \\
 & & \text{avec } \alpha_1 <_{ident} \ldots <_{ident} \alpha_n \\
tr'_\tau(\alpha) & := & \boldsymbol{\alpha} \\
tr'_\tau(prop) & := & \boldsymbol{Type} \\
tr'_\tau(\tau_1 \rightsquigarrow \tau_2) & := & tr'_\tau(\tau_1) \Rightarrow tr'_\tau(\tau_2) \\
tr'_\tau(\iota(\tau_1, \ldots, \tau_n)) & := & \boldsymbol{\iota}\ tr'_\tau(\tau_1)\ \ldots\ tr'_\tau(\tau_n)
\end{array}
$$

FIGURE 5.5 – Encodage des types

$$
\begin{array}{rcl}
tr(\Pi\alpha.\ t) & := & \forall \boldsymbol{\alpha : Type},\ tr(t) \\
 & & \\
tr(\bot) & := & \forall \boldsymbol{C : Type},\ \boldsymbol{C} \\
tr(\top) & := & tr(\bot) \Rightarrow tr(\bot) \\
tr(\neg\ t) & := & tr(t) \Rightarrow tr(\bot) \\
tr(t_1 \wedge t_2) & := & \forall \boldsymbol{C : Type},\ (tr(t_1) \Rightarrow tr(t_2) \Rightarrow \boldsymbol{C}) \Rightarrow \boldsymbol{C} \\
tr(t_1 \vee t_2) & := & \forall \boldsymbol{C : Type},\ (tr(t_1) \Rightarrow \boldsymbol{C}) \Rightarrow (tr(t_2) \Rightarrow \boldsymbol{C}) \Rightarrow \boldsymbol{C} \\
tr(t_1 \Rightarrow t_2) & := & tr(t_1) \Rightarrow tr(t_2) \\
tr(t_1 \Leftrightarrow t_2) & := & tr((t_1 \Rightarrow t_2) \wedge (t_2 \Rightarrow t_1)) \\
tr(t_1\ t_2) & := & tr(t_1)\ tr(t_2) \\
tr(\forall x : \tau.\ t) & := & \forall \boldsymbol{x} : tr'_\tau(\tau),\ tr(t) \\
tr(\exists x : \tau.\ t) & := & \forall \boldsymbol{C : Type},\ (\forall \boldsymbol{x} : tr'_\tau(\tau),\ tr(t) \Rightarrow \boldsymbol{C}) \Rightarrow \boldsymbol{C} \\
tr(\lambda x : \tau.\ t) & := & \boldsymbol{\lambda x} : tr'_\tau(\tau),\ tr(t)
\end{array}
$$

FIGURE 5.6 – Encodage imprédicatif des termes

Nous utilisons un encodage imprédicatif : la définition des opérateurs propositionnels s'appuie sur une quantification de formules. Remarquons que l'encodage des symboles, s'appuyant sur une information de typage, n'est pas décrit dans ces figures. Si un symbole x est monomorphe, alors on a :

$$tr(x) := \boldsymbol{x}$$

Remarquons que l'introduction de nouvelles variables liées donne toujours des termes monomorphes. Cependant, dans le cas général, un symbole peut représenter un terme polymorphe, et son encodage en rend compte en l'appliquant aux types qui l'instancient. Plus précisément, soit un symbole $(x)_{\tau'}$ (x est accompagné du type τ'). Posons $\tau := \Sigma(x)$ où $V_\alpha(\tau) = \{\alpha_1,\ \ldots,\ \alpha_n\}$ avec $\alpha_1 <_{ident} \ldots <_{ident} \alpha_n$. On obtient alors une substitution σ telle que $\tau' = \tau\sigma$. Pour chaque variable de type α_i posons $\tau_i = \alpha_i\sigma$. La traduction d'une telle occurrence de x est alors donnée par :

$$\boldsymbol{x}\ tr'_\tau(\tau_1)\ \ldots\ tr'_\tau(\tau_n)$$

Notons que cette définition de la traduction d'un symbole potentiellement polymorphe s'applique aussi bien aux variables qu'aux symboles interprétés.

Exemple 5.7. *Traduisons la formule de l'hypothèse H_2 de l'exemple 2.27 :*

$$t := \Pi\beta.\ \forall x : \beta.\ \forall s : set(\beta).\ mem\ x\ (add\ x\ s)$$

Le symbole x étant monomorphe, chaque occurrence de x est traduite par $\boldsymbol{x}$, et, pour les mêmes raisons, l'occurrence de s est traduite par $\boldsymbol{s}$. Dans la formule t, l'occurrence de mem est nécessairement accompagnée du type $\beta \rightsquigarrow set(\beta) \rightsquigarrow prop$, type qui est obtenu en substituant β à α dans $\Sigma(mem)$. Cette occurrence est traduite par $\boldsymbol{mem\ \beta}$. La traduction de t est alors donnée par :

$$tr(t) = \forall \boldsymbol{\beta : Type},\ \forall \boldsymbol{x : \beta},\ \forall \boldsymbol{s : set\ \beta},\ \boldsymbol{mem\ \beta\ x\ (add\ \beta\ x\ s)}$$

Exemple 5.8. *Reprenons l'exemple précédent et traduisons à la place la formule :*

$$u := \Pi\beta.\ \forall x : set(\beta).\ mem\ x\ (add\ x\ empty)$$

On traduit toujours x par $\boldsymbol{x}$. L'occurrence de mem est nécessairement accompagnée de $set(\beta) \rightsquigarrow set(set(\beta)) \rightsquigarrow prop$, type obtenu en substituant $set(\beta)$ à α dans le type de mem. Cette occurrence est donc traduite par $\boldsymbol{mem\ (set\ \beta)}$. Pour les mêmes raisons, l'occurrence de add est traduite par $\boldsymbol{add\ (set\ \beta)}$ et l'occurrence de empty est traduite par $\boldsymbol{empty\ (set\ \beta)}$. La traduction de u est alors donnée par :

$$tr(u) = \forall \boldsymbol{\beta : Type},\ \forall \boldsymbol{x : set\ \beta},\ \boldsymbol{mem\ (set\ \beta)\ x\ (add\ (set\ \beta)\ x\ (empty\ (set\ \beta)))}$$

Exemple 5.9. *Soit :*

$$\begin{aligned}
I &:= \iota : 0 \\
\Sigma &:= t_1 : \iota,\ t_2 : \iota, \\
&\quad\ eq : \alpha \rightsquigarrow \alpha \rightsquigarrow prop, \\
&\quad\ fst : \alpha \rightsquigarrow \alpha \rightsquigarrow \alpha \\
\Gamma &:= eqTrans : \forall x_1 : \alpha.\ \forall x_2 : \alpha.\ \forall x_3 : \alpha.\ eq\ x_1\ x_2 \Rightarrow eq\ x_2\ x_3 \Rightarrow eq\ x_1\ x_3, \\
&\quad\ fstDef : \forall x_1 : \alpha.\ \forall x_2 : \alpha.\ eq\ (fst\ x_1\ x_2)\ x_1 \\
\Delta &:= G : eq\ (fst\ (fst\ t_1\ t_2)\ t_2)\ t_1
\end{aligned}$$

alors la validité de la tâche $T := (I \mid \Sigma \mid \Gamma \vdash \Delta)$ s'exprime par la formule du Calcul des Constructions suivante :

$$\begin{aligned}
&\boldsymbol{\forall \iota : Type,} \\
&\boldsymbol{\forall t_1 : \iota,\ \forall t_2 : \iota,} \\
&\boldsymbol{\forall eq : (\forall \alpha : Type,\ \alpha \Rightarrow \alpha \Rightarrow Type),} \\
&\boldsymbol{\forall fst : (\forall \alpha : Type,\ \alpha \Rightarrow \alpha \Rightarrow \alpha),} \\
&\boldsymbol{(\forall \alpha : Type,\ \forall x_1 : \alpha,\ \forall x_2 : \alpha,\ \forall x_3 : \alpha,} \\
&\boldsymbol{\ eq\ \alpha\ x_1\ x_2 \Rightarrow eq\ \alpha\ x_2\ x_3 \Rightarrow eq\ \alpha\ x_1\ x_3) \Rightarrow} \\
&\boldsymbol{(\forall \alpha : Type,\ \forall x_1 : \alpha,\ \forall x_2 : \alpha,\ eq\ \alpha\ (fst\ \alpha\ x_1\ x_2)\ x_1) \Rightarrow} \\
&\boldsymbol{(eq\ \iota\ (fst\ \iota\ (fst\ \iota\ t_1\ t2)\ t2)\ t_1 \Rightarrow (\forall C : Type,\ C)) \Rightarrow \forall C : Type,\ C}
\end{aligned}$$

On peut montrer que T est valide en donnant un terme qui a ce type :

$$\begin{aligned}
&\boldsymbol{\lambda \iota\ t_1\ t_2\ eq\ fst\ eqTrans\ fstDef\ G,} \\
&\ \boldsymbol{G\ (eqTrans\ \iota\ (fst\ \iota\ (fst\ \iota\ t_1\ t_2)\ t_2)\ (fst\ \iota\ t_1\ t_2)\ t_1} \\
&\qquad \boldsymbol{(fstDef\ \iota\ (fst\ \iota\ t_1\ t_2)\ t_2)} \\
&\qquad \boldsymbol{(fstDef\ \iota\ t_1\ t_2))}
\end{aligned}$$

5.4.3 Incomplétude de la traduction vers CoC

La traduction de notre formalisme en CoC est partielle : les termes polymorphes de notre langage peuvent être instanciés par le type *prop* des formules alors que la quantification de type de CoC porte sur les termes de type $\boldsymbol{Type}$ et $\boldsymbol{Type}$ n'est pas de type $\boldsymbol{Type}$.

Exemple 5.10. *La tâche* $f : \alpha \rightsquigarrow prop \mid G : f\ \bot \vdash$ *serait traduite par*

$$\forall \boldsymbol{f} : (\forall \boldsymbol{\alpha} : \boldsymbol{Type},\ \boldsymbol{\alpha} \Rightarrow \boldsymbol{Type}),\ (\boldsymbol{f}\ \boldsymbol{Type}\ (\forall \boldsymbol{C} : \boldsymbol{Type},\ \boldsymbol{C})) \Rightarrow (\forall \boldsymbol{C} : \boldsymbol{Type},\ \boldsymbol{C})$$

formule qui est mal typée en CoC.

Cette incomplétude ne limite pas, dans notre application présentée au chapitre 7, les transformations de Why3 que nous pouvons certifier : en Why3 on ne peut pas quantifier sur des propositions. En effet, Why3 distingue le type des booléens (valeur de première classe) du "type" des formules logiques et utilise les conversions explicites entre les deux lorsque cela est nécessaire.

5.5 Vérification par encodage dans le $\lambda\Pi$-calcul

Nous détaillons, dans ce paragraphe, l'implémentation d'un vérificateur s'appuyant sur le vérificateur de preuve universel Lambdapi/Dedukti [4]. L'intérêt est de mettre à profit son mécanisme de règles de réécriture pour encoder simplement nos certificats. À chaque fois qu'une transformation est appliquée, une preuve Lambdapi de correction est produite et peut être vérifiée par l'implémentation d'un vérificateur de type pour ce langage.

Plus précisément, nous combinons l'encodage superficiel des tâches en CoC du paragraphe précédent avec un encodage profond de CoC en Lambdapi. La composition de ces encodages nous fournit un encodage des tâches de preuve dans Lambdapi : la validité d'une tâche T est encodée en une formule Lambdapi $\widehat{T}$. Si, sur une tâche T, on applique une transformation certifiante produisant une liste de tâches $L := [T_1;\ \ldots; T_n]$ et un certificat c, alors on construit à partir de c un terme Lambdapi qui est candidat pour une preuve de $\widehat{T_1} \to \ldots \to \widehat{T_n} \to \widehat{T}$, où $\to$ est la flèche de Lambdapi. Ce terme candidat est naturellement destiné à être transmis à Lambdapi pour vérification.

Dans cette deuxième approche, la base de confiance est constituée de la traduction des tâches en CoC, de l'encodage de CoC en Lambdapi, et du vérificateur de type de Lambdapi lui-même. Ainsi, comme expliqué dans le paragraphe 5.5.4.1, nous vérifions une fois pour toutes par ordinateur la correction de chacune des règles des certificats (théorème 3.17) selon l'approche autarcique. La traduction des tâches est correcte lorsque, pour toute tâche T, cette tâche est valide si et seulement si le type $\widehat{T}$ est habité. Pour limiter la base de confiance, nous faisons attention à ne pas ajouter d'axiomes, et choisissons à la place de partir des définitions et de prouver les propriétés nécessaires. En revanche, la fonction de traduction d'un certificat en un terme de preuve Lambdapi n'est pas dans la base de confiance. En effet, la façon dont la preuve a été obtenue n'importe pas, tant que cette preuve est correcte.

5.5.1 Présentation de Lambdapi

Lambdapi/Dedukti [4] est un vérificateur de preuve, qui est universel dans le sens où il est possible d'encoder de nombreuses logiques et de vérifier les preuves faites dans ces logiques. La possibilité de définir de nouvelles règles de réécriture offre une grande expressivité qui permet d'encoder tous les systèmes de type purs, et en particulier le Calcul des Constructions [21, 35].

Décrivons la syntaxe de Lambdapi et les fonctionnalités que nous utilisons dans la suite : nous expliquons comment définir de nouveaux termes, axiomes et règles de réécriture, et comment prouver des propriétés dans ce contexte.

Les termes sont construits à partir de variables `x`, d'applications `t1 t2` d'un terme `t1` à un terme `t2`, d'abstractions λ`x : A, t` d'une variable `x` de type `A` par rapport à un terme `t`, de produits dépendants Π (`x : A`), `B` d'une variable `x` de type `A` par rapport à `B`. On note `A` → `B` le produit Π (`x : A`), `B` lorsque `x` n'apparaît pas frais dans `B`. Cette syntaxe vient avec un système de type [22], et ici le symbole `TYPE` désigne l'unique sorte et permet de typer les types.

La déclaration d'un nouveau symbole `f` de type `A` se fait au moyen de l'instruction

```
symbol f : A;
```

ce qui revient à ajouter au contexte l'axiome énonçant la formule `A`. Remarquons que chaque instruction est terminée par un point-virgule.

Notons que l'encodage donné à titre d'exemple dans ce paragraphe n'est pas celui utilisé dans notre développement Lambdapi, celui-ci étant décrit à partir du paragraphe 5.5.2.

Exemple 5.11. *Le fragment de code suivant déclare le symbole de type des formules propositionnelles* `Formula` *ainsi que les constantes* `Top` *et* `Bot`*.*

```
symbol Formula : TYPE;
symbol ε : Formula → TYPE;
symbol Bot : Formula;
symbol Top : Formula;
symbol TopProof : ε Top;
```

La fonction ε permet de plonger les propositions dans Lambdapi, on peut alors donner l'axiome `TopProof` *énonçant que la proposition vraie a une preuve.*

L'instruction

```
symbol f (x1 : A1) ... (xn : An) : A;
```

nous offre une autre façon d'énoncer Π (`x1 : A1`) ... (`xn : An`), `A`. Notons que nous pouvons regrouper les variables de même type en écrivant plutôt (`x1 x2 : A1`) (`x3 : A3`) ... (`xn : An`) et même, lorsque c'est possible, laisser Lambdapi inférer leur type en écrivant seulement `x1 ... xn`.

Exemple 5.12. *On étend l'exemple précédent avec un symbole pour la conjonction accompagné d'axiomes d'introduction et d'élimination.*

```
symbol Conj : Formula → Formula → Formula;
symbol ConjIntro (p q : Formula) : ε p → ε q → ε (Conj p q);
symbol ConjElim1 (p q : Formula) : ε (Conj p q) → ε p;
symbol ConjElim2 (p q : Formula) : ε (Conj p q) → ε q;
```

La définition d'une nouvelle constante `f` passe par l'instruction :

```
symbol f : A ≔ t;
```

Selon la correspondance de Curry-Howard, on peut également voir cette définition comme une preuve de `A` car Lambdapi vérifie que t a le type `A`. L'instruction

`symbol f` $(x_1 : A_1) \dots (x_n : A_n) : A \coloneqq t$;

définit une fonction de type $\Pi\ (x_1 : A_1)\ \dots\ (x_n : A_n),\ A$. Cette syntaxe peut ainsi être utilisée pour énoncer (et prouver) des théorèmes quantifiés universellement.

Exemple 5.13. *En reprenant les définitions précédentes, on peut prouver que la conjonction est symétrique :*

`symbol ConjSym p q :` ε `(Conj p q)` $\to \varepsilon$ `(Conj q p)`
$\coloneqq \lambda$`H, ConjIntro q p (ConjElim2 p q H) (ConjElim1 p q H);`

L'instruction

`rule t1` $\hookrightarrow$ `t2;`

ajoute la règle de réécriture de `t1` en `t2`. Les variables apparaissant dans `t1` ou `t2` qui sont préfixées par $ sont implicitement quantifiées universellement au niveau de cette règle.

Exemple 5.14. *Continuons l'exemple en définissant un nouveau type : celui des couleurs des feux tricolores !*

`symbol Color : TYPE;`
`symbol Green : Color;`
`symbol Orange : Color;`
`symbol Red : Color;`

`symbol Activate : Color` $\to$ `Color;`
`symbol IsGreen : Color` $\to$ `Formula;`

On définit une fonction `Activate` *pour faire passer le feu à l'état suivant, et une fonction* `IsGreen`*, utile pour savoir si l'on peut passer. Puisqu'un feu passe à l'orange après le vert, la règle*

`rule Activate Green` $\hookrightarrow$ `Orange;`

permet de réécrire `Activate Green` *en* `Orange`*. On complète cette règle avec les suivantes :*

`rule Activate Orange` $\hookrightarrow$ `Red`
`with Activate Red` $\hookrightarrow$ `Green;`

`rule IsGreen Green` $\hookrightarrow$ `Top`
`with IsGreen Orange` $\hookrightarrow$ `Bot`
`with IsGreen Red` $\hookrightarrow$ `Bot;`

où le mot-clé `with` *permet de regrouper plusieurs règles sous une seule instruction. Remarquons que ces règles de réécriture ne définissent pas nécessairement des fonctions totales : le type* `Color` *peut être étendu par la suite.*

La vérification de type de Lambdapi se fait modulo les règles de réécriture, ce qui nous permet notamment de faire des preuves qui reposent sur le calcul. Nous ferons attention à ce que notre système de réécriture soit confluent et fortement normalisant, nous garantissant ainsi que le vérificateur de Lambdapi répond en temps fini à un problème de typage.

Exemple 5.15. *La définition suivante est vérifiée par Lambdapi et repose sur le fait que* `IsGreen (Activate (Activate (Orange)))` *se réécrit en* `Top` *:*

`symbol GreenAfterOrange : ε (IsGreen (Activate (Activate (Orange))))`

`≔ TopProof;`

Pour simplifier les notations, on note $\mathtt{t}_1 \equiv \mathtt{t}_2$ lorsque $\mathtt{t}_1$ et $\mathtt{t}_2$ sont convertibles, c'est-à-dire lorsqu'ils sont équivalents selon la clôture réflexive, symétrique et transitive de l'union des règles de réécriture de Lambdapi. Notons aussi que toute définition

$\mathtt{symbol\ f\ (x_1 : A_1) \dots (x_n : A_n) : A ≔ t;}$

ajoute une règle de réécriture quantifiée implicitement sur $\mathtt{x}_1, \dots, \mathtt{x}_n$ et réécrivant $\mathtt{f\ x_1\ \dots\ x_n}$ en `t`.

5.5.2 Encodage du Calcul des Constructions dans Lambdapi

En tant que *framework* logique, Lambdapi permet d'encoder de nombreuses logiques et notamment d'encoder les systèmes de type purs [35]. Nous reprenons l'encodage du Calcul des Constructions [21], que nous étendons pour pouvoir encoder les types interprétés de manière adéquate. Nous commençons par déclarer de nouveaux symboles Lambdapi que nous appelons des symboles "profonds". Le symbole ***Type*** de CoC a une déclaration d'un nouveau symbole associé `DType` dans Lambdapi (`D` pour *deep*, profond). Afin de traduire le produit de CoC en Lambdapi, nous déclarons également les symboles profonds $\forall_{\mathtt{tt}}, \forall_{\mathtt{kt}}, \forall_{\mathtt{tk}}$ et $\forall_{\mathtt{kk}}$. Les symboles profonds sont détaillés dans la suite (en particulier pourquoi il y a quatre symboles pour le produit) et nous verrons aussi (figure 5.8) que les autres constructions (telles que l'abstraction ou l'application) n'ont pas de symbole profond associé et sont directement traduites par la construction correspondante en Lambdapi. Nous définissons également des fonctions permettant de plonger cet encodage dans Lambdapi, celles-ci étant systématiquement appliquées aux termes encodés. Ces fonctions sont capitales dans la définition de l'encodage car elles donnent une sémantique aux symboles profonds à travers des règles de réécriture que nous détaillons dans la suite. Les symboles profonds sont alors réécrits dans la logique sous-jacente de Lambdapi, ce qui nous permet d'obtenir un encodage superficiel de CoC. On peut par exemple traiter l'encodage d'un produit CoC comme un produit Lambdapi par application de ces règles de réécriture.

Décrivons cet encodage, les sortes étant déclarées par les symboles `Type` et `Kind` :

`symbol Type : TYPE;`

`symbol Kind : TYPE;`

Ces sortes sont utilisées pour donner un type aux symboles profonds, ces derniers étant inscrits dans la logique sous-jacente de Lambdapi à travers deux fonctions, une pour chaque sorte :

$\mathtt{symbol}\ \varepsilon_{\mathtt{t}} : \mathtt{Type} \to \mathtt{TYPE};$

$\mathtt{symbol}\ \varepsilon_{\mathtt{k}} : \mathtt{Kind} \to \mathtt{TYPE};$

Il est maintenant possible de déclarer les symboles profonds nécessaires. Pour chacun d'entre eux nous donnons également sa sémantique en définissant une règle de réécriture lorsque celui-ci est plongé dans Lambdapi. Le type des formules est donné par le symbole `DType` défini par :

```
symbol DType : Kind;
rule εk DType ↪ Type;
```

La quantification universelle de CoC est alors traduite en un symbole profond à choisir parmi quatre, à savoir $\forall_{\mathtt{tt}}, \forall_{\mathtt{kt}}, \forall_{\mathtt{tk}}$ et $\forall_{\mathtt{kk}}$. En effet, il y a deux possibilités pour le type sur lequel on quantifie, et deux possibilités pour le type de retour, à chaque fois entre `Type` et `Kind`. La figure 5.7 donne les déclarations de ces symboles et les règles de réécriture correspondantes.

```
symbol ∀tt : Π (x : Type), (εt x → Type) → Type;
symbol ∀kt : Π (x : Kind), (εk x → Type) → Type;
symbol ∀tk : Π (x : Type), (εt x → Kind) → Kind;
symbol ∀kk : Π (x : Kind), (εk x → Kind) → Kind;

rule εt (∀tt $x $y) ↪ Π (w : εt $x), εt ($y w)
with εt (∀kt $x $y) ↪ Π (w : εk $x), εt ($y w)
with εk (∀tk $x $y) ↪ Π (w : εt $x), εk ($y w)
with εk (∀kk $x $y) ↪ Π (w : εk $x), εk ($y w);
```

FIGURE 5.7 – Encodage de la quantification de CoC en Lambdapi

Exemple 5.16. *La sémantique d'une tâche dans Lambdapi est donnée en quantifiant sur sa signature. Dans ce cas précis, il s'agit de quantifier sur des termes de type* `Kind`*, ce qui nous rend une formule de type* `Type`*. Le symbole correspondant à cette quantification est donc* $\forall_{\mathtt{kt}}$*.*

On utilise les notations suivantes pour les flèches non dépendantes

```
t1 ⇒tt t2 := ∀tt t1 (λ_, t2)
t1 ⇒kt t2 := ∀kt t1 (λ_, t2)
t1 ⇒tk t2 := ∀tk t1 (λ_, t2)
t1 ⇒kk t2 := ∀kk t1 (λ_, t2)
```

où les symboles _ désignent des variables liées non utilisées.

Étant donné une formule $\boldsymbol{t}$ de CoC, on peut alors donner son encodage $\|\boldsymbol{t}\|$ dans Lambdapi. Cet encodage [35] est défini en fonction d'informations de typage supplémentaires. Lorsque la sorte $\boldsymbol{s}$ est $\boldsymbol{Type}$, alors ε_s désigne ε_t et lorsque cette sorte est $\boldsymbol{Kind}$, alors ε_s désigne ε_k. L'encodage des formules s'appuie sur l'encodage des termes, noté $|\boldsymbol{t}|$, expression

qui est elle-même définie récursivement. La figure 5.8 résume ces encodages. La définition de $|\boldsymbol{\lambda x : t_1,\ t_2}|$ est donnée en fonction de $\boldsymbol{s}$, la sorte qui type $\boldsymbol{t_1}$ dans CoC. Similairement, la définition de $|\boldsymbol{\Pi x : t_1,\ t_2}|$, dépend de la sorte $\boldsymbol{s1}$ qui type $\boldsymbol{t_1}$ et de la sorte $\boldsymbol{s_2}$ qui type $\boldsymbol{t_2}$.

$$
\begin{array}{lcl}
\|\boldsymbol{t}\| & := & \varepsilon_{\mathtt{s}}\ |\boldsymbol{t}| \\
 & & \\
|\boldsymbol{Type}| & := & \mathtt{DType} \\
|\boldsymbol{x}| & := & \mathtt{x} \\
|\boldsymbol{t_1\ t_2}| & := & |\boldsymbol{t_1}|\ |\boldsymbol{t_2}| \\
|\boldsymbol{\lambda x : t_1,\ t_2}| & := & \lambda \mathtt{x} : \varepsilon_{\mathtt{s}}\ |\boldsymbol{t_1}|,\ |\boldsymbol{t_2}| \\
|\boldsymbol{\forall x : t_1,\ t_2}| & := & \forall_{\mathtt{s_1 s_2}}\ |\boldsymbol{t_1}|\ (\lambda \mathtt{x} : \varepsilon_{\mathtt{s_1}}\ |\boldsymbol{t_1}|,\ |\boldsymbol{t_2}|)
\end{array}
$$

FIGURE 5.8 – Encodage de CoC en Lambdapi

5.5.3 Encodage des tâches de preuve en Lambdapi

L'encodage de notre formalisme en Lambdapi combine l'encodage dans le Calcul des Constructions et l'encodage de celui-ci dans Lambdapi. L'encodage d'une tâche T en une formule Lambdapi $\widehat{T}$ est défini comme la validité de T exprimée dans CoC, elle-même encodée en Lambdapi :

$$\widehat{T} := \|tr(T)\|$$

Comme décrit par la figure 5.8, le quantificateur universel de CoC est traduit en un symbole parmi quatre nouveaux symboles déclarés en Lambdapi. Il y a donc un encodage profond de certaines constructions de CoC mais, puisque l'on applique une fonction $\varepsilon_{\mathtt{s}}$ à cet encodage, celui-ci se réécrit dans la logique sous-jacente à Lambdapi, nous donnant ainsi un encodage superficiel. En effet, si T est la tâche de la forme suivante

$$\iota_1 : i_1, \ldots, \iota_k : i_k \mid x_1 : \tau_1, \ldots, x_l : \tau_l \mid H_1 : t_1, \ldots, H_m : t_m \vdash G_1 : u_1, \ldots, G_n : u_n$$

alors, en désignant $\underbrace{\mathtt{Type} \to \ldots \to \mathtt{Type}}_{i_z \text{ fois}} \to \mathtt{Type}$ par $\mathtt{Type}^{i_z}$, $\widehat{T}$ se réécrit en

$$
\begin{array}{l}
\Pi\ (\iota_1 : \mathtt{Type}^{i_1})\ \ldots\ (\iota_{\mathtt{k}} : \mathtt{Type}^{i_k}), \\
\quad \Pi\ (\mathtt{x_1} : \|tr_{\tau}(\tau_1)\|)\ \ldots\ (\mathtt{x_1} : \|tr_{\tau}(\tau_l)\|), \\
\qquad \|tr(t_1)\| \to \ldots \to \|tr(t_m)\| \to \\
\qquad\quad (\|tr(u_1)\| \to \|tr(\bot)\|) \to \ldots \to (\|tr(u_n)\| \to \|tr(\bot)\|) \to \\
\qquad\qquad \|tr(\bot)\|
\end{array}
$$

où la quantification se fait par le symbole Π de Lambdapi. Un des avantages de cet encodage superficiel est de permettre de se passer d'un traitement explicite du contexte qui s'appuierait par exemple sur des lemmes d'inversion et d'affaiblissement, méthode qui ne passe pas à l'échelle.

Exemple 5.17. *Montrons que la tâche $t_1 : prop,\ t_2 : prop \mid H_1 : t_1,\ H_2 : t_2 \vdash G : t_1$ est valide en utilisant son encodage en Lambdapi :*

$\mathtt{symbol\ ToVerify} : \|\boldsymbol{\forall t_1 : Type,\ \forall t_2 : Type,\ t_1 \Rightarrow t_2 \Rightarrow (}tr(t_1) \Rightarrow tr(\bot)\boldsymbol{) \Rightarrow} tr(\bot)\|$
$:= \lambda \mathtt{t_1\ t_2\ H_1\ H_2\ G,\ G\ H_1};$

Cette définition est bien vérifiée par Lambdapi car la formule à vérifier se réécrit en

$$\Pi\ (\mathtt{t}_1 : \mathtt{Type})\ (\mathtt{t}_2 : \mathtt{Type}),\ \varepsilon_{\mathtt{t}}\ \mathtt{t}_1 \to \varepsilon_{\mathtt{t}}\ \mathtt{t}_2 \to (\varepsilon_{\mathtt{t}}\ \mathtt{t}_1 \to \widehat{\bot}) \to \widehat{\bot}$$

où l'on a noté $\widehat{\bot} := \Pi\ \mathtt{C} : \mathtt{Type},\ \varepsilon_{\mathtt{t}}\ \mathtt{C}$.

On écrit $\widetilde{t}$ pour désigner $|tr(t)|$ lorsque t est un terme, et on pose :

$$\begin{array}{lcl}
\widetilde{\neg} & := & \Pi\ (\mathtt{t} : \mathtt{Type}),\ \mathtt{t} \Rightarrow_{\mathtt{tt}} \widetilde{\bot} \\
\widetilde{\wedge} & := & \Pi\ (\mathtt{t}_1\ \mathtt{t}_2 : \mathtt{Type}),\ \forall_{\mathtt{kt}}\ \mathtt{DType}\ (\lambda \mathtt{C},\ (\mathtt{t}_1 \Rightarrow_{\mathtt{tt}} \mathtt{t}_2 \Rightarrow_{\mathtt{tt}} \mathtt{C}) \Rightarrow \mathtt{C}) \\
\widetilde{\vee} & := & \Pi\ (\mathtt{t}_1\ \mathtt{t}_2 : \mathtt{Type}),\ \forall_{\mathtt{kt}}\ \mathtt{DType}\ (\lambda \mathtt{C},\ (\mathtt{t}_1 \Rightarrow_{\mathtt{tt}} \mathtt{C}) \Rightarrow_{\mathtt{tt}} (\mathtt{t}_2 \Rightarrow_{\mathtt{tt}} \mathtt{C}) \Rightarrow_{\mathtt{tt}} \mathtt{C}) \\
\widetilde{\Leftrightarrow} & := & \Pi\ (\mathtt{t}_1\ \mathtt{t}_2 : \mathtt{Type}),\ (\mathtt{t}_1 \Rightarrow_{tt} \mathtt{t}_2)\ \widetilde{\wedge}\ (\mathtt{t}_2 \Rightarrow_{tt} \mathtt{t}_1)
\end{array}$$

Nous utilisons la notation infixe pour $\widetilde{\wedge}$ et pour $\widetilde{\vee}$. On montre alors que l'on a :

$$\begin{array}{lcl}
\widetilde{\bot} & = & \forall_{\mathtt{tt}}\ \mathtt{DType}\ (\lambda \mathtt{C},\ \mathtt{C}) \\
\widetilde{\top} & = & \widetilde{\bot} \Rightarrow_{\mathtt{tt}} \widetilde{\bot} \\
\widetilde{\neg\, t} & \equiv & \widetilde{\neg}\ \widetilde{t} \\
\widetilde{t_1 \wedge t_2} & \equiv & \widetilde{t_1}\ \widetilde{\wedge}\ \widetilde{t_2} \\
\widetilde{t_1 \vee t_2} & \equiv & \widetilde{t_1}\ \widetilde{\vee}\ \widetilde{t_2} \\
\widetilde{t_1 \Leftrightarrow t_2} & \equiv & \widetilde{t_1}\ \widetilde{\Leftrightarrow}\ \widetilde{t_2}
\end{array}$$

Lorsque t est un terme, on écrit aussi $\widehat{t}$ pour désigner $\|tr(t)\|$ et on utilise également la notation infixe pour $\widehat{\wedge}$ et pour $\widehat{\vee}$. On a alors :

$$\begin{array}{lcl}
\widehat{\bot} & \equiv & \Pi\ \mathtt{C} : \mathtt{Type},\ \varepsilon_{\mathtt{t}}\ \mathtt{C} \\
\widehat{\top} & \equiv & \widehat{\bot} \to \widehat{\bot} \\
\widehat{\neg\, t} & \equiv & \widehat{t} \to \widehat{\bot} \\
\widehat{t_1 \wedge t_2} & \equiv & \Pi\ \mathtt{C} : \mathtt{Type},\ (\widehat{t_1} \to \widehat{t_2} \to \varepsilon_{\mathtt{t}}\ \mathtt{C}) \to \varepsilon_{\mathtt{t}}\ \mathtt{C} \\
\widehat{t_1 \vee t_2} & \equiv & \Pi\ \mathtt{C} : \mathtt{Type},\ (\widehat{t_1} \to \varepsilon_{\mathtt{t}}\ \mathtt{C}) \to (\widehat{t_2} \to \varepsilon_{\mathtt{t}}\ \mathtt{C}) \to \varepsilon_{\mathtt{t}}\ \mathtt{C} \\
\widehat{t_1 \Leftrightarrow t_2} & \equiv & \Pi\ \mathtt{C} : \mathtt{Type},\ ((\widehat{t_1} \to \widehat{t_2}) \to (\widehat{t_2} \to \widehat{t_1}) \to \varepsilon_{\mathtt{t}}\ \mathtt{C}) \to \varepsilon_{\mathtt{t}}\ \mathtt{C}
\end{array}$$

De même que pour l'encodage de notre logique en CoC, on obtient un encodage imprédicatif de notre logique en Lambdapi et on retrouve la notation $\widehat{\bot}$ de l'exemple précédent.

Exemple 5.18. *Considérons l'application d'une transformation qui, à partir de la tâche* T_1 *définie par :*

$$t_1 : prop,\ t_2 : prop,\ t_3 : prop \mid H : t_1 \wedge (t_2 \vee t_3) \vdash G : t_2$$

produit la tâche résultante T_2 *suivante :*

$$t_1 : prop,\ t_2 : prop,\ t_3 : prop \mid H_1 : t_1,\ H_2 : t_3 \vdash G : t_2$$

La validité d'une telle application peut être garantie si on peut donner un terme et vérifier en Lambdapi qu'il a le type $\widehat{T_2} \to \widehat{T_1}$. *Ce type est convertible à :*

$$\begin{array}{l}
(\Pi\ \mathtt{t}_1\ \mathtt{t}_2\ \mathtt{t}_3,\ \varepsilon_{\mathtt{t}}\ \mathtt{t}_1 \to \varepsilon_{\mathtt{t}}\ \mathtt{t}_3 \to \varepsilon_{\mathtt{t}}\ (\widetilde{\neg}\ \mathtt{t}_2) \to \widehat{\bot}) \to \\
(\Pi\ \mathtt{t}_1\ \mathtt{t}_2\ \mathtt{t}_3,\ \varepsilon_{\mathtt{t}}\ (\mathtt{t}_1\ \widetilde{\wedge}\ (\mathtt{t}_2\ \widetilde{\vee}\ \mathtt{t}_3)) \to \varepsilon_{\mathtt{t}}\ (\widetilde{\neg}\ \mathtt{t}_2) \to \widehat{\bot})
\end{array}$$

On peut donc donner à vérifier à Lambdapi la définition suivante (on rappelle que les symboles _ désignent des variables liées non utilisées) :

```
symbol ToVerify' : T2^ → T1^
≔ λT2 t1 t2 t3 H G,
    T2 t1 t2 t3
       (H t1 (λhleft _, hleft))
       (H (t2 ∨~ t3) (λ_ hright, hright) t3
          (λH2, G H2 t3)
          (λH3, H3))
       G;
```

La preuve de correction de l'application de la transformation donnée par cette instruction peut être décrite comme suit :

1. *Nous commençons par introduire, à l'aide de λ-abstractions,*
   — `T2` *de type* `Π t1 t2 t3, εt t1 → εt t3 → εt (¬~ t2) → ⊥^`,
   — `t1, t2` *et* `t3` *de type* `Type`,
   — `H` *de type* `εt (t1 ∧~ (t2 ∨~ t3))`,
   — `G` *de type* `εt (¬~ t2)`.
2. *Nous appliquons* `T2 t1 t2 t3`*, il nous reste donc à fournir trois termes, un premier de type* `εt t1`*, un deuxième de type* `εt t3` *et un troisième de type* `εt (¬~ t2)`*, ce dernier étant directement fourni par* `G`.
3. *Le terme de type* `εt t1` *est fourni par* `H t1 (λhleft _, hleft)`*, c'est la projection gauche de la conjonction* `H`.
4. *Le terme de type* `εt t3` *est obtenu en considérant la projection droite de* `H` *et en faisant ensuite un raisonnement par cas à partir de la disjonction obtenue. Dans le premier cas nous donnons* `λH2, G H2 t3`*. En effet nous pouvons prouver la formule voulue à partir de* `εt t2` *puisque* `G` *est la négation de cette hypothèse. Le deuxième cas, donné par* `λH3, H3`*, est trivial.*

L'exemple précédent illustre l'utilité de la notation $\widetilde{\mathtt{t}}$ pour les termes : il est parfois nécessaire de donner des termes profonds, comme `t2 ∨~ t3` dans l'exemple.

On a vu dans les exemples précédents que, lors de la vérification d'une application qui produit $T_1, \ldots, T_n$ à partir de T, on choisit de demander à Lambdapi de vérifier que le type $\widehat{T_1} \to \ldots \widehat{T_n} \to \widehat{T}$ est habité et non un type qui serait réécrit à partir de ce type. La raison est que la traduction de la correction d'une application d'une transformation en Lambdapi est dans la base de confiance de notre outil et que l'on choisit donc de ne pas ajouter une étape supplémentaire à cette traduction. En revanche, la traduction du certificat comme un terme de preuve ne fait pas partie de la base de confiance, et nous nous permettons donc de donner des termes réduits.

De plus, étant donné que la logique des tâches est une logique classique, nous étendons CoC avec l'axiome du tiers-exclu, ce qui s'exprime à travers l'instruction Lambdapi suivante

```
symbol EM : εt (∀kt DType (λt, t ∨~ ¬~ t));
```

qui nous permet de définir l'élimination de la double négation :

```
symbol FromNN : εt (∀kt DType (λt, ¬~ ¬~ t ⇒tt t))
≔ λt nn, EM t t (λt, t) (λn, nn n t);
```

Dans le paragraphe suivant, nous allons nous servir de cet axiome, et plus précisément du symbole `FromNN`, pour la définition de certains constructeurs de certificats en Lambdapi.

5.5.4 Traduction du certificat

5.5.4.1 Traduction des différents constructeurs de certificat

Pour chaque règle d'inférence définissant le prédicat $T \downarrow c$, on associe un type Lambdapi que l'on appelle *type associé* à cette règle d'inférence. Ce type associé est obtenu en considérant cette même règle, modifiée en retirant le certificat ainsi que les déclarations (dans I, Σ, Γ ou Δ) qui restent les mêmes pour tous les séquents de cette règle. La règle modifiée est finalement traduite comme une formule, cette formule devant être quantifiée universellement sur les variables libres qu'elle contient.

Exemple 5.19. *Considérons la règle d'inférence pour* KSlice *s'appliquant à une hypothèse :*

$$\frac{\Gamma, H : t_1 \vdash \Delta \downarrow c_1 \qquad \Gamma, H : t_2 \vdash \Delta \downarrow c_2}{\Gamma, H : t_1 \vee t_2 \vdash \Delta \downarrow \mathsf{KSlice}(false, t_1, t_2, H, c_1, c_2)}$$

La règle modifiée devient :

$$\frac{H : t_1 \vdash \qquad H : t_2 \vdash}{H : t_1 \vee t_2 \vdash}$$

Le type associé à cette règle est donc :

$$\texttt{Π (t}_1\ \texttt{t}_2 : \texttt{Type}),\ (\varepsilon_{\texttt{t}}\ \texttt{t}_1 \to \widehat{\bot}) \to (\varepsilon_{\texttt{t}}\ \texttt{t}_2 \to \widehat{\bot}) \to \varepsilon_{\texttt{t}}\ (\texttt{t}_1\ \widetilde{\vee}\ \texttt{t}_2) \to \widehat{\bot}$$

où l'on a traduit en Lambdapi chacune des tâches apparaissant dans la règle modifiée et où l'on a quantifié sur les variables libres.

Afin de traduire le certificat comme un terme de preuve, nous définissons, pour chaque règle d'inférence, un terme en Lambdapi qui a le type associé à cette règle, à l'exception de la règle pour le certificat KHole (voir le paragraphe 5.5.4.2 suivant). Nous appelons ce terme le *terme associé* à cette règle d'inférence. On peut voir le terme associé à une règle d'inférence comme la preuve que celle-ci est correcte dans le sens suivant : si les tâches des prémisses de cette règle sont valides, alors la tâche de la conclusion de cette règle est valide également. Autrement dit, la définition des termes associés aux règles d'inférence nous donne une preuve vérifiée par ordinateur des différents cas du théorème 3.17.

Exemple 5.20. *En Lambdapi, on définit le terme associé à la règle pour* KSlice *dans les hypothèses par l'instruction :*

$$\texttt{symbol KSliceHyp (t}_1\ \texttt{t}_2 : \texttt{Type}) :\ (\varepsilon_{\texttt{t}}\ \texttt{t}_1 \to \widehat{\bot}) \to (\varepsilon_{\texttt{t}}\ \texttt{t}_2 \to \widehat{\bot}) \to \varepsilon_{\texttt{t}}\ (\texttt{t}_1\ \widetilde{\vee}\ \texttt{t}_2) \to \widehat{\bot}$$
$$:= \lambda \texttt{H}_1\ \texttt{H}_2\ \texttt{H},\ \texttt{H}\ \widetilde{\bot}\ \texttt{H}_1\ \texttt{H}_2;$$

Cette instruction est acceptée par Lambdapi car $\varepsilon_{\texttt{t}}\ (\texttt{t}_1\ \widetilde{\vee}\ \texttt{t}_2)$*, le type de* H*, se réécrit en :*

$$\texttt{Π C},\ (\varepsilon_{\texttt{t}}\ \texttt{t}_1 \to \varepsilon_{\texttt{t}}\ \texttt{C}) \to (\varepsilon_{\texttt{t}}\ \texttt{t}_2 \to \varepsilon_{\texttt{t}}\ \texttt{C}) \to \varepsilon_{\texttt{t}}\ \texttt{C}$$

Exemple 5.21. *Pour la règle du certificat* KSwap *dans les buts, on obtient la règle modifiée :*

$$\frac{P : t \vdash}{\vdash P : \neg t}$$

et donc le type associé est :

$$\texttt{Π (t : Type), } (\varepsilon_{\texttt{t}}\ \texttt{t} \to \widehat{\bot}) \to (\varepsilon_{\texttt{t}}\ (\widetilde{\neg}\ \texttt{t}) \to \widehat{\bot}) \to \widehat{\bot}$$

On définit alors le terme associé par :

```
symbol KSwapGoal (t : Type) : (εt t → ⊥̂) → (εt (¬̃ t) → ⊥̂) → ⊥̂
≔ λnt nnt, nnt nt;
```

Exemple 5.22. *La règle dans les buts pour le certificat* KUnfoldArr *est modifiée en :*

$$\frac{\vdash G : \neg t_1 \vee t_2}{\vdash G : t_1 \Rightarrow t_2}$$

On définit le terme associé KUnfoldArrGoal *en Lambdapi par :*

```
symbol KUnfoldArrGoal (t1 t2 : Type) :
  ((εt (¬̃ t1 ∨̃ t2) → ⊥̂) → ⊥̂) → (εt (t1 ⇒tt t2) → ⊥̂) → ⊥̂
≔ λS G, G (λH1, FromNN (¬̃ t1 ∨̃ t2) S t2 (λHn1, Hn1 H1 t2) (λH2, H2));
```

Exemple 5.23. *L'encodage superficiel nous permet de donner aisément la définition du terme associé à une règle d'instanciation d'un quantificateur. Pour la quantification universelle, on écrit l'instruction suivante :*

```
symbol InstQuantHyp T (t : εt T → Type) (u : εt T) :
  ((Π x, εt (t x)) → εt (t u) → ⊥̂) → (Π x, εt (t x)) → ⊥̂
≔ λS H, S H (H u);
```

Exemple 5.24. *Les règles du certificat* KIntroQuant *sont également facilement définies, la condition de fraîcheur de la variable étant naturellement assurée par l'utilisation d'un lieur.*

```
symbol IntroQuantGoal T (t : εt T → Type) :
  (Π x, (εt (t x) → ⊥̂) → ⊥̂) → ((Π x, εt (t x)) → ⊥̂) → ⊥̂
≔ λS G, G (λu, FromNN (t u) (S u));
```

5.5.4.2 Traduction d'un certificat complet

Étant donné une application d'une transformation qui a pour tâche initiale T et pour tâches résultantes $T_1, \ldots, T_n$, le principe du vérificateur Lambdapi est de vérifier que le type $\widehat{T_1} \to \ldots \to \widehat{T_n} \to \widehat{T}$ est habité. Il est possible d'écrire une preuve de correction de l'application directement en Lambdapi, comme illustré dans l'exemple 5.18. Cependant, le vérificateur Lambdapi ne fait pas de recherche de preuve, cette dernière étant générée directement à partir du certificat renvoyé par la transformation.

Décrivons le fonctionnement du vérificateur Lambdapi lors de la génération de preuve.

1. *Les noms sont introduits à l'aide de λ-abstractions.* Les tâches résultantes sont d'abord introduites ; on donne un nom unique à chacune d'entre elles. Les noms des déclarations des symboles de type, des symboles de terme et des formules de la tâche initiale sont introduits. Nous supposons toujours qu'il existe un ordre total sur ces noms noté $<_{ident}$ et nous le respectons pour l'introduction des déclarations.

2. *Les constructeurs de certificat sont remplacés par leur terme associé.* Le certificat fournit le corps de la preuve, chaque constructeur étant remplacé par son terme associé. Ce dernier est appliqué à tous les paramètres du constructeur traduits en Lambdapi, en faisant attention à l'ordre de ces paramètres. Les nouveaux noms sont introduits par des λ-abstractions.

3. *Les feuilles des certificats sont remplacées par l'application de la tâche aux noms de ses déclarations.* Chaque feuille représente une tâche, le nom unique donné à cette tâche est alors appliqué aux noms de ses symboles de types, de ses variables, de ses hypothèses et de ses buts, en respectant l'ordre $<_{ident}$.

Exemple 5.25. *Nous reprenons l'exemple 5.18 en supposant donné le certificat*

$$\begin{aligned}&\mathsf{KDestruct}(false, t_1, t_2 \vee t_3, H, H_1, H_2,\\ &\mathsf{KSlice}(false, t_2, t_3, H_2,\\ &\qquad \mathsf{KAxiom}(t_2, H_2, G),\\ &\qquad \mathsf{KHole}(T_2)))\end{aligned}$$

Nous procédons comme suit :

1. *Comme dans l'exemple, sont introduits* $\mathtt{T_2}$, $\mathtt{t_1}$, $\mathtt{t_2}$, $\mathtt{t_3}$, $\mathtt{H}$ *et* $\mathtt{G}$

2. *On traduit les constructeurs du certificat par*

```
KDestructHyp t1 (t2 ⩒ t3) (λH1 H2,
KSliceHyp t2 t3
          (λH2, KAxiom t2 H2 G)
          (λH2, ...)
          H2)
H
```

   Remarquons que les paramètres indiquant les formules auxquelles s'appliquent les certificats (tels que t_1 et $t_2 \vee t_3$ pour KDestruct*) sont également traduits en Lambdapi.*

3. *Le certificat* $\mathsf{KHole}(T_2)$ *est traduit en une preuve de* $\widehat{\bot}$ *par*

```
T2 t1 t2 t3 H1 H2 G
```

Finalement, l'instruction suivante est transmise à Lambdapi pour vérification :

```
symbol ToVerify' : T̂₂ → T̂₁
≔ λT₂ t₁ t₂ t₃ H G,
KDestructHyp t₁ (t₂ ∨̃ t₃) (λH₁ H₂,
KSliceHyp t₂ t₃
          (λH₂, KAxiom t₂ H₂ G)
          (λH₂, T₂ t₁ t₂ t₃ H₁ H₂ G)
          H₂)
H;
```

5.5.4.3 Sémantique des certificats en Lambdapi

Un certificat est traduit par une preuve en Lambdapi. Ainsi, lorsque l'on doit prouver une certaine propriété, l'application d'un constructeur de certificat nous permet de passer à la preuve de plusieurs nouvelles propriétés. C'est ce que l'on appelle la sémantique des certificats en Lambdapi, par analogie avec les règles du prédicat binaire $T \downarrow c$.

Une différence notable entre la sémantique des certificats Lambdapi et les règles d'inférence du prédicat binaire $T \downarrow c$ est la gestion du contexte des déclarations. En effet, comme on utilise un encodage superficiel pour ce dernier, chaque hypothèse ou but est accessible comme une variable locale en Lambdapi.

Par ailleurs, puisqu'un nom donné reste accessible sous une λ-abstraction (à condition qu'il soit différent de celui de la variable liée par cette abstraction), certains termes peuvent rester dans le contexte.

Exemple 5.26. *Reprenons l'exemple 5.25 précédent et plaçons-nous au niveau de l'appel*

```
KDestructHyp t₁ (t₂ ∨̃ t₃) (λH₁ H₂, ...) H
```

Avant cet appel, nous devons prouver $\widehat{\bot}$ *et nous avons :*

```
T₂ : Π t₁ t₂ t₃, εₜ t₁ → εₜ t₃ → εₜ (¬̃ t₂) → ⊥̂
t₁ : Type
t₂ : Type
t₃ : Type
 H : εₜ (t₁ ∧̃ (t₂ ∨̃ t₃))
 G : εₜ (¬̃ t₂)
```

Après l'appel, sous $(\lambda H_1\ H,\ \ldots)$*, nous avons :*

```
   ⋮
 H : εₜ (t₁ ∧̃ (t₂ ∨̃ t₃))
H₁ : εₜ t₁
H₂ : εₜ (t₂ ∨̃ t₃)
 G : εₜ (¬̃ t₂)
```

et le terme `H` *reste accessible.*

En revanche, si le nom d'une hypothèse est lié par abstraction, cette hypothèse est oubliée.

Exemple 5.27. *Toujours sur l'exemple 5.25, on considère maintenant l'appel*

$$\texttt{KSliceHyp t}_2\ \texttt{t}_3\ (\lambda\ \texttt{H}_2,\ \ldots)\ (\lambda \texttt{H}_2,\ \ldots)\ \texttt{H}_2$$

où, pour remplir les trous, il faut prouver $\widehat{\bot}$*, dans un cas on a* $\texttt{H}_2 : \varepsilon_{\texttt{t}}\ \texttt{t}_2$ *et dans l'autre cas on a* $\texttt{H}_2 : \varepsilon_{\texttt{t}}\ \texttt{t}_3$*. Remarquons que l'on n'a alors plus accès à un terme de type* $\varepsilon_{\texttt{t}}\ (\texttt{t}_2\ \widetilde{\vee}\ \texttt{t}_3)$*.*

Cette gestion du contexte peut être formalisée par un nouveau prédicat binaire $T \downarrow' c$ qui reprend les règles du prédicat $T \downarrow c$ en modifiant la façon dont les noms sont introduits et oubliés dans celles-ci. Les nouvelles règles n'oublient pas les hypothèses et les buts, sauf lorsqu'une nouvelle formule est introduite avec un nom qui était déjà présent dans la tâche. Dans ce cas, l'ancienne formule portant ce nom est supprimée de la tâche.

La règle pour KDestruct dans les hypothèses devient alors :

$$\frac{(\Gamma, H : t_1 \wedge t_2) \cup (H_1 : t_1, H_2 : t_2) \vdash \Delta \downarrow' c}{\Gamma, H : t_1 \wedge t_2 \vdash \Delta \downarrow' \mathsf{KDestruct}(false, t_1, t_2, H, H_1, H_2, c)}$$

On fera attention au fait que les hypothèses de la prémisse ont été notées

$$(\Gamma, H : t_1 \wedge t_2) \cup (H_1 : t_1, H_2 : t_2)$$

pour signifier que H_1 ou H_2 peuvent être identiques à H ou à des identifiants de Γ. Dans le premier cas, H n'est plus déclarée dans la prémisse.

Les règles pour KClear deviennent triviales :

$$\frac{T \downarrow' c}{T \downarrow' \mathsf{KClear}(\_, \_, \_, c)}$$

La règle pour KHole doit aussi être modifiée de la façon suivante :

$$\frac{I' \subseteq I \qquad \Sigma' \subseteq \Sigma \qquad \Gamma' \subseteq \Gamma \qquad \Delta' \subseteq \Delta}{I \mid \Sigma \mid \Gamma \vdash \Delta \downarrow' \mathsf{KHole}(I' \mid \Sigma' \mid \Gamma' \vdash \Delta')}$$

On montre que si $T \downarrow c$, alors $T \downarrow' c$, en remarquant que les tâches intermédiaires dans $T \downarrow' c$ ont plus de déclarations que dans $T \downarrow c$. Cela revient à dire que le vérificateur Lambdapi est adapté.

5.5.5 Théories interprétées en Lambdapi

Les symboles interprétés ne font pas partie de la signature de type ni de la signature des tâches. Ces symboles sont donc déclarés en amont de l'instruction à vérifier en Lambdapi, au même titre que les encodages des paragraphes 5.4.2 et 5.5.2.

En plus de l'existence de ces symboles, nous avons besoin de pouvoir définir un terme associé à chaque règle de certificat portant spécifiquement sur ces symboles. Au lieu de déclarer les symboles interprétés et de donner leurs termes associés comme des axiomes sur ces symboles, nous choisissons de donner des définitions aux symboles interprétés et de montrer les propriétés voulues dans Lambdapi. La confiance en ce vérificateur dépend donc du fait que nous avons correctement traduit les symboles interprétés, mais ne repose pas sur des axiomes supplémentaires donnés dans Lambdapi.

5.5.5.1 Théorie de l'égalité polymorphe

Nous utilisons comme définition de l'égalité en Lambdapi celle qui est traditionnellement attribuée à Leibniz :

```
symbol eq (T : Type) (x y : εt T) : Type
≔ ∀kt (T ⇒tk DType) (λQ, Q x ⇒tt Q y);
```

On peut alors aisément définir les termes associés aux règles des certificats KRewrite et KEqRefl. Pour KEqRefl, on écrit :

```
symbol KEqRefl (T : Type) (t : εt T) : (εt (eq T t t) → ⊥̂) → ⊥̂
≔ λneq, neq (λQ qt, qt);
```

On donne le terme associé à la règle KRewrite dans les hypothèses, le terme correspondant dans les buts étant obtenu de manière similaire :

```
symbol KRewriteHyp (T : Type) (t1 t2 : εt T) (t : εt T → Type) :
    (εt (eq T t1 t2) → εt (t t2) → ⊥̂) →
    (εt (eq T t1 t2) → εt (t t1) → ⊥̂)
≔ λS H H′, S H (H t H′);
```

Ces définitions sont suffisantes pour obtenir, par combinaison de certificats, les propriétés de symétrie, de transitivité et de congruence de l'égalité. Nous pouvons également démontrer ces propriétés en Lambdapi directement, comme dans les exemples suivants.

Exemple 5.28. *La propriété de symétrie de l'égalité se prouve de la façon suivante en Lambdapi :*

```
symbol Symmetry (T : Type) (t1 t2 : εt T) : εt (eq T t1 t2) → εt (eq T t2 t1)
≔ λe12, e12 (λt, eq T t t1) (λQ q1, q1);
```

Exemple 5.29. *La propriété de transitivité de l'égalité se prouve de la façon suivante :*

```
symbol Transitivity (T : Type) (t1 t2 t3 : εt T) :
    εt (eq T t1 t2) → εt (eq T t2 t3) → εt (eq T t1 t3)
≔ λe12 e23, e23 (λt, eq T t1 t) e12;
```

5.5.5.2 Théorie des entiers relatifs

La formalisation des théories interprétées, au même titre que la formalisation de la logique d'ordre supérieur, est contenue dans la base de confiance de notre outil. Nous préférons donc définir les types interprétés et montrer les propriétés nécessaires sur ces types plutôt que de les axiomatiser. Cela m'a amené à définir une bibliothèque pour les entiers en Lambdapi où le type des entiers est défini comme un type algébrique. L'encodage ternaire équilibré des entiers relatifs, tel qu'il est présenté par Contejean et al. [32], a été envisagé mais n'a pas

été retenu car cela demanderait de s'assurer que notre système de réécriture reste confluent après chaque nouvelle définition d'une fonction prenant un entier en argument. En effet, à partir d'un terme de la forme `f` (3×0), on peut appliquer directement la définition de `f` ou commencer par réécrire 3×0 en `0`. À la place de cet encodage, nous nous inspirons de l'encodage binaire des entiers de la bibliothèque standard de Coq [1] en reprenant notamment certaines notations.

Définition des entiers relatifs. Nous commençons par définir le type profond pour les entiers `DZ`, le type superficiel correspondant est le type des entiers et est donné par application de $\varepsilon_{\mathtt{t}}$:

```
symbol DZ : Type;
symbol Z ≔ εt DZ;
```

Supposons pour l'instant donnés un type profond `Dpositive` et le type superficiel correspondant `positive` qui désigne les entiers strictement positifs. On définit alors les entiers relatifs comme étant soit 0, soit des entiers strictement positifs, soit la négation d'entiers strictement positifs :

```
symbol Z0 : Z;
symbol Zpos : positive → Z;
symbol Zneg : positive → Z;
```

Ces définitions n'expriment pas le fait que les entiers relatifs sont nécessairement d'une de ces trois formes. Cette propriété est exprimée par le "destructeur" `ind_Z`, déclaré par :

```
symbol ind_Z : εt (∀kt (DZ ⇒tk DType) (λQ,
                   Q Z0 ⇒tt
                   (∀tt Dpositive (λp, Q (Zpos p)) ⇒tt
                   (∀tt Dpositive (λp, Q (Zneg p)) ⇒tt
                   ∀tt DZ (λz, Q z));
```

La déclaration de `ind_Z` est en fait le schéma d'induction sur les entiers relatifs ainsi définis. Puisque cette définition n'est pas récursive, il s'agit ici simplement d'une propriété nous permettant de raisonner par cas sur les entiers relatifs.

Le destructeur `ind_Z` nous permet également de définir des fonctions faisant une analyse de cas sur un argument entier. À cette fin, nous devons aussi donner des règles de réécriture pour chacun des cas possibles :

```
rule ind_Z _ $v0 _ _Z0 ↪ $v0
with ind_Z _ _ $vpos _ (Zpos $x) ↪ $vpos $x
with ind_Z _ _ _ $vneg (Zneg $x) ↪ $vneg $x;
```

Exemple 5.30. *La fonction de négation sur les entiers relatifs peut être donnée par :*

```
symbol opp : Z → Z
≔ ind_Z (λ_, DZ)
        Z0
        (λx, Zneg x)
        (λx, Zpos x);
```

On montre que cette fonction est involution en faisant usage des règles de réécriture de `ind_Z` *:*

```
symbol oppInvolution (z : Z) : εt (eq DZ (opp (opp z)) z)
≔ ind_Z (λz, eq DZ (opp (opp z)) z)
        (λQ qz, qz)
        (λx Q qz, qz)
        (λx Q qz, qz)
        z
```

Définition des entiers strictement positifs. Précisons maintenant comment le type `positive` et ses constructeurs sont définis.

```
symbol Dpositive : Type;
symbol positive ≔ εt Dpositive;

symbol xI : positive → positive;
symbol xO : positive → positive;
symbol xH : positive;
```

Cela définit des entiers binaires, le symbole `xH` décrivant l'entier 1, `xO` la multiplication par 2 et `xI` la multiplication par 2 à laquelle on ajoute 1. Ainsi, en utilisant ces constructeurs, un entier est noté en binaire où le *bit* de poids faible est à gauche, `xO` remplace le 0, `xI` remplace le 1 sauf pour le dernier *bit* (tout à droite, qui est à 1) qui est remplacé par `xH`.

Exemple 5.31. *Dans* `positive`*, l'entier* 13 *se note* `xI (xO (xI xH))` *et l'entier* 42 *se note* `xO (xI (xO (xI (xO xH))))`*. Dans* `Z`*,* −13 *se note* `Zneg (xI (xO (xI xH)))` *et l'entier* 3 *se note* `Zpos (xI xH)`*.*

Sur le type `positive` on définit également un destructeur, noté `ind_positive`.

```
symbol ind_positive : εt (∀kt (Dpositive ⇒tk DType) (λQ,
                          (∀tt Dpositive (λp, Q p ⇒tt Q (xI p)) ⇒tt
                          (∀tt Dpositive (λp, Q p ⇒tt Q (xO p)) ⇒tt
                          Q xH ⇒tt
                          ∀tt Dpositive (λx, Q x));
```

La définition de `positive` étant récursive, ce destructeur a des hypothèses supplémentaires, représentées ici par les deux occurrences de `Q p`, qui correspondent à l'hypothèse de récurrence ou à l'appel récursif.

Nous donnons également les règles de réécriture suivantes :

```
rule ind_positive $Q $vI $vO $vH (xI $p) ↪ $vI $p (ind_positive $Q $vI $vO $vH $p)
with ind_positive $Q $vI $vO $vH (xO $p) ↪ $vO $p (ind_positive $Q $vI $vO $vH $p)
with ind_positive _ _ _ $vH xH ↪ $vH;
```

Ces règles permettent de définir des fonctions récursives sur un argument de type `positive`. En effet, si cet argument est `xH` alors le cas `vH` est renvoyé, si cet argument est de la forme `xI p` (resp. `xO p`) alors le cas `vI` (resp. `vO`) est renvoyé, ce cas dépendant alors de `p` et de l'appel récursif sur celui-ci : `ind_positive Q vI vO vH p`.

Exemple 5.32. *En partant des équations suivantes*

$$\begin{array}{rcccl} (2*x+1) & + & 1 & = & 2*(x+1) \\ (2*x) & + & 1 & = & 2*x+1 \\ 1 & + & 1 & = & 2*1 \end{array}$$

on en vient à la définition de la fonction successeur sur `positive`*, notée* `succ` *:*

```
symbol succ : positive → positive
≔ ind_positive (λ_, Dpositive)
               (λp succ_p, xO succ_p)
               (λp succ_p, xI p)
               (xO xH);
```

Nous utilisons cette méthode générique pour définir des fonctions arithmétiques et démontrer leurs propriétés.

Exemple 5.33. *On souhaite définir la fonction sur* `positive` *qui multiplie par* 2 *et qui retire* 1*. En partant des équations suivantes*

$$\begin{array}{rcccccl} (2*x+1) & * & 2 & - & 1 & = & 2*(2*x)+1 \\ (2*x) & * & 2 & - & 1 & = & 2*(x*2-1)+1 \\ 1 & * & 2 & - & 1 & = & 1 \end{array}$$

on en vient à la définition :

```
symbol pred_double : positive → positive
≔ ind_positive (λ_, Dpositive)
               (λp pred_double_p, xI (xO p))
               (λp pred_double_p, xI pred_double_p)
               xH;
```

La preuve de la propriété $\varepsilon_{\texttt{t}}$ `(pred_double (succ p) = xI p)` *en Lambdapi se fait alors par induction. Les cas où* `p` *est égal à* `xH` *ou est de la forme* `xO q` *se réduisent à la réflexivité de l'égalité. Le cas où* `p` *est de la forme* `xI q` *se réduit à montrer qu'on a l'égalité :*

```
xI (pred_double (succ q)) = xI (xI q)
```

ce que l'on obtient par hypothèse de récurrence.

```
symbol ind_positive_peano :
  εt (∀tt Dpositive (λp,
     ∀kt (Dpositive ⇒tk DType) (λQ,
     Q xH ⇒tt
     (∀tt Dpositive (λx, Q x ⇒tt Q (succ x))) ⇒tt
     Q p)))
≔
ind_positive
(λp, ∀kt _ (λR, R xH ⇒tt (∀tt Dpositive (λx, R x ⇒tt R (succ x))) ⇒tt R p))
(λx rec_x Q qH qS,
  qS (x0 x) (rec_x Q² (qS xH qH) qS²))
(λx rec_x Q qH qS,
  rec_x Q² (qS xH qH) qS²)
(λQ qH qS, qH);
```

FIGURE 5.9 – Induction des entiers strictement positifs à la Peano

Récurrence à la Peano. La figure 5.9 définit le terme `ind_positive_peano` qui représente une preuve, sur les entiers strictement positifs binaires, de la propriété d'induction usuelle sur les entiers naturels. Le fait que la quantification sur `p` soit placée au début de cette propriété est uniquement lié à la présentation de sa preuve. En effet, cette dernière se fait par récurrence sur `p`, où l'on a noté `Q²` pour `λx, Q (x0 x)` et `qS²` pour

$$\lambda x\ q2x,\ qS\ (xI\ x)\ (qS\ (x0\ x)\ q2x)$$

Nous détaillons la preuve donnée dans la figure dans le cas où `p` est de la forme `xI x`, le cas `x0` est strictement plus simple et le cas `xH` est trivial. On suppose que l'on a accès aux termes ayant les types suivants

$$\begin{aligned} x &: positive \\ rec\_x &: \Pi\ R,\ \varepsilon_t\ (R\ xH) \to (\Pi\ x,\ \varepsilon_t\ (R\ x) \to \varepsilon_t\ (R\ (succ\ x))) \to \varepsilon_t\ (R\ x) \\ Q &: positive \to Type \\ qH &: \varepsilon_t\ (Q\ xH) \\ qS &: \Pi\ x,\ \varepsilon_t\ (Q\ x) \to Q\ (succ\ x) \end{aligned}$$

et on veut montrer qu'il existe un élément de type :

$$\varepsilon_t\ (Q\ (xI\ x))$$

En appliquant `qS (x0 x)`, cela revient à montrer qu'il existe :

$$q0 : \varepsilon_t\ (Q\ (x0\ x))$$

L'idée centrale de la preuve vient ensuite du fait que l'hypothèse d'induction est suffisamment générale pour nous permettre de choisir le prédicat `R`. On choisit le prédicat `Q` sur les entiers

pairs, c'est-à-dire $\texttt{Q}^2$. Il nous reste donc à donner deux termes, l'un de type `Q (xO xH)` ce que l'on fait par `qS xH qH`, et l'autre de type Π `x`, $\varepsilon_\texttt{t}$ `(Q (xO x))` $\to \varepsilon_\texttt{t}$ `(Q (xO (succ x)))`. Ce dernier terme est donné en appliquant deux fois `qS` : c'est $\texttt{qS}^2$.

Cette propriété Lambdapi est cruciale pour montrer l'induction forte sur les entiers relatifs, et est donc, à ce titre, centrale dans notre développement. Par ailleurs, cette propriété permet de prouver de nouvelles propriétés portant sur les entiers strictement positifs en s'appuyant sur la preuve donnée dans les entiers naturels, comme illustré par l'exemple 5.35.

On peut vouloir aller plus loin en montrant la propriété suivante :

```
symbol rew_ind_peano :
  Π p, Π Q, Π qH, Π qS,
  ε_t (ind_positive_peano Q qH qS (succ p) =
     qS p ind_positive_peano Q qH qS p)
≔ ...
```

Il est alors possible de définir une fonction récursive sur les entiers strictement positifs en utilisant `ind_positive_peano` à la place de `ind_positive` puis de prouver les propriétés sur cette fonction grâce à `rew_ind_peano`, exactement comme si nous travaillions avec des entiers unaires. Cependant, les définitions des fonctions obtenues de cette manière effectuent nécessairement un nombre d'opérations supérieur à l'entier donné en entrée. Sa complexité est donc exponentielle par rapport à la taille de l'entrée (le nombre de chiffres de l'entier binaire). Cela vient du fait que la fonction `qS` est modifiée à chaque appel récursif pour effectuer deux fois plus d'opérations. Nous choisissons donc de définir chaque fonction récursive à l'aide de `ind_positive`, ce qui demande de prouver qu'elle se comporte comme son équivalent dans les entiers unaires avant de pouvoir nous appuyer sur sa preuve. En particulier, nous définissons la fonction d'addition des entiers strictement positifs récursivement sur les entiers binaires donnés en entrée, ce qui rend sa complexité linéaire en la taille de l'entrée.

Addition. L'addition de deux entiers strictement positifs dépend d'une retenue, paramètre booléen qui est représenté par l'entier 0 ou l'entier 1. Cette fonction est définie récursivement, détaillons le cas où l'on souhaite ajouter `p` à `q` lorsque l'on a une retenue, que `p` est de la forme `xI x` et que `q` est de la forme `xO y`. À partir de l'équation $(2{*}x{+}1){+}(2{*}y){+}1 = 2{*}(x{+}y{+}1)$, ce cas récursif est donné par l'application de `xO` à l'addition de `x` avec `y` avec retenue.

Afin de pouvoir implémenter `add`, l'addition d'entiers relatifs de type $\texttt{Z} \to \texttt{Z} \to \texttt{Z}$, nous devons considérer le cas où les entiers n'ont pas le même signe. Cela nous amène à considérer la soustraction d'entiers strictement positifs (à résultat dans les entiers relatifs). Cette soustraction est définie à partir des opérations pour multiplier par 2, pour multiplier par 2 et ajouter 1 et pour multiplier par 2 et soustraire 1. Sur les entiers strictement positifs ce sont les opérations `xO`, `xI` et `pred_double`, et nous les adaptons aux entiers relatifs.

Nous souhaitons également prouver des propriétés arithmétiques, telles que l'associativité de l'addition. Ces preuves nécessitent de passer par des lemmes intermédiaires qui doivent parfois être généralisés dans le cas de preuves par récurrence.

Soustraction et multiplication. La soustraction sur les entiers relatifs est définie à partir de l'addition `add` et de la négation unaire `opp` :

```
symbol sub : Z → Z → Z
≔ λx y, add x (opp y);
```

La définition de la multiplication `mul` se fait également à partir de la définition de l'addition.

Comparaison. Nous définissons une fonction de comparaison `compare` sur les entiers strictement positifs qui renvoie un terme du type `comparaison`. Ce type a trois éléments : `Eq`, `Gt` et `Lt`. Lorsque `compare` renvoie `Eq` cela signifie que les arguments sont égaux, le premier argument est strictement supérieur quand `Gt` est renvoyé, strictement inférieur quand `Lt` est renvoyé. À partir de cette fonction nous pouvons facilement définir une fonction sur les entiers relatifs qui renvoie également un terme de type `comparaison`. Nous spécialisons ensuite cette fonction pour obtenir les fonctions de comparaison habituelles. On s'aide en particulier de la fonction `CompOpp` qui inverse une comparaison. Autrement dit, `CompOpp` est de type `comparaison → comparaison` et est telle que :

$$\varepsilon_{\mathtt{t}}\ (\mathtt{CompOpp\ Eq} = \mathtt{Eq})$$
$$\varepsilon_{\mathtt{t}}\ (\mathtt{CompOpp\ Lt} = \mathtt{Gt})$$
$$\varepsilon_{\mathtt{t}}\ (\mathtt{CompOpp\ Gt} = \mathtt{Lt})$$

La fonction `compare` est définie à l'aide d'une fonction auxiliaire `compare_acc` de type :

$$\mathtt{positive} \to \mathtt{positive} \to \mathtt{comparaison} \to \mathtt{comparaison}$$

le troisième argument est un accumulateur permettant de trancher en cas d'égalité et est initialisé à `Eq` pour `compare`.

Nous définissons une autre fonction de comparaison `compare'`, égale point à point avec `compare`, et qui est directement définie récursivement sur ses arguments. Cela permet de donner des preuves sans avoir à les généraliser pour prendre en compte l'accumulateur, ce qui explique l'intérêt de cette définition supplémentaire de la comparaison.

Exemple 5.34. *On souhaite prouver une propriété de commutativité sur la comparaison. Pour* `compare'`*, on peut montrer par récurrence que*

$$\varepsilon_{\mathtt{t}}\ (\mathtt{compare'\ x\ y} = \mathtt{CompOpp\ (compare'\ y\ x)})$$

pour tout `x` *et* `y`*. En revanche, lorsque l'on essaye de prouver par récurrence que*

$$\varepsilon_{\mathtt{t}}\ (\mathtt{compare\ x\ y} = \mathtt{CompOpp\ (compare\ y\ x)})$$

pour tout `x` *et tout* `y`*, l'hypothèse de récurrence ne suffit pas. En effet, cette dernière fait intervenir un accumulateur qui est fixé à* `Eq` *alors que les cas de la récurrence peuvent avoir modifié l'accumulateur dans le but à prouver. La propriété que l'on veut prouver n'est pas assez forte et il faut la généraliser pour prendre en compte l'accumulateur. Dans ce cas, une propriété généralisée qui convient est donnée par :*

$$\varepsilon_{\mathtt{t}}\ (\mathtt{compare\_acc\ x\ y\ acc} = \mathtt{CompOpp\ (compare\_acc\ y\ x\ (CompOpp\ acc))})$$

pour tout `x`*,* `y`*,* `acc`*.*

Par ailleurs, la définition de `compare` est récursive terminale et c'est cette fonction que nous utilisons pour définir les spécialisations de la comparaison (telle que le prédicat "inférieur ou égal"). Nous bénéficions ainsi des avantages des deux définitions de la comparaison. Remarquons aussi que certaines propositions s'expriment naturellement avec un accumulateur.

Exemple 5.35. *Nous utilisons la propriété* `ind_positive_peano` *afin de prouver que la comparaison est compatible avec l'addition. Il reste alors à montrer que :*

$$\varepsilon_{\mathtt{t}}\ (\mathtt{compare\_acc}\ (\mathtt{succ}\ \mathtt{x})\ (\mathtt{succ}\ \mathtt{y})\ \mathtt{acc} = \mathtt{compare\_acc}\ \mathtt{x}\ \mathtt{y}\ \mathtt{acc})$$

pour tout `x`*,* `y`*,* `acc`*. La preuve par récurrence nous amène à devoir prouver*

$$\varepsilon_{\mathtt{t}}\ (\mathtt{compare\_acc}\ (\mathtt{succ}\ \mathtt{x})\ \mathtt{y}\ \mathtt{Lt} = \mathtt{compare\_acc}\ \mathtt{x}\ \mathtt{y}\ \mathtt{Gt})$$

pour tout `x` *et tout* `y`*, propriété qui n'a pas d'équivalent simple faisant intervenir* `compare′`*.*

Preuve du principe d'induction. Le principe d'induction énoncé dans le paragraphe 4.3.2 adapte aux entiers relatifs le principe d'induction forte sur les entiers naturels et permet de choisir a, la borne inférieure.

Lambdapi offre un grand choix de symboles pour définir les opérations souhaitées, et il est possible de définir la précédence de ces symboles, de déclarer qu'un symbole utilise la notation infixe et s'il est associatif à gauche ou à droite. Ces qualités sont mises à profit dans le développement Lambdapi et nous les appliquons dans ce paragraphe par souci de concision. Par exemple, lorsque `x` et `y` sont de type `Z`, on note $\mathtt{x} < \mathtt{y}$ la comparaison stricte (obtenue par spécialisation de `compare`) de type `DType`.

En nous appuyant sur la propriété `ind_positive_peano`, nous commençons par démontrer le principe d'induction adapté aux entiers relatifs :

```
symbol natlike_rec Q :
  εt (Q 0) →
  (Π z, εt (0 ≤ z) → εt (Q z) → εt (Q (z + 1))) →
  Π z, εt (0 ≤ z) → εt (Q z)
≔ ...
```

Cette propriété nous permet alors de montrer le principe d'induction forte pour la borne 0 :

```
symbol strong_bound_0_rec Q :
  (Π z, εt (z ≤ 0) → εt (Q z)) →
  (Π z, εt (0 ≤ z) → (Π v, εt (v ≤ z) → εt (Q v)) → εt (Q (z + 1))) →
  Π z, εt (Q z)
≔ ...
```

La preuve consiste à appliquer `natlike_rec` à la propriété

$$\mathtt{R} \coloneqq \lambda\ \mathtt{v},\ \Pi\ \mathtt{u},\ \varepsilon_{\mathtt{t}}\ (\mathtt{u} \leq \mathtt{v}) \to \varepsilon_{\mathtt{t}}\ (\mathtt{Q}\ \mathtt{u})$$

ce qui nous donne `Π v, R v` et on termine la preuve en choisissant le maximum entre `0` et `z` pour `v`.

Enfin, la preuve de `strong_rec`, la propriété voulue, est obtenue par compatibilité de l'addition avec la comparaison, en considérant le prédicat translaté λ`z, Q (z + a)`.

```
symbol strong_rec a Q :
  (Π z, εt (z ≤ a) → εt (Q z)) →
  (Π z, εt (a < z) → (Π v, εt (v < z) → εt (Q v)) → εt (Q z)) →
  Π z, εt (Q z)
≔ ...
```

Règles de réécriture et certificat Conv en Lambdapi. Puisque Lambdapi offre la possibilité à l'utilisateur de définir de nouvelles règles de réécriture, la vérification du certificat Conv demande peu d'ajouts dans notre développement. Plus précisément, parmi les règles décrites dans le paragraphe 4.3.1, les règles d'application des opérateurs arithmétiques à des littéraux et les règles ayant un littéral en premier argument d'un opérateur sont déjà présentes dans notre développement, par définition de ces opérateurs. Nous complétons ces règles par les règles manquantes, à savoir :

```
rule add $x 0 ↪ $x
with sub $x 0 ↪ $x
with mul $x 0 ↪ 0
with mul $x 1 ↪ $x;
```

Nous nommons dans la suite R_2 cet ensemble de règles et R_1 l'ensemble des règles de réécriture introduites pour l'encodage des types algébriques tels que `positive` et `Z`. Montrons que l'ensemble constitué de toutes les règles de R_1, de toutes les règles de R_2 et de la β-réduction est confluent et fortement normalisant, cela garantit que Lambdapi répond en temps fini à un problème de typage. L'ensemble des règles de R_1 avec la β-réduction est fortement normalisant et confluent car il s'agit d'un schéma de récursion primitive (voir par exemple [14,15]). Les règles de R_2 réécrivant uniquement un terme par un de ses sous-termes, leur ajout à R_1 et à la règle de β-réduction reste un système fortement normalisant. Par ailleurs, l'ensemble de toutes les règles est localement confluent donc confluent.

5.6 Certificats de noyau et vérificateurs

Les ensembles des applications qui sont garanties correctes par ces vérificateurs définis dans ce chapitre sont distincts. En effet, un vérificateur peut ou non garantir qu'une application est correcte lorsque celle-ci (est correcte et) produit un certificat qui n'est pas adapté.

Nous avons vu dans le paragraphe 5.3.1 que le vérificateur OCaml permet de vérifier des certificats qui, du fait de leurs paramètres, ne sont pas adaptés. Le vérificateur Lambdapi, en revanche, fait usage de ces paramètres, comme décrit dans le paragraphe 5.5.4.2. Il existe donc des applications où le vérificateur OCaml renvoie *true* alors que le vérificateur Lambdapi renvoie $false$.

Exemple 5.36. *Considérons l'application qui, à partir de la tâche*

$$T := t_1 : prop, t_2 : prop \mid H : t_1 \wedge t_2 \vdash$$

produit la tâche

$$T' := H_1 : t_1, H_2 : t_2 \vdash$$

et le certificat

$$\mathsf{KDestruct}(false, t_2, t_2, H, H_1, H_2, \mathsf{KHole}(T'))$$

Cette application est vérifiée par le vérificateur OCaml mais pas par le vérificateur Lambdapi. En effet, on aura une erreur de typage quand on applique le terme `H` *de type* $\varepsilon_{\mathtt{t}}\ (\mathtt{t_1} \mathbin{\widetilde{\wedge}} \mathtt{t_2})$ *: le type* $\varepsilon_{\mathtt{t}}\ (\mathtt{t_2} \mathbin{\widetilde{\wedge}} \mathtt{t_2})$ *est attendu car* `KDestructHyp` *est appliqué à* $\mathtt{t_2}$ *et* $\mathtt{t_2}$*. Un certificat adapté (et donc vérifié par les deux vérificateurs) est donné par*

$$\mathsf{KDestruct}(false, t_1, t_2, H, H_1, H_2, \mathsf{KHole}(T'))$$

Un vérificateur donné peut utiliser ou non ces paramètres, ce qui représente une aide à la définition des vérificateurs.

D'un autre côté, la gestion des noms en Lambdapi diffère de celle de la sémantique des certificats (paragraphe 5.5.4.3), contrairement au vérificateur OCaml. Il existe donc des applications où le vérificateur Lambdapi renvoie *true* alors que le vérificateur OCaml renvoie $false$.

Exemple 5.37. *Considérons l'application qui produit la liste vide à partir de la tâche*

$$T := t_1 : prop, t_2 : prop \mid H : t_1 \wedge t_2 \vdash G : (t_1 \wedge t_2) \wedge t_2$$

Le certificat

$$\begin{aligned}&\mathsf{KDestruct}(false, t_1, t_2, H, H_1, H_2,\\ &\quad \mathsf{KSlice}(true, t_1 \wedge t_2, t_2, G,\\ &\quad\quad \mathsf{KAxiom}(t_1 \wedge t_2, H, G),\\ &\quad\quad \mathsf{KAxiom}(t_2, H_2, G)))\end{aligned}$$

est vérifié par le vérificateur Lambdapi. Il n'est pas vérifié par le vérificateur OCaml et échoue au niveau de $\mathsf{KAxiom}(t_1 \wedge t_2, H, G)$ *car la tâche ne contient alors plus la déclaration* $H : t_1 \wedge t_2$*.*

Un certificat adapté à cette application est donné par

$$\begin{aligned}&\mathsf{KSlice}(true, t_1 \wedge t_2, t_2, G,\\ &\quad \mathsf{KAxiom}(t_1 \wedge t_2, H, G),\\ &\quad \mathsf{KDestruct}(false, t_1, t_2, H, H_1, H_2,\\ &\quad\quad \mathsf{KAxiom}(t_2, H_2, G)))\end{aligned}$$

Par ailleurs, les certificats de noyau sont aussi élémentaires que possible, ce qui rend la définition de nouveaux vérificateurs plus simple. Un bénéfice de cette simplification est alors que les vérificateurs eux-mêmes sont plus simples à vérifier. Par exemple, pour le vérificateur OCaml, cela correspond à une implémentation plus naturelle et rend possible l'inspection de son code, ce qui renforce la confiance que l'on accorde à ce vérificateur. Nous souhaitons également faciliter l'instrumentation des transformations, ce qui nous amène à définir un nouveau format de certificat.

Chapitre 6

Certificats de surface

Les certificats de noyau ont été conçus afin de simplifier la vérification : un certificat renvoyé par application d'une transformation décrit précisément comment il s'applique et quelles sont les tâches résultantes. Il en découle les limitations des certificats de noyau qui sont exprimées dans le paragraphe 3.7 et qui peuvent nuire à la génération de certificats. Il devient alors délicat d'instrumenter les transformations afin qu'elles produisent des certificats lorsque ceux-ci sont verbeux et peu modulaires. Notre solution consiste à définir des *certificats de surface* (paragraphe 6.1), conçus pour faciliter la génération de certificats. L'intérêt des certificats de surface est qu'ils permettent un développement modulaire des transformations certifiantes (paragraphe 6.2 et paragraphe 6.3). Les transformations logiques sont donc instrumentées pour produire un certificat de surface qui est traduit en un certificat de noyau (paragraphe 6.4) afin d'être vérifié. Cette traduction ne fait pas partie de la base de confiance de notre outil, ce qui nous offre une certaine flexibilité, tant au niveau implémentation qu'au niveau de la représentation que nous souhaitons donner aux certificats manipulés par les transformations.

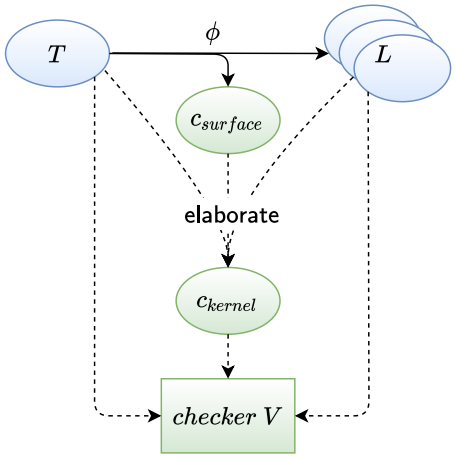

FIGURE 6.1 – Vérification des certificats de surface

$$
\begin{array}{rcl}
sc & ::= & \mathsf{Hole}(task) \\
& | & \mathsf{Let}(ident, bool \to formula \to sc) \\
& | & \mathsf{Clear}(ident, sc) \\
& | & \mathsf{Forget}(ident, sc) \\
& | & \mathsf{Assert}(ident, formula, sc, sc) \\
& | & \mathsf{Axiom}(ident, ident) \\
& | & \mathsf{Conv}(ident, formula, sc) \\
& | & \mathsf{Trivial}(ident) \\
& | & \mathsf{Swap}(ident, sc) \\
& | & \mathsf{Slice}(ident, sc, sc) \\
& | & \mathsf{Destruct}(ident, ident, ident, sc) \\
& | & \mathsf{Unfold}(ident, sc) \\
& | & \mathsf{IntroQuant}(ident, ident, sc) \\
& | & \mathsf{InstQuant}(ident, ident, term, sc) \\
& | & \mathsf{IntroType}(ident, ident, sc) \\
& | & \mathsf{InstType}(ident, ident, type, sc) \\
& | & \mathsf{Rewrite}(ident, ident, sc) \\
& | & \mathsf{Induction}(ident, ident, ident, ident, ident, term, sc, sc)
\end{array}
$$

FIGURE 6.2 – Grammaire des nœuds de certificat de surface

6.1 Définitions et notations

6.1.1 Nœuds de certificat de surface

La figure 6.2 définit les *nœuds de certificat de surface* désignés par *sc*. À l'exception du nœud Let qui prend en argument une fonction, les noms des constructeurs des nœuds de certificat de surface ont été choisis de façon à correspondre avec un ou plusieurs constructeurs d'un certificat de noyau. Par exemple, le nœud de certificat de surface Unfold correspond à KUnfoldArr et à KUnfoldIff.

6.1.1.1 Sémantique

De la même manière que pour les certificats de noyau, la sémantique des certificats est définie par un prédicat binaire $T \Downarrow c$, reliant une tâche initiale T à un nœud de certificat de surface c. Informellement, ce prédicat affirme que le nœud de certificat c garantit que la validité des feuilles de c entraîne celle de T. On donne la liste complète des règles d'inférence qui définissent le jugement $T \Downarrow c$ dans les figures 6.3 à 6.7.

À l'exception du constructeur Let, la sémantique d'un nœud de certificat de surface est similaire à celle du certificat de noyau dont le nom correspond. Par exemple, le constructeur Swap, de même que le certificat de noyau KSwap, permet aussi de traiter le cas de la négation en changeant la position d'une formule mais il ne prend en argument ni la formule en question, ni sa position initiale. On remarque que la plupart des nœuds de certificat de surface ne dépendent pas des formules et de leurs positions.

Le constructeur Let est utile quand on connaît le nom d'une hypothèse ou d'un but mais pas la formule ni la position qu'il représente. Le premier argument de Let est donc le nom et son deuxième argument est un certificat paramétré par la position et la formule ; c'est donc une fonction mathématique dans $bool \to formula \to sc$. Nous utilisons ici la notion de

syntaxe abstraite d'ordre supérieur [73] afin de définir le lieur Let.

Exemple 6.1. *Soit T une tâche et P un identifiant d'une formule de cette tâche. Considérons l'application d'une transformation qui clôt la tâche initiale T lorsque P désigne une formule triviale et qui ne change pas la tâche dans le cas contraire. On a $T \Downarrow c$ avec c défini par $\mathsf{Let}(P, f)$ où f est la fonction qui, à un booléen b et à une formule t, associe $\mathsf{Trivial}(P)$ dans le cas où b est $true$ et t est $\top$, associe aussi $\mathsf{Trivial}(P)$ dans le cas où b est $false$ et t est $\bot$, et associe $\mathsf{Hole}(T)$ sinon.*

Exemple 6.2. *Considérons les tâches*

$$T := \vdash P_1 : t_1, P_2 : t_2$$
$$T' := \vdash P : t_1 \vee t_2$$

Le certificat c défini par

$$\begin{array}{l}\mathsf{Assert}(P, t_1 \vee t_2,\\ \quad \mathsf{Clear}(P_1, \mathsf{Clear}(P_2, \mathsf{Hole}(T'))),\\ \quad \mathsf{Slice}(P, \mathsf{Axiom}(P, P_1), \mathsf{Axiom}(P, P_2)))\end{array}$$

est tel que $T \Downarrow c$. On définit un nœud de certificat c' indépendant des formules t_1 et t_2 par :

$$\begin{array}{l}\mathsf{Let}(P_1, b_1 \mapsto u_1 \mapsto\\ \mathsf{Let}(P_2, b_2 \mapsto u_2 \mapsto\\ \mathsf{Assert}(P, u_1 \vee u_2,\\ \quad \mathsf{Clear}(P_1, \mathsf{Clear}(P_2, \mathsf{Hole}(T'))),\\ \quad \mathsf{Slice}(P, \mathsf{Axiom}(P, P_1), \mathsf{Axiom}(P, P_2)))))\end{array}$$

Pour toutes formules v_1 et v_2, si T et T' sont maintenant définies par

$$T := \vdash P_1 : v_1, P_2 : v_2$$
$$T' := \vdash P : v_1 \vee v_2$$

on a encore $T \Downarrow c'$.

6.1.1.2 Correction

Par analogie avec les certificats de noyau, nous appelons *feuilles* de c la liste des tâches contenues dans les constructeurs Hole du nœud de certificat de surface c.

Théorème 6.3 (Correction du prédicat $T \Downarrow c$)**.** *Si $T \Downarrow c$ alors la validité des feuilles de c implique la validité de la tâche T.*

Nous omettons la preuve de ce théorème et l'explication détaillée des nœuds de certificat de surface pour les raisons suivantes. Premièrement, on peut observer une certaine similarité entre les prédicats $T \Downarrow c$ et $T \downarrow c$: la preuve de correction peut être obtenue de la même manière et nous donnons, dans la suite de ce chapitre, une explication plus formelle de la différence entre les deux sortes de certificats, notamment à travers des exemples. Deuxièmement, puisque ce sont les certificats de noyau et non les certificats de surface qui sont vérifiés, la preuve de correction des certificats de surface n'ajoute rien à la confiance que l'on peut accorder à notre système de vérification.

$$\frac{}{\Gamma \vdash \Delta \Downarrow \mathsf{Hole}(\Gamma \vdash \Delta)}$$

$$\frac{\Gamma, H : t \vdash \Delta \Downarrow f(false)(t)}{\Gamma, H : t \vdash \Delta \Downarrow \mathsf{Let}(H, f)} \qquad \frac{\Gamma \vdash \Delta, G : t \Downarrow f(true)(t)}{\Gamma \vdash \Delta, G : t \Downarrow \mathsf{Let}(G, f)}$$

$$\frac{\Gamma \vdash \Delta \Downarrow c}{\Gamma, H : t \vdash \Delta \Downarrow \mathsf{Clear}(H, c)} \qquad \frac{\Gamma \vdash \Delta \Downarrow c}{\Gamma \vdash \Delta, G : t \Downarrow \mathsf{Clear}(G, c)}$$

$$\frac{\Sigma \mid \Gamma \vdash \Delta \Downarrow c \qquad x \text{ n'apparaît pas dans } \Gamma, \Delta}{\Sigma, x : \tau \mid \Gamma \vdash \Delta \Downarrow \mathsf{Forget}(x, c)}$$

$$\frac{\Sigma \mid \Gamma \vdash \Delta, P : f \Downarrow c_1 \qquad \Sigma \mid \Gamma, P : f \vdash \Delta \Downarrow c_2 \qquad \Sigma \Vdash f : prop \qquad P \notin \Gamma \cup \Delta}{\Sigma \mid \Gamma \vdash \Delta \Downarrow \mathsf{Assert}(P, f, c_1, c_2)}$$

$$\frac{}{\Gamma, P_1 : t \vdash \Delta, P_2 : t \Downarrow \mathsf{Axiom}(P_1, P_2)} \qquad \frac{}{\Gamma, P_1 : t \vdash \Delta, P_2 : t \Downarrow \mathsf{Axiom}(P_2, P_1)}$$

FIGURE 6.3 – Sémantique des nœuds de certificat (groupe identité et groupe structurel)

$$\frac{\Gamma, H : f_2 \vdash \Delta \Downarrow c \qquad f_1 \equiv f_2}{\Gamma, H : f_1 \vdash \Delta \Downarrow \mathsf{Conv}(H, f_2, c)}$$

$$\frac{\Gamma \vdash \Delta, G : f_2 \Downarrow c \qquad f_1 \equiv f_2}{\Gamma \vdash \Delta, G : f_1 \Downarrow \mathsf{Conv}(G, f_2, c)}$$

FIGURE 6.4 – Sémantique des nœuds de certificat (conversion)

$$\frac{}{\Gamma, H : \bot \vdash \Delta \Downarrow \mathsf{Trivial}(H)} \qquad \frac{}{\Gamma \vdash \Delta, G : \top \Downarrow \mathsf{Trivial}(G)}$$

$$\frac{\Gamma \vdash \Delta, P : t \Downarrow c}{\Gamma, P : \neg t \vdash \Delta \Downarrow \mathsf{Swap}(P, c)} \qquad \frac{\Gamma, P : t \vdash \Delta \Downarrow c}{\Gamma \vdash \Delta, P : \neg t \Downarrow \mathsf{Swap}(P, c)}$$

$$\frac{\Gamma, H : t_1 \vdash \Delta \Downarrow c_1 \qquad \Gamma, H : t_2 \vdash \Delta \Downarrow c_2}{\Gamma, H : t_1 \vee t_2 \vdash \Delta \Downarrow \mathsf{Slice}(H, c_1, c_2)} \qquad \frac{\Gamma \vdash \Delta, G : t_1 \Downarrow c_1 \qquad \Gamma \vdash \Delta, G : t_2 \Downarrow c_2}{\Gamma \vdash \Delta, G : t_1 \wedge t_2 \Downarrow \mathsf{Slice}(G, c_1, c_2)}$$

$$\frac{\Gamma, H_1 : t_1, H_2 : t_2 \vdash \Delta \Downarrow c \qquad H_1, H_2 \notin \Gamma \cup \Delta}{\Gamma, H : t_1 \wedge t_2 \vdash \Delta \Downarrow \mathsf{Destruct}(H, H_1, H_2, c)}$$

$$\frac{\Gamma \vdash \Delta, G_1 : t_1, G_2 : t_2 \Downarrow c \qquad G_1, G_2 \notin \Gamma \cup \Delta}{\Gamma \vdash \Delta, G : t_1 \vee t_2 \Downarrow \mathsf{Destruct}(G, G_1, G_2, c)}$$

$$\frac{\Gamma, H : (t_1 \Rightarrow t_2) \wedge (t_2 \Rightarrow t_1) \vdash \Delta \Downarrow c}{\Gamma, H : t_1 \Leftrightarrow t_2 \vdash \Delta \Downarrow \mathsf{Unfold}(H, c)} \qquad \frac{\Gamma \vdash \Delta, G : (t_1 \Rightarrow t_2) \wedge (t_2 \Rightarrow t_1) \Downarrow c}{\Gamma \vdash \Delta, G : t_1 \Leftrightarrow t_2 \Downarrow \mathsf{Unfold}(G, c)}$$

$$\frac{\Gamma, H : \neg t_1 \vee t_2 \vdash \Delta \Downarrow c}{\Gamma, H : t_1 \Rightarrow t_2 \vdash \Delta \Downarrow \mathsf{Unfold}(H, c)} \qquad \frac{\Gamma \vdash \Delta, G : \neg t_1 \vee t_2 \Downarrow c}{\Gamma \vdash \Delta, G : t_1 \Rightarrow t_2 \Downarrow \mathsf{Unfold}(G, c)}$$

FIGURE 6.5 – Sémantique des nœuds de certificat (propositionnels)

$$\frac{\Sigma, y : \tau \mid \Gamma, H : t[x \mapsto y] \vdash \Delta \Downarrow c \qquad y \notin \Sigma}{\Sigma \mid \Gamma, H : \exists x : \tau.\ t \vdash \Delta \Downarrow \mathsf{IntroQuant}(H, y, c)}$$

$$\frac{\Sigma, y : \tau \mid \Gamma \vdash \Delta, G : t[x \mapsto y] \Downarrow c \qquad y \notin \Sigma}{\Sigma \mid \Gamma \vdash \Delta, G : \forall x : \tau. t \Downarrow \mathsf{IntroQuant}(G, y, c)}$$

$$\frac{\Sigma \mid \Gamma, H_1 : \forall x : \tau. t, H_2 : t[x \mapsto u] \vdash \Delta \Downarrow c \qquad \Sigma \Vdash u : \tau \qquad H_2 \notin (\Gamma, H_1 : \forall x : \tau.\ t) \cup \Delta}{\Sigma \mid \Gamma, H_1 : \forall x : \tau. t \vdash \Delta \Downarrow \mathsf{InstQuant}(H_1, H_2, u, c)}$$

$$\frac{\Sigma \mid \Gamma \vdash \Delta, G_1 : \exists x : \tau.\ t, G_2 : t[x \mapsto u] \Downarrow c \qquad \Sigma \Vdash u : \tau \qquad G_2 \notin \Gamma \cup (\Delta, G_1 : \exists x : \tau.\ t)}{\Sigma \mid \Gamma \vdash \Delta, G_1 : \exists x : \tau.\ t \Downarrow \mathsf{InstQuant}(G_1, G_2, u, c)}$$

$$\frac{I, \iota : 0 \mid \Sigma \mid \Gamma \vdash \Delta, G : f[\alpha \mapsto \iota] \Downarrow c \qquad \iota \notin I}{I \mid \Sigma \mid \Gamma \vdash \Delta, G : \Pi\alpha.\ f \Downarrow \mathsf{IntroType}(G, \iota, c)}$$

$$\frac{\Gamma, H_1 : \Pi\alpha.\ f, H_2 : f[\alpha \mapsto \tau] \vdash \Delta \Downarrow c \qquad V_\alpha(\tau) = \emptyset \qquad H_2 \notin (\Gamma, H_1 : \Pi\alpha.\ f) \cup \Delta}{\Gamma, H_1 : \Pi\alpha.\ f \vdash \Delta \Downarrow \mathsf{InstType}(H_1, H_2, \tau, c)}$$

FIGURE 6.6 – Sémantique des nœuds de certificat (quantificateurs)

$$\frac{}{\Gamma \vdash \Delta, G : t = t \Downarrow \mathsf{Trivial}(G)}$$

$$\frac{\Gamma, H : t_1 = t_2, H' : t[t_2] \vdash \Delta \Downarrow c}{\Gamma, H : t_1 = t_2, H' : t[t_1] \vdash \Delta \Downarrow \mathsf{Rewrite}(H, H', c)}$$

$$\frac{\Gamma, H : t_1 = t_2 \vdash \Delta, G : t[t_2] \Downarrow c}{\Gamma, H : t_1 = t_2 \vdash \Delta, G : t[t_1] \Downarrow \mathsf{Rewrite}(H, G, c)}$$

$$\frac{\begin{array}{c}\Gamma, H_i : i \leq a \vdash \Delta, G : t[i] \Downarrow c_{base} \qquad i \text{ n'apparaît pas dans } \Gamma \text{ ou } \Delta \\ \Gamma, H_i : a < i, H_{rec} : \forall n : int,\ n < i \Rightarrow t[n] \vdash \Delta, G : t[i] \Downarrow c_{rec}\end{array}}{\Gamma \vdash \Delta, G : t[i] \Downarrow \mathsf{Induction}(G, H_i, H_{rec}, i, a, c_{base}, c_{rec})}$$

FIGURE 6.7 – Sémantique des nœuds de certificat (théories interprétées)

L'intérêt d'énoncer le théorème précédent, et plus généralement de définir le prédicat $T \Downarrow c$ ainsi que ses règles, est de fournir une interface à l'instrumentation des transformations logiques. Le prédicat $T \Downarrow c$ peut alors être vu comme une documentation formelle des nœuds de certificat. Nous montrons dans le paragraphe 6.4 qu'une transformation peut être instrumentée avec succès en ne connaissant que la sémantique des certificats de surface et pas celle des certificats de noyau. Dans ce contexte, le théorème précédent peut être vu comme un moyen d'exprimer l'utilité des certificats de surface : ceux-ci peuvent nous permettre de montrer la correction des transformations.

6.1.2 Certificats de surface

Définition 6.4 (Certificat de surface)**.** *Un certificat de surface est une fonction prenant en argument un certain nombre de nœuds de certificat de surface et renvoyant un nœud de certificat de surface.*

Un certificat de surface est une fonction mathématique, ce qui permet d'utiliser le langage mathématique pour définir le nœud de certificat renvoyé, comme nous le verrons dans l'exemple 6.6. Nous utilisons, ici aussi, la notion de syntaxe abstraite d'ordre supérieur.

La plupart du temps, le résultat renvoyé par un certificat de surface est directement donné par un constructeur d'un nœud de certificat instancié avec les arguments du certificat de surface. Dans ce cas, on utilise la notation $\Lambda c_1\ \ldots\ c_n.\ c$ pour désigner le certificat de surface prenant les n nœuds de certificat de surface c_1, …, c_n en argument et renvoyant le nœud certificat de surface c.

Exemple 6.5. *Un certificat de surface peut n'avoir aucun argument :*

$$\mathsf{Trivial}(H)$$

Les certificats de surface peuvent aussi avoir plusieurs arguments, et sont invariants par α-renommage. La notation

$$\Lambda c_1\ c_2.\ \mathsf{Slice}(H, c_1, c_2)$$

et la notation

$$\Lambda c_2\ d_1.\ \mathsf{Slice}(H, c_2, d_1)$$

désignent le même certificat.

6.1.2.1 Sémantique

Les nœuds de certificat pris en argument d'un certificat de surface sont utiles pour combiner les certificats (paragraphe 6.2.2.2) et sont à considérer comme les trous des certificats de noyau : ils sont censés être, à terme, des tâches résultantes de l'application de la transformation certifiante. Pour une application d'une transformation certifiante qui produit la liste de tâches résultantes L de longueur n et le certificat de surface C ayant le même nombre n d'arguments, on dit qu'on associe les arguments de C à L. La définition de la sémantique des certificats de surface suivante précise comment comprendre ces associations.

Étant donné une tâche T, et une liste de tâches L, la sémantique d'un certificat de surface C est donnée par un prédicat ternaire $T \overset{C}{\Longleftarrow} L$. Informellement, celui-ci affirme que, en suivant les associations entre C et L, C garantit que la validité des tâches de L entraîne celle de T. Plus précisément, si la longueur de L est égale au nombre d'arguments n de C, en écrivant $L := [T_1;\ \ldots;\ T_n]$, alors le prédicat $T \overset{C}{\Longleftarrow} L$ est vrai si et seulement si, pour tous nœuds de certificat de surface $c_1, \ldots, c_n$ tels que $T_j \Downarrow c_j$, on a $T \Downarrow C(c_1)\ldots(c_n)$. La notation $C(c_1)\ldots(c_n)$ désigne le nœud de certificat de surface obtenu par applications successives du certificat de surface C vu comme une fonction.

Exemple 6.6. *Soit C le certificat de surface prenant un nœud de certificat c en argument, renvoyant* $\mathsf{Swap}(P_1, c')$ *si c est de la forme*

$$\begin{aligned}
&\mathsf{Assert}(P_2, t,\\
&\qquad \mathsf{Swap}(P_1, \mathsf{Axiom}(P_1, P_2)),\\
&\qquad \mathsf{Clear}(P_1, \mathsf{Assert}(P_1, t,\\
&\qquad\qquad\qquad \mathsf{Axiom}(P_2, P_1),\\
&\qquad\qquad\qquad \mathsf{Clear}(P_2, c'))))
\end{aligned}$$

et renvoyant c sinon. Notons que le certificat C renvoie un nœud de certificat structurellement plus petit que (ou égal à) celui pris en entrée, et montrons que l'on a $T \overset{C}{\Longleftarrow} [T]$ pour toute tâche T. C'est évident si c n'est pas de la forme précisée plus haut. Dans l'autre cas, supposons que l'on a $T \Downarrow c$, montrons que $T \Downarrow \mathsf{Swap}(P_1, c')$. En analysant les dérivations possibles de $T \Downarrow c$, on en déduit que la tâche T doit être de la forme $\Gamma \vdash \Delta, P_1 : \neg t$ et que $\Gamma, P_1 : t \vdash \Delta \Downarrow c'$. On conclut en appliquant la règle de la sémantique de Swap.

6.1.2.2 Définition syntaxique

Étant donné que les certificats de surface sont destinés à servir d'interface pour l'instrumentation des transformations, nous cherchons à donner un moyen plus direct d'établir le prédicat $T \overset{C}{\Longleftarrow} L$. L'objectif est donc de faciliter la compréhension des certificats de surface en montrant qu'ils permettent essentiellement, par rapport aux nœuds de certificat de surface, de choisir comment "remplir les trous".

Exemple 6.7. *Soit*

$$\begin{aligned} T &:= H : (t_1 \vee t_2) \vee t_3 \vdash \\ T_1 &:= H : t_1 \vdash \\ T_2 &:= H : t_2 \vdash \\ T_3 &:= H : t_3 \vdash \\ C &:= \Lambda c_1\ c_2\ c_3.\ \mathsf{Slice}(H, \mathsf{Slice}(H, c_1, c_2), c_3) \end{aligned}$$

Pour montrer qu'on a $T \overset{C}{\Longleftarrow} [T_1;\ T_2;\ T_3]$*, on doit vérifier que, pour* c_1*,* c_2 *et* c_3 *tels que*

$$\begin{aligned} T_1 &\Downarrow c_1 \\ T_2 &\Downarrow c_2 \\ T_3 &\Downarrow c_3 \end{aligned}$$

on a $T \Downarrow \mathsf{Slice}(H, \mathsf{Slice}(H, c_1, c_2), c_3)$*. On conclut en donnant la dérivation*

$$\dfrac{\dfrac{T_1 \Downarrow c_1 \qquad T_2 \Downarrow c_2}{H : t_1 \vee t_2 \vdash \Downarrow \mathsf{Slice}(H, c_1, c_2)} \qquad T_3 \Downarrow c_3}{T \Downarrow \mathsf{Slice}(H, \mathsf{Slice}(H, c_1, c_2), c_3)}$$

Rappelons aussi que la notation $L(i)$ désigne l'élément d'indice i de la liste L.

Dans le cas où le certificat de surface C est défini syntaxiquement, par exemple par $\Lambda c_1\ \ldots\ c_n.\ c$ où c est un nœud de certificat de surface sans Let, le prédicat $T \overset{C}{\Longleftarrow} L$ est vrai si et seulement si $T \Downarrow C(\mathsf{Hole}(L(1)))\ldots(\mathsf{Hole}(L(n)))$. En effet, supposons que $T \Downarrow C(\mathsf{Hole}(L(1)))\ldots(\mathsf{Hole}(L(n)))$ et montrons que $T \overset{C}{\Longleftarrow} L$, l'autre sens étant clair. Il s'agit de montrer que $T \Downarrow C(c_1)\ldots(c_n)$ pour des certificats c_1, …, c_n tels que $T_j \Downarrow c_j$. On commence par faire cette dérivation de la même manière que

$$T \Downarrow C\ (\mathsf{Hole}(T_1))\ \ldots\ (\mathsf{Hole}(T_k))$$

et à la place des feuilles

$$\dfrac{}{T_j \Downarrow \mathsf{Hole}(T_j)}$$

il faut dériver les prédicats $T_j \Downarrow c_j$, ce qui est vrai par hypothèse. Cela justifie, lorsque C est défini syntaxiquement, que l'on écrive :

$$\dfrac{T \Downarrow C(\mathsf{Hole}(L(1)))\ldots(\mathsf{Hole}(L(n)))}{T \overset{C}{\Longleftarrow} L}$$

Exemple 6.8. *La preuve de l'exemple 6.7 peut être donnée directement par la dérivation*

$$\dfrac{\dfrac{\dfrac{}{T_1 \Downarrow \mathsf{Hole}(T_1)} \quad \dfrac{}{T_2 \Downarrow \mathsf{Hole}(T_2)}}{H : t_1 \vee t_2 \vdash \Downarrow \mathsf{Slice}(H, \mathsf{Hole}(T_1), \mathsf{Hole}(T_2))} \quad \dfrac{}{T_3 \Downarrow \mathsf{Hole}(T_3)}}{T \Downarrow \mathsf{Slice}(H, \mathsf{Slice}(H, \mathsf{Hole}(T_1), \mathsf{Hole}(T_2)), \mathsf{Hole}(T_3))}$$
$$T \overset{C}{\Longleftarrow} [T_1;\ T_2;\ T_3]$$

Exemple 6.9. *On pourra écrire :*

$$\dfrac{\dfrac{}{\vdash G : \top \Downarrow \mathsf{Trivial}(G)}}{\vdash G : \top \overset{\Lambda c.\ \mathsf{Trivial}(G)}{\Longleftarrow} [\Gamma \vdash \Delta]}$$

6.1.2.3 Correction

Théorème 6.10 (Correction du prédicat $T \overset{C}{\Longleftarrow} L$)**.** *Si, pour un certificat C défini syntaxiquement, on a $T \overset{C}{\Longleftarrow} L$, alors la validité des tâches de L implique la validité de la tâche T.*

Démonstration. On a alors $T \Downarrow c$ pour un nœud de certificat de surface c tel que $\mathsf{leaf}(c)$ est une liste de tâches ne contenant que des éléments de L. Si l'on suppose la validité de toutes les tâches de L, alors toutes les tâches de $\mathsf{leaf}(c)$ sont valides aussi et, par correction de $T \Downarrow c$, T est valide. □

Exemple 6.11. *Reprenons l'exemple 3.13 et montrons que la tâche*

$$T := \vdash G : \forall q.\ q \vee \neg q$$

est valide. D'après le théorème précédent, il suffit de montrer qu'il existe un certificat C tel que $T \overset{C}{\Longleftarrow} []$. En posant

$$\begin{aligned} C := \mathsf{IntroQuant}(G, q,\\ \mathsf{Destruct}(G, P_1, P_2,\\ \mathsf{Swap}(P_2,\\ \mathsf{Axiom}(P_2, P_1)))) \end{aligned}$$

la dérivation suivante nous permet de conclure

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P_2 : q \vdash P_1 : q \Downarrow \mathsf{Axiom}(P_2, P_1)}}{\vdash P_1 : q, P_2 : \neg q \Downarrow \mathsf{Swap}(P_2, \_)}}{\vdash G : q \vee \neg q \Downarrow \mathsf{Destruct}(G, P_1, P_2, \_)}}{T \Downarrow \mathsf{IntroQuant}(G, q, \_)}}{T \overset{c}{\Longleftarrow} []}$$

6.1.3 Notation

Dans de nombreux cas d'applications de transformations, les tâches résultantes sont distinctes deux à deux et il est naturel de définir un certificat où les Hole qui y apparaissent dans l'ordre infixe correspondent à l'ordre de la liste des tâches renvoyées.

Pour cette raison, dans la suite on pourra noter les certificats de surface sous la forme de nœuds de certificat de surface où les $\mathsf{Hole}(T)$ sont remplacés par hole (sans argument). Cette notation désigne donc un certificat de surface qui a autant d'arguments que d'occurrences de hole, et où les arguments apparaissent dans l'ordre de parcours du certificat.

Exemple 6.12. *La notation $C := \mathsf{Slice}(G, \mathsf{hole}, \mathsf{hole})$ désigne le certificat de surface*

$$\Lambda c_1\ c_2.\ \mathsf{Slice}(G, c_1, c_2)$$

Lorsque l'on veut vérifier $T \overset{c}{\Longleftarrow} L$ pour un certificat c défini de cette manière, il s'agit de s'assurer $T \Downarrow c'$ pour un certificat c' qui est similaire à c mais où chaque hole a été remplacé par un nœud de certificat $\mathsf{Hole}(L(i))$ de sorte que $\mathsf{leaf}(c') = L$.

6.2 Abstraction dans les certificats

Les certificats de surface sont des fonctions, ce qui permet à la fois d'abstraire le contenu des tâches (paragraphe 6.2.1), et de définir la composition de certificats (paragraphe 6.2.2).

6.2.1 Abstraction des tâches

Dans un certificat de surface, les tâches résultantes sont abstraites et sont représentées par les arguments de celui-ci.

Exemple 6.13. *Le certificat* $\mathsf{Slice}(G, \mathsf{hole}, \mathsf{hole})$ *permet de certifier l'application d'une transformation qui, à partir de la tâche* $\Gamma \vdash G : t_1 \wedge t_2$*, produit la liste de tâches* $[\Gamma \vdash G : t_2; \Gamma \vdash G : t_1]$*. On remarque que le contexte* Γ *est arbitraire, il peut être modifié sans avoir besoin de changer le certificat. De la même manière, ce certificat permet de certifier le cas où la formule* t_1 *(resp.* t_2*) serait partout remplacée par une autre formule.*

D'une manière générale, le fait que les tâches résultantes soient abstraites du certificat permet de certifier différentes applications de transformations avec le même certificat à condition que ces applications aient le même comportement. Cela rend les certificats de surface réutilisables dans différentes situations et facilite ainsi le développement incrémental de transformations certifiantes.

6.2.2 Composition des transformations

Les certificats permettent une certification *modulaire* selon deux sens : d'une part il est possible de certifier une application d'une transformation indépendamment du traitement ultérieur des tâches résultantes, d'autre part il est possible de composer les certificats et cela permet notamment de certifier une transformation qui est obtenue par composition de transformations déjà certifiantes.

6.2.2.1 Composition de transformations logiques

Dans le cas général, l'application d'une transformation logique produit une liste de tâches qu'il nous reste à valider. Sur chacune de ces tâches il est ensuite possible d'appliquer une autre transformation. On en vient donc à la définition suivante.

Définition 6.14 (Composition de transformations)**.** *La composition d'une transformation* ϕ *avec la liste de transformations* $L := [\phi_1; \ldots; \phi_n]$*, notée* $\phi \circ L$*, est la transformation qui à partir d'une tâche* T *:*

- *applique* ϕ*, ce qui produit une liste de tâches* $\phi(T) := [T_1; \ldots; T_k]$
- *si* $k \neq n$*, alors la transformation* $\phi \circ L$ *échoue*
- *si* $k = n$*, alors on applique à chaque tâche* T_i *la transformation* ϕ_i*, ce qui produit les listes de tâches* $\phi_i(T_i)$
- *la liste de tâches résultantes est obtenue par concaténation des* $\phi_i(T_i)$

Exemple 6.15. *Considérons les transformations* $\mathsf{split\_1}$ *et* $\mathsf{split\_2}$ *qui scindent une conjonction dans les buts jusqu'aux profondeurs respectives 1 et 2. L'application de* $\mathsf{split\_1}$ *à la tâche* $T := \vdash G : (t_3 \wedge t_1) \wedge (t_2 \wedge t_1)$ *produit donc la liste de tâches* $[\vdash G : t_3 \wedge t_1; \vdash G : t_2 \wedge t_1]$*, alors que* $\mathsf{split\_2}$ *appliquée à* T *produit* $[\vdash G : t_3; \vdash G : t_1; \vdash G : t_2; \vdash G : t_1]$*. Remarquons que* $\mathsf{split\_1} \circ [\mathsf{split\_1}; \mathsf{split\_1}]$ *appliquée à* T *produit la même liste de tâches résultantes.*

Cette définition nous permet de traiter le cas général d'applications successives de transformations logiques. En particulier, si, après l'application d'une transformation, on ne souhaite pas appliquer de transformation à une des tâches obtenues, alors on peut considérer que l'on applique la transformation identité (la transformation qui produit la liste singleton composée de la tâche initiale). Remarquons aussi que la composition ne permet pas, en elle-même, de choisir l'ordre des tâches résultantes ni d'éliminer des tâches redondantes. Il serait en effet contre-intuitif si une tâche attendue n'apparaît pas parce qu'elle a déjà été obtenue par une transformation précédente.

Exemple 6.16. *Reprenons l'exemple précédent et définissons aussi* split_1_inv *qui scinde une conjonction dans les buts et qui inverse l'ordre des tâches obtenues. Alors,* split_1_inv *appliquée à la tâche* $T := \vdash G : (t_3 \wedge t_1) \wedge (t_2 \wedge t_1)$ *produit la liste de tâches* $[\vdash G : t_2 \wedge t_1; \vdash G : t_3 \wedge t_1]$. *On peut définir une transformation qui, appliquée à* T, *produit* $[\vdash G : t_1; \vdash G : t_3; \vdash G : t_1; \vdash G : t_2]$ *par* split_1 $\circ$ [split_1_inv; split_1_inv]. *En revanche on ne peut pas procéder de manière similaire pour obtenir une transformation qui, appliquée à* T, *produit* $[\vdash G : t_1; \vdash G : t_1; \vdash G : t_2; \vdash G : t_3]$.

6.2.2.2 Composition de certificats

Une transformation certifiante produit un certificat en plus d'une liste de tâches résultantes. Nous définissons la composition de certificats, ce qui nous permet alors de composer de telles transformations.

Définition 6.17 (Composition de certificats de surface)**.** *Soit un certificat de surface* C_{init} *ayant* k *arguments et une liste de certificats* $[C_1;\ \ldots;\ C_k]$ *de longueur* k *où* C_i *est un certificat de surface à* n_i *arguments. La composition de* C_{init} *avec* $[C_1;\ \ldots;\ C_k]$ *est notée* $C_{init} \otimes [C_1;\ \ldots;\ C_k]$ *et est définie par la fonction*

$$\Lambda c_1^1\ \ldots\ c_1^{n_1}\ \ldots\ c_k^1\ \ldots\ c_k^{n_k}.\ C_{init}(C_1(c_1^1)\ldots(c_1^{n_1}))\ \ldots\ (C_k(c_k^1)\ldots(c_k^{n_k}))$$

Soit C un certificat de surface et L une liste de certificats. Lorsque l'on compose C avec L on s'assurera que la longueur de L est égale au nombre d'arguments de C. Par abus de notation, étant donné un deuxième certificat C', nous écrivons $C \otimes C'$ pour désigner la composition $C \otimes [C'; \ldots; C']$ où la liste $[C'; \ldots; C']$ est de longueur égale au nombre d'arguments de C et est constituée uniquement du certificat C'.

Exemple 6.18. *La notation*

$$\mathsf{Slice}(G, \mathsf{hole}, \mathsf{hole}) \otimes \mathsf{Slice}(G, \mathsf{hole}, \mathsf{hole})$$

désigne le certificat

$$\Lambda c_1\ c_2\ c_3\ c_4.\ \mathsf{Slice}(G, \mathsf{Slice}(G, c_1, c_2), \mathsf{Slice}(G, c_3, c_4))$$

et permet de certifier par exemple l'application d'une transformation qui, à partir de la tâche $\vdash G : (t_1 \wedge t_2) \wedge (t_3 \wedge t_4)$*, produit la liste*

$$[\vdash G : t_1; \vdash G : t_2; \vdash G : t_3; \vdash G : t_4]$$

6.2.2.3 Composition de transformations certifiantes

Nous donnons dans ce paragraphe le théorème énonçant la correspondance entre la composition de transformations logiques et la composition de certificats dans le sens suivant : un

certificat obtenu par composition doit permettre de vérifier une application d'une transformation qui est également obtenue comme une composition de transformations. La cohérence de notre méthode ne repose pas sur ce théorème : le certificat obtenu par composition est vérifié dans tous les cas. Il est cependant intéressant de l'énoncer car, en plus de nous assurer que la définition de la composition de certificats est cohérente, ce théorème donne un cadre pour raisonner directement sur les certificats de surface sans passer par les nœuds de certificat de surface.

Théorème 6.19. *Si* $T_{init} \xleftarrow{C_{init}} [T_1; \ldots; T_k]$ *et pour tout* i, $T_i \xleftarrow{C_i} L_i$, *alors*

$$T_{init} \xleftarrow{C_{init} \otimes [C_1;\ \ldots;\ C_k]} L_1 +\!\!+ \ldots +\!\!+ L_k$$

Démonstration. En notant n_i la longueur de L_i, on voit que l'on cherche à montrer que

$$T_{init} \Downarrow C_{init}\ (C_1\ (c_1^1)\ \ldots\ (c_1^{n_1}))\ \ldots\ (C_k\ (c_k^1)\ \ldots\ (c_k^{n_k}))$$

pour des certificats c_i^j tels que $L_i(j) \Downarrow c_i^j$. Pour tout i, l'hypothèse $T_i \xleftarrow{C_i} L_i$ implique que $T_i \Downarrow C_i\ (c_i^1)\ \ldots\ (c_i^{n_i})$ et on conclut en appliquant l'hypothèse $T_{init} \xleftarrow{C_{init}} [T_1; \ldots; T_k]$. □

Ce théorème sera utilisé dans les dérivations en le notant comme une règle de la sémantique si ce n'est qu'on utilise une double barre pour séparer la conclusion des hypothèses. On écrira donc

$$\frac{T_{init} \xleftarrow{C_{init}} [T_1; \ldots; T_n] \qquad T_1 \xleftarrow{C_1} L_1 \qquad \ldots \qquad T_n \xleftarrow{C_n} L_n}{T_{init} \xleftarrow{C_{init} \otimes [C_1;\ \ldots;\ C_n]} L}$$

lorsque $L = L_1 +\!\!+ \ldots +\!\!+ L_n$. On écrira même

$$\frac{T_1 \xleftarrow{C_1} L_1 \qquad \ldots \qquad T_n \xleftarrow{C_n} L_n}{T_{init} \xleftarrow{C_{init} \otimes [C_1;\ \ldots;\ C_n]} L}$$

lorsqu'en plus la prémisse $T_{init} \xleftarrow{C_{init}} [T_1;\ \ldots;\ T_n]$ peut se déduire aisément. On en déduit, pour chaque règle de la sémantique des certificats, une règle avec double barre correspondante.

Exemple 6.20. *On peut écrire :*

$$\frac{P_1 : t \vdash P_2 : t \xleftarrow{C} L}{P_1 : t, P_2 : \neg t \vdash \xleftarrow{\mathsf{Swap}(P_2,\mathsf{hole}) \otimes C} L}$$

car on peut aisément dériver $P_1 : t, P_2 : \neg t \vdash \xleftarrow{\mathsf{Swap}(P_2,\mathsf{hole})} [P_1 : t \vdash P_2 : t]$ *de la façon suivante :*

$$\dfrac{\dfrac{\dfrac{}{P_1 : t \vdash P_2 : t \Downarrow \mathsf{Hole}\ (P_1 : t \vdash P_2 : t)}}{P_1 : t, P_2 : \neg t \vdash \Downarrow \mathsf{Swap}(P_2, \mathsf{Hole}\ (P_1 : t \vdash P_2 : t))}}{P_1 : t, P_2 : \neg t \vdash \xleftarrow{\mathsf{Swap}(P_2,\mathsf{hole})} [P_1 : t \vdash P_2 : t]}$$

Exemple 6.21. *Cela justifie également la notation suivante*

$$\frac{}{T \xLeftarrow{\mathsf{hole}} [T]}$$

ou encore la notation :

$$\frac{}{P_1 : t \vdash P_2 : t \xLeftarrow{\mathsf{Axiom}(P_1,P_2)} []}$$

parce que l'on peut voir $\mathsf{Axiom}(P_1, P_2)$ *comme* $\mathsf{Axiom}(P_1, P_2) \otimes []$. *On en déduit alors la dérivation suivante :*

$$\frac{\dfrac{}{P_1 : t \vdash P_2 : t \xLeftarrow{\mathsf{Axiom}(P_1,P_2)} []}}{P_1 : t, P_2 : \neg t \vdash \xLeftarrow{\mathsf{Swap}(P_2,\mathsf{hole})\otimes\mathsf{Axiom}(P_1,P_2)} []}$$

Par abus de notation, on note $[C_1;\ \ldots;\ C_n] \otimes L$ la liste de certificats $[C_1 \otimes L;\ \ldots;\ C_n \otimes L]$ et on note aussi $[C_1;\ \ldots;\ C_n] \otimes C$ le certificat $[C_1 \otimes C;\ \ldots;\ C_n \otimes C]$. L'opération $\otimes$ est donc une opération binaire sur l'union de l'ensemble des certificats et de l'ensemble des listes de certificats. Le symbole $\otimes$ est associatif à droite, ce qui nous permet d'écrire une séquence de compositions $C_1 \otimes \ldots \otimes C_n$ et de pouvoir appliquer le théorème de composition 6.19 à C_1 et à $C_2 \otimes \ldots \otimes C_n$. L'intérêt de la composition de certificats n'est pas limité à la certification de la composition de transformations. En effet, nous pouvons définir de nouveaux certificats de surface par composition sans qu'ils correspondent nécessairement à une composition de transformations certifiantes.

Exemple 6.22. *On montre que l'on a* $T \xLeftarrow{C} []$ *avec :*

$$\begin{aligned} T &:= H : t_1 \vdash G : (t_1 \vee t_2) \wedge (t_3 \vee t_1) \\ C &:= \mathsf{Slice}(G, \mathsf{hole}, \mathsf{hole}) \otimes \\ &\quad\ [C_2; C_3] \otimes \\ &\quad\ \mathsf{Axiom}(H, G_1) \end{aligned}$$

en notant $C_2 := \mathsf{Destruct}(G, G_1, G_2, \mathsf{hole})$ *et* $C_3 := \mathsf{Destruct}(G, G_3, G_1, \mathsf{hole})$. *En effet, on a la dérivation suivante :*

$$\dfrac{\dfrac{\dfrac{}{H : t_1 \vdash G_1 : t_1, G_2 : t_2 \xLeftarrow{\mathsf{Axiom}(H,G_1)} []}}{H : t_1 \vdash G : t_1 \vee t_2 \xLeftarrow{C_2 \otimes \_} []} \qquad \dfrac{\dfrac{}{H : t_1 \vdash G_3 : t_3, G_1 : t_1 \xLeftarrow{\mathsf{Axiom}(H,G_1)} []}}{H : t_1 \vdash G : t_3 \vee t_1 \xLeftarrow{C_3 \otimes \_} []}}{T \xLeftarrow{C} []}$$

6.3 Réordonner et retirer des tâches résultantes

Les certificats de surface permettent toujours de certifier les applications où les tâches résultantes redondantes ont été retirées ou réordonnées. Plus précisément, si un certificat C permet de certifier une certaine application d'une transformation produisant la liste L, alors on peut aisément, à partir de C, certifier l'application qui produit une liste L' telle que $L' \subseteq L$.

Exemple 6.23. *Considérons des tâches de la forme :*

$$\begin{aligned} T &:= \vdash G : (t_3 \wedge t_1) \wedge (t_2 \wedge t_1) \\ T_1 &:= \vdash G : t_1 \\ T_2 &:= \vdash G : t_2 \\ T_3 &:= \vdash G : t_3 \\ T_4 &:= \vdash G : t_4 \end{aligned}$$

On montre que l'application qui produit la liste de tâches résultantes $L := [T_3;\ T_1;\ T_2;\ T_1]$ est correcte en la certifiant par $C := \mathsf{Slice}(G, \mathsf{hole}, \mathsf{hole}) \otimes \mathsf{Slice}(G, \mathsf{hole}, \mathsf{hole})$. On a en effet la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{}{\vdash G : t_3 \overset{\mathsf{hole}}{\Longleftarrow} [T_3]} \quad \dfrac{}{\vdash G : t_1 \overset{\mathsf{hole}}{\Longleftarrow} [T_1]}}{\vdash G : t_3 \wedge t_1 \overset{\mathsf{Slice}(G,\mathsf{hole},\mathsf{hole})}{\Longleftarrow} [T_3;\ T_1]} \quad \dfrac{\vdots}{\vdash G : t_2 \wedge t_1 \overset{\mathsf{Slice}(G,\mathsf{hole},\mathsf{hole})}{\Longleftarrow} [T_2;\ T_1]}}{T \overset{C}{\Longleftarrow} L}$$

Considérons maintenant l'application d'une transformation qui produit la liste $L' := [T_3;\ T_1;\ T_2]$ et remarquons que c'est la liste L où $L(4)$, le dupliqué de $L(2)$, a été retiré de la liste. Pour certifier cette application, remarquons d'abord que le certificat C peut aussi s'écrire :

$$\Lambda c_3\ c_1\ c_2\ c_1'.\ \mathsf{Slice}(G, \mathsf{Slice}(G, c_3, c_1), \mathsf{Slice}(G, c_2, c_1'))$$

On fournit alors le certificat

$$C' := \Lambda c_3\ c_1\ c_2.\ \mathsf{Slice}(G, \mathsf{Slice}(G, c_3, c_1), \mathsf{Slice}(G, c_2, c_1))$$

qui est obtenu à partir de C en identifiant les deuxième et quatrième arguments. On a bien $T \overset{C'}{\Longleftarrow} L'$ car on peut appliquer la règle suivante :

$$\dfrac{T \Downarrow \mathsf{Slice}(G, \mathsf{Slice}(G, \mathsf{Hole}(T_3), \mathsf{Hole}(T_1)), \mathsf{Slice}(G, \mathsf{Hole}(T_2), \mathsf{Hole}(T_1)))}{T \overset{C'}{\Longleftarrow} L'}$$

On peut également certifier une application qui produit la liste $L'' := [T_1;\ T_2;\ T_3]$ grâce au certificat

$$C'' := \Lambda c_1\ c_2\ c_3.\ \mathsf{Slice}(G, \mathsf{Slice}(G, c_3, c_1), \mathsf{Slice}(G, c_2, c_1))$$

obtenu en réordonnant les arguments de C'. On passe alors par la règle suivante qui a la même prémisse que la règle précédente :

$$\dfrac{T \Downarrow \mathsf{Slice}(G, \mathsf{Slice}(G, \mathsf{Hole}(T_3), \mathsf{Hole}(T_1)), \mathsf{Slice}(G, \mathsf{Hole}(T_2), \mathsf{Hole}(T_1)))}{T \overset{C''}{\Longleftarrow} L''}$$

6.4 Élaboration de certificats

Nous avons défini, dans le chapitre 5, la vérification des certificats de noyau. Pourtant, comme nous l'avons vu dans ce chapitre, les transformations sont instrumentées afin que chacune de leurs applications fournisse un certificat de surface. L'*élaboration* de certificats

vise à combler ce manque en permettant d'obtenir un certificat de noyau à partir d'un certificat de surface. Notons que l'élaboration ne fait pas partie de la base de confiance de notre cadre de vérification : le certificat de noyau est vérifié après avoir été obtenu par élaboration.

Nous souhaitons que l'instrumentation des transformations ne nécessite de connaître que les certificats de surface et leur sémantique. Nous souhaitons aussi que l'élaboration respecte la propriété suivante : si $T \overset{C}{\Longleftarrow} L$ et si c est obtenu par élaboration de C, alors $T \downarrow c$ et $\mathsf{leaf}(c) \subseteq L$. Cette propriété n'a pas à être formellement vérifiée car si l'élaboration produit un certificat inadapté, alors la transformation échoue. Cela justifie la définition suivante.

Définition 6.24 (Certificat de surface adapté)**.** *Soit une application d'une transformation ϕ à la tâche initiale T produisant la liste de tâches résultantes L et le certificat de surface C. On dit que C est* adapté *lorsque $T \overset{C}{\Longleftarrow} L$.*

Ainsi, pour instrumenter une transformation il suffit de produire, pour chaque application, un certificat de surface adapté ; et ce travail ne nécessite pas de connaître la sémantique des certificats de noyau. Décrivons maintenant le fonctionnement de l'élaboration.

6.4.1 Abstraction des tâches

Une première différence entre les certificats de noyau et les certificats de surface vient de la gestion des trous. Le prédicat $T \overset{C}{\Longleftarrow} L$ est défini par application du certificat C vu comme une fonction. Dans un premier temps, l'élaboration fait également cette application, ce qui produit un nœud de certificat de surface. Pour simplifier la présentation de l'élaboration, nous supposons que cette étape est faite automatiquement de sorte qu'on dit qu'une application d'une transformation certifiante produit un nœud de certificat de surface en plus d'une liste de tâches résultantes.

Exemple 6.25. *Soit une transformation certifiante ϕ qui scinde les disjonctions. Plutôt que de dire que l'application de ϕ à la tâche initiale $T := \Gamma, H : t_1 \wedge t_2 \vdash \Delta$ produit la tâche résultante $T' := \Gamma, H_1 : t_1, H_2 : t_2 \vdash \Delta$ et le certificat $\Lambda c.\ \mathsf{Destruct}(H, H_1, H_2, c)$, nous dirons que le certificat produit est $\mathsf{Destruct}(H, H_1, H_2, \mathsf{Hole}(T'))$.*

6.4.2 Retrouver les champs manquants des certificats

L'élaboration est implémentée par la fonction elaborate qui prend en argument un nœud de certificat de surface et la tâche initiale. La fonction elaborate est définie récursivement sur le certificat et s'appuie sur la tâche afin de pouvoir donner tous les champs d'un certificat de noyau. De cette façon, l'élaboration de certificats résout le problème de verbosité énoncé dans le paragraphe 3.7.

Exemple 6.26. *On reprend l'exemple 3.6 : on suppose qu'une application d'une transformation à la tâche initiale de la forme*

$$T := P : \neg(\neg t) \vdash G : t$$

produit une liste vide de tâches résultantes et le certificat

$$c := \mathsf{Swap}(P, \mathsf{Swap}(P, \mathsf{Axiom}(G, P)))$$

L'élaboration de ce certificat cherche dans la tâche T la formule P. Puisque P est l'hypothèse $\neg(\neg t)$, l'élaboration donne le certificat $\mathsf{KSwap}(false, \neg t, P, c_1)$ où c_1 est obtenu par

élaboration de $\mathsf{Swap}(P, \mathsf{Axiom}(G, P))$ *sur la tâche* $T_1 := \vdash G : t, P : \neg t$. *Par un raisonnement similaire, on a* $c_1 := \mathsf{KSwap}(true, t, P, c_2)$ *où* c_2 *est obtenu par élaboration de* $\mathsf{Axiom}(G, P)$ *sur la tâche* $T_2 := P : t \vdash G : t$. *Enfin, on obtient* $c_2 := \mathsf{KAxiom}(t, P, G)$, *ce qui nous donne :*

$$c = \mathsf{KSwap}(false, \neg t, P, \mathsf{KSwap}(true, t, P, \mathsf{KAxiom}(t, P, G)))$$

comme dans l'exemple initial. Remarquons que le certificat $\mathsf{Axiom}(G, P)$ *a été élaboré en* c_2 *où* G *et* P *ont été inversés car* P *est une hypothèse et* G *est un but dans la tâche* T_2.

6.4.3 Distinguer les certificats de surface

Un même certificat de surface peut être élaboré en deux certificats de noyau différents pour deux tâches initiales différentes. Plus précisément, l'élaboration de certificats a accès aux formules qui constituent les tâches de preuve et cela conditionne, pour un certificat de surface donné, quel certificat de noyau est renvoyé.

Exemple 6.27. *Le certificat* $\mathsf{Trivial}(P)$ *permet de certifier une application qui clôt une tâche lorsque la formule* P *est triviale. L'élaboration de ce certificat renvoie :*

- $\mathsf{KTrivial}(true, P)$ *sur une tâche initiale de la forme* $\Gamma \vdash \Delta, P : \top$
- $\mathsf{KTrivial}(false, P)$ *sur une tâche initiale de la forme* $\Gamma, P : \bot \vdash \Delta$
- $\mathsf{KEqRefl}(t, P)$ *sur une tâche initiale de la forme* $\Gamma \vdash \Delta, P : t = t$

Exemple 6.28. *Reprenons l'exemple 3.19 et supposons donné un certificat de surface* rename *pour renommer les formules. Dans le cas d'une application d'une transformation qui produit la tâche* $T' := \Gamma \vdash \Delta, P' : f$ *à partir de la tâche initiale* $T := \Gamma \vdash \Delta, P : f$, *le certificat* $\mathsf{rename}(f, P, P', \mathsf{Hole}(T'))$ *peut être élaboré en* $\mathsf{krenameGoal}(f, P, P', \mathsf{KHole}(T'))$ *défini par :*

$$\begin{aligned}&\mathsf{KAssert}(P', f,\\ &\qquad \mathsf{KClear}(true, f, P, \mathsf{KHole}(T')),\\ &\qquad \mathsf{KAxiom}(f, P', P))\end{aligned}$$

On souhaite que ce certificat soit également adapté à une application qui produit $T' := \Gamma, P' : f \vdash \Delta$ *à partir de la tâche initiale* $T := \Gamma, P : f \vdash \Delta$. *Dans ce cas, l'élaboration doit produire un certificat différent, noté* $\mathsf{krenameHyp}(f, P, P', \mathsf{KHole}(T'))$, *qui peut être défini par :*

$$\begin{aligned}&\mathsf{KAssert}(P', f,\\ &\qquad \mathsf{KAxiom}(f, P, P'),\\ &\qquad \mathsf{KClear}(false, f, P, \mathsf{KHole}(T')))\end{aligned}$$

Exemple 6.29. *On souhaite définir un certificat de surface* eqSym *adapté à une application d'une transformation qui échange les arguments d'une égalité. Dans le cas où la tâche initiale est* $T := \Gamma, P : t_1 = t_2 \vdash \Delta$, *la tâche résultante est définie par* $T' := \Gamma, P : t_2 = t_1 \vdash \Delta$. *Un certificat de noyau adapté est alors donné par*

$$\begin{aligned}&\mathsf{krenameHyp}(t_1 = t_2, P, P',\\ &\mathsf{KAssert}(P, t_2 = t_1,\\ &\qquad \mathsf{KRewrite}(true, t_1, t_2, (\lambda t.\ t_2 = t), P', P,\\ &\qquad\quad \mathsf{KEqRefl}(t_2, P))\\ &\qquad \mathsf{KClear}(false, t_1 = t_2, P',\\ &\qquad\quad \mathsf{KHole}(T'))\end{aligned}$$

Similairement à l'exemple précédent, l'élaboration ne renvoie pas le même certificat dans le cas où l'égalité est un but au lieu d'être une hypothèse. Dans ce cas, le certificat renvoyé est donné par

$$\begin{aligned}
&\mathsf{krenameGoal}(t_1 = t_2, P, P',\\
&\mathsf{KAssert}(P, t_2 = t_1,\\
&\qquad \mathsf{KClear}(\mathit{true}, t_1 = t_2, P',\\
&\qquad\quad \mathsf{KHole}(T')),\\
&\qquad \mathsf{KRewrite}(\mathit{true}, t_2, t_1, (\lambda t.\ t_1 = t), P, P',\\
&\qquad\quad \mathsf{KEqRefl}(t_1, P')))
\end{aligned}$$

6.4.4 Mécanisme générique pour créer de nouveaux certificats de surface

Les certificats de surface ont pour objectif de faciliter la génération de certificats. Pour instrumenter les transformations, nous avons vu qu'il est plus facile de produire un certificat de surface qu'un certificat de noyau. En ce sens, les certificats de surface représentent une interface pour l'instrumentation des transformations. Cependant, les exemples précédents montrent que la définition d'un nouveau certificat de surface demande parfois de connaître la forme de la tâche à laquelle il s'applique. Une solution proposée est de définir l'élaboration de ces certificats où nous avons accès à la tâche en question, mais cette solution n'est pas entièrement satisfaisante car elle demande de manipuler des certificats de noyau. Nous souhaitons nous affranchir des certificats de noyau dans la définition de nouvelles transformations certifiantes. Nous le faisons en utilisant le constructeur de certificat Let qui inspecte la forme de la tâche à laquelle il s'applique. Une fois l'élaboration de ce certificat définie, nous n'avons plus besoin d'étendre l'élaboration pour la définition d'autres certificats de surface.

L'élaboration du certificat $\mathsf{Let}(P, f)$ sur une tâche de la forme $\Gamma, P : t \vdash \Delta$ est définie par l'élaboration du certificat $f(\mathit{false})(t)$ sur cette même tâche. De même, l'élaboration de $\mathsf{Let}(P, f)$ sur une tâche de la forme $\Gamma \vdash \Delta, P : t$ est définie par l'élaboration de $f(\mathit{true})(t)$ sur cette même tâche.

Exemple 6.30. *Reprenons l'exemple 6.28 et donnons un certificat de surface permettant de renommer un but P en P' :*

$$\begin{aligned}
\mathsf{renameGoal}(f, P, P') := \Lambda c.\ \mathsf{Assert}(&P', f,\\
&\mathsf{Clear}(P, c),\\
&\mathsf{Axiom}(P, P'))
\end{aligned}$$

De même, pour renommer une hypothèse :

$$\begin{aligned}
\mathsf{renameHyp}(f, P, P') := \Lambda c.\ \mathsf{Assert}(&P', f,\\
&\mathsf{Axiom}(P, P'),\\
&\mathsf{Clear}(P, c))
\end{aligned}$$

On note $\mathsf{rename}(P, P')$ *le certificat permettant de renommer une formule P en P'. Ce certificat est donné par* $\Lambda c.\ \mathsf{Let}(P, g)$ *où g est la fonction qui, appliquée à b et à f, renvoie* $\mathsf{renameGoal}(f, P, P', c)$ *si b est true et* $\mathsf{renameHyp}(f, P, P', c)$ *sinon. Par analogie avec les constructeurs des nœuds de certificat de surface, on utilise la notation* $\mathsf{rename}(P, P', c)$ *pour* $\mathsf{rename}(P, P')(c)$.

Exemple 6.31. *Nous reprenons l'exemple 6.2 et souhaitons généraliser le certificat défini pour qu'il soit indépendant de la position des formules P_1 et P_2. Plus précisément, nous voulons toujours que ce certificat de surface certifie une application d'une transformation qui produit la tâche résultante*

$$T' := \Gamma \vdash \Delta, P : t_1 \vee t_2$$

à partir d'une tâche initiale de la forme

$$T := \Gamma \vdash \Delta, P_1 : t_1, P_2 : t_2$$

mais nous voulons aussi que ce même certificat soit adapté lorsque la tâche initiale est de la forme

$$T := \Gamma, P_1 : t_1, P_2 : t_2 \vdash \Delta$$

et que la tâche résultante est de la forme

$$T' := \Gamma, P : t_1 \wedge t_2 \vdash \Delta$$

Ce certificat est noté $\mathsf{construct}(P_1, P_2, P)$ *(ou* $\mathsf{construct}(P_1, P_2, P, \mathsf{hole})$*) et est donné par* $\Lambda c.\ \mathsf{Let}(P_1, f_1)$ *où f_1 est une fonction qui, appliquée à b_1 et t_1, renvoie le certificat* $\mathsf{Let}(P_2, f_2)$ *où f_2 est la fonction qui, appliquée à b_2 et t_2, renvoie, lorsque b_1 est true :*

$$\begin{aligned}&\mathsf{Assert}(P, t_1 \vee t_2,\\ &\qquad \mathsf{Clear}(P_1, \mathsf{Clear}(P_2, c)),\\ &\qquad \mathsf{Slice}(P, \mathsf{Axiom}(P, P_1), \mathsf{Axiom}(P, P_2)))\end{aligned}$$

et renvoie, lorsque b_1 est $false$:

$$\begin{aligned}&\mathsf{Assert}(P, t_1 \wedge t_2,\\ &\qquad \mathsf{Slice}(P, \mathsf{Axiom}(P, P_1), \mathsf{Axiom}(P, P_2)),\\ &\qquad \mathsf{Clear}(P_1, \mathsf{Clear}(P_2, c)))\end{aligned}$$

6.5 Résultats et applications des certificats de surface

Les certificats de surface permettent de certifier les mêmes applications des transformations que les certificats de noyau mais l'instrumentation des transformations est facilitée. Nous avons procédé en deux étapes. Premièrement, nous avons défini les nœuds de certificat de surface, certificats qui sont, pour la plupart, affranchis des formules et de leurs positions, nous évitant ainsi d'avoir à donner ces détails lors de la production de certificats. Nous avons également défini le certificat Let permettant de retrouver à la volée la formule et la position qui correspondent à un nom donné. Ce certificat facilite la production des quelques nœuds de certificat de surface qui nécessitent toujours de donner ces détails. Deuxièmement, nous avons donné la définition des certificats de surface, certificats qui sont abstraits des tâches de preuve. Cela représente encore une fois une abstraction par rapport à une application donnée d'une transformation, ce qui permet un développement modulaire des certificats. Nous mettons en pratique les certificats de surface dans le chapitre suivant.

Chapitre 7

Application aux transformations de Why3

Présentons ici la méthode que j'ai utilisée concrètement pour obtenir des transformations certifiantes. La généralité de notre formalisme et de la logique sous-jacente aux tâches de preuve nous permet d'appliquer notre méthode de vérification à une grande variété de transformations. En particulier, le formalisme de Why3 [18], décrit dans le paragraphe 7.1, peut être traduit dans le nôtre (voir le paragraphe 7.1.2). Un pas technique est nécessaire pour contourner les contraintes liées à l'API de Why3 et pour s'abstraire de la représentation Why3 des tâches. Nous verrons, dans les paragraphes 7.2 à 7.6 différents exemples de transformations de Why3 qui ont été instrumentées avec succès, puis nous nous intéresserons, dans le paragraphe 7.7, à une transformation que nous envisageons de rendre certifiante. Nous conclurons en évaluant la pertinence de la méthode utilisée dans le paragraphe 7.8.

7.1 Formalisme logique de Why3

7.1.1 Tâches et transformations en Why3

Nous avons décrit, dans le paragraphe 1.2.2, le principe de la vérification de programmes en Why3 : cette vérification repose sur la spécification qui est donnée au programme afin de générer une tâche de preuve. La correction du programme repose alors sur la correction de la génération de la tâche de preuve et sur la validité de celle-ci. Dans notre contexte, nous nous concentrons sur ce second point et nous supposons donc que la tâche de preuve générée est valide si et seulement si le programme respecte sa spécification. Pour cette raison, nous nous attachons à décrire comment sont définies et modifiées les tâches de preuve sans nous soucier du lien avec les programmes et leurs spécifications.

Les formules et autres constructions logiques de Why3 implémentent la logique classique du premier ordre étendue avec polymorphisme et égalité. Why3 définit un certain nombre d'extensions et de théories intégrées qui sont résumées par Bobot et al. [18]. En particulier certains types de base sont prédéfinis en Why3 tels que le type des entiers noté `int` et le type des réels noté `real`. La sémantique de chacune de ces théories est donnée par un unique modèle. La théorie des entiers a pour modèle les entiers relatifs $\mathbb{Z}$, la théorie des réels a pour modèle les réels $\mathbb{R}$. Nous avons également accès à un prédicat d'égalité polymorphe qui est interprété comme l'égalité mathématique dans tous les domaines.

En Why3, le type des formules ne peut pas être mentionné explicitement mais intervient, par exemple, pour typer les prédicats.

```
use int.EuclideanDivision

predicate even (x : int)
= mod x 2 = 0
```

Les développements Why3 sont regroupés dans des modules. On a eu besoin, ici, de charger le module `EuclideanDivision` afin d'avoir accès à la définition de `mod` : nous obtenons une déclaration pour cette fonction, ainsi que des axiomes énonçant certaines de ses propriétés :

```
function mod int int : int

axiom Mod_bound :
  forall x:int, y:int. not y = 0 -> 0 <= mod x y < abs y

...
```

De la même façon, pour avoir accès à la multiplication, nous devons charger le module qui déclare les symboles usuels sur les entiers :

```
use int.Int

goal G : forall x : int. even (4*x)
```

Il est possible de définir des types polymorphes ainsi que des fonctions, des prédicats et des formules polymorphes. On déclare par exemple le type des multi-ensembles polymorphes de la façon suivante.

```
type bag 'a

constant empty : bag 'a

function add (x : 'a) (b : bag 'a) : bag 'a
```

le mot-clé `constant` étant une autre façon de déclarer des fonctions qui n'ont pas d'arguments. On peut alors déclarer une fonction polymorphe qui renvoie le cardinal du multi-ensemble donné en argument et énoncer des axiomes donnant sa définition.

```
function card (bag 'a): int

axiom Card_empty: card (empty : bag 'a) = 0

axiom Card_add: forall x : 'a, b : bag 'a. card (add x b) = 1 + card b
```

Notons aussi que Why3 permet la définition de types algébriques, et définit des extensions syntaxiques telles que `let` et `if`. Le polymorphisme de Why3 est implicite : il n'y a pas de quantificateur de type. Ce polymorphisme est prénexe et Why3 impose d'expliciter toutes les variables de type apparaissant dans une formule, ce qui explique la notation `empty : bag 'a` dans l'axiome `Card_empty` ci-dessus.

Une tâche de preuve dans Why3 est une liste de déclarations, chacune d'entre elles déclarant un nouveau type, une nouvelle variable, une hypothèse ou un but. Les tâches de preuve ont aussi la particularité d'avoir exactement un but. Remarquons que l'ordre des

déclarations importe : un type ou une variable ne peut pas être mentionné dans une déclaration antérieure qui définit ce type ou cette variable. Remarquons aussi qu'il est possible de quantifier sur plusieurs variables en même temps et que l'application d'une fonction n'est pas curryfiée.

Les transformations de Why3 sont essentiellement des fonctions OCaml qui convertissent une tâche T en une liste de tâches $[T_1; \ldots; T_n]$. Rappelons que, dans le contexte de Why3, les transformations logiques permettent, en plus de fournir une méthode de preuve interactive, de faire appel à des prouveurs automatiques. En effet, appliquer des transformations logiques bien choisies permet de transformer, étape par étape, les tâches de preuve dans la logique du prouveur automatique que l'on souhaite appeler. Par exemple, les constructions étendues (comme `let` et `if`) peuvent être éliminées d'une tâche par des transformations logiques appropriées.

Nous nous attachons maintenant à traduire les tâches de preuve de Why3 dans notre formalisme.

7.1.2 Traduction des tâches de Why3

Dans ce paragraphe, nous expliquons et motivons les différences entre les tâches de preuve de Why3 et les tâches de preuve de notre formalisme. Ces différences se manifestent au niveau de la syntaxe des tâches de preuve, de leur sémantique et au niveau de leur implémentation.

Afin de vérifier une transformation de Why3, nous définissons une fonction de traduction translate qui, suivant l'approche sceptique, est appelée à chaque application d'une transformation ϕ, à la fois sur la tâche Why3 initiale wT et sur les tâches Why3 résultantes wT_i, comme indiqué sur la figure 7.1.

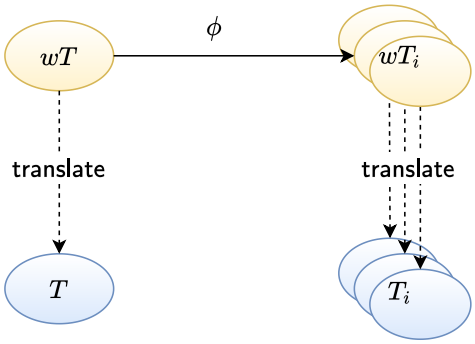

FIGURE 7.1 – Abstraction des tâches

On souhaite certifier que la validité des tâches $wT_1, \ldots, wT_n$ entraîne celle de wT. Notre solution consiste à vérifier que la validité de $T_1, \ldots, T_n$ entraîne celle de T et à faire confiance à la traduction de tâches. Cette dernière apparaît dans les deux sens : la validité de T doit entraîner celle de la tâche Why3 wT et la validité de chaque tâche Why3 wT_i doit entraîner la validité des tâches traduites T_i. Il doit donc y avoir équi-validité entre une tâche et sa traduction. À cette condition, les transformations de Why3 peuvent être vérifiées comme précédemment : elles doivent produire un certificat qui, une fois élaboré, nous permette de nous assurer que la validité des tâches résultantes $T_1, \ldots, T_n$ implique la validité de la tâche initiale T.

La traduction des tâches fait donc partie de la base de confiance de la vérification des transformations. Si, par exemple, la traduction donne toujours $\vdash G : \top$, alors toutes les

transformations peuvent être automatiquement vérifiées à tort. Pour cette raison, nous faisons attention à ce que la traduction soit relativement simple. Cela est rendu possible par le fait que notre formalisme étend la logique de Why3, et que la traduction consiste alors essentiellement en une injection de celle-ci dans notre formalisme.

7.1.2.1 Différence de syntaxe et de sémantique des tâches de preuve

Nous détaillons les différences entre les formalismes pour rendre cette inclusion explicite.

Polymorphisme implicite. Une première différence avec notre formalisme est que le polymorphisme est implicite en Why3. Voici un exemple de formule `f` avec polymorphisme implicite

```
(forall x, y : 'a. x = y) \/ not (forall x, y : 'a. x = y)
```

Par contraste, la quantification de type sur les termes est explicite dans notre formalisme. Cette formule se traduit donc par :

$$f := \Pi\alpha.\ (\forall x : \alpha.\ \forall y : \alpha.\ x = y) \vee \neg(\forall x : \alpha.\ \forall y : \alpha.\ x = y)$$

La traduction d'une tâche Why3 rend explicite la quantification de type sur les termes, cette quantification étant nécessairement en forme prénexe.

***Bug* lié au polymorphisme implicite.** Le fait d'avoir rendu explicite la quantification de type sur les termes nous a permis de trouver, dans certaines transformations de Why3, un *bug* [48] dont nous expliquons l'origine ici. On montre facilement, en reprenant la formule f précédente, que la tâche $\vdash P : f$ est valide. Il n'est cependant pas possible d'appliquer le certificat Slice à la tâche $P : f \vdash$ car la formule f n'est pas une disjonction. En Why3 la quantification de type est implicite, et l'implémentation de certaines transformations n'excluait pas de faire une analyse de cas quand la tâche avait une hypothèse de la forme de la formule f. De cette manière, il était possible de prouver n'importe quelle tâche de preuve en appliquant ces transformations de la façon suivante. Supposons, pour simplifier, que la tâche à prouver est la tâche avec seulement le but `false` :

```
goal G : false
```

L'application de la transformation `assert` avec pour paramètre

```
(forall x, y : 'a. x = y) \/ not (forall x, y : 'a. x = y)
```

nous donne deux tâches à prouver. Nous avons d'une part, la tâche

```
goal G : (forall x, y : 'a. x = y) \/ not (forall x, y : 'a. x = y)
```

qui est accessible aux prouveurs automatiques et est prouvée par E-prover par exemple, et d'autre part la tâche

```
axiom H : (forall x, y : 'a. x = y) \/ not (forall x, y : 'a. x = y)

goal G : false
```

On pouvait appliquer la transformation `destruct H` à cette dernière tâche et obtenir deux tâches, l'une avec l'hypothèse `forall x, y : 'a. x = y` et l'autre avec l'hypothèse `not (forall x, y : 'a. x =y)`. Notons qu'un certificat adapté à cette dernière application est

donné par Slice et non Destruct. Chacune des hypothèses ainsi obtenues étant contradictoire, on pouvait finalement prouver la tâche initiale. Le problème a donc été corrigé en restreignant les transformations telles que `case` ou `destruct` de sorte qu'elle ne soient appliquées qu'à des termes monomorphes.

Type des formules En Why3, le type des formules ne peut pas être mentionné directement, ce qui empêche de définir des fonctions ou prédicats prenant une formule en argument. Cette restriction n'est pas nécessaire dans notre formalisme, nous traduisons le type des formules de Why3 par *prop*.

Conclusion des séquents. Les tâches de preuve de Why3 n'admettent au plus qu'un seul but, similairement au système **LJ**, si ce n'est que la logique sous-jacente des tâches de preuve de Why3 est une logique classique et non une logique intuitionniste. Puisque cela représente une restriction par rapport à notre formalisme, ce point de la traduction est trivial.

Les séquents avec plusieurs buts permettent un traitement symétrique des buts et des hypothèses, ce qui nous a permis de regrouper des règles duales sous un même certificat de surface.

Exemple 7.1. *Le certificat* Destruct *ainsi que le certificat* construct *défini dans l'exemple 6.31 permettent de certifier des applications de transformations modifiant aussi bien une hypothèse qu'un but.*

Même si les tâches de Why3 n'ont qu'un seul but, et que, par conséquent, la tâche initiale et les tâches résultantes d'une application de transformation n'ont aussi qu'un seul but, il est tout de même possible de certifier une application d'une transformation de Why3 en passant par des tâches intermédiaires avec plusieurs buts.

Exemple 7.2. *Soit*

$$\begin{aligned} T &:= \vdash G : (t_1 \wedge t_2) \vee (u_1 \wedge u_2) \\ T_{1,1} &:= \vdash G : (t_1 \vee u_1) \\ T_{1,2} &:= \vdash G : (t_1 \vee u_2) \\ T_{2,1} &:= \vdash G : (t_2 \vee u_1) \\ T_{2,2} &:= \vdash G : (t_2 \vee u_2) \end{aligned}$$

L'application qui produit $L := [T_{1,1};\ T_{1,2};\ T_{2,1};\ T_{2,2}]$ *à partir de* T *est certifiée par*

$$\begin{aligned} C := \ & \mathsf{Destruct}(G, G_t, G_u, \mathsf{hole}) \otimes \\ & \mathsf{Slice}(G_t, \mathsf{hole}, \mathsf{hole}) \otimes \\ & \mathsf{Slice}(G_u, \mathsf{hole}, \mathsf{hole}) \otimes \\ & \mathsf{construct}(G_t, G_u, G, \mathsf{hole}) \end{aligned}$$

et les tâches intermédiaires de la dérivation $T \overset{C}{\Longleftarrow} L$ *ont plusieurs buts. On donne la dérivation suivante :*

$$\dfrac{\dfrac{D_1 \qquad D_2}{\vdash G_t : t_1 \wedge t_2, G_u : u_1 \wedge u_2 \overset{\mathsf{Slice}(G_t,\mathsf{hole},\mathsf{hole}) \otimes \_}{\Longleftarrow} L}}{T \overset{C}{\Longleftarrow} L}$$

où D_1 est la dérivation de

$$\vdash G_t : t_1, G_u : u_1 \wedge u_2 \xLeftarrow{\mathsf{Slice}(G_u,\mathsf{hole},\mathsf{hole}) \otimes C'} [T_{1,1};\ T_{1,2}]$$

et D_2 est la dérivation de

$$\vdash G_t : t_2, G_u : u_1 \wedge u_2 \xLeftarrow{\mathsf{Slice}(G_u,\mathsf{hole},\mathsf{hole}) \otimes C'} [T_{2,1};\ T_{2,2}]$$

avec $C' := \mathsf{construct}(G_t, G_u, \mathsf{hole})$. La dérivation D_2 s'obtient de façon similaire à D_1, qui, quant à elle, s'écrit :

$$\dfrac{\dfrac{\dfrac{}{T_{1,1} \xLeftarrow{\mathsf{hole}} [T_{1,1}]}}{\vdash G_t : t_1, G_u : u_1 \xLeftarrow{C'} [T_{1,1}]} \quad \dfrac{\dfrac{}{T_{1,2} \xLeftarrow{\mathsf{hole}} [T_{1,2}]}}{\vdash G_t : t_1, G_u : u_2 \xLeftarrow{C'} [T_{1,2}]}}{\vdash G_t : t_1, G_u : u_1 \wedge u_2 \xLeftarrow{\mathsf{Slice}(G_u,\mathsf{hole},\mathsf{hole}) \otimes C'} [T_{1,1};\ T_{1,2}]}$$

Entremêlement des déclarations. À l'exception du but qui est la dernière déclaration d'une tâche donnée, Why3 n'impose pas qu'un certain type de déclaration apparaisse avant un autre type de déclaration. Il est par exemple possible de déclarer une hypothèse avant de déclarer une nouvelle constante, tant que cette constante n'apparaît pas dans une déclaration antérieure. Les déclarations peuvent être données dans n'importe quel ordre respectant les dépendances entre les déclarations. Prenant en compte ces contraintes, notre définition des tâches de preuve regroupe toutes les déclarations des symboles de type dans un ensemble qui est accessible à l'ensemble de toutes les déclarations des symboles de termes, ce dernier ensemble étant lui-même accessible à l'ensemble des hypothèses et des buts. Regrouper les déclarations a facilité la définition de la sémantique des tâches de preuve, la traduction de celles-ci dans Lambdapi, et la définition des vérificateurs en général.

Exemple 7.3. *Dans la tâche de preuve Why3 suivante on a un prédicat défini avant un type, une hypothèse déclarée avant une fonction :*

```
predicate divides int int

constant degrees_triangle : int = 180

type shape

function vertex int : shape

axiom shape_by_vertex : forall s : shape. exists n : int. n >= 3 /\ s = vertex n

function degrees shape : int

axiom degrees3 : degrees (vertex 3) = degrees_triangle

axiom decompose : forall m, n. degrees (vertex (m + n + 2)) =
                  degrees (vertex (m + 2)) + degrees (vertex (n + 2))

goal G : forall s : shape. divides degrees_triangle (degrees s)
```

La traduction de cette tâche est donnée par $I \mid \Sigma \mid \Gamma \vdash \Delta$ *où* I, Σ, Γ *et* Δ *sont les ensembles contenant les déclarations suivantes :*

$$
\begin{aligned}
I := \; & shape : 0 \\
\Sigma := \; & degrees\_triangle : int, \\
& divides : int \rightsquigarrow int \rightsquigarrow prop, \\
& vertex : int \rightsquigarrow shape, \\
& degrees : shape \rightsquigarrow int \\
\Gamma := \; & degrees\_triangle\_def : degrees\_triangle = 180, \\
& shape\_by\_vertex : \forall s : shape.\ \exists n : int.\ n \geq 3 \wedge s = vertex\ n, \\
& degrees3 : degrees\ (vertex\ 3) = degrees\_triangle, \\
& decompose : \forall m\ n.\ degrees\ (vertex\ (m + n + 2)) = \\
& \qquad\qquad degrees\ (vertex\ (m + 2)) + degrees\ (vertex\ (n + 2)) \\
\Delta := \; & G : \forall s : shape.\ divides\ degrees\_triangle\ (degrees\ s)
\end{aligned}
$$

Remarquons que la tâche traduite a une hypothèse de plus que celles énoncées dans la tâche Why3. Cette hypothèse, nommée $degrees\_triangle\_def$*, rend compte de la définition du symbole* `degrees_triangle` *qui est simplement déclaré dans la traduction. Notons au passage que cette tâche n'est pas valide car l'interprétation de* `divides` *n'est pas fixée mais qu'elle le devient dès qu'on impose que cette fonction soit la divisibilité dans les entiers, en ajoutant des axiomes ou en la considérant comme un symbole interprété (voir le chapitre 4).*

Traduction des différentes fonctionnalités de Why3. Notons que nous nous intéressons aux tâches de preuve comme objet d'étude et que les fonctionnalités de Why3 qui interviennent pour générer ces tâches n'ont pas à être traduites. Nous ne nous soucions donc pas du langage de programmation WhyML ni de la manière dont a été générée la tâche de preuve initiale. Le système de module [46], par exemple, ne sera pas explicité ici. Nous ne considérons pas le langage de spécification associé à WhyML en tant que tel mais plutôt comme le langage qui décrit les formules des tâches de preuve. En particulier, le fait que Why3 permette l'utilisation de variables fantômes [45], facilitant ainsi la spécification, n'a pas d'influence sur la traduction des tâches de preuve.

Les types algébriques et les types enregistrements ne sont pas traduits dans notre formalisme et nous laissons ces améliorations pour des travaux futurs. Remarquons qu'il serait aisé de définir une traduction permettant de certifier des transformations qui ne portent pas spécifiquement sur les types enregistrements et les types algébriques. Pour un nouveau type algébrique il s'agit de déclarer un nouveau type et de traduire chaque constructeur comme une fonction qui produit ce type. Pour un type enregistrement, l'accès à un champ d'un type donné est traduit comme l'application d'une fonction qui produit un élément de ce type. Une telle traduction ne permet pas, par exemple, de certifier une transformation qui essaie de prouver que deux éléments d'un même type enregistrement sont égaux en comparant leurs champs. Une traduction plus satisfaisante aurait étendu à la fois la logique pour y inclure les types algébriques et les types enregistrements et les certificats afin de permettre les raisonnements spécifiques à ces types.

7.1.2.2 Différences d'implémentation

En plus des différences au niveau logique, la traduction doit prendre en compte les différences au niveau de l'implémentation.

Une première différence apparaît au niveau de la gestion des lieurs : Why3 adopte une approche défensive où les fonctions qui sont exposées rendent impossible les captures de variable. De notre côté, nous adoptons une représentation *locally nameless* [27] des termes, représentation qui a l'avantage de simplifier certaines preuves par induction. Puisque, nous ne vérifions pas formellement le code de la fonction ccheck pour l'instant (paragraphe 5.3.3), cette différence au niveau des lieurs n'est pas fondamentale : dans les deux cas l'implémentation a pour objectif de protéger des captures de variables.

Par ailleurs, en Why3, l'application n'est pas nécessairement binaire et il est possible de quantifier sur plusieurs variables en même temps. Nous choisissons au contraire de n'avoir que des applications binaires et que chaque lieur ne quantifie que sur une variable. Cela facilite la définition de certificats pour la quantification car nous souhaitons définir des certificats élémentaires. Par exemple, le certificat IntroQuant introduit exactement une nouvelle variable.

Enfin, les tâches de Why3 sont memoïsées et le fait que les déclarations des tâches soient ordonnées rend cette technique efficace : lorsque deux tâches sont créées après avoir chacune commencé par importer un même module, ces déclarations peuvent être partagées. De notre côté, les déclarations sont enregistrées dans des ensembles, ce qui peut rendre inefficace la memoïsation. Les tâches de notre formalisme ne sont pas memoïsées, nous laissons cette amélioration pour des travaux futurs.

7.2 Transformation rewrite

La transformation rewrite de Why3 permet de réécrire une hypothèse dans une formule qui peut être le but ou une hypothèse. L'hypothèse réécrite doit être une égalité mais peut être quantifiée universellement et peut être sous la forme d'implications. Dans ce cas, Why3 cherche une substitution qui permet d'unifier l'égalité avec un sous-terme de la formule dans laquelle on fait la réécriture. Si une telle substitution n'existe pas, alors la transformation échoue. Si elle existe, alors Why3 renvoie, d'une part, les tâches de preuve où le but est remplacé par les prémisses de l'implication auxquelles on a appliqué la substitution et, d'autre part, la tâche de preuve où la réécriture a été effectuée.

Exemple 7.4. *Plaçons-nous dans le contexte du tri par insertion de listes d'entiers où la fonction d'insertion* `ins` *et la fonction de tri* `sort` *sont axiomatisées.*

```
use list.List
use int.Int

function ins int (list int) : list int

axiom ins_nil : forall x. ins x Nil = Cons x Nil

axiom ins_lower : forall x, y, l . x <= y ->
                  ins x (Cons y l) = Cons x (Cons y l)

axiom ins_higher : forall x, y, l. x > y ->
                   ins x (Cons y l) = Cons y (ins x l)


function sort (list int) : list int

axiom sort_nil : sort Nil = Nil
```

```
axiom sort_cons : forall x, l. sort (Cons x l) = ins x (sort l)


goal G : forall n1, n2. n1 < n2 ->
         sort (Cons n2 (Cons n1 Nil)) = Cons n1 (Cons n2 Nil)
```

Après avoir introduit `n1`, `n2`, *l'hypothèse* `n1 < n2` *et utilisé les hypothèses* `ins_nil`, `sort_nil` *et* `sort_cons` *pour réécrire le but, on arrive au contexte de preuve Why3 suivant :*

```
[...] (* les déclarations précédentes sont omises *)
constant n1 : int

constant n2 : int

H : n1 < n2

goal G : ins n2 (Cons n1 Nil) = Cons n1 (Cons n2 Nil)
```

où l'application de `rewrite ins_higher` *produit une première tâche où le but est*

```
goal G : Cons n1 (ins n2 Nil) = Cons n1 (Cons n2 Nil)
```

et une deuxième où le but est

```
goal G : n2 > n1
```

Afin de réécrire une hypothèse H potentiellement quantifiée et ayant des prémisses, le certificat renvoyé commence par dupliquer H. Cela permet d'instancier cette nouvelle hypothèse sans que l'hypothèse H ne soit modifiée dans les tâches résultantes. La substitution trouvée par la transformation nous permet de savoir comment instancier les quantificateurs de cette nouvelle hypothèse. Nous obtenons le certificat voulu en appliquant cette hypothèse tant qu'elle a des prémisses et en réécrivant l'hypothèse ainsi obtenue. Nous n'oublions pas de retirer l'hypothèse qui a été ainsi dupliquée. La transformation rewrite permet aussi de réécrire une égalité de droite à gauche au lieu de gauche à droite, ce que nous certifions en appliquant la symétrie de l'égalité à l'hypothèse dupliquée.

Exemple 7.5. *Dans le contexte de l'exemple précédent, la substitution trouvée remplace* `x` *par* `n2`*, et le certificat*

$$\mathsf{InstQuant}(ins\_higher, H, n_2, c)$$

crée une nouvelle hypothèse H qui duplique l'hypothèse à réécrire ins_higher et instancie x par n_2 dans cette nouvelle hypothèse. La formule H est alors donnée par :

$$\forall y.\ \forall l.\ n_2 > y \Rightarrow ins\ n_2\ (Cons\ y\ l) = Cons\ y\ (ins\ n_2\ l)$$

Le certificat c instancie y par n_1 et l par Nil dans H. Ensuite, à partir de l'implication, ce certificat retire la prémisse $n_2 > n_1$ de H et crée une nouvelle tâche avec pour but $n_2 > n_1$ en appliquant notamment les certificats Unfold *et* Slice*. Enfin, le certificat* Rewrite *est appliqué, puis le certificat* Clear *afin de retirer l'hypothèse H spécifique à la certification.*

La transformation rewrite n'a pas été instrumentée afin de produire un certificat adapté lorsque l'on réécrit sous des quantificateurs. En effet, la substitution permettant d'instancier l'hypothèse à réécrire ne doit l'instancier qu'avec des termes clos. Afin de réécrire sous les

quantificateurs il est toujours possible de les introduire avant d'appliquer la transformation rewrite, comme nous l'avons fait dans les exemples précédents. Dans un deuxième temps nous pouvons aussi généraliser les variables ainsi introduites.

Nous laissons pour des travaux futurs la génération de certificats adaptés lorsque la transformation rewrite cherche à réécrire sous les quantificateurs. Nous rappelons que cela ne compromet pas la correction de la méthode. Remarquons que nous utilisons ici l'aspect incrémental de notre approche.

7.3 Transformation compute

La transformation compute s'applique à une déclaration identifiée au préalable et la modifie en effectuant différentes réductions. Les règles de réécriture appliquées ne contiennent pas seulement celles du paragraphe 4.3.1. Par exemple, la transformation compute peut s'appliquer à la théorie des nombres réels ou simplifier des formules propositionnelles.

Exemple 7.6. *L'application de* compute *à la tâche* $\vdash G : \top \wedge t$ *donne la tâche* $\vdash G : t$.

Cette transformation est rendue certifiante en fournissant le certificat KConv avec pour paramètre l'identifiant de la déclaration sur laquelle s'applique la transformation. Le certificat renvoyé n'est donc pas toujours adapté car les règles de réécriture ne couvrent pas l'ensemble des réductions que peut effectuer compute. Nous laissons ce point pour une amélioration future. Par ailleurs, nous rappelons qu'à l'heure actuelle, le certificat KConv fait automatiquement échouer les applications certifiantes qui utilisent le vérificateur OCaml.

7.4 Transformation blast

La transformation blast est une transformation que j'ai ajoutée à Why3 et qui a pour objectif de traiter la logique propositionnelle contenue dans la tâche initiale. La méthode utilisée pour définir blast est de décomposer complètement tous les symboles propositionnels ($\top$, $\bot$, $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$) en tête de formule sur le principe de la méthode des tableaux [75]. La transformation clôt ensuite toutes les tâches triviales ainsi obtenues, les tâches triviales étant les tâches qui contiennent la même formule dans les hypothèses et dans les buts, qui contiennent pour $\top$ dans les buts, ou qui contiennent $\bot$ dans les hypothèses. Dans ce paragraphe, nous appelons formule *p-atomique* une formule dont le symbole de tête n'est pas un symbole propositionnel.

Exemple 7.7. *Étant donné des formules p-atomiques* t_1, t_2 *et* t_3, *la transformation* blast *appliquée à la tâche*

$$H_1 : t_1, H_3 : t_3 \vdash G : t_1 \wedge (t_2 \wedge t_3)$$

produit l'unique tâche

$$H_1 : t_1, H_3 : t_3 \vdash G : t_2$$

Si cette transformation peut résoudre toutes les tautologies propositionnelles, le contrecoup est sa complexité. Plus précisément, le temps d'exécution de cette transformation, le nombre de tâches résultantes produites et donc la taille du certificat sont autant de facteurs qui peuvent augmenter exponentiellement par rapport à la taille de la tâche initiale.

Exemple 7.8. *La transformation* blast *appliquée à la tâche*

$$H_1 : t_{1,1} \vee t_{1,2}, \ldots, H_n : t_{n,1} \vee t_{n,2} \vdash$$

où les $t_{i,j}$ sont des formules p-atomiques produit la liste constituée des tâches

$$H_1 : t_{1,i_1}, \ldots, H_n : t_{n,i_n} \vdash$$

où $i_1, \ldots, i_n \in \{1, 2\}$.

Cette transformation certifiante donne donc un exemple de transformation qui est définie récursivement en composant des transformations certifiantes. En fonction de la tâche initiale, elle produit un certificat dont la taille peut être arbitrairement grande. Cela nous permettra de tester l'efficacité de la vérification des certificats au paragraphe 7.8.

La transformation blast est définie par récurrence comme composition de transformations certifiantes qui éliminent, chacune, un type de symbole situé en tête des formules apparaissant dans une tâche. Par exemple, la transformation unfold, détaillée dans le paragraphe 7.4.1, fait partie de ces transformations et permet de passer à une tâche où il n'y a plus de symbole d'implication $\Rightarrow$ ni de symbole d'équivalence $\Leftrightarrow$ en tête des formules apparaissant dans la tâche. Lorsque ces éliminations sont terminées, la transformation blast clôt toutes les tâches triviales. Nous détaillons maintenant les transformations dont blast est la composition.

7.4.1 Élimination de l'implication et de l'équivalence

La transformation unfold remplace toute déclaration d'une formule de la tâche initiale de la forme $P : t_1 \Rightarrow t_2$ par la déclaration $P : \neg t_1 \vee t_2$. Pour ce faire, cette transformation est définie récursivement sur la tâche vue comme une liste de déclarations et compose le certificat obtenu avec $\mathsf{Unfold}(P, \mathsf{hole})$. Notons aussi que cette transformation remplace toute formule de la tâche initiale de la forme $t_1 \Leftrightarrow t_2$ par la formule $(t_1 \Rightarrow t_2) \wedge (t_2 \Rightarrow t_1)$.

Exemple 7.9. *La transformation* unfold *appliquée à une tâche de la forme*

$$H : t_1 \Rightarrow t_2 \vdash G : t_3 \Leftrightarrow t_4$$

produit la tâche

$$H : \neg t_1 \vee t_2 \vdash G(t_3 \Rightarrow t_4) \wedge (t_4 \Rightarrow t_3)$$

et le certificat $\mathsf{Unfold}(H, \mathsf{Unfold}(G, \mathsf{hole}))$.

7.4.2 Élimination de la conjonction et de la disjonction

La transformation destruct_all permet de ne plus avoir de conjonction ni de disjonction comme symboles de tête des formules. Il y a quatre cas qui correspondent à la disjonction et à la conjonction, dans les buts ou dans les hypothèses.

Décomposition en disjonction dans les hypothèses ou en conjonction dans les buts. Une hypothèse de la forme $H : t_1 \vee t_2$ produit deux nouvelles tâches qui ont pour hypothèse $H : t_1$ et $H : t_2$. Un but de la forme $G : t_1 \wedge t_2$ produit deux nouvelles tâches, l'une ayant pour but $G : t_1$ et l'autre ayant pour but $G : t_2$. Dans les deux cas, on compose le certificat obtenu précédemment avec le certificat $\mathsf{Slice}(H, \mathsf{hole}, \mathsf{hole})$.

Décomposition en conjonction dans les hypothèses. Une hypothèse de la forme $H : t_1 \wedge t_2$ est remplacée par les deux hypothèses $H_1 : t_1$ et $H_2 : t_2$ où H_1 et H_2 sont de nouveaux noms frais. On compose le certificat obtenu avec le certificat $\mathsf{Destruct}(H, H_1, H_2, \mathsf{hole})$.

Décomposition en disjonction dans les buts. Lorsqu'un but est de la forme $G : t_1 \vee t_2$, il est remplacé par la formule t_2 et l' hypothèse $H : \neg t_1$ est ajoutée à la tâche où H est un nom frais. Le nouveau certificat est donné par $\mathsf{Destruct}(G, H, G, \mathsf{Swap}(H, \mathsf{hole}))$. Dans ce dernier cas, afin de définir une transformation certifiante, il a été nécessaire de ne pas dédoubler le but en le remplaçant par les buts $G_1 : t_1$ et $G_2 : t_2$. En effet, une transformation certifiante produit des tâches Why3 qui n'ont qu'un seul but.

Exemple 7.10. *La transformation* destruct_all *appliquée à* $H : t_1 \vee t_2 \vdash G : t_3 \vee t_4$ *produit la liste de tâches résultantes*

$$[H : t_1, H' : \neg t_3 \vdash G : t_4;\\ H : t_2, H' : \neg t_3 \vdash G : t_4]$$

et le certificat $\mathsf{Destruct}(G, H', G, \mathsf{Swap}(H', \mathsf{Slice}(H, \mathsf{hole}, \mathsf{hole})))$.

7.4.3 Traitement de la négation

De la même manière que l'on a dû faire attention à ne pas dédoubler les buts dans le cas d'une disjonction dans les buts, le traitement direct de la négation, qui consisterait à suivre la règle du certificat Swap correspondante, ne permet pas de garder un et un seul but dans la tâche. Pour résoudre ce problème, on s'autorise à garder des négations en tête de formule. Une formule *n-atomique* est soit une formule p-atomique soit la négation d'une formule p-atomique. On cherche à décomposer les formules de la tâche initiale de façon à n'avoir que des formules n-atomiques dans la tâche.

La transformation neg_decompose permet de traiter le cas où le symbole de tête d'une formule de la tâche est une négation. Si une déclaration d'une formule $P : \neg t$ apparaît dans la tâche, il y a une distinction de cas en fonction du symbole de tête de t.

Dans le cas où t est p-atomique, $\neg t$ est n-atomique, il n'y a rien à faire.

Dans le cas où le symbole de tête de t est un symbole propositionnel, on traite les différents cas comme des cas duaux de ce qui a été présenté au-dessus en utilisant le certificat Swap avant et après le certificat. Par exemple si t est de la forme $t_1 \vee t_2$ et si P est une hypothèse, on traite ce cas comme le cas de la conjonction dans les hypothèses. Autrement dit, l'hypothèse $\neg(t_1 \vee t_2)$ est remplacée par les deux hypothèses $H_1 : \neg t_1$ et $H_2 : \neg t_2$ où H_1 et H_2 sont de nouveaux noms frais. On compose le certificat obtenu précédemment avec le certificat $\mathsf{Swap}(P, \mathsf{Destruct}(P, H_1, H_2, \mathsf{Swap}(H_1, \mathsf{Swap}(H_2, \mathsf{hole}))))$.

7.4.4 Définition récursive de blast

Pour définir la transformation blast à partir des transformations précédentes, je définis également des itérateurs sur les transformations. En particulier la fonction repeat prend en argument une transformation ϕ et renvoie une transformation qui est obtenue en appliquant successivement ϕ jusqu'à ce que les tâches résultantes ne changent plus. Il s'agit ensuite d'appliquer cette fonction à la transformation qui applique d'abord unfold puis destruct_all et enfin neg_decompose et de clore les tâches triviales à l'aide de la transformation trivial définie par ailleurs.

7.4.5 Terminaison de blast

Soit ϕ la transformation qui applique unfold puis destruct_all et enfin neg_decompose. On s'intéresse au système de transition d'états constitué des listes de tâches et étiqueté uniquement avec ϕ : on a une transition de L_1 vers L_2 lorsque l'application de ϕ à chacune

des tâches de L_1 donne des listes de tâches dont la concaténation est L_2. Montrons que ce système termine.

Soit unf la fonction qui à un terme t associe le terme où tous les sous-termes de la forme $t_1 \Rightarrow t_2$ ont été remplacés par $\neg t_1 \vee t_2$ et tous les sous-termes de la forme $t_1 \Leftrightarrow t_2$ ont été remplacés par $(t_1 \Rightarrow t_2) \wedge (t_2 \Rightarrow t_1)$. Cette fonction est définie récursivement sur des sous-termes, ce qui assure sa terminaison. On définit la transformation unft qui applique unf à toutes les formules d'une tâche.

Soit R la relation associant à toute tâche T la tâche résultante obtenue par application de unft à T. On montre que R est une bisimulation et donc que l'on peut supposer, afin de prouver la terminaison de la transformation blast, que les tâches ne contiennent ni le symbole $\Rightarrow$ ni le symbole $\Leftrightarrow$. Nous omettrons les noms des formules dans les tâches, ces noms étant sans conséquence pour la preuve de terminaison.

Pour prouver que R est une bisimulation, puisque c'est en fait une fonction et que le système de transition n'a qu'une étiquette, il s'agit de montrer que ϕ et unft commutent, ce que l'on fait par récurrence sur la tâche. Le cas de base correspond à la tâche vide et est trivial. L'hérédité se prouve par analyse de cas de la formule à ajouter. Considérons par exemple le cas où le terme est une équivalence dans les hypothèses et notons $f :: L$ l'ajout d'une même formule f aux hypothèses de chacune des tâches de la liste L. À partir de $T := \Gamma, t_1 \Leftrightarrow t_2 \vdash \Delta$, on calcule :

$$\begin{aligned}
\phi([T]) &= t_1 \Rightarrow t_2 :: \\
&\quad\ t_2 \Rightarrow t_1 :: \\
&\quad\ \phi([\Gamma \vdash \Delta]) \\
\mathsf{unft}([T]) &= (\neg unf(t_1) \vee unf(t_2)) \wedge (\neg unf(t_2) \vee unf(t_1)) :: \\
&\quad\ \mathsf{unft}([\Gamma \vdash \Delta]) \\
\phi(\mathsf{unft}([T])) &= \neg unf(t_1) \vee unf(t_2) :: \\
&\quad\ \neg unf(t_2) \vee unf(t_1) :: \\
&\quad\ \phi(\mathsf{unft}([\Gamma \vdash \Delta])) \\
&= \mathsf{unft}(\phi([T]))
\end{aligned}$$

Intuitivement, ϕ et unft commutent car destruct_all et neg_decompose ne modifient une formule qu'en fonction de son symbole de tête et celui-ci est le même entre la formule $unf(f)$ et la formule f où l'on a *déplié* son symbole de tête avec unfold.

On prouve maintenant la terminaison de blast en supposant que les tâches ne contiennent ni le symbole $\Rightarrow$ ni le symbole $\Leftrightarrow$. On définit alors la mesure d'une tâche comme deux fois le nombre de symboles $\vee$ et de symboles $\wedge$ plus le nombre de symboles $\neg$. Cette multiplication par deux permet de prendre en compte le cas de l'application de la transformation destruct_all à une disjonction dans les buts. On remarque qu'une application de ϕ fait diminuer le maximum des mesures des tâches et que cette dernière valeur est dans $\mathbb{N} \cup \{-\infty\}$ qui est naturellement muni d'un ordre bien fondé, ce qui conclut la preuve de terminaison.

7.5 Transformation induction sur $\mathbb{Z}$

La transformation induction a pour paramètres un identifiant `i` et un terme entier `a` qui représente la borne inférieure de l'induction. Cette transformation s'applique à une tâche composée d'une liste de déclarations `Gamma1` où `i` n'apparaît pas comme une variable libre, d'une liste de déclarations `Gamma2` et d'un but `f` nommé `G`. Une telle tâche T est notée :

```
Gamma1
constant i : int
Gamma2
G : f
```

La transformations induction produit, à partir de T, deux tâches T_1 et T_2. La tâche T_1 a la forme

```
Gamma1
constant i : int
Hi : i <= a
Gamma2
G : f
```

et la tâche T_2 a la forme

```
Gamma1
constant i : int
Hi : a < i
Hrec : forall n. n < i -> P[n]
Gamma2
G : f
```

où il reste à déterminer `P`, la formule `P[n]` désignant `P` où l'on a remplacé `i` par `n`. Expliquons comment la formule `P` est déterminée par la transformation induction.

Dans le cas le plus simple, comme par exemple lorsque `Gamma2` est vide, `P` est la formule `f` et cette application est certifiée par Induction$(G, H_i, H_{rec}, i, a, \mathsf{hole}, \mathsf{hole})$.

Dans le cas général, `P` désigne la *généralisation* de `f` à une liste `L` de déclarations de `Gamma2`, c'est-à-dire l'implication des formules de ces déclarations et la quantification des variables déclarées. Par exemple, si `L` est constituée des déclarations `H1 : f1`, `constant x : t` et `H2 : f2`, alors `P` est la formule `f1 -> forall x : t. f2 -> f`. La liste `L` doit nécessairement contenir toutes les hypothèses de `Gamma2` où `i` apparaît comme une variable libre. Par ailleurs, afin de faciliter les preuves par induction, la transformation `induction` recherche toutes les variables susceptibles d'être utiles à la preuve telles que les variables apparaissant dans le but `f`. L'objectif de cette recherche est d'obtenir une hypothèse d'induction suffisamment générale en ajoutant des déclarations à `L`. Généraliser des variables nécessite de généraliser toutes les déclarations qui contiennent ces variables. Les autres variables de ces déclarations sont à leur tour généralisées, et ainsi de suite. La recherche effectuée par la transformation `induction` afin de déterminer la liste `L` n'a pas à être certifiée. Cette application est certifiée en généralisant le but `f` à toutes les déclarations de `L`, en appliquant le certificat Induction précédent, puis en introduisant les déclarations précédemment généralisées.

Exemple 7.11. *L'entier* `0` *étant la valeur par défaut de la borne inférieure, l'application de* `induction i` *à la tâche*

```
use int.Int

predicate p int int

predicate q int

constant i : int

constant u : int
```

```
axiom H : q u

goal G : p u i
```

produit une tâche pour le cas de base avec pour hypothèse supplémentaire

```
Init : i <= 0
```

et une autre tâche pour l'hérédité avec pour hypothèses supplémentaires

```
Init : 0 < i

Hrec : forall n:int. n < i -> (forall u:int. q u -> p u n)
```

et la version certifiante de cette transformation produit un certificat adapté pour cette application.

7.6 Transformation split

La transformation split est une transformation centrale très utilisée dans les développements Why3. En effet, elle permet de scinder la tâche courante en plusieurs tâches selon les connecteurs logiques propositionnels qui y apparaissent de façon à faciliter leur traitement par des prouveurs automatiques. Cette transformation a été définie de sorte à optimiser les appels des prouveurs automatiques, ce qui la rend d'un grand intérêt pratique. Cette transformation permet également de traiter des connecteurs asymétriques appelés `by` et `so` proposés et implémentés par Clochard [29].

La transformation split_core donne la décomposition de la formule d'une des hypothèses ou du but de la tâche courante. En fonction des paramètres donnés à la transformation split, split_core est appliquée à toutes ou à une partie des formules de la tâche. Ainsi, la transformation split peut être vue comme la composition d'un certain nombre de transformations split_core ce qui nous permet de ne considérer que la transformation split_core dans la suite de ce paragraphe. Par ailleurs, on distingue deux cas d'application de split_core : pour scinder le but et pour scinder une hypothèse. Dans tous les cas, la formule à scinder sera notée t et désignée par P, on aura donc une déclaration $P : t$ dans la tâche initiale.

Application à un but. Le premier cas d'application de split_core consiste à scinder le but : il s'agit de le décomposer comme une conjonction. Nous appelons ce cas de figure la décomposition conjonctive d'une formule de signe positif. Par exemple, si le but t de la tâche est de la forme $t_1 \vee (t_2 \wedge t_3)$, alors split_core scinde t en $t_1 \vee t_2$ et en $t_1 \vee t_3$. Dans le cas général, si split_core est appliquée au but et que la décomposition conjonctive de la formule t donne les formules $c_1, \ldots, c_m$, alors m tâches résultantes sont produites : ce sont des tâches qui ont le même contexte que la tâche initiale mais avec $c_1, \ldots, c_m$ comme but à la place de t. Pour montrer la correction d'une telle transformation, il s'agit essentiellement de montrer que l'on a bien $c_1 \wedge \ldots \wedge c_m \Rightarrow t$ et le certificat cp (c pour décomposition Conjonctive et p pour signe Positif) reflétera cette implication.

Afin de rendre possible cette première décomposition, il faut pouvoir traiter le cas du changement de signe de la formule en question. Pour ce faire, il faut également savoir comment décomposer la formule comme une disjonction. Par exemple si le but de la tâche est de la forme $\neg(t_1 \wedge (t_2 \vee t_3))$, alors la décomposition de $t_1 \wedge (t_2 \vee t_3)$ en la disjonction de $t_1 \wedge t_2$ et de $t_1 \wedge t_3$ nous permet de donner, en utilisant les lois de De Morgan, la décomposition

du but en la conjonction de $\neg t_1 \vee \neg t_2$ et de $\neg t_1 \vee \neg t_3$. Considérons le cas plus général où split_core est appliquée au but $\neg u$ et où la décomposition disjonctive de la formule u donne les formules $d_1, \ldots, d_n$. Dans ce cas, n tâches résultantes sont produites : ce sont des tâches qui ont le même contexte que la tâche initiale mais avec $\neg d_1, \ldots, \neg d_n$ comme but à la place de $\neg u$. Par contraposition, la propriété de correction est alors $u \Rightarrow d_1 \vee \ldots \vee d_n$ et le certificat dn (d pour décomposition Disjonctive et n pour signe Négatif) reflétera cette implication. Cette dualité est rendue plus précise et les certificats correspondants seront donnés dans la suite de ce paragraphe.

La transformation split_core maintient donc deux listes contenant des formules : $conj$ et $disj$. La liste $conj$ est à comprendre comme la conjonction de ces formules et la liste $disj$ est à comprendre comme la disjonction de ces formules. En réalité ces formules sont enregistrées dans une structure de données rendant efficace la concaténation mais nous passerons sur ce détail d'implémentation dans la suite.

Application à une hypothèse. Le deuxième cas d'application de split_core consiste à scinder une hypothèse : il s'agit de la décomposer comme une conjonction. Nous appelons ce cas de figure la décomposition conjonctive d'une formule de signe négatif.

De même que pour la première façon de scinder une formule, nous avons aussi, par dualité, la décomposition disjonctive d'une formule de signe positif. Ces deux nouveaux cas de figure demandent également de connaître comment seront nommées les formules qui remplacent t.

Exemple 7.12. *Si la tâche initiale est* $t_1 : prop,\ t_2 : prop \mid \vdash P : t_1 \wedge t_2$, *alors* split_core *appliquée à la formule* P *produit les tâches* $\vdash P : t_1$ *et* $\vdash P : t_2$. *Si la tâche initiale est* $P : t_1 \wedge t_2 \vdash$, *alors* split *produit la tâche* $P_1 : t_1, P_2 : t_2 \vdash$ *pour deux noms frais* P_1 *et* P_2.

J'ai choisi de modifier les listes $conj$ et $disj$ afin qu'elles contiennent aussi des noms pour chaque formule. Comme dans l'exemple précédent, la transformation split_core fait en sorte de ne pas générer de noms lorsqu'elle est appelée sur le but. Les noms sont alors tous égaux à P. On note donc :

$$conj := [(Pc_1, conj_1); \ldots; (Pc_m, conj_m)]$$
$$disj := [(Pd_1, disj_1); \ldots; (Pd_n, disj_n)]$$

En résumé, il y a deux cas d'application de split_core : le cas où l'on souhaite décomposer une formule de signe positif comme une conjonction, et le cas où l'on souhaite décomposer une formule de signe négatif, toujours comme une conjonction. Par dualité, comme expliqué précédemment, il faut également traiter les cas de la décomposition disjonctive de formules de signe négatif et de la décomposition disjonctive de formules de signe positif. Nous avons donc instrumenté la transformation split_core afin qu'elle maintienne quatre certificats, chacun d'entre eux correspondant à l'un des quatre cas de figure : cp, cn, dp et dn. Ces certificats sont associés à des applications de transformations que nous décrivons maintenant, ce qui permet de clarifier ce que nous entendons lorsque nous disons que ces certificats sont adaptés.

Décomposition conjonctive d'une formule de signe positif. Ce cas correspond à un cas d'application de split_core pour décomposer la formule contenue dans le but. À partir de la tâche $\vdash P : t$, la transformation produit les tâches $\vdash P : conj_i$ pour i allant de 1 à m. Le certificat cp est associé à cette application, on dira donc qu'il est adapté si :

$$\vdash P : t \xleftarrow{cp} [\vdash P : conj_1;\ \ldots;\ \vdash P : conj_m]$$

Décomposition conjonctive d'une formule de signe négatif. Ce cas correspond à un cas d'application de split_core pour décomposer une formule contenue dans une hypothèse de la tâche initiale. À partir de la tâche $P : t \vdash$, la transformation produit la tâche $Pc_1 : conj_1,\ \ldots,\ Pc_m : conj_m \vdash$. Le certificat *cn* est associé à cette application, on dira donc qu'il est adapté si :

$$P : t \vdash \overset{cn}{\Longleftarrow} [Pc_1 : conj_1,\ \ldots,\ Pc_m : conj_m \vdash]$$

Décomposition disjonctive d'une formule de signe négatif. Ce cas correspond au cas dual de la décomposition conjonctive d'une formule de signe positif et apparaît lorsque la formule considérée a changé de signe. Ce n'est donc pas un cas d'application de la transformation split_core mais on peut tout de même l'associer à l'application de la transformation qui, à partir de la tâche $P : t \vdash$, produit les tâches $P : disj_i \vdash$ pour i allant de 1 à n. Le certificat *dn* est associé à cette application, on dira donc qu'il est adapté si :

$$P : t \vdash \overset{dn}{\Longleftarrow} [P : disj_1 \vdash;\ \ldots;\ P : disj_n \vdash]$$

Décomposition disjonctive d'une formule de signe positif. De même que pour le cas précédent, ce cas est dual du cas de la décomposition conjonctive d'une formule de signe négatif. Ce n'est donc pas un cas d'application de la transformation split_core mais on peut tout de même l'associer à l'application de la transformation qui, à partir de la tâche $\vdash P : t$, produit la tâche $\vdash Pd_1 : disj_1,\ \ldots,\ Pd_n : disj_n$. Le certificat *dp* est associé à cette application, on dira donc qu'il est adapté si :

$$\vdash P : t \overset{dp}{\Longleftarrow} [\vdash Pd_1 : disj_1,\ \ldots,\ Pd_n : disj_n]$$

Lorsque split_core s'applique au but, on utilise la décomposition conjonctive d'une formule de signe positif et lorsqu'elle s'applique à une hypothèse on utilise la décomposition conjonctive d'une formule de signe négatif. L'utilité de la décomposition disjonctive vient du fait que pour calculer récursivement la décomposition conjonctive, il est nécessaire de connaître la décomposition disjonctive des sous-formules qui changent de signe, ce qui est notamment le cas lorsque la formule de départ est une négation.

La syntaxe de Why3 a également deux connecteurs `by` et `so`, qui sont des connecteurs propositionnels permettant de donner des indications de coupure, celles-ci restant locales aux formules dans lesquelles ils apparaissent. Lorsque leur traitement est activé, la propriété énonçant que t implique la conjonction de sa décomposition conjonctive et la propriété énonçant que la disjonction de sa décomposition disjonctive implique t ne sont plus vraies [29]. Cela implique que les certificats *cn* et *dp* ne peuvent pas être adaptés. Nous montrons par récurrence sur t que les certificats *cp* et *dn* sont adaptés et que si le traitement des `by` et des `so` est désactivé, alors les certificats *cn* et *dp* sont adaptés. En pratique, nous utiliserons les hypothèses de récurrence sur *cp* et *dn* pour montrer que les certificats *cp* et *dn* sont adaptés, et, de manière indépendante, nous montrons que *cn* et *dp* sont adaptés à partir des hypothèses de récurrence de *cn* et *dp*. Lorsque split_core est appliquée à une hypothèse, le traitement des `by` et `so` est systématiquement désactivé. On en déduit que les deux cas d'application de split_core sont corrects. Ainsi, même si nous n'avons pas instrumenté les cas `by` et `so` de la transformation split_core (définie récursivement), il reste possible de le faire sans changer les autres cas propositionnels, et l'argument qui explique pourquoi les certificats sont adaptés (donné dans les paragraphes 7.6.1 et 7.6.2) reste inchangé. Afin de certifier également le traitement des `by` et `so`, on peut envisager d'intégrer les certificats aux manipulation de données qui sont effectuées lors du traitement de ces connecteurs. Cette

solution est intrusive et, puisque ces manipulations sont nombreuses, cela demanderait un effort de certification important, nous la réservons pour des travaux futurs.

Notons aussi que le cas de la décomposition disjonctive d'une formule de signe positif introduit une application d'une transformation qui produit une tâche résultante avec potentiellement plus d'un but. Cette transformation ne correspond donc pas à une transformation de Why3 et cela fournit un bon exemple de l'intérêt d'avoir des tâches de preuve avec plusieurs buts.

La transformation split_core distingue les cas en fonction du symbole de tête de la formule à scinder et s'appelle récursivement sur les sous-formules. Nous détaillons dans la suite le cas où le symbole de tête est soit la négation, soit la conjonction. Ces deux cas nous permettent de détailler l'ensemble des techniques utilisées pour certifier tous les cas propositionnels.

7.6.1 Cas où la formule de départ est une négation

Comme expliqué précédemment, lorsque le symbole de tête de la formule à scinder est la négation, le problème est le dual de celui sans cette négation. Dans le cas général, lorsque split_core s'applique à $\neg u$, un appel récursif sur u nous donne les listes

$$conj' := [(Pc'_1, conj'_1); \ldots; (Pc'_m, conj'_m)]$$
$$disj' := [(Pd'_1, disj'_1); \ldots; (Pd'_n, disj'_n)]$$

ainsi que les certificats cp', cn', dn' et dp' et les décompositions de $\neg u$ s'écrivent :

$$conj := [(Pd'_1, \neg disj'_1);\ \ldots;\ (Pd'_n, \neg disj'_n)]$$
$$disj := [(Pc'_1, \neg conj'_1);\ \ldots;\ (Pc'_m, \neg conj'_m)]$$

Les certificats cp, cn, dn et dp sont obtenus à partir de cp', cn', dn' et dp' et sont adaptés si ces derniers le sont aussi. On définit notamment $cp := \mathsf{Swap}(P, \mathsf{hole}) \otimes dn' \otimes \mathsf{Swap}(P, \mathsf{hole})$ et si dn' est adapté, c'est-à-dire si $P : u \vdash \overset{dn'}{\Longleftarrow} [P : disj'_1 \vdash;\ \ldots;\ P : disj'_n \vdash]$, alors cp est aussi adapté :

$$\dfrac{\dfrac{\dfrac{}{\vdash P : \neg disj'_1 \overset{\mathsf{hole}}{\Longleftarrow} [\vdash P : \neg disj'_1]}}{P : disj'_1 \vdash \overset{\mathsf{Swap}(P,\mathsf{hole})}{\Longleftarrow} [\vdash P : \neg disj'_1]} \quad \ldots \quad \dfrac{\dfrac{}{\vdash P : \neg disj'_n \overset{\mathsf{hole}}{\Longleftarrow} [\vdash P : \neg disj'_n]}}{P : disj'_n \vdash \overset{\mathsf{Swap}(P,\mathsf{hole})}{\Longleftarrow} [\vdash P : \neg disj'_n]}}{\dfrac{P : u \vdash \overset{dn' \otimes \mathsf{Swap}(P,\mathsf{hole})}{\Longleftarrow} [\vdash P : \neg disj'_1;\ \ldots;\ \vdash P : \neg disj'_n]}{\vdash P : \neg u \overset{cp}{\Longleftarrow} [\vdash P : \neg disj'_1;\ \ldots;\ \vdash P : \neg disj'_n]}}$$

Les certificats cn, dn et dp s'obtiennent de manière similaire.

7.6.2 Cas où la formule de départ est une conjonction

Supposons que la formule t soit de la forme $t^1 \wedge t^2$ et que l'on ait déjà obtenu les certificats des appels récursifs sur t^1 et t^2. On note cp^1, cn^1, dn^1 et dp^1 les certificats de l'appel sur t^1 et les décompositions sont notées

$$conj^1 := [(Pc^1_1, conj^1_1); \ldots; (Pc^1_{m^1}, conj^1_{m^1})]$$
$$disj^1 := [(Pd^1_1, disj^1_1); \ldots; (Pd^1_{n^1}, disj^1_{n^1})]$$

et on procède de manière similaire pour l'appel sur t^2. Nous supposons avoir accès aux listes $conj$ et $disj$

$$\begin{aligned} conj &:= [(Pc_1, conj_1); \ldots; (Pc_m, conj_m)] \\ disj &:= [(Pd_1, disj_1); \ldots; (Pd_n, disj_n)] \end{aligned}$$

et ces listes sont telles que l'on a, d'une part

$$conj = conj^1 \mathbin{+\!\!+} conj^2$$

et d'autre part

$$\begin{aligned} [disj_1;\ \ldots;\ disj_n] = [&disj_1^1 \wedge disj_1^2;\ \ldots;\ disj_1^1 \wedge disj_{n^2}^2] \mathbin{+\!\!+} \\ &\ldots \mathbin{+\!\!+} \\ &[disj_{n^1}^1 \wedge disj_1^2;\ \ldots;\ disj_{n^1}^1 \wedge disj_{n^2}^2] \end{aligned}$$

On note $Pd_{i,j}$ pour désigner l'identifiant Pd_k associé à la formule $disj_i^1 \wedge disj_j^2$. Nous donnons alors les certificats cp, cn, dp et dn.

Certificat cp. En posant

$$\begin{aligned} L_1 &:= [\vdash P : conj_1^1;\ \ldots;\ \vdash P : conj_{m^1}^1] \\ L_2 &:= [\vdash P : conj_1^2;\ \ldots;\ \vdash P : conj_{m^2}^2] \end{aligned}$$

on montre que cp peut être simplement donné par $\mathsf{Slice}(P, \mathsf{hole}, \mathsf{hole}) \otimes [cp^1;\ cp^2]$:

$$\frac{\vdash P : t^1 \overset{cp^1}{\Longleftarrow} L_1 \qquad \vdash P : t^2 \overset{cp^2}{\Longleftarrow} L_2}{\vdash P : t^1 \wedge t^2 \overset{cp}{\Longleftarrow} L_1 \mathbin{+\!\!+} L_2}$$

Certificat cn. Étant donné deux noms frais P_1 et P_2, on définit le certificat cn par :

$$\begin{aligned} cn = \ &\mathsf{Destruct}(P, P_1, P_2, \mathsf{hole}) \otimes \\ &\mathsf{rename}(P_1, P, \mathsf{hole}) \otimes \\ &cn^1 \otimes \\ &\mathsf{rename}(P_2, P, \mathsf{hole}) \otimes \\ &cn^2 \end{aligned}$$

Posons

$$T := Pc_1^1 : conj_1^1,\ \ldots,\ Pc_{m^1}^1 : conj_{m^1}^1;\ Pc_1^2 : conj_1^2,\ \ldots,\ Pc_{m^2}^2 : conj_{m^2}^2 \vdash$$

et montrons que cn est adapté lorsque cn^1 et cn^2 sont adaptés :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Pc_1^1 : conj_1^1,\ \ldots,\ Pc_{m^1}^1 : conj_{m^1}^1,\ Pc_1^2 : conj_1^2,\ \ldots,\ Pc_{m^2}^2 : conj_{m^2}^2 \vdash \overset{\mathsf{hole}}{\Longleftarrow} [T]}}{Pc_1^1 : conj_1^1,\ \ldots,\ Pc_{m^1}^1 : conj_{m^1}^1,\ P : t^2 \vdash \overset{cn^2 \otimes \_}{\Longleftarrow} [T]}}{Pc_1^1 : conj_1^1,\ \ldots,\ Pc_{m^1}^1 : conj_{m^1}^1,\ P_2 : t^2 \vdash \overset{\mathsf{rename}(P_2,P,\mathsf{hole}) \otimes \_}{\Longleftarrow} [T]}}{P : t^1,\ P_2 : t^2 \vdash \overset{cn^1 \otimes \_}{\Longleftarrow} [T]}}{P_1 : t^1,\ P_2 : t^2 \vdash \overset{\mathsf{rename}(P_1,P,\mathsf{hole}) \otimes \_}{\Longleftarrow} [T]}}{P : t^1 \wedge t^2 \vdash \overset{cn}{\Longleftarrow} [T]}$$

Certificat dp**.** Soit

$$T := \vdash Pd_{1,1} : disj_1^1 \wedge disj_1^2;\ \ldots;\ Pd_{n^1,n^2} : disj_{n^1}^1 \wedge disj_{n^2}^2$$

la tâche résultante. Le certificat dp commence par $\mathsf{Slice}(P, \mathsf{hole}, \mathsf{hole})$ ce qui produit deux tâches $T_1 := \vdash P : t^1$ et $P_2 := \vdash P : t^2$. Nous montrons qu'il existe deux certificats à un seul argument C_1 et C_2 tels que $T_1 \overset{C_1}{\Longleftarrow} [T]$ et $T_2 \overset{C_2}{\Longleftarrow} [T]$. Par symétrie nous cherchons seulement à donner C_1 et nous concluons en construisant le certificat

$$dp := \Lambda t.\ \mathsf{Slice}(P, \mathsf{hole}, \mathsf{hole}) \otimes [C_1(t); C_2(t)]$$

Le certificat C_1 est obtenu en appliquant dp^1, ce qui nous donne la tâche

$$T' := \vdash Pd_1^1 : disj_1^1,\ \ldots,\ Pd_{n^1}^1 : disj_{n^1}^1$$

À partir d'une tâche ayant pour buts les déclarations de Δ qui contiennent au moins toutes les déclarations de T', on peut ajouter n'importe quel but de la forme $Pd_{i,j} : disj_i^1 \wedge disj_j^2$, ce qui nous donne une tâche T'', grâce au certificat

$$c'' := \mathsf{Assert}(Pd_{i,j}, disj_i^1 \wedge disj_j^2, \mathsf{hole}, \mathsf{Destruct}(Pd_{i,j}, P', P'', \mathsf{Axiom}(Pd'_v, Pd_i^1)))$$

où P' et P'' sont deux noms frais. En effet, on a la dérivation

$$\dfrac{\dfrac{}{T'' \overset{\mathsf{hole}}{\Longleftarrow} [T'']} \qquad \dfrac{\dfrac{}{P' : disj_i^1, P'' : disj_j^2 \vdash \Delta \overset{\mathsf{Axiom}(P', Pd_i^1)}{\Longleftarrow} []}}{Pd_{i,j} : disj_i^1 \wedge disj_j^2 \vdash \Delta \overset{\mathsf{Destruct}(Pd_{i,j}, P', P'', \_)}{\Longleftarrow} []}}{\vdash \Delta \overset{c''}{\Longleftarrow} [T'']}$$

ce qui nous permet d'obtenir la tâche contenant toutes les déclarations de T et toutes les déclarations de T'. Il suffit ensuite de composer le certificat obtenu avec

$$\mathsf{Clear}(Pd_{n^1}^1, \mathsf{hole}) \otimes \ldots \otimes \mathsf{Clear}(Pd_{n^1}^1, \mathsf{hole})$$

afin d'obtenir C_1. Le certificat dp est donc adapté lorsque dp^1 et dp^2 sont adaptés.

Certificat dn**.** Notons, pour i allant de 1 à n^1 et j allant de 1 à n^2 :

$$T_{i,j} := P : disj_i^1 \wedge disj_j^2 \vdash$$

Étant donné deux noms frais P_1 et P_2, on pose :

$$\begin{aligned} dn := \ & \mathsf{Destruct}(P, P_1, P_2, \mathsf{hole}) \otimes \\ & \mathsf{rename}(P_1, P, \mathsf{hole}) \otimes \\ & dn^1 \otimes \\ & \mathsf{rename}(P, P_1, \mathsf{hole}) \otimes \\ & \mathsf{rename}(P_2, P, \mathsf{hole}) \otimes \\ & dn^2 \otimes \\ & \mathsf{rename}(P, P_2, \mathsf{hole}) \otimes \\ & \mathsf{construct}(P_1, P_2, P, \mathsf{hole}) \otimes \\ & \mathsf{Clear}(P_1, \mathsf{Clear}(P_2, \mathsf{hole})) \end{aligned}$$

et on montre que, si dn' est adapté, alors dn adapté, c'est-à-dire que l'on a $P : t^1 \wedge t^2 \vdash$:

$$\dfrac{\dfrac{\dfrac{D_1^1 \quad \dots \quad D_{n^1}^1}{P : t^1,\ P_2 : t^2 \vdash \overset{dn^1 \otimes \_}{\Longleftarrow} [T_{1,1};\ \dots;\ T_{1,n^2};\ \dots;\ T_{n^1,1};\ \dots;\ T_{1,n^2}]}}{P_1 : t^1,\ P_2 : t^2 \vdash \overset{\mathsf{rename}(P_1,P,\mathsf{hole}) \otimes \_}{\Longleftarrow} [T_{1,1};\ \dots;\ T_{1,n^2};\ \dots;\ T_{n^1,1};\ \dots;\ T_{1,n^2}]}}{P : t^1 \wedge t^2 \vdash \overset{dn}{\Longleftarrow} [T_{1,1};\ \dots;\ T_{1,n^2};\ \dots;\ T_{n^1,1};\ \dots;\ T_{1,n^2}]}$$

où il s'agit, pour i allant de 1 à n^1, de donner la dérivation D_i s'écrivant :

$$\dfrac{\dfrac{\dfrac{D_{i,1} \quad \dots \quad D_{i,n^2}}{P_1 : disj_i^1,\ P : t^2 \vdash \overset{dn^2 \otimes \_}{\Longleftarrow} [T_{i,1};\ \dots;\ T_{i,n^2}]}}{P_1 : disj_i^1,\ P_2 : t^2 \vdash \overset{\mathsf{rename}(P_2,P,\mathsf{hole}) \otimes \_}{\Longleftarrow} [T_{i,1};\ \dots;\ T_{i,n^2}]}}{P : disj_i^1,\ P_2 : t^2 \vdash \overset{\mathsf{rename}(P,P_1,\mathsf{hole}) \otimes \_}{\Longleftarrow} [T_{i,1};\ \dots;\ T_{i,n^2}]}$$

où, nous devons donner les dérivations $D_{i,j}$ pour j allant de 1 à n^2. La dérivation $D_{i,j}$ s'écrit :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{P : disj_i^1 \wedge disj_j^2 \vdash \overset{\mathsf{hole}}{\Longleftarrow} [T_{i,j}]}}{P_2 : disj_j^2, P : disj_i^1 \wedge disj_j^2 \vdash \overset{\mathsf{Clear}(P_2,\mathsf{hole}) \otimes \_}{\Longleftarrow} [T_{i,j}]}}{P_1 : disj_i^1,\ P_2 : disj_j^2, P : disj_i^1 \wedge disj_j^2 \vdash \overset{\mathsf{Clear}(P_1,\mathsf{hole}) \otimes \_}{\Longleftarrow} [T_{i,j}]}}{P_1 : disj_i^1,\ P_2 : disj_j^2 \vdash \overset{\mathsf{construct}(P_1,P_2,P,\mathsf{hole}) \otimes \_}{\Longleftarrow} [T_{i,j}]}}{P_1 : disj_i^1,\ P : disj_j^2 \vdash \overset{\mathsf{rename}(P,P_2,\mathsf{hole}) \otimes \_}{\Longleftarrow} [T_{i,j}]}$$

Conclusion. La transformation split a donc été rendue certifiante et renvoie des certificats adaptés pour la plupart des constructions de Why3 à l'exception de la quantification, des `by` et `so`, et du *pattern-matching*. La certification de cette transformation a été grandement facilitée par l'utilisation des certificats de surface, notamment parce qu'ils sont modulaires et abstraits des détails des tâches de preuve.

7.7 Élimination du polymorphisme

Une approche possible de l'élimination du polymorphisme est de chercher, pour une formule polymorphe f donnée, un ensemble fini d'instances monomorphes de f tel que la validité de toutes ses formules est équivalente à la validité de f. Comme l'ont fait remarquer Bobot et Paskevich [20], il n'existe pas d'algorithme renvoyant un tel ensemble dans le cas général. Une approche alternative est de chercher à encoder les informations de typage dans les termes. Des extensions de cette deuxième approche, qui ont été implémentées en Why3 [20], s'attachent à limiter l'impact de l'encodage des informations de typage en terme de performances des prouveurs automatiques lorsqu'ils sont appelés sur les tâches produites. Une de ces extensions est présentée sous la forme d'une transformation discriminate introduisant des instances monomorphes de formules polymorphes. L'intérêt de cette transformation, appelée en amont de l'encodage des informations de typage, est de fournir des formules non encodées qui seront alors traduites de façon plus naturelle dans la logique des prouveurs automatiques. La transformation discriminate conserve les formules polymorphes initiales et cela nous assure la complétude de la méthode.

Nous présentons ici comment nous envisageons de rendre certifiante la transformation discriminate de Why3. Cette transformation se décompose en trois transformations appliquées les unes après les autres. Nous donnons, dans les paragraphes qui suivent, la méthode que nous prévoyons d'appliquer afin de rendre certifiante chacune de ces transformations. Nous prenons pour exemple le cas où la tâche initiale est définie par $I \mid \Sigma \mid \Gamma \vdash \Delta$, où :

$$
\begin{aligned}
I &:= \mathit{color} : 0,\ \mathit{list} : 1 \\
\Sigma &:= \mathit{red} : \mathit{color},\ \mathit{cons} : \alpha \rightsquigarrow \mathit{list}(\alpha) \rightsquigarrow \mathit{list}(\alpha) \\
&\quad\ \mathit{rev\_append} : \mathit{list}(\alpha) \rightsquigarrow \mathit{list}(\alpha) \rightsquigarrow \mathit{list}(\alpha) \\
\Gamma &:= H : \Pi\alpha.\ \forall x : \alpha.\ \forall l_1\ l_2 : \mathit{list}(\alpha). \\
&\qquad \mathit{rev\_append}\ (\mathit{cons}\ x\ l_1)\ l_2 = \mathit{rev\_append}\ l_1\ (\mathit{cons}\ x\ l_2) \\
\Delta &:= G : \Pi\alpha.\ \forall x\ y : \alpha.\ \forall l_1\ l_2 : \mathit{list}(\alpha). \\
&\qquad \mathit{rev\_append}\ (\mathit{cons}\ x\ (\mathit{cons}\ y\ l_1))\ l_2 = \\
&\qquad \mathit{rev\_append}\ l_1\ (\mathit{cons}\ y\ (\mathit{cons}\ x\ l_2))
\end{aligned}
$$

7.7.1 Élimination du polymorphisme du but

Le but de la tâche initiale peut être polymorphe et la première transformation introduit alors la variable de type. Le certificat IntroType permet d'instrumenter cette transformation.

Sur l'exemple, la variable α est introduite : un symbole de type frais ι est ajouté à la signature de type par $\iota : 0$. Le but devient

$$
\begin{aligned}
G : {} & \forall x\ y : \iota.\ \forall l_1\ l_2 : \mathit{list}(\iota). \\
& \mathit{rev\_append}\ (\mathit{cons}\ x\ (\mathit{cons}\ y\ l_1))\ l_2 = \\
& \mathit{rev\_append}\ l_1\ (\mathit{cons}\ y\ (\mathit{cons}\ x\ l_2))
\end{aligned}
$$

et le certificat $\mathsf{IntroType}(G, \iota, \mathsf{hole})$ est adapté à cette application.

7.7.2 Instanciation des déclarations polymorphes

Ensuite, la deuxième transformation cherche des instanciations pertinentes des hypothèses [20]. Sur l'exemple, seule l'hypothèse H est instanciée par ι, nous donnons alors le certificat $\mathsf{InstType}(H, H', \iota, \mathsf{hole})$. Bien que cette deuxième transformation soit complexe, la certification semble relativement aisée dans ce cas de figure. En effet, la complexité réside principalement dans la recherche des instanciations mais cette recherche n'a pas à être certifiée.

7.7.3 Introduction et réécriture des symboles monomorphes

La troisième transformation introduit un symbole monomorphe f_τ pour chaque instance monomorphe de type τ d'un symbole polymorphe f. Les applications de f de type τ sont alors remplacées par f_τ.

Sur cette nouvelle tâche, le symbole polymorphe rev_append a une application monomorphe de type $\tau := list(\iota) \rightsquigarrow list(\iota) \rightsquigarrow list(\iota)$. La transformation introduit donc un symbole rev_append_τ ayant le type τ et remplace toute occurrence de rev_append de type τ par rev_append_τ.

Nous obtenons finalement la tâche $I' \mid \Sigma' \mid \Gamma' \vdash \Delta'$ définie par :

$$
\begin{aligned}
I' &:= color : 0,\ list : 1, \iota : 0 \\
\Sigma' &:= red : color,\ cons : \alpha \rightsquigarrow list(\alpha) \rightsquigarrow list(\alpha) \\
&\quad\ rev\_append : list(\alpha) \rightsquigarrow list(\alpha) \rightsquigarrow list(\alpha) \\
&\quad\ rev\_append\_\tau : list(\tau) \rightsquigarrow list(\tau) \rightsquigarrow list(\tau) \\
\Gamma' &:= H : \Pi\alpha.\ \forall x : \alpha.\ \forall l_1\ l_2 : list(\alpha). \\
&\qquad rev\_append\ (cons\ x\ l_1)\ l_2 = rev\_append\ l_1\ (cons\ x\ l_2) \\
&\quad\ H' :\ \forall x : \iota.\ \forall l_1\ l_2 : list(\iota). \\
&\qquad rev\_append\_\tau\ (cons\ x\ l_1)\ l_2 = rev\_append\_\tau\ l_1\ (cons\ x\ l_2) \\
\Delta' &:= G :\ \forall x\ y : \iota.\ \forall l_1\ l_2 : list(\iota). \\
&\qquad rev\_append\_\tau\ (cons\ x\ (cons\ y\ l_1))\ l_2 = \\
&\qquad rev\_append\_\tau\ l_1\ (cons\ y\ (cons\ x\ l_2))
\end{aligned}
$$

Un certificat pour cette transformation s'appuie sur Rewrite. Sur l'exemple, ce certificat doit pouvoir réécrire rev_append en rev_append_τ et doit pouvoir le faire sous les quantificateurs. Avec cet objectif en tête, nous faisons une coupure (certificat Assert) sur la formule $\exists ra : \tau.\ ra = (rev\_append)_\tau$. Cette formule peut être prouvée par instanciation (certificat InstQuant) puis par réflexivité de l'égalité. Pour le deuxième certificat à fournir à Assert, le constructeur IntroQuant permet d'introduire le symbole rev_append_τ et d'obtenir l'égalité à réécrire en hypothèse. Nous n'oublions pas, ensuite, de supprimer l'hypothèse introduite par la coupure.

L'implémentation de la version certifiante de la transformation discriminate est en cours.

7.8 Implémentation et expérimentation

7.8.1 Validation de notre méthode

J'ai instrumenté une quinzaine de nouvelles transformations de Why3 pour générer des certificats et les vérifier à la volée lors de chaque utilisation. Cela m'a permis de tester chaque

nombre de variables	5	10	15	20	25	50	100	200	400	800
temps de la transformation (s)	~ 0	~ 0	0.01	0.01	0.02	0.06	0.29	1.22	5.4	25
temps de la transformation certifiante (s)	~ 0	~ 0	0.01	0.01	0.02	0.08	0.32	1.34	5.9	28
taille des certificats de surface (kB)	4	4	8	12	16	32	64	128	260	516
taille des certificats de noyau (kB)	8	16	28	36	52	124	344	1200	4000	15000
temps du vérificateur OCaml (s)	~ 0	~ 0	~ 0	~ 0	~ 0	~ 0	0.02	0.07	0.28	1.1
temps du vérificateur Lambdapi (s)	0.07	0.21	0.64	1.6	3.2	35	450	-	-	-

TABLE 7.1 – Tests sur une famille de problèmes propositionnels

constructeur de certificat en m'assurant qu'un certificat pouvait être généré et vérifié pour au moins une application d'une transformation.

Nous testons également que les certificats générés par nos transformations couvrent un grand nombre de cas. Ces tests portent sur les transformations split, induction et rewrite. Par exemple, pour cette dernière, un test consiste à se placer dans la théorie des groupes de la bibliothèque standard de Why3 pour réécrire des axiomes tels que l'associativité de la loi de composition interne.

Nous avons aussi testé l'efficacité de la vérification de certificats : taille des certificats de surface et des certificats de noyau et temps de vérification du certificat produit. Nous nous intéressons au temps pris par une transformation et au temps pris par la version certifiante de cette transformation, ce qui nous permet d'estimer le surcoût de temps dû à la génération du certificat. Le temps de vérification comprend également le temps nécessaire à l'élaboration du certificat ainsi que le temps pris par la traduction des tâches de Why3 en des tâches de notre formalisme. Nous utilisons la transformation blast afin de fournir des certificats de taille croissante en changeant la tâche de preuve donnée en entrée. L'énoncé de ce problème pour n variables propositionnelles est donné par

$$p_1 \Rightarrow (p_1 \Rightarrow p_2) \Rightarrow \ldots (p_{n-1} \Rightarrow p_n) \Rightarrow p_n$$

Les résultats de ces tests sont donnés dans le tableau 7.1. Nous concluons que le temps de génération des certificats reste négligeable en comparaison avec celui pris par la transformation elle-même. Nous tirons la même conclusion du vérificateur OCaml, ce qui nous permet de penser que notre méthode ne semble pas avoir de problème inhérent d'efficacité. Le temps de calcul pour la vérification des certificats par Lambdapi augmente très rapidement avec la taille du problème. Signalons que lors des premières expérimentations le temps de calcul augmentait tellement vite que nous n'arrivions pas à dépasser 15 variables. Ces problèmes d'efficacité ont fait l'objet de rapport de *bug* aux auteurs de Lambdapi et des améliorations successives ont été réalisées [49–53]. Le tableau 7.1 donne les temps de calcul pour une version améliorée de Lambdapi, avec laquelle on arrive à atteindre une centaine de variables. Le temps d'élaboration, compris dans le temps pris par le vérificateur OCaml, reste très faible.

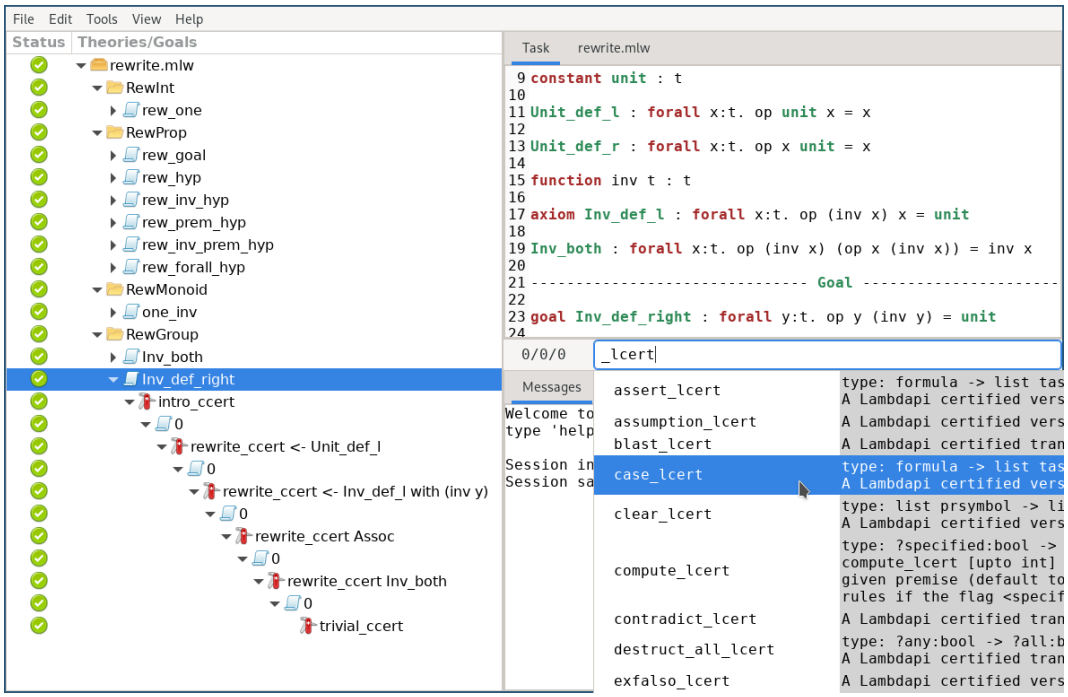

FIGURE 7.2 – Transformations certifiantes dans Why3

Cela motive le fait de calculer les types pendant l'élaboration pour pouvoir les expliciter à Lambdapi. Pour les problèmes testés, la taille des certificats de surface est linéaire par rapport à la taille du problème d'entrée. La taille des certificats de noyau n'est plus linéaire mais reste polynomiale : cela est dû au fait que des formules sont ajoutées à chaque nœud de certificat. Nous pensons que ces résultats peuvent encore être améliorés, à la fois au niveau de la taille des certificats et au niveau du temps pris par le vérificateur Lambdapi, par exemple en factorisant les formules de grande taille dans le certificat de noyau.

Le vérificateur OCaml est disponible pour toutes les transformations déjà certifiantes à l'exception, à ce jour, de la transformation compute. Il en est de même pour le mécanisme de vérification via Lambdapi. Nos expérimentations ont permis de montrer que les certificats générés par nos transformations certifiantes sont validés aussi bien par le vérificateur OCaml que par le vérificateur Lambdapi.

7.8.2 Implémentation

Les détails pour compiler notre travail et pouvoir ainsi tester les transformations certifiantes sont donnés dans le fichier `README_CERT.md` à la racine du dépôt suivant

`https://gitlab.inria.fr/why3/why3/tree/cert`

Ce fichier donne également les versions des logiciels utilisés et les conditions d'expérimentation. Nous détaillons ici la structure générale de notre projet. Nous utilisons des fichiers différents afin de séparer les fonctionnalités suivantes :

— la définition de nos structures de données telles que celle des termes, des types ou celle de nos tâches de preuve,
— la traduction des tâches de preuve de Why3 en nos tâches de preuve,

— la définition des certificats de surface, des certificats de noyau ainsi que de l'élaboration de certificats,
— les fonctionnalités facilitant le développement de transformations certifiantes, telles que la composition de ces dernières,
— l'instrumentation de transformations de Why3 ainsi que la définition de nouvelles transformations certifiantes,
— la définition du vérificateur OCaml,
— la définition du vérificateur Lambdapi,
— l'intégration des transformations certifiantes comme des transformations de Why3 dont le résultat a été vérifié, comme décrit dans le paragraphe 5.2,
— le test des transformations certifiantes ayant pour but de nous assurer que les certificats qu'elles renvoient sont adaptés dans différents cas.

Les nouvelles transformations obtenues sont accessibles depuis l'interface graphique de Why3 et suivent les conventions suivantes : les noms des transformations vérifiées par OCaml finissent par `_ccert` et ceux des transformations vérifiées par Lambdapi finissent par `_lcert`. Par exemple, on peut obtenir la liste des transformations vérifiées par Lambdapi en tapant `_lcert` (voir la figure 7.2).

Chapitre 8

Conclusion

Ce manuscrit a permis de présenter un cadre pour vérifier des transformations logiques. Même si notre méthode est générale, le formalisme dans lequel nous nous plaçons s'applique naturellement à la preuve de programmes. En effet, les programmes manipulent des structures de données prédéfinies, et la spécification et la preuve de ces programmes s'appuient sur les théories correspondantes comme la théorie des entiers. On utilise aussi naturellement une logique d'ordre supérieur et des types de données polymorphes.

Nous suivons l'approche sceptique : nous instrumentons les transformations pour qu'elles produisent des certificats et vérifions, à chaque application de l'une d'entre elles, le certificat ainsi généré. Cette approche est *incrémentale* : cela nous permet de certifier petit à petit une transformation donnée en augmentant le nombre de cas d'applications qui renvoient un certificat adapté. L'approche par certificat permet aussi une vérification *robuste* aux changements et mises à jour du code vérifié : les certificats n'ont pas à être modifiés tant que le comportement de la transformation reste inchangé.

Notre méthode s'articule autour de deux axes qui ont l'air antinomiques : d'un côté la vérification des certificats, résumée dans le paragraphe 8.1, et de l'autre la production des certificats, résumée dans le paragraphe 8.2. S'il est plus difficile de produire des certificats détaillés et spécifiques à une application d'une transformation, ces certificats sont plus simples à vérifier. Dans la suite de ce chapitre, nous résumons comment nous avons résolu ce dilemme, nous comparons les résultats obtenus avec des travaux connexes (paragraphe 8.3) et nous explicitons des travaux en perspective (paragraphe 8.4).

8.1 Vérification de certificats

Les certificats de noyau, présentés dans le chapitre 3, sont les certificats destinés à la vérification des applications des transformations. Ces certificats sont appropriés pour un tel objectif, d'abord parce qu'ils sont *précis*, ce qui permet de savoir sans mal le contexte de preuve de la vérification de ces certificats, et aussi parce que ces certificats sont *élémentaires*, ce qui permet plus facilement de se convaincre de leur correction. Les certificats bénéficient aussi d'une certaine *expressivité* attestée par l'ensemble des transformations que nous avons pu rendre certifiantes.

Nous avons défini deux vérificateurs de certificats. Le premier utilise une approche calculatoire. Cette approche a été reconnue par le passé pour être particulièrement efficace, par exemple, Grégoire *et al.* l'ont utilisé pour vérifier la primalité de nombres premiers [58], et Armand *et al.* pour la vérification de preuves d'insatisfaisabilité de formules SAT et SMT [3].

Le second repose sur un encodage superficiel à l'intérieur du prouveur interactif Lambdapi. Lorsque l'on définit un encodage superficiel, la correction de la méthode repose sur cet encodage et sur la correction du système cible. Par exemple, Contejean [31] génère un script de preuve Coq pour justifier la terminaison d'un système de réécriture, et Cauderlier *et al.* [26] génèrent un terme Dedukti à partir d'une trace de preuve produite par le prouveur Zenon Modulo.

8.2 Génération de certificats

Ce travail se fonde sur une notion de certificats à trous, qui s'inspire de notions analogues présentes dans le prouveur interactif Coq telles que les méta-variables ou les variables existentielles. Dans le contexte de l'approche sceptique, l'utilisation de certificats à trous est nouvelle à ma connaissance. Elle permet de refléter, au sein des certificats, la *modularité* et la possibilité de composer les transformations :

1. les certificats eux-mêmes sont composables, grâce à la possibilité de laisser des trous qui seront comblés par la suite
2. la vérification de ces certificats à trous nécessite de comparer les buts ouverts des certificats avec ceux produits par la transformation elle-même.

Outre la flexibilité dans l'écriture des transformations apportée par l'approche sceptique, cela permet à la fois d'instrumenter les transformations de manière totalement indépendante les unes des autres et de s'intéresser à des transformations plus élémentaires avant de les combiner. Les certificats de surface sont plus faciles à produire et sont abstraits des détails des tâches de preuve, ils permettent ainsi un développement modulaire de la génération de certificats.

L'élaboration de certificats permet d'instrumenter les transformations avec des certificats de surface, plus simples à produire, à la place des certificats de noyau. L'accent est donc mis sur la *facilité d'utilisation* de la méthode. L'élaboration est située en amont de la vérification, et ne fait donc pas partie de la base de confiance de notre approche.

Nous avons également étendu notre approche pour supporter des théories interprétées, dont les théories de l'arithmétique et de l'égalité. Cela nous a permis d'instrumenter, a posteriori, des transformations complexes telles que l'induction ou la réécriture.

8.3 Comparaison avec des travaux connexes

Dans ce paragraphe, nous présentons différents travaux permettant d'accroître la confiance accordée aux prouveurs et nous les comparons avec notre méthode.

8.3.1 Identification du noyau d'un prouveur et approche autarcique

L'identification du noyau d'un prouveur facilite la vérification de l'outil complet : la vérification est alors réduite à celle du noyau. Pour des raisons similaires, le format des certificats de noyau a été choisi de façon à être aussi minimal que possible, ce qui a été bénéfique afin de définir nos vérificateurs de certificats et de nous assurer de leur correction. Ce principe a été appliqué par Harrison afin de concentrer ses efforts sur 400 lignes de code OCaml [60], permettant ainsi de renforcer la confiance accordée au prouveur interactif HOL Light.

L'approche autarcique est particulièrement indiquée lorsqu'un noyau est identifié : en plus d'avoir un code réduit à vérifier, celui-ci représente une partie centrale de l'outil et

est donc peu susceptible de changer. Par exemple, Lescuyer a adopté l'approche autarcique afin de vérifier, en Coq, le noyau d'une version du prouveur automatique Alt-Ergo [68]. La difficulté de l'approche autarcique et sa forte dépendance au code vérifié motivent notre approche par certificats.

8.3.2 Définition de certificats dans les prouveurs automatiques

De nombreux prouveurs automatiques ont été instrumentés pour produire des certificats, un résumé des avancements dans ce domaine a été proposé par Barrett et al. [8]. Parmi eux, on trouve des prouveurs automatiques SAT, SMT ou d'autres : veriT [25], Vampire [64], CVC4 [7], zChaff [83], etc. Que ce soit le format de sortie de zChaff donnant une preuve par résolution, que ce soit le format LFSC [71], que ce soit le format de certificat TSTP [77] ou que ce soit le format Alethe [74], les certificats représentent des preuves dans la logique du premier ordre ou un fragment de la logique du premier ordre.

Nous souhaitons vérifier des transformations complexes et nous ne présupposons pas que les certificats sont générés par un prouveur automatique. Nous définissons donc un autre format de certificats, celui-ci s'appuyant sur la logique d'ordre supérieur. Contrairement aux travaux cités ici, notre approche a pour objectif de vérifier des preuves partielles dans le sens où certains buts sont laissés ouverts afin de représenter les tâches de preuve résultantes. Ces différences dans le format des certificats se retrouvent au niveau des travaux connexes traitant de la vérification des certificats.

8.3.3 Vérification de certificats

La vérification de certificats permet de renforcer la confiance accordée aux prouveurs automatiques. Un autre intérêt de la vérification de certificats est de pouvoir faire appel aux prouveurs automatiques de manière sceptique à l'intérieur de prouveurs interactifs, permettant ainsi d'améliorer l'automatisation de ces derniers. Cette deuxième approche est celle de CoqHammer [36] pour Coq et de SledgeHammer [12] pour Isabelle. Ces deux outils font appel à des prouveurs automatiques et utilisent des informations extraites des certificats, telles que l'ensemble des axiomes utilisés, afin de reconstruire une preuve dans le prouveur interactif en question. Le greffon SMTCoq [43] est également un outil pour Coq mais dont le fonctionnement est plus proche du nôtre car les certificats sont vus comme des preuves complètes plutôt que comme des indications de preuve. En effet, pour une propriété à prouver donnée et un certificat obtenu par l'appel d'un prouveur automatique, SMTCoq utilise un encodage profond du certificat comme une preuve de la propriété. Notre vérificateur Lambdapi, en revanche, utilise un encodage superficiel, à la fois des tâches de preuve comme des propriétés Lambdapi et des certificats comme des preuves de ces propriétés.

Les certificats offrent un cadre commun pour la vérification à travers le format choisi, c'est notamment l'objectif poursuivi par ProofCert [28]. Ce projet propose un format de certificats et une vérification de ceux-ci développés en λProlog. L'élaboration de certificats décrite dans le paragraphe 6.4 est une étape de traitement des certificats similaire à l'approche de ProofCert, si ce n'est que notre format de certificats a moins vocation à être universel : notre logique est fixée à la logique classique d'ordre supérieur. Nous souhaitons instrumenter progressivement des transformations, produisant des tâches résultantes, et cela justifie notre définition de certificats à trou. Nous nous différencions aussi de ProofCert par le fait que nous voulons mettre l'accent sur la facilité de production des certificats, et que ceux-ci sont exprimés dans une logique d'ordre supérieur. Citons aussi Ekstrakto [59] qui a pour objectif d'uniformiser, dans Lambdapi, la vérification de preuves renvoyées par les prouveurs automatiques en se fondant sur le format de certificat TSTP.

8.3.4 Combiner la production et la vérification de certificats

Lorsque l'on utilise l'approche sceptique, le code d'une transformation ou d'un prouveur est instrumenté pour produire des certificats et la vérification peut être effectuée dans un deuxième temps. De ce fait, il est possible et fréquent que la vérification soit faite par un outil externe, indépendamment de l'outil certifié. Dans notre cas, les certificats sont générés puis vérifiés au moment de l'application d'une transformation ; le format des certificats de noyau est vu comme une interface entre les différents vérificateurs, nous permettant ainsi de définir des vérificateurs indépendants. Une autre approche, inspirée de l'approche LCF [55] ou approche défensive [24, 76], aurait consisté en l'utilisation d'un noyau sûr qui représente le seul moyen de générer des transformations (ou étapes de transformations) certifiantes. Une approche défensive aurait donc pu se passer complètement des certificats et améliorer ainsi l'efficacité de la vérification, mais aurait contraint la vérification à une implémentation particulière.

Enfin, la définition d'une nouvelle tactique en Coq peut s'apparenter à la définition d'une nouvelle transformation certifiante : dans les deux cas, le but courant est remplacé par de nouveaux buts et une preuve de cette correction est produite. En ce qui concerne Coq, il y a une vérification finale qui s'assure que l'ensemble des pas de preuve générés par les tactiques est correct, alors que nos transformations sont vérifiées indépendamment les unes des autres. Notons aussi que les tactiques fournissent nécessairement un terme de preuve mais que les transformations peuvent être instrumentées ou non, et que cela permet d'instrumenter petit à petit des transformations existantes. Des travaux futurs pourraient d'ailleurs consister en la traduction des transformations certifiantes en des tactiques, permettant ainsi de les utiliser à l'intérieur de prouveurs interactifs tels que Coq.

8.4 Perspectives

Un objectif à court terme est de poursuivre l'application à Why3, afin de compléter la certification de la transformation split (paragraphe 7.6) et d'implémenter la certification de la transformation discriminate (paragraphe 7.7). Nous pouvons aussi nous intéresser à d'autres transformations couramment utilisées dans les développements Why3. Certaines de ces transformations peuvent nécessiter d'étendre (a) le format des tâches utilisé pour la validation, (b) le format de certificat et (c) nos vérificateurs. Par exemple, l'élimination des types algébriques représente un défi important, à commencer par l'ajout des types algébriques dans notre formalisme logique. Un autre défi est la certification complète des transformations d'élimination du polymorphisme telles qu'implémentées dans Why3 par Bobot et Paskevich [20]. On peut aussi envisager de définir de nouvelles théories interprétées telle qu'une théorie des réels qui ferait écho à celle intégrée à Why3.

Un passage à l'échelle va nécessairement poser des questions d'efficacité, soit par exemple dans l'élaboration, soit dans la vérification par Lambdapi. D'un point de vue programmation, il s'agirait de se rapprocher de l'implémentation concrète des tâches, notamment en étudiant la possibilité de conserver la mémoïsation. D'un point de vue plus théorique, la démarche présentée génère des certificats à petits grains pouvant donc rapidement devenir volumineux lorsque plusieurs étapes sont combinées. Il conviendrait d'étudier la compression de ces certificats, et notamment comment elle pourrait être menée à la volée lors de la combinaison de certificats.

La méthode utilisée n'est pas propre à Why3 et peut s'appliquer à la certification d'encodages logiques en toute généralité. Nous pouvons notamment envisager de l'utiliser pour certifier "à petits pas" des encodages de logiques d'assistants de preuve vers des prouveurs

automatiques, afin de pouvoir combiner ces deux types de prouveurs. Afin d'avoir une pleine confiance dans l'outil Why3, un objectif à plus long terme est de certifier de bout en bout la chaîne de Why3. En aval, il s'agirait de lier ce travail avec la vérification des preuves effectuées par les prouveurs automatiques, déjà proposée par Armand *et al.* dans SMTCoq [3] ou par Böhme et Weber [23] pour valider les certificats produits par le prouveur Z3. En amont, il s'agirait de proposer une méthode de certification de la génération des obligations de preuve. Pour aborder ce dernier point, on peut envisager une traduction monadique comme celle utilisée par Filliâtre [44] qui génère à la fois un terme Coq et son type attendu pour une fonction donnée. Le type attendu est une traduction, à l'aide de types dépendants, de la spécification de la fonction. Le terme Coq proposé combine le corps de la fonction et les termes de preuve des obligations de preuve générées auparavant, pour former un habitant du type attendu.

Bibliographie

[1] Coq Library : Binary Integers. `https://coq.inria.fr/library/Coq.Numbers.BinNums.html`.

[2] Alejandro Aguirre. Towards a provably correct encoding from F* to SMT. Master's thesis, Université Paris 7, 2016.

[3] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Thery, and Benjamin Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In Jean-Pierre Jouannaud and Zhong Shao, editors, *First International Conference on Certified Programs and Proofs*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150, Kenting, Taïwan, Province De Chine, December 2011. Springer. URL : `http://hal.inria.fr/hal-00639130`, `doi:10.1007/978-3-642-25379-9\_12`.

[4] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Expressing theories in the λΠ-calculus modulo theory and in the Dedukti system. In *22nd International Conference on Types for Proofs and Programs*, 2016. URL : `https://hal-mines-paristech.archives-ouvertes.fr/hal-01441751`.

[5] Henk Barendregt. Typed lambda calculi. In Abramsky et al., editor, *Handbook of Logic in Computer Science*. Oxford Univ. Press, 1993.

[6] Henk Barendregt and Erik Barendsen. Autarkic computations in formal proofs. *Journal of Automated Reasoning*, 2002. `doi:10.1023/A:1015761529444`.

[7] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11. Springer-Verlag, 2011. `http://cvc4.cs.stanford.edu/web/`.

[8] Clark Barrett, Leonardo De Moura, and Pascal Fontaine. Proofs in satisfiability modulo theories. *All about proofs, Proofs for all*, 55(1) :23–44, 2015.

[9] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard : Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, volume 13, page 14, 2010.

[10] Christoph Benzmüller, Chad E Brown, and Michael Kohlhase. Higher-order semantics and extensionality. *The Journal of Symbolic Logic*, 69(4) :1027–1088, 2004.

[11] Christoph Benzmüller, Florian Rabe, and Geoff Sutcliffe. THF0–the core of the TPTP language for higher-order logic. In *International Joint Conference on Automated Reasoning*, pages 491–506. Springer, 2008.

[12] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT solvers. In *Proceedings of the 23rd international conference*

on Automated deduction, volume 6803 of *Lecture Notes in Computer Science*, pages 116–130, Berlin, Heidelberg, 2011. Springer. URL : http://dl.acm.org/citation.cfm?id=2032266.2032277.

[13] Jasmin Christian Blanchette and Andrei Paskevich. TFF1 : The TPTP typed first-order form with rank-1 polymorphism. In *International Conference on Automated Deduction*, pages 414–420. Springer, 2013.

[14] Frédéric Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical structures in computer science*, 15(1) :37–92, 2005.

[15] Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. The calculus of algebraic constructions. In *International Conference on Rewriting Techniques and Applications*, pages 301–316. Springer, 1999.

[16] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *International Symposium on Formal Methods*, pages 460–475. Springer, 2006.

[17] Valentin Blot, Amina Bousalem, Quentin Garchery, and Chantal Keller. SMTCoq : automatisation expressive et extensible dans Coq. In Nicolas Magaud and Zaynah Dargaye, editors, *Trentièmes Journées Francophones des Langages Applicatifs*, Les Rousses, France, January 2019.

[18] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3 : Shepherd your herd of provers. In *Boogie 2011 : First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. https://hal.inria.fr/hal-00790310.

[19] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 2015. URL : https://doi.org/10.1007/s10009-014-0314-5.

[20] François Bobot and Andrei Paskevich. Expressing Polymorphic Types in a Many-Sorted Language, 2011. Preliminary report. http://hal.inria.fr/inria-00591414/.

[21] Mathieu Boespflug and Guillaume Burel. CoqInE : Translating the calculus of inductive constructions into the λΠ-calculus modulo. In *Proof Exchange for Theorem Proving—Second International Workshop, PxTP*, page 44, 2012.

[22] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The λΠ-calculus modulo as a universal proof language. *Proceedings of PxTP2012 : Proof Exchange for Theorem Proving*, pages 28–43, 2012.

[23] Sascha Böhme and Tjark Weber. Fast LCF-style proof reconstruction for Z3. In *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2010. doi:10.1007/978-3-642-14052-5_14.

[24] Sylvain Boulmé. *Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles)*. HDR thesis, Université Grenoble-Alpes, 2021.

[25] Thomas Bouton, Diego Caminha B de Oliveira, David Déharbe, and Pascal Fontaine. veriT : an open, trustable and efficient SMT-solver. In *International Conference on Automated Deduction*, pages 151–156. Springer, 2009.

[26] Raphaël Cauderlier and Pierre Halmagrand. Checking Zenon modulo proofs in Dedukti. In *Proof eXchange for Theorem Proving*, volume 186 of *EPTCS*, pages 57–73, 2015. doi:10.4204/EPTCS.186.7.

[27] Arthur Charguéraud. The locally nameless representation. *Journal of automated reasoning*, 49(3) :363–408, 2012.

[28] Zakaria Chihani, Dale Miller, and Fabien Renaud. Checking foundational proof certificates for first-order logic (extended abstract). In *PxTP 2013. Third International Workshop on Proof Exchange for Theorem Proving*, 2013. doi:10.29007/7gnr.

[29] Martin Clochard. Preuves taillées en biseau. In *vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA)*, 2017.

[30] Sylvain Conchon and Évelyne Contejean. The Alt-Ergo automatic theorem prover. http://alt-ergo.lri.fr/, 2008. APP deposit under the number IDDN FR 001 110026 000 S P 2010 000 1000. URL : http://alt-ergo.lri.fr/.

[31] Évelyne Contejean. Coccinelle, a Coq library for rewriting. In *Types*, 2008.

[32] Évelyne Contejean, Claude Marché, and Landy Rabehasaina. Rewrite systems for natural, integral, and rational arithmetic. Research report, Laboratoire de Recherche en Informatique, Université de Paris-Sud, Orsay, France, 1997.

[33] Thierry Coquand. An analysis of Girard's paradox. In *Proceedings of the First Symposium on Logic in Computer Science*, Cambridge, MA, June 1986. IEEE Comp. Soc. Press.

[34] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76 :95–120, February 1988.

[35] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the λΠ-calculus modulo. In *Typed lambda calculi and applications*, pages 102–117, 2007. S. Ronchi Della Rocca, Lecture Notes in Computer Science 4583, Springer-Verlag. URL : http://www.lsv.fr/~dowek/Publi/pts.pdf.

[36] Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq : Automation for dependent type theory. *Journal of automated reasoning*, 61(1) :423–453, 2018.

[37] Sylvain Dailler, Claude Marché, and Yannick Moy. Lightweight interactive proving inside an automatic program verifier. In *Proceedings of the Fourth Workshop on Formal Integrated Development Environment*, 2018. URL : https://hal.inria.fr/hal-01936302.

[38] Leonardo De Moura and Nikolaj Bjørner. Z3 : An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[39] David Delahaye. A tactic language for the system Coq. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2000. doi:10.1007/3-540-44404-1_7.

[40] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Formal models and semantics*, pages 243–320. Elsevier, 1990.

[41] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8) :453–457, 1975.

[42] Claire Dross, Carlo A. Furia, Marieke Huisman, Rosemary Monahan, and Peter Müller. Verifythis 2019 : A program verification competition (extended report). *CoRR*, abs/2008.13610, 2020. URL : https://arxiv.org/abs/2008.13610, arXiv:2008.13610.

[43] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. SMTCoq : a plug-in for integrating SMT solvers into Coq. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 126–133. Springer, 2017. doi:10.1007/978-3-319-63390-9_7.

[44] Jean-Christophe Filliâtre. *Preuve de programmes impératifs en théorie des types*. PhD thesis, Paris 11, 1999.

[45] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. In Armin Biere and Roderick Bloem, editors, *26th International Conference on Computer Aided Verification*, volume 8859 of *Lecture Notes in Computer Science*, pages 1–16, Vienna, Austria, July 2014. Springer.

[46] Jean-Christophe Filliâtre and Andrei Paskevich. Abstraction and genericity in Why3. In Tiziana Margaria and Bernhard Steffen, editors, *9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 12476 of *Lecture Notes in Computer Science*, pages 122–142, Rhodes, Greece, October 2020. Springer. `http://why3.lri.fr/isola-2020/`.

[47] Roger E Frye. Finding $95800^4 + 217519^4 + 414560^4 = 422481^4$ on the connection machine. In *Proceedings of supercomputing*, volume 88, pages 106–116, 1988.

[48] Quentin Garchery. destruct/case transformations incorrectly handle polymorphic formulas. `https://gitlab.inria.fr/why3/why3/-/issues/525/`, 2021.

[49] Quentin Garchery. Performance slowdown with variable shadowing. `https://github.com/Deducteam/lambdapi/issues/565`, 2021.

[50] Quentin Garchery. Performance with growing context. `https://github.com/Deducteam/lambdapi/issues/579`, 2021.

[51] Quentin Garchery. Performances with both new variables and hypotheses in context. `https://github.com/Deducteam/lambdapi/issues/595`, 2021.

[52] Quentin Garchery. Performances with linear propositional problem. `https://github.com/Deducteam/lambdapi/issues/649`, 2021.

[53] Quentin Garchery. Performances with nested applications in context. `https://github.com/Deducteam/lambdapi/issues/584`, 2021.

[54] Gerhard Gentzen. Untersuchungen über das logische schließen. i. *Mathematische zeitschrift*, 39(1) :176–210, 1935.

[55] Michael JC Gordon. Edinburgh LCF : a mechanised logic of computation. 1979.

[56] David Greenaway. *Automated proof-producing abstraction of C code*. PhD thesis, CSE, UNSW, 2015. URL : `http://unsworks.unsw.edu.au/fapi/datastream/unsworks:13743/SOURCE02?view=true`.

[57] David Greenaway, Japheth Lim, June Andronick, and Gerwin Klein. Don't sweat the small stuff : Formal verification of C code without the pain. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014. URL : `https://doi.org/10.1145/2594291.2594296`.

[58] Benjamin Grégoire, Laurent Théry, and Benjamin Werner. A computational approach to Pocklington certificates in type theory. In *Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 97–113. Springer, 2006. `doi:10.1007/11737414\_8`.

[59] Mohamed Yacine El Haddad, Guillaume Burel, and Frédéric Blanqui. Ekstrakto a tool to reconstruct dedukti proofs from TSTP files. *arXiv preprint arXiv :1908.09479*, 2019.

[60] John Harrison. Towards self-verification of hol light. In *International Joint Conference on Automated Reasoning*, pages 177–191. Springer, 2006.

[61] John Harrison and Laurent Théry. A skeptic's approach to combining HOL and maple. *J. Autom. Reason.*, 21(3) :279–294, 1998. `doi:10.1023/A:1006023127567`.

[62] Chantal Keller. *A Matter of Trust : Skeptical Communication Between Coq and External Provers*. PhD thesis, Ecole Polytechnique X, 2013.

[63] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c : A software analysis perspective. *Formal Aspects of Computing*, 27(3) :573–609, May 2015. doi:10.1007/s00165-014-0326-7.

[64] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

[65] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *26th European Symposium on Programming Languages and Systems*, 2017. URL : https://hal.archives-ouvertes.fr/hal-01633133.

[66] K. Rustan M. Leino. Dafny : An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[67] Xavier Leroy. Formal certification of a compiler back-end or : programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–54, 2006.

[68] Stéphane Lescuyer. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq*. PhD thesis, Université Paris-Sud, Orsay, France, 2011. URL : https://tel.archives-ouvertes.fr/tel-00713668.

[69] P. Müller, M. Schwerhoff, and A. J. Summers. Viper : A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[70] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL : a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[71] Duckki Oe, Andrew Reynolds, and Aaron Stump. Fast and flexible proof checking for SMT. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 6–13, 2009.

[72] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In Bruno Woltzenlogel Paleo and David Delahaye, editors, *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015. URL : https://hal.inria.fr/hal-01094195.

[73] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. *ACM sigplan notices*, 23(7) :199–208, 1988.

[74] Hans-Jörg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. Alethe : Towards a generic SMT proof format (extended abstract). *CoRR*, abs/2107.02354, 2021. URL : https://arxiv.org/abs/2107.02354, arXiv:2107.02354.

[75] Raymond M Smullyan. *First-order logic*. Courier Corporation, 1995.

[76] Michael Stueben. Defensive programming. In *Good Habits for Great Coding*, pages 123–126. Springer, 2018.

[77] Geoff Sutcliffe, Jürgen Zimmer, and Stephan Schulz. TSTP data-exchange formats for automated theorem proving tools. *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, 112 :201–215, 2004.

[78] Josef Svenningsson and Emil Axelsson. Combining deep and shallow embedding for EDSL. In *International symposium on trends in functional programming*, pages 21–36. Springer, 2012.

[79] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F*. In *Principles of Programming Languages*, pages 256–270. ACM, 2016. doi:10.1145/2837614.2837655.

[80] François Thiré. Sharing a library between proof assistants : reaching out to the HOL family. *arXiv preprint arXiv :1807.01873*, 2018.

[81] François Thiré. *Interoperability between proof systems using the logical framework Dedukti*. PhD thesis, ENS Paris-Saclay, 2020.

[82] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In Renate A. Schmidt, editor, *22nd International Conference on Automated Deduction*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009. doi:http://dx.doi.org/10.1007/978-3-642-02959-2_10.

[83] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker : Practical implementations and other applications. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 880–885. IEEE, 2003.

Liste des symboles

Nous donnons ici des indications sur les conventions d'utilisation des symboles que nous suivons dans ce manuscrit.

$\mathbb{N}$	entiers naturels
$\mathbb{Z}$	entiers relatifs
$\alpha,\ \beta,\ \gamma$	variable de type
ι	symbole de type
τ	type
$x,\ y$	variable
$f,\ t,\ u$	formule, terme ou variable
σ	substitution
$H,\ G,\ P$	identifiant d'une formule
I	signature de type
Σ	signature
Γ	dictionnaire des hypothèses
Δ	dictionnaire des buts
T	tâche de preuve
L	liste
ϕ, Φ	transformation
$\mathbb{D}$	collection de domaines
$\mathcal{M}$	modèle
θ	valuation de type
ξ	valuation
$c,\ C$	certificat

Index

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Certification de la transformation de tâches de preuve

Mots clés : méthodes formelles pour le génie logiciel, certificats, preuve de programmes, transformations logiques, prouveurs automatiques, prouveurs interactifs, outil de vérification déductive Why3, *framework* logique Lambdapi

Résumé : De nombreux prouveurs et outils de vérification font un usage intensif des *transformations logiques* afin de ramener un problème exprimé sous la forme d'une tâche de preuve à un certain nombre de tâches de preuve plus simples à valider. Les transformations font souvent partie de la base de confiance de l'outil de vérification. Cette thèse a pour objectif de renforcer la confiance accordée aux transformations logiques. Les transformations sont instrumentées pour produire des certificats puis ceux-ci sont vérifiés par un outil externe : c'est l'approche *sceptique*. De ce fait, notre méthode est incrémentale et robuste aux modifications apportées au code des transformations. Nous définissons deux formats de certificats ; les transformations génèrent des *certificats de surface* et ces certificats sont traduits en des *certificats de noyau* qui sont destinés à la vérification finale. L'accent est mis sur la facilité de production des certificats de surface et nous avons fait en sorte qu'ils soient, autant que possible, indépendants des tâches de preuve, facilitant ainsi leur composition et rendant la certification plus modulaire. Les certificats de noyau, au contraire, incluent de nombreux détails tout en restant élémentaires, de sorte que leur vérification est réalisable par un outil simple, dont la confiance est facile à établir. Nous proposons une procédure de traduction d'un certificat de surface en un certificat de noyau qui n'a pas besoin d'être certifiée. Les transformations logiques sont considérées dans une logique d'ordre supérieur avec polymorphisme de type, ce formalisme pouvant être étendu avec des théories interprétées telles que l'égalité ou l'arithmétique entière. Nous appliquons notre méthode à Why3, et notamment à des transformations complexes qui préexistent à notre travail. Nous implémentons également deux vérificateurs de certificats, le premier reposant sur une approche calculatoire efficace et l'autre s'appuyant sur un encodage superficiel des tâches de preuve dans le *framework* logique Lambdapi, donnant ainsi de fortes garanties de sa correction.

Title : Certification of the transformation of proof tasks

Keywords : formal methods for software engineering, certificates, program verification, logical transformations, automated theorem prover, proof assistant, deductive verification tool Why3, logical framework Lambdapi

Abstract : In various provers and deductive verification tools, *logical transformations* are used extensively in order to reduce a proof task into a number of simpler tasks. Logical transformations are often part of the trusted base of such tools. In this thesis, we develop a framework to improve confidence in their results. We follow a *skeptical* approach : transformations are instrumented to produce certificates that are checked by a third-party tool. Thus, we benefit from a modular approach that is also robust to changes in the source code of the transformations. We design two kinds of certificates. Transformations produce *surface certificates* that are then translated to *kernel certificates* which are destined for final verification. We made sure that surface certificates are easy to produce. Moreover, surface certificates are as independent of the transformation application as possible and this makes for a modular certification. On the contrary, kernel certificates include numerous details about the transformation application and are kept elementary. This helps to define simpler checkers and establish their correctness. We propose a translation procedure from surface certificates to kernel certificates which does not need to be trusted. Logical transformations are considered in a higher-order logic, with type polymorphism and built-in theories such as equality and integer arithmetic. We apply our framework to Why3 and use it to instrument pre-existing and complex transformations. Additionally, we implement two certificate checkers. The first one follows an efficient computational approach while the second is based on a shallow embedding of proof tasks inside the logical framework Lambdapi, thus exhibiting formal guaranties of its correctness.

Maison du doctorat de l'Université Paris-Saclay
2ème étage aile ouest, Ecole normale supérieure Paris-Saclay
4 avenue des Sciences,
91190 Gif sur Yvette, France