Programmes avec effets et leurs preuves dans la théorie des types : application à la compilation certifiée et aux traitements de paquets certifiés

Pierre Nigron

▶ To cite this version:

Pierre Nigron. Programmes avec effets et leurs preuves dans la théorie des types : application à la compilation certifiée et aux traitements de paquets certifiés. Langage de programmation [cs.PL]. Sorbonne Université, 2022. Français. NNT : 2022SORUS480 . tel-04028224

HAL Id: tel-04028224
https://theses.hal.science/tel-04028224

Submitted on 14 Mar 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT DE SORBONNE UNIVERSITÉ

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Pierre NIGRON

Pour obtenir le grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Sujet de la thèse :

Effectful programs and their proofs in type theory : application to certified compilation and certified packet processing

soutenue le 17 novembre 2022

devant le jury composé de :

Mme. Julia LAWALL	Directrice de thèse
M. Pierre-Évariste DAGAND	Encadrant de thèse
M. Sylvain BOULMÉ	Rapporteur
M. Yann RÉGIS-GIANAS	Rapporteur
Mme. Christine TASSON	Examinatrice
M. Emmanuel CHAILLOUX	Examinateur

Table des matières

1 Introduction **6**
- 1.1 Les monades 7
- 1.2 La monade libre 10
- 1.3 Raisonner avec la monade libre 11
  - 1.3.1 Triplets de Hoare 11
  - 1.3.2 Logique de séparation 11
  - 1.3.3 Transformateurs de prédicats 12
- 1.4 Contributions 13

2 Préliminaires **15**
- 2.1 Logique de séparation [🐓] 15
  - 2.1.1 Dénombrable 15
  - 2.1.2 Ensemble 15
  - 2.1.3 Prédicats d'ensemble 16
  - 2.1.4 MoSel 18
- 2.2 Monade libre [🐓] 21
  - 2.2.1 Erreur 22
  - 2.2.2 État 23
- 2.3 Transformateur de prédicats 24
- 2.4 Compositionnalité des effets 24

3 Fraîcheur **27**
- 3.1 Réétiquetage [🐓] 27
  - 3.1.1 Présentation algébrique [🐓] 29
- 3.2 Preuves monadiques en logique de séparation [🐓] 31
  - 3.2.1 Adéquation entre la logique de programme et l'interpréteur 33
- 3.3 Étude de cas : `SimplExpr` 34
  - 3.3.1 La monade [🐓] 35
  - 3.3.2 Preuves et Programmes [🐓] 37
  - 3.3.3 Quitter `iProp` [🐓] 42
  - 3.3.4 Problème de performance 43
- 3.4 Travaux connexes 45
- 3.5 Conclusion 46

4 Décodeurs zéro-copie et leur sûreté mémoire **48**
- 4.1 Décodeurs [🐓] . . . 49
  - 4.1.1 Décodeurs avec pointeurs [🐓] . . . 54
- 4.2 Sûreté mémoire et zéro-copie [🐓] . . . 56
  - 4.2.1 Sûreté mémoire . . . 58
  - 4.2.2 Sûreté en pratique . . . 58
  - 4.2.3 Zéro-copie [🐓] . . . 59
  - 4.2.4 Zéro-copie et Logique de séparation [🐓] . . . 59
- 4.3 Raisonnement sur les décodeurs en Coq . . . 60
  - 4.3.1 Preuve de zéro-copie des programmes [🐓] . . . 61
  - 4.3.2 Modèle de la sémantique [🐓] . . . 66
  - 4.3.3 Preuve de sûreté . . . 66
- 4.4 Formalisation de Nom . . . 67
  - 4.4.1 Types de bases supplémentaires . . . 67
  - 4.4.2 Les combinateurs [🐓] . . . 68
  - 4.4.3 Théorie Équationnelle [🐓] . . . 73
  - 4.4.4 Sémantique [🐓] . . . 75
- 4.5 Sémantique par transformateur de prédicats [🐓] . . . 79
  - 4.5.1 Cohérence [🐓] . . . 80
  - 4.5.2 Preuve de sûreté avec Nom [🐓] . . . 82
  - 4.5.3 Raisonnement [🐓] . . . 82
- 4.6 Cas pratique : Radius [🐓] . . . 83
  - 4.6.1 Formats . . . 83
  - 4.6.2 Structure de données et décodeurs [🐓] . . . 84
  - 4.6.3 Spécification et preuve de zéro-copie [🐓] . . . 86
- 4.7 Travaux connexes . . . 89
- 4.8 Conclusion . . . 91

5 Raffinement des décodeurs Gallina en C **92**
- 5.1 Raffinement `decode_packet_SSH` [🐓] . . . 93
  - 5.1.1 Représentation . . . 94
  - 5.1.2 Sémantique de la représentation . . . 95
  - 5.1.3 Relation . . . 95
  - 5.1.4 Règles d'inférence . . . 99
- 5.2 Réification des décodeurs Nom . . . 106
  - 5.2.1 Syntaxe et sémantique complètes [🐓] . . . 107
  - 5.2.2 Raffinement [🐓] . . . 108
- 5.3 Paramétricité des variables dans les programmes générés [🐓] . . . 114
- 5.4 Génération de code C [🐓] . . . 117
  - 5.4.1 Compilation vers C . . . 117
  - 5.4.2 Brève analyse du code produit . . . 119
  - 5.4.3 Reproductibilité des tests . . . 120
- 5.5 Travaux Connexes . . . 121

5.6 Conclusion et limitations . . . 122
5.6.1 Limitations . . . 123

6 Conclusion **124**
6.1 Fraîcheur . . . 124
6.1.1 Résumé de la contribution . . . 124
6.1.2 Limitations . . . 124
6.1.3 Pistes d'amélioration . . . 125
6.2 Décodeurs zéro-copie et leur sûreté mémoire . . . 125
6.2.1 Résumé de la contribution . . . 125
6.2.2 Limitations . . . 125
6.2.3 Pistes d'amélioration . . . 125
6.3 Raffinement des décodeurs Gallina en C . . . 126
6.3.1 Résumé de la contribution . . . 126
6.3.2 Limitations . . . 126
6.3.3 Pistes d'amélioration . . . 126
6.4 Contribution logicielle . . . 127

Chapitre 1

Introduction

Depuis plusieurs décennies, les assistants de preuves gagnent en popularité. Ce sont des systèmes informatiques dans lesquels les utilisateurs peuvent faire des mathématiques. Ils peuvent décrire des objets et raisonner dessus à travers des théorèmes et leurs preuves. Lorsqu'un utilisateur écrit une preuve, il doit expliquer chacune des étapes de son raisonnement. Ainsi, la cohérence du raisonnement est vérifiée automatiquement par le système. La confiance portée à la preuve est alors accrue. En effet, en plus de l'auteur, la fiabilité de la preuve se base sur l'assistant de preuves et donc sur son bon fonctionnement. Nous limitons la correction de la preuve, non plus au cas particulier qu'elle représente, mais à un système externe formalisant les règles de raisonnement. La cohérence logique est alors concentrée dans un seul programme où chaque augmentation de sa fiabilité améliore la fiabilité de la preuve, des autres déjà formalisées et des prochaines à venir.

Puisqu'un assistant de preuves permet de formaliser des notions mathématiques, il permet l'écriture de fonctions. Nous adoptons un point de vue constructiviste des mathématiques et pouvons donc concevoir une fonction comme une suite de calculs. Or, un programme est également une suite de calculs. En plus d'effectuer des raisonnements, le système nous fournit donc un environnement pour implémenter des programmes. Par exemple, si nous voulons implémenter la division par soustractions successives d'entiers positifs calculant le quotient, nous pouvons définir le programme ainsi :

```
div m n default := default                      si    n = 0
div m n default := 0                            si    m < n
div m n default := 1 + div (m - n) n default  sinon
```

Nous vérifions que le diviseur n'est pas 0, s'il l'est, nous renvoyons une valeur par défaut, sinon nous décrivons la division par soustraction. La définition (ici sous forme d'équation) est alors fonctionnelle. La fonction décrite est un programme.

Nous sommes en mesure d'utiliser pleinement les outils du système afin de décrire des propriétés vis-à-vis du comportement des programmes implémentés. À l'aide des règles de raisonnement du système, nous pouvons alors prouver sa correction, c'est-à-dire montrer que les propriétés attendues (la spécification) sont bien respectées par le programme.

Dans le même langage que la définition de `div`, nous pouvons spécifier et prouver la

correction de la fonction div :

```
∀ m n default, n ≠ 0 → ∃ r, n * div m n default + r = m ∧ 0 <= r < n
```

Cependant, un programme manipule généralement un état implicite, c'est-à-dire des paramètres externes utilisés pour produire la valeur. Par exemple, un programme peut récupérer la valeur stockée à une adresse mémoire donnée en argument. Le résultat du programme dépend donc de l'état de la mémoire à son appel. Deux appels à ce même programme pour la même adresse mémoire peut produire deux résultats différents. Les programmes ayant une dépendance à des paramètres extérieurs pour effectuer leur calcul (des effets de bord) sont appelés impurs (en opposition aux programmes purs qui n'ont pas d'effet de bord).

Cependant, les assistants de preuves permettent seulement de décrire des objets mathématiques. Il n'est donc possible que d'implémenter des fonctions pures, excluant ainsi des programmes impurs pourtant essentiels à la programmation. En effet, par soucis de cohérence logique, ces systèmes exigent que deux fonctions avec la même entrée produisent le même résultat. Cette propriété, permettant d'effectuer des raisonnements équationnels, interdit les effets de bord dans les programmes.

Toutefois, il est toujours possible de représenter et de simuler les comportements liés à un état implicite. La solution est d'expliciter l'état du programme à l'aide d'arguments supplémentaires et d'enrichir le retour des fonctions afin de décrire les modifications. L'égalité entre deux programmes prend alors en compte ces éléments supplémentaires et donc leur contexte d'appel, permettant ainsi d'avoir des raisonnements équationnels.

Pour la fonction div décrite précédemment, nous pouvons par exemple échouer plutôt que renvoyer une valeur par défaut. Pour cela, nous enrichissons le type de retour en ajoutant le cas ne retournant pas d'entier. Nous définissons un type de données ajoutant une valeur null aux entiers, représentant ainsi l'échec du calcul. Nous pouvons alors définir une nouvelle définition de div renvoyant un entier ou échouant ainsi :

```
div m n := null   si  n = 0
div m n := 0      si  m < n
div m n := 1 + v  si  div (m - n) n = v
div m n := null   si  div (m - n) n = null
```

Bien qu'il soit nécessaire de prendre en compte l'état lors des raisonnements (null dans le cas de div), il n'est en réalité par forcément nécessaire de le rendre explicite (on doit pouvoir se séparer de la quatrième équation de div lors de son implémentation). En effet, toutes les opérations ne vont pas forcément le manipuler. Il est alors pertinent de séparer les opérations en deux parties. Celles que nous appelons "pures" qui ne manipulent pas cet état implicite et les "impures", les effets.

1.1 Les monades

Les monades sont une abstraction possible. Elles permettent à la fois de séparer les opérations pures et impures mais aussi de camoufler la manipulation de l'état à travers

une couche d'opérateurs. Une monade est munie d'un type de données `M` encapsulant l'état implicite, d'un opérateur `ret` représentant les calculs purs et d'un opérateur `let _ in _` permettant de séquencer plusieurs opérations monadiques, c'est-à-dire plusieurs opérations pouvant manipuler l'état implicite.

```
M : Type → Type

ret : X → M X

let _ in _ : M X → (X → M Y) → M Y
```

Une opération de type `M X` est donc une opération manipulant potentiellement un état implicite et produisant une valeur de type `X`. Les monades sont accompagnées de trois équations capturant les propriétés attendues d'une séquence :

`let r := ret v in` `k r`	=	`k v`
`let r := m in` `ret r`	=	`m`

`let v := m in`
`let r := e v in`
`k r`
	=	`let r :=`
		` let v := m in`
		` e v`
		`in k r`

Chaque monade est constituée de ses propres opérateurs, correspondants à ses effets, c'est-à-dire aux manipulations permises sur l'état implicite décrit par son type de données. Les deux premières équations garantissent que l'opérateur `ret` capture bien les calculs indépendants de l'état implicite. Ces équations montrent alors qu'utiliser la séquence monadique (permettant à la fois de séquencer les effets **et** les calculs pour produire une valeur) est superflue puisque seuls des calculs purs sont effectués. La troisième équation garantit que la séquence est associative.

De plus, elles peuvent décrire les effets à l'aide d'une théorie équationnelle spécifiant les relations attendues entre les différents opérateurs. Prenons comme exemple deux monades bien connues. La première est celle manipulant un état mutable de type `S`. Elle a deux opérateurs, l'opérateur `get : unit → M S` permettant de connaître l'état actuel et `set : S → M unit` le modifiant. Leur comportement peut alors être décrit à l'aide des quatre équations suivantes :

`let _ := set s in`
`let _ := set t in`
`k`
	=	`let _ := set t in`
		`k`

`let s := get () in`
`let t := get () in`
`k s t`
	=	`let s := get () in`
		`k s s`

`let _ := set s  in`
`let t := get () in`
`k t`
	=	`let _ := set s in`
		`k s`

`let s := get () in`
`let _ := set s  in`
`k`
	=	`k`

Les deux premières équations garantissent que `set` ne fait que modifier l'état par la valeur qu'on lui donne et que `get` ne modifie pas l'état. Il est donc inutile d'appeler deux fois successivement ces opérateurs : `get` renvoie deux fois la même valeur et le second appel à `set` écrase la modification du premier appel. Les deux équations décrivent la relation entre les deux opérateurs. On récupère toujours l'état de la dernière modification et il ne sert à rien de modifier l'état par la dernière valeur récupérée si aucune modification n'a été faite entre-temps.

La deuxième est la monade d'erreur qui n'a qu'un seul opérateur `fail` signalant une erreur. La monade a alors une équation indiquant que `fail` interrompt le calcul :

`let v := fail in k v` = `fail`

Alors que les monades fournissent un environnement propice pour décrire des calculs avec un certain effet, elles souffrent de certaines limitations. Tout d'abord, plusieurs effets sont généralement nécessaires pour écrire des programmes. Il serait donc idéal d'avoir un moyen de composer deux monades afin de combiner leurs effets. Par exemple, en combinant la monade d'exception et d'état mutable, il est possible d'écrire des programmes manipulant les deux effets. Malheureusement, la composition de monades n'est pas close. Certaines compositions sont possibles mais ad-hoc. Le problème de composition est dû à la séquence, il n'y a pas de moyen de construire génériquement la séquence de deux monades à partir de leur séquence respective. La raison est que les opérations peuvent interférer entre elles. En effet, si on observe la signature de la séquence (`M X → (X → M Y) → M Y`), on remarque à travers le type du deuxième argument que le résultat de la première opération est directement accessible pour la suite du calcul. Par conséquent, les opérations des deux monades que l'on souhaite composer peuvent interagir. Cette expressivité impose aux monades d'avoir plus de structures et notamment d'avoir des opérations supplémentaires expliquant comment interagissent les états implicites.

1.2 La monade libre

Une approche pour pallier au problème de compositionnalité est la monade libre. L'idée est qu'un effet est simplement un ensemble d'opérateurs. Pour décrire un effet, il n'est alors plus nécessaire de définir l'état implicite à travers un type de données ad-hoc. Il suffit simplement de décrire la signature des opérateurs. En mettant de côté la description de l'état implicite, nous séparons alors la syntaxe et la sémantique des programmes. En effet, par simple construction syntaxique, la monade libre unit les deux types d'opérations des monades : celles effectuant des calculs purs et celles exploitant un effet à travers un appel à l'un de ces opérateurs.

```
  pure   : X → Free SIG X
| impure : SIG X → (X → Free SIG Y) → Free SIG Y
```

où `SIG` est la signature des opérateurs des effets. Là où le constructeur `pure` est analogue à `ret`, le constructeur `impure` est inédit et a deux arguments : un appel à un opérateur décrit dans la signature `SIG` produisant une valeur de type `X` et une continuation exploitant le résultat (de type `X`) de l'appel.

Il est important de noter que ces deux constructeurs sont indépendants des opérateurs de la monade. De plus, nous n'avons décrit aucun calcul, seulement de la syntaxe via des constructeurs. Ils sont ainsi indépendants de toute sémantique et donc de la description de l'état implicite. Cette construction plus générique permet alors d'avoir une définition pour la séquence de la monade indépendamment de ses effets :

```
let _ in _ : Free SIG X → (X → Free SIG Y) → Free SIG Y
let x := pure v     in k x := k v
let x := impure s c in k x := impure s (fun v ⇒ let! x := c v in k x)
```

Tous les effets ont ainsi la même séquence ! Il n'y a plus de problème de composition.

La monade libre est paramétrée par les opérateurs qu'elle manipule (le type `SIG`). Nous pouvons l'instancier en une monade simplement en donnant la signature de ces effets. Par exemple, un type `STATE` contenant deux constructeurs pour l'état mutable où `S` est le type de l'état

```
  get : unit → STATE S
| set : S → STATE unit
```

ou le type `ERROR` pour l'erreur avec un seul opérateur

```
  fail : ∀ X, ERROR X.
```

Il est alors possible de les combiner grâce à la somme ou au tenseur.

Les programmes implémentés sont alors indépendants de toute sémantique. Pour effectuer des calculs, il reste à définir des gestionnaires d'effets décrivant le comportement calculatoire de chacun des opérateurs.

1.3 Raisonner avec la monade libre

Les programmes n'étant que des objets syntaxiques, nous pouvons utiliser la théorie équationnelle de chacun des effets pour raisonner. Cependant, tous les effets n'en ont pas une. Une autre approche adaptée à la spécification et à la vérification de programmes avec effets est l'utilisation de logique de programmes. Plusieurs décennies de recherche ont été investies pour raisonner sur des programmes impératifs et pour développer des logiques de programmes adaptées. Or, un programme monadique est un programme impératif. Il est donc assez naturel que les résultats obtenus par ces travaux soient adaptés aux programmes monadiques.

1.3.1 Triplets de Hoare

Un résultat majeur de ces recherches est les triplets de Hoare [31]. Les jugements de cette logique de programme ont la forme suivante : $\{\ P\ \}\ e\ \{\ Q\ \}$. Informellement, si la précondition P est respectée avant l'exécution de e et que l'exécution de e termine alors le prédicat Q (la postcondition) est respecté par la valeur calculée. Les triplets de Hoare sont alors dérivés à partir de règles d'inférence. Les trois règles essentielles de cette logique de programme (en mettant de côté la règle d'itération) sont alors les suivantes :

$$\frac{}{\{\ Q\ v\ \}\ \mathbf{return}\ v\ \{\ Q\ \}}\ \textsc{règle-valeur}$$

$$\frac{\{\ P\ \}\ e\ \{\ H\ \} \qquad \forall\ v.\ \{\ H\ v\ \}\ k\ v\ \{\ Q\ \}}{\{\ P\ \}\ \mathbf{let}\ v\ =\ e\ \mathbf{in}\ k\ v\ \{\ Q\ \}}\ \textsc{règle-composition}$$

$$\frac{P \to H \qquad \{\ H\ \}\ e\ \{\ S\ \} \qquad \forall\ v.\ S\ v \to Q\ v}{\{\ P\ \}\ e\ \{\ Q\ \}}\ \textsc{règle-conséquence}$$

La première règle est directe, elle indique qu'une valeur respecte la postcondition si nous le savons déjà avant de la renvoyer. La règle de composition permet de décomposer le raisonnement sur la séquence en deux raisonnements : le premier sur le sous-programme en tête de la séquence et le deuxième sur la suite. Les propriétés obtenues sur le programme en-tête sont alors utilisées pour décrire la valeur calculée et sont donc transmises pour raisonner sur la suite du programme. En décomposant le raisonnement, il est alors simplifié. Il suffit de spécifier chacune des opérations élémentaires du langage pour raisonner sur n'importe quel programme. La dernière règle indique que nous pouvons librement affaiblir la précondition ou renforcer la postcondition du triplet que nous devons montrer.

1.3.2 Logique de séparation

O'Hearn et al. [58] étendent la logique de Hoare en permettant des raisonnements sur des notions de ressources [57, 63]. L'objectif est d'améliorer la modularité du raisonnement

en se limitant aux ressources manipulées localement par le programme. Par exemple, si un programme manipule une liste de cellules mémoires alors il est uniquement nécessaire de raisonner sur les cellules mémoires manipulées par cette liste. Les autres cellules sont localement retirées du raisonnement. De cette idée de localité dans le raisonnement est née la logique de séparation [66, 16]. Cette logique ajoute l'opérateur binaire où $P * Q$ exprime que P et Q raisonnent sur des portions disjointes de la mémoire. En utilisant cet opérateur, une nouvelle règle de raisonnement peut être ajoutée à la logique de Hoare :

$$\frac{\{\ P\ \}\ e\ \{\ Q\ \}}{\{\ P * R\ \}\ e\ \{\ Q * R\ \}}\ \text{RÈGLE-ENCADREMENT}$$

La règle d'encadrement permet de raisonner localement sur nos programmes. Lorsque nous montrons qu'un programme respecte sa spécification, nous garantissons également qu'il ne modifie pas d'autres portions de la mémoire. Ainsi, nous pouvons librement élargir le contexte (représenté par R dans la règle) sans craindre que d'autres portions mémoires soient altérées (R est toujours correcte après l'exécution du programme).

1.3.3 Transformateurs de prédicats

Toujours dans l'objectif de raisonner sur des programmes impératifs, les transformateurs de prédicats [25] sont un autre résultat majeur de ces recherches. Les transformateurs de prédicats sont des fonctions monotones entre des prédicats. Celui que nous utiliserons tout au long de cette thèse est le calcul de plus faible précondition. Le principe est de calculer la précondition la plus faible nécessaire pour que la valeur renvoyée par un programme respecte une postcondition. Nous notons le transformateur ainsi : `wp p Q` où `p` est le programme et `Q` la postcondition.

En adaptant cette construction à la monade, nous pouvons définir des règles de raisonnements communes à tous les effets. Pour les opérations pures, nous exprimons que la valeur calculée doit respecter la postcondition :

```
wp (ret v) Q = Q v
```

et pour la séquence, nous pouvons l'associer à la composition des transformateurs de prédicats :

```
wp (let v := m in k v) Q = wp m (fun v ⇒ wp (k v) Q)
```

En couplant l'approche par transformateurs de prédicats et la construction libre des monades, nous n'obtenons non plus des règles similaires pour raisonner sur les programmes avec effets mais littéralement les mêmes règles puisque tous les effets utilisent la même construction. Pour décrire les effets, il faut alors donner une spécification à chacun des opérateurs, similairement aux gestionnaires d'effets pour l'aspect calculatoire.

1.4 Contributions

Durant cette thèse, nous étudions l'utilisation de transformateurs de prédicats et de la logique de séparation pour raisonner mécaniquement sur des développements logiciels réalistes utilisant les effets. L'objectif est de profiter des bonnes propriétés de la logique de séparation lors de la correction des programmes tout en obtenant des spécifications satisfaisantes et utilisables à l'extérieur de cette logique. Dans cette thèse, nous nous demandons s'il est intéressant d'utiliser localement une logique de séparation définie dans `Prop` (les propositions de Coq). Nous nous demandons quels sont les surcoûts d'une telle pratique, pour spécifier et en terme d'effort de preuve, notamment lorsque nous devons interagir avec des propositions dans `Prop`. Est-ce que les conséquences d'un tel choix sont globalement positives ?

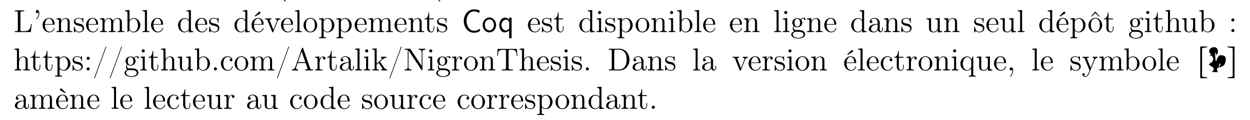
Lorsque la spécification d'un programme est simplifiée grâce à la logique de séparation, est-ce que la preuve de correction l'est également ? Est-ce que les spécifications dans `Prop` dépendantes de cette spécification sont complexifiées ? et leur preuve de correction ?

De plus, est-ce que les spécifications dans la logique de séparation sont aisément utilisables pour prouver des propriétés sur un logiciel spécifié globalement dans `Prop` ? Comment faire convenablement interagir les deux logiques ?

Nous n'abordons pas la compositionnalité. Cependant, la compositionnalité reste un atout majeur de l'utilisation de la monade libre. L'adaptation de ces travaux est donc un angle omniprésent des travaux futurs tout au long du manuscrit. Les contributions de cette thèse sont les suivantes :

- Nous commençons par utiliser la logique de séparation pour raisonner sur des programmes capables de générer des symboles frais (Chapitre 3). Nous illustrons l'approche sur l'exemple classique de vérification [32, 76] consistant à l'étiquetage d'un arbre (Section 3.1) et mettons en avant les points clés de la méthodologie (Section 3.2). Nous fournissons une étude de cas plus large portée sur le module `SimplExpr` de CompCert [11] dans la section 3.3. Ce module s'appuie majoritairement sur une monade offrant un générateur de nom frais ainsi que des exceptions non rattrapables. Nous montrons ainsi que la logique de séparation peut être utilisée localement, tandis que les théorèmes qui en résultent peuvent être intégrés dans un développement plus large (préexistant) se tenant uniquement dans `Prop`. Ces travaux ont été publiés et présentés lors la conférence ITP de 2021 [56].
- Nous raisonnons ensuite sur l'aspect zéro-copie de bibliothèques de décodeurs (Chapitre 4). Nous formalisons le concept de zéro-copie (Section 4.2) et proposons une méthodologie pour montrer qu'un décodeur zéro-copie est sûr (Section 4.2.4). Nous étudions l'approche sur la bibliothèque Nom [22] écrite en Rust. Nous proposons une formalisation de la bibliothèque (Section 4.4) puis définissons une logique de programme (Section 4.5) en suivant la méthode proposée.
- Dans le chapitre 5, nous transformons par raffinement nos décodeurs définis avec la bibliothèque vers un type de données intermédiaire. Ainsi, nous les optimisons et les compilons en C. Nous présentons ce type de données dans la Section 5.2 puis présentons la méthode de raffinement (Section 5.2.2). Pour finir, nous utilisons

l'extraction Coq pour extraire ce type de données et écrire un compilateur produisant un fichier C (Section 5.4).



L'ensemble des développements Coq est disponible en ligne dans un seul dépôt github : https://github.com/Artalik/NigronThesis. Dans la version électronique, le symbole [🐓] amène le lecteur au code source correspondant.

Chapitre 2

Préliminaires

Dans ce chapitre, nous présentons les objets `Coq` que nous manipulons tout au long du manuscrit. Nous commençons par la logique de séparation que nous instancions dans l'outil `MoSel`. Nous présentons ensuite la monade libre que nous accompagnons de quelques exemples. Nous abordons l'utilisation usuelle des transformateurs de prédicats en s'appuyant sur l'un des exemples. Pour finir, même si la compositionnalité fait partie des travaux futures, nous présentons deux méthodes pour l'obtenir avec les objets présentés.

2.1 Logique de séparation [🐓]

Nous commençons par définir la logique de séparation que nous utilisons dans les chapitres 3 et 4. L'objectif de cette logique est de raisonner sur la disjonction d'ensembles finis d'objets. Pour définir la logique, nous utilisons une définition d'ensembles finis et de dénombrabilité venant d'une librairie annexe [48].

2.1.1 Dénombrable

Avant de donner la définition d'un type dénombrable, nous introduisons le type `positive` dont les termes sont les entiers naturels strictement positifs représentés sous forme binaire et le type `option X` muni des constructeurs `None : option X` et `Some : X → option X`. Pour qu'un type `X` soit dénombrable (noté `Countable X`), il doit tout d'abord posséder une relation d'égalité décidable entre les termes de type `X`. Ensuite, il doit exister deux fonctions `encode : X → positive` et `decode : positive → option X` telles que `∀ x, decode (encode x) = Some x`.

2.1.2 Ensemble

Les ensembles finis sont de type `gset X` où `X` est le type des éléments. Nous n'avons pas besoin de connaître les détails de la définition des ensembles. Cependant, nous utilisons plusieurs opérateurs et prédicats associés et nous présentons donc leur syntaxe. La proposition `g ## h` signifie que `g` et `h` sont disjoints, c'est-à-dire qu'il existe aucun élément contenu

à la fois dans `g` et dans `h`. La notation ∅ dénote l'ensemble vide et `{[ l ]}` l'ensemble singleton contenant seulement l'élément `l`. Le calcul `g ∪ h` produit un ensemble contenant les éléments de `g` et de `h`. En plus de la libraire, nous définissons la fonction `inject m n` renvoyant un ensemble contenant les entiers de `m` à `n`. L'ensemble est vide si `n <= m`.

2.1.3 Prédicats d'ensemble

La logique de séparation raisonne sur des prédicats décrivant des propriétés sur un ensemble fini. L'ensemble doit être fini afin de garantir qu'il existe des objets distincts de ceux contenus dans l'ensemble. Les prédicats de la logique ont donc le type suivant :

```
Definition hprop := gset positive → Prop.
```

L'opérateur essentiel de la logique est la conjonction de séparation (notée ∗). Soit `P` et `Q` des prédicats de type `hprop`, le prédicat `P ∗ Q` décrit un ensemble fait de deux parties disjointes telles que `P` satisfasse l'une et `Q` l'autre. Nous définissons le prédicat (nommé `hstar` mais noté ∗) ainsi :

```
Definition hstar (P Q : hprop) : hprop :=
  fun h ⇒ ∃ h1 h2, P h1 ∧ Q h2 ∧ h1 ## h2 ∧ h = h1 ∪ h2.
```

Nous définissons les prédicats `hempty` (noté `emp`) garantissant que l'ensemble est vide, et `hsingle l` (noté `& l`) décrivant l'ensemble singleton contenant `l` :

```
Definition hempty : hprop := fun h ⇒ h = ∅.

Definition hsingle `{Countable X} (l : X) : hprop :=
  fun h ⇒  h = {[ encode l ]}.
```

En utilisant ces opérateurs, nous pouvons par exemple exprimer qu'une liste d'entiers ne contient pas de doublon :

```
Fixpoint NoDuplicate (l : list nat) :=
 match l with
 | nil ⇒ emp
 | h :: t ⇒ & h ∗ NoDuplicate t
 end.
```

Prenons l'exemple d'une liste contenant les entiers naturels `x`, `y` et `z`. Si nous déroulons la définition de `NoDuplicate [x; y; z]`, nous obtenons le prédicat suivant :

```
& x ∗ & y ∗ & z
```

qui se réduit ainsi :

```
fun h ⇒
 h = {[ x ]} ∪ ({[ y ]} ∪ {[ z ]}) ∧
 {[ y ]} ## {[ z ]} ∧
 {[ x ]} ## ({[ y ]} ∪ {[ z ]}).
```

L'absence de doublon se déduit alors naturellement des disjonctions d'ensembles singletons.

La logique de séparation nous permet également d'exprimer des propriétés indépendantes de l'ensemble. Nous les appelons **pure**. Pour les intégrer à la logique, nous définissons un nouvel opérateur que nous notons ⌜ P ⌝, où P est une proposition de type Prop :

```
Definition hpure (P : Prop) : hprop := fun _ ⇒ P.
```

Pour décrire des propriétés sur nos opérateurs (ou plus tard des règles de raisonnement), nous définissons l'implication P ⊢ Q signifiant qu'un ensemble satisfaisant P satisfait également Q, où P et Q sont des prédicats d'ensembles.

Nous la définissons ainsi :

```
Definition entails (P Q : hprop) : Prop := ∀ h, P h → Q h.
```

Une propriété fondamentale de notre approche est que deux prédicats singletons séparés par la conjonction de séparation expriment que les deux éléments des singletons sont différents :

```
Lemma singleton_neq `{Countable X} : ∀ (v t : X), & v ∗ & t ⊢ ⌜v ≠ t⌝.
```

Ce lemme est important car il permet d'obtenir des propriétés exprimées dans Prop à partir de propriétés exprimées avec la logique de séparation.

Faisons un petit aparté dans la définition des opérateurs pour remarquer que ⌜ P ⌝ est un prédicat **absorbant**. Un prédicat P est absorbant si pour tout Q : hprop, l'assertion P ∗ Q ⊢ P est vraie. Informellement, si la conséquence d'une implication est absorbante alors nous pouvons librement éliminer un impliquant. Par exemple, ⌜ P ⌝ ∗ Q ⊢ ⌜ P ⌝ est prouvable car le prédicat ⌜ P ⌝ est respecté par n'importe quel ensemble (puisque la proposition P est indépendante de l'ensemble). Ainsi dans les deux côtés de l'implication, le prédicat ⌜ P ⌝ peut être satisfait par deux ensembles différents. Si le prédicat est satisfait par l'ensemble vide en tant qu'impliquant, il est alors satisfait par l'ensemble satisfaisant Q en tant qu'impliqué. Nous définissons la modalité < absorb > P := P ∗ ⌜ True ⌝ représentant la version absorbante du prédicat P.

Nous transportons les opérateurs de la logique des propositions Coq et les quantificateurs dans notre logique de séparation. Nous utilisons les mêmes notations ($\wedge$, $\vee$, $\rightarrow$, $\forall$, $\exists$) et précisons lequel est utilisé lorsque le contexte ne le permet pas.

```
Definition hand (P Q : hprop) : hprop := (∗ P ∧ Q ∗)
  fun h ⇒ P h ∧ Q h.

Definition hor (P Q : hprop) : hprop := (∗ P ∨ Q ∗)
  fun h ⇒ P h ∨ Q h.

Definition himpl (P Q : hprop) := (∗ P → Q ∗)
  fun h ⇒ P h → Q h.

Definition hexist {A} (J : A → hprop) : hprop := (∗ ∃ a, J ∗)
  fun h ⇒ ∃ a, J a h.
```

```
Definition hforal {A} (J : A → hprop) : hprop := (* ∀ a, J *)
  fun h ⇒ ∀ a, J a h.
```

Nous transposons également la proposition toujours vraie et la proposition toujours fausse à partir du prédicat pure :

```
Definition htrue := ⌜ True ⌝.

Definition hfalse := ⌜ False ⌝.
```

Nous conservons les notations `Coq` dans la logique de séparation (les prédicats `htrue` et `hfalse` sont respectivement notés `True` et `False`).

Nous définissons une seconde modalité `<pers>` permettant de rendre un prédicat **persistant**. La modalité est définie ainsi : `<pers> P := fun _ ⇒ P ∅` et l'intuition est qu'un prédicat persistant peut être copié autant de fois que nécessaire comme l'énonce la règle suivante :

```
<pers> P ⊢ P ∗ <pers> P
```

Cette règle est respectée car le prédicat `P` est satisfait par l'ensemble vide (d'après la définition de `<pers>`), que l'ensemble vide est neutre par rapport à l'union et qu'il est toujours disjoint à un autre ensemble.

Pour finir, nous définissons l'implication de séparation (que nous notons $-\!\!*$), non-essentielle à la logique de séparation, mais utile pour décrire des spécifications concises. Nous voulons que l'opérateur respecte la règle du modus ponens `P ∗ (P -∗ Q) ⊢ Q` et le définissons ainsi :

```
Definition hwand (P Q : hprop) : hprop := (* P -∗ Q *)
 fun h1 ⇒ ∀ h2, h1 ## h2 ∧ P h2 → Q (h1 ∪ h2).
```

2.1.4 MoSel

Pour manipuler cette logique dans Coq, nous utilisons l'outil MoSel [39]. MoSel fournit un environnement de preuve contenant une riche collection de règles et de tactiques permettant de manipuler les assertions d'une logique **BI** (Bunched Implication [63]). Afin d'obtenir cet environnement, nous instancions notre logique dans MoSel introduisant ainsi un type `iProp` subsumant `hprop`. Lors de l'instanciation, nous devons prouver que la logique respecte les axiomes généraux des logiques **BI**. Nous prouvons que notre logique respecte ces axiomes, mais également quelques autres supplémentaires, permettant à MoSel de nous fournir des tactiques n'étant pas compatibles avec toutes les logiques **BI**.

Les axiomes relatifs aux opérateurs de la logique Coq, que nous avons transportés dans la logique de séparation, sont les règles standards de la logique intuitionniste. Les règles sont présentées Figure 2.1.

Les autres règles concernent le prédicat des propositions pures, les opérateurs de séparation et la modalité persistante.

```
and_elim_l P Q : P ∧ Q ⊢ P
and_elim_r P Q : P ∧ Q ⊢ Q
and_intro P Q R : (P ⊢ Q) → (P ⊢ R) → P ⊢ Q ∧ R

or_intro_l P Q : P ⊢ P ∨ Q
or_intro_r P Q : Q ⊢ P ∨ Q
or_elim P Q R : (P ⊢ R) → (Q ⊢ R) → P ∨ Q ⊢ R

impl_and P Q R : (P ∧ Q ⊢ R) ↔ (P ⊢ Q → R)

∀_intro {A} P (Q : A → hprop) : (∀ a, P ⊢ Q a) → P ⊢ ∀ a, Q a
∀_elim {A} (Q : A → hprop) a : (∀ a, Q a) ⊢ Q a

∃_intro {A} (Q : A → hprop) a : Q a ⊢ ∃ a, Q a
∃_elim {A} (Q : A → hprop) P : (∀ a, Q a ⊢ P) → (∃ a, Q a) ⊢ P
```

FIGURE 2.1 – Règles intuitionnistes

Les règles du prédicat des propositions sont des tautologies usuelles de la logique intuitionniste :

```
pure_intro (Q : Prop) P : Q → (P ⊢ ⌜ Q ⌝)
pure_elim (Q : Prop) P : (Q → True ⊢ P) → (⌜ Q ⌝ ⊢ P)
pure_∀ A (P : A → Prop) : (∀ a, ⌜ P a ⌝) ⊢ ⌜ ∀ a, P a ⌝
```

Nous pouvons remarquer que la règle d'introduction impose au prédicat d'être absorbant. Habituellement dans la logique de séparation [16], le prédicat des propositions pures n'est respecté que par l'ensemble vide. Afin d'être absorbant, nous différons de cette définition usuelle mais pouvons la retrouver avec la définition suivante : `fun P ⇒ ⌜ P ⌝ ∧ emp`.

Une propriété fondamentale de la logique de séparation est que (∗, `emp`) forme un monoïde commutatif :

```
sep_assoc P Q R : (P ∗ Q) ∗ R ⊢ P ∗ (Q ∗ R)
sep_comm P Q : P ∗ Q ⊢ Q ∗ P
emp_sep_r P : P ⊢ emp ∗ P
emp_sep_l P : emp ∗ P ⊢ P
```

Ces règles mettent en avant le fait que la conjonction de séparation décrit un ensemble pouvant être séparé en deux morceaux disjoints **indépendamment** de la construction de l'ensemble décrit. Nous ne nous intéressons qu'aux éléments que les ensembles contiennent. Nous avons une dernière règle spécifique à la conjonction de séparation établissant la monotonicité. En effet, si nous voulons décrire des propriétés sur deux ensembles disjoints, alors ces propriétés sont respectées par l'union des deux ensembles :

```
sep_mono P P' Q Q' : (P ⊢ Q) → (P' ⊢ Q') → (P ∗ P' ⊢ Q ∗ Q')
```

Nous attendons de l'implication de séparation qu'elle ait un comportement avec ∗ similaire à l'implication intuitionniste avec la conjonction non-séparante :

```
wand_sep P Q R : (P ∗ Q ⊢ R) ↔ (P ⊢ Q -∗ R)
```

Les dernières règles concernent la persistance. Tout d'abord, la persistance est monotone. De plus, rendre duplicable un prédicat qui l'est déjà est redondant. Elle est donc également idempotente :

```
persistently_mono P Q : (P ⊢ Q) → <pers> P ⊢ <pers> Q
persistently_idemp P : <pers> P ⊢ <pers> <pers> P
```

Le prédicat d'ensemble emp n'est respecté que par l'ensemble vide et est donc duplicable :

```
persistently_emp : emp ⊢ <pers> emp
```

Représentant les propositions Coq dans la logique, le prédicat pur doit également être duplicable (comme les propositions pures dans le contexte de preuves Coq). Pour cela, il est nécessaire que la modalité persistante soit absorbante (puisque le prédicat pur est absorbant) :

```
persistently_absorbing P Q : <pers> P ∗ Q ⊢ <pers> P
```

Les trois prochaines règles spécifient que de la modalité commute avec les quantificateurs et que, si nous avons deux propriétés duplicables sur un ensemble, alors la conjonction intuitionniste de ces deux propriétés est duplicable :

```
persistently_∃ {A} (P : A → hprop) : <pers> (∃ a, P a) ⊢ ∃ a, <pers> (P a)
persistently_∀ {A} (P: A → hprop), (∀ a, <pers> (P a)) ⊢ <pers> (∀ a, P a)
persistently_and P Q : (<pers> P) ∧ (<pers> Q) ⊢ <pers> (P ∧ Q)
```

Pour éliminer la modalité persistante, nous devons avoir la garantie que l'ensemble respectant le prédicat persistant est l'ensemble vide et que donc l'ensemble respecté par le prédicat ne dépend pas de la modalité ((fun _ ⇒ P ∅) ∅ est égal à P ∅). La règle d'élimination a la définition suivante : <pers> P ∧ emp ⊢ P. Cependant, cette règle et la règle de duplication (<pers> P ⊢ P ∗ <pers> P) découlent d'une règle plus générale :

```
and_sep_elim P Q : <pers> P ∧ Q ⊢ P ∗ Q
```

Intuitivement, cette règle indique que nous pouvons dupliquer P tout en continuant à exprimer des propriétés sur l'ensemble décrit par Q. La règle d'élimination est le cas particulier où Q est emp et la règle de duplication est le cas où Q est <pers> P.

Jusqu'ici, nous avons décrit les opérateurs de notre logique et les règles primitives de ces opérateurs. Notre volonté est d'utiliser cette logique de séparation et ensuite d'en extraire des propriétés intéressantes directement dans Prop. Or, nous pouvons obtenir de telles propositions (notamment grâce à la règle singleton_neq), mais elles restent exprimées à l'aide du prédicat pur et donc dans notre logique de séparation. Afin, de les transporter vers les propositions Coq, nous définissons un dernier opérateur que nous n'utilisons qu'à cette occasion. Cette opérateur (que nous notons && h) décrit simplement sur quel ensemble nous raisonnons :

```
Definition set_ctx (ctx : gset positive) : hprop := fun h ⇒ h = ctx.
```

Nous pouvons alors définir une relation transportant les propositions pures de notre logique de séparation vers les propositions Coq :

```
Lemma soundness_pure h (P : Prop) : (&& h ⊢ ⌜ P ⌝) → P.
```

Ce lemme exprime que si nous prouvons une propriété pure dans la logique de séparation alors peu importe l'ensemble sur lequel nous raisonnons, la propriété est vérifiée dans le monde des propositions Coq. L'utilisation de l'opérateur `&&` est essentielle car il nous permet de généraliser l'assertion à n'importe quel ensemble, tout en excluant les hypothèses compromettantes telles que `False` et ses équivalents.

Nous terminons la présentation de la logique avec une construction fournie par MoSel. La construction que nous notons `[∗ list] x ∈ l, P x` produit un prédicat constitué d'application successives du prédicat `P` sur chacun des éléments de la liste `l` séparé par la conjonction de séparation. Nous présentons alors deux lemmes relatifs à la liste vide et à la concaténation de listes fournis par MoSel à propos de cette construction. Si la liste est vide alors le prédicat produit est équivalent à `emp` :

```
∀ P, ([∗ list] x ∈ nil, P x) ⊣⊢ emp
```

et le prédicat produit par la construction appliquée sur la concaténation de deux listes est équivalent à la conjonction des prédicats produit par l'application sur chacune des listes :

```
∀ P l1 l2,
 ([∗ list] x ∈ l1 ++ l2, P x)
 ⊣⊢ ([∗ list] x ∈ l1, P x) ∗ ([∗ list] x ∈ l2, P x)
```

Notons que le prédicat `NoDuplicate l` aurait pu être défini ainsi : `[∗ list] x ∈ l, & x`.

2.2 Monade libre [❦]

Pour définir les effets, nous reprenons un variant de la monade libre [38, 42]. Le type de données de la monade libre est le suivant :

```
Inductive Free (SIG : Type → Type) (X : Type) : Type :=
| ret : X → Free SIG X
| op : ∀ Y, SIG Y → (Y → Free SIG X) → Free SIG X.
```

Le paramètre `SIG` est la signature des opérateurs primitifs de nos calculs et est appelé "interface". Le type du résultat produit par un opérateur est décrit dans la signature et est représenté par l'indice de `SIG`. Le constructeur `ret` correspond aux calculs purs produisant directement un résultat. Le constructeur `op` appelle un opérateur de l'interface et exploite son résultat à l'aide d'une continuation.

Nous pouvons alors composer des calculs munis d'une même interface en utilisant le combinateur `bind` :

```
Fixpoint bind (m : Free SIG X) (f : X → Free SIG Y) : Free SIG Y :=
  match m with
  | ret v ⇒ f v
  | op e k ⇒ op e (fun v ⇒ bind (k v) f)
  end.
```

En utilisant l'égalité extensionnel, l'opérateur `bind` (que nous notons `let! v := m in f`) et le type de données `Free` respectent alors les lois monadiques :

```
let! r := ret v in        =    k v
k r
-----------------------------------------
let! r := m in            =    m
ret r
-----------------------------------------
                               let! r :=
let! v := m in                  let! v := m in
let! r := e v in          =     e v
k r                            in k r
```

Pour simplifier la manipulation des opérateurs lorsque l'on écrit un programme, nous définissons `gen` permettant de construire génériquement des opérateurs à partir de leur signature :

```
Definition gen (m : SIG X) : Free SIG X := op m (@ret SIG X).
```

Illustrons l'utilisation de la monade libre avec quelques exemples minimalistes.

2.2.1 Erreur

Dans le cas de l'échec, nous avons un opérateur `fail` signalant justement un échec :

```
Inductive Error : Type → Type :=
| Fail : ∀ {X}, Error X.

Definition fail {X} : Free Error X := gen Fail.
```

Nous pouvons décrire la théorie équationnelle axiomatisant le comportement de l'opérateur. Pour ce cas, nous voulons l'équation suivante que nous avons décrit lors de l'introduction :

```
let! v := fail in k v  =  fail
```

En utilisant l'opérateur, nous pouvons alors définir une fonction `check` interrompant le calcul si une condition booléenne n'est pas respectée :

```
Definition check (b : bool) : Free Error unit :=
 if b then ret tt else fail.
```

Il est important de remarquer que l'opérateur `fail` et la fonction `check` ne possède pas d'implémentation. Nous avons seulement décrit syntaxiquement les opérateurs de la monade et décrit leur comportement à l'aide d'équations. Grâce à l'équation de `fail`, nous savons que pour toute implémentation respectant la théorie équationnelle de `Error`, `check` interrompra le calcul si la condition booléenne n'est pas respectée.

2.2.2 État

Pour l'état, nous pouvons définir l'interface suivante :

```
Inductive State (S : Type) : Type → Type :=
| Get : State S S
| Put : S → State S unit.

Definition get : Free (State S) S := gen Get.

Definition put (s : S) : Free (State S) unit := gen (Put s).
```

et définissons les équations des opérateurs ainsi :

```
let! _ := put s in
let! _ := put t in      =    let! _ := put t in
k                            k
-------------------------------------------------
let! _ := put s in
let! t := get in        =    let! _ := put s in
k t                          k s
-------------------------------------------------
let! s := get in
let! _ := put s in      =    k
k
-------------------------------------------------
let! s := get in
let! t := get in        =    let! s := get in
k s t                        k s s
```

Nous pouvons définir un interpréteur (le modèle) de la monade libre instancié avec les opérateurs d'état. Le modèle usuel pour les calculs manipulant un état est la monade d'état :

```
Fixpoint run (m : Free (State S) X) : S → X * S :=
 match m with
 | ret v ⇒ fun s ⇒ (v, s)
 | op Get k ⇒ fun s ⇒ run (k s) s
 | op (Put t) k ⇒ fun _ ⇒ run (k tt) t
 end.
```

2.3 Transformateur de prédicats

Les travaux de Swierstra et Baanen [76] présentent une approche consistant à séparer le calcul de la spécification. Pour cela, les auteurs utilisent la monade libre qui découple la syntaxe de la sémantique et implémente ainsi un transformateur de prédicats basé uniquement sur la syntaxe des programmes. Le transformateur de prédicats présenté pour raisonner sur la monade libre `Free (State S)` est le suivant :

```
Fixpoint statePT (m : Free (State S) X) (P : X * S → Prop) : S → Prop :=
 match m with
 | ret v ⇒ fun s ⇒ P (v, s)
 | op Get k ⇒ fun s ⇒ statePT (k s) P s
 | op (Put t) k ⇒ fun _ ⇒ statePT (k tt) P t
 end.
```

À partir d'un programme et d'une postcondition sur la sortie du programme, `statePT` calcule la précondition nécessaire sur l'état initial pour que la postcondition soit respectée par le programme.

Le transformateur de prédicats `statePT` est une logique de programme et nous permet donc d'avoir un raisonnement prédicatif sur les programmes de la monade libre `Free (State S)`. Cependant, si nous observons attentivement `run` et `statePT`, nous remarquons que `statePT` ne fait que redéfinir, sous forme propositionnelle, le modèle associé à `Free (State S)`. Ainsi, bien que les programmes de la monade `Free (State S)` soient indépendant du modèle, le raisonnement prédicatif permis par le transformateur est directement relié à l'implémentation de la monade d'état.

Nous pensons que le raisonnement prédicatif est une bonne méthode pour raisonner sur un programme. Cependant, le raisonnement prédicatif à partir d'un modèle de la monade libre n'est pas une solution systématique pour raisonner sur les programmes avec effets et notamment sur l'état. Certaines spécifications de programmes peuvent profiter d'une logique de programme plus spécialisée et donc plus adaptée. Particulièrement, lorsqu'une spécification fait appel à une notion de disjonction, nous affirmons que la logique de séparation est une logique de programme idéale. Le transformateur de prédicats a alors la signature suivante, isomorphe à celle de `statePT` :

```
∀ X, Free SIG X → (X → iProp) → iProp
```

2.4 Compositionnalité des effets

Afin de maintenir la compositionnalité des effets dans le raisonnement, comme le permet les équations, des travaux proposent différentes méthodes pour combiner des spécifications prédicatives sur des effets [42, 7]. L'idée est d'indiquer que nous ne manipulons qu'une partie des effets disponibles tout en généralisant autour de la totalité des effets. Pour cela, ces travaux fournissent une spécification aux effets à l'aide d'une précondition et d'une postcondition mais la construction globale diffère.

Baanen et Swierstra [7] redéfinissent la monade libre en la paramétrisant par une liste de signatures. Le constructeur des calculs purs est inchangé tandis que le constructeur `op` prend maintenant un index indiquant la position de l'effet dans la liste :

```
Definition SIG := Type → Type.
```

```
Inductive Free (es : list SIG) X :=
| ret : X → Free es X
| op : ∀ e, e ∈ es → ∀ Y, e Y → (Y → Free es X) → Free es X.
```

Nous pouvons alors définir l'opérateur générique en fournissant la position de l'effet dans la liste :

```
Definition gen (i : e ∈ es) (m : e X) : Free es X := op i m ret.
```

La spécification d'un effet est alors représentée sous forme d'un enregistrement contenant un transformateur de prédicats et une preuve de sa monotonie :

```
Record PT (e : SIG) := {
    pt : ∀ {X}, e X → (X → Prop) → Prop;
    mono : ∀ X (c : e X) (P P' : X → Prop),
     (∀ x , P x → P' x) → pt c P → pt c P' }.
```

La monotonie est une condition assez naturelle du point de vue de la vérification. En effet, si la postcondition que nous voulons respecter est plus forte qu'une autre, il est assez intuitif que la précondition nécessaire le soit aussi.

Pour définir la sémantique de la monade, nous utilisons alors une liste de spécifications que nous paramétrons par la liste des signatures

```
Inductive PTs : list SIG → Type :=
| Nil : PTs nil
| Cons : ∀ e, PT e → ∀ es, PTs es → PTs (e :: es).
```

et d'une fonction `lookupPT` renvoyant la spécification d'un effet à partir de son index dans la liste :

```
lookupPT (pts : PTs es) (i : e ∈ es) : ∀ {X}, e X → (X → Prop) → Prop.
```

La sémantique du variant de la monade libre est alors directe :

```
Fixpoint semFree (pts : PTs es) (e : Free es X) (P : X → Prop) : Prop :=
 match e with
 | ret v ⇒ P v
 | op i m k ⇒ lookupPT pts i m (fun v ⇒ semFree pts (k v) P)
 end.
```

Plutôt que modifier la définition de la monade, Letan et Régis-Gianas [42] exploitent les classes pour mettre en relation des signatures. Ces classes mettent en évidence (à l'aide de fonctions) le fait qu'une signature fournit toutes les opérations d'une autre signature. Ainsi, une notion d'inclusion entre les signatures est introduite et la généralisation sur la signature jouant le rôle d'extension permet d'obtenir la compositionnalité. Par exemple, l'effet générique peut être défini ainsi :

```
Definition gen `{Provide IX I} {X} (m : I X) : Free IX X := op (inj m) ret.
```

La classe `Provide` indique que `IX` est une extension de `I`. La classe fournit les méthodes `inj : I X → IX I` et `proj : IX I → option (I X)` associant les opérations de `I` et de `IX`.

En utilisant ces méthodes associant les opérations de deux signatures, il est alors possible de construire une précondition et une postcondition génériques généralisant les préconditions et postconditions sur les opérations d'un effet particulier aux opérations d'une signature plus large :

```
Definition gen_pre `{Provide IX I} (pre : ∀ {X}, I X → Prop)
 {X} (e : IX X) : Prop :=
  match proj e with
  | Some e ⇒ pre e
  | None ⇒ True
  end.

Definition gen_post `{Provide IX I} (post : ∀ {X}, I X → (X → Prop))
 {X} (e : IX X) (x : X) : Prop :=
  match proj e with
  | Some o ⇒ post o x
  | None ⇒ True
  end.
```

Si l'opération de la signature plus large `IX` correspond bien à une opération de la signature `I` (le cas où `proj e = Some o`) alors nous appelons la pré/postcondition correspondante. Sinon (le cas où `proj e = None`), nous pouvons librement appeler l'opération et ne spécifions pas son résultat. Ainsi, nous pouvons composer des spécifications simplement grâce à la conjonction :

```
Definition prod_pre `{Provide IX I, Provide IX J}
  (preI : ∀ {X}, I X → Prop) (preJ : ∀ {X}, J X → Prop)
  {X} (e : IX X) : Prop :=
  gen_pre preI e ∧ gen_pre preJ e.

Definition prod_post `{Provide IX I, Provide IX J}
  (postI : ∀ {X}, I X → (X → Prop)) (postJ : ∀ {X}, J X → (X → Prop))
  {X} (e : IX X) (v : X) : Prop :=
  gen_post postI e v ∧ gen_post postJ e v.
```

Chapitre 3

Fraîcheur

Ce chapitre est consacré à la génération de symboles frais, un effet classique dans la programmation. Bien qu'il soit aisé de montrer qu'un générateur produit des symboles différents, il est plus difficile de définir une spécification satisfaisante afin de permettre des raisonnements modulaires et locaux.

Hutton et Fulger [32] ont réussi à trouver un exemple minimale illustrant très bien le problème consistant à réétiqueter un arbre binaire. Il faut alors montrer que l'arbre produit ne contient pas deux fois la même étiquette. Plusieurs travaux [29, 76] ont repris cet exemple et ont proposé différentes approches.

Nous commençons par présenter le problème, puis différentes solutions. La première reprend le style du module `SimplExpr` de `CompCert` que nous utilisons comme étude de cas. Nous faisons à nouveau la preuve avec l'approche de Swierstra et Baanen [76] puis avec la nôtre. Pour finir, nous présentons notre approche sur l'étude de cas.

3.1 Réétiquetage [🐓]

```
Inductive Tree (X: Type) :=
| Leaf: X → Tree X
| Node: Tree X → Tree X → Tree X.
```

Le défi consiste à implémenter une fonction `label : Tree X → Tree nat` qui étiquette chaque feuille avec un symbole frais, ici un entier naturel. Pour implémenter cette procédure de réétiquetage en `Coq`, nous sommes naturellement amenés à définir le variant suivant de la monade d'état [60] :

```
Definition Fresh X := nat → X * nat.

Definition ret (x: X): Fresh X :=
  fun n ⇒ (x, n).
Definition bind (m: Fresh X)(f: X → Fresh Y): Fresh Y :=
  fun n ⇒ let (x, n') := m n in f x n'.
Definition gensym (tt: unit): Fresh nat :=
  fun n ⇒ (n, 1+n).

Notation "'do' x '← ' m ';' f" := (bind m (fun x ⇒ f)).
```

Le réétiquetage est alors le programme impératif qu'on aurait écrit dans n'importe quel langage ML :

```
Fixpoint label {X} (t: Tree X): Fresh (Tree nat) :=
  match t with
  | Leaf _ ⇒
    do n ← gensym tt;
    ret (Leaf n)
  | Node l r ⇒
    do l ← label l;
    do r ← label r;
    ret (Node l r)
  end.
```

La fonction `label` est correcte si la structure de l'arbre est préservée et si chaque feuille enregistre un nombre unique. En mettant de côté la question de préservation de la forme de l'arbre, Hutton et Fulger [32] ont proposé la spécification formelle suivante pour cette dernière propriété :

```
Lemma label_spec : ∀ t n ft n',
    label t n = (ft, n') → n < n' ∧ flatten ft = interval n (n'-1).
```

où `flatten` accumule chaque valeur des feuilles durant un parcours de la gauche vers la droite et `interval a b` produit la liste des entiers dans l'intervalle $[a, b]$. Notons que cette spécification est extrêmement prescriptive puisqu'elle exige que `label` numérote consécutivement les feuilles de l'arbre à partir de l'état initial n du générateur de noms frais jusqu'à l'état final n' de gauche à droite.

Il est alors facile de déduire l'absence de doublon, capturé par le prédicat `NoDup` de la librairie standard de `Coq` :

```
Definition relabel (t: Tree X): Tree nat := fst (label t 0).

Lemma relabel_spec : ∀ t ft, relabel t = ft → NoDup (flatten ft).
```

ce qui constitute une interface publique raisonnable à exposer, contrairement à la propriété établie par `label_spec`. La correction du l'étiquetage repose sur notre capacité à prouver `label_spec`. Dans le style de `SimplExpr`, il est évidement possible de traiter `label` comme une fonction pure (puisqu'après tout c'en est une) et donc de manipuler l'encodage fonctionnel de notre variante de la monade d'état. Par exemple, pour raisonner sur la séquence d'opérations, nous utilisons le lemme d'inversion suivant :

```
Remark bind_inversion: ∀ m f y n1 n3,
    (do x ← m; f x) n1 = (y, n3) →
    ∃ v n2, m n1 = (v, n2) ∧ f v n2 = (y, n3).
```

qui réifie, par le biais d'un existentiel, l'état intermédiaire produit entre la première et la seconde opération, nous permettant de raisonner sur des fragments du programme global.

Ici, la preuve se fait par induction sur l'arbre `t`. Par exemple, pour le cas des nœuds, nous avons l'hypothèse suivante

```
(do l ← label t1;
 do r ← label t2;
 ret (Node l r)) n = (t', n')
```

que nous inversons deux fois en utilisant `bind_inversion` révélant alors les états intermédiaires `n2`, `n3` et les résultats intermédiaires `t1'`, `t2'` :

— `label t1 n = (t1', n2)`
— `label t2 n2 = (t2', n3)`
— `Node t1' t2' = t'`
— `n3 = n'`

Nous pouvons alors procéder par induction sur les deux premières hypothèses afin de déduire `flatten t1 = interval n (n2-1)` (avec `n < n2`) d'un côté et `flatten t2 = interval n2 (n3-1)` (avec `n2 < n3`) de l'autre. Les propriétés des intervalles nous autorisent alors à déduire que `flatten (Node t1 t2) = interval n n'`, ce qui établit l'invariant souhaité. La preuve qui en résulte est donc un va-et-vient entre les étapes de raisonnement liées à la structure monadique du programme (par exemple, `bind_inversion` ci-dessus) et des étapes de raisonnement liées aux invariants préservés par le programme (par exemple, la concaténation d'intervalles ci-dessus).

3.1.1 Présentation algébrique [❦]

Afin de découpler la structure monadique (dont le rôle est de séquentialiser les effets) des interprétations spécifiques de cette structure, nous pouvons donner une présentation algébrique des effets [62]. Nous donnons la signature de `gensym`, le seul opérateur de la monade `Fresh`, que nous utilisons dans la construction de la monade libre (Section 2.2) :

```
Inductive Fresh : Type → Type :=
| gensymOp : unit → Fresh nat.
```

Nous définissons une syntaxe pour un langage impératif intégré ainsi qu'un opérateur `gensym`. Pour donner une sémantique au langage, un programmeur `Coq` avide affirmerait qu'un interpréteur est une sémantique dénotationnelle aussi bonne que n'importe quelle autre :

```
Fixpoint eval (m: Free Fresh X): nat → X * nat :=
  match m with
  | ret v ⇒ fun n ⇒ (v, n)
  | op (gensymOp _) k ⇒ fun n ⇒ eval (k n) (1 + n)
  end.
```

Un disciple zélé de Dijsktra (qui pourrait être son petit-neveu [76]) donnerait peut-être une sémantique basée sur les transformateurs de prédicats, en utilisant par exemple un calcul de plus faible précondition :

```
Fixpoint wp (m: Free Fresh X)(Q: X → nat → Prop): nat → Prop :=
  match m with
  | ret v ⇒ fun n ⇒ Q v n
  | op (gensymOp _) k ⇒ fun n ⇒ wp (k n) Q (1+n)
  end.
```

Pour les amener à se mettre d'accord, nous devons prouver l'adéquation des deux sémantiques :

```
Lemma adequacy: ∀ m Q n n' v,
    wp m Q n → eval m n = (v, n') → Q v n'.
```

Bien que nous nous soyons opposés à tout raisonnement directement sur la sémantique des programmes monadiques (ce qui correspond à `eval m` ici), le lemme d'adéquation nous donne l'opportunité de passer à un style plus prédicatif. En particulier, les triplets de Hoare [31], chers aux programmeurs impératifs, peuvent être obtenus avec un simple jeu de notation :

```
Notation "{{ P }} m {{ Q }}" := (∀ n, P n → wp m Q n)
```

à partir de laquelle nous pouvons facilement prouver les règles usuelles de la logique de Hoare [61]

```
Lemma rule_value: ∀ Q v,
    (*-----------------------*)
    {{ Q v }} ret v {{ Q }}.

Lemma rule_composition: ∀ m f P Q R,
    {{ P }} m {{ Q }} →
    (∀ v, {{ Q v }} f v  {{ R }}) →
    (*-------------------------------*)
    {{ P }} do x ← m; f x {{ R }}.

Lemma rule_gensym: ∀ k,
    (*-----------------------------------------------------*)
    {{ fun n ⇒ n = k }} gensym tt {{ fun v n' ⇒ v = k ∧ n' = 1+k }}.

Lemma rule_consequence: ∀ P P' Q Q' m,
    {{ P' }} m {{ Q' }} →
    (∀ n, P n → P' n) →
    (∀ x n, Q' x n → Q x n) →
    (*-----------------------*)
    {{ P }} m {{ Q }}.
```

ou, dit autrement, nous obtenons une représentation de la logique de Hoare dans la logique de Coq.

Bien que, syntaxiquement, le code de label soit inchangé, il n'est maintenant qu'un arbre syntaxique abstrait. Par conséquent, le lemme de correction est naturellement exprimé comme un triplet de Hoare :

```
Lemma label_spec: ∀ t k,
    {{ fun n ⇒ n = k }}
      label t
    {{ fun ft n' ⇒ k < n' ∧ flatten ft = interval k (n'-1) }}.
```

Cette spécification reste insatisfaisante : nous avons encore sur-spécifié le comportement d'un compteur alors que, *in fine*, nous sommes seulement intéressés par `NoDup (flatten t)`. Pour le prouver, nous avons seulement besoin de l'assurance que chaque appel à `gensym tt` produise un entier distinct de tout appel précédent (ce qui est en effet vérifié par une implémentation produisant des nombres consécutifs, mais c'est un détail d'implémentation).

3.2 Preuves monadiques en logique de séparation [🐓]

Équipés d'une logique de séparation (Section 2.1), nous pouvons redéfinir notre calcul de plus faible précondition pour tirer parti de la structure ajoutée. Nous utilisons l'ensemble manipulé implicitement à travers `iProp` pour représenter les entiers générés dans le programme :

```
Fixpoint wp (m: Free Fresh X)(Q: X → iProp): iProp :=
  match m with
  | ret v ⇒ Q v
  | op (gensymOp _) k ⇒ ∀ (v: nat), & v -∗ wp (k v) Q
  end.
```

La spécification des valeurs est similaire à celle du transformateur de prédicats de la section précédente (Section 3.1.1). Pour `gensym`, nous spécifions que l'entier (`v`) renvoyé par l'opérateur est différent de tous les prochains entiers générés. Pour cela, nous utilisons le prédicat singleton et l'implication de séparation. En combinant ces deux connecteurs, nous indiquons à la fois que l'entier produit par `gensym` fait partie des entiers générés par le programme (`& v -∗`) **et** qu'il est différent de tous ceux qui sont générés ultérieurement (`-∗ wp (k v) Q`). Si nous déroulons la définition des opérateurs logiques, ces propriétés sont alors bien visibles :

```
| op (gensymOp _) k ⇒
 fun h ⇒ ∀ v, {[ v ]} ## h → wp (k v) Q ({[ v ]} ∪ h)
```

Nous sommes en train de spécifier l'état du programme après un appel à `gensym`, l'ensemble `h` est donc l'ensemble des entiers générés dans la suite du programme. Nous spécifions l'entier produit en indiquant que pour la suite du programme, nous savons qu'il est différent des prochains entiers générés (`{[ v ]} ## h`) et exprimons qu'il est lui-même un entier généré (`{[ v ]} ∪ h`).

À partir du transformateur de prédicat, nous dérivons naturellement les triplets de Hoare et leur logique associée [16] :

```
Notation "{{ P }} m {{ v ; Q }}" := (P ⊢ wp m (fun v ⇒ Q))

Lemma rule_consequence: ∀ P P' Q Q' m,
  ({{ P' }} m {{ v; Q' v }}) →
  (P ⊢ P') →
  (∀ v, Q' v ⊢ Q v) →
 (*----------------------*)
  {{ P }} m {{ v; Q v }}.

Lemma rule_frame: ∀ P Q R m,
  ({{ P }} m {{ v; Q v }}) →
  (*---------------------------*)
  {{ P ∗ R }} m {{ v; Q v ∗ R }}.

Lemma rule_gensym: {{ emp }} gensym tt {{ ident; & ident }}.
```

tandis que l'énoncé des lemmes précédents `rule_value` et `rule_composition` reste essentiellement inchangé (mais leur signification a changé !). La règle d'encadrement `rule_frame` permet de soustraire temporairement des propriétés (`R` dans l'énoncé) dont la suite du raisonnement est indépendant. Grâce au connecteur ∗, nous exprimons que les entiers décrits dans `R` sont disjoints de ceux décrits dans `Q`. La règle de `gensym` spécifie qu'à partir de rien (`emp`), un appel à `gensym` produit un entier frais. Avec ces deux règles, nous pouvons alors dériver que dans n'importe quel contexte, un appel à `gensym` génère un entier frais :

```
{{ R }} gensym tt {{ ident; & ident ∗ R }}
```

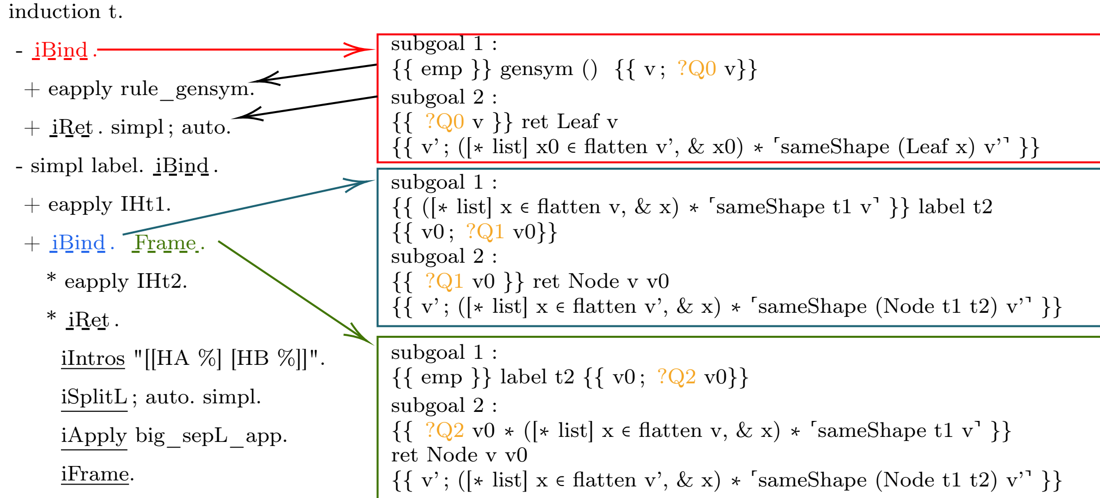
Nous sommes maintenant en mesure de spécifier `label` en exploitant activement la conjonction de séparation :

```
Lemma label_spec_aux : ∀ t,
    {{ emp }}
      label t
    {{ ft; ([∗ list] x ∈ flatten ft, & x) ∗ ⌜sameShape t ft⌝ }}.
```

Par ce passage à la logique de séparation, nous avons déchargé la gestion de la fraîcheur vers la logique, qui nous fournit la règle d'encadrement (`rule_frame`) pour abstraire sur des ensembles disjoints d'identifiants. Contrairement à la preuve de la Section 3.1 où nous devions maintenir un invariant global pendant toute l'exécution du programme, nous pouvons maintenant nous contenter d'un invariant *local*. Ainsi, nous n'avons plus aucun raisonnement arithmétique. Les preuves arithmétiques de `label_spec_aux` sont alors remplacées par des règles élémentaires sur les connecteurs de la logique de séparation, simplifiant la preuve.

Grâce à MoSel, le script de preuve se résume maintenant aux instructions suivantes, qui sont presque intelligibles. L'outil MoSel fournit les tactiques soulignées, que nous avons

étendues avec des tactiques personnalisées (cette fois soulignées avec des tirets) manipulant spécifiquement les triplets de Hoare :



Dans le cas de la feuille, la preuve se résume essentiellement à l'application de `rule_gensym`. La puissance de l'approche se manifeste dans le cas des nœuds, où nous avons accès aux cas récursifs par le biais de la règle de composition, et lorsque chaque preuve est terminée : la règle d'encadrement nous permet de combiner automatiquement les résultats des deux sous-appels.

Cependant, à ce stade, nous n'avons qu'une preuve dans `iProp`, alors que nos utilisateurs s'attendent à une proposition Coq pure, vivant dans `Prop`. Nous pouvons d'abord réduire l'écart entre les deux logiques en montrant que la post-condition non pure de `label_spec_aux` équivaut à une post-condition pure

```
∀ idents, ⊢ ([∗ list] i ∈ idents, & (i: ident)) -∗ ⌜NoDup idents⌝.
```

et, par conséquent, nous obtenons une spécification avec une post-condition pure

```
Lemma label_spec: ∀ t,
    {{ emp }} label t {{ ft; ⌜NoDup (flatten ft) ∧ sameShape t ft⌝ }}.
```

3.2.1 Adéquation entre la logique de programme et l'interpréteur

Le fossé est finalement comblé par un lemme d'adéquation qui relie l'exécution des programmes monadiques avec le générateur défini à 0

```
Definition run (m: Free Fresh X): X := fst (eval m 0).
```

avec des post-conditions pures obtenues dans la logique de séparation

```
Lemma adequacy : ∀ {X} {m: Free Fresh X} {Q},
    {{ emp }} m {{ v; ⌜Q v⌝ }} →
    Q (run m).
```

En corollaire de `label_spec` et de l'`adequacy`, nous obtenons une fonction de réétiquetage utilisable publiquement ainsi qu'une spécification exprimée à un niveau de détail approprié :

```
Definition relabel (t: Tree X): Tree nat := run (label t).

Lemma relabel_spec : ∀ t ft,
    relabel t = ft → NoDup (flatten ft) ∧ sameShape t ft.
```

Pour montrer le lemme `adequacy`, nous devons effectuer des raisonnements arithmétiques afin de prouver que les entiers produits par les appels à `gensym` sont bien frais. Il est important de noter que c'est en effectuant ces raisonnements lors de la preuve d'adéquation, que nous pouvons décharger la gestion de la fraîcheur à la logique de séparation et donc éviter des raisonnements arithmétiques dans les preuves des programmes.

Ainsi, nous prouvons un lemme intermédiaire décrivant explicitement l'ensemble sur lequel nous raisonnons avec notre logique. Pour l'énoncer, nous utilisons une fonction `inject n` produisant un ensemble contenant les entiers de `0` à `n` exclu et prouvons le lemme suivant :

```
Lemma soundness {X} : ∀ (e : Free Fresh X) (Q : X → iProp) n n' v,
    {{ && (inject n) }} e {{ v; Q v }} →
    eval e n = (v, n') →
    && (inject n') ⊢ Q v.
```

Le lemme exprime que si nous raisonnons depuis l'état d'entrée (`&& (inject n)`) et que nous montrons avec la logique de programme que le programme `e` respecte une postcondition `Q` alors la postcondition est respectée par la valeur renvoyée par l'évaluation du programme et l'état de sortie (`&& (inject n')`).

La preuve de `soundness` réside principalement sur le fait que `gensym` renvoie l'état du programme et que l'état croît strictement à chacun des appels. Plus précisément, la valeur renvoyée par l'opérateur n'est pas incluse dans l'ensemble produit par `inject` à partir de l'état d'entrée car elle est plus grande ou égal à l'état d'entrée :

```
∀ n v, v ≥ n → v ∉ inject n.
```

L'adéquation est alors une conséquence directe de `soundness` et de `soundness_pure` (`(&& h ⊢ ⌜ P ⌝) → P` introduit dans la Section 2.1).

3.3 Étude de cas : SimplExpr

Pour évaluer notre approche, nous nous attaquons à un programme certifié pré-existant, le module `SimplExpr` du compilateur certifié CompCert. Ce module implémente une traduction de CompCert-C vers Clight [12]. CompCert-C est le morceau de C formalisé par CompCert et Clight est une version simplifiée de C. La simplification réside sur plusieurs aspects dont l'unification des boucles, la fixation de l'ordre d'évaluation des expressions ou encore la suppression des effets de bord dans les expressions. Ainsi, les expressions dans Clight ne correspondent plus qu'au fragment pur des expressions C, relayant ainsi l'affectation et les appels de fonctions au rang d'instructions.

Pour retirer les effets de bord des expressions, il peut être nécessaire de stocker des résultats intermédiaires. Par exemple, lors de la présence d'un point de séquence (sequence point), la phase de compilation doit s'assurer que tous les effets de bord de la première expression ont été effectués avant le calcul de la seconde expression et stocke ainsi le résultat de la première expression avant le calcul de la seconde. Un autre cas est lors des affectations, le résultat de la partie droite de l'affectation est d'abord stocké laissant la possibilité d'effectuer plusieurs instructions, selon la volatilité du type, avant l'affectation au membre de gauche.

Afin de stocker ces résultats intermédiaires, cette phase de compilation crée des variables temporaires. Les variables temporaires sont des variables locales et ont la particularité de ne pas faire référence à un espace mémoire. Pour éviter des conflits de nommage entre les variables temporaires générées, CompCert prouve qu'elles ont toutes un nom différent. Pour cela, la preuve de la phase de compilation utilise des propriétés arithmétiques similaires à celle de l'exemple de `relabel` dans la Section 3.1 (les noms n'étant en réalité que des entiers).

Nous proposons une comparaison côte à côte de la spécification originale et de la nôtre, en exploitant la logique de séparation (Section 2.1) pour raisonner sur la fraîcheur. Nous commençons par matérialiser la monade dans la Section 3.3.1, sa sémantique dynamique et celle par transformateur de prédicats. Nous examinons ensuite les avantages d'avoir une riche logique d'assertion (Section 3.3.2) pour porter les preuves. Pour finir, nous montrons comment ces propriétés peuvent être traduites et interagir avec les propositions pures Coq (Section 3.3.3), afin d'être utilisables dans la preuve de correction de l'ensemble du compilateur.

3.3.1 La monade [🐓]

Comme pour l'exemple `relabel`, nous nous appuyons sur une description syntaxique de la monade `mon` utilisée par le module `SimplExpr`. Cette monade, qui a été étudiée dans la littérature [75], présente trois opérations : un opérateur `error e`, signalant une erreur `e` pendant l'exécution ; un opérateur `gensym ty`, pour générer des symboles frais associés à un type `ty`, et un opérateur `trail`, afin d'obtenir la liste d'associations des identifiants à leur type construits jusqu'à présent.

En utilisant la construction usuelle, nous définissons l'interface de la monade à partir de laquelle nous dérivons de façon usuelle (Section 2.2) les opérateurs `gensym`, `error` et `trail` :

```
Inductive MON : Type → Type :=
| errorOp : Errors.errmsg → ∀ {X}, MON X
| gensymOp : type → MON ident
| trailOp : unit → MON (list (ident * type)).

Definition mon := Free MON.
```

La sémantique de `mon` est légèrement plus riche que celle de `Free Fresh` (Section 3.1.1). Premièrement, nous devons gérer l'ajout d'une erreur non rattrapable lors de l'exécution.

Nous nous appuyons sur l'implémentation de la monade d'erreur de CompCert

```
Inductive res (A: Type) : Type :=
| OK: A → res A
| Error: errmsg → res A.
```

et nous utilisons essentiellement le transformateur habituel de la monade d'erreur sur la monade d'état afin de maintenir l'état interne de l'opérateur `gensym`. Cependant, contrairement au cas précédent, `gensym` associe désormais les nouveaux identifiants à leur type fourni. Ceci est reflété dans la sémantique, qui maintient une liste d'associations d'`ident` et de type avec le prochain `ident` frais :

```
Record generator : Type := mkgenerator { gen_next : ident;
                                          gen_trail: list (ident * type) }.
```

La sémantique dynamique revient à l'interprétation habituelle des erreurs dans `res` et des opérations à état dans `generator → M (generator * X)` :

```
Fixpoint eval {X} (m : mon X) : generator → res (generator * X) :=
  match m with
  | ret v ⇒ fun s ⇒ OK (s, v)
  | op (errorOp e) _ ⇒ fun s ⇒ Error e
  | op (gensymOp ty) f ⇒
      fun s ⇒
        let h := gen_trail s in
        let n := gen_next s in
        eval (f n) (mkgenerator (n + 1) ((n,ty) :: h))
  | op (trailOp _) f ⇒
      fun s ⇒
        let h := gen_trail s in
        eval (f h) s
  end.
```

La passe du compilateur est exécutée avec un `initial_generator` fourni par le pilote OCaml qui reste opaque à Coq jusqu'à l'extraction :

```
Definition run {X} (m: mon X): res X :=
  match eval m (initial_generator tt) with
  | OK (_, v) ⇒ OK v
  | Error e ⇒ Error e
  end.
```

La sémantique par transformateur de prédicats est alors donnée par un calcul de plus faible précondition :

```
Fixpoint wp {X} (e1 : mon X) (Q : X → iProp) : iProp :=
  match e1 with
  | ret v ⇒ Q v
  | op (errorOp e) _ ⇒ True
  | op (gensymOp _) f ⇒ ∀ l, & l -∗ wp (f l) Q
  | op (trailOp _) f ⇒ ∀ l, wp (f l) Q
  end.
```

où la sémantique de `gensym` suit exactement la définition précédente. La sémantique de `error` ne nécessite aucune précondition (mais, comme nous allons le voir dans le lemme d'adéquation, cela implique que la postcondition est vérifiée seulement si le compilateur ne renvoie pas d'erreur). La spécification de `trail` est volontairement peu engageante : CompCert ne fait aucune hypothèse sur la sortie de `trail` (dans une tournure plutôt élégante, le fait que les identifiants sont tous distincts est une propriété décidable, vérifiée à l'exécution, dans une passe de compilation ultérieure : `trail` est en effet libre de renvoyer n'importe quelle liste d'identifiants, mais CompCert refusera simplement de compiler un morceau de code déclenchant une sortie invalide).

Comme dans la Section 3.1.1, nous dérivons les triplets de FLoyd-Hoare `{{ P }} m {{ v; Q }}` depuis notre calcul de plus faible précondition, ainsi que les règles structurelles habituelles. Les opérateurs spécifiques à la monade sont spécifiés ainsi :

```
Lemma rule_gensym t : {{ emp }} gensym t {{ l; & l }}.

Lemma rule_error P Q e : {{ P }} error e {{ v; Q v }}.

Lemma rule_trail  : {{ emp }} trail tt {{ _; emp  }}.
```

En particulier, l'erreur de l'opérateur équivaut à une "carte de sortie de preuve gratuite", nous permettant de nous décharger de toute post-condition en refusant de faire tout travail.

Nous établissons un lien entre la sémantique dynamique et la sémantique par transformateur de prédicats au moyen d'un lemme d'adéquation

```
Lemma adequacy: ∀ m Q v,
    {{ emp }} m {{ v; ⌜ Q v ⌝}} →
    run m = OK v →  Q v.
```

qui ne prouve la post-condition que lorsque l'évaluation réussit à produire une valeur.

3.3.2 Preuves et Programmes [♞]

La passe de simplification des expressions fait partie du front-end de CompCert. Il se compose de 3 fichiers : `cfrontend/SimplExpr.v` (qui contient les programmes monadiques), `cfrontend/SimplExprspec.v` (qui contient une spécification de type Prolog

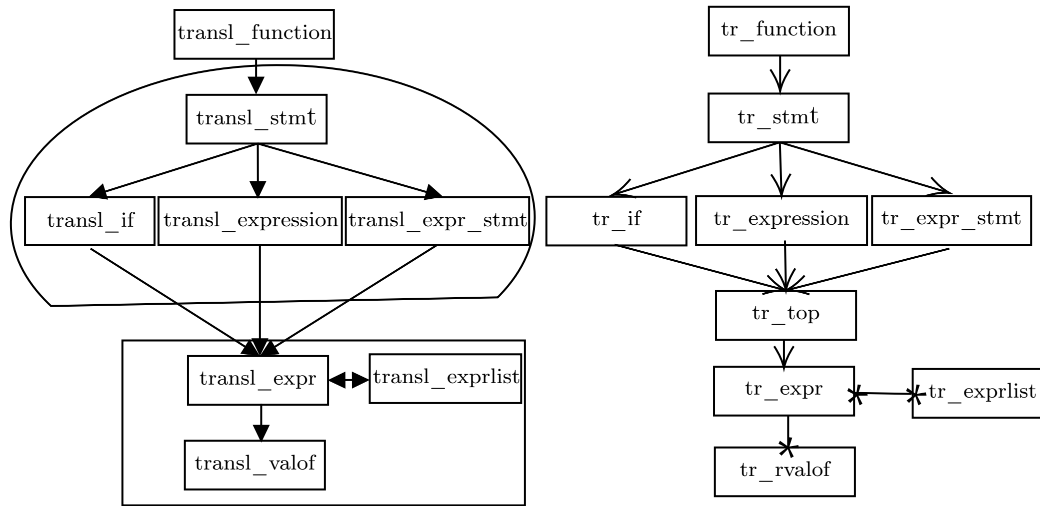


FIGURE 3.1 – graphe d'appel de `SimplExpr` (gauche) et les spécifications correspondantes (droite)

des programmes monadiques par le biais de relations inductives, ainsi que la preuve reliant les programmes monadiques à leur spécification) et `cfrontend/SimplExprproof.v` (qui contient la preuve de la correction de la passe de compilation, exploitant les spécifications relationnelles). Syntaxiquement, `cfrontend/SimplExpr.v` reste inchangé lorsque nous remplaçons la monade original de Leroy par la nôtre : nous avons pris soin d'implémenter la même interface que l'original. Toutefois, la sémantique est très différente : alors que la monade originale construisait un calcul, la nôtre se contente de construire un arbre syntaxique abstrait. Nous devons donc ajouter un appel approprié à `run` pour transformer cette syntaxe en un calcul.

Nous donnons un aperçu du module `SimplExpr` à travers son graphe d'appel (Figure 3.1). La raison d'être de ce module est de définir `transl_function: Csyntax.function → res function` qui effectue la simplification sur les fonctions. C'est le (seul) point d'entrée dans la monade d'erreur `res`. Elle héberge l'appel à `run`. `transl_function` dépend de fonctions opérant dans le fragment `error` et `trail` de la monade, regroupées dnas le cadre circulaire (Figure 3.1). Crucialement, aucune de ces fonctions n'invoque elle-même un générateur de symboles frais. Un troisième groupe de fonctions, toutes utilisées à partir de `transl_expr` et rassemblées dans le cadre rectangulaire (Figure 3.1), est constitué de fonctions qui génèrent de nouveaux symboles et doivent donc appartenir à la monade à part entière `mon`.

Dans la suite, nous présentons plusieurs programmes extraits ou modifiés de CompCert, ainsi que leurs spécifications. Dans ceux-ci, les aspects liés à la fraîcheur des noms sont un moyen de parvenir à un résultat global de correction. Par conséquent, les programmes et les spécifications impliquent une colonne vertébrale d'opérations et de propriétés traitant de la fraîcheur, étoffée par d'autres transformations et propriétés implémentant le processus de compilation souhaité. Afin de mettre en avant les propriétés de fraîcheur, nous adoptons

une astuce typographique : les parties du programme et de la preuve qui n'impliquent pas la fraîcheur ont une police de petite taille. Dans le cadre de notre travail, nous avons été amenés à remplacer les définitions originales de CompCert par de nouvelles définitions : lorsque nous rappelons l'originale, nous l'affichons sur un fond gris pour la distinguer.

Commençons l'exploration du module SimplExpr à partir de transl_expr, qui implique à la fois la génération de noms frais et des erreurs

```
Fixpoint transl_expr (dst: destination) (a: Csyntax.expr) : mon (list statement * expr)
```

Son argument dst peut envelopper, dans le cas de For_set, un identifiant dans la valeur de type set_destination

```
Inductive set_destination : Type :=
  | SDbase (tycast ty: type) (tmp: ident)
  | SDcons (tycast ty: type) (tmp: ident) (sd: set_destination).

Inductive destination : Type :=
  | For_val
  | For_effects
  | For_set (sd: set_destination).
```

Le type destination spécifie comment transmettre le résultat d'une expression donnée, si la contribution d'une expression réside dans sa valeur retournée, ou uniquement dans ses effets de bords, ou dans une variable temporaire dans laquelle sa dénotation a été gardée.

Pour la correction de cette passe d'optimisation, il est crucial que cet identifiant soit frais par rapport à tout identifiant que transl_expr pourrait produire. La fonction transl_expr est définie par filtrage de motif sur l'AST source. Nous nous concentrons ici sur le cas de l'affectation :

```
| Csyntax.Eassign l1 r2 ty ⇒
  do (sl1, a1) ← transl_expr For_val l1;
  do (sl2, a2) ← transl_expr For_val r2;
  do bf ← is_bitfield_access a1;
  let ty1 := Csyntax.typeof l1 in
 let ty2 := Csyntax.typeof r2 in
  match dst with
  | For_val | For_set _ ⇒
    do t ← gensym ty1;
    ret (finish dst
      (sl1 ++ sl2 ++ Sset t (Ecast a2 ty1) :: make_assign bf a1 (Etempvar t ty1) :: nil)
      (make_assign_value bf (Etempvar t ty1)))
  | For_effects ⇒
     ret (sl1 ++ sl2 ++ make_assign bf a1 a2 :: nil,
     dummy_expr)
  end
```

Deux appels récursifs sont effectués avec aucun identifiant frais impliqué. Cependant, lorsque la destination est une valeur (For_val) ou une assignation (For_set), alors il y a un

appel à gensym. La spécification a besoin de refléter le fait que les identifiants générés par les appels récursifs sont tous distincts entre-eux, mais également distincts de l'identifiant potentiellement généré dans la cas de l'assignation. Dans CompCert, ceci est achevé en utilisant explicitement des listes (dans notre cas, tmp, tmp1 et tmp2) d'identifiants générés et en affirmant leur disjonction :

```
Inductive tr_expr: temp_env → destination → Csyntax.expr → list statement → expr →
                   list ident → Prop :=

  | tr_assign_val: ∀ le dst e1 e2 ty sl1 a1 tmp1 sl2 a2 tmp2 t tmp ty1 ty2 bf,
      tr_expr le For_val e1 sl1 a1 tmp1 →
      tr_expr le For_val e2 sl2 a2 tmp2 →
      incl tmp1 tmp → incl tmp2 tmp →
      list_disjoint tmp1 tmp2 →
      In t tmp → ~In t tmp1 → ~In t tmp2 →
      ty1 = Csyntax.typeof e1 →
      ty2 = Csyntax.typeof e2 →
      tr_is_bitfield_access a1 bf →
      tr_expr le dst (Csyntax.Eassign e1 e2 ty)
              (sl1 ++ sl2 ++
               Sset t (Ecast a2 ty1) ::
               make_assign bf a1 (Etempvar t ty1) ::
               final dst (make_assign_value bf (Etempvar t ty1)))
              (make_assign_value bf (Etempvar t ty1)) tmp
```

Pour exprimer dans la précondition sur transl_expr tout identifiant potentiellement dans dst est frais, CompCert introduit le prédicat suivant

```
Definition sd_temp (sd: set_destination) :=
  match sd with SDbase _ _ tmp ⇒ tmp | SDcons _ _ tmp _ ⇒ tmp end.

Definition dest_below (dst: destination) (g: generator) : Prop :=
  match dst with
  | For_set sd ⇒ Plt (sd_temp sd) g.(gen_next)
  | _ ⇒ True
  end.
```

qui, d'une manière très opérationnelle, affirme que les identifiants stockés dans dst sont apparus antérieurement dans l'exécution du générateur de noms frais et sont donc distincts de tout identifiant futur (puisqu'ils sont produits sous forme de nombre consécutif).

Ayant accès à une notion de fraîcheur dans notre langage d'assertions, nous pouvons empêcher la fuite de ces détails opérationnels et simplement affirmer qu'un tel identifiant doit être frais :

```
Definition dest_below (dst: destination) : iProp :=
  match dst with
  | For_set sd ⇒ & (sd_temp sd)
  | _ ⇒ emp
  end.
```

Dans l'approche originale, l'implémentation de `transl_expr` est alors abstraite grâce à la spécification relationnelle donnée par `tr_expr` de la manière suivante

```
Lemma transl_meets_spec:
   (∀ r dst g sl a g' I,
    transl_expr ce dst r g = Res (sl, a) g' I →
    dest_below dst g →
    ∃ tmps, (∀ le, tr_expr le dst r sl a (add_dest dst tmps)) ∧
            contained tmps g g')
```

où `g` et `g'` représentent respectivement l'état du générateur de noms frais au début et à la fin de la transformation. Ils sont nécessaires pour affirmer que tout `ident` dans `dst` est effectivement frais (par `dest_below`) et les temporaires produits par `transl_expr` n'entreront pas en conflit avec une utilisation antérieure ou postérieure du générateur (grâce à `contained tmps g g'`, qui garantit que tous les identifiants dans `tmps` ont été produits entre `g` et `g'`).

Avec notre approche, la fraîcheur des identifiants produits dans les sous-appels et de l'identifiant généré localement est capturé avec des conjonctions de séparation :

```
Fixpoint tr_expr (le : temp_env) (dst : destination) (e : Csyntax.expr)
         (sl : list statement ) (a : expr)  : iProp :=

| Csyntax.Eassign e1 e2 ty ⇒
    match dst with
    | For_val | For_set _ ⇒
    ∃ sl2 a2 sl3 a3 bf  t,
       tr_expr le For_val e1 sl2 a2  *
       tr_expr le For_val e2 sl3 a3  *
       & t ∗
       dest_below dst ∗
       ⌜ tr_is_bitfield_access a2 bf ∧
       sl = sl2 ++ sl3 ++ Sset t (Ecast a3 (Csyntax.typeof e1)) ::
            make_assign bf a2 (Etempvar t (Csyntax.typeof e1)) ::
            final dst (make_assign_value bf (Etempvar t (Csyntax.typeof e1))) ∧
     a = make_assign_value bf (Etempvar t (Csyntax.typeof e1)) ⌝
```

Le type `iProp` n'étant pas primitif à `Coq`, nous ne définissons pas le prédicat sous forme inductive nous permettant ainsi de faire disparaître l'ensemble des identifiants générés de la spécification.

De la même manière, la relation entre `transl_expr` et `tr_expr` est maintenant directe, la contrainte selon laquelle `dst` est frais par rapport aux identifiants produits par `transl_expr` étant naturellement exprimée avec une implication de séparation

```
Lemma transl_meets_spec :
  (∀ r dst,
      {{ emp }} transl_expr ce dst r
      {{ res; dest_below dst -∗ ∀ le, tr_expr le dst r res.1 res.2 }})
```

Grâce à ce processus, nous avons entièrement éliminé le besoin laborieux de suivre l'état opérationnel des générateurs de noms et de maintenir des invariants globaux sur la fraîcheur relative à des fragments de programme. De ce fait, nous avons élevé notre spécification et réussi à la découpler des aspects opérationnels de la génération d'identifiants frais. En prime, nous pouvons maintenant compter sur MoSel pour prouver que notre implémentation répond à sa spécification. En pratique, nous observons que la longueur des scripts de preuve est divisée par deux lors du passage à MoSel, mais nous résisterons à la tentation de tirer toute conclusion d'une métrique aussi peu fiable.

3.3.3 Quitter `iProp` [🐓]

Le raisonnement sur la fraîcheur n'intervient que dans les fonctions situées sous `transl_expr` dans le graphe d'appel. Pour les fonctions (et leur spécification respective) au-dessus de `transl_expr`, l'ensemble des identifiants frais couverts par la spécification est toujours existentiellement quantifié. Puisque, par construction, `iProp` est isomorphe à `gset positive → Prop` (Section 2.1), nous avons intégré cette discipline dans une spécification enveloppante

```
Inductive tr_top: destination → Csyntax.expr → list statement → expr → Prop :=
| tr_top_base: ∀ dst r sl a tmp,
    tr_expr le dst r sl a () tmp →
    tr_top dst r sl a.
```

Par conséquent, les fonctions au-dessus de `transl_expr` n'ont pas besoin de propager l'invariant de fraîcheur. Ainsi, `Prop` est un véhicule suffisant pour écrire leur spécification. Cependant, pour montrer que ces fonctions satisfont leur spécification, nous avons également porter ces preuves à MoSel. Par exemple, dans notre configuration, la fonction `transl_stmt` traduisant les instructions est spécifiée de cette façon :

```
Lemma transl_stmt_meets_spec : ∀ s,
    {{ emp }} transl_stmt ce s {{ res; ⌜ tr_stmt s res ⌝}}
```

qui n'est qu'à un iota de l'original

```
Lemma transl_stmt_meets_spec:
  ∀ s g ts g' I, transl_stmt ce s g = Res ts g' I → tr_stmt s ts
```

Bien qu'il s'agisse d'un changement purement esthétique, cela nous a permis de rationaliser les preuves, qui étaient conçues autour de lemmes d'inversion sur la structure monadique (eux-mêmes enveloppées dans des tactiques). Notons que cet effort n'était pas strictement nécessaire : nous aurions pu garder les définitions préexistantes et leur preuve.

Pour rétablir la preuve de correction globale du compilateur, nous devons rétablir un lemme de simulation reliant le programme source et celui cible. Ce travail s'effectue uniquement sur les spécifications des différentes fonctions (partie droite de la figure 3.1). Au dessus de `tr_top` (inclus), la spécification vit dans `Prop` donc les preuves restent inchangées. Pour `tr_expr`, où la spécification vit dans `iProp`, nous raisonnons en logique de séparation : nous avons donc mis à jour les prédicats originaux afin d'exploiter pleinement les connecteurs de séparation afin de gérer la fraîcheur. Nous réalisons cette partie de la preuve de simulation dans MoSel. Pour combler l'écart entre `iProp` et `Prop`, qui se produit lorsqu'on passe par `tr_top`, nous avons recours à des lemmes tels que `singleton_neq` (Section 2.1) qui traduit les assertions de fraîcheur en faits propositionnels.

3.3.4 Problème de performance

Un problème connu de la monade libre est la performance des programmes. L'inefficacité vient de la construction de la composition qui traverse l'argument de gauche :

```
Fixpoint bind (m: Free SIG X) (f: X → Free SIG Y): Free SIG Y :=
 match m with
 | ret v ⇒ f v
 | op e k ⇒ op e (fun n ⇒ bind (k n) f)
 end.
```

Lorsque nous composons deux opérations monadiques, le terme de gauche est entièrement parcouru afin de reconstruire la syntaxe du programme. Ainsi, similairement à la concaténation de liste (bien que l'opération soit extensionnellement associative), le calcul est plus coûteux dans le cas gauche de l'associativité :

```
bind (bind m k) t = bind m (fun v ⇒ bind (k v) t)
```

En effet, la fonction `bind` va parcourir deux fois `m`. Une première fois lors de la composition de `m` et `k` et une deuxième fois lors de la composition de `bind m k` et `t`. Dans le cas de l'associativité à droite, `m` ne sera parcouru qu'une seule fois.

Pour nos travaux sur CompCert, la construction du `bind` ralentit le compilateur. Un exemple simple de programme C où les performances du compilateur sont fortement impactées est le suivant :

```
int main() {
 int x = 1;
 x = x+++(x+++(x+++(...(x++)...)));
 return x;
}
```

Ce programme (au comportement indéfini) incrémente la variable et additionne sa valeur n-fois. La compilation de ce programme par CompCert génère autant de variable temporaire que d'incrémentation, attribue à chaque variable les valeurs allant de 1 à n, puis les additionne.

La phase de compilation pour ce programme du CompCert-C vers Clight fait appel n-fois à `gensym` et à des opérations négligeables (comme des concaténations de listes où la liste de gauche est de petite taille ou du simple filtrage par motif). Cependant, la profondeur de la récursion est également de n. Or, si nous déroulons la récursion, le programme est de la forme gauche de l'associativité :

```
do v ←
 do t0 ← gensym ty0;
 do k0 ←
  do t1 ← gensym ty1;
  do k1 ←
   (...)
   ret (fn tn)
  ret (f1 k1)
 ret (f0 k0)
ret (f v)
```

où `f`, `f0`, `f1`, ..., `fn` sont les transformations du programme vers Clight.

Dans la version originale de CompCert, cela n'a pas importance et le temps d'exécution n'est donc pas impacté. Dans le cas de la monade libre, nous sommes dans le cas le plus coûteux. Afin de construire l'arbre de syntaxe du programme, le premier appel à `gensym` va être traversé n-fois, le deuxième $n-1$ fois, etc... La construction a donc un coût quadratique.

Ce problème de performance de la monade libre est bien connu, documenté et de nombreuses solutions sont proposées [81, 38, 37, 83, 80]. Globalement, la solution consiste à stocker la suite du programme dans une structure de données plutôt que de reconstruire l'arbre du programme.

Une solution simple dans notre cas est de définir la syntaxe abstraite complète de la monade :

```
Inductive mon {SIG : Type → Type}: Type → Type :=
| ret {X}: X → Free X
| gen {X}: SIG X → mon X
| bind {X Y}: mon X → (X → mon Y) → mon Y.
```

La construction n'est plus la monade libre. Cependant, nous n'exploitons pas la compositionnalité des effets dans `SimplExpr`. De plus, nous avons toujours la séparation entre la syntaxe et la sémantique. Nous pouvons donc avec peu d'effort adapter les travaux à cette construction [🐓]. Maintenant, la séquence est partie intégrante de la syntaxe de nos programmes à travers le constructeur `bind`. Ainsi, plus aucune transformation syntaxique n'est effectuée et le surcoût lié aux transformations syntaxiques a disparu.

Pour évaluer ce gain, nous mesurons le temps de compilation sur des exemples que

nous jugeons peu avantageux pour la monade libre. Ces exemples sont toujours de la même forme : une affectation dont le membre droit est une expression de taille importante où un motif se répète. L'idée est de forcer le compilateur à effectuer des appels récursifs profonds et à appeler `gensym` dans chacun d'eux. Les appels récursifs permettent de s'assurer que le programme effectue la reconstruction de l'arbre la plus coûteuse.

Dans le tableau ci-dessous, nous indiquons le motif, combien de fois il est répété, le temps de compilation du module `SimplExpr` de la version originale de CompCert, de la version avec la monade libre et celle avec la construction contenant le `bind` dans sa syntaxe (version paramétrée) :

Programme	n	original	monade libre	version paramétrée
`x+++(..(x++))`	10.000	0.01s	1.61s	0.02s
`x+++(..(x++))`	20.000	0.03s	11.60s	0.04s
`(((x++)+x++)..)`	10.000	1.22s	3.31s	1.22s
`(((x++)+x++)..)`	20.000	8.05s	26.17s	8.01s
`id(..(id(1)))`	5.000	0.18s	1.02s	0.18s
`id(..(id(1)))`	10.000	0.96s	6.41s	0.97s
`id(..(id(1)))`	20.000	5.67s	43.33s	5.57s

Les deux premières lignes correspondent donc aux programmes C présentés plus tôt dans la section. Chaque compilation a été lancée 10 fois et le temps indiqué est le temps moyen. Nous pouvons remarquer qu'en intégrant la séquence dans la syntaxe et en évitant donc de reconstruire l'arbre des programmes, les performances obtenues sur les mauvais cas se rapprochent fortement de celles du CompCert original.

3.4 Travaux connexes

Très tôt, la théorie des types dépendants a été utilisée pour développer divers modèles de la logique de Hoare [54, 69], dont plusieurs basés sur la logique de séparation [53, 15, 36]. Cependant, ces formalismes ont été introduits pour raisonner sur des *modèles* de programmes impératifs ou concurrents : la théorie des types n'était pas encore reconnue comme un instrument pour écrire des programmes avec effets. CompCert a contribué à montrer que des programmes avec effets non triviaux pouvaient être écrits dans un assistant de preuve. Cela a inspiré les travaux de Swierstra [75], visant à rationaliser et à généraliser la construction de monades d'état indexées introduite par Leroy, spécifiquement dans `SimplExpr`.

La recherche sur les monades de Dijkstra [30, 72, 3, 4, 73], menée par Swamy et ses collaborateurs, a démontré que la programmation avec effets a sa place dans le contexte de la programmation certifiée en F*. Au cours de leur périple, les concepteurs de F* ont montré les avantages d'une approche modulaire des effets (polymonades), chacune étant équipée d'une logique de programme appropriée (monade de Dijkstra) qui - dans certains cas - pourrait être automatiquement dérivée de la monade sous-jacente (en utilisant une interprétation dans la monade de continuation). Cependant, cette ligne de travail exploite activement l'approche du typage de F* basée sur le raffinement (en s'appuyant largement sur un solveur SMT pour décider de la conversion des indices). En l'état, cela ne convien-

drait pas à un assistant de preuve basé sur la théorie des types dépendants, où la conversion n'est pas aussi riche et où le fait de s'appuyer sur des valeurs fonctionnelles au niveau du type rendrait l'expérience pénible. Notre approche est ancrée dans la pragmatique de la programmation indexée dans la théorie des types dépendants et de Coq en particulier. À cet égard, MoSel offre l'environnement de développement ultime pour raisonner (de manière naturelle) sur des programmes efficaces dans Coq.

Avant nous, cette approche a été poursuivie dans le cadre du projet FreeSpec Letan et Régis-Gianas [42] dans Coq. Alors que son champ d'application était limité à la modélisation et au raisonnement sur les interfaces (matérielles), FreeSpec a montré les avantages d'un traitement syntaxique des monades (à travers la construction de monades libres) et la façon de construire des logiques spécifiques à travers des paires pré/post.

Dans Agda, Swierstra et Baanen [76] ont montré comment l'approche de FreeSpec (basée sur les monades libres) et les monades de Dijsktra (dérivant les logiques de programme à partir de monades) pouvaient être combinées de manière fructueuse, produisant une bibliothèque de transformateurs de prédicats qui opère sur le modèle syntaxique de la monade. En étant dans Coq, nous bénéficions également de l'imprédicativité de `Prop` et, par extension, de `iProp`, ce qui nous évite la stratification des univers lors de la définition de la sémantique du transformateur de prédicat.

Alors qu'une grande partie du travail ci-dessus se concentre sur l'émulation d'une certaine forme de logique de Hoare dans la théorie des types, il existe également une ligne de travail parallèle et riche qui mise sur la puissance du raisonnement équationnel pour des programmes avec effets. Gibbons et Hinze [29] ont contribué à illustrer - sur papier - comment utiliser les présentations algébriques des monades pour prouver la correction des programmes implémentés dans celles-ci. En particulier, ils ont revisité le programme `relabel` de Hutton et Fulger et ont donné une preuve purement équationnelle de sa correction. Affeldt et al. [2] reprennent l'approche dans l'assistant de preuves Coq, s'appuyant largement sur SSReflect [85] pour permettre un traitement compositionnel des monades et pour raisonner efficacement sur les programmes monadiques par réécriture.

Interaction Trees Xia et al. [87] sont entre le traitement purement équationnel de Affeldt et al. [2] et le traitement syntaxique de FreeSpec. Tout comme FreeSpec, les Interaction Trees sont construits à partir d'une signature des opérations possibles. Cependant, les auteurs manipulent la version co-inductive permettant de raisonner sur un déploiement infini du graphe de flot de contrôle. L'équivalence de programmes est alors prouvée en établissant une bi-similarité entre deux déroulements : en pratique, on y parvient par un raisonnement équationnel, en substituant des fragments de programme équivalents les uns aux autres.

3.5 Conclusion

Ce chapitre rend compte d'une expérience : utiliser l'une des technologies les plus avancées pour le raisonnement sur les programmes impératifs - la logique de séparation, incarnée par le framework MoSel - pour raisonner sur des programmes monadiques en Coq. Pour évaluer cette approche, nous avons porté le module `SimplExpr` de CompCert et utilisé une

logique de séparation pour le raisonnement sur les noms frais. Notre version de `SimplExpr` est complète et intégrée dans le reste du pipeline du compilateur. La définition de la monade introduit 400 lignes de code supplémentaire [🐓]. Les spécifications et leur preuve passent de 1150 lignes de code à 750 lignes de code [🐓]. Les modifications sur la preuve de correction maintiennent environ 1000 lignes de code [🐓].

Il faut cependant être prudent dans l'interprétation de ces chiffres, car la taille du code n'est qu'un mauvais indicateur de la qualité d'un développement. Il est cependant clair que cette expérience pointe vers le développement d'une bibliothèque intégrée de monades et de leur sémantique opérationnelle (à la FreeSpec [42] et Interaction Trees [87]) ainsi que de leur sémantique par transformateur de prédicats (à la Dijsktra monad [3]), ce qui nous permettrait d'amortir une partie de ces 400 lignes de code supplémentaires.

Une autre métrique pourrait se concentrer autour de la lourdeur des raisonnements. En effet, nous avons remplacé des raisonnements arithmétiques par des raisonnements autour des connecteurs de la logique de séparation. Nous pourrions juger que l'approche est bénéfique si les preuves sont plus simples. Un moyen de comparaison peut être l'évaluation de la puissance des tactiques utilisées. Pour `relabel`, la preuve se basant sur des raisonnements arithmétiques de la Section 3.1.1 fait appel à une procédure de décision pour l'arithmétique linéaire afin d'obtenir une taille raisonnable. Avec la logique de séparation, des règles d'inférences relativement simples suffisent. Pour l'étude de cas sur CompCert, la comparaison semble moins pertinente. En effet, les lemmes arithmétiques nécessaires sont relativement simples et sont regroupés dans une base de données qui est ensuite appelée avec la tactique `eauto`. Le résultat est que les preuves arithmétiques sont automatiques et donc peu encombrantes. Le principal gain dans les preuves de correction de `SimplExpr` est, qu'avec la logique de séparation, il n'est plus nécessaire de raisonner explicitement sur des listes d'identifiants.

Il serait également intéressant d'étudier comment nos preuves se comportent par rapport aux preuves originales lorsque le code sous-jacent évolue. Nous pensons que le style de raisonnement abstrait permis par la logique de séparation offre davantage de possibilités d'automatisation, ce qui devrait faciliter le processus de mise à jour des preuves. La seule expérience effectuée consiste à ajuster nos preuves de la version de CompCert du 4 février 2021 à la version du 03 août 2022. Bien que certaines modifications concernaient des programmes que nous avons spécifiés, les modifications ne généraient aucun nom frais et étaient donc indépendantes de la fraîcheur. Par conséquent, aucun gain évident ou difficulté particulière n'a été constaté. D'autres expériences sont nécessaires pour confirmer ou réfuter cette hypothèse. Une piste pourrait être de reprendre une plus vieille version du module et de refaire son histoire en appliquant successivement les différentes modifications. Il pourrait être alors intéressant d'évaluer l'effort de preuve lors de chaque modification. Des points forts et faibles pourraient alors être mis en évidence.

Chapitre 4

Décodeurs zéro-copie et leur sûreté mémoire

Dans de nombreux domaines tels que le réseau ou les systèmes distribués, les programmes permettant la communication entre différentes entités sont des composants majeurs et sensibles des systèmes. Ces programmes sont critiques car ils traitent potentiellement des données venant du monde extérieur et ne peuvent donc atteindre aucune garantie sur ces données. Ainsi, la moindre erreur peut provoquer des comportements inattendus et donc créer des failles de sécurité. Certaines failles sont très graves car elles permettent d'accéder à des données normalement inaccessibles ou peuvent arrêter le système. L'un des exemples les plus célèbres est Heartbleed [26] où un simple oubli de vérification d'une longueur dans le protocole TLS permettait à une entité (client ou serveur) d'accéder à des données interdites.

Afin d'augmenter la robustesse des programmes traitant des données, de nombreux travaux se sont concentrés sur la génération du code. Des analyseurs comme Lex [41] et Yacc [34] ont été développés. Les utilisateurs doivent décrire comment identifier les données et leur format. Cependant, ces outils sont faits pour décrire des langages conséquents tels que des langages de programmation. La nécessité de produire un analyseur lexical devient alors trop coûteux lorsque l'on veut décrire un simple protocole de communication. De plus, ces outils génèrent des automates pour des langages algébriques, leur expressivité n'est donc pas adaptée pour des protocoles de communication.

Reprenant l'essence de Yacc, des langages de descriptions de données (LDD) [8, 27, 50, 43, 9] ont été développés. Ces langages fournissent alors une collection de types de base faisant office de token dans un analyseur lexical. L'effort consiste alors simplement à décrire un format dans la grammaire du LDD à l'aide des types fournis. Afin d'être simple et intuitif, ces langages proposent une grammaire similaire à celle de C pour décrire des types de données. Les compilateurs de ces langages vont alors produire un décodeur pour le format décrit et toutes les vérifications nécessaires (évitant alors des oublis pouvant avoir de tristes conséquences [26]).

Dans la tradition des décodeurs monadiques [33], une approche pour construire les décodeurs consiste à utiliser des bibliothèques de combinateurs de décodage [40, 22, 64, 47].

Les bibliothèques fournissent une abstraction permettant aux utilisateurs d'oublier les détails d'implémentation. La manipulation du flux de données ou la gestion de certaines erreurs sont alors totalement opaques à l'utilisateur et les seules erreurs possibles provoquées par un utilisateur sont alors algorithmiques. Pour cela, elles fournissent de nombreux combinateurs élémentaires (faisant une tâche très simple et générique) et des opérations permettant de les combiner. Afin de construire un décodeur d'un format complexe, un utilisateur doit alors connaître le rôle des combinateurs fournis par la bibliothèque (sans pour autant connaître leur implémentation) et doit les composer afin d'en produire un plus spécifique répondant à son besoin précis.

Afin d'avoir un contrôle plus fin de la gestion mémoire, certaines bibliothèques sont définies dans des langages bas-niveaux [22, 64]. Ainsi, les bibliothèques peuvent garantir des propriétés mémoires sur les décodeurs produits par les utilisateurs. Les propriétés auxquelles nous nous intéressons sont la sûreté mémoire et l'aspect zéro-copie des décodeurs. Un décodeur est sûr lorsque les accès mémoires qu'il produit, censés donner accès à des données dans le tampon d'entrée, ne dépassent pas les bornes du tampon. Le zéro-copie est une propriété supplémentaire à la sûreté mémoire assurant que tous les accès mémoires donnent accès à des espaces mémoires disjoints. Cette disjonction permet d'éviter des problèmes de corruption mémoire, notamment dans des contextes concurrents. La bibliothèque Nom [22] garantit à l'aide du borrow-checker cette propriété de zéro-copie. Dans ce chapitre, nous formalisons la sûreté mémoire et le zéro-copie puis nous définissons une logique de programme permettant de prouver qu'un décodeur est effectivement zéro-copie. Pour cela, nous définissons une bibliothèque fortement inspirée de Nom sous la forme d'une syntaxe abstraite que nous compilons vers C dans le chapitre suivant.

La chapitre est construit de la manière suivante : nous commençons par présenter l'approche des combinateurs de décodage dans la Section 4.1, puis nous l'adaptons afin d'avoir un contrôle des accès mémoires dans la Section 4.1.1 ; nous formalisons les définitions de sûreté mémoire et de zéro-copie dans la Section 4.2 ; pour illustrer l'approche, nous définissons une logique de programme sur une syntaxe abstraite minimale afin de garantir que les espaces accessibles sont disjoints dans la Section 4.3 ; pour finir, nous définissons une bibliothèque de combinateurs de décodage dans la Section 4.4 et présentons sa logique de programme dans la Section 4.5

4.1 Décodeurs [🐓]

Plaçons nous dans un contexte où nous voulons décoder des données obtenues sous forme binaire et pour lesquelles nous n'avons aucune garantie structurelle. Nous représentons en Coq les octets par les entiers naturels N (non-bornés pour le moment) et représentons les données sous forme de liste :

```
Definition octet := N.

Definition data := list octet.
```

Un décodeur est alors un programme récupérant des données et produisant un résultat à partir de ces données. Cependant, les données reçues peuvent être malformées, c'est-à-dire différer du format attendu. Dans ce cas, notre décodeur doit alors échouer. Par exemple, nous ne recevons pas assez de données pour produire une valeur. Le décodeur peut alors être dans l'impossibilité de produire un résultat. Les décodeurs sont donc des fonctions partielles ayant comme entrée des données à décoder et en sortie une valeur produite à partir de ces données. Nous voulons donc construire des fonctions ayant le type `data → option X` où `X` est le type de la valeur produite. Cependant, un décodeur n'est pas obligé de consommer toutes les données reçues. Afin de permettre leur composition, les décodeurs renvoient également les données non traitées :

```
Definition Decodeur X := data → option (X * data).
```

Le type `option` est le même que celui décrit Section 2.1. Un décodeur de type `Decodeur X` est donc un programme ayant comme état des données à traiter et produisant potentiellement une valeur de type `X`.

Nous implémentons la fonction `decode` appliquant un décodeur sur des données :

```
Definition decode {X} (d : Decodeur X) (l : data) : option X :=
  match d l with
  | None ⇒ None
  | Some (v, _) ⇒ Some v
  end.
```

Après avoir défini un décodeur `d : Decodeur X`, nous obtenons alors la fonction de décodage `decode d : data → option X` qui a l'interface souhaité.

Nous pouvons construire un premier décodeur `decode_pair` renvoyant la paire des deux prochains octets à traiter :

```
Definition decode_pair : Decodeur (octet * octet) :=
  fun d ⇒
    match d with
    | [_] | [] ⇒ None
    | e0 :: e1 :: t ⇒ Some ((e0, e1), t)
    end.
```

Si la liste ne contient pas deux éléments, le décodeur est incapable de produire la paire et échoue. Si les deux octets sont présents, la paire peut alors être formée et renvoyée avec les éléments restant à décoder.

Afin de manipuler la valeur produite par un décodeur sans manipuler les données, nous définissons une fonction `fmap` :

```
Definition fmap (f : X → Y) (e : Decodeur X) : Decodeur Y :=
  fun s ⇒
    match e s with
    | None ⇒ None
    | Some (v, s) ⇒ Some (f v, s)
    end.
```

Nous pouvons alors construire à partir de `decode_pair` des décodeurs renvoyant la somme et le produit des deux prochains octets :

```
Definition decode_add : Decodeur N :=
  fmap (fun v ⇒ fst v + snd v) decode_pair.

Definition decode_mult : Decodeur N :=
  fmap (fun v ⇒ fst v * snd v) decode_pair.
```

En plus d'un gain en modularité, l'utilisation de `fmap` nous assure que la manipulation des données dans les programmes `decode_add` et `decode_mult` se limite aux manipulations effectuées par `decode_pair`. Cette propriété est garantie par les lois des foncteurs que `Decodeur` et `fmap` respectent :

```
fmap id       = id
fmap (f ∘ g) e = (fmap f ∘ fmap g) e
```

Ne pas modifier la valeur décodeur avec `fmap` revient à ne pas modifier le décodeur, et appliquer successivement deux fonctions sur le résultat à l'aide de `fmap` revient à directement appliquer leur composition.

L'intérêt de cette construction reste cependant limité. En effet, le principal avantage des bibliothèques de décodeurs est de pouvoir composer plusieurs décodeurs. Pour composer des décodeurs, nous définissons l'opérateur `<*>` appliquant le résultat du premier décodeur sur le résultat du second :

```
Definition app (f : Decodeur (X → Y)) (x : Decodeur X) : Decodeur Y :=
  fun s ⇒
    match f s with
    | None ⇒ None
    | Some (f,s) ⇒ fmap f x s
    end.

Notation " f '<*>' g" := (app f g)
```

La composition des décodeurs nous permet alors de construire des programmes entièrement définis dans le type de donnée `Decodeur`. Nous devons alors avoir la possibilité d'intégrer des valeurs **pures** dans nos calculs, c'est-à-dire des valeurs indépendantes des données. Pour cela, nous ajoutons l'opérateur `pure` :

```
Definition pure {X} : X → Decodeur X := fun x s ⇒ Some (x,s).
```

Nous pouvons redéfinir le décodeur `decode_pair` en le dérivant à partir du combinateur plus rudimentaire et plus générique `decode_next` renvoyant le prochain octet à décoder :

```
Definition decode_next : Decodeur octet :=
  fun s ⇒
    match s with
    | [] ⇒ None
    | h :: t ⇒ Some (h, t)
    end.
```

Construire la paire des deux prochains octets revient alors simplement à appliquer deux fois la fonction formant des paires sur le résultat de deux appels à `decode_next` :

```
Definition decode_pair : Decodeur (octet * octet) :=
  pure (fun a b ⇒ (a,b)) <*> decode_next <*> decode_next.
```

Les opérateurs `pure` et `<*>` forment une structure `Applicative` [49]. Cette structure, bien connue dans les langages fonctionnels, a été utilisée à de multiples reprises pour définir des bibliothèques de décodeurs [40, 14]. Cependant, bien que cette structure nous permette de composer les décodeurs, elle ne nous permet pas de les séquencer. Cette limitation est visible dans le type de l'opérateur applicatif :

```
<*> : Decodeur (X → Y) → Decodeur X → Decodeur Y.
```

En effet, nous n'appliquons pas le premier décodeur sur le résultat du deuxième, mais le **résultat** du premier décodeur sur le résultat du deuxième. Cette différence est capitale car elle met en évidence le fait que la manipulation des données par les décodeurs ne dépend pas des résultats déjà obtenus. Cette limitation peut être problématique pour décoder certains formats. Par exemple, de nombreux protocoles se basent sur des formats type *tag-length-value* (TLV), c'est-à-dire des formats où la longueur d'un champ est indiquée dans un champ précédent. Nous avons alors besoin de décoder la longueur du champ puis d'utiliser ce résultat pour déterminer le comportement des prochains décodeurs.

Pour permettre cette dépendance, nous construisons l'opérateur de séquence `bind`, que nous notons `let! _ := _ in _`, muni d'un décodeur et d'une continuation. Séquencer deux décodeurs revient simplement à appliquer le premier sur les données puis à fournir le résultat et les données restantes au second :

```
Definition bind (d : Decodeur X) (f : X → Decodeur Y) : Decodeur Y :=
  fun s ⇒
    match d s with
    | None ⇒ None
    | Some (v,s) ⇒ f v s
    end.
```

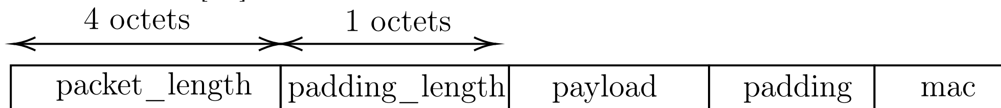
Cette fois-ci, le second décodeur est une continuation (`X → Decodeur Y`). Ainsi, en appliquant le second décodeur sur le résultat du premier, nous permettons aux effets du second décodeur de dépendre du résultat du premier.

La séquence étant une généralisation de l'opérateur applicatif, nous pouvons le dériver :

```
mf <*> mx :=
 let! f := mf in
 let! x := mx in
 ret! (f x)
```

Par cohérence avec les notations usuelles de la séquence, nous ajoutons la notation `ret! v` pour l'opérateur `pure v`.

Pour illustrer ce gain en expressivité, prenons comme exemple le protocole des paquets binaires de **SSH** [45].



Les paquets sont composés de cinq champs. Les deux premiers champs `packet_length` et `padding_length` du format font respectivement 4 et 1 octets. La taille des champs `payload` et `padding` n'est pas connue statiquement. En effet, leur longueur est justement indiquée dans les deux premiers champs. Ainsi pour décoder le paquet, nous devons être capables d'utiliser les valeurs obtenues à partir des deux champs. Le champ `payload` contient les informations du paquet et est de longueur `packet_length` - `padding_length` - 1. Le champ `padding` ne contient aucune information et est de longueur `padding_length`. Le champ `mac` permet de vérifier l'intégrité des données reçues. Nous ne nous intéressons pas à ce dernier champ pour le moment.

Posons nous comme objectif de récupérer le contenu du champ `payload`. Pour connaître la longueur du contenu, nous devons d'abord traiter les données des champs `packet_length` et `padding_length`.

En utilisant la fonction `to_u32 : octet → octet → octet → octet → N` transformant 4 octets en un entier, nous définissons un décodeur récupérant la valeur du champ `packet_length` :

```
Definition decode_packet_length : Decodeur N :=
  pure to_u32 <*> decode_next <*> decode_next <*> decode_next <*> decode_next.
```

Pour récupérer les données de `payload`, nous définissons le combinateur `take n` récupérant les `n` premières données [1] :

```
Equations take (n : N) : Decodeur data :=
  take 0 := ret! [];
  take n :=
    let! v := decode_next in
    let! vs := take (pred n) in
    ret! (v :: vs).
```

Pour récupérer les données du champ `payload`, nous pouvons alors décoder les longueurs encodées puis, à l'aide de la séquence, transmettre le résultat à l'opérateur `take`. Nous devons tout de même vérifier que le paquet ne contient pas d'erreur sémantique, c'est-à-dire que les informations décrites sont cohérentes par rapport aux formats du paquet. Dans le cas des paquets **SSH**, la valeur du champ `packet_length` doit au moins valoir `padding_length` + 1. Si ce n'est pas le cas, le décodage doit échouer. Nous implémentons donc un combinateur échouant en toutes circonstances :

```
Definition fail {X} : Decodeur X := fun _ ⇒ None.
```

1. `N` n'étant pas le type des entiers de Peano (mais une représentation binaire), la récursion n'est pas structurelle. Nous utilisons `Equation` pour montrer que la récursion est bien fondée.

La définition du décodeur est alors directe :

```
Definition decode_payload_SSH : Decodeur data :=
  let! packet_length := decode_packet_length in
  let! padding_length := decode_next in
  if padding_length + 1 <=? packet_length
  then
    let! payload := take (packet_length - padding_length - 1) in
    let! padding := take padding_length in
    ret! payload
  else
    fail.
```

4.1.1 Décodeurs avec pointeurs [❦]

Pour avoir un meilleur contrôle des performances, des bibliothèques de combinateurs sont implémentées à un niveau permettant une gestion plus fine des accès et des allocations mémoires. Les combinateurs ne renvoient alors plus une représentation des données, mais des accès vers des espaces mémoires stockant ces données. Pour éviter que des pointeurs se chevauchent (*aliasing*), c'est-à-dire que plusieurs pointeurs donnent accès à un même espace mémoire et risque donc de corrompre les données, les bibliothèques définissent des combinateurs allouant de nouveaux espaces afin de stocker les données traitées. Cependant, ces allocations ne sont pas forcément nécessaires et peuvent réduire les performances des programmes. Certaines bibliothèques [22, 64] n'allouent pas de nouveaux espaces mémoires et manipulent donc les données sans les recopier. Ces bibliothèques se disent alors "zéro-copie" mais ne donnent pas de définition formelle. En nous appuyant sur leurs comportements, nous proposons une définition formelle du zéro-copie dans la Section 4.2. Nous ajustons alors la définition des combinateurs afin d'être plus fidèle à leur comportement.

Nous définissons les accès à la mémoire à travers des `span`. Un `span` contient deux entiers représentant la position des données dans la mémoire auxquelles il donne accès, et la longueur de ces données :

```
Record span := mk_span {pos : N; len : N}.
```

Pour représenter les accès aux données en mémoire dans nos décodeurs, nous les redéfinissons ainsi :

```
Definition DecodeurM (X : Type) := data → span → option (X * span).
```

Un décodeur prend en entrée une représentation globale des données en mémoire et un `span` donnant accès à l'espace mémoire des données à décoder puis renvoie (sauf en cas d'échec) une valeur **et** un accès mémoire vers le reste des données à traiter.

Le rôle des décodeurs est de déterminer les positions des données dans la mémoire et de générer des accès par l'intermédiaire de `spans`. Toute modification des données lors de la phase de décodage n'a donc pas lieu d'être. La valeur de la position de départ des données,

à laquelle donne accès un `span`, est relative à la représentation globale des données. Par exemple, si nous traitons les données `[0; 1; 2; 3; 4]` et que nous voulons qu'un décodeur traite les données allant de `1` à `3`, nous appliquons le décodeur sur la représentation globale des données en mémoire (`[0; 1; 2; 3; 4]`) puis l'appliquons sur un `span` donnant accès à la mémoire de la position 1 (la position 0 étant celle du premier élément) et de longueur 3 (`mk_span 1 3`).

Nous définissons `decode` nous permettant d'obtenir des fonctions publiquement satisfaisantes des décodeurs :

```
Definition decodeM (d : DecodeurM X) (l : data) : option X :=
  match d l (mk_span 0 (length l)) with
  | None ⇒ None
  | Some (v, _) ⇒ Some v
  end.
```

Nous pouvons alors redéfinir l'opérateur de séquence et celui des valeurs pures tout en conservant les notations `let! _ := _ in _` et `ret! _`. La séquence est similaire à la définition précédente, au détail près qu'au lieu de directement transmettre de nouvelles données, nous transmettons des accès mémoires vers ces données :

```
Definition bind (d : DecodeurM X) (k : X → DecodeurM Y) : DecodeurM Y :=
  fun data l ⇒
    match d data l with
    | None ⇒ None
    | Some (v, s) ⇒ k v data s
    end.

Definition pure (v : X) : DecodeurM X := fun data l ⇒ Some (v,l).
```

Reprenons l'exemple des paquets binaires **SSH**. Avec cette nouvelle méthode de décodage, un décodeur renvoyant les données du champ `payload` ne renverra plus une liste d'octets mais le `span` fournissant l'accès en mémoire. Pour redéfinir `decode_payload_SSH`, nous devons d'abord redéfinir les combinateurs de notre monade. Le combinateur `take n` ne renvoie plus les `n` prochaines données, mais un accès mémoire à ces `n` données. Il doit maintenant vérifier que nous avons accès à au moins `n` données :

```
Definition takeM (n : N) : DecodeurM span :=
  fun _ s ⇒
    if n <=? len s
    then
      let res := mk_span (pos s) n in
      let sres := mk_span (pos s + n) (len s - n) in
      Some (res, sres)
    else
      None.
```

Pour connaître la longueur du champ `payload`, nous devons lire les données stockées en mémoire dans les champs `packet_length` et `padding_length`. Cependant, les données ne sont plus directement manipulées. Nous définissons alors un combinateur `read s n` permettant de récupérer la valeur de la `n`-ème case mémoire à laquelle donne accès le `span s`. Pour définir ce combinateur, nous utilisons la fonction partielle `lookup n (l : data) : option N` renvoyant la valeur à la `n`-ième donnée.

```
Definition readM (s : span) (n : N) : DecodeurM N :=
  fun d t ⇒
    if n <? len s
    then
      match lookup (pos s + n) d with
      | None ⇒ None
      | Some v ⇒ Some (v,t)
      end
    else
      None.
```

À partir de `take` et `read`, nous pouvons alors dériver le décodeur `decode_next` renvoyant la valeur de la prochaine donnée à traiter :

```
Definition decode_next : DecodeurM N :=
  let! s := takeM 1 in
  readM s 0.
```

Pour finir l'implémentation du décodeur, nous redéfinissons `fail` permettant d'échouer :

```
Definition failM {X} : DecodeurM X := fun _ _ ⇒ None.
```

Le décodeur `decode_payload_SSH : DecodeurM span` (quasiment syntaxiquement inchangé) ne renvoie alors plus directement les données, mais un accès vers ces données.

Nous avons donné une représentation aux décodeurs ne manipulant pas directement les données, mais des accès mémoires vers ces données. Nous définissons formellement la sûreté d'un décodeur et la propriété de zéro-copie (Section 4.2). Nous reprenons l'exemple des paquets **SSH** et introduisons une méthode pour montrer l'aspect zéro-copie d'un décodeur (Section 4.3). Finalement, nous formalisons un morceau de la bibliothèque zéro-copie Nom écrite en Rust, et l'accompagnons d'une logique de programme adaptée à la preuve de correction zéro-copie (Section 4.4).

4.2 Sûreté mémoire et zéro-copie [🐓]

La création de langages de descriptions de données a impliqué des études sur les structures utilisées pour représenter les données à l'intérieur des programmes. van Geest et Swierstra [82] définissent récursivement un format comme une valeur ou une paire dépendante. Les valeurs d'un format sont définies comme les valeurs d'un type où il existe un isomorphisme semi-partiel [65, 82] entre les valeurs de ce type et les vecteurs de bit. Un

isomorphisme semi-partiel consiste en une fonction d'encodage, une fonction partielle de décodage et une preuve que décoder une valeur encodée réussit et laisse la valeur inchangée.

```
Definition Bit := bool.

Class ValuesFormat (X : Type) :=
  mk_values {
      size : nat;
      encode : X → Vector.t Bit size;
      decode : Vector.t Bit size → option X;
      spec : ∀ (x : X), decode (encode x) = Some x; }.
```

Dans de nombreux langage de descriptions de données, les enregistrements sont populaires pour représenter des formats [70, 8, 27, 50, 43, 9, 64]. Dans ces langages de descriptions, les enregistrements sont des généralisations des paires dépendantes pour lesquelles chaque champ est nommé. Les valeurs des enregistrements ont des types formant trois groupes :

— Les types de bases définis par le langage et possédant un isomorphisme semi-partiel (généralement implicitement fourni)
— Des pointeurs vers des données comme des tableaux ou des chaînes de caractères
— Des structures définies précédemment dans le langage

Nous formalisons les types du premier groupe comme des instances de la classe `ValuesFormat` et pour le deuxième, nous utilisons les `spans`. Une structure est représentée par une fonction associant une étiquette à un résultat. Les étiquettes sont des valeurs d'un type **fini** où il existe un ensemble `gset` (voir Section 2.1) contenant toutes les étiquettes :

```
Class Etiquette `{Countable etiquette} :=
  mk_etiquette {
      set_etiquette : gset etiquette;
      set_etiquette_spec: ∀ (e : etiquette), e ∈ set_etiquette; }.
```

Nous formalisons alors le résultat d'un décodeur comme une valeur faisant partie de l'un des trois groupes :

```
Inductive Result :=
| Value : ∀ `{ValuesFormat X}, X → Result
| Span : span → Result
| Struct : ∀ `{Etiquette etiquette}, (etiquette → Result) → Result.
```

Le type de données `Result` est donc une généralisation des structures pouvant être renvoyées par un décodeur si nous excluons les valeurs contenues dans la structure qui sont indépendantes des données à décoder. Nous supposons donc que les structures renvoyées par les décodeurs peuvent être transformées en un type de données `Result` isomorphe.

Nous commençons par formaliser la sûreté (Section 4.2.1) puis nous donnons une définition au zéro-copie (Section 4.2.3)

4.2.1 Sûreté mémoire

Un décodeur sûr est une fonction dont les accès mémoires renvoyés, donnant accès à des cases mémoires du message d'origine, ne dépassent pas les bornes du tampon d'entrée.

Pour formaliser cette notion, nous définissons une notion d'inclusion entre les espaces mémoires auxquels donnent accès des `span`s (pour rappel, `inject n m : gset N` est une fonction produisant l'ensemble des entiers de `n` à `m`) :

```
Definition set_span (s : span) := inject (pos s) (pos s + len s).

Definition scope_in (s t : span) := set_span s ⊆ set_span t.
```

L'assertion `scope_in s t` indique que toutes les cases mémoires accessibles depuis `s` le sont aussi depuis `t`.

Le résultat d'un décodeur sûr doit donc respecter le prédicat suivant où `s` est le `span` en entrée et `r` le résultat du décodeur :

```
Fixpoint ResultSafe (s : span) (r : Result) : Prop :=
  match r with
  | Value _ ⇒ True
  | Span v ⇒ scope_in v s
  | Struct ft ⇒ ∀ e, ResultSafe s (ft e)
  end.
```

Dans les cas des valeurs, nous n'avons aucun accès mémoire. Si le résultat renvoyé est un accès mémoire, nous devons nous assurer que les cases accessibles l'étaient déjà. Dans le cas des structures, tous les espaces accessibles dans les champs doivent déjà être accessibles à partir du `span` d'entrée `s`.

4.2.2 Sûreté en pratique

Pour montrer qu'un décodeur est sûr, nous devons montrer que la structure renvoyée (lorsque le décodage réussit) respecte le prédicat `ResultSafe` en toute circonstance. Cependant, le prédicat exprime seulement que tous les `span`s contenus dans la structure donnent accès à un espace inclus dans l'espace accessible depuis le `span` en entrée du décodage. Plutôt que transformer les structures vers `Result`, nous pouvons directement imposer aux structures d'avoir une fonction permettant d'accéder à tous les `span`s qu'elles contiennent.

Pour cela, nous leur demandons d'avoir une instance de la classe `Foldable` [🐓]. La classe `Foldable` représente les structures de données pouvant être parcourues un élément à la fois. La classe contient une fonction `foldMap` construisant la valeur d'un monoïde.

```
Class Foldable (t : Type → Type) :=
  { foldMap : ∀ M `{_ : Monoid M} A, (A → M) → t A → M; }.
```

Les listes forment un monoïde avec la concaténation et la liste vide. Nous pouvons donc implémenter une fonction transformant n'importe quelle structure `Foldable` en une liste :

```
Definition M_to_list `{Foldable M} {X} (m : M X) : list X :=
  Foldable.foldMap (list X) X (fun s ⇒ [s]) m.
```

L'idée est que l'argument de `t` est le type des éléments que nous parcourons. Dans notre cas, nous voulons parcourir les `span`s, les types des structures sur lesquelles nous raisonnons sont donc appliqués sur le type `span`.

Nous définissons alors un prédicat équivalent indiquant que tous les espaces accessibles l'étaient déjà depuis le `span` d'entrée :

```
Definition Result_in `{Foldable M} (r : M span) (s : span) :=
 ∀ v, v ∈ M_to_list r → scope_in v s.
```

Un décodeur est donc sûr si le prédicat `Result_in` est toujours respecté par le résultat du décodage et le `span` en entrée.

4.2.3 Zéro-copie [🐓]

Habituellement, un utilisateur d'une bibliothèque de décodeurs copient les données car il manque d'informations sur les manipulations prévues sur ces données. En effet, ne pas les copier dans un autre espace mémoire serait dangereux car plusieurs accès vers l'espace initial pourraient être créés et risqueraient alors d'engendrer une corruption des données. La méthode usuelle pour s'assurer que nous pouvons nous abstenir de copier les données sans risquer de les corrompre est de garantir que tous les pointeurs manipulés donnent accès à des espaces mémoires disjoints. Deux `span`s donnent accès à des espaces mémoires disjoints si les deux ensembles contenant les positions des cases mémoires accessibles sont disjoints :

```
Definition disjoint (s t : span) : Prop := set_span s ## set_span t.
```

Nous définissons alors un décodeur comme zéro-copie, si tous les `span`s contenus dans le résultat sont deux à deux disjoints. Nous pouvons aisément exprimer cette propriété en utilisant la classe `Foldable` définie dans la Section 4.2.2 :

```
Definition all_disjoint (l : list span) := ∀ s t,
    s ≠ t →
    s ∈ l →
    t ∈ l →
    disjoint s t.

Definition all_disjointM `{Foldable M} (m : M span) : Prop :=
  all_disjoint (M_to_list m).
```

4.2.4 Zéro-copie et Logique de séparation [🐓]

La logique de séparation [66] est idéale pour raisonner à propos des propriétés de disjonction. Nous exprimons alors la propriété de zéro-copie dans notre logique de séparation

(Section 2.1). Pour cela, nous représentons une case mémoire par le prédicat singleton `&`. La proposition `& n` ne peut alors être respectée que par l'espace mémoire ne contenant qu'une seule case et étant à la position `n`. D'après `singleton_neq`, l'assertion `& v ∗ & z` exprime alors que `v` et `z` sont deux cases mémoires distinctes.

Nous représentons l'espace mémoire auquel donne accès un `span` par la conjonction de toutes ces cases mémoires :

```
Equations IsFresh_aux (pos : N) (len : N): iProp :=
  IsFresh_aux pos 0 := emp;
  IsFresh_aux pos len := & pos ∗ IsFresh_aux (succ pos) (pred len).

Definition IsFresh (s : span) := IsFresh_aux (pos s) (len s).
```

Le prédicat de séparation `IsFresh s ∗ IsFresh t` est uniquement respecté par l'ensemble contenant les cases mémoires pointées par les `spans` `s` et `t`. De plus, cet ensemble peut être séparé en deux ensembles disjoints, l'un contenant les cases mémoires accessibles via `s` et l'autre les cases accessibles via `t`. Le prédicat garantit donc que `s` et `t` sont disjoints :

```
Lemma IsFresh_spec : ∀ s t, IsFresh s ∗ IsFresh t ⊢ ⌜disjoint s t⌝.
```

Dans la section précédente, nous avons défini un décodeur zéro-copie comme un décodeur renvoyant une structure où les `spans` sont deux à deux disjoints. Nous exprimons également cette propriété en logique de séparation

```
Definition all_disjointSL (l : list span) := [∗ list] v ∈ l, IsFresh v.

Definition all_disjointMSL `{Foldable M} (m : M span) : iProp :=
  all_disjointSL (M_to_list m).
```

puis montrons que cette nouvelle définition implique bien les propriétés attendues :

```
Theorem all_disjointM_spec `{Foldable M} m :
    all_disjointMSL m ⊢ ⌜ all_disjointM m ⌝.
```

Un décodeur est donc zéro-copie si les `spans` contenus dans la structure renvoyée respectent le prédicat `all_disjointSL`.

4.3 Raisonnement sur les décodeurs en Coq

Dans la section précédente (Section 4.2.3), nous affirmons qu'un résultat est zéro-copie si tous les `spans` qu'il contient sont disjoints. Pour exprimer cette propriété, nous utilisons une fonction construisant la liste des `spans` contenus dans le résultat et montrons qu'ils respectent tous le prédicat `IsFresh` (reliés par la conjonction de séparation).

Maintenant, nous présentons les outils permettant de prouver qu'un décodeur est zéro-copie dans la Section 4.3.1. Nous illustrons l'approche en reprenant l'exemple des paquets binaires **SSH** (Section 4.1) Nous définissons une syntaxe abstraire minimale puis développons une logique de programme spécialisée dans la preuve de zéro-copie. Nous implémentons ensuite un modèle à travers un interpréteur (Section 4.3.2) que nous montrons

adéquate vis-à-vis de la logique de programme. Nous finissons par exposer que la logique de programme permet, en plus du zéro-copie, de montrer qu'un décodeur est sûr.

4.3.1 Preuve de zéro-copie des programmes [🐓]

Pour décoder un paquet **SSH**, nous avons besoin des combinateurs `take`, `read` et `fail`. Le combinateur `take n` renvoie un accès vers les `n` prochains éléments à décoder, `read s n` permet de lire la valeur stockée dans la `n`-ième case mémoire accessible depuis le `span s` et `fail` échoue. Nous pouvons définir la signature des opérateurs et reprenons la définition d'un `octet` comme d'un entier naturel `N` :

```
Inductive DECODE : Type → Type :=
| TakeOp : N → DECODE span
| ReadOp : span → N → DECODE octet
| FailOp : ∀ {X}, DECODE X.
```

Les décodeurs sont alors des programmes définis dans l'interface `DECODE` :

```
Definition Decodeur := Free DECODE.

Definition take (n : N) : Decodeur span := gen (TakeOp n).

Definition read (s : span) (pos : N): Decodeur N := gen (ReadOp s pos).

Definition fail {X} : Decodeur X := gen FailOp.
```

Pour raisonner sur l'aspect zéro-copie de nos programmes, nous définissons une sémantique par calcul de plus faible précondition dans la logique de séparation. Le but de cette sémantique est de décrire les espaces accessibles dans le programme et de garantir leur disjonction :

```
Fixpoint wp {X} (m: Decodeur X) (Q : X → iProp) : iProp :=
  match m with
  | ret v ⇒ Q v
  | op (TakeOp n) k ⇒ ∀ v, IsFresh v -∗ wp (k v) Q
  | op (ReadOp s n) k ⇒ ∀ v, wp (k v) Q
  | op FailOp _ ⇒ True
  end.
```

Nous renvoyons une structure avec des accès mémoires uniquement si le calcul réussit. De plus, lorsque nous appelons `fail`, le calcul est interrompu. Un programme appelant `fail` est donc sûr en toutes circonstances. L'opérateur `read` n'influe pas sur l'accessibilité des données et nous n'attendons aucune propriété particulière sur l'octet lu. L'opérateur `take` permet d'"avancer" dans les données que nous voulons décoder. Son rôle est donc de renvoyer un accès vers des données qui n'ont pas encore été traitées. Nous nous attendons alors à ce que le `span` produit par l'opérateur soit un nouvel accès vers des données qui étaient jusqu'ici inaccessibles dans le programme. De plus, nous nous attendons à ce que les prochains nouveaux accès mémoires soient également vers des données inaccessibles.

L'espace mémoire accessible depuis le `span` renvoyé par `take` doit donc être disjoint des espaces mémoires déjà accessibles et disjoint de ceux qui ne le sont pas encore. Nous formalisons donc cette propriété avec le prédicat `IsFresh` défini Section 4.2.4 et l'implication de séparation. L'implication de séparation exprime que l'espace accessible depuis le `span` et tous les prochains espaces décrits par un prédicat (donc tous les espaces accessibles grâce à un nouvel appel à `take`) sont disjoints.

Il est important de comprendre que la logique de programme présentée se contente d'exprimer des propriétés de zéro-copie. Il est évidemment possible de l'enrichir en ajoutant des propositions pures. Par exemple, nous pourrions ajouter la longueur du `span` produit dans la spécification de `TakeOp n` :

```
| op (TakeOp n) k ⇒ ∀ v, ⌜len v = n⌝ -∗ IsFresh v -∗ wp (k v) Q
```

ou encore, en généralisant la précondition pour qu'elle raisonne sur les données en mémoire, nous pourrions maintenir les informations que nous connaissons sur les valeurs stockées dans les cases mémoires (`lookup n data : option octet` renvoie la donnée de la `n`-ième case) :

```
Fixpoint wp {X} (m: Decodeur X) (Q : X → iProp) : data → iProp :=
fun data ⇒
 (...)
 | op (ReadOp s n) k ⇒
  ∀ v, ⌜lookup (pos s + n) data = Some v ⌝ -∗ wp (k v) Q data
 (...)
```

Cependant, nous nous concentrons sur l'aspect zéro-copie des décodeurs et ne spécifions donc pas ces propriétés pour ne pas polluer le propos.

Par le même jeu syntaxique que dans la Section 3.2, nous obtenons les triplets de Hoare et ses règles usuelles :

```
Lemma rule_ret v H Q:
 (H ⊢ Q v) →
 {{ H }} ret v {{ v'; Q v' }}.

Lemma rule_bind e f Q Q' H :
 {{ H }} e {{ v; Q' v }} →
 (∀ v, {{ Q' v }} f v {{ v'; Q v' }}) →
 {{ H }} do v ← e; f v {{ v; Q v}}.

Lemma rule_consequence P P' Q Q' m :
 {{ P' }} m {{ v; Q' v }} →
 (P ⊢ P') →
 (∀ v, Q' v ⊢ Q v) →
 (*----------------------*)
 {{ P }} m {{ v; Q v }}.
```

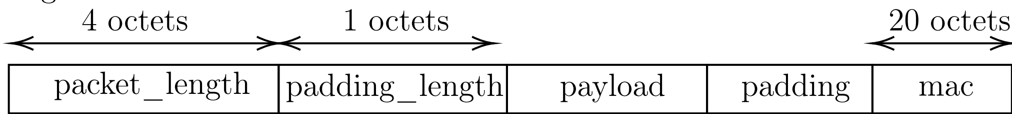
auxquelles nous ajoutons les règles de nos opérateurs :

```
Lemma rule_take n : {{ emp }} take n {{ v; IsFresh v }}.

Lemma rule_read s res : {{ emp }} read s res {{ _; emp }}.

Lemma rule_fail H Q : {{ H }} fail {{ v; Q v }}.
```

Pour rappel, un paquet binaire **SSH** contient 5 champs. Le premier `packet_length` fait 4 octets et indique la taille des trois champs suivants. Le deuxième `padding_length` fait 1 octet et indique la taille du troisième champ. Le troisième `payload` contient les données du message. Le quatrième `padding` rembourre le paquet en ajoutant des octets sans information. Le dernier `mac` permet de vérifier l'intégrité du message, nous supposons sa longueur à 20 octets.



Notre objectif est de produire une structure qui stocke les valeurs des champs `packet_length`, `padding_length` et un accès mémoire vers les champs `payload` et `mac` du paquet. Nous définissons la structure suivante :

```
Record packet_SSHS (S : Type) :=
  mk_ssh {
      packet_length : N;
      padding_length : N;
      payload : S;
      mac : S; }.

Definition packet_SSH := packet_SSHS span.
```

Afin d'instancier la classe `Foldable`, nous abstrayons le type `spans` en argument de la structure et nous définissons la fonction `foldMap` permettant d'appliquer une transformation sur ces `spans` (`Monoid.f` étant la loi de composition interne) :

```
Definition foldMap `{Monoid M} {S} (m : S → M) (p : packet_SSHS S) : M :=
  Monoid.f (m (payload p)) (m (mac p)).
```

Nous voulons alors définir un programme `decode_packet_SSH` qui décode un paquet et produit une structure `packet_SSH` respectant le prédicat `all_disjointSL` :

```
Lemma rule_decode_packet_SSH :
  {{ emp }} decode_packet_SSH {{ v; <absorb> all_disjointMSL v }}.
```

La postcondition est **absorbante** car à la sortie du programme, nous n'avons pas besoin d'assurer que tous les accès à la mémoire sont disjoints, mais seulement ceux renvoyés par le décodeur.

Pour le définir, nous ré-utilisons le décodeur `decode_next` qui est syntaxiquement inchangé (Section 4.1) et lui dérivons une spécification :

```
Definition decode_next : Decodeur N :=
  let! s := take 1 in
  read s 0.
```

```
Lemma rule_next : {{ emp }} decode_next {{ _; True }}.
```

Nous pouvons appeler le décodeur dans n'importe quel contexte. La précondition est donc `emp`. Le combinateur produit un nouvel accès à la mémoire afin de lire la valeur de la prochaine case à décoder. La postcondition doit donc spécifier que de nouveaux accès mémoire ont potentiellement été créés. Par contre, elle n'a pas besoin de les décrire car ils ne sont pas renvoyés et donc inutilisables après l'appel du combinateur.

En ré-utilisant la fonction `to_u32 : octet → octet → octet → octet → N` produisant un entier à partir de 4 octets, nous pouvons définir un combinateur décodant les 4 prochains octets à traiter et les transformant en un entier. Similairement à `decode_next`, le combinateur génère des accès mémoire sans renvoyer les `spans`. Il a donc la même spécification :

```
Definition decode_u32 :=
  let! a := decode_next in
  let! b := decode_next in
  let! c := decode_next in
  let! d := decode_next in
  ret (to_u32 a b c d).

Lemma rule_u32 : {{ emp }} decode_u32 {{ _; True }}.
```

Nous avons maintenant tous les combinateurs nécessaires pour implémenter le décodeur d'un paquet :

```
Definition decode_packet_SSH : Decodeur packet_SSH :=
  let! packet_length := decode_u32 in
  let! padding_length := decode_next in
  if padding_length + 1 <=? packet_length
  then
    let! payload := take (packet_length - padding_length - 1) in
    let! padding := take padding_length in
    let! mac := take 20 in
    ret (mk_ssh packet_length padding_length payload mac)
  else
    fail.
```

Pour s'assurer que le décodeur `decode_packet_SSH` est zéro-copie, nous montrons qu'il respecte sa spécification.

La preuve consiste simplement à utiliser la règle de **séquence** (`rule_bind`) afin de découper la preuve en sous-preuves pour chacun des combinateurs, d'utiliser la règle d'**encadrement** (`rule_frame`) afin de raisonner localement sur chacun de ces combinateurs, puis de montrer que les propriétés obtenues à partir des combinateurs impliquent la postcondition. Nous devons prouver les deux branches de la conditionnelle. La deuxième branche étant triviale, nous ne l'illustrons pas.

```
Definition decodeFM_packet_SSH :=

{{ emp }} précondition

let* packet_length := decode_u32 in

{{ True }} rule_u32

let* padding_length := decode_next in

{{ True }} encadrement + rule_next

let* payload := take (packet_length - padding_length - 1) in

{{ True * IsFresh payload }} encadrement + rule_take

let* padding := take padding_length in

{{ True * IsFresh payload * IsFresh padding }} encadrement + rule_take

let* mac := take 20 in

{{ True * IsFresh payload * IsFresh padding * IsFresh mac }} encadrement + rule_take

{{ <absorb> IsFresh padding * IsFresh mac }} conséquence

ret (mk_ssh packet_length padding_length payload mac).

{{v; <absorb> alldisjointMSL m }} rule_ret

{{v; ⌜ all_disjointM m ⌝ }} conséquence + all_disjointM_spec
```

4.3.2 Modèle de la sémantique [🐓]

Nous avons donné une sémantique par transformateur de prédicats à notre monade libre. Cependant, nous ne lui avons pas encore donner de modèle nous permettant d'effectuer des calculs. Pour cela, nous utilisons la monade `DecodeurM` définie Section 4.1.1 à partir de laquelle nous implémentons un interpréteur :

```
Fixpoint eval (m : Decodeur X) : DecodeurM X :=
  match m with
  | ret v ⇒ ret! v
  | op (TakeOp n) k ⇒
      let! v := takeM n in
      eval (k v)
  | op (ReadOp s pos) k ⇒
      let! v := readM s pos in
      eval (k v)
  | op FailOp _ ⇒
      failM
  end.
```

Nous pouvons alors définir la fonction `decode` appliquant un décodeur sur des données :

```
Definition decode (d : Decodeur X) (data : data) : option X :=
  match eval d data (mk_span 0 (length data)) with
  | Some (v, _) ⇒ Some v
  | _ ⇒ None
  end.
```

Nous définissons alors un théorème garantissant que les deux sémantiques sont en adéquation. Nous montrons donc que, si nous prouvons une propriété sur un programme avec la logique de séparation, alors l'interprétation du programme renvoie une valeur respectant la propriété. De plus, ce théorème est essentiel car il permet de faire le lien entre la logique de séparation et la logique des propositions `Coq`. Nous pouvons alors à la fois avoir des propriétés dans `Prop` sur nos décodeurs et raisonner localement avec la logique de séparation.

```
Theorem adequacy : ∀ d (Q : X → Prop),
  {{ emp }} d {{ v; ⌜Q v⌝ }} →
  ∀ data v,
   decode d data = Some v →
   Q v.
```

4.3.3 Preuve de sûreté

Nous savons par construction du modèle que les `span`s renvoyés par `take` donnent accès à des espaces mémoires déjà accessibles par le `span` en entrée. Pour montrer que le décodeur

est sûr, nous pouvons donc simplement nous assurer que tous les spans ont été générés par un appel à take. Or, dans notre logique de programme, seuls les spans obtenus par un appel à take respectent le prédicat IsFresh. Nous pouvons donc en déduire que si nous montrons que tous les spans dans le résultat respectent le prédicat IsFresh alors le décodeur est sûr. Or, nous montrons déjà que tous les spans respectent le prédicat IsFresh lorsque nous prouvons que le résultat respecte le prédicat all_disjointMSL. Nous prouvons donc simultanément la sûreté mémoire et le zéro-copie. Nous formalisons cette propriété dans notre développement ainsi :

```
Lemma eval_safe `{Foldable M}: ∀ e s,
 {{ emp }} e {{ res; <absorb> all_disjointMSL res }} →
 ∀ (data : data) (res : M span) s_res,
  eval e data s = Some (res, s_res) →
  Result_in res s.
```

4.4 Formalisation de Nom

Nom [22] est une bibliothèque de combinateurs écrite en Rust [46]. Son objectif est de fournir les outils permettant à un développeur Rust d'écrire facilement des décodeurs, zéro-copie si possible, sûrs et rapides. Pour garantir la sûreté des décodeurs zéro-copie, Nom utilise le borrow checker de Rust qui soulèvera une erreur si plusieurs accès mutables donnent accès à une même case mémoire.

Nous formalisons les décodeurs fournis par la bibliothèque tout au long de la Section 4.4. L'objectif est de fournir un modèle Coq le plus fidèle possible à la bibliothèque Nom. Pour cela, nous définissons une syntaxe abstraite, dans la Section 4.4.2, représentant des combinateurs élémentaires que nous utilisons pour implémenter les différents combinateurs que fournit la bibliothèque Nom. Nous spécifions sous forme équationnelle les combinateurs dans la Section 4.4.3 et donnons une sémantique opérationnelle dans la Section 4.4.4. À partir de cette syntaxe, nous pouvons écrire des décodeurs et pouvons les tester à l'aide de la sémantique opérationnelle. Nous abordons la question de l'efficacité dans le Chapitre 5 où nous compilons les décodeurs. Dans le Section 4.5, nous développons une logique de programme permettant de garantir la propriété de zéro-copie des programmes définis avec les combinateurs de Nom. Similairement à l'exemple minimale de la Section 4.3.1, nous pouvons aussi utiliser la logique de programme afin de montrer qu'un décodeur est sûr. Avant de décrire les combinateurs de la bibliothèque, nous définissons plusieurs types de données utiles pour la formalisation.

4.4.1 Types de bases supplémentaires

Nous avons déjà défini les spans (Section 4.1.1) lors des exemples précédents. Afin d'être plus fidèles à certaines valeurs manipulées dans Nom, nous définissons d'autres types de données.

Entiers bornés [🐓]

Les entiers retournés par les décodeurs ne peuvent pas être infinis comme les définitions Coq des entiers le suggèrent. Nous définissons donc des entiers bornés à l'aide d'un simple enregistrement paramétré par la "longueur" en binaire de l'entier. L'enregistrement stocke la valeur de l'entier et une preuve qu'il est inférieur à sa borne.

```
Record natN (len : N) : Set :=
  mk_natN { val : N ; len_correct : val < 2 ^ len}.
```

Nous définissons une fonction _ ↑ _ concaténant deux entiers binaires de même longueur :

```
n ↑ m : ∀ p, natN p → natN p → natN (2 * p).
```

Vecteurs [🐓]

Les vecteurs (ou tableaux dynamiques) sont représentés par un enregistrement contenant la taille, la capacité et les valeurs du vecteur. Les valeurs sont caractérisées par une map `values A` associant une position à une valeur :

```
Record vector A :=
 mk_vector { capacity : N; size : N; values : values A }.
```

Nous formalisons un vecteur bien formé, dans une structure `Vector : ∀ A, vector A → Prop`, par deux propriétés :

1. La taille d'un vecteur est la position + 1 de la valeur à la position la plus grande. Sa taille est 0 s'il y a aucune valeur.
2. La taille d'un vecteur est au plus sa capacité.

Tous les vecteurs manipulés doivent être bien formés :

```
Definition VECTOR A := {vec : vector A | Vector vec}.
```

Nous construisons les opérations `make`, `set`, `get`, `add` permettant respectivement de créer un vecteur, changer une valeur, récupérer une valeur et ajouter une valeur au vecteur.

```
make : ∀ A, N → VECTOR A
set  : ∀ A, VECTOR A → N → A → VECTOR A
get  : ∀ A, VECTOR A → N → option A
add  : ∀ A, VECTOR A → A → VECTOR A
```

4.4.2 Les combinateurs [🐓]

Pour formaliser la bibliothèque Nom, nous ré-définissons les opérateurs `take`, `read` et `fail` (Section 4.3.1). Nous avons également un opérateur `length` renvoyant le nombre de données restant à traiter. Nous ajoutons un opérateur `alt d1 d2` faisant appel au décodeur `d2` si le décodeur `d1` échoue. Nous utilisons l'opérateur `scope s d`, où `s` est un `span` et

d un décodeur. Le décodeur permet de traiter les données accessibles depuis s à l'aide du décodeur d. Nous définissons ensuite le décodeur peek d renvoyant la valeur produite par le décodeur d, mais sans consommer de données. Pour finir, nous construisons deux décodeurs permettant de boucler. Le premier repeat_n n d b, où n est un entier, d un décodeur unaire et b une valeur. Le décodeur fait appel n-fois au décodeur d. Le premier appel est appliqué sur la valeur de départ b et les prochains sur la valeur produite par l'itération précédente. Si un appel au décodeur d échoue, alors repeat_n n d b échoue également. Le second est many d b. Cet opérateur va appeler le décodeur unaire d jusqu'à son échec. Lorsque le décodeur échoue, la valeur de l'itération précédente est alors renvoyée.

La bibliothèque Nom peut être utilisée pour décoder des formats binaires, des formats textes ou bit par bit. Nos opérateurs sont indépendants de ce niveau de détail et sont, par conséquent, paramétrisés par un type atom. La signature des opérateurs est la suivante :

```
Context {atom : Type}.

Inductive NOM : Type → Type :=
| FAIL : ∀ X, NOM X
| LENGTH : NOM N
| READ : span → N → NOM atom
| TAKE : N → NOM span
| ALT : ∀ X, Free NOM X → Free NOM X → NOM X
| LOCAL : option span → ∀ X, Free NOM X → NOM X
| REPEAT : option N → ∀ X, (X → Free NOM X) → X → NOM X.
```

Les opérateurs scope et peek sont dérivés à partir du constructeur LOCAL et repeat_n et many à partir de REPEAT.

Sous-termes en Coq

Bien que la définition NOM soit légale en Coq, le typeur Coq ne reconnaît pas les arguments des constructeurs de type Free NOM X comme des sous-termes lorsque nous implémentons des fonctions récursives par filtrage de motif. Par conséquent, le typeur n'est pas capable d'identifier que la récursion est structurelle (et donc que la fonction termine).

Avec la présence d'une continuation dans la syntaxe de la monade libre, nous ne pouvons pas définir de fonction calculant la taille d'un arbre de type Free NOM X. En effet, nous devons pour cela traverser la continuation mais le seul moyen d'interagir avec une fonction en Gallina est de l'appeler. Or, nous n'avons pas de valeur à lui donner. Il n'est pas possible de montrer à l'aide d'une récursion bien fondée sur les entiers naturels que l'arbre décroît lorsque nous définissons une fonction récursive.

Une autre approche [79, 44] consiste à enrichir l'hypothèse d'induction de types inductifs imbriqués. Pour cela, l'hypothèse d'induction d'un type de données portent des propriétés sur le type imbriqué à l'aide d'une traduction paramétrique unaire. Il est possible d'enrichir l'hypothèse d'induction de Free et NOM en utilisant cette approche mais cela complique la définition des fonctions et notamment celle qui sont mutuellement récursives sur les types de

données imbriqués. Nous laissons donc l'utilisation de cette approche dans les travaux futurs et produisons simplement la syntaxe à partir de deux inductifs mutuellement inductifs. L'un est la signature présentée ci-dessus et l'autre `NomG` est le variant de la monade libre présenté Section 2.2 :

```
with NomG : Type → Type :=
| ret : ∀ X, X → NomG X
| op : ∀ Y, NOM Y → ∀ X, (Y → NomG X) → NomG X.
```

Nous ré-implémentons donc les opérations liées à cette syntaxe et maintenons les notations usuelles. Cependant, les définitions ne sont pas de type `Free NOM`, mais de type `NomG` !

Dérivation des combinateurs

Depuis cette syntaxe, nous dérivons les combinateurs présents dans la bibliothèque Nom [21], à l'exception des combinateurs `tuple`, `permutation l` et certains permettant de reconnaître des caractères.

Des combinateurs ont des versions adaptées au streaming. Concrètement, ces combinateurs signalent (sous forme d'erreur) le nombre de données manquantes à la place d'un autre message d'erreur. Nous n'avons pas étendu les erreurs pouvant être soulevées et ne gérons donc pas cet aspect.

Nom présente plusieurs modules de combinateur dans sa documentation. Pour chaque module, nous indiquons dans une figure la signature des combinateurs dans notre formalisation et donnons un lien vers la documentation. Certains décodeurs ont besoin de comparer la valeur de plusieurs données. Dans ce cas, nous instancions `atom` aux entiers de taille 8 (`NomB := @NomG nat8`). Dans Nom, les combinateurs sont évidement plus génériques. En effet, plutôt qu'instancier à un type précis, la bibliothèque Rust utilisent un trait proche de la classe `ValuesFormat` de la Section 4.2.

Nous avons dans la Figure 4.1 les combinateurs élémentaires (lien vers la doc) [🐓]. Les combinateurs permettant de composer des décodeurs sont dans la Figure 4.2 (lien vers la doc) [🐓]. Les combinateurs permettant de répéter des décodeurs sont dans la Figure 4.3 (lien vers la doc) [🐓]. Les combinateurs permettant d'analyser des décodeurs sont dans la Figure 4.4 (lien vers la doc) [🐓]. Les combinateurs permettant de reconnaître des nombres sont dans la Figure 4.5 (lien vers la doc) [🐓]. Nous ne traitons ni les flottants ni les entiers signés.

```
all_consuming : NomG X → NomG X.
cond : bool → NomG X → NomG (option X).
consumed : NomG X → NomG (span * X).
eof : NomG unit.
fail : NomG X.
flat_map : NomG X → (X → NomG Y) → NomG Y.
into : NomG  X → (X → Y) → NomG Y.
map : NomG  X → (X → Y) → NomG Y.
map_opt : NomG X → (X → option Y) → NomG Y.
map_parser  : NomG span → NomG X → NomG X.
not : NomG X → NomG unit.
opt : NomG X → NomG (option X).
peek : NomG X → NomG X.
recognize : NomG X → NomG span.
rest : NomG span.
rest_len : NomG N.
success : X → NomG X.
value : X → NomG Y → NomG X.
verify : NomG X → (X → bool) → NomG X
```

FIGURE 4.1 – Combinateurs élémentaires

```
delimited : ∀ X Y Z, NomG X → NomG Y → NomG Z → NomG Y.
pair : ∀ X Y, NomG X → NomG Y → NomG (X * Y).
preceded : ∀ X Y, NomG X → NomG Y → NomG Y.
separated_pair : ∀ X Y Z, NomG X → NomG Z → NomG Y → NomG (X * Y).
terminated : ∀ X Y, NomG X → NomG Y → NomG X .
```

FIGURE 4.2 – Combinateurs de séquence

```
count : N → NomG X → NomG (VECTOR X).
fill : VECTOR X → NomG Y → NomG (VECTOR Y).
fold_many0 : ∀ X Y, NomG X → Y → (Y → X → Y) → NomG Y.
fold_many1 : ∀ X Y, NomG X → Y → (Y → X → Y) → NomG Y.
fold_many_m_n : ∀ X Y, N → N → NomG X → Y → (Y → X → Y) → NomG Y.
length_count : ∀ X, NomG N → NomG X → NomG (VECTOR X).
length_data : NomG N → NomG span.
length_value : ∀ X, NomG N → NomG X → NomG X.
many0 : ∀ X, NomG X → NomG (VECTOR X).
many0_count : ∀ X, NomG X → NomG N.
many1 : ∀ X, NomG X → NomG (VECTOR X).
many1_count : ∀ X, NomG X → NomG N.
many_m_n : ∀ X, N → N → NomG X → NomG (VECTOR X).
many_till : ∀ X Y, NomG X → NomG Y → NomG (VECTOR X * Y).
separated_list0 : ∀ X Y, NomG X → NomG Y → NomG (VECTOR Y).
separated_list1 : ∀ X Y, NomG X → NomG Y → NomG (VECTOR Y).
```

FIGURE 4.3 – Combinateurs de répétition

```
is_a : string → NomB span.
is_not : string → NomB span.
tag : string → NomB span.
tag_while : (nat8 → bool) → NomB span.
take_till : (nat8 → bool) → NomB span.
take_till1 : (nat8 → bool) → NomB span.
take_until : string → NomB span.
take_until1 : string → NomB span.
take_while : (nat8 → bool) → NomB span.
take_while1 : (nat8 → bool) → NomB span.
take_while_m_n : N → N → (nat8 → bool) → NomB span.
```

FIGURE 4.4 – Combinateurs d'analyse

```
be_u8/16/32/64/128 : NomB nat8/16/32/64/128.
le_u8/16/32/64/128 : NomB nat8/16/32/64/128.
```

FIGURE 4.5 – Combinateurs des nombres

4.4.3 Théorie Équationnelle [🐓]

Une façon naturelle de spécifier les opérateurs est de fournir une théorie équationnelle [29, 2, 1]. Contrairement aux modèles via un interpréteur (nous en présentons un dans la section suivante), nous n'avons pas ici à nous préoccuper de problème de terminaison et gagnons donc en simplicité.

Tout d'abord, les opérateurs `bind` et `ret` modélisent respectivement l'identité et la composition séquentielle. Les deux opérations doivent donc satisfaire les trois lois monadiques [51]. Les opérateurs `bind` (noté `let!`) et `ret` forment un monoïde :

```
let! v := ret v in   =   f v
f v

let! v := e in       =   e
ret v

let! v :=
 let! r := e in          let! r := e in
 f r in              =   let! v := f r in
g v                      g v
```

L'opérateur `fail` interrompt le calcul et doit donc être un élément absorbant à gauche de la composition séquentielle [29] :

```
let! _ := fail in e  =  fail
```

L'idée de `alt m f` est d'exécuter `m`, mais de donner le contrôle à `f` si une erreur est levée lors de l'exécution de `m`. `alt` et `fail` forment donc un monoïde [29] :

```
alt fail e        =  e
alt e fail        =  e
alt e (alt m k)   =  alt (alt e m) k
```

Les opérations ne pouvant pas échouer sont absorbantes à gauche de `alt` :

```
alt (ret v) e   =  ret v
alt length e    =  length
```

Les opérateurs `length` et `read` permettent de récupérer des informations sur l'état du programme sans le modifier. Ils sont donc idempotents et commutents :

```
let! v := length in                let! v := length in
let! r := length in        =       k v v
k v r

let! v := read s n in              let! v := read s n in
let! r := read s n in      =       k v v
k v r

let! v := read s n in              let! w := read t m in
let! w := read t m in      =       let! v := read s n in
k v w                              k v w

let! v := read s n in              let! len := length in
let! len := length in      =       let! v := read s n in
k v len                            k v len
```

L'opérateur `read` récupère des informations indépendantes des prochaines données à décoder. L'opérateur n'interfère donc pas avec `take`. Ils peuvent donc commuter :

```
let! v := read s n in              let! s := take n in
let! s := take n in        =       let! v := read s n in
k v s                              k v s
```

L'opération `length` permet de connaître le nombre de données restant à traiter et `take n` permet de récupérer les `n` prochaines données. Il est donc possible de les commuter à l'aide de l'addition ou de la soustraction d'entiers :

```
let! len := length in              let! v := take n in
let! v := take n in        =       let! len := length in
k len v                            k (len + n) v

let! v := take n in                let! len := length in
let! len := length in      =       let! v := take n in
k len v                            k (len - n) v
```

L'opérateur `local` permet d'utiliser un décodeur localement sur des données, sans modifier l'état courant du programme. L'opération est donc l'identité lorsqu'elle est appliquée sur des opérations indépendantes des données à décoder :

```
local o (ret v)      =   ret v
local o fail         =   fail
local o (read s n)   =   read s n
```

Forcer un décodeur à ne pas modifier l'état courant, alors qu'il est déjà indépendant de l'état courant, revient également à appliquer l'identité :

```
local o (scope s e)  =  scope s e
```

L'opération `peek` (qui comme `scope` est un cas particulier de `local`) permet de s'assurer qu'un décodeur ne modifie pas l'état courant du programme, tout en l'appliquant quand même sur l'état courant. S'assurer qu'un décodeur ne modifie pas l'état courant est idempotent :

```
local o (peek e)  =  local o e
```

De plus, `length` ne manipule pas les données. Donc l'opérateur `peek` appliqué à `length` est l'identité :

```
peek length  =  length
```

L'opération `local` est distributive par rapport à `alt` :

```
local o (alt e1 e2)  =  alt (local o e1) (local o e2)
```

Nous terminons par les équations relatives à `repeat`. Répéter autant de fois que possible l'échec, appliquer 0 fois un décodeur ou un nombre de fois fixe `ret`, revient à renvoyer la valeur de départ :

```
repeat_n 0 e v          =  ret v
repeat_n n ret v        =  ret v
many (fun _ ⇒ fail) v   =  ret v
```

Nous pouvons déplier une itération des boucles :

```
repeat_n (succ n) e b  =  let! v := e b in
                          repeat_n n e v

many e b               =  alt (let! v := e b in many e v) (ret b)
```

4.4.4 Sémantique [🐓]

Nous donnons une sémantique opérationnelle à nos décodeurs. Pour cela, nous définissons un interpréteur qui nous permet également de tester nos décodeurs. En effet, même si les performances des programmes Gallina sont limitées, elles sont suffisantes pour effectuer des tests sans utiliser l'extraction. L'utilisation d'un interpréteur a pour conséquence l'obligation de vérifier dynamiquement, et à chaque appel d'un opérateur, que l'opération est bien valide. Ces vérifications spécifient alors les conditions nécessaires pour qu'un opérateur soit appelé.

Nous définissons la sémantique avec une monade d'état et d'exception dans laquelle les programmes ne terminent pas forcément (par exemple, `length` ne pouvant pas échouer, `many (fun _ ⇒ length) n` ne termine pas). Pour assurer la terminaison de notre interpréteur (comme l'exige Coq pour des raisons de cohérence logique du système), nous

utilisons un fuel. Le fuel est une borne, choisi au début de l'interprétation, limitant le nombre de répétition effectuée par l'opérateur `many`. Si la borne est atteinte, le programme échoue.

Afin de différencier les erreurs sémantiques et les erreurs liées au fuel, nous utilisons le type de données `Result` :

```
Inductive Result X :=
| Res (x : X)
| NoRes
| NoFuel.
```

Le constructeur `NoRes` permet d'indiquer que le programme a échoué pour des raisons sémantiques, alors que `NoFuel` indique qu'il a échoué par manque de fuel. Nous utilisons donc la monade suivante :

```
Definition MonSem X := span → Result (span * X).

Definition run_ret (x : X) : MonSem X := fun s ⇒ Res (s, x).

Definition run_bind (e : MonSem X) (f : X → MonSem Y) : MonSem Y :=
  fun s ⇒
    match e s with
    | NoRes ⇒ NoRes
    | NoFuel ⇒ NoFuel
    | Res (s, x) ⇒ f x s
    end.

Definition run_fail {X} : MonSem X := fun _ ⇒ NoRes.

Definition run_try_with (e : MonSem X) (f : MonSem X): MonSem X :=
  fun s ⇒
    match e s with
    | NoFuel ⇒ NoFuel
    | NoRes ⇒ f s
    | Res (s,v) ⇒ Res (s,v)
    end.

Definition run_get : MonSem span := fun s ⇒ Res (s,s).

Definition run_set (s : span) : MonSem unit := fun _ ⇒ Res (s,tt).
```

Nous retrouvons les définitions usuelles des opérateurs (nous notons `run_bind` avec `let*`). Cependant, il est important de noter que l'erreur ne propage pas l'état. En effet, si une erreur est rattrapée (avec `run_try_with`), le programme effectue un retour arrière (backtrack). De plus, les erreurs liées au manque de fuel ne sont pas rattrapées. Nous voulons que le fuel serve seulement à garantir la terminaison. Les calculs doivent donc en être indépendants. Or, rattraper une erreur due au manque de fuel, risque de nous faire renvoyer une valeur qui n'aurait pas été obtenue si le fuel n'était pas présent.

La sémantique de `take` vérifie dynamiquement que les données sont disponibles. Si la vérification échoue, l'interprétation également.

```
Definition run_take (n : N) : MonSem span :=
  let* s := run_get in
  if n <=? len s then
    let* _ := run_set (mk_span (pos s + n) (len s - n)) in
    run_ret (mk_span (pos s) n)
  else
    run_fail.
```

L'interprétation de `read` vérifie que nous lisons une donnée accessible depuis la `span` puis utilise `lookupN d n` renvoyant la donnée dans `d` à la `n`-ième position.

```
Definition run_read (range : span) (n : N) (a : data) : MonSem atom :=
  if n <? len range
  then
    lookupN a (pos range + n)
  else
    run_fail.
```

Obtenir le nombre de données restant à décoder revient à récupérer la longueur de l'état :

```
Definition run_length :=
  let* s := run_get in
  run_ret (len s).
```

Tout comme `try_with`, `alt` permet de rattraper les erreurs. En supposant que nous ayons un interpréteur pour nos décodeurs, nous pouvons définir aisément la sémantique de l'opérateur :

```
Definition run_alt (run : ∀ {X}, NomG X → data → MonSem X)
  e f (data : data) : MonSem X :=
  run_try_with (run e data) (run f data).
```

L'opérateur `local` doit s'assurer que l'état n'est pas modifié. Cela revient simplement à sauvegarder l'état avant l'appel au décodeur puis de le restaurer. Si le premier argument de `local` contient un `span`, nous modifions l'état avant l'appel au décodeur :

```
Definition run_local (run : ∀ {X}, NomG X → data → MonSem X)
  (range : option span) e (data : data) : MonSem X :=
  let* save := run_get in
  let* _ := match range with
            | Some range ⇒ run_set range
            | None ⇒ run_ret tt
            end in
  let* v := run e data in
  let* _ := run_set save in
  run_ret v.
```

Dans le cas de `repeat_n n e b`, nous répétons `n`-fois le décodeur `e` et échouons si l'une des itérations échoue :

```
Equations run_repeat_n {run : ∀ X, NomG X → list atom → MonSem X}
  (n : N) (e : X → NomG X) (b : X) (d: list atom) : MonSem X :=
  run_repeat_n 0 e b d := run_ret b;
  run_repeat_n n e b d :=
    let* v := run (e b) d in
    run_repeat_n (n - 1) e v d.
```

Dans le cas de `many e b`, nous répétons le décodeur `e` jusqu'à ce qu'il échoue, le nombre d'itération est borné par un fuel :

```
Equations run_many {run : ∀ X, NomG X → list atom → MonSem X}
  (fuel : N) (e : X → NomG X) (b : X) (d: list atom) : MonSem X :=
  run_many 0 e b d := fun _ ⇒ NoFuel;
  run_many fuel e b d :=
    run_try_with
      (let* v := run (e b) d in
       run_many (fuel - 1) e v d)
      (run_ret b).
```

L'interpréteur est alors un simple filtrage par motif où nous appelons l'interprétation de chaque constructeur :

```
Fixpoint run (fuel : nat) {X} (m : NomG X) (data : data) {struct m} : MonSem X :=
  match m with
  | ret v ⇒ run_ret v
  | op o c ⇒
      let* v := run_op fuel o data in
      run fuel (c v) data
  end

with run_op (fuel : nat) {X} (m : NOM X) (data : data) : MonSem X :=
  match m with
  | FAIL  ⇒ run_fail
  | LENGTH ⇒ run_length
  | READ range pos ⇒ run_read range pos data
  | TAKE n ⇒ run_take n
  | ALT c1 c2 ⇒ run_alt (@run fuel) c1 c2 data
  | LOCAL o e ⇒ run_local (@run fuel) o e data
  | @REPEAT _ (Some n) T e b ⇒
      @run_repeat_n (@run fuel) n T e b data
  | @REPEAT _ None T e b ⇒
      @run_many (@run fuel) fuel T e b data
  end.
```

Nous définissons finalement `parse`, une fonction avec une signature satisfaisante pour être visible et donc pour tester les décodeurs :

```
Definition parse (m : NomG X) (nb_iter_max : nat) data : option X :=
  let len := lengthN data in
  match run nb_iter_max m data (mk_span 0 len) with
  | Res (_, v) ⇒ Some v
  | _ ⇒ None
  end.
```

Nous montrons une correction partielle de la théorie équationnelle présentée dans la Section 4.4.3,. En effet, nous montrons pour chacune des équations que si le calcul ne manque pas de fuel, alors l'égalité est respectée.

4.5 Sémantique par transformateur de prédicats [🐓]

Nous avons maintenant une formalisation de Nom. Afin d'être capable de montrer que des décodeurs sont zéro-copies, nous fournissons à la bibliothèque une logique de programme **partielle** (car nous ne cherchons pas à garantir la terminaison) similaire à celle de la Section 4.3.1.

Pour rappel, l'objectif de cette logique de programme est de nous permettre aisément de montrer que les espaces mémoires accessibles à partir des `spans` renvoyés par un décodeur sont disjoints. La logique de programme raisonne donc exclusivement sur les espaces mémoires manipulés par les programmes. Il est important de noter que la sémantique n'est donc pas complète. Nous excluons volontairement les propriétés relatives aux données ou aux valeurs produites telles que la longueur d'un `span` produit par `take` ou encore la position de la valeur lue dans les données après un `read`. Ajouter ces propriétés ne crée pas de difficultés particulières. Ainsi, nous sous-spécifions volontairement les programmes afin de nous concentrer sur l'objectif premier de cette logique de programme qui est le zéro-copie. Comme pour l'exemple de la Section 4.3.1, nous définissons la sémantique par transformateur de prédicats :

```
Fixpoint wp {X} (m: @NomG atom X) (Q : X → iProp) : iProp :=
  match m with
  | ret v ⇒ Q v
  | op FAIL _ ⇒ True
  | op (LOCAL None _) k | op (READ _ _) k | op LENGTH k ⇒
    ∀ v, wp (k v) Q
  | op (TAKE n) k ⇒
    ∀ v, IsFresh v -∗ wp (k v) Q
  | op (ALT e1 e2) k ⇒
    wp e1 (fun v ⇒ wp (k v) Q) ∧ wp e2 (fun v ⇒ wp (k v) Q)
  | op (LOCAL (Some range) e) k ⇒
    IsFresh range ∗ wp e (fun v ⇒ wp (k v) Q)
```

```
| op (REPEAT _ e b) k ⇒ ∃ Q',
  Q' b ∗
  (<pers> ∀ res, Q' res -∗ wp (e res) Q') ∗
  (∀ res, Q' res -∗ wp (k res) Q)
end.
```

Les rôles des opérateurs `fail`, `read` et `take` sont les mêmes que dans l'exemple. Leur sémantique est donc inchangée.

Similairement à `read`, l'opérateur `length` n'influe pas sur l'accessibilité des données. Sa sémantique est donc identique.

L'opérateur `peek` (donc `LOCAL None _` dans le filtrage) applique un décodeur sur l'état courant et sans consommer de données. Le décodeur peut donc générer des `span` vers de nouvelles données. Cependant, ne faisant pas évoluer l'état du programme, nous ne pouvons pas garantir que la suite du calcul ne génère pas d'autres accès vers ces mêmes données. Raisonnant uniquement sur la non-interférence des zones mémoires, nous ne pouvons donc garantir aucune propriété sur la valeur renvoyée par l'opérateur.

Dans le cas du `alt`, nous demandons à ce que la plus faible précondition soit respectée pour les deux sous-termes. Puisque l'opérateur `alt` parcourt le second sous-terme uniquement si le premier a échoué, nous savons donc, qu'au plus, un seul résultat a été entièrement calculé et donc renvoyé. Il est donc acceptable que les deux préconditions exigent des propriétés sur des espaces mémoires communs. Nous formalisons cette particularité grâce à la conjonction non-séparante de la logique de séparation.

Pour rappel, l'opérateur `scope range e` (donc `LOCAL (Some range) e` dans le filtrage) applique le décodeur `e` sur le `span range`, sans changer l'état courant du calcul. Le décodeur `e` ne peut donc produire que des accès vers l'espace mémoire accessible depuis `range`. Pour obtenir la garantie que ces espaces mémoires sont bien disjoints des espaces manipulés dans la suite du programme, nous devons alors avoir la garantie que l'espace mémoire accessible depuis `range` est lui-même disjoint des futurs espaces mémoires manipulés.

Les opérateurs `repeat_n` et `many` sont des boucles. Leur spécification est donc la suivante : Il existe un invariant de boucle qui est respecté avant les itérations et, en supposant qu'après une itération l'invariant soit respecté, alors il l'est encore à la fin de l'itération suivante.

4.5.1 Cohérence [🐓]

Nous avons d'un côté un modèle pour notre monade via un interpréteur et d'un autre une sémantique par transformateur de prédicats. Ces deux sémantiques spécifient indépendamment notre monade. Nous voulons alors nous assurer que les propriétés décrites par notre sémantique définie dans la logique de séparation sont cohérentes avec les calculs produits par la sémantique opérationnelle.

Pour cela, nous montrons la propriété suivante :

```
∀ X (e : NomG X) (Q : X → iProp) fuel data v s sres,
 run fuel e a s = Res (sres, v) →
  ⊢ wp e (fun v ⇒ <absorb> Q v) -∗
  ([∗ set] n ∈ inject (pos s) (pos sres), & n) -∗ <absorb> Q v.
```

Pour n'importe quelle entrée de l'évaluation, si une valeur est calculée (`run fuel e a s = Res (sres, v)`), que la plus faible précondition nécessaire est vérifiée (`wp e (fun v ⇒ <absorb> Q v)`) et que nous raisonnons bien dans l'espace mémoire compris entre l'état d'entrée et l'état de sortie (`[∗ set] n ∈ inject (pos s) (pos sres), & n`), alors la valeur calculée respecte la postcondition.

Il est nécessaire que la postcondition soit **absorbante** (Section 2.1) car nous ne voulons pas imposer à la postcondition de raisonner sur tout l'espace mémoire. En effet, certains opérateurs, comme `scope` par exemple, peuvent ne produire des accès que vers un sous-ensemble de données qui leur sont accessibles. Par conséquent, nous devons rendre la postcondition absorbante afin d'indiquer que nous ne raisonnons potentiellement pas sur tout l'espace mémoire.

Pour prouver la cohérence, nous commençons par raisonner sur l'évolution de l'état tout au long d'une évaluation. Lorsque nous produisons un nouvel accès vers des données (par un appel à `take` par exemple), nous spécifions dans la sémantique `wp` que l'espace nouvellement accessible est disjoint de tous les espaces accessibles à partir des accès produits ultérieurement dans le programme. Pour garantir cet invariant global sur nos programmes, nous prouvons que l'état dans notre sémantique opérationnelle est monotone [4, 71]. Cependant, nous nous attendons à ce que l'état soit monotone entre chaque opération et pas en tout point du programme. En effet, l'opérateur `scope` nous permet de modifier l'état et donc de casser localement cette monotonie. Ce comportement nous oblige à ajouter une contrainte supplémentaire à la monotonie : l'espace mémoire accessible depuis l'état de sortie doit être inclus dans l'espace mémoire accessible depuis l'état d'entrée.

Nous montrons que notre sémantique respecte ces deux propriétés [🐓] :

```
∀ fuel X (e : NomG X) data s sres v,
 run fuel e data s = Res (sres, v) →
 pos s <= pos sres ∧ pos sres + len sres <= pos s + len s
```

Une fois la preuve de monotonie terminée, le gros du travail pour la preuve de cohérence est fait. Le seul point indépendant de cette monotonie est simplement de montrer que le `span` renvoyé par l'interprétation de `take` permet d'accéder à un espace mémoire compris entre l'état d'entrée et l'état de sortie :

```
∀ n s s_res res,
 run_take n s = Res (s_res, res) →
 set_span res ⊆ inject (pos s) (pos s_res).
```

4.5.2 Preuve de sûreté avec Nom [🐓]

Similairement à l'exemple de la Section 4.3.1, nous savons par le modèle que les `spans` renvoyés par `take` donnent directement accès aux données d'entrée. Pour montrer que le décodeur est sûr, nous pouvons toujours utiliser sa preuve de zéro-copie. En effet, en couplant la logique de programme avec l'interpréteur faisant modèle de la syntaxe abstraite, nous montrons simultanément la sûreté mémoire et le fait qu'un décodeur est zéro-copie. Nous l'énonçons (et le prouvons) de la manière suivante :

```
Lemma Fresh_Safe `{Foldable M} e :
 {{ emp }} e {{ res; <absorb> all_disjointMSL res }} →
 ∀ (data : list atom) fuel (res : M span) s s_res,
  run fuel e data s = Res (s_res, res) →
  Result_in res s.
```

4.5.3 Raisonnement [🐓]

Nous retrouvons par construction syntaxique les triplets de Hoare et sommes capables de donner une spécification aux opérateurs de la monade et aux opérateurs dérivés.

```
Lemma fail_rule P Q : {{ P }} fail {{ v; Q v }}.

Lemma length_rule : {{ emp }} length  {{ _; emp }}.

Lemma take_rule n : {{ emp }} take n {{ v; IsFresh v }}.

Lemma read_rule s res : {{ emp }} read s res {{ _; emp }}.

Lemma peek_rule e : {{ emp }} peek e {{ _; emp }}.

Lemma scope_rule e s P Q :
  {{        P      }}     e     {{ v; Q v }} →
  {{ IsFresh s ∗ P }} scope s e {{ v; Q v }}.

Lemma alt_rule e1 e2 P Q :
  {{ P }}     e1    {{ v; Q v }} →
  {{ P }}     e2    {{ v; Q v }} →
  {{ P }} alt e1 e2 {{ v; Q v }}.

Lemma repeat_rule (e : X →  NomG X) Q :
  (∀ res, {{ Q res }}     e res    {{ v ; Q v }}) →
   ∀ n b, {{  Q b  }} repeat n e b {{ v ; Q v }}.
```

Nous avons maintenant les outils pour montrer que nos programmes sont zéro-copies. Il nous reste alors à montrer le lemme d'adéquation permettant d'extraire les propriétés pures de notre logique de séparation vers `Prop`. Le lemme est une conséquence de la cohérence de nos deux sémantiques (Section 4.5.1) et la cohérence entre notre logique de séparation et `Prop` (`soundness_pure` Section 2.1) :

```
Corollary adequacy_pure : ∀ e Q,
  {{ emp }} e {{ v; ⌜Q v⌝ }} →
  ∀ fuel a v,
   parse e fuel a = Some v →
   Q v.
```

4.6 Cas pratique : Radius [🐓]

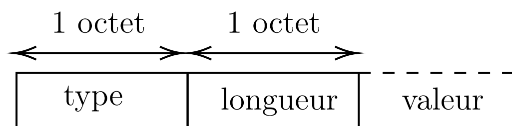
Nous sommes maintenant en mesure de montrer qu'un décodeur défini avec la bibliothèque Nom est sûr et zéro-copie. Prenons comme exemple le protocole Radius [67]. Nous présentons ce protocole car il nécessite l'utilisation de toutes les opérations à l'exception de peek, il nécessite un décodeur bouclant un nombre indéterminé de fois et reste relativement simple.

Le protocole Radius est utilisé pour communiquer avec un serveur contenant les informations permettant d'identifier les utilisateurs. Ainsi, plusieurs serveurs peuvent alors s'occuper des connexions externes tout en ayant centraliser toutes les informations d'identification dans une unique base évitant ainsi des duplications. Lorsqu'un utilisateur indique un nom et un mot de passe, le serveur envoie une requête en Radius à la base d'utilisateur et reçoit une réponse également en Radius.

4.6.1 Formats

Le format (Figure 4.6) comprend les champs **code**, **identifiant**, **longueur** et **authentifiant** faisant respectivement 1, 1, 2 et 16 octets. Ces champs sont suivis d'un nombre non-fixé de champs **attribut** eux-même de tailles variables.

Le champ **code** indique l'objet de la requête (demande de connexion, acceptation, refus etc...). Le champ **identifiant** permet de faire correspondre les demandes et les réponses. Le champ **longueur** correspond à la taille **totale** en octet du paquet incluant donc également les champs **attribut**. Le champ **authentifiant** permet d'authentifier la réponse du serveur. Lors de la demande d'accès, la valeur est tirée aléatoirement sur 2 octets et est utilisée dans la formule permettant de générer la valeur qui permet d'authentifier la réponse du serveur=. Pour rester concis, nous ne détaillons pas cette formule. Les champs **attribut** contiennent les informations nécessaires pour accepter ou rejeter la requête. Par exemple, des champs peuvent contenir le nom et le mot de passe donnés par un utilisateur. Ils sont construits ainsi :



Le champ **type** indique quelle information est contenue dans le champ **valeur** et le champ **longueur** indique la taille en octet de l'**attribut**. Pour des raisons de concision, nous n'énumérons pas tous les **type**s possibles.

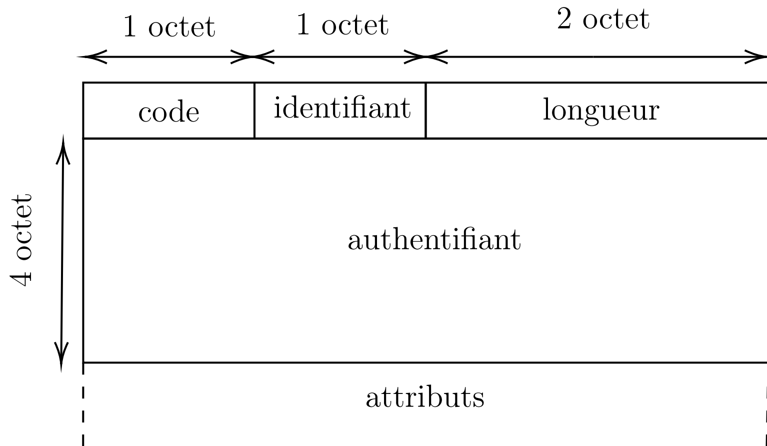


FIGURE 4.6 – Format Radius

4.6.2 Structure de données et décodeurs [🐓]

Nous commençons par représenter la valeur contenue dans les **attribut**s. Pour chaque **type**, nous définissons une représentation de la **valeur**. Pour cela, nous définissons un `Variant` [2] [🐓] :

```
Variant RadiusAttributeS {S : Type} : Type :=
| UserName (_ : S)
(...)
| ChapPassword (_ : nat8) (_ : S)
(...).

Definition RadiusAttribute := @RadiusAttributeS span.
```

Les cas où **type** vaut 1 et 3 sont respectivement représentés par les constructeurs `UserName` et `ChapPassword`. Nous ignorons les autres cas dans le manuscrit et instancions le type `RadiusAttributeS` à la classe `Foldable` (Section 4.2.2). La fonction `foldMap` est alors une simple analyse de cas.

Dans le cas où **type** vaut 1, le champ **valeur** contient uniquement une chaîne de caractère, nous voulons donc renvoyer un `span` donnant accès à cette chaîne de caractère et nous le stockons comme un argument de `UserName`. Lorsque **type** vaut 3, nous avons un identifiant représenté par un octet suivi d'une chaîne de caractère. La chaîne doit contenir au moins un élément. Nous voulons renvoyer l'identifiant et un accès vers la chaîne. Nous stockons ces deux éléments avec le constructeur `ChapPassword`.

Nous commençons par définir le décodeur récupérant les données du champ **valeur**. Nous supposons que ce décodeur ne reçoit que les données associées à ce champ. Pour définir ce décodeur, nous utilisons, en plus des opérateurs de la monade, les combinateurs dérivés `rest : NomG span` et `be_u8 : NomG nat8` fournis par la bibliothèque. Le combinateur `rest` renvoie un `span` donnant accès à toutes les données restantes et le combinateur `be_u8`

2. un `Variant` est simplement un `Inductive` non-récursif

renvoie la valeur de l'octet en tête. Nous définissons le décodeur en faisant une analyse de cas sur la valeur du champ **type** que nous donnons en argument :

```
Definition parse_attribute_content (t : nat8) : NomG RadiusAttribute :=
 match val t with
 | 1 ⇒ let! i := rest in ret (UserName i)
 (...)
 | 3 ⇒
    let! len := length in
    if len <? 2
    then
      fail
    else
      let! v := be_u8 in
      let! i := rest in
      ret (ChapPassword v i)
 (...)
 end.
```

Pour finir de traiter les **attribut**s, nous utilisons les deux combinateurs `map_parser : NomG span → NomG X → NomG X` et `verify : NomG X → (X → bool) → NomG X` également fournis par la bibliothèque. Le combinateur `map_parser s d` appelle le décodeur `s : NomG span` puis utilise le décodeur `d` sur les données accessibles à partir du résultat de `s`. Le combinateur `verify d b` appelle le décodeur `d` et vérifie que le résultat respecte bien le prédicat `b`. Si le prédicat n'est pas respecté, le décodeur `verify d b` échoue. En utilisant ces deux combinateurs, nous pouvons définir le décodeur ainsi :

```
Definition parse_radius_attribute : NomG RadiusAttribute :=
  let! t := be_u8 in
  let! l := verify be_u8 (fun n ⇒ 2 <=? val n) in
  map_parser (take (val l - 2)) (parse_attribute_content t).
```

Nous avons maintenant un décodeur pour les **attribut**s. Il nous reste à décoder l'ensemble du paquet. Nous commençons par définir une structure de données pour l'ensemble du paquet [🐓] :

```
Record RadiusDataS (S : Type) : Type :=
 mk_radiusdata
  { code : nat8;
    identifier : nat8;
    length : nat16;
    authenticator : S;
    attributes : option (VECTOR (@RadiusAttributeS S)) }.

Definition RadiusData := RadiusDataS span.
```

Nous utilisons le type `option` pour les **attribut**s. Il n'y a pas de raison d'initialiser un vecteur, bien qu'il soit vide, lorsqu'il n'y a pas d'**attribut**. Pour écrire le décodeur, nous utilisons deux nouveaux combinateurs de la bibliothèque. Le combinateur `cond : bool → NomG X → NomG (option X)` appelle le décodeur en argument si le booléen est vrai et renvoie la valeur. Si le booléen est faux, la valeur `None` est renvoyée et le décodeur en argument n'est pas appelé. Le second combinateur est `many1 : NomG X → NomG (VECTOR X)`. Le combinateur exécute en boucle le décodeur en argument et stocke chacun des résultats dans un vecteur. Le vecteur est renvoyé lorsqu'une exécution échoue ou si aucune nouvelle donnée n'a été décodée lors de la dernière itération. Le décodeur d'un paquet Radius commence par décoder les quatre premiers champs. Si la longueur du paquet indique qu'il reste des données, c'est qu'il y a des **attribut**s à récupérer. Le décodeur fait alors appel à `parse_radius_attribute` :

```
Definition parse_radius_data : NomG RadiusData :=
  let! code := be_u8 in
  let! identifier := be_u8 in
  let! length := be_u16 in
  let! authenticator := take 16 in
  let! attributes :=
     cond (20 <? val length)
          (map_parser
             (take (val length - 20))
             (many1 parse_radius_attribute)) in
  ret (mk_radiusdata code identifier length authenticator attributes).
```

Nous obtenons alors un décodeur pour des paquets du protocole Radius en utilisant la fonction `parse` de la Section 4.4.4 :

```
Definition parse_radius : nat → list nat8 → option RadiusData :=
  parse parse_radius_data.
```

Pour tester l'implémentation, nous vérifions l'ensemble des tests du décodeur Radius de rusticata (lien).

4.6.3 Spécification et preuve de zéro-copie [🐓]

Pour faire la preuve de correction des décodeurs `parse_attribute_content`, `parse_radius_attribute` et `parse_radius_data`, nous devons utiliser les spécifications des combinateurs utilisés. Ces spécifications sont fournies par la bibliothèque.

Spécification des combinateurs

Le combinateur `rest : NomG span` renvoie un accès vers toutes les données non-traitées. Similairement aux `span`s renvoyés par l'opérateur `take`, le résultat de `rest` donne donc accès à des données qui était inaccessibles. La spécification est donc la même que pour `take` [🐓] :

```
Lemma rest_rule : {{ emp }} rest {{ v; IsFresh v }}.
```

Les combinateurs lisant les données en tête (`be_uX : NomG natX`) génèrent des accès mémoires mais ne les renvoient pas. Ils ont alors tous la même spécification qui est la suivante [🐓] :

```
Lemma uX_rule : {{ emp }} be_uX {{ _; True }}.
```

La postcondition est `True` car nous ne partageons aucune information mais indiquons quand même que des données peuvent avoir été traitées (ce qui est toujours le cas pour ces combinateurs).

La spécification du décodeur `map_parser s d` est plus intéressante. La précondition du décodeur doit être suffisamment forte pour respecter la précondition de `s` et `d`. De plus, afin d'obtenir des propriétés vis-à-vis des `span`s renvoyés par `d` et notamment leurs disjonctions vis-à-vis des prochains `spans` générés, nous devons avoir la garantie que le `span` renvoyé par `s` (et donc celui donnant accès aux données que nous traitons avec `d`) soient lui-même disjoints des futurs `span`s générés. Nous exprimons ces propriétés ainsi [🐓] :

```
Lemma map_parser_rule s d H Q R :
  {{ H }} s {{ v; IsFresh v * R }} →
  {{ R }} d {{ v; Q v }} →
  {{ H }} map_parser s d {{ v; Q v }}.
```

La précondition `H` est fournie aux triplets de `s` afin de montrer que nous pouvons appeler `s`. Des propriétés contenues dans `H` peuvent ne pas être nécessaires pour que `s` respecte sa postcondition. Ces propriétés seront alors contenues dans `R` grâce à la règle d'encadrement et pourront être utilisées pour montrer que la précondition de `d` est respectée. Le décodeur `map_parser s d` se termine après l'exécution de `d`. Les propriétés obtenues à la fin de l'exécution de `d` sont donc les propriétés que nous avons à la fin de l'exécution de `map_parser s d`.

Puisque nous exécutons `d` sur les données accessibles à partir du résultat de `s`, nous exigeons que le résultat respecte le prédicat `IsFresh` afin d'obtenir les propriétés de disjonctions énoncées avant le triplet. Le respect de ce prédicat par le `span` renvoyé par `s` est alors nécessaire pour faire appel à `d` et n'est donc pas transmis jusqu'à la postcondition de `d`.

Ne nous intéressant qu'à la disjonction des `span`s, la spécification de `verify e c` est la spécification de `e` (elle pourrait facilement être enrichie en indiquant que `c` est respecté par le résultat) [🐓] :

```
Lemma verify_rule e c H Q :
 {{ H }} e {{ v; Q v }} →
 {{ H }} verify e c {{ v; Q v }}.
```

Pour `cond b e`, nous exécutons `e` que si le booléen `b` est vrai et renvoyons `None` sinon. La spécification de `cond b e` est donc celle de `e` à laquelle on ajoute le fait que la précondition doit impliquer la postcondition lorsque le booléen est faux (puisque nous n'échouons pas mais renvoyons `None`) [🐓] :

```
Lemma cond_rule b H Q :
 {{ H }} e {{ v; Q (Some v) }} →
 (H ⊢ Q None) →
 {{ H }} cond b e {{ v; Q v }}.
```

Le dernier combinateur est `many1 : NomG X → NomG (VECTOR X)`. Le décodeur en argument est lancé un nombre indéterminé de fois et chaque itération ne dépend pas du résultat de l'exécution précédente (hormis si elle provoque l'arrêt des itérations). Nous spécifions que le décodeur doit pouvoir être appelé sans aucune condition nécessaire. La postcondition indique que chacun des éléments du vecteur renvoyé doit respecter la postcondition du décodeur en argument. Nous l'énonçons ainsi à l'aide de la fonction `[]↓ : VECTOR X → list (N * X)` transformant un vecteur en une liste de paire de positions et valeurs [🐓] :

```
Lemma many1_rule e Q :
 {{ emp }} e {{ v; Q v }} →
 {{ emp }} many1 e {{ res; [∗ list] v ∈ []↓ res, Q v.2 }}.
```

Spécifications et preuves des décodeurs [🐓]

Comme nous l'avons illustré dans l'exemple de la Section 4.3.1, spécifier qu'un programme ne renvoie que des accès mémoires disjoints se fait à partir de `all_disjointMSL`. Les spécification des programmes sont donc les suivantes :

```
Lemma parse_attribute_content_spec : ∀ n,
 {{ emp }} parse_attribute_content n {{ v; <absorb> all_disjointMSL v }}.

Lemma parse_radius_attribute_spec :
 {{ emp }} parse_radius_attribute {{ v; <absorb> all_disjointMSL v }}.

Lemma parse_radius_data_spec :
 {{ emp }} parse_radius_data {{ v; <absorb> all_disjointMSL v }}.
```

Pour prouver la correction des programmes, nous devons alors simplement utiliser les tactiques associées à chacune des opérations. La bibliothèque fournit une tactique très élémentaire, analysant simplement la syntaxe du programme et effectuant une étape de raisonnement lorsque le programme appelle un combinateur fourni. Pour cela, la tactique appelle la règle associée au combinateur et est donc de la forme suivante [🐓] :

```
match goal with
| |- {{ _ }} operator {{ _; _ }} ⇒ eapply operator_rule
```

De plus, la tactique effectue une analyse de cas lorsqu'un `match` est utilisé dans le programme :

```
| |- {{ _ }} match ?x with _ ⇒ _ end {{ _; _ }} ⇒ destruct x
```

Simplement avec ces tactiques, les preuves de `parse_attribute_content` et `parse_radius_attribute` [🐓] sont alors quasi-automatiques car elles nécessitent respectivement des appels à 5 et 7 tactiques relativement peu évoluées de Coq et MoSel.

Pour la preuve de `parse_radius_data` [🐓], un petit effort supplémentaire est nécessaire. La raison est que nous n'utilisons pas la fonction `[]↓` lorsque nous définissons la fonction `foldMap` de la classe `Foldable` pour la structure `RadiusData`. Nous devons alors montrer que les `span`s parcourus par `foldMap` dans les **attribut**s correspondent bien aux `span`s dans les **attribut**s une fois le vecteur transformé en liste. Cette sous-preuve reste très raisonnable et la preuve de correction de `parse_radius_data` est de 9 lignes.

Avec `Fresh_Safe` de la Section 4.5.2 et `parse_radius_data_spec`, nous avons comme corollaire que `parse_radius_data` est sûr. En utilisant `all_disjointM_spec` (Section 4.2.4) et l'adéquation des deux sémantiques (Section 4.5.1), nous pouvons conclure que tous les `span`s sont disjoints :

```
Lemma parse_radius_specification : ∀ nb_attrib_max data res,
  parse_radius nb_attrib_max data = Some res → all_disjointM res.
```

4.7 Travaux connexes

En lien avec CompCert, Jourdan et al. [35] ont effectué des travaux de vérification de parseurs en Coq. Ces travaux présentent une méthode pour garantir qu'un automate produit à partir d'une grammaire LR(1) reconnaît bien cette grammaire. Pour cela, ils traduisent la grammaire en un automate *pushdown* et un certificat d'équivalence entre le langage de l'automate et celui de la grammaire. Un validateur va alors vérifier le certificat tandis que l'automate sera interprété par un interpréteur vérifié.

Fisher et al. [28] formalisent ce qu'est un bon langage de descriptions de données et s'interroge donc également sur les propriétés que l'on peut attendre d'un décodeur. L'une des contributions de l'équipe de Pads est de raisonner sur la structure de données produite et le décodeur à travers un descripteur de décodage. Ce descripteur est également une structure de données, mais au lieu de contenir les données, elle contient des informations sur les potentielles erreurs rencontrées lors du décodage. Bien que le descripteur de décodage n'ait pas survécu dans des travaux plus récents, l'idée de raisonner à travers un autre objet a quand à elle perduré. En effet, de nombreux langages de description vérifiés [70, 82, 64, 77] produisent également un encodeur et établissent une relation d'"aller-retour". Intuitivement, cette relation plus ou moins forte selon les langages exprime au minima que décoder des données que nous avons encodées ne doit pas modifier les données.

Une deuxième contribution de Fisher et al. [28] est de définir une méta-théorie décrivant des propriétés de typage que doivent respecter les langages de descriptions de données. Oury et Swierstra [59] montrent que la théorie des types dépendants est un support suffisant pour garantir ces propriétés. van Geest et Swierstra [82] poussent l'approche en prônant que le format et son encodage devrait être dissocier. En effet, un format est simplement la description d'une structure contenant les valeurs à stocker (le résultat) alors que l'encodage,

comme le boutisme (endianness) d'un entier, est le chemin pour y accéder. Par conséquent, en plus d'un langage permettant de décrire des formats, les auteurs définissent un type de données décrivant l'encodage. Un décodeur et un encodeur respectant la relation d'aller-retour sont alors dérivés du format et de l'encodage.

RockSalt [52] utilise une grammaire d'expressions régulières afin de modéliser x86 en Coq. Avec une sémantique dénotationnelle, Morrisett et al. [52] spécifie et génère des décodeurs vérifiés produisant une syntaxe abstraite à partir d'un binaire. Par la suite, Tan et Morrisett [77] ont étendu l'approche afin de permettre également la génération d'un encodeur (respectant la propriété d'aller-retour) dans un langage uniforme. Bien que la sémantique dénotationnelle soit utilisable pour obtenir certaines propriétés sur la grammaire, l'utiliser pour montrer la cohérence entre l'encodeur et le décodeur générés était trop coûteux. En effet, la génération séparée des deux programmes crée deux structures internes différentes. Les preuves ont alors besoin d'une relation entre les deux structures. De plus, pour la plupart des constructeurs de la grammaire, l'encodeur peut être automatiquement calculé. Cet avantage est inexploité si nous les définissons séparément. La bi-grammaire présentée permet alors de générer simultanément les deux programmes et contient ainsi les preuves de cohérence lorsque les décodeurs et encodeurs ne peuvent pas automatiquement être générés.

Narcissus [70] utilise les structures de données Coq pour représenter les données. Similairement à van Geest et Swierstra [82], la librairie fournit deux collections de combinateurs découplant ainsi le format et son encodage. Une collection permet de décrire des formats. Les formats sont des spécifications dans `Prop` décrivant une relation entre la source (les données) et la cible (la structure de données). Les encodeurs et décodeurs sont alors corrects si l'entrée et la sortie respectent la relation. Ainsi, Suriyakarn et al. [70] fournissent également une collection de combinateurs de décodage pour définir des encodeurs et décodeurs. Les combinateurs de décodage sont prouvés correct par rapport à des combinateurs de formats. Un utilisateur doit donc décrire un format avec les combinateurs de formats puis dériver par raffinement des décodeurs et encodeurs à l'aide des combinateurs et de leur preuve de correction. Les programmes sont alors une preuve dans laquelle nous fournissons un témoin que le format peut être encodé et décodé. La librairie est extensible (mais peu accessible ?) et les programmes générés sont finalement transformés en programmes OCaml grâce au mécanisme d'extraction Coq.

EverParse [64, 74] est constitué d'une bibliothèque de combinateurs et d'un langage de description de données. Les décodeurs sont des fonctions produisant un message à partir d'une chaîne de bits. La bibliothèque LowParse est écrite en F* et construite en trois parties.

Une partie contient les spécifications relatives à l'ensemble des décodeurs. Par exemple, les auteurs définissent un décodeur sécurisé comme acceptant une et une seule représentation binaire pour chaque message. Un corollaire est alors que, pour chaque décodeur sécurisé, il existe un sérialiseur qui est l'inverse (une forme forte de la propriété d'aller-retour).

Deux autres parties de la bibliothèque sont des librairies de combinateurs. Une librairie est écrite en F* et permet de spécifier le comportement fonctionnel des combinateurs. La seconde librairie écrite en Low*, un langage bas-niveaux intégré à F*, a accès à un

modèle mémoire dans le style de C et garantit des propriétés de sûreté mémoire sur les combinateurs. Les combinateurs Low*sont alors prouvés fonctionnellement corrects à partir des combinateurs F*. La librairie F* est un meilleur interface pour définir des décodeurs. Les décodeurs Low* sont alors dérivés à partir des décodeurs F* avec potentiellement l'aide de tactiques F* spécifiques à cet usage.

Le langage de descriptions de données [74] permet de décrire des formats de données à partir de quatre constructions : les types de bases ; les types raffinés (refinement type) ; les paires dépendantes ; l'analyse de cas. Les décodeurs sont produits à partir des formats. Les formats ne sont pas des spécifications par rapport aux décodeurs. Le fait que le décodeur généré décode le format décrit fait alors partie du code de confiance du système.

4.8 Conclusion

Dans ce chapitre, nous avons présenté les décodeurs zéro-copie et nous avons formalisé leur correction. Nous avons présenté une méthode utilisant la logique de séparation afin de créer une logique de programme adaptée.

Pour étudier cette méthode sur un exemple réel, nous avons formalisé la bibliothèque Nom [22]. La formalisation fournit une théorie équationnelle et une sémantique opérationnelle de la bibliothèque. Finalement, nous avons implémenté une logique de programme à partir de laquelle nous pouvons prouvé la correction zéro-copie des décodeurs de la bibliothèque Nom. Bien que cette logique de programme soit définie dans la logique de séparation, nous sommes en mesure de fournir dans `Prop` la correction des décodeurs Nom.

Chapitre 5

Raffinement des décodeurs Gallina en C

Après avoir écrit des décodeurs avec notre bibliothèque, nous voulons pouvoir effectuer des transformations sur ces programmes. Nous souhaitons par exemple produire des programmes dans un langage manipulant des pointeurs tels que C ou encore analyser le programme afin d'éliminer les vérifications redondantes de notre sémantique opérationnelle (Section 4.4.4). Cependant, nous nous confrontons à deux problèmes.

Le premier est que nous définissons nos programmes dans une monade libre et manipulons donc des continuations. Pour parcourir et analyser le programme, nous devons traverser ses continuations. Or, en Gallina, le seul moyen d'interagir avec une fonction est de l'appeler. Reprenons le combinateur `decode_next` (Section 4.3.1). La définition du combinateur est un appel à `take` suivi d'un appel à `read`.

```
Definition decode_next : Decodeur N :=
  let! s := take 1 in
  read s 0.
```

Si nous désucrons ce programme, nous observons que le constructeur de `ReadOp` est contenu dans une continuation :

```
op (TakeOp 1) (fun s ⇒ op (ReadOp s 0) ret).
```

Ainsi, pour parcourir l'ensemble du programme et donc atteindre `ReadOp`, nous devons appeler cette continuation avec un entier naturel.

Le second problème concerne également le parcours du programme. En effet, l'utilisation d'une monade nous permet de définir un langage impératif tout en empruntant les structures de contrôle de Coq. Par conséquent, lorsque nous définissons un programme dans la monade libre, l'arbre que nous construisons (l'arbre de type `NomG`) ne contient pas les informations nécessaires pour analyser le programme. De plus, nous n'avons pas les mêmes libertés que si nous avions défini un langage intégré profondément "deep embedding".

Pour palier à ce problème, une solution est d'identifier les constructions et les fonctions Coq utilisées dans les décodeurs et de définir une structure de données capturant l'ensemble de ces fonctionnalités. Nous transformons alors le programme vers une version réifiée. Pour produire une version réifiée d'un programme, nous définissons une relation

entre nos décodeurs et les programmes de cette nouvelle représentation. Nous accompagnons cette relation de règles d'inférence pour chacune des opérations, puis, produisons par raffinement [86, 23, 13, 24, 68] un programme réifié respectant la relation. Nous réifions les structures de contrôle Coq et les fonctions relatives aux types de bases. Nous profitons de cette réification pour supprimer certaines vérifications redondantes et produire des programmes C.

Nous commençons par illustrer l'approche sur l'exemple `decode_packet_SSH` (Section 4.3.1). Nous présentons une représentation minimale et suffisante pour ce cas particulier (Section 5.1). Nous définissons la relation sur laquelle se base le raffinement et plusieurs règles d'inférence. Nous complétons ensuite la représentation avec les constructions que nous avons identifiées comme nécessaires pour définir des décodeurs de la bibliothèque Nom (Section 5.2.1) et présentons certaines règles d'inférence supplémentaires (Section 5.2.2). Nous discutons des difficultés rencontrées avec la paramétricité dans la Section 5.3 et terminons avec une brève présentation de la compilation de notre représentation vers C (Section 5.4).

5.1 Raffinement `decode_packet_SSH` [🐓]

Pour rappel, voici la définition de `decode_packet_SSH` :

```
Definition decode_packet_SSH : Decodeur packet_SSH :=
  let! packet_length := decode_u32 in
  let! padding_length := decode_next in
  if padding_length + 1 <=? packet_length
  then
    let! payload := take (packet_length - padding_length - 1) in
    let! padding := take padding_length in
    let! mac := take 20 in
    ret (mk_ssh packet_length padding_length payload mac)
  else
    fail.
```

Commençons par identifier les constructions utilisées dans chaque étape de calcul de notre décodeur. Nous définirons alors un constructeur dans notre représentation pour chacune des constructions. Dans le cas de `decode_packet_SSH`, nous avons la séquence, `take`, `fail`, le branchement booléen, les constantes d'entiers, la comparaison `<=?`, l'addition et la soustraction. Pour définir `decode_next`, nous utilisons l'opérateur `read` et pour `decode_u32`, nous utilisons la fonction `to_u32` transformant 4 octets en un entier de 32 bits. Nous voulons également rendre explicite les vérifications faites dans l'interpréteur de la monade (Section 4.3). Nous devons donc représenter la comparaison stricte `<?` et la projection de la longueur d'un `span` présentes dans l'interprétation de `read`.

Nous ne mentionnons pas volontairement la fonction `mk_ssh`. Cette fonction est spécifique à ce décodeur et nous la représentons, pour l'exemple, en utilisant des paires. Nous

reviendrons sur celle-ci plus tard (Section 5.1.3).

5.1.1 Représentation

Pour définir la représentation, nous utilisons une syntaxe abstraite d'ordre supérieur paramétrisée par le type des variables [84, 19]. L'utilisation d'une syntaxe d'ordre supérieur permet d'encoder les lieurs en utilisant les liaisons Gallina et obtenons gratuitement la substitution. En paramétrisant le type des variables, nous évitons des analyses de cas sur les variables. En effet, permettre une telle analyse nous aurait alors plongé dans des travers que nous voulons justement éviter, c'est-à-dire des étapes de calculs dépendantes de branchements Gallina invisibles lors de l'analyse de la structure du programme.

Nous pouvons alors définir la représentation contenant les fonctionnalités nécessaires pour `decode_packet_SSH` :

```
Context {var : Type → Type}.

Inductive VAL : Type → Type :=
| Var : ∀ {X}, var X → VAL X
| Const : N → VAL N
| Add : VAL N → VAL N → VAL N
| Sub : VAL N → VAL N → VAL N
| Le : VAL N → VAL N → VAL bool
| Lt : VAL N → VAL N → VAL bool
| Len : VAL span → VAL N
| Pair : ∀ {X Y}, VAL X → VAL Y → VAL (X * Y)
| Fst : ∀ {X Y}, VAL (X * Y) → VAL X
| Snd : ∀ {X Y}, VAL (X * Y) → VAL Y
| ToU32 : VAL octet → VAL octet → VAL octet → VAL octet → VAL N.

Inductive Expr : Type → Type :=
| Val : ∀ {X}, VAL X → Expr X
| LetIn : ∀ {X}, Expr X → ∀ {Y}, (var X → Expr Y) → Expr Y
| IfThenElse: VAL bool → ∀ {X}, Expr X → Expr X → Expr X
| Take : VAL N → Expr span
| Read : VAL span → VAL N → Expr octet
| Length : Expr N
| Fail : ∀ {X}, Expr X.
```

Nous définissons quelques notations pour gagner en lisibilité dans les prochaines sections : la séquence est notée `let% _ := _ in _`; le symbole `%` pour les constructeurs de `VAL`, par exemple `+%` pour l'addition; nous écrivons `If _ Then _ Else _` pour le branchement booléen.

5.1.2 Sémantique de la représentation

Nous donnons une sémantique à grands pas. Pour cela, nous instancions les variables aux valeurs produites par les opérateurs. Les signature `Coq` de la sémantique de `VAL` et de `Expr` sont les suivantes :

```
Definition ID (X : Type) := X.

Inductive sem_VAL : ∀ {X}, @VAL ID X → ID X → Prop :=

Inductive sem_Expr {data : list N} :
  span → ∀ {X}, @Expr ID X → option (ID X * span) → Prop
```

Globalement, la sémantique des opérateurs est assez directe (Figure 5.1 et Figure 5.2).

Dans le cas de `VAL`, nous ne faisons qu'associer un constructeur à l'opération qu'il représente. Dans le cas de `Expr`, les seuls cas intéressants sont ceux de `Take` et `Read`.

```
| sem_Take : ∀ s vn n,
    sem_VAL vn n →
    sem_Expr s (Take vn) (Some ({| pos := pos s; len := n |},
                                 {| pos := pos s + n; len := len s - n |}))

| sem_Read : ∀ s vrange range vn n v default,
    sem_VAL vrange range →
    sem_VAL vn n →
    lookup_default data (pos range + n) default = v →
    sem_Expr s (Read vrange vn) (Some (v, s))
```

Il est important de noter que la sémantique des constructeurs `Take` et `Read` est radicalement différente de celle des opérateurs `take` et `read`. Les sémantiques des opérateurs `Take` et `Read` ne peuvent pas échouer contrairement à celles de leur homonyme. En effet, l'interprétation des opérateurs `take` et `read` effectue des vérifications afin d'éviter des comportements inappropriés vis-à-vis des données (Section 4.4.4). Cependant, pour être plus fidèle à la sémantique de `C`, les opérateurs `Take` et `Read` ne font aucune vérification. L'opérateur `Take n` se contente de produire un `span` donnant accès aux cases mémoires des `n` prochaines données (même si elles n'existent pas) et de décaler l'état du programme. L'opérateur `Read` cesse de vérifier que la case lue est effectivement accessible depuis le `span` en argument. Si la case existe dans la représentation des données, nous renvoyons la valeur stockée, sinon nous renvoyons une valeur quelconque (représentée par `lookup_default l n d` renvoyant la `n`-ième valeur de la liste `l` et `d` si la liste n'est pas assez longue).

5.1.3 Relation

Nous voulons que les propriétés prouvées sur le résultat de l'évaluation d'un programme défini dans la monade `Decodeur` soient toujours validées par le résultat de l'évaluation sur la version réifiée. Les résultats qui nous intéressent sont ceux issus d'une évaluation où

```
Inductive sem_Expr {data : list N} :
  span → ∀ {X}, @Expr ID X → option (ID X * span) → Prop :=
| sem_Val : ∀ s X (v :VAL X) res,
    sem_VAL v res →
    sem_Expr s (Val v) (Some (res, s))
| sem_LetInF : ∀ X ex Y (k : X → Expr Y) s,
    sem_Expr s ex None →
    sem_Expr s (LetIn ex k) None
| sem_LetInS : ∀ X ex Y (k : X → Expr Y) s v sv res,
    sem_Expr s ex (Some (v, sv)) →
    sem_Expr sv (k v) res →
    sem_Expr s (LetIn ex k) res
| sem_IfThenElseT : ∀ vb X (et ee : Expr X) res s,
    sem_VAL vb true →
    sem_Expr s et res →
    sem_Expr s (IfThenElse vb et ee) res
| sem_IfThenElseF : ∀ vb X (et ee : Expr X) res s,
    sem_VAL vb false →
    sem_Expr s ee res →
    sem_Expr s (IfThenElse vb et ee) res
| sem_Take : ∀ s vn n,
    sem_VAL vn n →
    sem_Expr s (Take vn)
     (Some ({| pos := pos s; len := n |},
            {| pos := pos s + n; len := len s - n |}))
| sem_Read : ∀ s vrange range vn n v default,
    sem_VAL vrange range →
    sem_VAL vn n →
    lookup_default data (pos range + n) default = v →
    sem_Expr s (Read vrange vn) (Some (v, s))
| sem_Length : ∀ s, sem_Expr s Length (Some (len s, s))
| sem_Fail : ∀ X s, sem_Expr s (Fail : Expr X) None.
```

FIGURE 5.1 – Sémantique Expr

```
Inductive sem_VAL : ∀ {X}, @VAL ID X → ID X → Prop :=
| sem_Const : ∀ n, sem_VAL (Const n) n
| sem_Add : ∀ vm m vn n,
    sem_VAL vm m →
    sem_VAL vn n →
    sem_VAL (Add vm vn) (m + n)
| sem_Sub : ∀ vm m vn n,
    sem_VAL vm m →
    sem_VAL vn n →
    sem_VAL (Sub vm vn) (m - n)
| sem_Le : ∀ vm vn m n,
    sem_VAL vm m →
    sem_VAL vn n →
    sem_VAL (Le vm vn) (m <=? n)
| sem_Lt : ∀ vm vn m n,
    sem_VAL vm m →
    sem_VAL vn n →
    sem_VAL (Lt vm vn) (m <? n)
| sem_Len : ∀ vs s,
    sem_VAL vs s →
    sem_VAL (Len vs) (len s)
| sem_Pair : ∀ X Y va (a : ID X) vb (b : ID Y),
    sem_VAL va a →
    sem_VAL vb b →
    sem_VAL (Pair va vb) (a,b)
| sem_Fst : ∀ X Y v (a : ID X) (b : ID Y),
    sem_VAL v (a, b) →
    sem_VAL (Fst v) a
| sem_Snd : ∀ X Y v (a : ID X) (b : ID Y),
    sem_VAL v (a, b) →
    sem_VAL (Snd v) b
| sem_ToU32 : ∀ va vb vc vd a b c d,
    sem_VAL va a →
    sem_VAL vb b →
    sem_VAL vc c →
    sem_VAL vd d →
    sem_VAL (ToU32 va vb vc vd) (to_u32 a b c d)
| sem_Var : ∀ X (v : ID X), sem_VAL (Var v) v.
```

FIGURE 5.2 – Sémantique VAL

le `span` en entrée donne uniquement accès aux données que nous voulons décoder. Nous définissons alors un `span` bien formé pour une évaluation comme un `span` ne donnant accès qu'à des cases mémoires contenues dans la représentation des données. Pour rappel, les positions des cases mémoire accessibles depuis un `span` sont relatives à la représentation des données (la position 0 est la position de la première donnée). Par conséquent, un `span` est bien formé par rapport à des données si les positions des cases mémoires auxquelles il donne accès sont inférieures ou égales aux nombres de données :

```
Definition span_data_wf (data : list N) (s : span) :=
    pos s + len s <= length data.
```

La sémantique de la représentation `Expr` est plus permissive que celle de `Decodeur`. En effet, la sémantique de `Decodeur` spécifie les comportements légaux des opérateurs à l'aide des vérifications effectuées. Tout comportement dangereux, tel que des accès vers des zones mémoires en dehors des données, fait alors échouer le programme. En retirant ces vérifications dans la sémantique de `Expr`, nous permettons la définition de programmes avec des comportements dangereux sans pour autant que leur évaluation échoue.

Nous voulons que la relation entre les programmes `Expr` et `Decodeur` garantisse que, si nous réifions une opération dangereuse d'un programme de type `Decodeur`, c'est-à-dire `take` ou `read`, alors la version raffinée de cette opération effectue les vérifications nécessaires. Pour cela, nous voulons avoir la garantie que, si le programme `Expr` généré réussit à produire un résultat, alors le programme `Decodeur` en produit également un. En ayant cette implication, nous avons la garantie que le programme généré ne réussit que lorsque les vérifications du programme d'origine sont respectées. Nous définissons alors la relation ainsi :

```
Definition adequate {X Y} (R : X → Y → Prop) (d : Decodeur X) (e : Expr Y)
  (data : list N) :=
  ∀ s res,
    span_data_wf data s →
    @sem_Expr data s Y e res →
    match res with
    | None ⇒ eval d data s = None
    | Some (v, t) ⇒ ∃ r, eval d data s = Some (r, t) ∧ R r v
    end.
```

Notons que nous imposons au programme d'origine d'échouer si le programme généré échoue. Cela impose à la réification de produire un programme pertinent et non simplement un programme ne faisant qu'échouer. Sans cette condition supplémentaire, `Fail` serait `adequate` avec n'importe quel programme. De plus, nous imposons que les deux états de sortie soient égaux. En effet, un décodeur peut traiter des données qu'il n'utilise pas dans le résultat. Nous voulons alors que ces données soient également traitées dans le programme généré.

Deux programmes sont `adequate`s par rapport à une relation `R` et une représentation des données. En plus de spécifier des propriétés entre les résultats produits par les évaluations

des programmes, la relation est utilisée pour décrire des propriétés relatives aux données.

Nous aurions pu être tentés de fixer R avec l'égalité. Cependant, nous considérons que c'est trop restrictif car nous ne souhaitons pas forcément que les deux versions renvoient des valeurs égales. Dans notre exemple, nous manipulons des paquets **SSH** que nous construisons avec la fonction `mk_ssh`. Renvoyer un objet du même type nous aurait obligés à définir un constructeur dans `VAL` pour cette fonction. Notre exemple étant dédié au décodeur **SSH**, cela aurait été évidement possible. Cependant, l'objectif de l'approche est de permettre la réification des décodeurs utilisant les opérations que nous avons représentées, sans nous limiter à des formats précis. Ainsi, pour les constructeurs de structures de données tels que `mk_ssh`, nous devons utiliser une nouvelle structure pouvant être produite dans cette nouvelle représentation des programmes. Dans l'exemple, nous utilisons une paire de paire contenant tous les champs du type de données `packet_SSH` (Section 4.3) et définissons une relation entre les deux :

```
Definition RpacketSSH (s : packet_SSH) (v : (N * N) * (span * span)) :=
  v.1.1 = packet_length s ∧
  v.1.2 = padding_length s ∧
  v.2.1 = payload s ∧
  v.2.2 = mac s.
```

Ainsi le décodeur réifié de `decode_packet_SSH` ne renverra pas une valeur de type `packet_SSH`, mais une paire de type `(N * N) * (span * span)`.

De plus, si vous voulons supprimer les vérifications redondantes, nous devons garder une trace du programme lors de la réification. Plutôt que de garder la trace complète du programme, nous pouvons simplement définir, pour chaque opération, une relation décrivant les propriétés potentiellement utiles pour la suite du programme.

5.1.4 Règles d'inférence

Pour produire la version réifiée, nous construisons des règles d'inférence que nous utilisons pour raffiner le programme.

D'abord, deux valeurs doivent simplement respecter la relation :

```
Lemma ret_adequate (R : X → Y → Prop) (v : X) (va : VAL Y) a data :
    sem_VAL va a →
    R v a →
    adequate R (ret v) (Val va) data.
```

Dans les deux représentations, la sémantique de la séquence est similaire. Les séquences sont adequates si les sous-programmes le sont :

```
Lemma bind_adequate R R0
  (e : Decodeur X) (ke : X → Decodeur Y)
  (h : Expr X0) (kh : ID X0 → Expr Y0) data :

    adequate R0 e h data  →
    (∀ res vres, R0 res vres → adequate R (ke res) (kh vres) data) →
    adequate R (let! v := e in ke v) (let% v := h in kh v) data.
```

La relation R0 ne dépend que de la règle d'adéquation entre les premiers sous-programmes des séquences. Nous l'utilisons donc pour exprimer des propriétés sur les valeurs produites. Ces propriétés sont alors propagées dans la suite du raffinement et nous gardons ainsi une trace de l'exécution du programme.

Pour la règle d'adéquation de l'opérateur take, nous définissons une nouvelle relation entre le span que l'opérateur renvoie et le span que la version réifiée doit renvoyer. Ces spans doivent d'abord être égaux. Cette condition est nécessaire pour que les propriétés montrées avec la logique de programme du chapitre précédent (Section 4.5) restent correctes avec la nouvelle représentation des programmes. Ensuite, le span renvoyé doit être bien formé par rapport aux données. Cette propriété découle de la monotonie de eval. Nous l'avons déjà montré car elle est nécessaire pour l'adéquation de la sémantique opérationnelle et de celle par transformateur de prédicats (Section 4.5.1). Pour finir, la longueur du span doit être égale au nombre de données demandées. Décrire ces trois propriétés dans la règle de take nous permet de les conserver tout au long du raffinement.

```
Definition span_eq_take data n (s0 s1 : span) : Prop :=
  s0 = s1 ∧ span_data_wf data s1 ∧ len s1 = n.
```

Pour rappel (Section 5.1.2), les sémantiques des opérateurs take et Take sont différentes. L'opérateur take renvoie un accès mémoire vers des données uniquement si ces données sont bien disponibles, alors que l'opérateur Take n'effectue aucune vérification.

Pour que les opérateurs soient adequates, nous devons alors faire apparaître la vérification dans la syntaxe du programme réifiée :

```
Lemma take_adequate (hn : VAL N) data (n : N) :
    sem_VAL hn n →
    adequate (span_eq_take data n) (take n)
      (let% len := Length in
       If hn <=% Var len
       Then Take hn
       Else Fail) data.
```

Dans la sémantique opérationnelle de read (Section 4.4.4), deux vérifications sont effectuées. La première vérifie que le span donne accès à suffisamment de données, nous effectuons alors la vérification explicitement dans le programme généré. La deuxième vérifie que la donnée demandée est bien dans la représentation globale. Nous avons cette

garantie si le `span` est bien formé par rapport à la représentation globale. La règle est donc la suivante :

```
Lemma read_adequate hs hn data n s :
    span_data_wf data s →
    sem_VAL hs s →
    sem_VAL hn n →
    adequate eq (read s n)
      (If hn <% Len hs
       Then Read hs hn
       Else Fail) data.
```

Raffinement `decode_next`

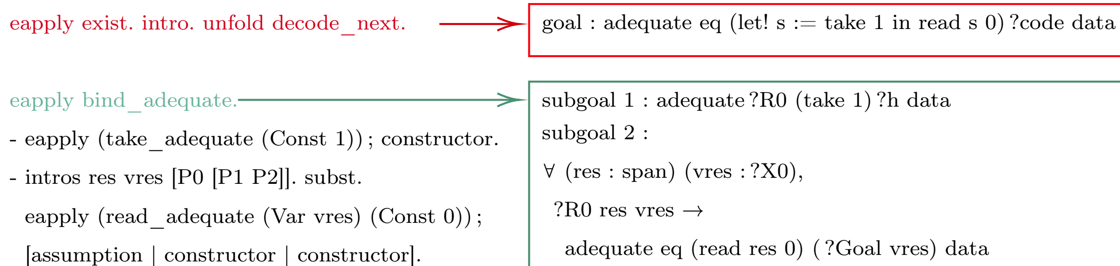
Nous avons suffisamment de règles pour réifier le programme `decode_next` de la Section 4.3.1 :

```
Definition decode_next : Decodeur N :=
  let! s := take 1 in
  read s 0.
```

La réification de `decode_next` est automatisable par simple analyse syntaxique. Cependant, pour illustrer le fonctionnement de l'approche, nous n'utilisons pas d'automatisation. Notre objectif est de produire une version réifiée s'évaluant en une valeur égale au résultat de l'évaluation de `decode_next`. Nous l'exprimons en `Coq` en utilisant les sigma-types :

```
Definition decode_next_DF :
  {code | ∀ data, adequate eq decode_next code data}.
```

L'énoncé exprime qu'il existe un terme `code` satisfaisant le prédicat `adequate`. Le programme utilise successivement la séquence, `take` puis `read`.
La preuve est donc un appel aux trois règles associées :



Nous commençons par produire une variable existentielle représentant le programme que nous voulons produire et que nous instancions progressivement tout au long de la preuve. La règle de composition instancie la variable existentielle avec la séquence et génère deux autres variables existentielles pour chacun des deux sous-programmes. Le premier

sous-programme doit être `adequate` avec `take`. Nous utilisons donc directement la règle de `take` avec les bons arguments. Le deuxième doit être `adequate` avec `read`. Nous savons grâce à la relation de la règle de `take` que le `span` utilisé est bien formé par rapport aux données. Nous pouvons donc utiliser la règle de `read`. Le raffinement est alors terminé et produit un terme contenant explicitement les vérifications effectuées dans l'évaluation de `decode_next` (Section 4.3) :

```
let% s :=
 let% l := Length in
 If Const 1 <=% Var l
 Then
  Take (Const 1)
 Else
  Fail
in
If Const 0 <% Len (Var s)
Then
 Read (Var s) (Const 0)
Else
 Fail
```

```
fun (data : list N) (s : span) ⇒
 match
 (if 1 <=? len s
  then
   ...
  else
   ...) with
 | Some (s, t) ⇒
  if 0 <? len s
  then
   ...
  else
   ...
  | None ⇒ ...
 end
```

Nous pouvons remarquer que la vérification précédant le `Read` est inutile. En effet, nous produisons un `span` donnant accès à une et une seule donnée, puis vérifions qu'il donne accès à plus de zéro donnée. La condition est donc toujours vérifiée.

Pour détecter l'inutilité de cette vérification, nous définissons une deuxième règle pour `read`. L'objectif est de définir une règle où `read` est directement `adequate` avec `Read`. Pour que cette règle soit vraie, il suffit d'ajouter l'hypothèse que la condition de la vérification est vraie. Pour l'exprimer, nous utilisons la relation `span_eq_take data n0`. L'argument `n0` nous indique le nombre de données accessibles depuis le `span`. Il suffit donc d'exiger que l'argument de `read` soit strictement inférieur à `n0` :

```
Lemma read2_adequate hs hn data n n0 s :
    span_eq_take data n0 s s →
    sem_VAL hs s →
    sem_VAL hn n →
    n < n0 →
    adequate eq (read s n) (Read hs hn) data.
```

Pour automatiser la preuve de cette hypothèse supplémentaire, nous utilisons la tactique `lia` de `Coq` permettant de prouver des propositions d'arithmétique linéaire. Dans le cas de `decode_next`, elle réussit à prouver que `0 < 1`. Nous obtenons alors une nouvelle version réifiée :

```
let% s :=
 let% l := Length in
 If Const 1 <=% Var l
 Then
  Take (Const 1)
 Else
  Fail
in Read (Var s) (Const 0)
```

Nous connaissons maintenant un terme `adequate` à `decode_next` et ajoutons donc une nouvelle règle :

```
Lemma decode_next_adequate data :
  adequate eq decode_next (`decode_next_DF) data.
```

La projection ` permet de récupérer le terme produit par raffinement.

Raffinement `decodeFM_packet_SSH`

Pour raffiner `decode_packet_SSH`, il nous reste à définir des règles d'inférence pour le branchement booléen, `fail` et `decode_u32`. Commençons par `decode_u32` :

```
Definition decode_u32 :=
  let! a := decode_next in
  let! b := decode_next in
  let! c := decode_next in
  let! d := decode_next in
  ret (to_u32 a b c d).
```

Nous voulons que les valeurs produites par les deux représentations soient égales :

```
Definition decode_u32_DF : { code | ∀ data, adequate eq decode_u32 code data}.
```

Nous avons les règles d'inférence de `decode_next`, de la séquence et celle des valeurs. Nous renvoyons une valeur produite par appel à la fonction `to_u32` que nous avons dans la représentation. La réification est automatique et le terme produit est le suivant :

```
let% v := `decode_next_DF in
let% v0 := `decode_next_DF in
let% v1 := `decode_next_DF in
let% v2 := `decode_next_DF in
Val (ToU32 (Var v) (Var v0) (Var v1) (Var v2))
```

Nous pouvons alors définir la règle de `decode_u32` :

```
Lemma decode_u32_adequate data : adequate eq decode_u32 (`decode_u32_DF) data.
```

Dans les deux sémantiques, l'échec ne produit pas de résultat et ne modifie pas l'état. Les opérations `fail` et `Fail` sont donc `adequate`s pour n'importe quelle relation :

```
Lemma fail_adequate R data : adequate R fail Fail data.
```

Les deux versions de `if then else` sont simplement un branchement booléen. Elles sont donc `adequate` par rapport à une relation si les sous-programmes le sont aussi par rapport à cette même relation :

```
Lemma ite_adequate hb data R b (et : Decodeur X) (ht : Expr Y) ef hf :
    sem_VAL hb b →
    (b = true → adequate R et ht data) →
    (b = false → adequate R ef hf data) →
    adequate R (if b then et else ef) (IfThenElse hb ht hf) data.
```

Nous avons maintenant toutes les règles nécessaires pour raffiner `decode_packet_SSH`. Pour rappel, la syntaxe du décodeur est la suivante :

```
Definition decode_packet_SSH : Decodeur packet_SSH :=
  let! packet_length := decode_u32 in
  let! padding_length := decode_next in
  if padding_length + 1 <=? packet_length
  then
    let! payload := take (packet_length - padding_length - 1) in
    let! padding := take padding_length in
    let! mac := take 20 in
    ret (mk_ssh packet_length padding_length payload mac)
  else
    fail.
```

De plus, comme nous l'expliquions Section 5.1.3, nous voulons produire une version réification renvoyant une paire de paire contenant les champs d'un enregistrement `packet_SSH` et respectant la relation suivante :

```
Definition RpacketSSH (s : packet_SSH) (v : (N * N) * (span * span)) :=
 v.1.1 = packet_length s ∧
 v.1.2 = padding_length s ∧
 v.2.1 = payload s ∧
 v.2.2 = mac s.
```

Nous produisons alors le nouveau programme à partir de l'énoncé suivant :

```
Definition decode_packet_SSH_DF :
  { code | ∀ data, adequate RpacketSSH decode_packet_SSH code data}.
```

L'effort de preuve lors du raffinement est alors de définir la valeur que nous voulons renvoyer dans le programme réifié. En effet, le raffinement s'arrête lorsque nous devons produire un programme `adequate` à `ret (mk_ssh packet_length padding_length payload mac)` respectant la relation `RpacketSSH`. Or, simplement en lisant le programme, notre système ne peut pas déduire la paire à renvoyer. Nous devons alors appliquer la règle `ret_adequate` en indiquant explicitement que la valeur renvoyée est

`(! (! Var vpacket_l, Var vpadding_l), (! Var vpayload, Var vmac))` où nous savons à cette étape du raffinement que `vpacket_l`, `vpadding_l`, `vpayload` et `vmac` sont respectivement égaux à `packet_length`, `padding_length`, `payload` et `mac`. La preuve que la relation est respectée est alors triviale. Le terme produit par raffinement est le suivant :

```
let% v := ‘decode_u32_DF in
let% v0 := ‘decode_next_DF in
If Var v0 +% Const 1 <=% Var v
Then
 let% v1 :=
  let% len := Length in
  If Var v -% Var v0 -% Const 1 <=% Var len
  Then
   Take (Var v -% Var v0 -% Const 1)
  Else
   Fail in
 let% v2 :=
  let% len := Length in
  If Var v0 <=% Var len
  Then
   Take (Var v0)
  Else
   Fail in
 let% v3 :=
  let% len := Length in
  If Const 20 <=% Var len
  Then
   Take (Const 20)
  Else
   Fail in
 Val (%(%Var v, Var v0), (%Var v1, Var v3))
Else
 Fail
```

5.2 Réification des décodeurs Nom

Avant d'enrichir la représentation dans le but de permettre la réification de plus de décodeurs, plusieurs défauts sont à relever sur la réification de `decode_packet_SSH`.

Tout d'abord, puisque nous faisons des analyses de cas sur notre sémantique, pour avoir l'égalité entre nos valeurs, nous devons être en mesure d'avoir l'égalité entre leur type. Pour cela, nous utilisons la technique des univers [59]. Cette technique nous permet d'obtenir l'égalité décidable entre les types et donc de les comparer. Nous abordons cet aspect dans la Section 5.2.1.

Ensuite, la relation `adequate` de la Section 5.1.3 ne permet pas d'obtenir des propriétés sur le `span` courant lors du décodage. Par conséquent, nous ne pouvons pas définir une seconde règle pour `take` qui, similairement à `read2_adequate`, permettrait d'éliminer la vérification. Pour palier à ce problème, nous enrichissons sa signature et ajustons les règles d'inférence dans la Section 5.2.2.

Pour finir, les termes que nous produisons ne sont pas paramétriques. Par conséquent, bien que les règles d'inférence permettent de l'éviter, nous avons par exemple la possibilité de définir des termes `adequate`s analysant le motif des variables. Nous nous retrouverions alors avec les mêmes problèmes sur le terme généré que sur celui du terme initial de type `NomG` : certaines constructions n'ont pas été réifiées et sont donc inaccessibles lors du parcours du programme, mais nous ne pouvons pas non plus générer de variables afin de traverser les continuations.

Nous ne proposons pas de réelle solution à ce problème. La difficulté vient de la nécessité de raisonner sur les termes produits et donc d'instancier les variables avec la définition utilisée dans la sémantique. Par exemple, la seconde prémisse de la règle de composition (`bind_adequate` dans la Section 5.1.4) illustre bien le problème :

```
(∀ res vres, R0 res vres → adequate R (ke res) (kh vres) data)
```

Pour propager les propriétés, `vres` doit respecter la relation `R0` spécifiant le calcul précédent. La variable est donc du type des valeurs produites (`ID` dans le cas précédent) et impose à la suite du programme (représenté par `kh`) d'avoir les variables instanciées avec ce même type.

Pour contourner le problème, nous utilisons le fait qu'il est très aisé de montrer qu'un programme ne dépend pas de la définition des variables. À l'aide d'une relation inductive, nous montrons alors que le terme produit est bien indépendant de la définition des variables et générons simultanément un nouveau terme syntaxiquement identique et cette fois-ci paramétrique. Nous présentons la relation dans la Section 5.3.

5.2.1 Syntaxe et sémantique complètes [🐓]

Pour obtenir l'égalité décidable entre les types des valeurs que nous manipulons, nous les décrivons dans un type de données :

```
Inductive type : Type :=
| Nat
| NatN : N → type
| Bool
| Vector : type → type
| String
| Unit
| Span
| Unknown : string → type
| Option : type → type
| Pair : type → type → type
| Sum : type → type → type.
```

pour lequel chaque constructeur correspond à un type :

```
Fixpoint type_to_Type (ty : type) : Type :=
  match ty with
  | Nat ⇒ N
  | NatN n ⇒ natN n
  | Bool ⇒ bool
  | Vector ty ⇒ VECTOR (type_to_Type ty)
  | String ⇒ string
  | Unit ⇒ unit
  | Span ⇒ span
  | Option ty ⇒ option (type_to_Type ty)
  | Pair ty0 ty1 ⇒ type_to_Type ty0 * type_to_Type ty1
  | Sum ty0 ty1 ⇒ type_to_Type ty0 + type_to_Type ty1
  | Unknown s ⇒ True
  end.
```

Le constructeur `Unknown ty` représente un type inconnu se nommant `ty`. Dans l'objectif de produire du code dans un langage plus bas-niveaux, nous voulons avoir la possibilité de produire des structures déjà existantes dans une bibliothèque externe. Pour cela, nous ajoutons le constructeur `ExternStruct ty name l` à la représentation. Les arguments `ty` et `name` sont deux chaînes de caractères représentant respectivement le type (dans le langage cible) de la valeur produite et le nom de la fonction produisant cette valeur. L'argument `l` est la liste des arguments de la fonction.

Nous définissons une représentation plus riche. Excepté son type qui est maintenant `PHOAS : (type → Type) → type → Type`, la structure de données est similaire à celle de la Section 5.1.1. Par conséquent, nous ne rappelons pas explicitement sa construction,

mais nous présentons l'ensemble des fonctionnalités présentes dans la représentation Figure 5.3.

Les valeurs de notre sémantique ont le type calculé par `type_to_Type`. Par conséquent, la sémantique a la signature suivante :

```
Definition val := type_to_Type.

Inductive sem_PHOAS (data : list nat8):
  ∀ {X : type}, span → @PHOAS val X  → option (val X * span) → Prop
```

Les sémantique des opérateurs `Take` et `Read` sont les mêmes que dans l'exemple de la Section 5.1. Les autres opérateurs ont la même sémantique que dans la sémantique opérationnelle de `NomG`. Nous pouvons remarquer qu'étant maintenant dans `Prop`, nous n'avons plus besoin d'un `fuel` malgré la possibilité d'écrire des programmes dont l'évaluation ne termine pas.

La seule règle particulière est celle de `ExternStruct`. Nous n'avons aucune information sur la valeur produite. Nous supposons cependant que la fonction ne peut pas échouer. N'offrant aucun moyen de récupérer l'état courant dans le calcul (seulement des informations), nous savons que la fonction externe ne modifie pas l'état du programme. Nous la spécifions donc ainsi :

```
| SExternStruct : ∀ ty f l v s,
    sem_PHOAS data s (ExternStruct ty f l) (Some (v, s))
```

5.2.2 Raffinement [🐓]

Rappelons les points importants vis-à-vis de la relation `adequate` présentée pendant la réification de `decode_packet_SSH` (Section 5.1).

Afin d'éviter des comportements dangereux lorsque le programme d'origine fait appel à `take` ou `read`, la relation oblige le programme réifié à effectuer les vérifications nécessaires. Ces vérifications sont alors explicites dans le programme.

Lorsque nous utilisons la règle de la séquence, la relation en argument d'`adequate` sur les deux résultats des programmes permet de garder une trace du programme à travers des propriétés sur les résultats intermédiaires. Nous enrichissons alors les règles d'inférence en incluant dans cette relation des propriétés potentiellement utiles lors de l'analyse de la suite du programme (par exemple, nous ajoutons dans la règle `take_adequate` que le `span` produit est bien formé par rapport aux données). Par conséquent, directement lors du parcours du programme effectué pour le raffinement, nous pouvons "gratuitement" profiter de ces propriétés et éliminer certaines vérifications redondantes. L'élimination de ces vérifications arithmétiques se fait en utilisant la tactique `lia`. Si la tactique échoue, nous n'éliminons pas la vérification.

Pour les décodeurs `Nom`, nous effectuons deux modifications sur la relation `adequate` :

— Nous voulons raisonner sur l'état du programme. Nous donnons donc le `span` d'entrée explicitement et ne le quantifions donc plus universellement. De plus, nous

Variables	x			
Constantes	c	$::=$	n	$(n \in \texttt{N})$
			$\mid bin$	$(bin \in \texttt{natN})$
			$\mid b$	$(b \in \texttt{bool})$
			$\mid s$	$(s \in \texttt{string})$
			$\mid tt$	$(tt \in \texttt{unit})$
Opérateurs	o	$::=$	$v + v \mid v - v \mid v \times v \mid v \div v \mid v \textbf{ mod } v$	(opérateurs arithmétiques)
			$\mid v = v \mid v \leq v \mid v \geq v$	(comparaisons d'entier)
			$\mid \neg\, v \mid v \wedge v \mid v \vee v$	(opérateurs booléens)
			$\mid \downarrow v$	(cast de `natN` vers `N`)
			$\mid \textbf{inl } v \mid \textbf{inr } v$	(constructeurs de la somme)
			$\mid \textbf{pair } v\ v$	(constructeur des paires)
			$\mid \textbf{fst } v \mid \textbf{snd } v$	(projections des paires)
			$\mid \textbf{strlen } v$	(longueur d'une chaîne de caractères)
			$\mid \textbf{strget } v\ v$	(récupère un caractère de la chaîne)
			$\mid \textbf{len } v$	(longueur d'un `span`)
			$\mid \textbf{None} \mid \textbf{Some } v$	(constructeurs du type option)
			$\mid \textbf{make } v$	(produit un vecteur)
			$\mid \textbf{get } v\ v$	(récupère une valeur dans un vecteur)
			$\mid \textbf{add } v\ v$	(ajoute une valeur à un vecteur)
			$\mid v \uparrow v$	(concaténation de `natN`)
Valeurs	v	$::=$	$x \mid c \mid o$	
Branches	b	$::=$	$\textbf{default } e \mid \textbf{case } c : e;\ b$	
Expressions	e	$::=$	v	
			$\mid \textbf{let } v := e \textbf{ in } e$	
			$\mid \textbf{If } v \textbf{ Then } e \textbf{ Else } e$	(élimination booléenne)
			$\mid \textbf{case } v : e\ ;(v \Rightarrow e)$	(élimination optionnelle)
			$\mid \textbf{switch } v\ b$	(élimination des entiers naturels)
			$\mid \textbf{fail}$	
			$\mid \textbf{take } v$	
			$\mid \textbf{length}$	
			$\mid \textbf{read } v\ v$	
			$\mid \textbf{alt } e\ e$	
			$\mid \textbf{local } v\ e$	
			$\mid \textbf{repeat } v\ (v \Rightarrow e)\ v$	
			$\mid \textbf{ExternStruct } c\ c\ \vec{v}$	

Figure 5.3 – Syntaxe PHOAS

enrichissons la relation en argument pour qu'elle permette également de décrire le `span` de sortie.
— La sémantique a changé et les programmes peuvent ne plus terminer. Cependant, tous les décodeurs doivent normalement terminer. Nous ne nous intéressons donc pas particulièrement à cet aspect, mais devons quand même raisonner avec le fuel de la sémantique opérationnelle de `NomG`. Nous le quantifions alors existentiellement indiquant que les programmes terminent.

La relation pour les programmes Nom est alors la suivante :

```
Definition adequate {X Y} (R : span → X → type_to_Type Y → Prop)
  (n : NomG X) (e : PHOAS Y) (data : list nat8) (s : span) :=
  span_data_wf data s →
  ∀ res,
    sem_PHOAS data s e res →
    match res with
    | None ⇒
        ∃ fuel, run fuel n data s = NoRes
    | Some (v, t) ⇒
        ∃ r, R t r v ∧
        ∃ fuel, run fuel n data s = Res (t, r)
    end.
```

Un corollaire intéressant de cette règle d'adéquation est que si nous avons montré que :
— l'évaluation du programme `NomG` respecte le prédicat `all_disjointM` avec la logique de programme de la Section 4.5
— la relation entre les deux résultats produits conserve le prédicat sur le résultat

alors l'évaluation du programme réifié respecte le prédicat `all_disjointM` :

```
Lemma adequacy_disjoint `{Foldable X} `{Foldable (fun s ⇒ val (Y s))}:

  ∀ (e : NomG (X span)) (h : PHOAS (Y span)),
    {{ emp }} e {{ v; ⌜all_disjointM v⌝ }} →

    ∀ R data s,
      adequate R e h data s →

      ∀ v sres,
        sem_PHOAS data s h (Some (v,sres)) →
        span_data_wf data s →
        (∀ x, all_disjointM x → R sres x v → all_disjointM v) →
        all_disjointM v.
```

Règles de la logique [🐓]

Nous regroupons Figure 5.4 des règles reliant des programmes structurellement similaires. Ces règles n'effectuent aucune transformation, mais restent intéressantes dans la réification car elles produisent des informations que nous pouvons utiliser pour effectuer des transformations plus tard dans le programme. Par exemple, la règle `length_adequate` associe les deux versions de l'opérateur permettant de connaître le nombre de données restant à traiter. Cette transformation ne nous apporte rien lorsque nous faisons une analyse du programme puisque les opérations monadiques apparaissent déjà quand nous analysons les programmes `NomG`. Cependant, nous produisons trois informations : l'état n'a pas été modifié, les deux programmes renvoient la même valeur et, surtout, ils renvoient le nombre de données restantes. Ainsi pour la suite du raffinement, nous avons une variable contenant le nombre de données. Nous pouvons alors garder une trace des comparaisons effectuées sur cette variable et mieux connaître le contexte dans lequel nous nous trouvons.

Nous réunissons Figure 5.5 les deux règles de `take` et les deux règles de `read`. Les règles `read_verif_adequate` et `take_verif_adequate` sont fidèles à l'interprétation de `NomG` en vérifiant explicitement dans le programme que les opérateurs peuvent être appelés. Les règles `read_adequate` et `take_adequate` suppriment les vérifications à la condition d'avoir une preuve qu'elles sont respectées. Une preuve que nous obtenons grâce aux informations propagées notamment par les opérations précédentes.

Pour illustrer cette propagation d'information, nous prenons comme exemple le combinateur `rest`. Le rôle de ce combinateur est de renvoyer un `span` donnant accès à toutes les données restant à décoder. Pour cela, nous le définissons ainsi :

```
Definition rest : NomG span :=
  let! l := length in
  take l.
```

Nous commençons par récupérer le nombre de données restant à traiter puis nous produisons un accès vers ces données.

Nous spécifions alors l'opérateur et décrivons la relation entre les valeurs renvoyées par les deux versions. Tout d'abord, indépendamment de la réification, nous nous attendons à ce qu'il ne reste plus aucune donnée à décoder et que le `span` renvoyé soit bien formé par rapport aux données. Ensuite, nous voulons que la réification renvoie le même résultat. Nous spécifions tout cela avec le prédicat suivant :

```
Definition rest_spec data t x y := len t = 0 ∧ span_eq data x y ∧ x = y.
```

Comme pour l'exemple (Section 5.1), le lemme de réification est défini avec un sigma type :

```
Definition rest_adequate_DF :
  {code | ∀ data s, adequate (rest_spec data) rest code data s}.
```

```
Lemma ret_adequate data s
  (R : span → X → type_to_Type Y → Prop) (v : X) (va : VAL Y) a :
  sem_VAL va a →
  R s v a →
  adequate R (ret v) (Val va) data s.

Lemma bind_adequate data s T R
  (e : NomG X) (ke : X → NomG Y)
  (h : PHOAS X0) (kh : val X0 → PHOAS Y0) :
    adequate T e h data s →
    (∀ vres r t, T t r vres → adequate R (ke r) (kh vres) data t) →
    adequate R (let! v := e in ke v) (let% v := h in kh v) data s.

Definition length_spec s t x y := s = t ∧ x = y ∧ x = len t.

Lemma length_adequate data s : adequate (length_spec s) length Length data s.

Lemma alt_adequate data s R (e0 : NomG X) e1 (h0 : PHOASV Y) h1 :
  adequate R e0 h0 data s →
  adequate R e1 h1 data s →
  adequate R (alt e0 e1) (Alt h0 h1) data s.

Lemma repeat_adequate data s (ho : VAL (Option Nat)) (hb : VAL Y)
  (R : span → X → type_to_Type Y → Prop) o e b he rb :
  sem_VAL hb rb →
  sem_VAL ho o →
  R s b rb →
  (∀ rv v t, R t v rv → adequate R (e v) (he rv) data t) →
  adequate R (repeat o e b) (Repeat ho he hb) data s.
```

Figure 5.4 – Exemples de règles sans transformation syntaxique

```
Definition span_eq data x y := x = y ∧ span_data_wf data x.

Definition span_eq_take data s n t x y : Prop :=
  span_eq data x y ∧ len x = n ∧ len t = len s - n.

Lemma take_verif_adequate data s (hn : VAL Nat) (n : N) :
  sem_VAL hn n →
  adequate (span_eq_take data s n) (take n)
    (let% len := Length in
     If hn <=! Var len
     Then Take hn
     Else Fail) data s.

Lemma take_adequate data s (hn : VAL Nat) (n : N) :
    sem_VAL hn n →
    n <= len s →
    adequate (span_eq_take data s n) (take n) (Take hn) data s.

Definition read_spec (s t : span) (x : nat8) y := s = t ∧ x = y.

Lemma read_verif_adequate data s ht hn n t :
  span_data_wf data t →
  sem_VAL ht t →
  sem_VAL hn n →
  adequate (read_spec s) (read t n)
    (If hn <! EUna ELen ht
     Then Read ht hn
     Else Fail) data s.

Lemma read_adequate data s ht hn n t :
  span_data_wf data t →
  sem_VAL ht t →
  sem_VAL hn n →
  n < len t →
  adequate (read_spec s) (read t n) (Read ht hn) data s.
```

FIGURE 5.5 – Règles de `take` et `read`

Nous indiquons alors à Coq qu'il existe effectivement un terme respectant cette relation et que nous allons la définir tout au long de la preuve. Coq la représente alors avec une variable existentielle et nous demande de montrer l'assertion suivante :

```
adequate (rest_spec data) (let! l := length in take l) ?code data s
```

Après avoir utilisé la règle de la séquence et de `length`, nous nous retrouvons dans la situation ci-dessous :

```
s = t ∧ r = vres ∧ r = len t →
 adequate (rest_spec data) (take r) (?Goal vres) data t
```

où `r` est le résultat de `length`, `vres` celui de la version réifiée et `t` est le nouveau `span` courant.

Pour terminer la réification, nous pouvons appeler la règle supprimant la vérification de `take`. En effet, nous devons montrer que `r <= len t` et savons que `r = len t`. La réification est alors terminée et le programme produit est le suivant :

```
let% v := Length in
Take (Var v)
```

Pour finir, nous présentons un dernier ensemble de règles (Figure 5.6) permettant l'analyse du programme. Le rôle de ces règles est de réifier les éliminations Coq utilisées dans le programme où nous devons produire un programme `adequate` pour chaque cas. Nous pouvons donner du contexte lors de la réification de chaque cas en indiquant quel filtrage a réussi. Dans le cas du `if _ then _ else`, nous indiquons si le booléen est égal à `true` ou `false`.

5.3 Paramétricité des variables dans les programmes générés [❦]

Comme nous l'avons abordé au début de la Section 5.2, les programmes produits par raffinement ne sont pas paramétriques sur la définition des variables malgré le fait que la définition de la structure de données le permette. En effet, les programmes que nous générons ont des variables de type `val`. Cette instanciation nous pose deux problèmes.

Tout d'abord, nous ne pouvons pas raisonnablement créer des variables arbitraires de type `val`. Par conséquent, nous ne pouvons pas traverser les lieurs lorsque nous voulons parcourir un programme produit par raffinement. Ensuite, nous avons la possibilité de créer des termes exotiques, c'est-à-dire des termes manipulant la définition de `val`. Certaines constructions peuvent donc être invisibles lorsque nous parcourons la représentation d'un programme. En pratique (bien qu'il reste possible de construire des termes exotiques), les règles et les tactiques limitent les risques de produire accidentellement de tels termes.

Pour palier à ces deux problèmes, nous devons montrer que la structure du programme ne dépend pas de la définition des variables et en définir une nouvelle syntaxiquement équivalente et paramétrique.

```
Lemma ite_adequate data s (hb : VAL Bool) b R
  (et : NomG X) (ht : PHOASV Y) ef hf :
  sem_VAL hb b →
  (b = true → adequate R et ht data s) →
  (b = false → adequate R ef hf data s) →
  adequate R (if b then et else ef) (If hb Then ht Else hf) data s.

Fixpoint case_adequate (R : span → X → Y → Prop) (e : N → NomG X)
  (c : case_switch Y) (l : list N) data s : Prop :=
  match c with
  | LSnil p ⇒
   ∀ n, n ∉ l → adequate R (e n) p data s
  | LScons n p c ⇒
   adequate R (e n) p data s ∧ case_adequate R e c (n :: l) data s
  end.

Lemma match_N_adequate data s
  (hn : VAL Nat) n R (cases : case_switch Y)
  (e1 : NomG X) e2 :
  sem_VAL hn n →
  case_adequate R (fun n ⇒ match n with
                        | 0 ⇒ e1
                        | Npos p ⇒ e2 p
                        end) cases nil data s →
  adequate R (match n with
              | 0 ⇒ e1
              | Npos p ⇒ e2 p
              end) (Switch hn cases) data s.

Lemma LScons_adequate data s n R cases (e : N → NomG X) (h : PHOAS Y) l :
    adequate R (e n) h data s →
    case_adequate R e cases (n :: l) data s →
    case_adequate R e (LScons n h cases) l data s.

Lemma LSnil_adequate data s R (e : N → NomG X) (h : PHOAS Y) l :
    (∀ n, n ∉ l → adequate R (e n) h data s) →
    case_adequate R e (LSnil h) l data s.
```

FIGURE 5.6 – Exemples de règles permettant la réification

Pour cela, nous définissons une relation d'équivalence entre deux programmes avec deux instanciations potentiellement différentes des variables [18]. Deux programmes sont équivalents s'ils sont syntaxiquement équivalents malgré des variables de types différents. Par conséquent, si deux programmes sont équivalents, alors les opérations du programme ne dépendent pas de la définition des variables. Pour garantir que la syntaxe est équivalente, nous devons montrer que les mêmes opérations sont appelées dans le même ordre et que les variables utilisées représentent les résultats des mêmes opérations.

Pour s'assurer que les liaisons dans les deux programmes sont similaires, nous définissons un environnement stockant des couples de variables :

```
Inductive Env :=
| Nil : Env
| Cons : ∀ {X}, var1 X → var2 X → Env → Env.
```

Informellement, deux variables forment un couple dans l'environnement si elles représentent la même liaison. Nous définissons un prédicat indiquant si deux variables forment un couple :

```
Inductive In {X} (v1 : var1 X) (v2 : var2 X) : Env → Prop :=
| Here : ∀ env, In v1 v2 (Cons v1 v2 env)
| Later : ∀ X (t1 : var1 X) (t2 : var2 X) env,
    In v1 v2 env →
    In v1 v2 (Cons t1 t2 env).
```

Nous pouvons alors définir l'équivalence entre deux programmes. La relation utilise un environnement contenant toutes les variables générées par les programmes. Nous commençons par les valeurs où le seul cas intéressant est celui des variables :

```
Inductive equiv_VAL : Env → ∀ X, @VAL var1 X → @VAL var2 X → Prop :=
| EquivVar : ∀ X env v1 v2,
    In v1 v2 env →
    equiv_VAL env X (Var v1) (Var v2)

(...)
```

Deux appels à une variable sont équivalents si les deux variables représentent la même liaison, c'est-à-dire si elles forment un couple dans l'environnement. Pour les expressions, prenons le cas de la séquence :

```
Inductive equiv_prog : Env → ∀ X, @PHOAS var1 X → @PHOAS var2 X → Prop :=

| EquivLetIn : ∀ X Y env e0 e1 k0 k1,
    equiv_prog env X e0 e1 →
    (∀ v0 v1, equiv_prog (Cons v0 v1 env) Y (k0 v0) (k1 v1)) →
    equiv_prog env Y (LetIn e0 k0) (LetIn e1 k1)

(...)
```

Les premières instructions peuvent légitimement utiliser toutes les variables précédemment générées dans le programme. Nous nous attendons donc à ce que les deux instructions soient équivalentes avec le même environnement que celui de la séquence. Pour la suite de la séquence, nous devons générer une variable liant la valeur de la première instruction. Pour montrer l'équivalence, nous devons donc agrandir l'environnement des variables utilisables avec ces deux nouvelles variables que nous utilisons pour parcourir les lieurs.

Une fois cette relation définie, il est très aisé de générer un programme paramétrique sur la définition des variables à partir d'un programme non-exotique et non-paramétrique. La définition se fait alors avec un sigma-type et la génération en utilisant la tactique `econstructor` de Coq trouvant elle-même les constructeurs respectant la relation. Dans le cas de `rest`, nous obtenons la définition et la preuve suivante :

```
Lemma rest_equiv {var} :
  { code : @PHOAS var _ | equiv_prog Nil _ (`rest_adequate_DF) code}.
Proof. eapply exist. repeat econstructor. Qed.
```

et le terme produit est le suivant :

```
rest_equiv =
fun var : type → Type ⇒
  (let% v1 := Length in Take (Var v1))
  ↾ (...)
```

Le terme `rest_equiv` est donc paramétrique, nous garantissant qu'il n'est pas exotique et nous permettant de choisir librement le type des variables pour de futures manipulations.

5.4 Génération de code C [🐓]

Une fois que nous avons généré un programme paramétrique sur le type des variables, nous pouvons instancier le type des variables à un type nous permettant de générer des variables arbitraires. Ainsi, nous pouvons parcourir les lieurs de nos programmes et donc effectuer des transformations. Pour produire le code C, nous utilisons l'extraction de Coq afin d'obtenir la structure `PHOAS` en OCaml [🐓].

5.4.1 Compilation vers C

Afin de compiler les opérateurs vers C, nous définissons plusieurs types de données et fonctions C sur lesquels nous nous appuyons lors de la compilation. Par exemple, nous définissons les `spans` comme une structure contenant un pointeur vers des données et une longueur [🐓] :

```
typedef struct Span {
    uint8_t *pos;
    uint64_t length;
} span;
```

ou encore le type `option` que nous monomorphisons à l'aide de macro [🐓] :

```
#define DEFINE_OPTION_TYPE(_name,_type)                               \
    typedef struct _name {                                            \
        bool ok;                                                      \
        _type val;                                                    \
    } _name;

#define DEFINE_OPTION(_type)                       \
    DEFINE_OPTION_TYPE(option_##_type,_type);
```

La macro `DEFINE_OPTION` prend en argument le type de la valeur potentiellement contenue dans le type `option` et produit un nouveau type de donnée qui est une structure contenant deux membres. Le premier membre `ok` indique si le deuxième contient une valeur. Si c'est le cas, c'est alors la version C du constructeur `Some` et sinon du constructeur `None`. En définissant de nouveaux types de données (lorsque cela est nécessaire), nous pouvons alors associer un type C à chacun des types de notre représentation `PHOAS` (Section 5.2.1).

Il est aisé de compiler les valeurs de notre représentation `PHOAS`. Par exemple, pour le constructeur représentant la valeur `None` du type `option`, l'extraction produit le constructeur OCaml suivant :

```
| ENone of coq_type
```

où `coq_type` est le type de données produit par l'extraction des constructeurs de type `type` (Section 5.2.1). La compilation de `ENone` consiste alors simplement à créer une structure dont le membre `ok` vaut `false`.

Les décodeurs compilés ont une forme commune. Ils ont tous en premier argument un pointeur, que nous nommons `bin`, vers le `span` permettant d'accéder aux données à décoder et en dernier argument, un pointeur, que nous supposons initialisé, vers la valeur renvoyée par le décodeur. Pour l'illustrer, prenons comme exemple l'opérateur `take`. Pour compiler `take`, nous implémentons directement une fonction C ayant le comportement attendu de l'opérateur et l'appelons. Cette fonction a trois arguments : le premier argument est un pointeur vers l'accès aux données ; le deuxième est l'entier indiquant le nombre de données auxquelles nous voulons avoir accès ; et le dernier un pointeur vers le résultat, qui est l'accès mémoire résultant, et donc un `span` :

```
void take (span* bin, uint64_t size, span* pres) {
    span res;
    res.pos = bin→pos;
    res.length = size;
    bin→pos = bin→pos + size;
    bin→length = bin→length - size;
    *pres = res;
}
```

En utilisant les types de données C représentant chacun des types de la Section 5.2.1 et des fonctions telles que `take` ci-dessus, la compilation vers C n'a pas de difficulté particulière.

5.4.2 Brève analyse du code produit

Nous avons ré-implémenté plusieurs Rust de la bibliothèque rusticata dont Radius que nous avons présenté dans la Section 4.6. L'ensemble des tests fournis par rusticata des différents décodeurs ré-impléméntés ont été effectués sur les décodeurs C et tous les tests ont réussi.

Lisibilité

Les décodeurs C sont d'une taille beaucoup plus importante que leurs versions Rust et Gallina. En effet, le décodeur C de Radius fait 1705 lignes [🐓] alors que les versions en Rust et en Gallina [🐓] font environ 70 lignes. Les deux principales raisons sont que toutes les fonctions (et donc tous les combinateurs) sont dépliées et qu'un simple appel à `take` (que ce soit en Rust ou en Gallina) encapsule une vérification et donc un branchement booléen, là où dans la version C, tous les branchements sont maintenant explicites.

Par exemple, un appel au combinateur `be_u8` (donc une seule ligne) est compilé ainsi :

```
uint64_t var2 = bin->length;
span var3;
if (1 <= var2) {
 span var4;
 take(bin, 1, &var4);
 var3 = var4;
} else {
 return 0;
}
uint8_t var5;
var5 = var3.pos[0];
```

où `var5` est le résultat du combinateur et l'instruction `return 0;` signale une erreur lors du décodage.

Nous avons donc, dans ce cas, une seule ligne qui en devient 11. Lorsque nous utilisons les combinateurs `be_u16` ou `be_u32` le rapport entre le nombre de ligne Rust/Gallina est alors quasiment deux fois et quatre fois plus important. La lisibilité du code produit est donc plutôt mauvaise car il est de taille bien plus importante.

Performances

Pour mesurer l'efficacité du code produit, nous avons exécuté les décodeurs Rust et C sur les tests fournis par rusticata. Les tests étant des paquets de petites tailles, nous mesurons le temps nécessaire pour décoder 100.000.000 fois le paquet. Les mesures ont été

effectuées 10 fois en utilisant clang avec le mode -O3. Le temps que nous retenons est le temps moyen des 10 exécutions :

	décodeur C	décodeur rusticata (Rust)
radius	14.132 s	16.286 s
ssh : client_dh_init	1.698 s	2.690 s
ssh : server_dh_reply	2.610 s	3.420 s
ssh : new_keys	1.313 s	2.460 s

Bien que les mesures de performance soient insuffisantes pour établir une véritable conclusion sur l'efficacité globale des décodeurs produits, nous pouvons au moins observer que dans les cas de Radius et SSH, les décodeurs rivalisent en terme de performance avec les décodeurs de rusticata.

Pour de fin de reproductibilité, nous détaillons la procédure dans la section 5.4.3.

Certification

La phase de compilation n'est pas certifiée et fait donc partie des travaux futurs. Une possibilité pourrait être de compiler vers Clight [12], le langage cible du module de CompCert étudié dans le chapitre 3. Nous profiterions alors d'une sémantique pré-existante en plus d'appuis possibles grâce aux nombreux travaux scientifiques compilant déjà vers Clight.

5.4.3 Reproductibilité des tests

Dans cette sous-section, nous expliquons comment reproduire les tests de performance. Les mesures ont été faites avec la version `1.66.1 (ad779e08b 2023-01-10)` de cargo et la version `14.0.0-1ubuntu1` de clang sur un ordinateur portable avec une puissance de calcul relativement faible. Pour lancer les tests des codes C, il suffit d'exécuter la commande `make test` à la racine du github de cette thèse [🐓].

Radius

Pour le test de Radius, la mesure de performance pour le code Rust a été effectuée sur la version 0.5.0 du décodeur de Radius sur le github **radius-parser** de rusticata. Le fichier **main.rs** [🐓] effectuant le test de performance doit être ajouté dans le dossier **src** et il suffit ensuite d'utiliser la commande `cargo run --release` à la racine du github.

SSH

Pour les tests de SSH, la mesure de performance pour le code Rust a été effectuée sur la version 0.5.0 du décodeur de SSH sur le github **ssh-parser** de rusticata. La manipulation est similaire à celle de Radius. Le fichier **main.rs** [🐓] (même nom, mais localisation différente) doit être ajouté dans le dossier **src** et la commande `cargo run --release` doit être utilisée à la racine du github.

5.5 Travaux Connexes

Obtention de programmes performants

Afin d'obtenir du code bas-niveau efficace, Swamy et al. [74] utilisent Low*, un langage dédié bas-niveau intégré à F*. LowParse permet de garantir des propriétés de sûreté mémoire telles que l'absence de double désallocation. Contrairement à notre représentation intermédiaire qui ne permet que de représenter des décodeurs de la bibliothèque, LowParse est un véritable langage et est donc plus versatile et complexe. Ainsi, le langage n'est pas spécifique aux décodages et est utilisé pour produire d'autres types de programme. Afin d'obtenir de bonnes performances, LowParse est compilé vers du C grâce au compilateur KaRaMeL. Similairement à notre prototype de compilateur vers C, le code produit n'est donc pas certifié et le compilateur fait donc partie du code de confiance.

Afin d'obtenir les garanties mémoires, une méthode possible serait de compiler vers Bedrock [20]. Bedrock est une librairie Coq permettant d'implémenter et vérifier des programmes bas-niveaux. La librairie fournit un assembleur dont le comportement est entièrement spécifié et un ensemble de macros permettant de raisonner sur des programmes à un niveau d'abstraction proche du C. Comme EverParse le fait avec Low*, il devrait être possible de relier nos décodeurs haut-niveaux à des décodeurs bas-niveaux spécifiés à l'aide de Bedrock. Nous pourrions ainsi à la fois avoir une correction fonctionnelle des programmes bas-niveaux et de leur sûreté mémoire.

Afin d'obtenir des performances raisonnables de leurs programmes écrits dans l'assistant de preuves, Suriyakarn et al. [70], Tan et Morrisett [77], Ye et Delaware [88] exécutent les programmes OCaml produits par l'extraction Coq.

Plusieurs travaux visent à compiler des programmes Gallina vers des langages bas-niveaux tels que C. Tanaka et al. [78] fournissent un plugin permettant de compiler un sous-ensemble de Gallina en C. La compilation se fait en deux étapes. Tout d'abord, toutes les fonctions polymorphes sont monomorphisées. Ensuite, le programme est "monadifié". L'utilisateur fournit au plugin une monade dans laquelle les programmes doivent être définis puis ajoute une-à-une des fonctions pures ou impures au plugin qui sont transformées en C. Les auteurs affirment que les types de données ne doivent pas être automatiquement transformés pour des questions de performance. Ainsi, pour chaque type de données, un utilisateur doit en partie fournir le type C, un constructeur et des accesseurs de champs. L'article présente alors la compilation de différentes constructions Coq telle que le filtrage par motif ou l'application.

Anand et al. [5] produisent un compilateur de Gallina, écrit en Gallina, vers **Clight**, un langage intermédiaire du C. Ce compilateur a pour objectif de produire du code vérifié et est toujours en développement.

Syntaxe abstraite paramétrique d'ordre supérieur

Washburn et Weirich [84] étendent l'utilisation d'une syntaxe abstraite d'ordre supérieur (**HOAS**) pour définir des langages non-intégrés (c'est-à-dire utiliser les variables et

les lieurs du langage hôte) grâce au polymorphisme afin d'éviter la production de fonctions qui ne sont pas paramétriques.

Atkey et al. [6] reprennent cette idée et font le lien entre des termes HOAS non-exotiques et une représentation de Bruijn bien formée. Ils présentent ensuite différentes méthodes afin d'étendre le langage. Les extensions ajoutent des constructeurs aux langages afin de représenter des fonctionnalités similaires à celles que nous avons directement définies dans notre représentation intermédiaire (Section 5.2.1). Il serait profitable d'adapter leur approche afin de permettre l'extension de notre représentation.

De plus, les travaux présentés sont définis en Haskell [84, 6] et la définition du langage n'est pas implémentable telle quelle car elle nuirait à la cohérence logique de Coq [17]. Chlipala [18, 19] reprend cette approche de paramétricité introduite par Washburn et Weirich [84], mais ajuste la définition pour qu'elle soit compatible avec l'assistant de preuves Coq. Nous reprenons l'ajustement présenté par Chlipala pour définir la structure de données de la Section 5.2.1.

Raisonnement relationnel

De nombreux travaux ont été effectués pour construire des logiques de Hoare relationnelles et Naumann [55] en font un très bon résumé. Benton [10] introduit une logique de Hoare relationnelle entre deux programmes d'un même langage. L'objectif de cette logique est de prouver la correction de transformation de programmes telle que la suppression de branchements (similaire à nos règles `take_adequate` et `read_adequate`) ou la suppression de codes inutiles. Une partie des règles présentées concerne la théorie équationnelle du langage impératif présentée dans ce papier (lois monadiques, distributivité du `if _ then _ else`,...). Nous avons également une théorie équationnelle (Section 4.4.3). Nous devrions donc pouvoir identifier les motifs des équations et les appliquer lorsqu'elles optimisent les programmes.

5.6 Conclusion et limitations

Dans ce chapitre, nous avons implémenté un raffinement afin de réifier des décodeurs définis dans la formalisation de la bibliothèque Nom du chapitre 4. L'objectif est de permettre le parcours du programme en supprimant les continuations et de supprimer les vérifications redondantes de la sémantique opérationnelle (Section 4.4.4).

La méthode consiste à définir une structure intermédiaire permettant de représenter les fonctionnalités utilisées dans l'implémentation d'un décodeur. Cependant, nous modifions la sémantique de certains opérateurs dans la structure intermédiaire. En particulier, nous évaluons des comportements que la sémantique opérationnelle de Nom évite en effectuant des vérifications. À l'aide de la relation utilisée pour le raffinement, nous garantissons que le programme produit effectue quand même ces vérifications. Ces vérifications peuvent néanmoins être redondantes. Nous profitons alors du parcours du programme effectué durant le raffinement pour inférer des propriétés sur le contexte. Ainsi, avant de générer une véri-

fication, nous déterminons si les propriétés garantissent que celle-ci est toujours respectée et donc si elle est nécessaire. Les vérifications sont alors dans la syntaxe du programme et sont donc générées pendant le raffinement.

Pour résultat, nous obtenons alors un programme effectuant moins de vérifications que dans la sémantique opérationnelle. De plus, les continuations ont été supprimées et nous pouvons par conséquent parcourir le programme. Pour illustrer les transformations possibles, nous avons implémenté un prototype de compilateur produisant du code C.

5.6.1 Limitations

Les travaux contiennent plusieurs limitations.
Tout d'abord, nous ne pouvons pas étendre la représentation intermédiaire. Cette limitation n'est cependant pas une fatalité. En effet, comme énoncé dans la Section 5.5, des travaux proposent déjà des méthodes pour étendre des syntaxes abstraites paramétriques d'ordre supérieur et devraient donc être adaptables à notre cas.

D'autre part, il est important de pouvoir appeler des structures externes, notamment pour des structures définies en C où de nombreux choix de représentations sont possibles et souvent spécifiques à une implémentation. Cependant, lors du raffinement, nous n'avons aucune propriété sur ces structures nous imposant de sous-spécifier les programmes qui les manipulent. De plus, nous ne pouvons pas accéder à leurs champs. Par conséquent, après avoir construit une telle structure, les seules manipulations possibles sont de renvoyer la structure ou de l'inclure dans une autre. Un moyen peu satisfaisant d'éviter ces limitations est de construire la structure à la fin du programme et donc de manipuler indépendamment chacun de ces futurs champs.

En enrichissant le type de ces structures, nous devrions être en mesure de donner la possibilité aux utilisateurs de fournir des accesseurs. Le fait qu'accéder à un champ permette de renvoyer la valeur fournie pour construire la structure pourrait être une axiomatisation raisonnable des accesseurs et des structures. Ainsi, après avoir produit une structure, nous devrions être en mesure de récupérer la valeur de chacun des champs et donc de raisonner sur ces valeurs lors du raffinement.

Chapitre 6

Conclusion

6.1 Fraîcheur

6.1.1 Résumé de la contribution

Nous montrons dans le chapitre 3 que la logique de séparation est un bon outil pour raisonner *localement* sur des propriétés de fraîcheur.

Nous avons tout d'abord comparé notre approche aux autres travaux existant en utilisant le problème d'étiquetage d'arbre de Hutton et Fulger [32]. Nous commençons par raisonner de la même façon que dans le module `SimplExpr` de CompCert. Ensuite, nous ré-adaptons en Coq la preuve de Swierstra et Baanen [76] utilisant des transformateurs de prédicats. Finalement, nous utilisons la logique de séparation et dérivons, de nos preuves de correction, une spécification satisfaisante du problème.

Pour appuyer nos propos, nous appliquons l'approche sur le module `SimplExpr` de CompCert. Comme résultat, nous obtenons un module totalement intégré au reste du compilateur avec plusieurs améliorations : des spécifications plus concises et plus fonctionnelles ; des preuves de corrections simplifiées ; des preuves plus modulaires, indépendantes de tout raisonnement arithmétique.

6.1.2 Limitations

Une limitation de ces travaux est qu'ils ne se basent que sur un exemple extrêmement concis (le problème d'étiquetage) et sur un seul exemple de valeur (le module `SimplExpr` de CompCert). Afin d'étudier plus profondément l'approche et de légitimer les résultats de la première véritable étude de cas, il pourrait être utile que des travaux soient effectués sur d'autres cas. Le module `RTLgen` de CompCert pourrait être un très bon exemple puisque, plutôt que générer des variables, il génère à la fois des noms de registre et des labels assembleurs. Il devrait donc être profitable d'utiliser la logique de séparation pour spécifier les programmes. Un point essentiel est que la véritable difficulté est d'identifier les invariants indépendants de la fraîcheur mais dépendants de ces variables. Étudier d'autres cas devrait donc permettre d'évaluer si la logique de séparation est dans certains cas encom-

brante (notamment si nous avons besoin de décrire des éléments implicites de la logique de séparation) et également d'identifier de nouveaux opérateurs utiles (a minima pour la compilation certifiée lors des études sur les modules de CompCert).

6.1.3 Pistes d'amélioration

Les définitions liées à la monade sont plus nombreuses dans cette nouvelle version du module engendrant un surplus d'effort. Certaines de ces définitions supplémentaires sont généralisables à la monade libre. En profitant de la généricité apportée par cette construction, il est possible de définir une bibliothèque d'effets et de leurs logiques de programme. Les définitions indépendantes de la fraîcheur ne sont alors plus nécessaires dans CompCert. De plus, si la fraîcheur fait partie de cette bibliothèque, les seules définitions restantes sont celles propres à `SimplExpr` amenuisant considérablement le travail relatif à la monade.

6.2 Décodeurs zéro-copie et leur sûreté mémoire

6.2.1 Résumé de la contribution

Dans le chapitre 4, nous formalisons la notion de zéro-copie abordée dans les bibliothèques de décodeurs et dans les langages de description de données. Le principal argument mis en avant est l'efficacité des décodeurs produits. Nous présentons alors une formalisation de la bibliothèque zéro-copie Nom implémentée en Rust.

En reprenant l'essence du chapitre 3, nous accompagnons la bibliothèque d'une logique de programme définie avec la même logique de séparation permettant de garantir que les décodeurs sont zéro-copies.

6.2.2 Limitations

Cette logique de programme ne raisonne que sur l'aspect zéro-copie des décodeurs. Cependant, il est relativement aisé de l'étendre pour obtenir d'autres propriétés et notamment d'avoir des garanties fonctionnelles.

6.2.3 Pistes d'amélioration

Bien que les preuves de zéro-copie sont aisées et concises, elles restent relativement peu automatisées. Quelques efforts supplémentaires, notamment en définissant des tactiques, devraient permettre la quasi-complète automatisation des preuves de zéro-copie.

De plus, certaines constructions de la bibliothèque permettent la définition de décodeurs ne terminant pas. Bien qu'il est évident qu'un décodeur doit terminer, nous n'exigeons pas (et ne souhaitons pas) de preuve de terminaison. Par conséquent, la sémantique calculatoire est définie à l'aide d'un fuel. Il pourrait possiblement être intéressant d'étudier l'utilisation d'une construction co-inductive comme celle des Interaction Trees [87].

6.3 Raffinement des décodeurs Gallina en C

6.3.1 Résumé de la contribution

Dans l'objectif de produire des décodeurs efficaces, nous réifions partiellement les décodeurs dans le chapitre 5. Le résultat de cette réification est double. Tout d'abord, nous éliminons les continuations introduites dans les décodeurs par l'utilisation de la monade libre et permettons ainsi l'analyse de ces programmes. De plus, nous supprimons des vérifications redondantes effectuées dans la sémantique opérationnelle de Nom. Nous avons alors défini un prototype de compilateur vers C. Des décodeurs pour les protocoles Radius, SSH et Ipsec ont été implémentés et prouvés sûr. De plus, les décodeurs pour Radius et SSH ont été compilés vers C. Les décodeurs C obtenus sont alors peu lisibles, mais aussi efficaces que ceux de rusticata.

6.3.2 Limitations

La représentation intermédiaire vers laquelle le raffinement est effectué n'est pas extensible. Cependant, nous pouvons librement appeler des structures externes afin d'obtenir des structures parfaitement adaptées au format et donc de générer des programmes efficaces. Malgré tout, nous n'exigeons aucune propriétés sur ces structures. Ainsi, alors que le raffinement permet de garantir des propriétés sur les décodeurs raffinés, nous sommes obligés de sous-spécifier ceux manipulant des structures C.

De plus, le type des ces structures lors du raffinement est trop maigre. Par conséquent, nous n'avons aucun accesseur permettant de manipuler leurs champs. Les seules manipulations possibles sont alors de renvoyer la structure ou de l'inclure dans une autre.

6.3.3 Pistes d'amélioration

L'une des pistes d'amélioration est d'adapter les travaux présentés par Atkey et al. [6] afin de rendre extensible la représentation intermédiaire.

De plus, la phase de réification est actuellement plus exigeante que souhaité. En effet, le raffinement est plus long en terme de ligne que la définition des décodeurs que nous voulons raffiner. Avec un travail important sur l'automatisation, il devrait être possible d'obtenir un raffinement beaucoup mieux accompagné et donc beaucoup plus concis. Pour cela, une manière directe d'augmenter l'automatisation serait d'écrire un programme, dans le méta-langage `Ltac`, effectuant les étapes les plus prévisibles du raffinement. L'utilisateur n'aurait plus qu'à interagir pendant le raffinement lorsqu'il y a de véritables choix. Par exemple, si une structure doit être utilisée.

Actuellement, la phase de compilation de la représentation intermédiaire vers C n'est pas certifiée. Pour garantir que les propriétés obtenues lors du raffinement sont conservées, il faudrait définir un compilateur certifié. Une cible adéquate pourrait être le langage Clight formalisé dans Coq. En plus d'avoir une formalisation complète, ce langage est déjà la cible

de nombreux compilateurs dont celui du module `SimplExpr` que nous étudions lors de l'étude de cas du chapitre 3.

6.4 Contribution logicielle

Dans cette thèse, nous présentons une bibliothèque de combinateurs en Gallina. Cette bibliothèque permet de décrire des décodeurs de paquets dans Coq et donc de raisonner, dans l'assistant de preuve, sur ces décodeurs. Nous offrons aux utilisateurs de la bibliothèque une logique de programme permettant de montrer qu'un décodeur est zéro-copie.

De plus, nous proposons une représentation intermédiaire des décodeurs et un compilateur de cette représentation vers C. Pour obtenir des décodeurs dans cette représentation intermédiaire, nous l'accompagnons d'une logique et d'une collection de règles permettant de les produire par raffinement.

Bibliographie

[1] Reynald Affeldt et David Nowak. Extending equational monadic reasoning with monad transformers. In Ugo de'Liguoro, Stefano Berardi, et Thorsten Altenkirch, editors, *26th International Conference on Types for Proofs and Programs, TYPES 2020, March 2-5, 2020, University of Turin, Italy*, volume 188 of *LIPIcs*, pages 2 :1–2 :21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi : 10.4230/LIPIcs.TYPES.2020.2. URL `https://doi.org/10.4230/LIPIcs.TYPES.2020.2`.

[2] Reynald Affeldt, David Nowak, et Takafumi Saikawa. A hierarchy of monadic effects for program verification using equational reasoning. In Graham Hutton, editor, *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*, volume 11825 of *Lecture Notes in Computer Science*, pages 226–254. Springer, 2019. doi : 10.1007/978-3-030-33636-3_9. URL `https://doi.org/10.1007/978-3-030-33636-3_9`.

[3] Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martínez, Gordon D. Plotkin, Jonathan Protzenko, Aseem Rastogi, et Nikhil Swamy. Dijkstra monads for free. In Giuseppe Castagna et Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 515–529. ACM, 2017. doi : 10.1145/3093333.3009878.

[4] Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, et Nikhil Swamy. Recalling a witness : foundations and applications of monotonic state. *Proc. ACM Program. Lang.*, 2(POPL) :65 :1–65 :30, 2018. doi : 10.1145/3158153. URL `https://doi.org/10.1145/3158153`.

[5] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, et Matthew Weaver. Certicoq : A verified compiler for coq. In *The third international workshop on Coq for programming languages (CoqPL)*, 2017.

[6] Robert Atkey, Sam Lindley, et Jeremy Yallop. Unembedding domain-specific languages. In Stephanie Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages

37–48. ACM, 2009. doi : 10.1145/1596638.1596644. URL https://doi.org/10.1145/1596638.1596644.

[7] Anne Baanen et Wouter Swierstra. Combining predicate transformer semantics for effects : a case study in parsing regular languages. In Max S. New et Sam Lindley, editors, *Proceedings Eighth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2020, Dublin, Ireland, 25th April 2020*, volume 317 of *EPTCS*, pages 39–56, 2020. doi : 10.4204/EPTCS.317.3.

[8] Godmar Back. Datascript - A specification and scripting language for binary data. In Don S. Batory, Charles Consel, et Walid Taha, editors, *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings*, volume 2487 of *Lecture Notes in Computer Science*, pages 66–77. Springer, 2002. doi : 10.1007/3-540-45821-2_4. URL https://doi.org/10.1007/3-540-45821-2_4.

[9] Julian Bangert et Nickolai Zeldovich. Nail : A practical tool for parsing and generating data formats. *login Usenix Mag.*, 40(1), 2015. URL https://www.usenix.org/publications/login/feb15/bangert.

[10] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In Neil D. Jones et Xavier Leroy, editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 14–25. ACM, 2004. doi : 10.1145/964001.964003. URL https://doi.org/10.1145/964001.964003.

[11] Sandrine Blazy et Xavier Leroy. Formal verification of a memory model for C-like imperative languages. In *International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2005.

[12] Sandrine Blazy et Xavier Leroy. Mechanized semantics for the clight subset of the C language. *J. Autom. Reason.*, 43(3) :263–288, 2009. doi : 10.1007/s10817-009-9148-3. URL https://doi.org/10.1007/s10817-009-9148-3.

[13] Sylvain Boulmé. Intuitionistic refinement calculus. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *Lecture Notes in Computer Science*, pages 54–69. Springer, 2007. doi : 10.1007/978-3-540-73228-0_6. URL https://doi.org/10.1007/978-3-540-73228-0_6.

[14] Paolo Capriotti et Huw Campbell. URL https://github.com/pcapriotti/optparse-applicative.

[15] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Manuel M. T. Chakravarty, Zhenjiang Hu, et Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 418–430. ACM, 2011. doi : 10.1145/2034773.2034828. URL `https://doi.org/10.1145/2034773.2034828`.

[16] Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *Proc. ACM Program. Lang.*, 4(ICFP) :116 :1–116 :34, 2020. doi : 10.1145/3408998. URL `https://doi.org/10.1145/3408998`.

[17] Adam Chlipala. URL `http://adam.chlipala.net/cpdt/html/Cpdt.InductiveTypes.html`.

[18] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook et Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143–156. ACM, 2008. doi : 10.1145/1411204.1411226. URL `https://doi.org/10.1145/1411204.1411226`.

[19] Adam Chlipala. A verified compiler for an impure functional language. In Manuel V. Hermenegildo et Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 93–106. ACM, 2010. doi : 10.1145/1706299.1706312. URL `https://doi.org/10.1145/1706299.1706312`.

[20] Adam Chlipala. The bedrock structured programming system : combining generative metaprogramming and hoare logic in an extensible program verifier. In Greg Morrisett et Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 391–402. ACM, 2013. doi : 10.1145/2500365.2500592. URL `https://doi.org/10.1145/2500365.2500592`.

[21] Geoffroy Couprie. URL `https://github.com/Geal/nom/blob/main/doc/choosing_a_combinator.md`.

[22] Geoffroy Couprie. Nom, A byte oriented, streaming, zero copy, parser combinators library in rust. In *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*, pages 142–148. IEEE Computer Society, 2015. doi : 10.1109/SPW.2015.31. URL `https://doi.org/10.1109/SPW.2015.31`.

[23] Willem P. de Roever et Kai Engelhardt. *Data Refinement : Model-oriented Proof Theories and their Comparison*, volume 46 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998. ISBN 0-521-64170-5.

[24] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, et Adam Chlipala. Fiat : Deductive synthesis of abstract data types in a proof assistant. In Sriram K. Rajamani

et David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 689–700. ACM, 2015. doi : 10.1145/2676726.2677006. URL `https://doi.org/10.1145/2676726.2677006`.

[25] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8) :453–457, 1975. doi : 10.1145/360933.360975. URL `https://doi.org/10.1145/360933.360975`.

[26] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et Vern Paxson. The matter of heartbleed. In Carey Williamson, Aditya Akella, et Nina Taft, editors, *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*, pages 475–488. ACM, 2014. doi : 10.1145/2663716.2663755. URL `https://doi.org/10.1145/2663716.2663755`.

[27] Kathleen Fisher et David Walker. The PADS project : an overview. In Tova Milo, editor, *Database Theory - ICDT 2011, 14th International Conference, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 11–17. ACM, 2011. doi : 10.1145/1938551.1938556. URL `https://doi.org/10.1145/1938551.1938556`.

[28] Kathleen Fisher, Yitzhak Mandelbaum, et David Walker. The next 700 data description languages. *J. ACM*, 57(2) :10 :1–10 :51, 2010. doi : 10.1145/1667053.1667059. URL `https://doi.org/10.1145/1667053.1667059`.

[29] Jeremy Gibbons et Ralf Hinze. Just do it : simple monadic equational reasoning. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 2–14, 2011. doi : 10.1145/2034773.2034777.

[30] Michael Hicks, Gavin M. Bierman, Nataliya Guts, Daan Leijen, et Nikhil Swamy. Polymonadic programming. In Paul Levy et Neel Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*, volume 153 of *EPTCS*, pages 79–99, 2014. doi : 10.4204/EPTCS.153.7. URL `https://doi.org/10.4204/EPTCS.153.7`.

[31] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12 (10) :576–580, 1969. doi : 10.1145/363235.363259.

[32] Graham Hutton et Diana Fulger. Reasoning About Effects : Seeing the Wood Through the Trees. In *Proceedings of the Symposium on Trends in Functional Programming*, Nijmegen, The Netherlands, May 2008.

[33] Graham Hutton et Erik Meijer. Monadic parser combinators, 1996.

[34] Stephen C Johnson et al. *Yacc : Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[35] Jacques-Henri Jourdan, François Pottier, et Xavier Leroy. Validating LR(1) parsers. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 397–416. Springer, 2012. doi : 10.1007/978-3-642-28869-2_20. URL https://doi.org/10.1007/978-3-642-28869-2_20.

[36] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, et Derek Dreyer. Iris from the ground up : A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28 :e20, 2018. doi : 10.1017/S0956796818000151.

[37] Ohad Kammar, Sam Lindley, et Nicolas Oury. Handlers in action. In Greg Morrisett et Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 145–158. ACM, 2013. doi : 10.1145/2500365.2500590. URL https://doi.org/10.1145/2500365.2500590.

[38] Oleg Kiselyov et Hiromi Ishii. Freer monads, more extensible effects. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 94–105. ACM, 2015. ISBN 978-1-4503-3808-0. doi : 10.1145/2804302.2804319.

[39] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, et Derek Dreyer. Mosel : a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.*, 2(ICFP) :77 :1–77 :30, 2018. doi : 10.1145/3236772.

[40] Daan Leijen et Erik Meijer. Parsec : Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, July 2001. URL https://www.microsoft.com/en-us/research/publication/parsec-direct-style-monadic-parser-combinators-for-the-real-world/. User Modeling 2007, 11th International Conference, UM 2007, Corfu, Greece, June 25-29, 2007.

[41] Michael E Lesk et Eric Schmidt. *Lex : A lexical analyzer generator*. Bell Laboratories Murray Hill, NJ, 1975.

[42] Thomas Letan et Yann Régis-Gianas. Freespec : specifying, verifying, and executing impure computations in Coq. In Jasmin Blanchette et Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs*

and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020, pages 32–46. ACM, 2020. doi : 10.1145/3372885.3373812.

[43] Olivier Levillain. Parsifal : A pragmatic solution to the binary parsing problems. In *35. IEEE Security and Privacy Workshops, SPW 2014, San Jose, CA, USA, May 17-18, 2014*, pages 191–197. IEEE Computer Society, 2014. doi : 10.1109/SPW.2014.35. URL `https://doi.org/10.1109/SPW.2014.35`.

[44] Bohdan Liesnikov, Marcel Ullrich, et Yannick Forster. Generating induction principles and subterm relations for inductive types using metacoq. *CoRR*, abs/2006.15135, 2020. URL `https://arxiv.org/abs/2006.15135`.

[45] Chris M. Lonvick et Tatu Ylonen. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, January 2006. URL `https://www.rfc-editor.org/info/rfc4253`.

[46] Nicholas D. Matsakis et Felix S. Klock II. The rust language. In Michael Feldman et S. Tucker Taft, editors, *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology, HILT 2014, Portland, Oregon, USA, October 18-21, 2014*, pages 103–104. ACM, 2014. doi : 10.1145/2663171.2663188. URL `https://doi.org/10.1145/2663171.2663188`.

[47] Kazutaka Matsuda et Meng Wang. Flippr : A system for deriving parsers from pretty-printers. *New Gener. Comput.*, 36(3) :173–202, 2018. doi : 10.1007/s00354-018-0033-7. URL `https://doi.org/10.1007/s00354-018-0033-7`.

[48] Iris development team. coq-std++. URL `https://plv.mpi-sws.org/coqdoc/stdpp`.

[49] Conor McBride et Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1) :1–13, 2008. doi : 10.1017/S0956796807006326. URL `https://doi.org/10.1017/S0956796807006326`.

[50] Peter J. McCann et Satish Chandra. Packet types : Abstract specifications of network protocol messages. In Craig Partridge, editor, *Proceedings of the ACM SIGCOMM 2000 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 28 - September 1, 2000, Stockholm, Sweden*, pages 321–333. ACM, 2000. doi : 10.1145/347059.347563. URL `https://doi.org/10.1145/347059.347563`.

[51] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1) :55–92, 1991. doi : 10.1016/0890-5401(91)90052-4.

[52] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, et Edward Gan. Rocksalt : better, faster, stronger SFI for the x86. In Jan Vitek, Haibo Lin, et Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 395–404. ACM, 2012. doi : 10.1145/2254064.2254111. URL `https://doi.org/10.1145/2254064.2254111`.

[53] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, et Lars Birkedal. Ynot : dependent types for imperative programs. In James Hook et Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 229–240. ACM, 2008. doi : 10.1145/1411204.1411237.

[54] Aleksandar Nanevski, J. Gregory Morrisett, et Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6) :865–911, 2008. doi : 10.1017/S0956796808006953. URL `https://doi.org/10.1017/S0956796808006953`.

[55] David A. Naumann. Thirty-seven years of relational hoare logic : Remarks on its principles and history. In Tiziana Margaria et Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation : Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part II*, volume 12477 of *Lecture Notes in Computer Science*, pages 93–116. Springer, 2020. doi : 10.1007/978-3-030-61470-6_7. URL `https://doi.org/10.1007/978-3-030-61470-6_7`.

[56] Pierre Nigron et Pierre-Évariste Dagand. Reaching for the star : Tale of a monad in coq. In Liron Cohen et Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 29 :1–29 :19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi : 10.4230/LIPIcs.ITP.2021.29. URL `https://doi.org/10.4230/LIPIcs.ITP.2021.29`.

[57] Peter W. O'Hearn et David J. Pym. The logic of bunched implications. *Bull. Symb. Log.*, 5(2) :215–244, 1999. doi : 10.2307/421090. URL `https://doi.org/10.2307/421090`.

[58] Peter W. O'Hearn, John C. Reynolds, et Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001. doi : 10.1007/3-540-44802-0_1. URL `https://doi.org/10.1007/3-540-44802-0_1`.

[59] Nicolas Oury et Wouter Swierstra. The power of pi. In James Hook et Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 39–50. ACM, 2008. doi : 10.1145/1411204.1411213. URL `https://doi.org/10.1145/1411204.1411213`.

[60] Simon Peyton Jones. *Tackling the awkward squad : monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, pages 47–96. IOS Press, January 2001. ISBN ISBN 1 58603 1724.

[61] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, et Brent Yorgey. *Programming Language Foundations*. Software Foundations series, volume 2. Electronic textbook, May 2018.

[62] Gordon D. Plotkin et John Power. Adequacy for algebraic effects. In Furio Honsell et Marino Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001. doi : 10.1007/3-540-45315-6_1.

[63] David J. Pym. *The semantics and proof theory of the logic of bunched implications*, volume 26 of *Applied logic series*. Kluwer, 2002. ISBN 978-1-4020-0745-3.

[64] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, et Jonathan Protzenko. Everparse : Verified secure zero-copy parsers for authenticated message formats. In Nadia Heninger et Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1465–1482. USENIX Association, 2019. URL `https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud`.

[65] Tillmann Rendel et Klaus Ostermann. Invertible syntax descriptions : unifying parsing and pretty printing. In Jeremy Gibbons, editor, *Proceedings of the 3rd ACM SIGPLAN Symposium on Haskell, Haskell 2010, Baltimore, MD, USA, 30 September 2010*, pages 1–12. ACM, 2010. doi : 10.1145/1863523.1863525. URL `https://doi.org/10.1145/1863523.1863525`.

[66] John C. Reynolds. Separation logic : A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. ISBN 0-7695-1483-9. doi : 10.1109/LICS.2002.1029817.

[67] Carl Rigney, Steve Willens, Allan C. Rubens, et William Allen Simpson. Remote authentication dial in user service (RADIUS). *RFC*, 2865 :1–76, 2000. doi : 10.17487/RFC2865. URL `https://doi.org/10.17487/RFC2865`.

[68] Boubacar Demba Sall. *Programmation impérative par raffinements avec l'assistant de preuve Coq*. PhD thesis, 2020. URL `http://www.theses.fr/2020SORUS181`. Thèse de doctorat dirigée par Chailloux, Emmanuel et Peschanski, Frédéric Informatique Sorbonne université 2020.

[69] Christoph Sprenger et David A. Basin. A monad-based modeling and verification toolbox with application to security protocols. In Klaus Schneider et Jens Brandt, editors,

Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings, volume 4732 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2007. doi : 10.1007/978-3-540-74591-4_23. URL https://doi.org/10.1007/978-3-540-74591-4_23.

[70] Sorawit Suriyakarn, Clément Pit-Claudel, Benjamin Delaware, et Adam Chlipala. Narcissus : Deriving correct-by-construction decoders and encoders from binary formats. *CoRR*, abs/1803.04870, 2018. URL http://arxiv.org/abs/1803.04870.

[71] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, et Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In Hans-Juergen Boehm et Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 387–398. ACM, 2013. doi : 10.1145/2491956.2491978. URL https://doi.org/10.1145/2491956.2491978.

[72] 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, et Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In Rastislav Bodík et Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016. doi : 10.1145/2837614.2837655. URL https://doi.org/10.1145/2837614.2837655.

[73] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, et Guido Martínez. Steelcore : an extensible concurrent separation logic for effectful dependently typed programs. *Proc. ACM Program. Lang.*, 4(ICFP) :121 :1–121 :30, 2020. doi : 10.1145/3409003. URL https://doi.org/10.1145/3409003.

[74] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiridonova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, Omar Cardona, et Arti Gupta. Hardening attack surfaces with formally proven binary format parsers. In Ranjit Jhala et Isil Dillig, editors, *PLDI '22 : 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 31–45. ACM, 2022. doi : 10.1145/3519939.3523708. URL https://doi.org/10.1145/3519939.3523708.

[75] Wouter Swierstra. The hoare state monad (proof pearl), 2009.

[76] Wouter Swierstra et Tim Baanen. A predicate transformer semantics for effects (functional pearl). *Proc. ACM Program. Lang.*, 3(ICFP) :103 :1–103 :26, 2019. doi : 10.1145/3341707.

[77] Gang Tan et Greg Morrisett. Bidirectional grammars for machine-code decoding and encoding. In Sandrine Blazy et Marsha Chechik, editors, *Verified Software. Theories,*

Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers, volume 9971 of *Lecture Notes in Computer Science*, pages 73–89, 2016. doi : 10.1007/978-3-319-48869-1_6. URL https://doi.org/10.1007/978-3-319-48869-1_6.

[78] Akira Tanaka, Reynald Affeldt, et Jacques Garrigue. Safe low-level code generation in coq using monomorphization and monadification. *J. Inf. Process.*, 26 :54–72, 2018. doi : 10.2197/ipsjjip.26.54. URL https://doi.org/10.2197/ipsjjip.26.54.

[79] Enrico Tassi. Deriving proved equality tests in coq-elpi : Stronger induction principles for containers in coq. In John Harrison, John O'Leary, et Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 29 :1–29 :18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi : 10.4230/LIPIcs.ITP.2019.29. URL https://doi.org/10.4230/LIPIcs.ITP.2019.29.

[80] Tarmo Uustalu. Explicit binds : Effortless efficiency with and without trees. In Tom Schrijvers et Peter Thiemann, editors, *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings*, volume 7294 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2012. doi : 10.1007/978-3-642-29822-6_25. URL https://doi.org/10.1007/978-3-642-29822-6_25.

[81] Atze van der Ploeg et Oleg Kiselyov. Reflection without remorse : revealing a hidden sequence to speed up monadic reflection. In Wouter Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 133–144. ACM, 2014. doi : 10.1145/2633357.2633360. URL https://doi.org/10.1145/2633357.2633360.

[82] Marcell van Geest et Wouter Swierstra. Generic packet descriptions : verified parsing and pretty printing of low-level data. In Sam Lindley et Brent A. Yorgey, editors, *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2017, Oxford, UK, September 3, 2017*, pages 30–40. ACM, 2017. doi : 10.1145/3122975.3122979. URL https://doi.org/10.1145/3122975.3122979.

[83] Janis Voigtländer. Asymptotic improvement of computations over free monads. In Philippe Audebaud et Christine Paulin-Mohring, editors, *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings*, volume 5133 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 2008. doi : 10.1007/978-3-540-70594-9_20. URL https://doi.org/10.1007/978-3-540-70594-9_20.

[84] Geoffrey Washburn et Stephanie Weirich. Boxes go bananas : encoding higher-order abstract syntax with parametric polymorphism. In Colin Runciman et Olin Shivers,

editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 249–262. ACM, 2003. doi : 10.1145/944705.944728. URL https://doi.org/10.1145/944705.944728.

[85] Iain Whiteside, David Aspinall, et Gudmund Grov. An essence of ssreflect. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, et Volker Sorge, editors, *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, volume 7362 of *Lecture Notes in Computer Science*, pages 186–201. Springer, 2012. doi : 10.1007/978-3-642-31374-5_13. URL https://doi.org/10.1007/978-3-642-31374-5_13.

[86] Niklaus Wirth. Program development by stepwise refinement. *Commun. ACM*, 14 (4) :221–227, 1971. doi : 10.1145/362575.362577. URL https://doi.org/10.1145/362575.362577.

[87] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, et Steve Zdancewic. Interaction trees : representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.*, 4(POPL) :51 :1–51 :32, 2020. doi : 10.1145/3371119.

[88] Qianchuan Ye et Benjamin Delaware. A verified protocol buffer compiler. In Assia Mahboubi et Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 222–233. ACM, 2019. doi : 10.1145/3293880.3294105. URL https://doi.org/10.1145/3293880.3294105.