Logique de séparation et vérification déductive

François Bobot

Université Paris-Sud XI

École Doctorale d'Informatique de Paris-Sud

Laboratoire de Recherche en Informatique

THÈSE DE DOCTORAT

présentée par : **François Bobot**

soutenue le : **12 décembre 2011**

pour obtenir le grade de : **Docteur en Sciences de l'Université Paris-Sud XI**

Discipline : **Informatique**

Logique de séparation et vérification déductive

THÈSE dirigée par

M. Filliâtre Jean-Christophe	*Chargé de Recherche, CNRS, LRI*

RAPPORTEURS

M. Blazy Sandrine	*Professeur, Université Rennes 1 IRISA*
M. Ringeissen Christophe	*Chargé de recherche, INRIA-Lorraine LORIA*

EXAMINATEURS

M. Demri Stéphane	*Directeur de recherche, CNRS, ENS de Cachan LSV*
M. Hivert Florent	*Professeur, CNRS, LRI*

À ma nanie,

Je voudrais en premier lieu remercier chaleureusement mon directeur de thèse, Jean-Christophe Filliâtre, pour avoir toujours été présent lorsque j'en avais besoin. Sa gentillesse, son dynamisme, sa productivité seront toujours un exemple pour moi. Je serais toujours admiratif pour sa capacité à trouver des programmes très petits mais d'une très grande complexité à prouver. Je serais également toujours impressionné par sa rigueur et sa capacité à réaliser des merveilles en programmation ou ailleurs. Je le remercie enfin de m'avoir remuer la tête et les jambes tous les mercredi midi en jouant au foot avec de joyeux lurons.

Je remercie respectueusement Sandrine Blazy et Christophe Ringeissen d'avoir minutieusement relu mon manuscrit et pour les remarques qu'ils ont pu faire. Je remercie également Stéphane Demri et Florent Hivert pour avoir accepté de faire partie de mon jury.

Je remercie Andrei Paskevich pour sa bonne humeur et les discussions passionnantes que nous avons pu avoir ensemble sur de nombreux sujets.

Je souhaite remercier tous les membres actuels et passés de l'équipe Proval pour le plaisir que j'ai eu à travailler avec eux. Les discussions au coin café autour d'une gourmandise ont toujours été passionnantes. Merci sincèrement à tous. J'espère que l'équipe Proval gardera toujours cette chaleur humaine et cette joie dans la recherche. Je remercie Romain d'avoir partagé mon bureau durant ces trois années. Je remercie Sylvain, Mohamed, Alain et Évelyne des discussions que nous avons pu avoir autour de la preuve automatique. Je remercie également Johannes, Stéphane et Jean-Christophe pour m'avoir permit de faire mes figures dans mon langage favori.

Merci à tous mes amis, à toute ma famille pour m'avoir toujours soutenu bien avant d'avoir commencé cette thèse. Merci Francine [1] de m'avoir donné la passion de l'éducation, merci Pierre [2] de m'avoir donné la passion de l'invention. Merci à mes grand parents d'être des exemples parfait à suivre.

Et bien sûr autant de merci que de millimètre entre la terre et la lune, et retour, à Marianne [3] qui est mon roc, mon foyer et mon amour.

1. maman
2. papa
3. ma chérie

Table des matières

1. Introduction

L'omniprésence de l'informatique est désormais un lieu commun, et malheureusement avec elle la présence d'erreurs informatiques. Si votre autoradio refuse de s'éteindre et de changer de fréquence cela ne devrait aller qu'à l'encontre de vos goûts musicaux. Si en revanche votre régulateur de vitesse refuse de s'arrêter et de modifier la vitesse souhaitée vous pourriez aller au devant de découvertes plus désagréables. Heureusement on peut espérer que tourner la clé de contact arrêtera votre voiture. Pour un informaticien il est triste de devoir faire plus confiance à un système mécanique qu'à un système informatique. Les systèmes mécaniques sont sans doute actuellement mieux conçus, compris et testés que les systèmes informatiques. Cependant, l'évolution inévitable consiste à remplacer des systèmes mécaniques par des systèmes informatiques car ces derniers permettent de résoudre des problèmes plus complexes, comme par exemple la correction du faible équilibre latéral de l'Airbus A380 qui oblige à utiliser des commandes de vols numériques. Une erreur, dans ces conditions, est alors très critique. Il serait donc souhaitable que l'on possède des techniques performantes pour vérifier la correction des programmes informatiques.

Tester un programme dans différents environnements est la méthode traditionnelle de l'industrie et elle est encore utilisée. Cependant, on ne peut tester le nombre infini de vols possibles d'un avion. Ces méthodes de test permettent de connaître le comportement réel du programme dans un environnement précis ; à l'inverse, d'autres méthodes utilisent un modèle de l'exécution du programme pour déterminer une approximation du comportement du programme dans de nombreux environnements. Même si le résultat est une approximation, cette méthode détectera si le programme se comporte correctement. En revanche elle peut indiquer qu'un programme n'est pas correct ou elle peut ne pas se prononcer alors qu'il est correct. Mais ceci ne porte en aucun cas préjudice à la sécurité. Ce que l'on attend du programme est décrit par une spécification, par exemple sous forme d'une formule mathématique. Ces méthodes vérifient que leur modélisation du comportement du programme suit la spécification demandée.

On s'intéresse ici particulièrement à des méthodes de vérification déductive qui vont prouver que le programme suit sa spécification en utilisant une suite d'étapes de déduction. Une logique de programme définit ces étapes de déduction. Avec la logique de programme de Hoare [29], on peut produire, à partir du code source du programme et de la spécification, une formule mathématique dont la validité assure que le programme suit sa spécification. La validité de cette formule mathématique peut être prouvée à l'aide d'un démonstrateur automatique. Dans le cas où le démonstrateur ne l'a pas prouvé automatiquement, on doit la prouver "à la main" avec un assistant de preuve, c'est-à-dire un programme qui vérifie notre preuve. Des assistants de preuves ont pu être utilisés sur de gros projets [6] ; cependant c'est assez coûteux. On peut permettre une plus grande

automatisation en agissant à trois niveaux principaux : les méthodes utilisées pour modéliser le comportement des programmes, les outils utilisés pour décrire les spécifications, les techniques pour utiliser au mieux les démonstrateurs automatiques.

Dans le cas où un programme utilise des structures de données allouées en mémoire, c'est-à-dire des arrangements spécifiques de l'information de manière à pouvoir les traiter plus efficacement, on veut s'assurer qu'elles vérifient des invariants ou des propriétés de bonne formation. Cependant, elles sont souvent si complexes que l'on ne peut prouver simplement qu'elles sont conservées à chaque instant du programme. En montrant que deux parties d'une structure de données sont indépendantes, on peut s'assurer qu'en modifiant une partie on ne modifiera pas l'autre. Une logique particulière, appelée logique de séparation, est apparue pour exprimer aisément que différentes structures de données sont disjointes en mémoire [51]. Avec cette logique, les spécifications peuvent indiquer quelles structures de données sont séparées les unes des autres. La logique de séparation est également une logique de programme. Malheureusement les formules dans cette logique ne peuvent être prouvées directement par les démonstrateurs automatiques de la logique du premier ordre, là où le plus d'innovation est apporté.

Cette thèse reprend les idées de la logique de séparation dans un contexte de logique du premier ordre. Comme dans la logique de séparation on permet l'écriture de spécifications à l'aide de propriétés spatiales de séparation. Ces propriétés sont alors automatiquement traduites dans la logique du premier ordre traditionnelle. Un soin particulier est apporté dans cette traduction afin d'exploiter au mieux les capacités des démonstrateurs automatiques.

Nous allons maintenant voir plus précisément ce qu'est la vérification déductive, puis expliquer le fonctionnement de la logique de séparation pour identifier les idées que l'on va reprendre. Nous verrons également pourquoi le polymorphisme apparaît naturellement dans le cadre de notre travail.

1.1. Vérification déductive

La technique de vérification déductive qui nous intéresse ici a été introduite par Dijkstra et consiste à prendre un programme avec une spécification sous forme de pré- et postcondition et à calculer une formule qui exprime la correction, appelée la plus faible précondition du programme [26]. Si on suppose qu'on l'exécute en respectant une précondition, le mauvais fonctionnement d'un programme peut avoir trois causes, la précondition n'était donc pas assez restrictive :

- Le programme s'interrompt de manière inattendue ; par exemple sur un déréférencement de pointeur nul.
- Le programme fait une opération qui ne respecte pas la modélisation, par exemple une opération arithmétique qui produit un dépassement de capacité.
- Le programme s'arrête mais l'état final ne respecte pas la postcondition.

On peut remarquer qu'un programme qui ne s'arrête jamais mais qui n'est jamais dans l'un des deux premiers cas de figures n'est pas un mauvais programme. Prouver la terminaison est un problème en soi que l'on ne considère pas ici.

Voici un exemple de programme écrit en `C` qui calcule la moyenne de deux entiers positifs. On le spécifie dans le langage de spécification `ACSL` [3] :

```
/*@
requires 0 <= x <= y;
ensures \result == (x + y) / 2; @*/
int mean (int x, int y){
  return x + (y - x) / 2;
}
```

Le mot clé `requires` indique la précondition de la fonction `mean` et le mot clé `ensures` indique sa postcondition. Ce programme suit sa spécification et est sûr si pour des arguments x, y tels que $0 \leq x \leq y$, le programme s'exécute sans erreur (la sûreté du programme), et renvoie une valeur *result* qui vérifie $result = (x + y)/2$ (son comportement). Même un programme si simple peut ne pas être sûr en contenir. Par exemple on ne définit pas cette fonction en faisant la moyenne par $(x+y)/2$. Car si sur une machine 64 bits on exécute la fonction avec $x = y = 2^{62}$, l'addition $x + y = 2^{62} + 2^{62}$ ne donne pas 2^{63} mais -2. Ensuite la division par 2 renverra la valeur -1 qui n'est bien sûr pas la moyenne de 2^{62} et 2^{62} En effet les valeurs de type `int` représentent seulement les entiers mathématiques entre -2^{63} (= `int_min`) et $2^{63} - 1$ (= `int_max`) sur une machine 64 bits. Et donc $2^{62} + 2^{62} = 2^{63}$ n'est pas représentable. Ainsi même si la moyenne de 2^{62} avec lui-même est représentable, comme un dépassement de capacité est apparu pendant le calcul le résultat final est faux. Cette erreur est restée longtemps cachée dans l'algorithme de recherche dichotomique de la bibliothèque standard de `Java` jusqu'à ce que quelqu'un l'utilise avec des tableaux et des indices assez grands sur une machine 32 bits [7]. Maintenant nous allons prouver qu'il n'y a pas de dépassement de capacité avec cette version corrigée.

De manière générale on demande que les arguments e_1, e_2 de toute opération d'addition $e_1 + e_2$ vérifient $\texttt{int\_min} \leq e_1 + e_2 \leq \texttt{int\_max}$. Ici l'addition + n'est pas l'addition machine qui vérifierait toujours cette condition mais bien l'opération mathématique. Si la condition est vérifiée le résultat de l'opération machine est la valeur représentant le résultat mathématique. Dans les spécifications, on utilise les opérations mathématiques justement pour pouvoir spécifier les opérations machines et également parce que c'est l'opération d'addition que connaissent les démonstrateurs.

La plus faible précondition du programme, c'est-à-dire la précondition minimale qui permette à la postcondition et à la sûreté du programme d'être vérifiée, est ici :

$$
\begin{aligned}
& \texttt{int\_min} \leq y - x \leq \texttt{int\_max} \\
\wedge\ & 2 \neq 0 \\
\wedge\ & \texttt{int\_min} \leq (y - x)/2 \leq \texttt{int\_max} \\
\wedge\ & \texttt{int\_min} \leq x + (y - x)/2 \leq \texttt{int\_max} \\
\wedge\ & x + (y - x)/2 = (x + y)/2
\end{aligned}
$$

Il ne reste plus qu'à montrer que la précondition implique bien la plus faible précondition, ce qui est fait en prouvant la validité de la formule suivante :

$$\forall x : \mathbf{Z}, y : \mathbf{Z}$$
$$\begin{cases} 0 \leq x \leq y \\ \wedge \texttt{int\_min} \leq x \leq \texttt{int\_max} \\ \wedge \texttt{int\_min} \leq y \leq \texttt{int\_max} \\ \wedge \texttt{int\_max} = \quad 9223372036854775807 \\ \wedge \texttt{int\_min} = -9223372036854775808 \end{cases} \Rightarrow \begin{cases} \texttt{int\_min} \leq y - x \leq \texttt{int\_max} \\ \wedge 2 \neq 0 \\ \wedge \texttt{int\_min} \leq (y-x)/2 \leq \texttt{int\_max} \\ \wedge \texttt{int\_min} \leq x + (y-x)/2 \leq \texttt{int\_max} \\ \wedge\, x + (y-x)/2 = (x+y)/2 \end{cases}$$

On peut le faire à la main (cela demande peu de connaissances en arithmétique) ou à l'aide d'un démonstrateur. Par exemple `Alt-ergo` [8] la prouve instantanément.

Dans ce premier exemple aucune donnée n'a été allouée en mémoire. On n'a donc pas eu besoin de la modéliser. L'exemple suivant va illustrer la nécessité de modéliser la mémoire et les pointeurs :

```
void double (int * x, int * y){
  *x = 2 * (*x);
  *y = 2 * (*y);
}
```

Il semble que la valeur pointée par x et la valeur pointée par y vont doubler après un appel à cette fonction. Une postcondition serait alors :

```
ensures *x == 2 * \old(*x)
```

Ici `\old(*x)` représente la valeur de `*x` avant l'exécution de la fonction. Cependant, s'il se trouve que x et y sont égaux, `*x` (et donc `*y`) sera quadruplé. Lorsque deux pointeurs sont égaux on dit aussi qu'ils sont alias l'un de l'autre. La postcondition que l'on va prouver sera donc plutôt `ensures x != y ==> *x == 2 * \old(*x)`. Pour modéliser la mémoire on utilise des tableaux purement applicatifs, c'est à dire que les opérations ne modifie pas le tableau mais donne un nouveau tableau où la modification a été appliqué. Les tableaux purement applicatifs possèdent deux opérations : $\texttt{get}(m, x)$ renvoie le contenu de la case x dans le tableau m et $\texttt{set}(m, x, v)$ renvoie un nouveau tableau identique en chaque case à m, excepté en x où il vaut v. On peut exprimer cette propriété par deux formules du premier ordre :

$$\forall m.\forall x.\forall v.\, \texttt{get}(\texttt{set}(m, x, v), x) = v \tag{1.1}$$
$$\forall m.\forall x, y.\forall v.\, y \neq x \Rightarrow \texttt{get}(\texttt{set}(m, y, v), x) = \texttt{get}(m, x) \tag{1.2}$$

En chaque point du programme la mémoire sera représentée par un tableau. Ainsi le code de la fonction devient en explicitant la mémoire ainsi modélisée (tas^0 représente la mémoire avant l'exécution de la fonction) :

$$tas^1 = \texttt{set}(tas^0, x, 2 * \texttt{get}(tas^0, x));$$
$$tas^2 = \texttt{set}(tas^1, y, 2 * \texttt{get}(tas^1, y));$$

et la postcondition devient :

$$x \neq y \Rightarrow \texttt{get}(tas^2, x) = 2 * \texttt{get}(tas^0, x)$$

On note que la mémoire est modélisée à l'aide d'un tableau infini dont toutes les cases sont accessibles. Pourtant un programme possède une vision plus parcellaire de la mémoire : il ne peut accéder qu'aux zones mémoires qui lui ont été allouées. Si un programme essaie d'accéder à une case de la mémoire qui ne lui a pas été allouée, une erreur de segmentation est levée et le programme s'arrête. Pour s'en prémunir, on vérifie que chaque accès `*z` et chaque modification `*z = ...` de la mémoire est réalisés avec un pointeur valide. Les propriétés de sûreté de ce programme consistent donc à ce que les pointeurs `x` et `y` soient valides. On étend notre modèle en introduisant un prédicat $\texttt{valid}(tas, z)$ qui indique que dans la mémoire représentée par le tableau tas, le pointeur z est valide. Dès lors la sûreté du programme est exprimée par $\texttt{valid}(tas^0, x) \wedge \texttt{valid}(tas^1, y)$. On ajoute dans notre modèle la propriété qu'une affectation ne modifie pas la validité des pointeurs :

$$\forall m. \forall x, y. \forall v. \texttt{valid}(\texttt{set}(m, x, v), y) \Leftrightarrow \texttt{valid}(m, y) \tag{1.3}$$

On va demander en précondition `requires valid(x) /\ valid(y)` ce qui revient dans notre modèle mémoire à la formule $valid(tas^0, x) \wedge valid(tas^1, y)$.

On obtient ainsi la plus faible précondition de la sûreté et du comportement de la fonction. Dans la formule, $\texttt{let}\ tas^1 = e\ \texttt{in}$ définie tas^1 comme un raccourci pour l'expression mathématique e.

$$\begin{aligned}
&\texttt{valid}(tas^0, x) \wedge \\
&\texttt{let}\ tas^1 = \texttt{set}(tas^0, x, 2 * \texttt{get}(tas^0, x))\ \texttt{in} \\
&\texttt{valid}(tas^1, y) \wedge \\
&\texttt{let}\ tas^2 = \texttt{set}(tas^1, y, 2 * \texttt{get}(tas^1, y))\ \texttt{in} \\
&x \neq y \Rightarrow \texttt{get}(tas^2, x) = 2 * \texttt{get}(tas^0, x)
\end{aligned}$$

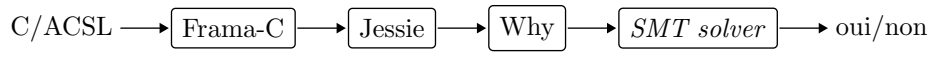


FIGURE 1.1.: Schéma de l'enchaînement des différents outils.

Maintenant il faut prouver que la précondition de la fonction implique bien la plus faible précondition :

$$\forall tas^0.\forall x, y.$$

$$\texttt{valid}(tas^0, x) \wedge \texttt{valid}(tas^0, y) \Rightarrow \begin{cases} \texttt{valid}(tas^0, x) \wedge \\ \texttt{let}\, tas^1 = \texttt{set}(tas^0, x, 2 * \texttt{get}(tas^0, x))\, \texttt{in} \\ \texttt{valid}(tas^1, y) \wedge \\ \texttt{let}\, tas^2 = \texttt{set}(tas^1, y, 2 * \texttt{get}(tas^1, y))\, \texttt{in} \\ x \neq y \Rightarrow \texttt{get}(tas^2, x) = 2 * \texttt{get}(tas^0, x) \end{cases}$$

Pour prouver que quand x et y sont différents $\texttt{get}(tas^2, x) = 2 * \texttt{get}(tas^0, x)$ on va utiliser les propriétés de la théorie des tableaux (1.1,1.2). Le tas^2 est la modification du tas^1 en y et comme on sait que $x \neq y$ la propriété (1.2) donne que $\texttt{get}(tas^2, x) = \texttt{get}(tas^1, x)$. Ensuite le tas^1 est la modification du tas tas^0 en x par la valeur $2 * \texttt{get}(tas^0, x)$ donc la propriété (1.1) donne que $\texttt{get}(tas^1, x) = 2 * \texttt{get}(tas^0, x)$. Ce qui finit de prouver l'égalité. Finalement $\texttt{valid}(tas^0, x)$ se trouve dans les hypothèses de la formule et $\texttt{valid}(tas^1, y)$ se prouve avec la propriété (1.3).

On peut remarquer que la postcondition ne donne aucune information si x est égal à y. Dans ce cas, on peut néanmoins utiliser cette fonction mais on n'aura aucune information sur l'état de la mémoire après son exécution. On aurait pu également mettre la condition $x \neq y$ dans la précondition de la fonction. Dans ce cas, on n'aurait pas pu appeler la fonction quand x et y coïncident.

Les démonstrateurs sont capables de prouver instantanément cette formule soit en utilisant les propriétés (1.1,1.2,1.3) ajoutées à la formule à prouver, soit en utilisant une procédure de décision pour la théorie des tableaux [38]. Les procédures de décision sont des algorithmes pour prouver des propriétés de théories particulières de manière complète et plus efficace qu'en utilisant des formules du premier ordre qui les définissent.

Les transformations à effectuer sur le programme, le calcul de la plus faible précondition et l'envoi de la formule obtenue à un démonstrateur automatique peuvent être totalement automatisés. Par exemple, la plate-forme `Why` [28] permet de faire automatiquement une telle suite d'étapes. La plate-forme `Why` est notamment utilisée en conjonction avec la plate-forme `Frama-C` pour la vérification déductive de programme `C`. `Frama-C` lit et type des fichiers `C` avec des annotations `ACSL` puis appel le greffon `Jessie` qui transforme les différentes constructions `C` supportées dans le langage de `Jessie`. Ensuite c'est l'outil `Jessie` [43] qui transforme un programme `Jessie` utilisant des pointeurs en un programme `Why` où la mémoire est devenue explicite. L'outil `Why` peut alors effectuer le calcul de la plus faible précondition et appeler les prouveurs. La figure 1.1 résume l'enchaînement des différents outils.

Dans l'exemple précédent, le modèle mémoire consistait à ce que l'état mémoire soit représenté par un unique tableau. Dans ce modèle deux types d'informations étaient attachés pour chaque adresse mémoire : sa valeur et sa validité. Or ces deux notions étant indépendantes, on peut associer à chaque état mémoire, un tableau associant à chaque adresse une valeur et un tableau associant à chaque adresse sa validité. Ainsi on n'a plus besoin de la propriété (1.3) puisque les affectations de valeurs ne modifient pas le tableau relatif à la validité. La sûreté du programme devient alors triviale à prouver :

$$\texttt{valid}(tas^0, x) \wedge \texttt{valid}(tas^0, y) \Rightarrow \texttt{valid}(tas^0, x) \wedge \texttt{valid}(tas^0, y)$$

Cette technique, que nous appellerons séparation par champs, est également utile lorsque le programme utilise des enregistrements. Si par exemple le programme fait usage d'un type `C` tel que :

```
struct data {
 int field1;
 int field2;
 int field3;
} data;
```

alors on peut modéliser les valeurs en mémoire non pas à l'aide d'un unique tableau, mais à l'aide de trois tableaux $field_1$, $field_2$, $field_3$. Dans ce nouveau modèle, vérifier la postcondition du programme

```
/*@
  requires valid(d);
  ensures (*d).field2 = 42
@*/
data init(data * d){
(*d).field2 = 42;
(*d).field3 = 85;
}
```

revient à montrer la formule :

$$\begin{aligned}\forall field_2, field_3.\texttt{let}\ field_2^2 = set(field_2, d, 42)\ in\\ \texttt{let}\ field_3^2 = set(field_3, d, 85)\ in\\ get(field_2^2, d) = 42\end{aligned}$$

Pour prouver cette formule on n'a plus besoin d'utiliser 1.2, ce qui simplifie la recherche de preuve et donc augmente l'automatisation.

Cette technique de séparation a été décrite initialement par Burstall [13] puis reprise plus récemment dans de nombreux systèmes [12, 28]. Des expériences ont montré que cette séparation statique des champs permettait une plus grande automatisation avec des démonstrateurs automatiques que d'autres modèles mémoire ne séparant pas les champs [14]. Cette technique de séparation est à la base du modèle mémoire de `Jessie`.

Ce modèle a même été raffiné [30] pour ajouter une séparation à l'intérieur même d'un champ. En effet si l'on sait que deux ensembles de pointeurs de structures ne seront jamais alias on peut alors utiliser un tableau mémoire différent pour chacun des ensembles. Ainsi `Jessie` utilise de nombreux tableaux, certains contenant des entiers, d'autres des modélisations de nombres flottants, d'autres des pointeurs de structure. Tous ces tableaux partagent la même axiomatisation, c'est-à-dire au moins les axiomes (1.1,1.2). Il est donc naturel, comme on le fait en programmation depuis longtemps [45, 23], de factoriser tous ces symboles qui ne diffèrent que par leur type en un seul symbole polymorphe. Le polymorphisme permet que l'on définisse "une boîte de quelque chose". Ses propriétés pourraient être "on peut mettre plusieurs choses dans la boîte mais toutes les choses que l'on y mettra devront être de la même sorte". Si on retire quelque chose de la boîte, il est bien de la même sorte que ce qu'on avait mis dedans. On a donc, pour tous les "quelque chose" possibles, définit ce qu'est une "boîte à quelque chose", par exemple une "boîte de bonbons", une "boîte d'entiers", une "boîte de pointeurs de structure d'élément de listes", ...

Si on prend maintenant l'exemple des tableaux, la théorie des tableaux peut s'axiomatiser comme suit :

$$\texttt{type}\ m\,\alpha\,\beta$$
$$\texttt{get}(m\,\alpha\,\beta)\,\alpha : \beta$$
$$\texttt{set}(m\,\alpha\,\beta)\,\alpha\,\beta : m\,\alpha\,\beta$$

$$axiom\ \texttt{eq} \ : \forall t : m\,\alpha\,\beta.\forall x : \alpha.\forall v : \beta.\ \texttt{get}(\texttt{set}(m,x,v),x) = x \tag{1.4}$$
$$axiom\ \texttt{diseq} : \forall t : m\,\alpha\,\beta.\forall x,y : \alpha.\forall v : \beta.\ x \neq y \Rightarrow \texttt{get}(\texttt{set}(m,x,v),y) = \texttt{get}(m,y) \tag{1.5}$$

`Jessie` utilise ainsi cette seule axiomatique des tableaux polymorphes $(m\,\alpha\,\beta)$ pour décrire les propriétés de tous les tableaux utilisés dans son modèle mémoire quelque soit la nature des clés (α) ou des valeurs (β). Malheureusement, à l'exception d'`Alt-Ergo`, les démonstrateurs automatiques ne connaissent pas le polymorphisme. Il faut donc l'éliminer pour pouvoir les utiliser.

1.2. La logique de séparation

La séparation par champs atteint ses limites lorsque l'on travaille avec des "structures chaînées" récursives :

```
struct node {
  ...
  node * ...
  ...
}
```

Dans ce cas-là, deux problèmes surviennent. Le premier est que toutes ces sous-structures possèdent les mêmes champs et donc la séparation par champs ne peut intervenir. Le

deuxième problème est que cette structure possède souvent un nombre non déterminé d'éléments ; il faut donc souvent faire un raisonnement par récurrence, ce que les démonstrateurs automatiques ne savent généralement pas faire. La logique de séparation [51], s'est trouvée être très bien adaptée pour traiter ces problèmes, grâce à une notion prédéfinie de séparation qui permet d'éviter certains raisonnement par récurrence. Cela permet à la fois d'être plus précis qu'avec une analyse statique par champs, et de traiter certains raisonnements par récurrence de manière implicite.

Dans la logique de séparation on décrit explicitement qu'un pointeur est alloué et vers quelle valeur il pointe. On utilise la notation $x \mapsto w$ pour signifier que x est alloué et pointe vers w. La logique de séparation introduit un connecteur logique $\star$ pour exprimer la séparation spatiale de la mémoire. Ainsi une formule $P \star Q$ décrit intuitivement que les pointeurs qui apparaissent dans P à gauche de $\mapsto$ sont disjoints de ceux qui apparaissent dans Q à gauche de $\mapsto$. La règle de mutation pour la logique de séparation est alors :

$$\{x \mapsto \_ \star P\}x := v\{x \mapsto v \star P\}$$

Ainsi on a $\{x \mapsto \_ \star y \mapsto 3\}x := 4\{x \mapsto 4 \star y \mapsto 3\}$. L'étoile $\star$ oblige x et y à ne pas être alias. Ainsi c'est la structure de la formule $x \mapsto \_ \star y \mapsto 3$ qui exprime l'absence d'alias entre x et y.

Étudions comment on aurait pu traiter le cas de la fonction `double` avec la logique de séparation. On va supposer que l'on a choisi de mettre la condition de non-alias dans la précondition. La précondition en logique de séparation s'écrit alors $x \mapsto v_x \star y \mapsto v_y$, en supposant que v_x et v_y sont quantifiées universellement au niveau de toute la fonction (comme on l'a vu, $x \mapsto v_x$ signifie que x est un pointeur valide et qu'il pointe vers la valeur v_x). On veut montrer que

$$\{x \mapsto v_x \star y \mapsto v_y\}\texttt{double(x,y)}\{x \mapsto 2 \times v_x\}$$

On va pour cela appliquer les règles de preuve de programme de la logique de séparation sur le code de `double(x,y)` :

$$\{x \mapsto v_x \star y \mapsto v_y\}$$
```
double(x,y)
```
$$\{x \mapsto v_x \star y \mapsto v_y\}$$
```
*x = 2 * (*x);*y = 2 * (*y);
```
(application de la règle de mutation sur $x \mapsto v_x$)
$$\{x \mapsto 2 * v_x \star y \mapsto v_y\}$$
```
*y = 2 * (*y);
```

(application de la règle de mutation sur $y \mapsto v_y$)
$$\{x \mapsto 2 * v_x \star y \mapsto 2 * v_y\}$$

(affaiblissement)
$$\{x \mapsto 2 * v_x\}$$

Les règles sont très simples à appliquer, principalement parce que le connecteur de séparation implique structurellement l'absence d'alias.

La formule $x \mapsto 2 * v_x \star y \mapsto 2 * v_y$ implique $x \mapsto 2 * v_x$ dans la logique de séparation intuitionniste, car elle permet d'oublier une partie de la mémoire. C'est la première forme de logique de séparation à apparaître [34, 52]. Ensuite une version classique [34, 49] a été décrite qui permet de rendre compte des fuites mémoires : une case mémoire ne peut disparaître que si elle est explicitement libérée.

L'intérêt de la logique de séparation apparaît encore plus clairement lorsque l'on utilise des structures de données récursives comme des listes. Nous introduisons une constante `null` pour indiquer le pointeur de fin de la liste. En logique de séparation classique on introduit la constante de prédicat `emp` pour indiquer que cette formule ne peut être vérifiée qu'avec une mémoire vide. La déclaration suivante définit $list(p)$ comme le fait qu'une liste bien formée commence au pointeur p :

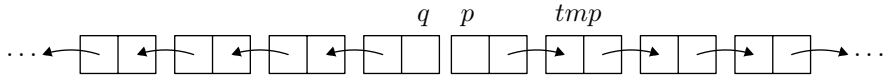
$$list(p) := \texttt{if} p = \texttt{null}\ \texttt{then emp else}\ \exists v.\, p.next \mapsto v \star list(v)$$

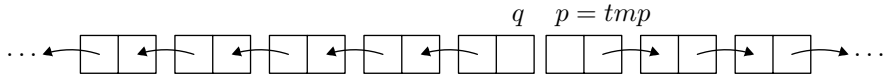
Cette définition indique notamment grâce la conjonction séparante $\star$, que p n'apparaît pas dans le reste de la liste. On pourrait le déduire du fait que la liste se termine par `null`, mais on préfère le faire apparaître explicitement de manière à pouvoir utiliser directement cette propriété de séparation.

Le code suivant fait partie de la fonction d'inversion d'une liste en place. Il retire l'élément en tête de la liste partant de `p` pour le mettre dans la liste partant de `q`. Si `p` est vide on ne fait rien.

```
{A}
  if (p != null){
    tmp = p->next;
    p->next = q;
    q = p;
    p = tmp;
  }
{A}
```

On cherche ici à prouver une propriété A qui est conservée par ce code (invariant). Pour la prouver on a besoin de s'assurer que les deux ensembles de pointeurs partant de q et p sont disjoints comme illustré ci-dessous.





On l'exprime très simplement en logique de séparation par la formule suivante :

$$A = list(p) \star list(q)$$

Si `p == null` le résultat est directement prouvé. Dans le cas où `p != null` on le prouve avec les mêmes règles que précédemment, excepté que l'on peut aussi utiliser la définition du prédicat *list*. La formule $(p \wedge q) \star r$ est équivalente à $p \wedge (q \star r)$ lorsque p est pure, c'est-à-dire ne contient ni `emp`, ni $\_ \mapsto \_$; on peut donc séparer la partie pure de la partie impure :

$\{list(p) \star list(q) \wedge p \neq \texttt{null}\}$
`tmp = p->next; p->next = q;q = p; p = tmp`

($list(p)$ est remplacé par sa définition, et $p \neq$ `null` la simplifie en $p.next \mapsto n_p \star list(n_p)$) (1.6)
$\{p- > next \mapsto n_p \star list(n_p) \star list(q) \wedge p \neq \texttt{null}\}$
`tmp = p->next; p->next = q;q = p; p = tmp`

(On ajoute $tmp = n_p$ aux propriétés non spatiales car tmp n'est pas alloué sur le tas)
$\{p.next \mapsto n_p \star list(n_p) \star list(q) \wedge p \neq \texttt{null} \wedge tmp = n_p\}$
`p->next = q;q = p; p = tmp`

(On applique la règle de mutation sur $p.next \mapsto n_p$)
$\{p.next \mapsto q \star list(n_p) \star list(q) \wedge p \neq \texttt{null} \wedge tmp = n_p\}$
`q = p; p = tmp`

(On remplace q par son ancienne valeur, notée v_q)
$\{p.next \mapsto v_q \star list(n_p) \star list(v_q) \wedge p \neq \texttt{null} \wedge tmp = n_p \wedge q = p\}$
`p = tmp`

(On remplace p par son ancienne valeur, ici q)
$\{q.next \mapsto v_q \star list(n_p) \star list(v_q) \wedge q \neq \texttt{null} \wedge tmp = n_p \wedge q = q \wedge p = tmp\}$

(On referme $q.next \mapsto v_q \star list(v_p)$ en $list(q)$ à l'aide de $q \neq$ `null`) (1.7)
$\{list(q) \star list(n_p) \wedge q \neq \texttt{null} \wedge tmp = n_p \wedge p = tmp\}$

(On termine par un affaiblissement)
$\{list(q) \star list(p)\}$

Des outils ont été développés pour effectuer les étapes précédentes automatiquement. Smallfoot [5] est un tel outil. Il est basé sur des règles de réécriture prédéfinies. Des

règles sont définies par exemple pour le prédicat de liste, que nous avons précédemment déclaré, qui permettent notamment d'ouvrir ou de fermer sa définition. Smallfoot utilise également des techniques d'interprétation abstraite pour découvrir automatiquement des invariants de boucle. Du point de vue de l'interprétation abstraite [21], on peut dire que le domaine utilisé par Smallfoot est constitué par des formules de logique de séparation n'utilisant que la conjonction séparante ($\star$) et la conjonction habituelle ($\wedge$). Ces formules logiques sont appelées *tas symboliques* [4, 27]. On ne peut donc prouver des propriétés qui demanderaient d'autres connecteurs logiques pour être exprimées. De plus on ne peut définir ses propres prédicats récursifs, ce qui empêche de prouver des programmes utilisant leurs propres structures de données.

Un autre outil, VeriFast [35, 36], sacrifie un peu d'automatisation pour permettre de définir ses propres prédicats récursifs. Il utilise lui aussi des formules sous forme de tas symboliques. Les opérations d'ouverture et de fermeture des prédicats que l'on a utilisées dans la preuve précédente (1.6,1.7) deviennent explicites dans VeriFast. La règle de mutation est appliquée à l'aide de critères syntaxiques ; c'est donc à la charge de l'utilisateur de faire apparaître le nécessaire $x.next \mapsto v$ à l'aide des ouvertures et fermetures. Pour les étapes de raisonnement concernant des prédicats purs, VeriFast utilise le démonstrateur SMT Z3 [24], comme un solveur de contraintes. Ainsi le code que l'on veut prouver est décrit en VeriFast par :

```
if (p != null){
  /*@ open list(p) @*/
  tmp = p->next;
  p->next = q;
  q = p;
  p = tmp;
  /*@ close list(q) @*/
}
```

`open list(p)` indique qu'il faut remplacer le prédicat `list(p)` par sa définition dans le tas symbolique. `close list(q)` indique qu'il faut remplacer la définition de `list(q)` par `list(q)` dans le tas symbolique.

Smallfoot et VeriFast représentent le tas à l'aide d'un tas symbolique. D'autres techniques ont été utilisées pour le représenter. La *shape analysis* [37, 17] modélise le tas par un graphe, dont les nœuds dénotent des objets et dont les arcs dénotent les valeurs des champs des objets. Certaines versions de la shape analysis autorisent à couper en deux un prédicat récursif. Ainsi une liste allant d'un pointeur au pointeur `null` peut être remplacée par une liste allant d'un pointeur à un autre puis de ce dernier au pointeur `null` [15]. Cela permet de traiter des programmes qui parcourent une liste bien plus facilement.

La logique de séparation définie également la règle d'encadrement qui ressemble à la règle de mutation et qui permet de plonger une fonction dans un contexte plus grand que celui utilisé lors de sa définition et qui ne sera pas modifié :

$$\frac{\{P\}f(\vec{x})\{Q\}}{\{P \star R\}f(\vec{x})\{Q \star R\}}$$

La logique de séparation et cette règle en particulier a excité l'imagination pour donner également naissance aux *dynamic frames* [39]. Le principe est d'ajouter des variables fantômes, représentant des ensembles de pointeurs qui ne participent pas au calcul mais seulement à la spécification. Ensuite on peut, grâce à certains opérateurs spécifier qu'une formule ne dépend que d'un certain ensemble de pointeurs, ou que l'ensemble des pointeurs n'a pas été modifié entre deux états mémoires. On peut, définir ces ensembles de pointeurs de différentes manières. Dans Dafny on modifie explicitement ces ensembles [40]. La méthode par *Implicit Dynamic Frame* utilise implicitement ces notions en calculant ces ensembles à partir des préconditions [54]. Cependant en faisant ainsi ces derniers restreignent la forme que peuvent avoir les pré- et postconditions. Nous verrons au cours de ce manuscrit des comparaisons avec ce travail.

Pour revenir à l'intérêt de la logique de séparation mentionnée au début de ce chapitre, remarquons que l'étoile séparante $\star$ a permis de décrire simplement la séparation entre différents ensembles de pointeurs dans la preuve de l'extrait de l'inversion de liste. Mais, plus important, elle a évité de faire un raisonnement par récurrence. En effet ce type de raisonnement serait indispensable sans logique de séparation pour montrer que lorsque l'on modifie p on ne modifie pas $list(q)$ pour la raison que p ne se trouve pas dans les pointeurs de q. Ici le raisonnement se trouve dans la preuve de correction de la règle de mutation.

1.3. Contributions de cette thèse

La logique de séparation montre que les propriétés de séparation peuvent être à la fois très utiles mais également très efficaces. Malheureusement les outils qui automatisent la preuve de programmes en logique de séparation n'utilisent qu'un sous-ensemble de la logique de séparation et doivent se passer de nombreux connecteurs logiques. Cette thèse reprend les idées de la logique de séparation, c'est-à-dire la possibilité pour l'utilisateur de définir, à partir de la définition d'une structure de données comme `list`, des prédicats exprimant la séparation spatiale de cette structure avec d'autres structures, du type `list` ou de tout autre type. Ainsi une spécification peut introduire un tel prédicat `sep` pour écrire $sep(p, q) \wedge list(p) \wedge list(q)$ là où la logique de séparation écrirait $list(p) \star list(q)$. Ces propriétés sont définies de sorte que l'axiome utilisé en cas de modification du tas soit aussi simple à appliquer que la règle de mutation. Ainsi, des raisonnements génériques continuent d'être utilisés. Le choix d'ouvrir et fermer un prédicat devient le choix d'utiliser ou non un axiome. Dans notre cas, ce choix est réalisé par les démonstrateurs automatiques du premier ordre. De manière à utiliser au mieux leurs capacités, et donc à augmenter l'automatisation, cette thèse s'attache également à améliorer l'utilisation des théories prédéfinies des démonstrateurs. Les démonstrateurs automatiques sont né en prenant pour base des logiques simples. D'un autre coté les utilisateurs préfère utiliser des logiques riches. Dans notre cas nous utilisons la logique polymorphe ainsi que des théories spécifiques comme la théorie de l'arithmétique linéaire et la théorie des tableaux. Il existait précédemment des techniques pour transformer des problèmes de la logique polymorphe avec arithmétique linéaire. Cette thèse les a étendu pour traiter des types

complexes et des types finis. On a également développé une technique permettant de distinguer certaines applications monomorphes de symboles polymorphes pour éviter de les encoder, ce qui s'applique très bien à la théorie des tableaux. Expérimentalement cette technique s'est montrée très efficace, particulièrement sur les conditions de vérification générées par les prédicats de séparations.

Première partie

Prédicats de séparation

2. Intuition

Nous allons voir à l'aide de l'exemple simple de l'inversion de liste en place comment on peut traiter la problématique de séparation en restant en logique du premier ordre.

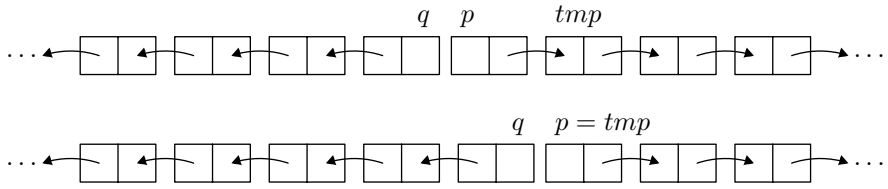
Les listes sont ici simplement chaînées par un champ `next`. Le modèle mémoire consiste ici en un tableau de pointeurs vers pointeur représentant le champ `next`. Par simplicité on considère que tous les pointeurs sont toujours valides. On écrit le programme en `Why` de manière à montrer ce que l'on va ensuite générer automatiquement depuis le programme `Jessie`. L'inversion de liste s'écrit en `Why` de la manière suivante, ici le type `t` est le type des tableaux polymorphes et provient de la théorie `array.Array` :

```
use import int.Int
use import module stdlib.Ref
use import array.Array

type pointer
type next = t pointer pointer
function null : pointer

parameter value : ref (t pointer int)
parameter next : ref next

let list_rev (p : ref pointer) =
  let q = ref null in
  while !p ≠ null do
    let tmp = get !next !p in
    next := set !next !p !q;
    q := !p;
    p := tmp
  done;
  q
```

La déclaration **`use import`** permet d'importer une théorie dans l'espace de nom courant. La déclaration **`type`** déclare ou défini un constructeur de type ou déclare un alias de type. La déclaration **`function`** défini une fonction logique, **`predicate`** défini un prédicat logique. La déclaration **`parameter`** défini ici des variables globales. La déclaration **`let`** défini une fonction de programme. Les fonctions de programmes `ref`, `:=`, `!` permettent comme en `OCaml` de créer, modifier ou lire une référence.

La fonction contient une boucle qui prend le premier élément de la liste commençant par `p` pour l'ajouter en tête de la liste commençant par `q` jusqu'à ce que `!p`=`null`.

`next := set !next !p !q;` correspond à `p->next = q`. L'action de cette boucle est schématisée ci-dessous :



Dans un premier temps nous allons simplement prouver que si la fonction `list_rev` prend en argument un pointeur vers une liste finie alors elle renvoie une liste finie. Les listes finies peuvent se formaliser simplement à l'aide du prédicat inductif suivant :

```
inductive islist (next : next) (p : pointer) =
  | islist_null:
      ∀ next : next, p : pointer.
      p = null ⇒ islist next p
  | islist_next:
      ∀ next : next, p : pointer.
      p ≠ null ⇒ islist next (get next p) ⇒ islist next p
```

Comme l'inductif est le plus petit point fixe, dans un état mémoire `next`, si `p` vérifie `islist next p` on sait que `null` est accessible depuis `p` par l'intermédiaire du champ `next`.

Pour prouver que `list_rev` conserve la bonne formation de la liste, la précondition est `islist next p` et la postcondition est `islist next result`, en posant `result` comme le résultat de la fonction.

Pour prouver que la fonction vérifie son contrat on doit ajouter un invariant de boucle. L'invariant de boucle indique ce qui est vrai à chaque itération de la boucle. Lors de la sortie de la boucle on peut ainsi l'utiliser pour prouver la suite du programme. Pour prouver qu'une formule est un invariant de boucle il faut prouver qu'au début de la première itération elle est bien vérifiée (c'est l'initialisation) et prouver que si elle est vraie avant une itération alors elle est vraie après (c'est la préservation). Le plus naturel est de spécifier que `p` et `q` pointent vers des listes finies.

```
...
while !p ≠ null do
  invariant { islist next p ∧ islist next q}
  ...
do
...
```

L'initialisation de l'invariant est facilement prouvable. En effet `islist next null` est vrai quelle que soit `next`. Il reste à prouver la préservation de l'invariant, c'est-à-dire

que si `p` et `q` pointent vers des listes finies avant le corps de la boucle, c'est toujours le cas après une exécution du corps de la boucle.

En utilisant l'invariant de boucle on obtient alors facilement la postcondition : on a `islist next q` qui est exactement `islist next result`.

Le corps de la boucle modifie le tableau représentant le champ `next` en `p`. En C cela représenterait l'action de modifier le champ `next` de la structure pointée par `p`. On notera l'état mémoire après l'affectation `next'` ; on a donc `next' = (set next p v)`. En sachant seulement que les deux listes sont bien formées avant cette affectation, on ne peut pas prouver que modifier `p` ne va pas modifier la liste pointée par `q`. En l'occurrence faire pointer `p` vers `q` pourrait créer un cycle et donc `islist (set next p q) q` ne serait pas vérifié. Cependant cela n'est pas le cas ici car à chaque itération `p` ne fait pas partie des pointeurs dont dépend `islist next q`.

Cette notion de pointeur dont dépend un prédicat est importante et sera nommée *empreinte* du prédicat. Cette empreinte dépend de l'état mémoire actuellement considéré ainsi que des arguments passés au prédicat. L'empreinte du prédicat nous servira de deux manières. Premièrement, comme l'on vient de voir, on utilisera l'empreinte pour savoir quand un prédicat n'est pas modifié par une affectation. Deuxièmement, on pourra déduire d'une empreinte des propriétés sur une autre empreinte. En effet si l'on sait qu'une première empreinte est disjointe d'une seconde, en sachant qu'un pointeur appartient à l'une, on en déduit qu'il n'appartient pas à l'autre. Et comme dans le premier point, la modification de ce pointeur ne modifie pas la seconde empreinte.

Dans le cas des listes on notera l'empreinte par `islist_ft next p`. Les empreintes sont ici d'un type abstrait `ft` et on se donne un prédicat d'appartenance `in_ft`.

```
type ft

predicate in_ft (p:pointer) (ft:ft)
```

Mathématiquement, l'empreinte d'une liste pourrait être définie ainsi :

$$\texttt{islist\_ft}(next, p) = \begin{cases} \emptyset & \text{si } p = \texttt{null} \\ \texttt{islist\_ft}(next[p]) \cup \{p\} & \text{si } p \neq \texttt{null} \text{ et } \texttt{islist}(next[p]) \end{cases}$$

Les démonstrateurs automatiques préfèrent souvent que l'on utilise des concepts moins généraux, comme l'union et l'ensemble vide. De plus ils préfèrent souvent que les formules soient plus simples. Les quatre axiomes suivant définissent l'empreinte de cette manière :

1. L'empreinte de la liste vide est vide :

   ```
   function islist_ft (next : next) (p : pointer) : ft

   axiom islist_ft_node_null : ∀ next : next, q : pointer,p :
   pointer. q=null ⇒ ~in_ft p (islist_ft next q)
   ```

2. Le pointeur peut appartenir au reste de la liste :

```
axiom islist_ft_node_next1 : ∀ next : next, q : pointer, p :
pointer. q≠null ⇒ islist next (get next q) ⇒ in_ft p (islist_ft
next (get next q)) ⇒ in_ft p (islist_ft next q)
```

3. ou est simplement le pointeur de tête :

```
axiom islist_ft_node_next2 : ∀ next : next, q : pointer.
q≠null ⇒ islist next (get next q) ⇒ in_ft q (islist_ft next q)
```

4. Réciproquement si un pointeur de l'empreinte n'est pas le pointeur de tête alors il appartient au reste de la liste :

```
axiom islist_ft_node_next_inv : ∀ next : next, q : pointer, p :
pointer. q≠null ⇒ islist next (get next q) ⇒ q≠p ⇒ in_ft p
(islist_ft next q) ⇒ in_ft p (islist_ft next (get next q))
```

Maintenant que l'empreinte est définie, on peut ajouter un axiome d'encadrement qui spécifie une condition suffisante pour qu'une modification de l'état mémoire n'invalide pas le prédicat `islist` :

```
lemma frame_islist :
  ∀ next : next, q : pointer, p : pointer, v : pointer
  [islist (set next p v) q].
  (~ in_ft p (islist_ft next q)) ⇒
  islist next q ⇒ islist (set next p v) q
```

D'autres conditions suffisantes peuvent être imaginées à la place de la condition suivante `~ in_ft p (islist_ft next q)`, par exemple `v=null` ou `islist next v`. Une modification qui vérifie une de ces deux conditions modifiera peut-être la longueur ou les éléments contenus dans des listes mais pas le fait qu'elles vérifient `islist`. La condition que l'on a choisie est d'abord celle qui nous intéresse dans notre cas, mais c'est également la seule des conditions prises en exemple qui utilise réellement le fait que les listes soient séparées. En effet les deux autres conditions sont toujours vraies quelle que soit la manière dont les listes peuvent se trouver en mémoire.

On doit également ajouter dans l'invariant de boucle `~in_ft p (list_ft next q)`. Cela permet de prouver que `islist next' q` est vérifié après l'affectation. Cependant cela ne permet pas de prouver que `islist next' tmp` est préservé ni que la partie de l'invariant que l'on vient d'ajouter, `~in_ft p' (islist_ft next' q')`, est vérifiée après le corps de la boucle. La première propriété se prouve grâce à l'axiome d'encadrement de `islist` et grâce au fait qu'avant l'affectation on a `~in_ft p (islist_ft next tmp)`. Lui même découle par un raisonnement par récurrence de la finitude des listes considérées. On a donc besoin du lemme suivant qui pourra se prouver avec un outil capable de faire de l'induction, comme l'assistant de preuve `Coq` :

```
lemma acyclic_list : ∀ next : next, p : pointer.
p ≠ null ⇒ islist next p                                    (2.1)
⇒ ~in_ft p (islist_ft next (get next p))
```

La deuxième propriété, `~in_ft p' (islist_ft next' q')`, ne peut être prouvée avec l'actuel invariant car on ne sait rien sur `p'` l'élément suivant de la liste. On doit renforcer `~in_ft p (islist_ft next q)`, en exprimant l'idée que tous les éléments de la liste pointée par `p` ne sont pas dans l'empreinte de `q`. Ce qui revient à :

```
∀ x : pointer.
  in_ft x (islist_ft next p) ⇒ ~ in_ft x (islist_ft next q)
```

Cette propriété est symétrique, comme le montrent les règles de De Morgan. Elle exprime la séparation de deux listes. Pour simplifier l'utilisation de cette propriété on introduit un prédicat `sep_islist_islist` :

```
predicate sep_islist_islist (next : next) (p1 p2 : pointer) =
  ∀ q : pointer.
  (~ in_ft q (islist_ft next p1)) ∨ (~in_ft q (islist_ft next p2))
```

Finalement l'invariant de boucle devient

```
invariant { islist next p ∧ islist next q
            ∧ sep_islist_islist next p q }
```

Pour prouver la conservation il ne reste plus qu'à remarquer que si l'on ne modifie pas un pointeur de l'empreinte `islist_ft next q` alors l'empreinte elle-même n'est pas modifiée.

```
lemma frame_islist_ft :
  ∀ next : next, p : pointer, q : pointer, v : pointer.
  islist next p ⇒
  (~ in_ft q (islist_ft next p)) ⇒
  islist_ft next p ⇔ islist_ft (set next q v) p
```

Le programme avec les annotations finales se trouve dans la figure 2.1.

La condition de vérification est déchargée automatiquement en moins d'une seconde par les démonstrateurs CVC3 [2] et Z3 [24]. Les deux lemmes d'encadrement ainsi que le lemme d'acyclicité ont été prouvés en `Coq`.

Comportement Nous n'avons pour l'instant pas prouvé que la fonction `list_rev` inverse bien la liste. Pour le prouver on définit un modèle pour les listes simplement chaînées en utilisant le type algébrique des listes de la bibliothèque standard de `Why`.

```
use import list.Append
use import list.Reverse

function model (next : next) (p : pointer) : list pointer

axiom model_def1 : ∀ next : next, p : pointer.
  p = null ⇒ model next p = Nil

axiom model_def2 : ∀ next : next, p : pointer[model next p].
```

```
let list_rev (p : ref pointer) =
  { islist next p }
  let q = ref null in
  while !p ≠ null do
    invariant { islist next p ∧ islist next q
                ∧ sep_islist_islist next p q }
    let tmp = get !next !p in
    next := set !next !p !q;
    q := !p;
    p:= tmp
  done;
  q
  { islist next result }
```

FIGURE 2.1.: Annotation de sûreté de l'inversion de liste en place.

```
  p ≠ null ⇒ islist next (get next p) ⇒
  model next p = Cons p (model next (get next p))
```

Ensuite on remarque que ce modèle est lui aussi encadré par `islist_ft` :

```
lemma frame_model :
  ∀ ft : ft, next : next,
  p : pointer, q : pointer, v : pointer[model (set next q v) p].
  islist next (get next p) ⇒
  ~ in_ft q (islist_ft next p) ⇒
    model next p = model (set next q v) p
```

Le programme devient alors :

```
let list_rev_behv (p : ref pointer) =
{ islist next p }
label L:
let q = ref null in
while !p ≠ null do
  invariant { islist next p ∧ islist next q
              ∧ sep_islist_islist next p q ∧
              reverse (at (model next p) L) =
                (reverse (model next p)) ++ (model next q)}
  let tmp = get !next !p in
  let bak_next = !next in
  next := set !next !p !q;
  q := !p;
  p:= tmp
```

```
done;
q
{ islist next result ∧
  reverse (old (model next p)) = model next result}
```

La condition de vérification est déchargée automatiquement en quelques secondes par les démonstrateurs CVC3 et Z3.

Dans la partie suivante nous allons voir comment générer automatiquement `islist_ft`, `sep_islist_islist` ainsi que tous les axiomes d'encadrement correspondants à partir d'une définition inductive telle que `islist`.

Avant d'aller plus loin on peut remarquer que des prédicats qui ne sont pas récursifs n'ont pas besoin d'axiome d'encadrement. En effet si le prédicat n'est pas récursif pour montrer qu'il est conservé aucune preuve par récurrence n'est requise. Ainsi l'ensemble des pointeurs qui ont été modifiés, la définition du prédicat qui fait intervenir `get` et `set` et la théories des tableaux suffisent à montrer que le prédicat est conservé.

3. Automatisation

Nous allons maintenant expliquer comment ce qui a été présenté dans le chapitre précédent est automatisé dans l'outil `Jessie`. En particulier nous allons décrire le langage d'entrée, le langage de sortie et comment s'effectue la génération de prédicats de séparation. La figure 3.1 résume, dans le cas de l'inversion de liste, le programme source `Jessie` et le programme `Why` obtenu. Dans la colonne de gauche, on note la présence de la déclaration `pragma` : `sepll`(`islist`, `islist`) qui demande la génération d'un prédicat `sepll` exprimant la séparation entre deux listes. Les axiomes `get_set_eq` et `get_set_diseq` sont les axiomes habituelles de la théorie des tableaux persistants polymorphes (cf. 1.4 et 1.5 p.8). Nous ferons référence à cette figure tout au long de ce chapitre.

3.1. Description des langages d'entrée et de sortie

3.1.1. Syntaxe et typage de mini-`Jessie`

Bien qu'en `Jessie` on puisse écrire des programmes, ici nous ne les formaliserons pas. En effet, le but est de montrer que nos transformations sont correctes. Les transformations sur les programmes sont identiques à celles de `Jessie`. Nous pourrons donc définir les transformations uniquement sur les formules et ensuite montrer que les spécifications de programme sont vraies dans un état mémoire donné si et seulement si la spécification transformée dans l'état mémoire transformé est vraie.

Le langage d'entrée sera un *mini*-`Jessie`. Ce langage permettra de décrire des fonctions récursives, des prédicats récursifs, des prédicats inductifs ainsi que des fonctions abstraites, des prédicats abstraits et des axiomatiques. Les types sont limités aux entiers relatifs I et aux pointeurs P. Dans les termes on peut accéder aux différents champs d'un pointeur et donc la sémantique des fonctions logiques dépend d'un état mémoire. Pour simplifier la présentation nous supposons qu'il n'y a qu'une seule structure qui contient tous les champs. Ainsi tous les pointeurs peuvent être déréférencés sur tous les champs. Les champs peuvent être soit des entiers relatifs, soit des pointeurs.

	$f_0 : \tau_0$	$\ldots$	$f_n : \tau_n$	$\ldots$
p_1	7		p_3	
$\vdots$				
p_m	42		p_1	
$\vdots$				

Les champs sont fixés et ils seront référencés par un numéro. Le type du $i^{\text{ième}}$ champ est noté τ_i. Dans l'exemple précédent τ_0 vaut I et τ_n vaut P.

Jessie	Why
	$\texttt{type}\,\mathsf{I}$
	$\texttt{type}\,\mathsf{P}$
$\texttt{struct}\,\texttt{list}\{$	$\texttt{type}\,\mathsf{M}\,\alpha$
$\quad\texttt{next} : \texttt{list};$	$\texttt{fun}\,\texttt{get}(m : \mathsf{M}\,\alpha, p : \mathsf{P}) : \alpha$
$\quad\texttt{val} : \mathsf{I};$	$\texttt{fun}\,\texttt{set}(m : \mathsf{M}\,\alpha, p : \mathsf{P}, v : \alpha) : \alpha$
$\}$	$\texttt{axiom}\,get\_set\_eq\ldots$
	$\texttt{axiom}\,get\_set\_diseq\ldots$
$\texttt{ind}\,\texttt{islist}(p : list) =$	$\texttt{ind}\,\texttt{islist}(m_1 : \mathsf{M}\,\mathsf{P}, p : \mathsf{P}) =$ (3.1)
$\quad\lvert nil : \forall p : list.p \approx null \Rightarrow$	$\quad\lvert nil : \forall m_1 : \mathsf{M}\,\mathsf{P}, p : \mathsf{P}.p \approx null \Rightarrow$
$\quad\texttt{islist}(p)$	$\quad\texttt{islist}(m_1, p)$
$\quad\lvert cons : \forall p : list.p \not\approx null \Rightarrow$	$\quad\lvert cons : \forall m_1 : \mathsf{M}\,\mathsf{P}, \forall p : \mathsf{P}.p \not\approx null \Rightarrow$
$\quad\texttt{islist}(p.next) \Rightarrow \texttt{islist}(p)$	$\quad\texttt{islist}(m_1, \texttt{get}(m_1, p)) \Rightarrow \texttt{islist}(m_1, p)$
	$\texttt{axiom}\,islist\_disjoint\ldots$ (3.2)
	$\texttt{type}ft$ (3.3)
	$\texttt{pred}\,\in\!: (p : \mathsf{P}, s : ft)$ (3.4)
	$\texttt{fun}\,\texttt{ft}_{\texttt{islist}}(m : \mathsf{M}\,\mathsf{P}, p : \mathsf{P}) : ft$ (3.5)
$\texttt{pragma}\,\texttt{sepll}(\texttt{islist}, \texttt{islist})$	$\texttt{axiom}\,ft\_def\ldots$ (3.6)
	$\texttt{axiom}\,ft\_frame\ldots$ (3.7)
	$\texttt{axiom}\,ft\_autoframe\ldots$ (3.8)
	$\texttt{pred}\,\texttt{sepll}(m : \mathsf{M}\,\mathsf{P}, p_1 : \mathsf{P}, p_2 : \mathsf{P}) =$ (3.9)
	$\forall q : \mathsf{P}.\neg q \in \texttt{ft}_{\texttt{islist}}(m, p_1) \vee \neg q \in \texttt{ft}_{\texttt{islist}}(m, p_2)$
$\texttt{list}\,\texttt{rev}(p : \texttt{list})\{$	
...	$\texttt{let}\,\texttt{rev}(m : \mathsf{M}\,\mathsf{P}, p : \mathsf{P}) =$
$\texttt{invariant}\begin{pmatrix}\texttt{islist}(p)\wedge\\ \texttt{islist}(q)\wedge\\ \texttt{sepll}(p,q)\end{pmatrix}$	...
	$\texttt{invariant}\begin{pmatrix}\texttt{islist}(m,p)\wedge\texttt{islist}(m,q)\\ \wedge\,\texttt{sepll}(m,p,q)\end{pmatrix}$
...	...
$\}$	

FIGURE 3.1.: Plan d'ensemble des axiomes, fonctions et prédicats générés.

$$
\begin{aligned}
T &::= \mathsf{I} \mid \mathsf{P} \\
t &::= x \mid t.i \mid \phi(t, \ldots, t) \\
f &::= t \approx t \mid p(\mathbf{x}) \mid \neg(t \approx t) \\
f^E &::= f \mid \forall y.f^E \mid f^E \wedge f^E \mid \neg f^E \\
c &::= c, \mathtt{C} : \forall \mathbf{x}.f \Rightarrow \cdots \Rightarrow f \Rightarrow p(\mathbf{x}) \mid \\
d &::= \mathtt{fun}\, \phi(\mathbf{x}) : T = t \\
&\quad \mid \mathtt{pred}\, p(\mathbf{x}) = f \\
&\quad \mid \mathtt{fun}\, \phi(\mathbf{x}) : T\{n, \ldots, n\} \\
&\quad \mid \mathtt{pred}\, p(\mathbf{x})\{n, \ldots, n\} \\
&\quad \mid \mathtt{ind}\, p(\mathbf{x}) = c \\
&\quad \mid \mathtt{axiom} : f^E \\
&\quad \mid \mathtt{pragma} : p(p, p) \\
\Delta &::= d; \Delta \mid
\end{aligned}
$$

FIGURE 3.2.: Syntaxe d'entrée de mini-`Jessie`.

La syntaxe d'entrée est décrite dans la figure 3.2. On dénote $x_1 : T_1, \ldots, x_n : T_n$ par $\mathbf{x}$. On utilise ici $\approx$ pour dénoter une égalité de manière à la distinguer dans les preuves de l'égalité $=$ de la logique utilisé dans les preuves de ce manuscrit. Les quantifications ne peuvent apparaître que dans des endroits particuliers des définitions de prédicats inductifs ; f correspond aux formules restreintes et f^E correspond aux formules étendues. Ces restrictions sont enlevées pour les axiomes, les définitions de fonctions et de prédicats non inductifs. On dit qu'un symbole de fonction est abstrait si, lors de sa déclaration, il n'a pas été défini. Un prédicat inductif est dit totalement défini si sa définition ne contient que des symboles de prédicats inductifs totalement définis ou bien des symboles de fonctions ou de prédicats qui ne dépendent pas de l'état mémoire. Une suite de déclarations est notée Δ. Découvrons l'exemple utilisé dans ce chapitre : la définition de liste bien formée. On suppose que le champ 1 correspond au champ `next` et donc $\tau_1 = \mathsf{P}$:

$$
\begin{aligned}
&\mathtt{ind}\, \mathtt{islist}(p : \mathsf{P}) = \\
&\quad \mid \mathit{islist\_null} : \\
&\qquad \forall p : \mathsf{P}.\ p \approx \mathtt{null} \Rightarrow \mathtt{islist}(p) \\
&\quad \mid \mathit{islist\_next} : \\
&\qquad \forall p : \mathsf{P}.\ \neg p \approx \mathtt{null} \Rightarrow \mathtt{islist}(p.1) \Rightarrow \mathtt{islist}(p)
\end{aligned}
$$

La forme des inductifs choisie semble assez restrictive. Les formules ne doivent pas

contenir de connecteur logique, l'occurrence positive doit être exactement l'application du prédicat inductif $p(\mathbf{x})$ et enfin il n'y a pas de quantification pour d'autres variables que $\mathbf{x}$. De plus on demande que tous les cas soient distincts, c'est-à-dire que deux cas ne peuvent pas être vrais en même temps. Nous allons voir que ces restrictions peuvent être souvent contournées :

- Si un cas est de la forme $|C : \forall x.f_1 \vee f_2 \Rightarrow p(x)$, il peut être transformé en

$$|C_1 : \forall x.f_1 \Rightarrow p(x)$$
$$|C_2 : \forall x.f_2 \Rightarrow p(x)$$

  Bien sûr dans ce cas f_1 et f_2 doivent être distinctes. On pourrait en fait autoriser les disjonctions mais on devrait alors demander que les deux possibilités soient distinctes, ce qui reviendrait au même qu'actuellement.
- Si un cas est de la forme $|C : f \Rightarrow p(t)$, il peut être transformé en $|C : \forall x.f \Rightarrow x = t \Rightarrow p(x)$.
- Si un cas est de la forme $|C : \forall x \forall y.f[y] \Rightarrow p(x)$ et que la valeur de y ne dépend pas de l'état mémoire alors on peut introduire une fonction non interprétée en tant que symbole de skolem : $|C : \forall x.f[\phi(x)] \Rightarrow p(x)$.
- Si les cas de l'inductif ne sont pas distincts on peut s'y ramener en modifiant la définition de l'inductif. Par exemple, avec A et B des formules qui peuvent être vraies en même temps, la déclaration

$$\begin{aligned}&\texttt{ind}\, p(x : \mathsf{P}) :\\ &\quad |C_1 : \forall x : \mathsf{P}.A \Rightarrow p(x : \mathsf{P})\\ &\quad |C_2 : \forall x : \mathsf{P}.B \Rightarrow p(x : \mathsf{P})\end{aligned}$$

  peut être transformée en un inductif qui vérifie la condition :

$$\begin{aligned}&\texttt{ind}\, p(x : \mathsf{P}) :\\ &\quad |C_1 : \forall x : \mathsf{P}.A \Rightarrow \neg B \Rightarrow p(x : \mathsf{P})\\ &\quad |C_2 : \forall x : \mathsf{P}.B \Rightarrow \neg A \Rightarrow p(x : \mathsf{P})\\ &\quad |C_3 : \forall x : \mathsf{P}.A \Rightarrow B \Rightarrow p(x : \mathsf{P})\end{aligned}$$

  Malheureusement si A est un appel de prédicat, $\neg A$ ne vérifie plus la contrainte syntaxique. On peut étendre la syntaxe pour accepter les négations de prédicats purs, sans changer la formalisation. Pour ce qui est des prédicats impures cela est bien plus compliqué car il faut traiter sa négation de manière particulière et comme elle ne respectent pas la propriété des cas disjoints les empreintes générées seront très peu précise.

  On reviendra plus tard sur l'utilité de cette condition. On peut remarquer que `islist` vérifie cette condition : p ne peut pas être à la fois `null` et non `null`.

Les règles de typage sont décrites dans la figure 3.3. Γ est l'environnement de typage : il contient à la fois le type des variables libres, $y : T$, mais également les signatures des fonctions et prédicats, $\phi : \langle T_1, \ldots, T_n \rangle$. Δ dénote une séquence de déclarations. Les règles de typage déterminent les termes, formules ou les listes de déclarations bien formés :

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash t : \mathsf{P}}{\Gamma \vdash t.i : \tau_i} \qquad \frac{\Gamma \vdash t_i : T_i \qquad \phi : \langle T_1, \ldots, T_n, T\rangle \in \Gamma}{\Gamma \vdash \phi(t_1, \ldots, t_n) : T}$$

$$\frac{\Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1 \approx t_2} \qquad \frac{\Gamma \vdash t_i : T_i \qquad p : \langle T_1, \ldots, T_n\rangle \in \Gamma}{\Gamma \vdash p(t_1, \ldots, t_n)} \qquad \frac{\Gamma, y : T \vdash f}{\Gamma \vdash \forall y : T.f}$$

$$\frac{\Gamma \vdash f}{\Gamma \vdash \neg f} \qquad \frac{\Gamma \vdash f_1 \qquad \Gamma \vdash f_2}{\Gamma \vdash f_1 \wedge f_2}$$

$$\frac{\Gamma' = \Gamma, \phi : \langle T_1, \ldots, T_n, T\rangle \qquad \Gamma', x_1 : T_1, \ldots, x_n : T_n \vdash t : T \qquad \Gamma' \vdash \Delta}{\Gamma' \vdash \mathtt{fun}\, \phi(x_1 : T_1, \ldots, x_n : T_n) : T = t; \Delta}$$

$$\frac{\Gamma' = \Gamma, p : \langle T_1, \ldots, T_n\rangle \qquad \Gamma', x_1 : T_1, \ldots, x_n : T_n \vdash f \qquad \Gamma' \vdash \Delta}{\Gamma \vdash \mathtt{pred}\, p(x_1 : T_1, \ldots, x_n : T_n) = f; \Delta}$$

$$\frac{\Gamma' = \Gamma, p : \langle T_1, \ldots, T_n\rangle \qquad \Gamma', \mathbf{x}, \vdash f_{i,j} \qquad \Gamma' \vdash \Delta}{\Gamma \vdash \mathtt{ind}\, p(x_1 : T_1, \ldots, x_n : T_n) = C_i : \forall \mathbf{x}. \forall \mathbf{y_i}. f_{i,1} \Rightarrow \cdots \Rightarrow f_{i,n} \Rightarrow p(\mathbf{x}); \Delta}$$

$$\frac{\Gamma, \phi : \langle T_1, \ldots, T_n, T\rangle \vdash \Delta}{\Gamma \vdash \mathtt{fun}\, \phi(x_1 : T_1, \ldots, x_n : T_n) : T; \Delta} \qquad \frac{\Gamma, p : \langle T_1, \ldots, T_n\rangle \vdash \Delta}{\Gamma \vdash \mathtt{pred}\, \phi(x_1 : T_1, \ldots, x_n : T_n); \Delta}$$

$$\frac{\Gamma \vdash f \qquad \Gamma \vdash \Delta}{\Gamma \vdash \mathtt{axiom} : f; \Delta} \qquad \frac{\begin{array}{c}\Gamma, p : \langle T_{1,1}, \ldots, T_{1,n_1}, T_{2,1}, \ldots, T_{2,n_2}\rangle \vdash \Delta \\ p_i : \langle T_{i,1}, \ldots, T_{i,n_i}, \_\rangle \in \Gamma \qquad p_i \text{ est totalement défini}\end{array}}{\Gamma \vdash \mathtt{pragma} : p(p_1, p_2); \Delta}$$

FIGURE 3.3.: Règles de typage de mini-`Jessie`.

- $\Gamma \vdash t : T$ signifie que dans l'environnement de typage Γ le terme t est bien formé et est de type T.
- $\Gamma \vdash f$ signifie que dans l'environnement de typage Γ le terme f est bien formé.
- $\Gamma \vdash \Delta$ signifie que dans l'environnement de typage Γ la liste Δ de déclarations est bien formée.

Dans la dernière règle, $p_i : \langle\, T_{i,1}, \ldots, T_{i,n_i}\,\rangle \in \Gamma$ signifie que p_i est un symbole de prédicat inductif déjà déclaré et prenant des arguments de type $T_{i,1}, \ldots, T_{i,n_i}$.

Une définition de fonction ou de prédicat ne fait apparaître qu'un nombre fini de champs. Il est utile de calculer l'ensemble de ces champs car cela représente la séparation obtenue par la séparation par champs. On le notera $\mathtt{MEM}(\phi)$ (resp. $\mathtt{MEM}(p)$) pour une fonction ϕ (resp. un prédicat p). Pour une fonction (resp. pour un prédicat) définie, $\mathtt{fun}\, \phi(\mathbf{x}) : T = t$ (resp. $\mathtt{pred}\, p(\mathbf{x}) = f$), on définit $\mathtt{MEM}(\phi)$ comme $\mathtt{MEM}(t)$ (resp. $\mathtt{MEM}(p)$ comme $\mathtt{MEM}(f)$). `MEM` est défini sur les termes et les formules par récurrence comme décrit

$$\texttt{MEM}(x) \triangleq \emptyset \qquad \texttt{MEM}(t.i) \triangleq \texttt{MEM}(t) \cup \{i\} \qquad \texttt{MEM}(t_1 \approx t_2)) \triangleq \texttt{MEM}(t_1) \cup \texttt{MEM}(t_2)$$

$$\texttt{MEM}(\phi(t_1, \ldots, t_n)) \triangleq \texttt{MEM}(\phi) \cup \bigcup_i \texttt{MEM}(t_i) \qquad \texttt{MEM}(p(t_1, \ldots, t_n)) \triangleq \texttt{MEM}(p) \cup \bigcup_i \texttt{MEM}(t_i)$$

$$\texttt{MEM}(\neg f) \triangleq \texttt{MEM}(f) \qquad \texttt{MEM}(\forall y : T.f) \triangleq \texttt{MEM}(f) \qquad \texttt{MEM}(f_1 \wedge f_2) \triangleq \texttt{MEM}(f_1) \cup \texttt{MEM}(f_2).$$

FIGURE 3.4.: Définition de MEM pour calculer l'ensemble dont dépend une définition de fonction ou prédicat.

dans la figure 3.4. Dans le cas des fonctions et prédicats abstraits cet ensemble est décrit par l'utilisateur lors de la déclaration. Par exemple la déclaration d'un symbole abstrait de fonctions $\texttt{fun}\,\phi(\mathbf{x}) : T\{i_1, \ldots, i_m\}$ indique que $\texttt{MEM}(\phi) \triangleq \{i_1, \ldots, i_m\}$. Dans le cas d'un inductif $\texttt{ind}\,p(\mathbf{x}) = \texttt{C}_\texttt{i} : \forall \mathbf{x}.f_{i,1} \Rightarrow \cdots \Rightarrow f_{i,n} \Rightarrow p(\mathbf{x})$, on définit $\texttt{MEM}(p) \triangleq \bigcup_i \bigcup_j \texttt{MEM}(f_{i,j})$. Dans le cas des prédicats de séparation définis à partir d'un pragma, $\texttt{pragma} : p(p_1, p_2)$, on définit $\texttt{MEM}(p) \triangleq \texttt{MEM}(p_1) \cup \texttt{MEM}(p_2)$. On a pas besoin ici d'un point fixe, ou plutôt en partant de $\texttt{MEM}(p) = \emptyset$ on atteint tout de suite le point fixe puisqu'il n'y a pas de fonctions mutuellement récursives. Pour `islist` on a $\texttt{MEM}(\texttt{islist}) = 1$.

Quelle que soit la manière de définir ϕ, $\texttt{MEM}(\phi)$ est tel que $\phi(\mathbf{x})$ est identique dans deux états mémoires dont les restrictions à $\texttt{MEM}(\phi)$ sont identiques. Un ensemble trop petit peut induire une incohérence, comme dans l'exemple suivant :

$$\begin{aligned}
&\texttt{fun}\,p(x : \mathsf{P}) : T\{\} \\
&\texttt{axiom} : \forall x : \mathsf{P}.x.1 = \texttt{null} \Rightarrow p(x) \\
&\texttt{axiom} : \forall x : \mathsf{P}.\neg x.1 = \texttt{null} \Rightarrow \neg p(x)
\end{aligned}$$

En effet les deux axiomes impliquent que la valeur de $p(x)$ dépend du champ 1 alors que sa déclaration indique que p ne dépend pas de l'état mémoire.

3.1.2. Interprétation

L'interprétation $\mathfrak{I}$ d'une liste Δ de déclarations bien formée, c'est-à-dire $\vdash \Delta$, est définie de manière habituelle. Elle associe un domaine pour les entiers relatifs $\mathcal{D}^{\mathfrak{I}}_{\mathsf{I}}$ et un domaine pour les pointeurs $\mathcal{D}^{\mathfrak{I}}_{\mathsf{P}}$. À partir de ces domaines on définit le domaine des états mémoires globaux $\mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}$ comme l'ensemble des fonctions totales du type :

$$i : \mathbb{N} \mapsto \mathcal{D}^{\mathfrak{I}}_{\mathsf{P}} \mapsto \mathcal{D}^{\mathfrak{I}}_{\tau_i}$$

L'interprétation $\mathfrak{I}$ interprète également les symboles de fonctions et de prédicats :

– l'interprétation d'un symbole de fonction $\phi : \langle T_1, \dots T_n, T\rangle \in \Delta$ est une fonction mathématique totale $\phi^{\mathfrak{I}} : \mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}, \mathcal{D}^{\mathfrak{I}}_{T_1}, \dots, \mathcal{D}^{\mathfrak{I}}_{T_n} \mapsto \mathcal{D}^{\mathfrak{I}}_{T}$,
– l'inteprétation d'un symbole de prédicat $p : \langle T_1, \dots T_n\rangle \in \Delta$ est un prédicat mathématique total $p^{\mathfrak{I}} : \mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}, \mathcal{D}^{\mathfrak{I}}_{T_1}, \dots, \mathcal{D}^{\mathfrak{I}}_{T_n}$.

L'interprétation des termes et formules est alors définie dans un état mémoire $m \in \mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}$ en utilisant une valuation ψ, c'est-à-dire une fonction totale des symboles de variables vers le domaine de leur type. On note l'extension d'une valuation ψ par $\psi[x \mapsto v]$ et elle est définie par

$$\psi[x \mapsto v](y) = \begin{cases} v & \text{si } x = y \\ \psi(y) & \text{sinon} \end{cases}.$$

On a alors

$$\begin{aligned}
\mathfrak{I}_{m,\psi}(x) &= \psi(x), \\
\mathfrak{I}_{m,\psi}(\phi(t_1, \dots, t_n)) &= \phi^{\mathfrak{I}}(m, \mathfrak{I}_{m,\psi}(t_1), \dots, \mathfrak{I}_{m,\psi}(t_n)), \\
\mathfrak{I}_{m,\psi}(t.i) &= m(i)(\mathfrak{I}_{m,\psi}(t)), \\
\mathfrak{I}_{m,\psi}(p(t_1, \dots, t_n)) &= p^{\mathfrak{I}}(m, \mathfrak{I}_{m,\psi}(t_1), \dots, \mathfrak{I}_{m,\psi}(t_n)), \\
\mathfrak{I}_{m,\psi}(t_1 \approx t_2) &= \mathfrak{I}_{m,\psi}(t_1) = \mathfrak{I}_{m,\psi}(t_2), \\
\mathfrak{I}_{m,\psi}(\neg f) &= \neg \mathfrak{I}_{m,\psi}(f), \\
\mathfrak{I}_{m,\psi}(\forall y : T.\ f) &= \forall v : \mathcal{D}^{\mathfrak{I}}_{T}. \mathfrak{I}_{m,\psi[y \mapsto v]}(f), \\
\mathfrak{I}_{m,\psi}(f_1 \wedge f_2) &= \mathfrak{I}_{m,\psi}(f_1) \wedge \mathfrak{I}_{m,\psi}(f_2).
\end{aligned}$$

L'interprétation satisfait une déclaration d de Δ dans les cas suivants :

– c'est la déclaration d'une fonction ou d'un prédicat abstrait $\phi : \langle T_1, \dots, T_n, \_\rangle$ qui ne dépend que des champs de $\texttt{MEM}(\phi)$, c'est-à-dire que pour tous états mémoire $m_1, m_2 \in \mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}$, pour toutes valeurs $v_i \in \mathcal{D}^{\mathfrak{I}}_{T_i}$ si pour tout champ i qui n'est pas dans $\texttt{MEM}(\phi)$ on a $m_1(i) = m_2(i)$ alors on a $\phi(m_1, v_1, \dots, v_n) = \phi(m_2, v_1, \dots, v_n)$,
– c'est la définition d'un symbole de fonction $\texttt{fun}\,\phi(x_1 : T_1, \dots, x_n : T_n) : T = t$ et pour tout état mémoire $m \in \mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}$, pour toute valuation ϕ, pour toutes valeurs $v_i \in \mathcal{D}^{\mathfrak{I}}_{T_i}$ on a $\phi^{\mathfrak{I}}(m, v_1, \dots, v_n) = \mathfrak{I}_{m,\psi[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]}(t)$,
– c'est la définition d'un symbole de prédicat $\texttt{pred}\,p(x_1 : T_1, \dots, x_n : T_n) = f$ et pour tout état mémoire $m \in \mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}$, pour toute valuation ϕ, pour toutes valeurs $v_i \in \mathcal{D}^{\mathfrak{I}}_{T_i}$ on a $p^{\mathfrak{I}}(m, v_1, \dots, v_n)$ si et seulement si $\mathfrak{I}_{m,\psi[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]}(f)$,
– c'est la définition d'un prédicat inductif $\texttt{ind}\,p(\mathbf{x}) = \texttt{C}_\texttt{i} : \forall \mathbf{x}. f_{i,1} \Rightarrow \dots \Rightarrow f_{i,n_i} \Rightarrow p(\mathbf{x})$. On pose ψ un raccourci pour $[x_1 \mapsto v_1, \dots, x_n \mapsto v_n]$ et on définit :

$$p^{\mathfrak{I}} \triangleq \mu p^{\mathfrak{I}}. \lambda m, v_1, \dots, v_n. \left(\bigvee_i \mathfrak{I}_{m,\psi}(f_{i,1} \wedge \dots \wedge f_{i,n_i}) \right).$$

μ est l'opérateur de plus petit point fixe. $p^{\mathfrak{I}}$ apparaît dans la disjonction par $\mathfrak{I}_{m,\psi}(p(\mathbf{t})) = p^{\mathfrak{I}}(\mathfrak{I}_{m,\psi}(\mathbf{t}))$ si p est bien récursif.

L'interprétation doit en plus vérifier que tous les cas sont mutuellement exclusifs, c'est-à-dire que pour tout cas distinct C_i et C_j, $i \neq j$, de l'inductif p :

$$\mathfrak{I}m, \psi(f_{i,1} \wedge \cdots \wedge f_{i,n_i} \wedge f_{j,1} \wedge \cdots \wedge f_{j,n_j}) = \bot \tag{3.10}$$

– c'est un axiome `axiom` : f et pour tout état mémoire $m \in \mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}$, pour toute valuation ϕ l'interprétation $\mathfrak{I}m, \phi(f)$ est vraie.

Pour simplifier la présentation, nous ne permettons pas d'avoir d'inductif mutuellement récursif.

Nous allons maintenant définir l'interprétation des prédicats déclarés par pragma. Soit un prédicat p défini par `pragma` : $p(p_1, p_2)$ avec le prédicat $p_1 : \langle T_{1,1}, \ldots, T_{1,n_1} \rangle$ et $p_2 : \langle T_{2,1}, \ldots, T_{2,n_2} \rangle$. Nous allons définir une sur-approximation du fait que p_1 et p_2 ne dépendent pas des mêmes pointeurs. La dépendance d'un prédicat envers un pointeur sur un champ peut être caractérisée par la modification de la véracité du prédicat si l'on modifie la valeur du champ pour ce pointeur. Plus formellement, dans un état $m \in \mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}$ et pour des valeurs $(v_1, \ldots, v_n)$ tels que $p_1^{\mathfrak{I}}(m, v_1, \ldots, v_n)$ est vrai, $p_1^{\mathfrak{I}}(v_1, \ldots, v_n)$ dépend de la paire (q, i), avec $q \in \mathcal{D}^{\mathfrak{I}}_{\mathsf{P}}$ un pointeur et i un champ, si et seulement si il existe un état mémoire m' qui ne diffère de m qu'en ce champ i et en ce pointeur q et tel que $p_1^{\mathfrak{I}}(m', v_1, \ldots, v_n)$ est faux. Enfin on dit que dans un état mémoire $m \in \mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}$ pour des valeurs $v_{k,l} \in \mathcal{D}^{\mathfrak{I}}_{T_{k,l}}$, $p^{\mathfrak{I}}(m, v_{1,1}, \ldots, v_{1,n_1}, v_{2,1}, \ldots, v_{2,n_2})$ est vrai quand $p_1^{\mathfrak{I}}(m, v_{1,1}, \ldots, v_{1,n_1})$ et $p_2^{\mathfrak{I}}(m, v_{2,1}, \ldots, v_{2,n_2})$ sont vrais si et seulement si les ensembles de paires de pointeurs et de champs dont dépendent $p_1^{\mathfrak{I}}(m, v_{1,1}, \ldots, v_{1,n_1})$ et $p_2^{\mathfrak{I}}(m, v_{2,1}, \ldots, v_{2,n_2})$ sont disjoints. On veut utiliser ces définitions pour pouvoir déterminer que $p_1^{\mathfrak{I}}(m, v_{1,1}, \ldots, v_{1,n_1})$ est inchangé entre m et m' par des informations que l'on connaît sur $p_2^{\mathfrak{I}}(m, v_{1,1}, \ldots, v_{1,n_1})$. Or savoir si $p_2^{\mathfrak{I}}(m, v_{1,1}, \ldots, v_{1,n_1})$ dépend d'un pointeur sur un champ peut être aussi difficile à déterminer que de savoir si $p_1^{\mathfrak{I}}(m, v_{1,1}, \ldots, v_{1,n_1})$ est toujours vrai après avoir modifié ce pointeur sur ce champ. On choisit donc de prendre une sur-approximation de l'ensemble de paires de champs et pointeurs dont dépend $p_2^{\mathfrak{I}}(m, v_{1,1}, \ldots, v_{1,n_1})$ de sorte qu'il soit plus simple de prouver qu'une paire de champs et de pointeurs y appartient.

Pour cela à partir d'une interprétation $\mathfrak{I}$ on définit pour chaque prédicat $p : T_1, \ldots, T_n$ totalement défini, son empreinte $(\mathtt{ft}_p)^{\mathfrak{I}} : \mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}, \mathcal{D}^{\mathfrak{I}}_{T_1}, \ldots, \mathcal{D}^{\mathfrak{I}}_{T_n} \mapsto 2^{\mathbb{N} \times \mathcal{D}^{\mathfrak{I}}_{\mathsf{P}}}$ par point fixe. Pour une formule ou un terme t on définit la fonction $\overline{\mathtt{FT}}_p(t)$ comme une fonction f de type $\mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}, \mathcal{D}^{\mathfrak{I}}_{T_1}, \ldots, \mathcal{D}^{\mathfrak{I}}_{T_n} \mapsto 2^{\mathbb{N} \times \mathcal{D}^{\mathfrak{I}}_{\mathsf{P}}}$, qui à une valuation ψ et un état mémoire m renvoie un élément de $2^{\mathbb{N} \times \mathcal{D}^{\mathfrak{I}}_{\mathsf{P}}}$.

$$
\begin{aligned}
\overline{\mathtt{FT}}_p(x)(f,\psi,m) &= \emptyset \\
\overline{\mathtt{FT}}_p(t.i)(f,\psi,m) &= \overline{\mathtt{FT}}_p(t)(f,\psi,m) \cup \{(i, \mathfrak{I}_{m,\psi}(t))\} \\
\overline{\mathtt{FT}}_p(t_1 \approx t_2)(f,\psi,m) &= \overline{\mathtt{FT}}_p(t_1)(f,\psi,m) \cup \overline{\mathtt{FT}}_p(t_2)(f,\psi,m) \\
\overline{\mathtt{FT}}_p(\phi(\mathbf{t}))(f,\psi,m) &= \bigcup_i \overline{\mathtt{FT}}_p(t_i)(f,\psi,m) \\
\overline{\mathtt{FT}}_p(p'(\mathbf{t}))(f,\psi,m) &= \bigcup_i \overline{\mathtt{FT}}_p(t_i)(f,\psi,m) \cup \begin{cases} f(m, \mathfrak{I}_{m,\psi}(\mathbf{t})) & \text{si } p' = p \\ (\mathtt{ft}_{p'})^{\mathfrak{I}}(m, \mathfrak{I}_{m,\psi}(\mathbf{t})) & \text{sinon} \end{cases}
\end{aligned}
$$

Soit $p : T_1, \ldots, T_n$ défini par $\mathtt{ind}\, p(\mathbf{x}) = C_i : \forall \mathbf{x}, f_{i,1} \Rightarrow \cdots \Rightarrow f_{i,m_i} \Rightarrow p(\mathbf{x})$. On se place dans l'extension point à point du treillis naturel de $2^{\mathbb{N} \times \mathcal{D}^{\mathfrak{I}}_{\mathsf{P}}}$ aux fonctions de type $\mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}, \mathcal{D}^{\mathfrak{I}}_{T_1}, \ldots, \mathcal{D}^{\mathfrak{I}}_{T_n} \mapsto 2^{\mathbb{N} \times \mathcal{D}^{\mathfrak{I}}_{\mathsf{P}}}$. On définit alors l'interprétation de l'empreinte de p par :

$$
(\mathtt{ft}_p)^{\mathfrak{I}} = \mu f.\lambda m, z_1, \ldots, z_n. \bigcup_{\substack{i \text{ tq } \mathfrak{I}_{m,\psi}(f_{i,j}) \text{ est vrai} \\ \text{pour tout } j}} \bigcup_j \overline{\mathtt{FT}}_p(f_{i,j})(f,\psi,m) \tag{3.11}
$$

avec $\psi = [x_1 \mapsto z_1, \ldots, x_n \mapsto z_n]$. Cette formule exprime que l'on fait l'union de l'empreinte de toutes les cas qui sont vérifiés. Par la propriété des cas distincts on sait en réalité qu'un seul au plus des cas est vrai. Donc l'empreinte est soit l'empreinte du cas vrai si le prédicat est vrai sinon l'empreinte est vide.

Exemple : L'empreinte de `islist` est interprétée comme suit, avec $\psi = [p \mapsto r, m \mapsto t]$:

$$
\begin{aligned}
(\mathtt{ft}_{\mathtt{islist}})^{\mathfrak{I}} =& \mu f.\lambda t, r.\sigma_{\mathfrak{I}_{m,\psi}(p \not\approx \mathtt{null} \wedge \mathtt{islist}(m,p))}\,(\{r\} \cup f(t, t(1)(r))) \\
\text{avec } \sigma_l(s) =& \begin{cases} s & \text{si } l \text{ est vrai} \\ \emptyset & \text{sinon} \end{cases}
\end{aligned}
$$

Le prédicat p défini par `pragma` : $p(p_1, p_2)$ avec le prédicat $p_1 : \langle T_{1,1}, \ldots, T_{1,n_1} \rangle$ et le prédicat $p_2 : \langle T_{2,1}, \ldots, T_{2,n_2} \rangle$ est alors interprété pour toute mémoire $m \in \mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}$, pour toutes valeurs $v_{i,j} : \mathcal{D}^{\mathfrak{I}}_{T_{i,j}}$ par

$$
\begin{aligned}
p^{\mathfrak{I}}(m, v_{1,1}, \ldots, v_{1,n_1}, v_{2,1}, \ldots, v_{2,n_2}) \triangleq& \\
& (\mathtt{ft}_{p_1})^{\mathfrak{I}}(m, v_{1,1}, \ldots, v_{1,n_1}) \cap (\mathtt{ft}_{p_2})^{\mathfrak{I}}(m, v_{2,1}, \ldots, v_{2,n_2}) = \emptyset.
\end{aligned}
$$

Ce symbole ainsi interprété est bien une sur-approximation de l'intuition de la séparation décrite plus haut. Par exemple le prédicat inductif `islist'`, défini ci-après, ne possède pas la même empreinte qu'`islist`, alors qu'il lui est équivalent. $(2, val)$ appartient à l'empreinte bien que sa modification ne modifie pas la véracité de `islist'` :

$$
\begin{array}{l}
\texttt{ind}\,\texttt{islist}'(p:\mathsf{P}) = \\
\quad |\mathit{is\_list\_null}: \\
\quad\quad \forall p:\mathsf{P}.\ p \approx \texttt{null} \Rightarrow \texttt{islist}(p) \\
\quad |\mathit{is\_list\_next}: \\
\quad\quad \forall p:\mathsf{P}.\ p.2 \approx p.2 \Rightarrow p \not\approx \texttt{null} \Rightarrow \texttt{islist}(p.1) \Rightarrow \texttt{islist}(p)
\end{array}
$$

Lemme 3.1.1. *Soient Δ une séquence de déclarations bien typées $\vdash \Delta$, un modèle $\mathfrak{I}$ de Δ, un symbole de prédicat inductif $p : T_1, \ldots, T_n$ défini dans Δ, et une mémoire $m \in \mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}$. On a alors pour toutes valeurs $v_i : \mathcal{D}^{\mathfrak{I}}_{T_i}$ que si $p(m, v_1, \ldots, v_n)$ dépend de la valeur sur un champ c du pointeur* `ptr` *alors $(c, \texttt{ptr})$ appartient à $(\texttt{ft}_p)^{\mathfrak{I}}(m, v_1, \ldots, v_n)$.*

Démonstration. On va prouver ce résultat par récurrence sur la liste Δ de déclarations. Soit une liste de déclarations contenant la déclaration d d'un prédicat inductif p. Si la déclaration n'est pas la dernière par l'hypothèse d'induction le résultat est prouvé. Sinon on a $\Delta; d$. Comme Δ est bien typé, par hypothèse d'induction ce résultat est vrai pour tous les symboles qui apparaissent dans la définition de p. Et on a $\Gamma \vdash d$.

Le prédicat inductif p est défini par $\texttt{ind}\, p(\mathbf{x}) = C_i : \forall \mathbf{x}, f_{i,1} \Rightarrow \cdots \Rightarrow f_{i,n_i} \Rightarrow p(\mathbf{x})$. Nous utilisons le schéma d'induction de l'inductif pour prouver le résultat. A l'intérieur de cette induction, nous allons prouver par induction structurelle sur les formules réduites F, tel que $\Gamma, p : \langle T_1, \ldots, T_n \rangle \vdash F$, pour toute valuation ψ, si $\mathfrak{I}_{m,\psi}(F)$ est vrai et $\mathfrak{I}_{m[c,\texttt{ptr}\mapsto v],\psi}(F)$ ne l'est pas alors $(c, \texttt{ptr})$ appartient à $\texttt{FT}^{\mathfrak{I}}_p(F)(\texttt{ft}_p, \psi, m)$.

- Soit un terme $t = x$; il ne peut dépendre de la mémoire, donc la condition du lemme est fausse dans ce cas.
- Soit un terme $t = t'.i$ qui dépend du champ c pour le pointeur `ptr`. Première possibilité, t' dépend lui-même du champ c pour le pointeur `ptr` et $(c, \texttt{ptr})$ appartient à $\texttt{FT}^{\mathfrak{I}}_p(t')(\texttt{ft}_p, \psi, m)$ par hypothèse d'induction et donc par définition $(c, \texttt{ptr})$ appartient à $\texttt{FT}^{\mathfrak{I}}_p(t)(\texttt{ft}_p, \psi, m)$. Deuxième possibilité, $\mathfrak{I}_{m,\psi}(t')$ est égal à $\mathfrak{I}_{m[c,\texttt{ptr}\mapsto v],\psi}(t')$, et i doit donc être égal à c et `ptr` à $\mathfrak{I}_{m,\psi}(t')$. Ainsi par définition $(c, \texttt{ptr})$ appartient à $\texttt{FT}^{\mathfrak{I}}_p(t)(\texttt{ft}_p, \psi, m)$.
- Soit un terme $t = \phi(t_1, \ldots, t_n)$ qui dépend du champ c et du pointeur `ptr`, $\mathfrak{I}_{m,\psi}(t)$ est différent de $\mathfrak{I}_{m[c,q\mapsto v],\psi}(t)$. Puisque p est totalement défini, ϕ ne dépend d'aucun champ. Donc il existe au moins un argument t_i, tel que $\mathfrak{I}_{m,\psi}(t_i)$ est différent de $\mathfrak{I}_{m[c,\texttt{ptr}\mapsto v],\psi}(t_i)$. On en déduit que $\mathfrak{I}_{m,\psi}(t_i)$ dépend du champ c pour le pointeur `ptr`, et ainsi par hypothèse d'induction la paire $(c, \texttt{ptr})$ doit appartenir à $\texttt{FT}^{\mathfrak{I}}_p(t_i)(\texttt{ft}_p, \psi, m)$. Et donc par définition de $\texttt{FT}^{\mathfrak{I}}_p(t)(\texttt{ft}_p, \psi, m)$, la paire $(c, \texttt{ptr})$ doit lui appartenir.
- Soit une formule $t_1 \approx t_2$ ou $\neg t_1 \approx t_2$ qui dépend du champ c pour le pointeur `ptr`, l'un des deux termes t_1 ou t_2 doit au moins dépendre du champ c pour le pointeur `ptr`, et donc par hypothèse de récurrence et par définition $(c, \texttt{ptr})$ appartient à $\texttt{FT}^{\mathfrak{I}}_p(t)(\texttt{ft}_p, \psi, m)$.

– Soit une formule $f = p'(t_1, \ldots, t_n)$ qui dépend du champ c et du pointeur $\texttt{ptr}$, $\mathfrak{I}_{m,\psi}(f)$ est différent de $\mathfrak{I}_{m[c,\texttt{ptr}\mapsto v],\psi}(f)$. Première possibilité pour tous les arguments t_i, $\mathfrak{I}_{m,\psi}(t_i)$ est égale à $\mathfrak{I}_{m[c,\texttt{ptr}\mapsto v],\psi}(t_i)$ et donc $p(\mathfrak{I}_{m,\psi}(t_1), \ldots, \mathfrak{I}_{m,\psi}(t_n))$ doit dépendre du champ c pour le pointeur $\texttt{ptr}$; ainsi par hypothèse d'induction, de la liste de déclarations si $p \neq p'$ et de celle du prédicat inductif p si $p = p'$, la paire (c, p) doit appartenir à $(\texttt{ft}_p)^{\mathfrak{I}}(\mathfrak{I}_{m,\psi}(t_i), \ldots, \mathfrak{I}_{m,\psi}(t_n))$. Enfin par définition de $\texttt{FT}_p^{\mathfrak{I}}(t)(\texttt{ft}_p, \psi, m)$ la paire (c, p) doit lui appartenir. Deuxième possibilité il existe un argument $\mathfrak{I}_{m,\psi}(t_i)$ qui est différent de $\mathfrak{I}_{m[c,\texttt{ptr}\mapsto v],\psi}(t_i)$ et donc $\mathfrak{I}_{m,\psi}(t_i)$ dépend du champ c pour le pointeur $\texttt{ptr}$, et ainsi par hypothèse d'induction sur les formules la paire $(c, \texttt{ptr})$ doit appartenir à $\texttt{FT}_p^{\mathfrak{I}}(t_i)(\texttt{ft}_p, \psi, m)$. Donc par définition de $\texttt{FT}_p^{\mathfrak{I}}(f)(\texttt{ft}_p, \psi, m)$ la paire $(c, \texttt{ptr})$ doit encore lui appartenir.

Ainsi pour toute formule réduite F, telle que $\Gamma, p : \langle T_1, \ldots, T_n\rangle \vdash F$, pour toute valuation ψ, si $\mathfrak{I}_{m,\psi}(F)$ est vrai et $\mathfrak{I}_{m[c,\texttt{ptr}\mapsto v],\psi}(F)$ ne l'est pas alors $(c, \texttt{ptr})$ appartient à $\texttt{FT}_p^{\mathfrak{I}}(F)(\texttt{ft}_p, \psi, m)$.

Maintenant si $p(m, v_1, \ldots, v_n)$ est vrai et $p(m[c, \texttt{ptr} \mapsto v], v_1, \ldots, v_n)$ est faux alors, en posant $\psi' = [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$, par inversion de l'inductif il existe un cas i tel que $I_{m,\psi'}(f_{i,1} \wedge \cdots \wedge f_{i,n_i})$ est vrai. Par ailleurs on sait, par l'absurde, que $I_{m[c,\texttt{ptr}\mapsto v],\psi'}(\forall \mathbf{y_i}.f_{i,1} \wedge \cdots \wedge f_{i,n_i})$ est faux. Donc il existe j tel que $I_{m,\psi'}(f_{i,j})$ est vrai et $I_{m[c,p\mapsto v],\psi'}(f_{i,j})$ est faux, et ainsi $(c, \texttt{ptr})$ appartient à $\texttt{FT}_p^{\mathfrak{I}}(f_{i,j})(\texttt{ft}_p, \psi', m)$.

Finalement on en déduit que $(c, \texttt{ptr})$ appartient à $\texttt{ft}_p(m, v_1, \ldots, v_n)$.

□

Un autre élément important de ces empreintes est la propriété d'auto-encadrement. C'est-à-dire que si $(c, \texttt{ptr})$ n'appartient pas à l'empreinte d'un prédicat p alors l'empreinte de p n'est pas modifiée lorsque l'on modifie le champ c au pointeur $\texttt{ptr}$. On prend un certain état m, où trois instances d'un prédicat p, $p(m, v_1)$, $p(m, v_2)$, $p(m, v_3)$ sont vraies. On suppose également que $p(m, v_1), p(m, v_2), p(m, v_3)$ sont séparés ($sep\_p\_p(m, v_1, v_2)$, $sep\_p\_p(m, v_2, v_3)$, $sep\_p\_p(m, v_3, v_1)$). Maintenant, dans un état m' où $p(m', v_1)$ n'est plus vérifié, on sait par séparation que $p(m', v_2)$ et $p(m', v_3)$ sont encore vérifiés. En revanche on ne sait pas si $p(m', v_2)$ et $p(m', v_3)$ sont encore séparés. En effet on ne sait pas si leurs empreintes ont été modifiées. Cela dépend vraiment de la définition de l'empreinte. Si on reprend la définition précise, et non la sur-approximation, de l'empreinte avec l'inductif suivant :

$$\texttt{ind}\,\texttt{pos}(p : \mathsf{P}, q : \mathsf{P}) = \tag{3.12}$$

$$|def : p.1 * q.1 \geq 0 \Rightarrow pos(p, q) \tag{3.13}$$

Dans un état m tel que $m(\texttt{ptr}_1) = m(\texttt{ptr}_2) = m(\texttt{ptr}_3) = 0$ et $m(\texttt{ptr}_4) > 0$,

– $\texttt{pos}(m, \texttt{ptr}_1, \texttt{ptr}_4)$ ne dépend que de $(1, \texttt{ptr}_1)$,
– $\texttt{pos}(m, \texttt{ptr}_2, \texttt{ptr}_4)$ ne dépend que de $(1, \texttt{ptr}_2)$,
– $\texttt{pos}(m, \texttt{ptr}_1, \texttt{ptr}_2)$ ne dépend de rien (quel que soit le pointeur que l'on modifie au champ 1, $\texttt{ptr}_1.1 * \texttt{ptr}_2.1 = 0$)

Ainsi $\texttt{pos}(m, \texttt{ptr}_1, \texttt{ptr}_4)$, $\texttt{pos}(m, \texttt{ptr}_2, \texttt{ptr}_4)$, $\texttt{pos}(m, \texttt{ptr}_3, \texttt{ptr}_4)$, sont à la fois vrais et séparés deux à deux. En revanche dans l'état $m' \triangleq m[1, \texttt{ptr}_1 \mapsto -1]$, $\texttt{pos}(m, \texttt{ptr}_2, \texttt{ptr}_4)$ et $\texttt{pos}(m, \texttt{ptr}_1, \texttt{ptr}_2)$ dépendent tous deux de $(1, \texttt{ptr}_2)$ et ne sont donc pas séparés.

La sur-approximation quant à elle vérifie bien la propriété d'auto-encadrement.

Lemme 3.1.2. *Soient Δ une séquence de déclarations bien typées $\vdash \Delta$, un modèle $\mathfrak{I}$ de Δ, un symbole de prédicat $p : T_1, \ldots, T_n$ défini dans Δ, une mémoire $m \in \mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}$. Si $p^{\mathfrak{I}}(m, v_1, \ldots, v_n)$ est vrai et si $(c, \texttt{ptr})$ n'appartient pas à $(\texttt{ft}_p)^{\mathfrak{I}}(m, v_1, \ldots, v_n)$, alors pour tout v on a $(\texttt{ft}_p)^{\mathfrak{I}}(m, v_1, \ldots, v_n) = (\texttt{ft}_p)^{\mathfrak{I}}(m[c, \texttt{ptr} \mapsto v], v_1, \ldots, v_n)$.*

Démonstration. On va prouver ce résultat par récurrence sur la liste Δ de déclarations. On va noter la propriété suivante $A(m, v_1, \ldots, v_n)$:

$$\begin{aligned}
&\text{si } (c,p) \notin (\texttt{ft}_p)^{\mathfrak{I}}(m, v_1, \ldots, v_n)\\
&\text{alors } (\texttt{ft}_p)^{\mathfrak{I}}(m, v_1, \ldots, v_n) = (\texttt{ft}_p)^{\mathfrak{I}}(m', v_1, \ldots, v_n)\\
&\text{avec } m' = m[c, \texttt{ptr} \mapsto v]
\end{aligned}$$

On va prouver que pour tout $m, v_1, \ldots, v_n$ si $p(m, v_1, \ldots, v_n)$ est vrai alors la propriété $A(m, v_1, \ldots, v_n)$ l'est aussi en utilisant le schéma d'induction de p.

Ainsi, pour un cas C_i, si $\mathfrak{I}_{m,\psi}(f_{i,j})$ est vrai pour tout j, prouvons que $A(m, v_1, \ldots, v_n)$ est vraie en supposant l'hypothèse d'induction, c'est-à-dire que $A(m, u_1, \ldots, u_n)$ est vraie pour tous les $p(m, u_1, \ldots, u_n)$ qui apparaissent pour tout j dans les $f_{i,j}$. Comme p possède des cas distincts 3.10, pour tout i' différent de i, $\mathfrak{I}_{m,\psi}(f_{i',j})$ est faux pour au moins un j. Ainsi $(\texttt{ft}_p)^{\mathfrak{I}}(m, v_1, \ldots, v_n) = \bigcup_j \overline{\texttt{FT}}_p(f_{i,j})((\texttt{ft}_p)^{\mathfrak{I}}, \psi, m)$. De même $\mathfrak{I}_{m,\psi}(f_{i,j})$ est égale à $\mathfrak{I}_{m',\psi}(f_{i,j})$ puisque $(c, \texttt{ptr}) \notin \overline{\texttt{FT}}_p(f_{i,j})((\texttt{ft}_p)^{\mathfrak{I}}, \psi, m)$ et donc on a ainsi que $(\texttt{ft}_p)^{\mathfrak{I}}(m', v_1, \ldots, v_n) = \bigcup_j \overline{\texttt{FT}}_p(f_{i,j})((\texttt{ft}_p)^{\mathfrak{I}}, \psi, m')$. On va maintenant prouver que $\overline{\texttt{FT}}_p(f_{i,j})((\texttt{ft}_p)^{\mathfrak{I}}, \psi, m) = \overline{\texttt{FT}}_p(f_{i,j})((\texttt{ft}_p)^{\mathfrak{I}}, \psi, m')$ en déconstruisant les $f_{i,j}$ (il n'y a qu'un ensemble fini et connu de sous-termes). La seule difficulté se situe lors de l'application de prédicats. Soit la formule $F = p'(t_1, \ldots, t_k, \ldots, t_{n'})$ appartenant à $f_{i,j}$ pour un certain j. On a $(c, \texttt{ptr}) \notin \overline{\texttt{FT}}_p(F)((\texttt{ft}_p)^{\mathfrak{I}}, \psi, m)$. Puisque $(c, \texttt{ptr}) \notin \overline{\texttt{FT}}_p(t_k)((\texttt{ft}_p)^{\mathfrak{I}}, \psi, m)$ on a $\mathfrak{I}_{m,\psi}(t_k) = \mathfrak{I}_{m',\psi}(t_k)$. De plus, puisque $(c, \texttt{ptr}) \notin (\texttt{ft}_{p'})^{\mathfrak{I}}(m, \mathfrak{I}_{m,\psi}(\mathbf{t}))$, si p' est différent de p on a par hypothèse de récurrence sur la liste de déclarations que $(\texttt{ft}_{p'})^{\mathfrak{I}}(m, \mathfrak{I}_{m,\psi}(\mathbf{t})) = (\texttt{ft}_{p'})^{\mathfrak{I}}(m', \mathfrak{I}_{m',\psi}(\mathbf{t}))$. À l'inverse, si p' est égal à p, on a la même égalité par hypothèse d'induction de p. Enfin on prouve $\overline{\texttt{FT}}_p(t_k)((\texttt{ft}_p)^{\mathfrak{I}}, \psi, m) = \overline{\texttt{FT}}_p(t_k)((\texttt{ft}_p)^{\mathfrak{I}}, \psi, m')$ en continuant l'opération de déconstruction. Ainsi :

$$\begin{aligned}
\overline{\texttt{FT}}_p(p'(\mathbf{t}))((\texttt{ft}_p)^{\mathfrak{I}}, \psi, m) &= \bigcup_i \overline{\texttt{FT}}_p(t_i)((\texttt{ft}_p)^{\mathfrak{I}}, \psi, m) \cup \begin{cases} (\texttt{ft}_{p'})^{\mathfrak{I}}(m, \mathfrak{I}_{m,\psi}(\mathbf{t})) & \text{si } p' = p \\ (\texttt{ft}_{p'})^{\mathfrak{I}}(m, \mathfrak{I}_{m,\psi}(\mathbf{t})) \end{cases}\\
&= \bigcup_i \overline{\texttt{FT}}_p(t_i)((\texttt{ft}_p)^{\mathfrak{I}}, \psi, m') \cup \begin{cases} (\texttt{ft}_{p'})^{\mathfrak{I}}(m', \mathfrak{I}_{m',\psi}(\mathbf{t})) & \text{si } p' = p \\ (\texttt{ft}_{p'})^{\mathfrak{I}}(m', \mathfrak{I}_{m',\psi}(\mathbf{t})) \end{cases}\\
&= \overline{\texttt{FT}}_p(p'(\mathbf{t}))((\texttt{ft}_p)^{\mathfrak{I}}, \psi, m').
\end{aligned}$$

□

On pourrait également permettre de définir mutuellement un prédicat inductif et un pragma y faisant référence. Cela est utile si l'on veux définir les arbres en excluant le

partage entre sous-arbres. Cela demande, cependant, de travailler avec des définitions mutuellement inductives ce qui complique beaucoup la formalisation.

$$\texttt{pragma}\,\texttt{sep\_tree\_tree}(\texttt{istree}, \texttt{istree})$$

$$\begin{aligned}
&\texttt{ind}\,\texttt{istree}(p : \mathsf{P}) = \\
&|\mathit{is_tree_null} : \\
&\quad \forall p : \mathsf{P}.\ p = \texttt{null} \Rightarrow \texttt{istree}(p) \\
&|\mathit{is_list_next} : \\
&\quad \forall p : \mathsf{P}.\ \neg p = \texttt{null} \Rightarrow \texttt{istree}(p.1) \\
&\quad \Rightarrow \texttt{istree}(p.2) \Rightarrow \texttt{sep_tree_tree}(p.1, p.2) \Rightarrow \texttt{islist}(p)
\end{aligned}$$

Sémantique des spécifications Les spécifications sont des formules qui dépendent d'un état mémoire global. On les utilise pour définir l'ensemble des états mémoires qui les satisfont et ainsi on peut définir la sémantique d'un programme et définir si un programme suit sa spécification. Par exemple si on exécute un programme en partant d'un état mémoire qui satisfait sa précondition on atteint un état mémoire qui satisfait sa postcondition. On va donc définir ce que signifie pour un état mémoire de satisfaire une formule dans un certain contexte. Soient une suite Δ de déclarations, une valuation ψ, une formule F et une mémoire globale m. On dit que m avec la valuation ψ vérifie la formule F dans le contexte Δ si quelle que soit l'interprétation $\mathfrak{M}$ qui satisfait Δ l'interprétation de F, $\mathfrak{I}_{m,\psi}(F)$, est vraie. On le note $\Delta \models F, m, \psi$. La valuation ψ peut être ici utilisée pour modéliser la valeur des paramètres d'une fonction. Par exemple si l'on reprend la précondition du programme de la figure 2.1 il sera écrit dans notre syntaxe : $F = \texttt{islist}(p)$. Cette formule est vérifiée dans tout contexte Δ par toutes valuation ψ et mémoire m telles qu'il existe i avec $m^i(\psi(p)) = \texttt{null}^{\mathfrak{M}}$ et $\mathfrak{M}$ un modèle de Δ. m^i est la composition de i fois la fonction m.

3.1.3. Le langage d'arrivée

Le langage de sortie ressemble à celui d'entrée, excepté que l'état mémoire est devenu explicite ; les champs dont dépend une fonction ou un prédicat apparaissent en tant qu'arguments. La syntaxe de sortie est définie dans la figure 3.5. Dans le langage de sortie il n'y a pas de restriction des formules qui peuvent apparaître en définition. Les règles de typage sont similaires à celles du langage d'entrée. Seule la règle pour le déréférencement est enlevée.

Comme les champs deviennent explicites, on les traduit par des tableaux des pointeurs vers les entiers, de type $\mathsf{M_I}$ ou vers les pointeurs, de type $\mathsf{M_P}$; on les appellera des mémoires par champ. On n'utilise pas ici le polymorphisme pour simplifier le système

$$
\begin{aligned}
T &::= \mathsf{I} \mid \mathsf{P} \mid \mathsf{M}_\mathsf{I} \mid \mathsf{M}_\mathsf{P} \\
t &::= x \mid \phi(t, \ldots, t) \\
f &::= t = t \mid p(\mathbf{x}) \mid \neg f \mid \forall y.f \mid f \wedge f \\
c &::= c, \mathtt{C} : \forall \mathbf{x}.f \Rightarrow \cdots \Rightarrow f \Rightarrow p(\mathbf{x}) \mid \\
d &::= \mathtt{fun}\, \phi(\mathbf{x}) : T = t \\
&\quad \mid \mathtt{pred}\, p(\mathbf{x}) = f \\
&\quad \mid \mathtt{fun}\, \phi(\mathbf{x}) : T \\
&\quad \mid \mathtt{pred}\, p(\mathbf{x}) \\
&\quad \mid \mathtt{ind}\, p(\mathbf{x}) = c \\
&\quad \mid \mathtt{axiom} : f
\end{aligned}
$$

FIGURE 3.5.: Syntaxe de mini-`Why`.

de type de cette partie. On utilisera cependant des symboles `get` et `set` surchargés. L'accès dans une table d'association de type M_T est effectué par $\mathtt{get} : \langle \mathsf{M}_T, \mathsf{P}, T \rangle$ et la modification est effectuée par $\mathtt{set} : \langle \mathsf{M}_T, \mathsf{P}, T, \mathsf{M}_T \rangle$. Le champ i de type τ_i est traduit par un terme de type M_{τ_i}. Cette transformation est une simplification de celle utilisé par Jessie [43]. La traduction d'un problème Δ sans pragmas dans la logique d'arrivée est réalisée inductivement par la transformation $\mathcal{T}$ présentée à la figure 3.6. Dans le plan d'ensemble de la figure 3.1 c'est ainsi que l'inductif `islist` 3.1 est transformé. On ajoute aux fonctions et prédicats autant d'arguments que de champs dont ils dépendent. Par exemple si $\mathtt{MEM}(p) = \{i_1, \ldots, i_n\}$ avec les $i_1 \ldots, i_n$ dans l'ordre croissant, $p(x_1, \ldots, x_m)$ devient $p(m_{i_1}, \ldots, m_{i_n}, x_1, \ldots, x_m)$. Par clarté, on notera dans ce cas $\mathbf{m}_{\mathtt{MEM}(p)}$ pour $m_{i_1} : \tau_{i_1}, \ldots, m_{i_n} : \tau_{i_n}$ et donc $p(x_1, \ldots, x_n)$ devient alors $p(\mathbf{m}_{\mathtt{MEM}(p)}, x_1, \ldots, x_n)$.

L'inductif `islist` est alors transformé en :

$$
\begin{aligned}
&\mathtt{ind}\, \mathtt{islist}(m_1 : \mathsf{M}_\mathsf{P}, p : \mathsf{P}) = \\
&\quad \mid \mathit{islist\_null} : \\
&\qquad \forall m_1 : \mathsf{M}_\mathsf{P}. \forall p : \mathsf{P}.\ p \approx \mathtt{null} \Rightarrow \mathtt{islist}(m_1, p) \\
&\quad \mid \mathit{islist\_next} : \\
&\qquad \forall m_1 : \mathsf{M}_\mathsf{P}. \forall p : \mathsf{P}.\ p \not\approx \mathtt{null} \Rightarrow \mathtt{islist}(m_1, \mathtt{get}(m_1, p)) \Rightarrow \mathtt{islist}(m_1, p)
\end{aligned}
$$

Interprétation L'interprétation du langage d'arrivée est plus classique que celle du langage d'entrée. En effet puisque la mémoire devient explicite elle est traitée de la même manière que des paramètres de fonctions. L'interprétation ne dépend alors que de la valuation ψ. Les types M_P et M_I ont pour domaine des tables d'association à valeurs dans

$$\mathcal{T}(t_1 = t_2) \triangleq \mathcal{T}(t_1) = \mathcal{T}(t_2) \qquad \mathcal{T}(f_1 \wedge f_2) \triangleq \mathcal{T}(f_1) \wedge \mathcal{T}(f_2) \qquad \mathcal{T}(\neg f) \triangleq \neg\mathcal{T}(f)$$

$$\mathcal{T}(x) \triangleq x \qquad \mathcal{T}(t.i) \triangleq \mathtt{get}(m_i, \mathcal{T}(t)) \qquad \mathcal{T}(\forall x : T.f) \triangleq \forall x : T.\mathcal{T}(f)$$

$$\mathtt{axiom} : f \triangleq \mathtt{axiom} : \forall \mathbf{m}_{\mathtt{MEM}(f)}.\mathcal{T}(f)$$

$$\mathcal{T}(\phi(t_1, \ldots, t_n)) \triangleq \phi(\mathbf{m}_{\mathtt{MEM}(\phi)}, \mathcal{T}(t_1), \ldots, \mathcal{T}(t_n))$$

$$\mathcal{T}(p(t_1, \ldots, t_n)) \triangleq p(\mathbf{m}_{\mathtt{MEM}(p)}, \mathcal{T}(t_1), \ldots, \mathcal{T}(t_n))$$

$$\mathcal{T}(\mathtt{fun}\,\phi(\mathbf{x}) : T = t) \triangleq \mathtt{fun}\,\phi(\mathbf{m}_{\mathtt{MEM}(\phi)}, \mathbf{x}) : T = \mathcal{T}(t)$$

$$\mathcal{T}(\mathtt{pred}\,p(\mathbf{x}) = f) \triangleq \mathtt{pred}\,p(\mathbf{m}_{\mathtt{MEM}(p)}, \mathbf{x}) = \mathcal{T}(f)$$

$$\mathcal{T}(\mathtt{fun}\,\phi(\mathbf{x}) : T\{i_1, \ldots, i_n\}) \triangleq \mathtt{fun}\,\phi(\mathbf{m}_{\mathtt{MEM}(\phi)}, \mathbf{x}) : T$$

$$\mathcal{T}(\mathtt{pred}\,p(\mathbf{x}) : T\{i_1, \ldots, i_n\}) \triangleq \mathtt{pred}\,p(\mathbf{m}_{\mathtt{MEM}(p)}, \mathbf{x}) : T$$

$$\mathcal{T}(\mathtt{ind}\,p(\mathbf{x}) = \mathtt{C_i} : \forall \mathbf{x}.f_{i,1} \Rightarrow \cdots \Rightarrow f_{i,n} \Rightarrow p(\mathbf{x})) \triangleq$$
$$\mathtt{ind}\,p(\mathbf{m}_{\mathtt{MEM}(p)}, \mathbf{x}) = \mathtt{C_i} : \forall \mathbf{m}_{\mathtt{MEM}(p)}.\forall \mathbf{x}.\mathcal{T}(f_{i,1}) \Rightarrow \cdots \Rightarrow \mathcal{T}(f_{i,n}) \Rightarrow p(\mathbf{m}_{\mathtt{MEM}(p)}, \mathbf{x})$$

FIGURE 3.6.: Traduction des termes, formules, axiomes et déclarations de fonctions ou de prédicats.

le domaine des pointeurs et le domaine des entiers respectivement. `get` est interprété comme l'accès et `set` comme la modification de ces tables d'association. Maintenant les interprétations du langage de départ et d'arrivée sont définies et la traduction entre ces deux langages est définie sur les déclarations qui ne sont pas des pragmas. On peut prouver que cette transformation est correcte et complète sur les séquences de déclarations qui ne contiennent pas de pragma . Avant de prouver ce résultat, nous allons définir des transformations entre une mémoire globale et des mémoires par champs. Soit une mémoire globale $m \in \mathcal{D}^{\mathfrak{M}}_{\mathsf{M}}$. On définit $\zeta_i(m)$ une table d'association de type $\mathfrak{I}^{\mathfrak{M}}_{\mathsf{P}} \mapsto \mathcal{D}^{\mathfrak{I}}_{\tau_i}$ qui est l'application partielle de m aux numéros du champ : $\zeta_i(m) \triangleq m(i)$. On dérive de cette définition différentes notations pour une mémoire globale m et une valuation ψ :

$$\begin{aligned}
\zeta(m,\psi) &\triangleq \psi[m_1 \mapsto \zeta_1(m), \ldots, m_n \mapsto \zeta_n(m), \ldots] \\
\zeta_{\mathtt{MEM}(f)}(m) &\triangleq \zeta_{\sigma_1}(m), \ldots, \zeta_{\sigma_n}(m) \quad \text{quand } \mathtt{MEM}(f) = \sigma_1, \ldots, \sigma_n \\
\zeta_{\mathtt{MEM}(f)}(m,\psi) &\triangleq \psi[m_{\sigma_1} \mapsto \zeta(m)_{\sigma_1}, \ldots, m_{\sigma_n} \mapsto \zeta(m)_{\sigma_n}, \ldots]
\end{aligned}$$

On définit inversement une mémoire globale à partir d'une valuation du langage d'arrivée. Soit une valuation ψ telle que $\psi(m_i) : \mathfrak{I}^{\mathfrak{M}}_{\mathsf{P}} \mapsto \mathcal{D}^{\mathfrak{I}}_{\tau_i}$ pour tout champ i, on définit ζ^{-1} par :

$$\begin{aligned}
\zeta^{-1}(\psi)(i) &\triangleq \psi(m_i) \\
\zeta^{-1}(\mathbf{t}_{\mathtt{MEM}(f)})(i) &\triangleq \begin{cases} t_i & \text{si } i \in \mathtt{MEM}(f) \\ \lambda p.c_i & \text{sinon} \end{cases},
\end{aligned}$$

où la constante c_i est arbitraire mais fixée et est de type τ_i. $\zeta^{-1}(\psi)$ est bien du type d'une mémoire globale. Soient une suite Δ de déclarations de `mini-Why`, une formule F et une valuation ψ. De même que l'on a défini dans le langage de départ $\Delta \models F, m, \psi$, on définit $\Delta \models F, \psi$ par quel que soit le modèle $\mathfrak{I}$ de `mini-Why` qui vérifie Δ on a $\mathfrak{I}_\psi(F)$ vrai.

Lemme 3.1.3. *Si une liste Δ de déclarations sans pragma est satisfiable alors $\mathcal{T}(\Delta)$ est satisfiable.*

Démonstration. On va prouver que si Δ possède un modèle $\mathfrak{M}$ alors il existe un modèle $\mathfrak{I}$ tel que pour toute valuation ψ qui vérifie que $\psi(m_i) : \mathcal{D}^{\mathfrak{I}}_{\mathsf{P}} \mapsto \mathcal{D}^{\mathfrak{I}}_{\tau_i}$ pour tout champ i, pour tout terme t ou toute formule f, on a :

$$\mathfrak{I}_\psi(\mathcal{T}(t)) = \mathfrak{M}_{\zeta^{-1}(\psi),\psi}(t) \qquad \mathfrak{I}_\psi(\mathcal{T}(f)) = \mathfrak{M}_{\zeta^{-1}(\psi),\psi}(f) \tag{3.14}$$

Le modèle $\mathfrak{I}$ est défini à partir du modèle $\mathfrak{M}$. Le domaine des entiers $\mathcal{D}^{\mathfrak{I}}_{\mathsf{I}}$ est $\mathcal{D}^{\mathfrak{M}}_{\mathsf{I}}$ et $\mathcal{D}^{\mathfrak{I}}_{\mathsf{P}}$ est $\mathcal{D}^{\mathfrak{M}}_{\mathsf{P}}$.

Pour toute fonction $f : \langle T_1, \ldots, T_n, T\rangle$, pour toute mémoire partielle $t_i : \mathcal{D}^{\mathfrak{I}}_{\mathsf{P}} \mapsto \mathcal{D}^{\mathfrak{I}}_{\tau_i}$ avec i appartenant à $\mathtt{MEM}(f)$, pour toutes valeurs v_i appartenant à $\mathcal{D}^{\mathfrak{I}}_{T_i}$, on définit $f^{\mathfrak{I}}(\mathbf{t}_{\mathtt{MEM}(f)}, v_1, \ldots, v_n)$ par $f^{\mathfrak{M}}(\zeta^{-1}(\mathbf{t}_{\mathtt{MEM}(f)}), v_1, \ldots, v_n)$. On définit de même l'interprétation des prédicats.

Prouvons maintenant les deux égalités par induction sur les termes et formules :

– Pour une variable x :

$$\begin{aligned}\mathfrak{I}_\psi(\mathcal{T}(x)) =& \psi(x)\\ =& \mathfrak{M}_{\zeta^{-1}(\psi),\psi}(x)\end{aligned}$$

– Pour une application de fonction $\phi(t_1,\ldots,t_n)$, on remarque que $\zeta^{-1}(\psi(\mathbf{m}_{\texttt{MEM}(\phi)}))$ coïncide avec $\zeta^{-1}(\psi)$ sur les champs de $\texttt{MEM}(\phi)$ et on a donc :

$$\begin{aligned}\mathfrak{I}_\psi(\mathcal{T}(\phi(t_1,\ldots,t_n))) =& \mathfrak{I}_\psi(\phi(\mathbf{m}_{\texttt{MEM}(\phi)},\mathcal{T}(t_1),\ldots,\mathcal{T}(t_n)))\\ =& \phi^{\mathfrak{I}}(\mathfrak{I}_\psi(\mathbf{m}_{\texttt{MEM}(\phi)}),\mathfrak{I}_\psi(\mathcal{T}(t_1)),\ldots,\mathfrak{I}_\psi(\mathcal{T}(t_n)))\\ =& \phi^{\mathfrak{I}}(\psi(\mathbf{m}_{\texttt{MEM}(\phi)}),\mathfrak{I}_\psi(\mathcal{T}(t_1)),\ldots,\mathfrak{I}_\psi(\mathcal{T}(t_n)))\\ =& \phi^{\mathfrak{M}}(\zeta^{-1}(\psi(\mathbf{m}_{\texttt{MEM}(\phi)})),\mathfrak{M}_{\zeta^{-1}(\psi),\psi}(t_1),\ldots,\mathfrak{M}_{\zeta^{-1}(\psi),\psi}(t_n))\\ =& \mathfrak{M}_{\zeta^{-1}(\psi),\psi}(\phi(t_1,\ldots,t_n))\end{aligned}$$

– Pour un déréférencement $t = t'.i$:

$$\begin{aligned}\mathfrak{I}_\psi(\mathcal{T}(t'.i)) =& \mathfrak{I}_\psi(\texttt{get}(m_i,t'))\\ =& \psi(m_i)(\mathfrak{I}_\psi(t'))\\ =& \zeta^{-1}(\psi)(i)(\mathfrak{M}_{\zeta^{-1}(\psi),\psi}(t'))\\ =& \mathfrak{M}_{\zeta^{-1}(\psi),\psi}(t'.i)\end{aligned}$$

– Pour une égalité $F = t_1 \approx t_2$, l'égalité découle directement de l'hypothèse de récurrence.
– Pour une quantification universelle $\forall x : T.f$, puisque x ne peut pas être l'un des m_i, $\zeta^{-1}(\psi[x \mapsto v]) = \zeta^{-1}(\psi)$:

$$\begin{aligned}\mathfrak{I}_\psi(\mathcal{T}(\forall x : T.f)) &= \mathfrak{I}_\psi(\forall x : T.\mathcal{T}(f))\\ &= \bigwedge_{v\in\mathcal{D}_T^{\mathfrak{I}}} \mathfrak{I}_{\psi[x\mapsto v]}(f)\\ &= \bigwedge_{v\in\mathcal{D}_T^{\mathfrak{I}}} \mathfrak{M}_{\zeta^{-1}(\psi[x\mapsto v]),\psi[x\mapsto v]}(f)\\ &= \mathfrak{M}_{\zeta^{-1}(\psi),\psi}(\forall x : T.f)\end{aligned}$$

Soit un axiome $\texttt{axiom} : f$. $\mathfrak{I}$ satisfait cet axiome transformé si $\mathfrak{I}_\psi(\forall \mathbf{m}_{\texttt{MEM}(f)}.\mathcal{T}(f))$ est vrai pour toute valuation ψ, c'est-à-dire si $\mathfrak{I}_{\psi'}(\mathcal{T}(f))$ est vrai pour tout ψ' tel que $\psi(m_i) : \mathfrak{I}_{\mathsf{P}}^{\mathfrak{M}} \mapsto \mathcal{D}_{\tau_i}^{\mathfrak{I}}$ pour tout champ i de $\texttt{MEM}(f)$. L'interprétation de $\mathcal{T}(f)$ ne dépend pas de la valuation pour les champs qui ne sont pas dans $\texttt{MEM}(f)$. Donc on peut modifier la valuation pour que cela soit pour tout champ i que $\psi(m_i) : \mathfrak{I}_{\mathsf{P}}^{\mathfrak{M}} \mapsto \mathcal{D}_{\tau_i}^{\mathfrak{I}}$.Grâce aux égalités prouvées précédemment cela revient à $\mathfrak{M}_{\zeta^{-1}(\psi'),\psi'}(f)$ qui est vrai comme $\mathfrak{M}$ satisfait $\texttt{axiom} : f$. On prouve de même pour les symboles de fonctions et de prédicats.

Soit un inductif $\texttt{ind}\, p(x_1 : T_1, \ldots, x_n : T_n) = \texttt{C}_\texttt{i} : \forall \mathbf{x}. f_{i,1} \Rightarrow \cdots \Rightarrow f_{i,n} \Rightarrow p(\mathbf{x})$. On pose ψ un raccourci pour $[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n, \mathbf{m}_{\texttt{MEM}(p)} \mapsto \mathbf{t}_{\texttt{MEM}(p)}]$ et on a

$$
\begin{aligned}
&p^{\mathfrak{I}}(\mathbf{t}_{\texttt{MEM}(p)}, v_1, \ldots, v_n) \\
&\quad = \left(\mu p^{\mathfrak{I}}.\lambda \mathbf{t}_{\texttt{MEM}(p)}, v_1, \ldots, v_n. \bigvee_i \mathfrak{I}_\psi(\mathcal{T}(f_{i,1})) \wedge \cdots \wedge \mathfrak{I}_\psi(\mathcal{T}(f_{i,n_i}))\right) \\
&\quad\quad (\mathbf{t}_{\texttt{MEM}(p)}, v_1, \ldots, v_n) \\
&\quad = \left(\mu p^{\mathfrak{M}}.\lambda \mathbf{t}_{\texttt{MEM}(p)}, v_1, \ldots, v_n. \bigvee_i \mathfrak{M}_{\zeta^{-1}(\psi),\psi}(f_{i,1}) \wedge \cdots \wedge \mathfrak{M}_{\zeta^{-1}(\psi),\psi}(f_{i,n_i})\right) \\
&\quad\quad (\mathbf{t}_{\texttt{MEM}(p)}, v_1, \ldots, v_n) \\
&\quad = \left(\mu p^{\mathfrak{M}}.\lambda \mathbf{t}_{\texttt{MEM}(p)}, v_1, \ldots, v_n. \bigvee_i \mathfrak{M}_{\zeta^{-1}(\mathbf{t}_{\texttt{MEM}(p)}),\psi}(f_{i,1}) \wedge \cdots \wedge \mathfrak{M}_{\zeta^{-1}(\mathbf{t}_{\texttt{MEM}(p)}),\psi}(f_{i,n_i})\right) \\
&\quad\quad (\mathbf{t}_{\texttt{MEM}(p)}, v_1, \ldots, v_n) \\
&\quad = \left(\mu p^{\mathfrak{M}}.\lambda m, v_1, \ldots, v_n. \bigvee_i \mathfrak{M}_{m,\psi}(f_{i,1}) \wedge \cdots \wedge \mathfrak{M}_{m,\psi}(f_{i,n_i})\right) \\
&\quad\quad (\zeta^{-1}(\mathbf{t}_{\texttt{MEM}(p)}), v_1, \ldots, v_n) \\
&\quad = p^{\mathfrak{M}}(\zeta^{-1}(\mathbf{t}_{\texttt{MEM}(p)}), v_1, \ldots, v_n)
\end{aligned}
$$

L'avant dernière égalité est obtenue car on peut intervertir l'opérateur de plus petit point fixe et le premier lambda puisque pour chaque application de $p^{\mathfrak{M}}$, $\mathbf{t}_{\texttt{MEM}(p)}$ est le premier argument. Ensuite on peut substituer $\zeta^{-1}(\mathbf{t}_{\texttt{MEM}(p)})$ par m. Ainsi on obtient : $\lambda \mathbf{t}_{\texttt{MEM}(p)}.(\lambda m(\mu p^{\mathfrak{M}} \ldots))\zeta^{-1}(\mathbf{t}_{\texttt{MEM}(p)})$, et comme $\mathbf{t}_{\texttt{MEM}(p)}$ n'apparaissent plus dans le point fixe, une η-conversion suivie d'une nouvelle inversion entre un lambda et un point-fixe nous donne l'égalité attendue. □

Lemme 3.1.4. *Soit une liste Δ de déclarations sans pragma, si $\mathcal{T}(\Delta)$ est satisfiable alors Δ l'est également.*

Démonstration. On va prouver que si $\mathcal{T}(\Delta)$ possède un modèle $\mathfrak{M}$ alors il existe un modèle $\mathfrak{I}$ tel que pour toute mémoire m et pour toute valuation ψ :

$$\mathfrak{I}_{m,\psi}(t) = \mathfrak{M}_{\zeta(m,\psi)}(\mathcal{T}(t)) \tag{3.15}$$

$$\mathfrak{I}_{m,\psi}(f) = \mathfrak{M}_{\zeta(m,\psi)}(\mathcal{T}(f)) \tag{3.16}$$

pour tout terme t et toute formule f.

Le modèle $\mathfrak{I}$ est défini à partir du modèle $\mathfrak{M}$. Le domaine des entiers $\mathcal{D}_{\mathsf{I}}^{\mathfrak{I}}$ est $\mathcal{D}_{\mathsf{I}}^{\mathfrak{M}}$ et $\mathcal{D}_{\mathsf{P}}^{\mathfrak{I}}$ est $\mathcal{D}_{\mathsf{P}}^{\mathfrak{M}}$.

Pour toutes fonctions $\phi : \langle T_1, \ldots, T_n, T\rangle$, pour toute mémoire globale $t : \mathbb{N} \mapsto \mathcal{D}_{\mathsf{P}}^{\mathfrak{I}} \mapsto \mathcal{D}_{\tau_i}^{\mathfrak{I}}$, pour toutes valeurs $v_i \in \mathcal{D}_{T_i}^{\mathfrak{I}}$, on définit $\phi^{\mathfrak{I}}(t, v_1, \ldots, v_n) \triangleq \phi^{\mathfrak{M}}(\zeta_{\texttt{MEM}(f)}(t), v_1, \ldots, v_n)$.

Cette définition implique bien que $\phi^{\mathfrak{I}}$ ne dépend que des champs de $\texttt{MEM}(\phi)$ puisque $\zeta_{\texttt{MEM}(\phi)}(t)$ est la projection sur ces derniers. On définit de même l'interprétation des prédicats.

Maintenant prouvons les égalités par induction sur les termes et formules :

– Pour une variable x :

$$\begin{aligned}\mathfrak{I}_{m,\psi}(x) =& \psi(x)\\ =& \zeta(m,\psi)(x)\\ =& \mathfrak{M}_{\zeta(m,\psi)}(\mathcal{T}(x))\end{aligned}$$

– Pour une application de fonction $\phi(t_1,\ldots,t_n)$:

$$\begin{aligned}\mathfrak{I}_{m,\psi}(\phi(t_1,\ldots,t_n)) =& \phi^{\mathfrak{I}}(m,\mathfrak{I}_{m,\psi}(t_1),\ldots,\mathfrak{I}_{m,\psi}(t_n))\\ =& \phi^{\mathfrak{M}}(\zeta_{\texttt{MEM}(\phi)}(m),\mathfrak{I}_{m,\psi}(t_1),\ldots,\mathfrak{I}_{m,\psi}(t_n))\\ =& \phi^{\mathfrak{M}}(\zeta(m,\psi)(\mathbf{m}_{\texttt{MEM}(\phi)}),\mathfrak{M}_{\zeta(m,\psi)}(\mathcal{T}(t_1)),\ldots,\mathfrak{M}_{\zeta(m,\psi)}(\mathcal{T}(t_n)))\\ =& \mathfrak{M}_{\zeta(m,\psi)}(\phi(\mathbf{m}_{\texttt{MEM}(\phi)},\mathcal{T}(t_1),\ldots,\mathcal{T}(t_n)))\\ =& \mathfrak{M}_{\zeta(m,\psi)}(\mathcal{T}(\phi(t_1,\ldots,t_n)))\end{aligned}$$

– Pour un déréférencement $t = t'.i$:

$$\begin{aligned}\mathfrak{I}_{m,\psi}(t'.i) =& m(i)(\mathfrak{I}_{m,\psi}(t'))\\ =& \zeta(m,\psi)(m_i)(\mathfrak{M}_{\zeta(m,\psi)}(t'))\\ =& \mathfrak{M}_{\zeta(m,\psi)}(\texttt{get}(m_i,t'))\\ =& \mathfrak{M}_{\zeta(m,\psi)}(\mathcal{T}(t'.i))\end{aligned}$$

– Pour une égalité $F = t_1 \approx t_2$ cela découle directement de l'hypothèse de récurrence.

– Pour une quantification universelle $\forall x : T.f$, puisque x ne peut pas être l'un des m_i, $\zeta(m,\psi[x\mapsto v]) = \zeta(m,\psi)[x\mapsto v]$:

$$\begin{aligned}\mathfrak{I}_{m,\psi}(\forall x : T.f) =& \bigwedge_{v\in\mathcal{D}_T^{\mathfrak{I}}} \mathfrak{I}_{m,\psi[x\mapsto v]}(f)\\ =& \bigwedge_{v\in\mathcal{D}_T^{\mathfrak{I}}} \mathfrak{M}_{\zeta(m,\psi[x\mapsto v])}(f)\\ =& \bigwedge_{v\in\mathcal{D}_T^{\mathfrak{I}}} \mathfrak{M}_{\zeta(m,\psi)[x\mapsto v]}(f)\\ =& \mathfrak{M}_{\zeta(m,\psi)}(\mathcal{T}\forall x : T.f)\end{aligned}$$

Le reste de la preuve est identique à la fin de la preuve du lemme 3.1.3

□

Lemme 3.1.5. *Soit une liste Δ de déclarations sans pragma, une mémoire m globale, une formule f, une valuation ψ, on a $\Delta \models f, m, \psi$ si et seulement si $\mathcal{T}(\Delta) \models \mathcal{T}(f), \zeta(m,\psi)$.*

Démonstration. En utilisant les égalités des interprétations prouvées dans les deux précédents lemmes on prouve aisément l'affirmation. □

La traduction des déclarations et la génération des prédicats de séparation sont décrites dans la partie suivante.

3.2. Prédicats de séparation et empreintes de prédicats

On va pour chaque prédicat inductif qui apparaît dans un pragma, axiomatiser son empreinte et ajouter la propriété d'encadrement et d'auto-encadrement de l'empreinte en tant qu'axiome.

Soit p un prédicat inductif déclaré par :

$$\texttt{ind}\, p(\mathbf{m}, \mathbf{x}) = \mathtt{C_i} : \forall \mathbf{x}. f_{i,1} \Rightarrow \cdots \Rightarrow f_{i,n_i} \Rightarrow p(\mathbf{x}).$$

Soit un champ $c \in \mathtt{MEM}(p)$ dont dépend p. On définit le symbole de fonction $\mathtt{ft}_p^c(\mathbf{m}, \mathbf{x}) : ft$. ft est le type des empreintes (3.5 dans le plan d'ensemble). Le prédicat d'appartenance à une empreinte est noté $\in\!: \mathsf{P}, ft$. Pour définir l'axiomatique de l'empreinte on définit $\mathtt{FT}_{c,q}$ qui, à partir d'un littéral ou d'un terme renvoie une formule qui indique si le pointeur q appartient à l'empreinte du littéral ou du terme par rapport au champ c.

$$\begin{aligned}
\mathtt{FT}_{c,q}(x) &= \bot \\
\mathtt{FT}_{c,q}(t.j) &= \mathtt{FT}_{c,q}(t) \vee \begin{cases} q \approx t & \text{if } c = j \\ \bot & \text{sinon} \end{cases} \\
\mathtt{FT}_{c,q}(t_1 = t_2) &= \mathtt{FT}_{c,q}(t_1) \vee \mathtt{FT}_{c,q}(t_2) \\
\mathtt{FT}_{c,q}(p(\mathbf{m}, \mathbf{t})) &= \bigvee_j \mathtt{FT}_{c,q}(t_j) \vee \begin{cases} q \in \mathtt{ft}_p^c(\mathbf{m}, \mathbf{t}) & \text{si } c \in \mathtt{MEM}(p) \\ \bot & \text{sinon} \end{cases}
\end{aligned}$$

Et l'axiome est défini par :

$$\forall q. \forall \mathbf{m}. \forall \mathbf{x}.\, q \in \mathtt{ft}_p^c(\mathbf{m}, \mathbf{t}) \Leftrightarrow \bigvee_i \left(\bigwedge_j f_{i,j} \wedge \bigvee_j \mathtt{FT}_{c,q}(f_{i,j}) \right). \tag{3.17}$$

Pour chaque cas C_i et C_j, on traduit l'exclusion de ces cas de l'inductif p par l'axiome

$$\forall \mathbf{x}.\, f_{i,1} \wedge f_{i,2} \wedge \cdots \wedge f_{j,1} \wedge f_{j,2} \wedge \cdots \Rightarrow \bot. \tag{3.18}$$

L'ensemble de ces axiomes pour l'inductif p, c'est-à-dire l'union de l'ensemble des axiomes 3.17 pour chaque $c \in \mathtt{MEM}(p)$ et de l'ensemble des axiomes 3.18 pour chaque paire distincte de cas de p, est noté $\mathtt{FT}(p)$. C'est ainsi que les axiomes 3.6 et 3.2 sont définis dans le plan d'ensemble de la figure 3.1. La formule 3.18 est définie comme un

axiome, mais on peut également l'ajouter en temps que lemme de manière à prévenir l'utilisateur si les cas ne sont pas distincts.

Le prédicat de séparation peut alors être défini pour le pragma $\texttt{pragma} : p(p_1, p_2)$, avec $p_1 : \langle T_{1,1}, \ldots, T_{1,n_1} \rangle$ et $p_2 : \langle T_{2,1}, \ldots, T_{2,n_2} \rangle$ par :

$$\begin{aligned}&\texttt{pred}\, p(T_{1,1}, \ldots, T_{1,n_1}, T_{2,1}, \ldots, T_{2,n_2}) = \\ &\qquad \bigwedge_{c \in \texttt{MEM}(p_1) \cap \texttt{MEM}(p_2)} \forall (q : \mathsf{P}).q \notin \texttt{ft}^c_{p_1} \vee q \notin \texttt{ft}^c_{p_2}\end{aligned}$$

Cela correspond exactement à l'interprétation que l'on a donnée précédemment. C'est l'axiome 3.9 dans le plan d'ensemble page 26.

Soit un ensemble Δ de déclarations, on définit $\texttt{FT}(\Delta)$ par l'ensemble des $\texttt{FT}(p)$ et l'ensemble des prédicats de séparation :

$$\begin{aligned}\texttt{FT}(\Delta) \triangleq\ &\{\texttt{FT}(p) | p \text{ un inductif totalement défini de } \Delta\} \\ &\cup \{\texttt{pred}\, p(\ldots) = \ldots | \texttt{pragma} : p(p_1, p_2) \text{ déclaré dans } \Delta\}.\end{aligned}$$

Exemple : Pour `islist` l'axiome 3.17 est :

$$\forall q. \forall m. \forall p. q \in \texttt{ft}^1_{\texttt{islist}}(m, p) \Leftrightarrow \left(\begin{array}{l} (p \approx \texttt{null} \wedge \bot) \\ \vee p \not\approx \texttt{null} \wedge \texttt{islist}(m, \texttt{get}(m, p)) \\ \quad \wedge ((q \approx p \vee q \in \texttt{ft}^1_{\texttt{islist}}(m, \texttt{get}(m, p)))) \end{array} \right) \tag{3.19}$$

Remarque Cet axiome peut être découpé et simplifié en plusieurs axiomes qui seront alors plus facilement utilisés par les prouveurs. On peut en premier séparer l'équivalence en deux axiomes séparés et simplifier les formules qui contiennent $\bot$. Puis dans l'axiome où $p \not\approx \mathit{null}$ apparaît en position positive, on élimine la quantification sur p par la substitution de p par `null`. Enfin dans l'axiome où $q \approx p$ apparaît en position négative, on élimine la quantification sur q par la substitution de q par p.

$$\begin{aligned}&\forall q. \forall m. q \notin \texttt{ft}^1_{\texttt{islist}}(m, \texttt{null})) \\ &\forall q. \forall m. \forall p. q \in \texttt{ft}^1_{\texttt{islist}}(m, q) \Rightarrow \texttt{islist}(m, \texttt{get}(m, p)) \\ &\forall q. \forall m. \forall p. q \in \texttt{ft}^1_{\texttt{islist}}(m, q) \Rightarrow p \approx q \vee q \in \texttt{ft}^1_{\texttt{islist}}(m, \texttt{get}(m, p)) \\ &\forall m. \forall p. p \not\approx \texttt{null} \Rightarrow \texttt{islist}(m, \texttt{get}(m, p)) \Rightarrow p \in \texttt{ft}^1_{\texttt{islist}}(m, p) \\ &\forall q. \forall m. \forall p. p \not\approx \texttt{null} \Rightarrow \texttt{islist}(m, \texttt{get}(m, p)) \Rightarrow q \in \texttt{ft}^1_{\texttt{islist}}(m, \texttt{get}(m, p)) \\ &\qquad \Rightarrow q \in \texttt{ft}^1_{\texttt{islist}}(m, q)\end{aligned}$$

Nous allons prouver que les axiomes de $\texttt{FT}(\delta)$ ne rentrent pas en contradiction avec le modèle car on peut étendre les modèles pour intégrer les nouveaux symboles logiques de manière qu'ils soient vérifiés.

Lemme 3.2.1. *Si un ensemble Δ de déclarations admet un modèle $\mathfrak{M}$ alors il existe un modèle $\mathfrak{I}$ de $\mathcal{T}(\Delta) \cup \mathtt{FT}(\Delta)$.*

Démonstration. On reprend la preuve du lemme 3.1.3 qui était restreinte aux ensembles Δ de déclarations sans pragma.

Nous allons montrer, pour chaque prédicat inductif p totalement défini que l'axiomatisation de $\mathtt{FT}(p)$ est satisfaite par son empreinte dans le modèle $\mathfrak{M}$. On définit l'interprétation de $(\mathtt{ft}_p^c)^{\mathfrak{I}}(\mathbf{t}_{\mathtt{MEM}(p)}, v_1, \ldots, v_n)$ par $\{v | (c, v) \in (\mathtt{ft}_p)^{\mathfrak{M}}(\zeta^{-1}(\mathbf{t}_{\mathtt{MEM}(p)}), v_1, \ldots, v_n)\}$.

Prouvons tout d'abord que pour tout $v \in \mathsf{P}$, pour tout champ c, pour toute valuation ψ, on a :

$$\begin{aligned}\mathfrak{I}_{\psi[q\mapsto v]}(\mathtt{FT}_{c,q}(f)) =& (c, v) \in \overline{\mathtt{FT}}_p(f)((\mathtt{ft}_p)^{\mathfrak{M}}, \psi, \zeta^{-1}(\psi))\\ \mathfrak{I}_{\psi[q\mapsto v]}(\mathtt{FT}_{c,q}(t)) =& (c, v) \in \overline{\mathtt{FT}}_p(t)((\mathtt{ft}_p)^{\mathfrak{M}}, \psi, \zeta^{-1}(\psi))\end{aligned}$$

pour toute formule f et tout terme t. Procédons par récurrence structurelle sur les formules et les termes :

– soit une variable $t = x$,

$$\begin{aligned}\mathfrak{I}_{\psi[q\mapsto v]}(\mathtt{FT}_{c,q}(x)) =& \bot\\ =& (c, v) \in \emptyset\\ =& (c, v) \in \overline{\mathtt{FT}}_p(x)((\mathtt{ft}_p)^{\mathfrak{M}}, \psi, \zeta^{-1}(\psi))\end{aligned}$$

– soit un déréférencement $t = t'.j$:

$$\begin{aligned}\mathfrak{I}_{\psi[q\mapsto v]}(\mathtt{FT}_{c,q}(t'.j)) =& \mathfrak{I}_{\psi[q\mapsto v]}(\mathtt{FT}_{c,q}(t)) \vee \begin{cases}\mathfrak{I}_{\psi[q\mapsto v]}(q \approx t') & \text{si } c = j\\ \bot & \text{sinon}\end{cases}\\ =& (c, v) \in \overline{\mathtt{FT}}_p(t')((\mathtt{ft}_p)^{\mathfrak{M}}, \psi, \zeta^{-1}(\psi)) \vee \begin{cases}v = \mathfrak{I}_{\psi}(t') & \text{si } c = j\\ \bot & \text{sinon}\end{cases}\\ =& (c, v) \in \overline{\mathtt{FT}}_p(t')((\mathtt{ft}_p)^{\mathfrak{M}}, \psi, \zeta^{-1}(\psi)) \vee (c, v) \in \{j, \mathfrak{I}_{\psi}(t')\}\\ =& (c, v) \in \overline{\mathtt{FT}}_p(t'.j)((\mathtt{ft}_p)^{\mathfrak{M}}, \psi, \zeta^{-1}(\psi))\end{aligned}$$

– soit une application de prédicat $f = p'(t_1, \ldots, t_n)$:

$$\begin{aligned}&\mathfrak{I}_{\psi[q\mapsto v]}(\mathtt{FT}_{c,q}(p'(t_1, \ldots, t_n)))\\ =& \bigvee_j \mathfrak{I}_{\psi[q\mapsto v]}(\mathtt{FT}_{c,q}(t_j)) \vee \begin{cases}\mathfrak{I}_{\psi[q\mapsto v]}(q \in \mathtt{ft}_{p'}^c(\mathbf{m}, \mathbf{t})) & \text{si } c \in \mathtt{MEM}(p)\\ \bot & \text{sinon}\end{cases}\\ =& \bigvee_j (c, v) \in \overline{\mathtt{FT}}_p(t_j)(\mathtt{ft}_p)^{\mathfrak{M}}, \psi, \zeta^{-1}(\psi)) && \text{(par HR)}\\ &\vee \begin{cases}(c, v) \in (\mathtt{ft}_{p'})^{\mathfrak{M}}(\zeta^{-1}(\psi), \mathfrak{M}_{\zeta^{-1}(\psi),\psi}(\mathbf{t})) & \text{si } c \in \mathtt{MEM}(p)\\ \bot & \text{sinon}\end{cases} && \text{(par déf.)}\end{aligned}$$

$$
\begin{aligned}
&= (c,v) \in \bigcup_j \overline{\mathtt{FT}}_p(t_j)(\mathtt{ft}_p)^{\mathfrak{M}}, \psi, m) \\
&\quad \cup \begin{cases} (\mathtt{ft}_p)^{\mathfrak{M}}(\zeta^{-1}(\psi), \mathfrak{M}_{\zeta^{-1}(\psi),\psi}(\mathbf{t})) & \text{si } p' = p \\ (\mathtt{ft}_{p'})^{\mathfrak{M}}(\zeta^{-1}(\psi), \mathfrak{M}_{\zeta^{-1}(\psi),\psi}(\mathbf{t})) \end{cases} \\
&= (c,v) \in \overline{\mathtt{FT}}_p(p'(\mathbf{t}))(\mathtt{ft}_p)^{\mathfrak{M}}, \psi, \zeta^{-1}(\psi))
\end{aligned}
$$

Soit un prédicat inductif p complètement défini par $p(\mathbf{x}) : \mathtt{C_i} : \forall \mathbf{x}. f_{i,1} \Rightarrow \cdots \Rightarrow f_{i,n_i} \Rightarrow \ldots p(\mathbf{x})$. Nous allons montrer que tous les axiomes de $\mathtt{FT}(p)$ sont vérifiés par $\mathfrak{I}$. Soient un champ c de $\mathtt{MEM}(p)$ et une valuation ψ. On prouve le premier sens de l'implication 3.17 : on a $\mathfrak{I}_\psi(q \in \mathtt{ft}_p^c(\mathbf{m}_{\mathtt{MEM}(p)}, \mathbf{x}))$ vrai, donc $\psi(q) \in (\mathtt{ft}_p^c)^{\mathfrak{I}}(\mathfrak{I}_\psi(\mathbf{m}_{\mathtt{MEM}(p)}), \mathfrak{I}_\psi(\mathbf{x}))$ qui revient par définition à $(c, \psi(q)) \in (\mathtt{ft}_p)^{\mathfrak{M}}(\zeta^{-1}(\mathfrak{I}_\psi(\mathbf{m}_{\mathtt{MEM}(p)})), \mathfrak{I}_\psi(\mathbf{x}))$. Par définition de $(\mathtt{ft}_p)^{\mathfrak{M}}$ il existe un cas i de l'inductif p tel que pour tout j on a $\mathfrak{M}_{\zeta^{-1}(\mathfrak{I}_\psi(\mathbf{m}_{\mathtt{MEM}(p)})),\psi}(f_{i,j})$. D'autre part, il existe une condition j_0 du cas i telle que $(c, \psi(q)) \in \overline{\mathtt{FT}}_p(f_{i,j_0}, \psi, \zeta^{-1}(\mathfrak{I}_\psi(\mathbf{m}_{\mathtt{MEM}(p)})))$ soit vérifié. À l'aide des égalités que l'on vient de prouver, on sait que $\mathfrak{I}_\psi(\mathtt{FT}_{c,q}(f_{i,j_0})))$ est vrai et grâce aux égalités du lemme 3.1.3 on a $\mathfrak{I}_\psi(f_{i,j})$ vrai pour tout j. Ainsi

$$\mathfrak{I}_\psi(f_{i,1} \wedge \cdots \wedge f_{i,n_i} \wedge \bigvee_j \mathtt{FT}_{c,q}(f_{i,j}))$$

est vrai.

Le deuxième sens se prouve de manière similaire.

□

Lemme 3.2.2. *Soit un ensemble Δ de déclarations et Δ' est la restriction de Δ sans pragma. S'il existe un modèle $\mathfrak{M}$ de $\mathcal{T}(\Delta') \cup \mathtt{FT}(\Delta)$ alors Δ admet un modèle $\mathfrak{I}$.*

Démonstration. On reprend la preuve du lemme 3.1.4 qui était restreinte aux ensembles Δ de déclarations sans pragma.

Pour tous les prédicats inductifs p, seuls les $\mathtt{ft}_p^c$ avec $c \in \mathtt{MEM}(p)$ apparaissent dans $\mathcal{T}(\Delta') \cup \mathtt{FT}(\Delta)$. On peut donc choisir pour les $c \notin \mathtt{MEM}(p)$ l'interprétation $\emptyset$ pour $(\mathtt{ft}_p^c)^{\mathfrak{M}}$. Nous allons montrer, pour chaque prédicat inductif totalement défini, que son empreinte dans le modèle $\mathfrak{M}$ correspond à l'interprétation requise pour l'interprétation dans Δ. On définit $(\mathtt{ft}_p)^{\mathfrak{I}}(m, v_1, \ldots, v_n)$ par l'ensemble $\{(c,v) | v \in (\mathtt{ft}_p^c)^{\mathfrak{M}}(\zeta_{\mathtt{MEM}(p)}(m), v_1, \ldots, v_n)\}$.

Prouvons tout d'abord que pour tout $v \in \mathsf{P}$, pour toute mémoire m, pour toute valuation ψ, pour toute formule f et tout terme t :

$$
\begin{aligned}
(c,v) \in \overline{\mathtt{FT}}_p(f)((\mathtt{ft}_p)^{\mathfrak{I}}, \psi, m) &= \mathfrak{M}_{\zeta(m,\psi)[q \mapsto v]}(\mathtt{FT}_{c,q}(f)) \\
(c,v) \in \overline{\mathtt{FT}}_p(t)((\mathtt{ft}_p)^{\mathfrak{I}}, \psi, m) &= \mathfrak{M}_{\zeta(m,\psi)[q \mapsto v]}(\mathtt{FT}_{c,q}(t))
\end{aligned}
\tag{3.20}
$$

Procédons par induction structurelle sur les formules et les termes :

– soit une variable $t = x$,

$$
\begin{aligned}
(c,v) \in \overline{\mathtt{FT}}_p((\mathtt{ft}_p)^{\mathfrak{I}}, \psi, m) &= (c,v) \in \emptyset \\
&= \bot \\
&= \mathfrak{M}_{\zeta(m,\psi)[q \mapsto v]}(\mathtt{FT}_{c,q}(x))
\end{aligned}
$$

– soit un déréférencement $t = t'.j$:

$$\begin{aligned}
(c,v) \in \overline{\mathtt{FT}}_p(t'.j)((\mathtt{ft}_p)^{\mathfrak{I}},\psi,m) =& (c,v) \in \overline{\mathtt{FT}}_p(t')((\mathtt{ft}_p)^{\mathfrak{I}},\psi,m) \\
& \vee (c,v) \in \{j, \mathfrak{I}_{m,\psi}(t')\} \\
=& (c,v) \in \overline{\mathtt{FT}}_p(t')((\mathtt{ft}_p)^{\mathfrak{I}},\psi,m) \\
& \vee \begin{cases} v = \mathfrak{I}_{m,\psi}(t') & \text{if } c = j \\ \bot & \text{sinon} \end{cases} \\
=& \mathfrak{M}_{\zeta(m,\psi)}(\mathtt{FT}_{c,q}(t)) \vee \begin{cases} \mathfrak{M}_{\zeta(m,\psi)[q\mapsto v]}(q \approx t') & \text{if } c = j \\ \bot & \text{sinon} \end{cases} \\
=& \mathfrak{M}_{\zeta(m,\psi)[q\mapsto v]}(\mathtt{FT}_{c,q}(t'.j))
\end{aligned}$$

– soit une application de prédicat $f = p'(t_1, \ldots, t_n)$:

$$\begin{aligned}
& (c,v) \in \overline{\mathtt{FT}}_p(p'(\mathbf{t}))((\mathtt{ft}_p)^{\mathfrak{I}},\psi,m) \\
& \quad = (c,v) \in \bigcup_j \overline{\mathtt{FT}}_p(t_j)((\mathtt{ft}_p)^{\mathfrak{I}},\psi,m) \\
& \qquad \cup \begin{cases} (\mathtt{ft}_{p'})^{\mathfrak{I}}(m, \mathfrak{I}_{m,\psi}(\mathbf{t})) & \text{si } p' = p \\ (\mathtt{ft}_{p'})^{\mathfrak{I}}(m, \mathfrak{I}_{m,\psi}(\mathbf{t})) \end{cases} \\
& \quad = \bigvee_j (c,v) \in \overline{\mathtt{FT}}_p(t_j)((\mathtt{ft}_p)^{\mathfrak{I}},\psi,m) \\
& \qquad \vee \begin{cases} (c,v) \in (\mathtt{ft}_{p'})^{\mathfrak{I}}(m, \mathfrak{I}_{m,\psi}(\mathbf{t}))) & \text{si } c \in \mathtt{MEM}(p') \\ \bot & \text{sinon} \end{cases} \\
& \quad = \bigvee_j \mathfrak{M}_{\zeta(m,\psi)[q\mapsto v]}(\mathtt{FT}_{c,q}(t_j)) && \text{(par HR)} \\
& \qquad \vee \begin{cases} \mathfrak{M}_{\zeta(m,\psi)[q\mapsto v]}(q \in \mathtt{ft}^c_{p'}(\mathbf{m},\mathbf{t})) & \text{si } c \in \mathtt{MEM}(p) \\ \bot & \text{sinon} \end{cases} && \text{(par déf.)} \\
& \quad = \mathfrak{M}_{\zeta(m,\psi)[q\mapsto v]}(\mathtt{FT}_{c,q}(p'(t_1,\ldots,t_n)))
\end{aligned}$$

Soit un prédicat inductif totalement défini $p(\mathbf{x}) : \mathtt{C_i} : \forall \mathbf{x}. f_{i,1} \Rightarrow \cdots \Rightarrow f_{i,n_i} \Rightarrow \ldots p(\mathbf{x})$. Nous allons montrer que $(\mathtt{ft}_p)^{\mathfrak{I}}$ vérifie :

$$(\mathtt{ft}_p)^{\mathfrak{I}} = \mu f.\lambda m, z_1, \ldots, z_n. \bigcup_{\substack{i \text{ tq } \mathfrak{I}_{m,\psi'}(f_{i,j}) \text{ est vrai} \\ \text{pour tout } j}} \bigcup_j \overline{\mathtt{FT}}_p(f_{i,j})(f,\psi,m)$$

Pour cela nous allons montrer tout d'abord que $(\mathtt{ft}_p)^{\mathfrak{I}}$ est un des points fixes possibles.

Soit $v \in \mathcal{D}_{\mathsf{P}}^{\mathfrak{I}}$ et c un champ de $\mathtt{MEM}(p)$:

$$\begin{aligned}
(c,v) \in & \bigcup_{\substack{i \text{ tq } \mathfrak{I}_{m,\psi}(f_{i,j}) \text{ est} \\ \text{vrai pour tout } j}} \bigcup_{j} \overline{\mathtt{FT}}_p(f_{i,j})((\mathtt{ft}_p)^{\mathfrak{I}}, \psi, m) \\
= & \bigvee_{\substack{i \text{ tq } \mathfrak{M}_{\zeta(m,\psi)}(f_{i,j}) \\ \text{est vrai pour tout } j}} \bigvee_{j} \mathfrak{M}_{\zeta(m,\psi)[q\mapsto v]}(\mathtt{FT}_{c,q}(f_{i,j})) && \text{par (3.20)} \\
= & \bigvee_{i} \left(\bigwedge_{j} \mathfrak{M}_{\zeta(m,\psi)}(f_{i,j}) \wedge \bigvee_{j} \mathfrak{M}_{\zeta(m,\psi)[q\mapsto v]}(\mathtt{FT}_{c,q}(f_{i,j})) \right) \\
= & \mathfrak{M}_{\zeta(m,\psi)[q\mapsto v]}(\bigvee_{i} \bigwedge_{j} f_{i,j} \wedge \bigvee_{j} \mathtt{FT}_{c,q}(f_{i,j})) \\
= & \mathfrak{M}_{\zeta(m,\psi)[q\mapsto v]}(q \in \mathtt{ft}_p^c(\mathbf{x})) && \text{par (3.17)} \\
= & v \in (\mathtt{ft}_p^c)^{\mathfrak{M}}(\mathfrak{M}_{\zeta(m,\psi)[q\mapsto v]}(\mathbf{x})) \\
= & (c,v) \in (\mathtt{ft}_p)^{\mathfrak{I}}(\mathfrak{I}_{m,\psi}(\mathbf{x})) && \text{par def}
\end{aligned}$$

Nous allons maintenant prouver que

$$(\mathtt{ft}_p)^{\mathfrak{I}} \subset \mu f.\lambda m, z_1, \ldots, z_n. \bigcup_{\substack{i \text{ tq } \mathfrak{I}_{m,\psi'}(f_{i,j}) \text{ est vrai} \\ \text{pour tout } j}} \bigcup_{j} \overline{\mathtt{FT}}_p(f_{i,j})(f, \psi, m). \tag{3.21}$$

Soit $c \in \mathtt{MEM}(p)$ et $v \in \mathcal{D}_{\mathsf{P}}^{\mathfrak{I}}$. On va prouver que pour tout $v_i \in \mathcal{D}_{\mathsf{P}}^{\mathfrak{I}}$, $t \in \mathcal{D}_{\mathsf{M}}^{\mathfrak{I}}$, si on a $p^{\mathfrak{I}}(m, v_1, \ldots, v_n)$ et $(c,v) \in (\mathtt{ft}_p)^{\mathfrak{I}}(t,r)$, alors on a

$$(c,v) \in \overbrace{\mu f.\lambda m, z_1, \ldots, z_n. \bigcup_{\substack{i \text{ tq } \mathfrak{I}_{m,\psi'}(f_{i,j}) \text{ est vrai} \\ \text{pour tout } j}} \bigcup_{j} \overline{\mathtt{FT}}_p(f_{i,j})(f, \psi, m)}^{A}(t,r)$$

par induction structurelle sur $p^{\mathfrak{I}}(m, v_1, \ldots, v_n)$. Le principe d'induction d'un p arbitraire est assez fastidieux à utiliser ; il est donc difficile de faire la preuve dans le cas général. On va donc réaliser la preuve sur un cas particulier, `islist` ; elle pourrait être réalisée de même pour tout autre prédicat inductif.

Ainsi traitons le premier cas de l'inductif `islist` : $r = \mathtt{null}^{\mathfrak{I}}$. On pose $\psi = [q \mapsto v, p \mapsto r, m_1 \mapsto \zeta_1(t)]$:

$$\begin{aligned}
(c,v) \in (\mathtt{ft}_{\mathtt{islist}})^{\mathfrak{I}}(t,r) = & v \in (\mathtt{ft}^c_{\mathtt{islist}})^{\mathfrak{M}}(\zeta_1(t), v) \\
= & \mathfrak{M}_{\psi}(p \not\approx \mathtt{null} \wedge \mathtt{islist}(m, get(m,p)) \\
& \wedge ((q \approx p \vee q \in \mathtt{ft}^1_{\mathtt{islist}}(m, get(m,p))))) \\
= & \bot \\
= & (c,v) \in \overline{\mathtt{FT}}(p \approx \mathtt{null})(A, \psi, t) \\
= & (c,v) \in A(t,r)
\end{aligned}$$

La dernière équation est obtenue à l'aide de l'équation 3.19.

Dans le second cas on a $r \neq \texttt{null}^{\mathfrak{I}}$ et $\texttt{islist}^{\mathfrak{I}}(t, t(1)(r))$. Donc par hypothèse de récurrence on a $(c, v) \in (\texttt{ft}_{\texttt{islist}})^{\mathfrak{I}}(t, t(1)(r))$ implique $(c, v) \in A(t, t(1)(r))$.

$$\begin{aligned}
(c, v) \in (\texttt{ft}_{\texttt{islist}})^{\mathfrak{I}}(t, r) =& v \in (\texttt{ft}^{c}_{\texttt{islist}})^{\mathfrak{M}}(\zeta_1(t), v) \\
=& \mathfrak{M}_{\psi}(p \not\approx \texttt{null} \wedge \texttt{islist}(m_1, get(m_1, p)) \\
& \wedge ((q \approx p \vee q \in \texttt{ft}^{1}_{\texttt{islist}}(m_1, get(m_1, p))))) \\
=& \mathfrak{M}_{\psi}(q \approx p) \vee \mathfrak{M}_{\psi}(q \in \texttt{ft}^{1}_{\texttt{islist}}(m_1, get(m_1, p))) \\
=& (c, v) \in \{(c, v)\} \vee v \in (\texttt{ft}^{1}_{\texttt{islist}})^{\mathfrak{M}}(t(1), t(1), p) \\
=& (c, v) \in \{(c, v)\} \vee (c, v) \in (\texttt{ft}_{\texttt{islist}})^{\mathfrak{I}}(t, t(1), p) \\
=& (c, v) \in A(t, r)
\end{aligned}$$

Ainsi si $\texttt{islist}(t, r)$ est vérifié l'équation (3.21) est vérifiée. Dans le cas où $\texttt{islist}(t, r)$ n'est pas vérifié les deux ensembles sont vides. Ce qui termine la preuve. □

On peut noter que cette démonstration utilise le fait que l'on génère l'empreinte d'un inductif. Si $\mathcal{T}(\Delta)$ ne contenait plus d'inductifs mais seulement des axiomes (ce n'est plus nécessairement le plus petit), le résultat ne serait plus valable. En effet gardons l'exemple des listes. Si l'on perd la minimalité de la définition inductive des listes, un modèle peut accepter les listes de taille non nulle qui bouclent. Une telle liste possède avec les axiomes une empreinte non vide. En revanche dans le modèle initial, une telle liste est interdite et donc son empreinte est vide (eq.3.11, page 33), puisque aucun des cas de l'inductif est vrai on fait l'union sur aucun ensemble.

Axiome d'encadrement Maintenant que l'empreinte est définie on peut définir l'axiome d'encadrement (axiome 3.7 dans le plan d'ensemble) pour chaque prédicat inductif totalement défini $p(\mathbf{x})$ pour chaque champ $c \in \texttt{MEM}(p)$:

$$\begin{aligned}
\forall q.\forall v.\forall \mathbf{m}.\forall \mathbf{x}.\neg q \in \texttt{ft}^{c}_{p}(\mathbf{m}, \mathbf{x}) \Rightarrow \\
p(m_1, \ldots, m_c, \ldots, m_k, \mathbf{x}) \Rightarrow p(m_1, \ldots, \texttt{set}(m_c, q, v), \ldots, m_k, \mathbf{x})
\end{aligned}$$

Exemple : Pour `islist` l'axiome d'encadrement est :

$$\forall q.\forall v.\forall m_1.\forall p.\neg q \in \texttt{ft}^{1}_{\texttt{islist}}(m_1, p) \Rightarrow \texttt{islist}(m_1, q) \Rightarrow \texttt{islist}(\texttt{set}(m_1, q, v), q)$$

Axiome d'auto-encadrement Maintenant, on définit l'axiome d'auto-encadrement de l'empreinte (axiome 3.8 dans le plan d'ensemble) d'un prédicat inductif $p(\mathbf{m}, \mathbf{x})$ pour la mémoire m_c :

$$\begin{aligned}
\forall q.\forall v.\forall \mathbf{m}.\forall \mathbf{x}.\neg q \in \texttt{ft}^{c}_{p}(\mathbf{m}, \mathbf{x}) \Rightarrow p(\mathbf{m}, \mathbf{x}) \Rightarrow \\
\texttt{ft}^{c}_{p}(m_1, \ldots, m_c, \ldots, m_k, \mathbf{x}) = \texttt{ft}^{c}_{p}(m_1, \ldots, \texttt{set}(m_c, q, v), \ldots, m_k, \mathbf{x})
\end{aligned}$$

Exemple : Pour `islist` l'axiome d'auto-encadrement est :

$$\forall q.\forall v.\forall m_1.\forall p.\neg q \in \mathtt{ft}^1_{\mathtt{islist}}(m_1,p) \Rightarrow \mathtt{islist}(m_1,p) \Rightarrow \mathtt{ft}^1_{\mathtt{islist}}(m_1,p) = \mathtt{ft}^1_{\mathtt{islist}}(\mathtt{set}(m_1,q,v),p)$$

Pour un prédicat inductif p, on définit $\mathtt{FT}'(p)$ comme l'ensemble des axiomes d'encadrement et d'auto-encadrement pour le prédicat p et pour chaque champ $c \in \mathtt{MEM}(p)$.

Lemme 3.2.3. *Soit une liste Δ de déclarations. Si $\mathcal{T}(\Delta) \cup \mathtt{FT}(\Delta)$ est vraie dans un modèle $\mathfrak{M}$ alors les axiomes d'encadrement et d'auto-encadrement le sont également dans le même modèle.*

Démonstration. Les lemmes 3.1.4 et 3.1.1 montrent la validité des axiomes d'encadrement. Les lemmes 3.1.4 et 3.1.2 montrent la correction des axiomes d'auto-encadrement. □

Théorème 3.2.4. *Si une liste Δ de déclarations est satisfiable alors $\mathcal{T}(\Delta) \cup \mathtt{FT}(\Delta) \cup \mathtt{FT}'(\Delta)$ est satisfiable.*

Démonstration. Les lemmes 3.2.1 et 3.2.3 montrent le résultat. □

Théorème 3.2.5. *Soit une liste Δ de déclarations. Si $\mathcal{T}(\Delta) \cup \mathtt{FT}(\Delta) \cup \mathtt{FT}'(\Delta)$ est satisfiable alors Δ est satisfiable.*

Démonstration. Ce résultat est l'affaiblissement du lemme 3.2.2 (plus exactement le lemme 3.2.3 montre qu'il est équivalent) . □

Théorème 3.2.6. *Soit une liste Δ de déclarations, une mémoire m globale, une formule F et une valuation ψ. On a $\Delta \models F, m, \psi$ si et seulement si $\mathcal{T}(\Delta) \cup \mathtt{FT}(\Delta) \cup \mathtt{FT}'(\Delta) \models \mathcal{T}(F), \zeta(m,\psi)$.*

Démonstration. Supposons que l'on a $\Delta \models F, m, \psi$. Soit un modèle $\mathfrak{M}$ de $\mathcal{T}(\Delta) \cup \mathtt{FT}(\Delta) \cup \mathtt{FT}'(\Delta)$, par le lemme 3.2.2 il existe un modèle $\mathfrak{I}$ de Δ tel que leur interprétation sur les formules sont identiques. Donc comme $\mathfrak{I}_{\zeta^{-1}(\zeta(m,\psi)),\zeta(m,\psi)}(F) = \mathfrak{I}_{m,\psi}(F)$ est vraie, $\mathfrak{M}_{\zeta(m,\psi)}(F)$ est vrai. Ainsi on a le premier sens. La réciproque ce prouve de la même manière avec le lemme 3.2.1 et le lemme 3.2.3. □

Séparation avec un pointeur Nous avons vu comment on peut définir un prédicat de séparation entre deux inductifs. Il est également utile de définir un prédicat de séparation entre un prédicat inductif et un seul pointeur. On peut demander sa génération ainsi :

$$\texttt{pragma} : p(i, p)$$

Un prédicat p défini par `pragma` : $p(i, p_1)$ avec un prédicat inductif totalement défini $p_1 : \langle T_1, \ldots, T_n \rangle$ est alors interprété pour toute mémoire $m \in \mathcal{D}^{\mathfrak{I}}_{\mathsf{M}}$, pour tout pointeur $v : \mathcal{D}^{\mathfrak{I}}_{\mathsf{P}}$ et pour toutes valeurs $v_j : \mathcal{D}^{\mathfrak{I}}_{T_j}$ par

$$p^{\mathfrak{I}}(m, v, v_1, \ldots, v_n) \triangleq (i, v) \notin (\texttt{ft}_{p_1})^{\mathfrak{I}}(m, v_1, \ldots, v_n)$$

L'interprétation utilise la même empreinte du prédicat p_1 que précédemment. Si p_1 apparaît dans un autre pragma on peut réutiliser l'empreinte déjà définie sinon on la définit comme précédemment. La définition de ce pragma est alors simplement, dans le cas où $i \in \texttt{MEM}(p_1)$:

$$\texttt{pred}\, p(\mathbf{m}_{\texttt{MEM}(p)}, x, x_1 : T_1, \ldots, x_n : T_n) : T = x \notin \texttt{ft}^i_{p_1}(m, x_1, \ldots, x_n)$$

et sinon :

$$\texttt{pred}\, p(\mathbf{m}_{\texttt{MEM}(p)}, x, x_1 : T_1, \ldots, x_n : T_n) : T = \top$$

Exemple : Avec cette extension du pragma, un utilisateur peut déclarer le lemme qui indique que le pointeur de tête n'est pas présent dans le reste de la liste (2.1 p.20).

```
/*@ pragma : sep_node_list(1,islist)

axiom acyclic_list : ∀ p : struct * node.
   p != null ⇒ islist(p) ⇒ sep_node_list(p,p->next)
@*/
```

Travail relié La technique *Implicit Dynamic Frame* définit ces mêmes axiomes pour chaque méthode pure définit [55]. L'empreinte des prédicats est défini à l'aide de sa définition et d'opérateurs sur les ensembles : union, intersection, appartenance. L'évaluation des prédicats doit terminer ; des critères syntaxiques ou des critères de strictes inclusions des empreintes des appels récursifs sont utilisés pour s'assurer de cela. Les prédicats ne peuvent utiliser que des formules qui contiennent des conjonctions, des conjonctions séparantes (qui sont traduites comme une conjonction normale avec l'ajout que les empreintes des deux formules sont disjointes) ou des tests. Dans le contexte qui a été présenté les conjonctions séparantes doivent être exprimer à l'aide d'un prédicat de frame. Les tests, sous la forme de `if then else`, pourraient être ajoutée à notre algèbre de termes. En effet les deux cas sont forcément distincts ce qui assure que l'empreinte s'auto-encadre. Les axiomes ont été trouvés dans un premier temps indépendemment de ce travail, ainsi que des Dynamic Frame[39]. Cependant ces travaux ont ensuite conforté des notions, particulièrement celle de l'auto-encadrement.

3.3. Empreintes de programmes

Lorsque l'on appelle une fonction, l'état de la mémoire est modifié. On ne peut donc pas supposer que les propriétés qui étaient vraies avant l'appel de la fonction le sont toujours après. Ainsi dans l'exemple ci-dessous, les seules propriétés que l'on sait vraies après l'appel de la fonction `foo` dans la fonction `bar` sont les propriétés de la postcondition de `foo`, c'est-à-dire `post_foo`.

```
/*@
requires pre_foo;
ensures post_foo;
@*/
int foo (...){
  ...
}

/*@
requires pre_bar;
ensures post_bar;
@*/
int bar (...){
  ...
  foo(...)
  ...
}
```

Dit autrement, on abstrait le corps de la fonction par sa spécification pour permettre une vérification modulaire. Une solution habituelle consiste à ajouter à la postcondition de `foo` une formule indiquant quelle partie de la mémoire n'a pas été modifiée par l'appel de la fonction. La difficulté consiste à décrire cette partie de la mémoire. `Jessie` permet de décrire l'ensemble des zones mémoires qui sont modifiées à l'aide d'une union comprenant des parties de tableaux comprises entre deux indices ou des valeurs de pointeurs ou des champs de structures.

Cependant, cela ne suffit pas pour certains programmes. En effet, si l'on reprend l'exemple de l'inversion de liste, `Jessie` ne nous permet pas d'exprimer que seuls les champs `next` des pointeurs de la liste ont été modifiés. `Jessie` nous permet seulement d'exprimer que *tous* les champs `next` possibles peuvent avoir été modifiés, qu'ils appartiennent à la liste ou non. Dans l'exemple suivant on voit les limites de cette approximation :

```
/*@
requires islist(p) ∧ islist(q);
ensures islist(p) ∧ islist(q);
@*/
void bar (struct node * p, struct node * q){
  p = rev(p);
```

```
  /*@ assert is_list(q) @*/
}
```

Puisque `rev` indique que tous les champs `next` ont pu être modifiés, on ne peut prouver l'assertion. Seul `islist(p)` est prouvable car il fait partie de la postcondition.

La logique de séparation, quant à elle, traite ce problème différemment de `Jessie`. Toutes les zones mémoires qui pourront être modifiées dans une fonction doivent apparaître dans sa précondition ou être allouées au sein de la fonction. Cela permet de définir la règle d'encadrement qui permet de plonger une fonction dans un contexte plus grand que celui utilisé lors de sa définition et qui ne sera pas modifié :

$$\frac{\{P\}f(\vec{x})\{Q\}}{\{P \star R\}f(\vec{x})\{Q \star R\}}$$

On peut remarquer, comme pour la règle de mutation, que si les pointeurs présents dans R et dans P ne sont pas disjoints, cette règle reste correcte car elle revient à prouver $\{false\}f(\vec{x})\{Q \star R\}$ qui est toujours vrai. Dans le cas de l'inversion de liste on a :

$$\frac{\{\texttt{islist}(p)\}\texttt{rev}(p)\{\texttt{islist}(result)\}}{\{\texttt{islist}(p) \star \texttt{islist}(q)\}\texttt{rev}(p)\{\texttt{islist}(result) \star \texttt{islist}(q)\}}$$

Et ainsi on peut prouver l'assertion.

Revenons à `Jessie`. Pour définir des zones mémoires plus précises, on va utiliser les empreintes de prédicats inductifs calculées précédemment. Ensuite on décrira un axiome d'encadrement qui précisera que, si une fonction n'a modifié que des zones mémoires qui sont séparées de l'empreinte d'un autre prédicat, alors ce prédicat est conservé par l'appel de fonction. Ainsi la fonction suivante devient prouvable :

```
/*@
requires islist(p) ∧ islist(q) ∧ sep_list_list(p,q);
ensures islist(p) ∧ islist(q) ∧ sep_list_list(p,q);
@*/
void bar (struct node * p, struct node * q){
  *p = rev(p);
  /*@ assert islist(q) @*/
}
```

Pour que la séparation soit elle aussi conservée par la fonction `bar` il faut ajouter à la postcondition de la fonction `rev` que l'empreinte de `islist` (p) n'a pas grandi lors de l'appel. On regroupe tout cela au sein d'un prédicat généré automatiquement.

```
/*@ #pragma_frame : list_frame list @*/
```

De manière plus générale on va définir un prédicat de frame à partir d'un prédicat inductif. Ces prédicats de frame sont particuliers car ils dépendent de deux états mémoires. De plus, ils dépendent de deux instances du prédicat inductif, c'est-à-dire qu'ils possèdent deux fois plus d'arguments que le prédicat inductif.

Dans `Jessie` les états mémoires sont référencés à l'aide de points de programme. Les points de programme peuvent être définis à l'aide d'une construction qui ressemble aux labels des goto en `C`, `Init: q = Null;`. Ils correspondent alors à l'état mémoire lors du dernier passage du programme par ce point. D'autres point de programme sont prédéfinis, par exemple `Here` correspond à l'état mémoire actuel, `Old`, lorsqu'il est utilisé dans la postcondition, correspond à l'état mémoire juste avant l'exécution de la fonction.

Ainsi un prédicat p défini par `pragma_frame` : $p\,p_1$ avec $p_1 : \langle T_1, \ldots, T_n \rangle$, est utilisé pour deux points de programme A, B et pour des termes $t_{1,1} : T_1, \ldots, t_{1,n} : T_n$ et $t_{2,1} : T_1, \ldots, t_{2,n} : T_n$ sous la forme de $p\{A, B\}(t_{1,1}, \ldots, t_{1,n}, t_{2,1}, \ldots, t_{2,n})$. Ce prédicat exprime alors que les seules modifications effectuées sur l'état mémoire entre A et B sont incluses dans l'empreinte de $p_1(v_{1,1}, \ldots, v_{1,n})$ calculée dans l'état A. Le prédicat $p(v_{1,1}, \ldots, v_{1,n}, v_{2,1}, \ldots, v_{2,n})$ exprime de plus que l'empreinte de $p_1(v_{2,1}, \ldots, v_{2,n})$ dans l'état B est incluse dans l'empreinte de $p_1(v_{1,1}, \ldots, v_{1,n})$ dans l'état A.

On peut donc donner à `rev` la spécification suivante :

```
/*@
requires list(p);
ensures list(\result) ∧ list_frame{Old,Here}(p,result);
@*/
struct node * rev(struct node * p);
```

Pour prouver cette spécification il faut modifier l'invariant comme dans la figure 3.7. Le pragma `#pragma_sub : list_sub list` définit le même prédicat que `list_frame` à l'exception près qu'il exprime seulement l'inclusion des deux empreintes et n'indique pas que les modifications sont incluses dans la première empreinte.

Tous ces prédicats générés automatiquement peuvent être définis à partir des empreintes des prédicats inductifs qui ont été définies dans le chapitre précédent. Par exemple l'inclusion est définie par :

$$\forall ft_1, ft_2.\, ft_1 \subset ft_2 \equiv \forall x.\, x \in ft_1 \Rightarrow x \in ft_2$$

De même, le fait que toutes les modifications entre deux états mémoires m^1 et m^2 sont incluses dans l'empreinte d'un prédicat inductif p s'exprime par la formule :

$$\bigwedge_i \begin{cases} \forall x : \mathsf{P}.\, \mathtt{get}(m_i^1, x) \not\approx \mathtt{get}(m_i^1, x) \Rightarrow \mathtt{ft}_p^i(m_{\mathtt{MEM}(p)}) & \text{si } i \in \mathtt{MEM}(p) \\ m_i^1 \approx m_i^2 & \text{sinon} \end{cases}$$

Pour faciliter le travail des démonstrateurs automatiques on préfère, comme on l'a fait avec l'empreinte $x \in \mathtt{ft}_p^i$, définir directement à l'aide de la définition de p les propriétés que vérifie l'inclusion de l'empreinte de $\mathtt{ft}_p^i$ par rapport à une autre empreinte ou elle-même. Par exemple l'empreinte de la queue de la liste est incluse dans l'empreinte de la liste. Ou bien encore si l'empreinte de la queue d'une liste est incluse dans une autre empreinte alors l'empreinte de la liste est incluse dans l'autre empreinte si le pointeur de tête de la liste appartient à l'autre empreinte. Toutes ces propriétés peuvent être déduites à partir de la définition d'une empreinte. Cependant, ajouter ces propriétés

```
struct node {
  [...]
}

/*@
inductive list(struct node * p) =
[...]
@*/

/*@
#pragma : sep_list_list(list,list)
#pragma : sep_node_list(node,list)
#pragma_frame : list_frame list
#pragma_sub : list_sub list list
@*/

/*@
axiom acyclic : [...]
@*/

/*@
requires list(p);
ensures list(\result) ∧ list_frame{Old,Here}(p,result);
@*/
struct node * rev(struct node * p){
  struct node * q;
  Init : q = NULL;

  /*@ loop invariant list(p) ∧ list(q) ∧ sep_list_list(p,q) ∧
  list_frame{Init,Here}(\at(p,Init),q) ∧
  list_sub{Init,Here}(\at(p,Init),p);
  @*/
    while(p != NULL){
        struct node * tmp = p->tl;
        p->tl = q;
        q = p;
        p = tmp;
    }
    return q;
}
```

FIGURE 3.7.: Spécification de l'inversion de liste en place avec un prédicat d'encadrement.

plus précises évite de devoir faire des étapes de raisonnement supplémentaires et facilite l'instanciation des axiomes.

La règle de frame semble au final plus simple à utiliser que les techniques décrites ci-dessus. En effet on obtient immédiatement avec la règle de frame que `islist`(q) est vrai mais aussi qu'il est séparé de `islist`(p). Dans notre cas, on prouve d'une part que `islist`(q) est toujours vrai après l'appel de la fonction, mais d'autre part rien n'empêche que la liste partant de p se finisse par la liste partant de q et ainsi n'interdise à la séparation d'être conservée même si `islist`(q) n'est pas touché. Dans la logique de séparation, quand on prouve $\{$`islist`$(p)\}$`rev`$(p)\{$`islist`$(result)\}$, `islist`($result$) ne peut contenir que des pointeurs apparaissant dans `islist`(p) ou alloués dans `rev`(p), donc en l'occurrence aucun pointeur qui appartienne à `islist`(q). *Implicit Dynamic Frame* utilise ces techniques pour réduire les annotations que doit mettre l'utilisateur. Ces raisonnements sont des meta-raisonnements de la logique de séparation. Peut-on faire les mêmes dans notre contexte ? Dans notre cas puisque `islist`(p) et `islist`(q) sont séparés, `islist`(p), qui est la précondition de `rev`, ne peut pas impliquer la validité d'un pointeur de `islist`(q). Donc la postcondition de `rev` ne peut pas impliquer la validité d'un pointeur de `islist`(q). Ainsi `islist`(q) ne peut être inclus dans `islist`($result$), et donc `islist`($result$) est bien séparé de `islist`(q). On obtient ainsi bien une simulation de la règle de frame. Cependant cette étude n'a pas été poussée plus avant.

4. Implémentation et études de cas

4.1. L'architecture de `Jessie`

`Jessie` a été conçu pour transformer un programme pouvant contenir des alias ainsi que sa spécification en un programme sans alias en utilisant un modèle mémoire à la Burstall-Bornat. La transformation est séparée en plusieurs étapes à la manière d'un compilateur :

1. lecture et typage du programme ;
2. calcul du graphe des appels de fonctions ;
3. calcul de l'ensemble des champs utilisés par chaque fonction (`MEM` dans la formalisation de la section 3.1) ;
4. transformation de chaque déclaration ($\mathcal{T}$ en est une grande simplification) ;

Au cours du temps de nombreuses fonctionnalités ont été ajoutées à `Jessie` :

1. ajout des unions de structure [46] ;
2. raffinement du modèle de Burstal-Bornat en utilisant des régions statiques [47] ;
3. ajout de notions de *pack/unpack* [1] ;

4.2. L'implémentation dans `Jessie`

Mon implémentation des prédicats de séparation s'est insérée au sein des différentes étapes de transformation. Pour ne pas risquer de modifier le comportement de `Jessie` par inadvertance, toutes les étapes ajoutées ont été protégées par une option de la ligne de commande, "-gen_frame_rule_with_ft". Ces nouvelles étapes sont ainsi activées seulement en présence de cette option :

- Le parseur reconnaît les pragmas et les ajoute en tant que déclarations particulières.
- Lors du typage des pragmas, on utilise les signatures des prédicats inductifs précédemment déclarés pour calculer la signature des prédicats de séparation. Comme on ne connaît pas encore les champs (`MEM`) des prédicats, on ne peut donner la définition exacte du prédicat de séparation. On donne donc une définition temporaire, à savoir la conjonction des prédicats inductifs. De cette manière les champs (`MEM`) calculés pour le prédicat de séparation sont bien l'union des deux ensembles de champs des prédicats inductifs.
- Une fois les champs (`MEM`) des prédicats inductifs connus, on peut calculer l'ensemble des empreintes qui seront générées à l'étape suivante. Les empreintes que l'on doit définir ne sont pas seulement celles qui apparaissent directement dans un pragma :

on doit également définir les empreintes des prédicats qui les définissent, et ainsi de suite.
- Après que des transformations préexistantes ont ajouté les champs en tant qu'arguments explicites, l'axiomatique des empreintes de prédicats inductifs est générée à partir de la définition de ces prédicats. Les axiomes d'encadrement et d'auto-encadrement sont générés en même temps.

Le code ajouté représente environ 1500 lignes de code.

4.3. Étude de cas

Nous reprenons le cas qui nous a accompagné durant toute cette partie : l'inversion de liste en place. La spécification utilisée est celle de la section précédente (fig.3.7). Cependant lors de l'implémentation, il n'a pas été possible d'ajouter les pragmas, c'est à dire les déclarations de prédicats de séparation, dans le parseur et le typeur de `Frama-C`. Le système de plugin de `Frama-C` ne permet pas encore à un plugin d'ajouter à sa convenance des règles de grammaire ainsi que de typage dans le langage logique. Les exemples précédents qui utilisaient des pragmas dans un programme `C` ne peuvent donc pas être traités directement par `Frama-C`. En réalité il faut traduire le programme `C` avec sa spécification en utilisant des définitions vides à la place des pragmas. Puis prendre le fichier `Jessie` et remplacer les définitions vides par le pragma voulu. Ci-dessous le fichier `Jessie` utilisé avec les pragmas :

Définition des types utilisés et de la structure utilisée :

```
type int32 = -2147483648..2147483647

tag node = {
  int32 hd; 32;
  node[..] tl; 32;
}

type node = [node]
```

On définit le prédicat inductif `list`. La validité est exprimée par `Jessie` par `(\offset_min(p_0) <= 0 && \offset_max(p_0) >= 0)` :

```
predicate list{L}(node[..] p) {
  case nil{L}: list{L}(null);
  case cons{L}: \forall node[..] p_0;
    p_0 != null ==>
     (\offset_min(p_0) <= 0 && \offset_max(p_0) >= 0) ==>
     list{L}(p_0.tl) ==> list{L}(p_0);
}
```

On demande la génération automatique du prédicat de séparation entre deux listes, du prédicat de séparation entre une cellule et une liste, du prédicat d'encadrement d'une

liste et d'un prédicat qui indique que l'empreinte d'une liste est incluse dans l'empreinte d'une autre liste.

```
/*@
#Gen_Separation sep_list_list(list,list)
#Gen_Separation sep_node_list(node[..] ,list)
#Gen_Frame list_framed list
#Gen_Sub list_sub list list

lemma list_sep{L} :
(∀ node[..] p_2; (list{L}(p_2)
                          ⇒ sep_node_list{L}(p_2, p_2.tl)))
@*/
```

Attention dans l'implémentation le mot clé `#pragma :` est remplacé par `#Gen_Separation`, `#pragma_frame :` par `#Gen_Frame` et `#pragma_sub :` par `#Gen_Sub`.

Enfin la fin du fichier `Jessie` comprenant la définition de la fonction `rev` et la fonction `foo` qui l'utilise se trouve dans la figure 4.1

Ensuite à partir de ce fichier on utilise la sortie 3 de `Jessie`. En premier on simplifie les buts à prouver par des méthodes génériques (découpage des formules au niveau des conjonctions, remplacement des prédicats par leurs définitions). Ensuite on lance les démonstrateurs (`Alt-Ergo`, Z3, CVC3) sur les vingt cinq buts obtenus. Tous les buts ont été prouvés en moins d'une seconde par au moins un démonstrateur. Les prouveurs Z3 et CVC3 ont été utilisés après avoir encodé le problème à l'aide des techniques de la partie suivante. Si on n'utilise pas l'une de ces techniques, la distinction, la moitié des conditions de vérification ne sont plus prouvées.

Cette partie a montré comment les prédicats de séparations peuvent être engendrés automatiquement et comment on peut les utiliser pour prouver des programmes utilisant structures de données mutables. On souhaite ensuite prouver automatiquement les obligations de preuve contenant ces prédicats. Nous avons utilisée une logique polymorphe car elle permet une plus grande souplesse et plus de concision dans les spécifications et axiomatiques. Cependant de nombreux prouveurs ne connaissent pas cette logique. Les obligations de preuve en logique polymorphe sont alors encodées dans leurs propre logique. Cela ajoute du bruit dans les obligations de preuve rendant plus difficile leurs preuves automatiques. Dans la partie suivante nous allons voir comment on peut réduire ce bruit en utilisant de nouveaux encodages.

```
node[..] rev(node[..] p_0)
  requires (C_18 : list{Here}(p_0));
  ensures (C_19 : list{Here}(\result)
                  && list_framed{Old,Here}(p_0,\result));
{
  var node[..] q_0;
  var node[..] tmp;
  var int32 tmp_0;
  C_2 : (q_0 = null);
    {loop
       invariant (C_3 :
         list{Here}(p_0) &&
         list{Here}(q_0) &&
         sep_list_list{Here}(p_0, q_0) &&
         list_framed{C_2,Here}(\at(p_0,C_2),q_0) &&
         list_sub{C_2,Here}(\at(p_0,C_2),p_0));
         while (true)
         {
            (if (p_0 != null) then () else goto while_0_break);
            tmp = (C_10 : p_0.tl);
            p_0.tl = q_0;
            q_0 = p_0;
            p_0 = tmp;
         };
         (while_0_break : ())
      };
      (return q_0)
}

node[..] foo(node[..] p_0, node[..] q_0)
  requires (C_18 : list{Here}(p_0) && list{Here}(q_0)
                   && sep_list_list{Here}(p_0, q_0));
behavior default:
  ensures (C_19 : list{Here}(\result) && list{Here}(q_0)
                  && sep_list_list{Here}(\result, q_0));
{
   return rev(p_0);
}
```

FIGURE 4.1.: Définition de l'inversion de liste en place en `Jessie` et utilisation.

Deuxième partie

Élimination du polymorphisme

5. Logique Polymorphe

En premier lieu nous allons définir notre logique polymorphe. Nous verrons ensuite l'impossibilité d'encoder cette logique polymorphe vers la logique multi-sortée en se limitant à de l'instanciation. Ensuite nous verrons différents encodages, certains originaux, d'autres classiques. Enfin nous les comparerons et discuterons de leurs performances relatives.

5.1. Définition de la logique polymorphe

La logique $\mathbf{FOL}_{\mathrm{T}}$, présentée ci-après, est une extension de la logique classique du premier ordre multi-sortée. Dans $\mathbf{FOL}_{\mathrm{T}}$, les types sont construits à partir de constantes de types (comme "entier"), de constructeurs de types (comme "liste de") et de variables de types. Le rôle des variables de type est d'être instanciées par des types arbitraires. Une formule polymorphe représente une famille potentiellement infinie de formules sans variables de type obtenues en remplacant chaque variable de type de la formule polymorphe par tous les types possibles.

5.1.1. Notation

Types Les types sont construits à partir de *variables de types* (notées par α, β, γ) et de *constructeurs de types* d'arité fixée (dénotés par des capitales sans-serif).

$$T ::= \alpha | \mathsf{M}(T, \ldots, T)$$

Par exemple, β, I, $\mathsf{F}(\mathsf{I}, \gamma)$ sont des types bien formés.

La différence principale entre la logique polymorphe et la logique multi-sortée est la présence de variables de type dans les formules. Certaines logiques multi-sortées intègrent des constructeurs de types pour des théories prédéfinies ainsi que des symboles logiques surchargés (par exemple la théorie des tableaux dans le prouveur Z3 ou CVC3). Cependant, dans de telles logiques, l'utilisateur ne peut pas définir ses propres fonctions ou prédicats polymorphes.

On appellera un constructeur de type d'arité zero *type constant*. On appellera un type qui ne contient pas de variables de type *type monomorphe* ou *sorte*. On appellera un vecteur de types $\langle T_1, \ldots, T_n \rangle$ *signature de type*.

On indique l'ensemble des variables de types par $\mathbb{V}_{\mathrm{T}}$, l'ensemble des constructeurs de types par $\mathtt{FT}$, l'ensemble des types construits à partir de $\mathbb{V}_{\mathrm{T}}$ et $\mathtt{FT}$ par $\mathcal{T}(\mathtt{FT}, \mathbb{V}_{\mathrm{T}})$ et l'ensemble de toutes les sortes par $\mathcal{T}(\mathtt{FT})$. On suppose que $\mathbb{V}_{\mathrm{T}}$ est infini et que $\mathtt{FT}$ contient au moins un type constant.

On définit les *substitutions de types* et les *correspondances de types* (matching) comme elles sont habituellement définies sur les termes. Un type T correspond à un type T' si $T = T'\tau$ avec τ une substitution de type.

On utilise les lettres S et T pour les types, les lettres en caractères gras $\mathbf{S}$, $\mathbf{T}$ pour les signatures de type, et les lettres grecques τ, θ, π pour les subtitutions de types.

Termes et formules On utilise la notion traditionnelle de termes et formules du premier ordre formés à partir de symboles de variables (indiqués par u, v, w), de symboles de fonctions (indiqués par f, g, h), et de symboles de prédicats (indiqués par p, q), avec les ajouts suivants :

- Tous les termes, qu'ils s'agissent de variables ou d'applications de fonctions, sont annotés par leur type, par exemple : $w : \mathsf{C}(\mathsf{I})$, $w : \mathsf{C}(\alpha)$, $f(u : \alpha, v : \mathsf{L}(\alpha)) : \mathsf{L}(\alpha)$. Les deux premiers termes sont considérés comme différents même s'ils partagent le même symbole de variable. On indique les termes avec les lettres s et t, et par abus de notation, on écrit parfois le type du terme à droite de la lettre : $s : T_1$, $t : T_2$.
- À chaque symbole de fonction d'arité n on associe une signature de longueur $n+1$. Un terme de la forme $f(t_1 : T_1, \ldots, t_n : T_n) : T$ est bien formé si et seulement si la signature de f correspond à $\langle T_1, \ldots, T_n, T\rangle$.
- À chaque symbole de prédicat d'arité n on associe une signature de longueur n. Un terme de la forme $p(t_1 : T_1, \ldots, t_n : T_n)$ est bien formé si et seulement si la signature de p correspond à $\langle T_1, \ldots, T_n\rangle$.
- Les quantificateurs lient les variables, c'est-à-dire les symboles de variables typés : $\forall(u : \alpha)\, p(u : \alpha, u : \mathsf{C})$. Dans cet exemple, le premier argument de p est lié mais le second est libre.

On considère l'égalité ($\approx$), la conjonction ($\wedge$) et les quantificateurs universels ($\forall$) comme symboles logiques. On utilise ces constructions pour définir de manière habituelle la disjonction ($\vee$), l'implication ($\Rightarrow$), l'équivalence ($\Leftrightarrow$), la diségalité ($\not\approx$), et la quantification existentielle ($\exists$).

On utilise les lettres x, y, z pour les variables, les lettres F, G, H pour les formules, et les lettres grecques Γ, Δ pour les ensembles de formules.

On indique l'ensemble des symboles de variables par $\mathbb{V}$, l'ensemble des symboles de fonctions par $\mathbb{F}$ et l'ensemble des symboles de prédicats par $\mathbb{P}$. On suppose que les ensembles $\mathbb{V}_\mathrm{T}$, $\mathtt{FT}$, $\mathbb{V}$, $\mathbb{F}$, $\mathbb{P}$ sont disjoints deux à deux.

Soit un terme ou une formule e. L'ensemble des variables de type apparaissant dans e est noté $\mathrm{FV}_\mathrm{T}(e)$ et l'ensemble des variables libres est noté $\mathrm{FV}(e)$. Si $\mathrm{FV}_\mathrm{T}(e)$ est vide, e est dit *monomorphe*. Si $\mathrm{FV}(e)$ est vide, e est dit *clos*.

Toutes les substitutions que l'on va appliquer à des termes ou des formules, remplacent des variables de termes libres par des termes du même type. On indique les substitutions par σ et δ. On ne va appliquer des substitutions de types qu'à des formules closes ; elles remplaceront tous les symboles de variables liées en nouveaux symboles de variables fraîches. De cette manière, on évite les collisions de variables : par exemple, la substitution de type $[\mathsf{I}/\alpha]$ n'instanciera pas $\forall(u : \alpha)\forall(u : \mathsf{I})\, p(u : \alpha, u : \mathsf{I})$ en $\forall(u : \mathsf{I})\forall(u : \mathsf{I})\, p(u : \mathsf{I}, u : \mathsf{I})$, mais en $\forall(u' : \mathsf{I})\forall(u'' : \mathsf{I})\, p(u' : \mathsf{I}, u'' : \mathsf{I})$.

L'exemple suivant est utilisé tout au long de ce chapitre pour illustrer le propos. Il est assez simple pour ne pas trop interférer avec les explications. Certaines annotations de type ont été omises pour une meilleure lecture.

$$\forall(m : \mathsf{M}(\alpha, \mathsf{I}))\, \forall(c : \alpha)\, \mathtt{get}(\mathtt{set}(m, c, 6) : \mathsf{M}(\alpha, \mathsf{I}), c) : \mathsf{I} * 7 \approx 42. \tag{5.1}$$

Le constructeur de type I représente les entiers. Le type $\mathsf{M}(\alpha, \beta)$ représente les tables d'association polymorphe du type α vers le type β. Dans la dernière partie de ce chapitre (p.108) on peut voir une comparaison de certains encodages sur un exemple tiré des prédicats de séparation discutés dans la partie précédente.

5.1.2. Satisfiabilité

Soient des ensembles $\mathtt{FT}$, $\mathbb{F}$, $\mathbb{P}$. Une interprétation $\mathfrak{I}$ est définie par trois tables d'association :

- pour chaque sorte $S \in \mathcal{T}(\mathtt{FT})$, on associe un *domaine* non vide $\mathcal{D}^{\mathfrak{I}}_S$;
- pour chaque symbole $f \in \mathbb{F}$ et chaque vecteur de sortes $\mathbf{S} = \langle S_1, \ldots, S_n, S\rangle$ compatible avec la signature de type de f, on associe une fonction $f^{\mathfrak{I}}_{\mathbf{S}} : \mathcal{D}^{\mathfrak{I}}_{S_1} \times \cdots \times \mathcal{D}^{\mathfrak{I}}_{S_n} \to \mathcal{D}^{\mathfrak{I}}_S$;
- pour chaque symbole $p \in \mathbb{P}$ et chaque vecteur de sortes $\mathbf{S} = \langle S_1, \ldots, S_n\rangle$ compatible avec la signature de type de p, on associe une fonction $p^{\mathfrak{I}}_{\mathbf{S}} : \mathcal{D}^{\mathfrak{I}}_{S_1} \times \cdots \times \mathcal{D}^{\mathfrak{I}}_{S_n} \to \{\top, \bot\}$, où $\top$ et $\bot$ sont les constantes booléennes "true" et "false", respectivement.

Dans la suite on se donne une interprétation $\mathfrak{I}$.

On appelle *valuation de types* une substitution qui instancie chaque variable de type dans $\mathbb{V}_{\mathrm{T}}$ par une sorte. Pour une valuation π de types donnée, on appelle *valuation de variables dépendant de* π une fonction qui associe chaque variable de $u : T$ à un élément de $\mathcal{D}^{\mathfrak{I}}_{T\pi}$. On dit simplement *valuation de variables*, ou *valuation*, quand le contexte indique clairement la valuation de type référencée. Dans un contexte monomorphe où tous les types sont déjà clos la valuation des variables est indépendante de toute valuation de type et on retrouve alors les valuations de variables comme définies dans la section 3.1.2.

Soit π une valuation de types et ψ une valuation de variables dépendant de π. On définit l'évaluation des termes et formules de la manière suivante :

$$\begin{aligned}
\mathfrak{I}_{\pi,\psi}(u : T) &\triangleq \psi(u : T) & \mathfrak{I}_{\pi,\psi}(t_1 \approx t_2) &\triangleq (\mathfrak{I}_{\pi,\psi}(t_1) = \mathfrak{I}_{\pi,\psi}(t_2)) \\
\mathfrak{I}_{\pi,\psi}(f(\mathbf{t} : \mathbf{T})) : T) &\triangleq f^{\mathfrak{I}}_{\langle \mathbf{T}, T\rangle\pi}(\mathfrak{I}_{\pi,\psi}(\mathbf{t})) & \mathfrak{I}_{\pi,\psi}(\neg F) &\triangleq \neg\, \mathfrak{I}_{\pi,\psi}(F) \\
\mathfrak{I}_{\pi,\psi}(p(\mathbf{t} : \mathbf{T})) &\triangleq p^{\mathfrak{I}}_{\mathbf{T}\pi}(\mathfrak{I}_{\pi,\psi}(\mathbf{t})) & \mathfrak{I}_{\pi,\psi}(F \wedge G) &\triangleq \mathfrak{I}_{\pi,\psi}(F) \wedge \mathfrak{I}_{\pi,\psi}(G) \\
& & \mathfrak{I}_{\pi,\psi}(\forall(u : T)F) &\triangleq \bigwedge_{a \in \mathcal{D}^{\mathfrak{I}}_{T\pi}} \mathfrak{I}_{\pi,\psi[u : T \mapsto a]}(F)
\end{aligned}$$

où $\mathbf{t} : \mathbf{T}$ dénote un vecteur de termes $t_1 : T_1, \ldots, t_n : T_n$ et $\psi[u : T \mapsto a]$ est la valuation qui coïncide avec ψ partout excepté en $u : T$ qui est associé à a.

Il est clair que l'évaluation d'un terme ou d'une formule e par $\mathfrak{I}_{\pi,\psi}$ ne dépend que des valeurs de π et ψ associées aux variables (de types) qui apparaissent dans e.

Lemme 5.1.1. *Soit F une formule et $\mathfrak{I}$ une interprétation. Soient π_1 et π_2 deux valuations de types qui coïncident sur toutes les variables de type de $\mathrm{FV_T}(F)$. Soit ψ_1 et ψ_2 deux valuations, dépendant respectivement de π_1 et π_2, qui coïncident sur toutes les variables de $\mathrm{FV}(F)$. On a alors $\mathfrak{I}_{\pi_1,\psi_1}(F) = \mathfrak{I}_{\pi_2,\psi_2}(F)$.*

Ainsi dans ce qui suivra lors de l'évaluation de formules closes (respectivement monomorphes) on omettra souvent la valuation de variables (respectivement de types).

Lemme 5.1.2. *Soit F une formule close et $\mathfrak{I}$ une interprétation. Pour toute valuation de types π, $\mathfrak{I}_\pi(F) = \mathfrak{I}(F\pi)$.*

Comme indiqué précédemment, nous traitons les variables de types comme implicitement universellement quantifiées en tête des formules polymorphes. C'est pourquoi, on dit qu'une formule close F est *satisfaite* par $\mathfrak{I}$ si et seulement si $\mathfrak{I}_\pi(F)$ est vrai pour toutes les valuations de types π. Une formule close est *satisfiable* si et seulement si elle est satisfaite par une interprétation, appelée le *modèle* de cette formule. Ces définitions sont naturellement étendues à des ensembles de formules closes.

Pour prouver une formule polymorphe G dans un contexte polymorphe Γ, on prouve, comme dans le cas de la logique monomorphe, l'incohérence du contexte lorsqu'on lui ajoute la négation du but. Ici on effectue la négation de G avec ses quantifications universelles implicites et, comme il n'y a pas de quantification existentielle, on les skolemise. Ainsi on prend une substitution de type τ qui remplace toutes les variables de type dans G par des constantes fraîches de type et on montre que l'ensemble $\Gamma, \neg G\tau$ est insatisfiable. Par exemple pour essayer de prouver la validité du but 5.1 on pourra essayer de prouver l'insatisfiabilité de la formule suivante :

$$\exists(m : \mathsf{M}(\mathsf{Fresh}, \mathsf{I}))\, \exists(c : \alpha)\, \mathtt{get}(\mathtt{set}(m, c, 6) : \mathsf{M}(\mathsf{Fresh}, \mathsf{I}), c) : \mathsf{I} * 7 \not\approx 42.$$

dans une signature comprenant la variable de type Fresh en plus.

On définit la logique multi-sortée comme la restriction de la logique polymorphe aux termes et formules monomorphes. Cela coïncident bien avec la logique multi-sortée traditionnelle avec sortes disjointes [42]. De plus on voit par le lemme 5.1.2 qu'une formule polymorphe représente bien une famille possiblement infinie de formules monomorphes. De manière plus précise on dénote l'ensemble des instances de type monomorphes d'une formule close F par $\mathrm{MI}(F)$. De manière similaire à partir d'un ensemble de formules closes Γ, l'ensemble $\mathrm{MI}(\Gamma)$ contient toutes les instances monomorphes de type des formules de Γ. Un corollaire trivial du lemme 5.1.2 est que Γ est satisfiable dans $\mathbf{FOL}_\mathrm{T}$ si et seulement si $\mathrm{MI}(\Gamma)$ l'est dans la logique multi-sortée.

5.2. Problème de décision des types clos requis

Dès qu'une formule F n'est pas monomorphe et qu'il y a au moins un constructeur de type non-constant, F possède une infinité d'instances monomorphes de type. Pouvons-nous trouver, en un temps fini, toutes les sortes pertinentes pour un problème donné et ainsi travailler avec le sous-ensemble fini des instances les faisant intervenir ?

D'une part cela ressemble au calcul de toutes les instances pertinentes d'un ensemble de formules du premier ordre, qui est un problème bien connu comme indécidable [56, 18]. D'autre part cela n'a pas besoin d'être aussi difficile que la recherche de preuve en général, les monomorphisations complètes sont souvent possibles dans des langages de programmation (par exemple les templates de C++).

Théorème 5.2.1. *Il n'y a pas de fonction calculable qui associe à un ensemble de formules closes arbitraires Δ un ensemble fini et équisatisfiable d'instanciations de type monomorphe de Δ.*

Démonstration. On appellera ici une telle fonction *fonction parfaite d'instanciation.*

Il se trouve que notre système de type est assez expressif pour encoder une logique indécidable, la logique combinatoire [22]. Considérons la signature suivante :

$$\mathtt{FT} = \{\, \mathsf{A}(\cdot,\cdot),\ \mathsf{S},\ \mathsf{K} \,\} \qquad \mathbb{F} = \{\, \mathtt{A} : \langle \alpha, \beta, \mathsf{A}(\alpha,\beta)\rangle,\ \mathtt{S} : \langle \mathsf{S}\rangle,\ \mathtt{K} : \langle \mathsf{K}\rangle \,\} \qquad \mathbb{P} = \{\, \mathtt{R} : \langle \alpha, \beta\rangle \,\}$$

avec ces cinq axiomes (par clarté, on omet quelques annotations de types) :

$$\forall (u:\alpha)\,\forall (v:\beta)\,\forall (w:\gamma)\ ((\mathtt{R}(u,v) \wedge \mathtt{R}(v,w)) \Rightarrow \mathtt{R}(u,w))$$

$$\forall (u:\alpha)\,\forall (v:\beta)\,\forall (w:\gamma)\ (\mathtt{R}(u,v) \Rightarrow \mathtt{R}(\mathtt{A}(u,w) : \mathsf{A}(\alpha,\gamma), \mathtt{A}(v,w) : \mathsf{A}(\beta,\gamma)))$$

$$\forall (u:\alpha)\,\forall (v:\beta)\,\forall (w:\gamma)\ (\mathtt{R}(u,v) \Rightarrow \mathtt{R}(\mathtt{A}(w,u) : \mathsf{A}(\gamma,\alpha), \mathtt{A}(w,v) : \mathsf{A}(\gamma,\beta)))$$

$$\forall (u:\alpha)\,\forall (v:\beta)\ \mathtt{R}(\mathtt{A}(\mathtt{A}(\mathtt{K},u),v) : \mathsf{A}(\mathsf{A}(\mathsf{K},\alpha),\beta),\ u:\alpha)$$

$$\forall (u:\alpha)\,\forall (v:\beta)\,\forall (w:\gamma)\ \mathtt{R}(\mathtt{A}(\mathtt{A}(\mathtt{A}(\mathtt{S},u),v),w) : \mathsf{A}(\mathsf{A}(\mathsf{A}(\mathsf{S},\alpha),\beta),\gamma),\ \mathtt{A}(\mathtt{A}(u,w),\mathtt{A}(v,w)) : \mathsf{A}(\mathsf{A}(\alpha,\gamma),\mathsf{A}(\beta,\gamma)))$$

Le symbole de fonction binaire $\mathtt{A}$ correspond à l'application de termes et le symbole de prédicat binaire $\mathtt{R}$ à la LC-réduction. Le problème de LC-réduction consiste à décider si un terme peut se réduire en un autre par les règles de la logique combinatoire. On peut remarquer que tous les termes clos sont reflétés dans leur type. Ce qui implique le point clé de la démonstration : toute formule monomorphe peut être associée à une formule close monomorphe sans quantificateur qui est interprétée de la même manière par toute interprétation. On passe de l'instanciation de types à l'instanciation de termes.

Ainsi on peut réduire le problème de LC-réduction.

Soient U,V deux termes de la logique combinatoire sous forme SK,
Est-ce que U se réduit en V ?

dans le problème suivant :

Soient deux termes U et V écrits dans $\mathbb{F}$ et bien typés.
Est-ce que $\neg\mathtt{R}(U,V)$ avec les cinq axiomes précédents est insatisfiable ?

Si l'on possède une fonction parfaite d'instanciation le problème précédent est équivalent au problème suivant. :

Soient deux termes U et V écrits dans $\mathbb{F}$ et bien typés.
Est-ce que la monomorphisation de $\neg \mathtt{R}(U, V)$ avec les cinq axiomes précédents est insatisfiable ?

Ce qui est décidable puisque l'ensemble après monomorphisation peut être associé à un ensemble de formules closes sans quantificateur. Or le problème de LC-réduction ne l'est pas. Donc une telle fonction parfaite d'instanciation n'existe pas.

□

Cet exemple montre bien que les types peuvent contenir autant d'information que les termes. Il est donc nécessaire de les découvrir durant la preuve. Dans le prouveur `Alt-Ergo` les techniques traitant le premier ordre, et particulièrement les quantificateurs universelles, ne font pas de différence entre les variables de types et les variables de termes [9].

Bien que ce résultat indique que des instanciations ne peuvent suffire pour trouver une formulation équisatisfiable et monomorphe d'un problème, nous verrons qu'il est très efficace de l'utiliser de manière partielle en conjonction des méthodes d'encodages.

6. Élimination du polymorphisme

Les premières procédures de décisions décrites sont naturellement apparues sur des logiques simples qui ne contenaient que peu de concepts, comme par exemple l'arithmétique de Presburger [50]. Outre le fait que choisir des logiques plus complexes mène rapidement à l'indécidabilité, résoudre un problème compliqué implique de résoudre tous les problèmes plus simples qui le composent. À l'inverse, les logiques choisies pour représenter les mathématiques formellement sont bien plus complexes, de manière à offrir une plus grande expressivité à l'utilisateur. Ainsi lorsque l'on veut tenter de prouver automatiquement un problème mathématique à l'aide de démonstrateurs automatiques, on doit franchir le fossé entre logique riche et logique plus simple. En particulier l'immense majorité des démonstrateurs automatiques ne considère que des logiques non typées ou simplement typées. Les logiques offrant plus d'expressivité à l'utilisateur possèdent un système de types riches, c'est-à-dire un système de classement des objets plus précis. On s'intéresse ici aux techniques pour passer d'un système de types riches à un système de types plus simple.

Exemple : on veut convertir le texte suivant, utilisant une logique multi-sortée, vers une logique non-sortée :

$$\forall x : \texttt{unit}.x \approx ()$$
$$\texttt{true} \not\approx \texttt{false}$$
$$\forall x : \texttt{bool}.x \approx \texttt{true} \vee x \approx \texttt{false}$$
$$\forall y : \mathsf{I}.\exists x : \mathsf{I}.y \leq x + 1$$

Cet ensemble de formules est satisfiable.

Une idée naïve de traduction consiste à oublier tous les types, ou, de manière équivalente, à remplacer tous les types par un seul type, dit universel :

$$\forall x.x \approx ()$$
$$\texttt{true} \not\approx \texttt{false}$$
$$\forall x.x \approx \texttt{true} \vee x \approx \texttt{false}$$
$$\forall y.\exists x.y \leq x + 1$$

Cependant les deux premières formules sont alors insatisfiables et donc la transformation n'est pas correcte.

Cette transformation a toutefois été utilisée dans un contexte où les démonstrateurs produisent des preuves qui sont ensuite vérifiées dans la logique initiale. Hurd [32, 33] l'a ainsi utilisée pour prouver des formules d'une logique riche comme HOL [48], en utilisant

des démonstrateurs sur la logique du premier ordre non-sortée, en oubliant simplement les types.

Dans le cas où on doit faire confiance aux démonstrateurs, cette technique ne peut être utilisée. Le typage ajoutant des informations supplémentaires, il a été proposé de les traduire à l'aide de prédicats ou fonctions de typage [44]. Exemple : on introduit un symbole de fonction `typeof` et des constantes `unit`,`bool`, I qui représentent les types correspondants sous forme de termes :

$$\forall x.\texttt{typeof}(x) = \texttt{unit} \Rightarrow x = ()$$
$$\texttt{true} \not\approx \texttt{false}$$
$$\forall x.\texttt{typeof}(x) = \texttt{bool} \Rightarrow x = \texttt{true} \vee x = \texttt{false}$$
$$\forall y.\exists x.\texttt{typeof}(x) = \mathsf{I} \Rightarrow y \leq x + 1$$
$$\texttt{typeof}(1) = \mathsf{I}$$
$$\forall x.\forall y.\texttt{typeof}(x) = \mathsf{I} \Rightarrow \texttt{typeof}(y) = \mathsf{I} \Rightarrow \texttt{typeof}(x + y) = \mathsf{I}$$

Cette traduction-là est correcte.

Cependant cette technique, en ajoutant des prédicats ou fonctions de typage en tête de formules, modifie la structure des formules et peut perturber les démonstrateurs automatiques. De plus les démonstrateurs automatiques ont commencé à utiliser un système de type plus riche de manière à différencier les entiers, pour lequels ils ajoutent un raisonnement complet sur l'arithmétique linéaire, des autres valeurs. Ce système de type est appelé multi-sorté. Les encodages précédents ne permettent pas d'utiliser ces nouvelles possibilités de raisonnement sur les entiers puisque tous les types sont plongés dans le type universel. Couchot et Lescuyer [20] ont proposé une méthode pour traiter ces deux problèmes. Ils utilisent la décoration de termes qui avait déjà été esquissée par Hurd [32], à laquelle ils ajoutent un traitement particulier des termes de l'arithmétique. Exemple : les termes qui ne sont pas de l'arithmétique sont décorés vers le type universel U à l'aide de la fonction `deco` :

$$\forall x : \mathsf{U}.\texttt{deco}(\texttt{unit}, x) = \texttt{deco}(\texttt{unit}, ())$$
$$\texttt{deco}(\texttt{bool}, \texttt{true}) \not\approx \texttt{deco}(\texttt{bool}, \texttt{false})$$
$$\forall x : \mathsf{U}.\texttt{deco}(\texttt{bool}, x) = \texttt{deco}(\texttt{bool}, \texttt{true}) \vee \texttt{deco}(\texttt{bool}, x) = \texttt{deco}(\texttt{bool}, \texttt{false})$$

Les termes de l'arithmétique sont traduits directement :

$$\forall y : \mathsf{I}.\exists x : \mathsf{I}.y \leq x + 1$$

Dans le cas où on ajoute du polymorphisme et des constructeurs de type, les différentes techniques précédentes ajoutent des variables de termes pour représenter les variables de types dans les termes qui représentent les types et les constructeurs de type deviennent des symboles de fonctions :

Exemple, la formule suivante :

$$\forall x : \alpha.\forall y : \mathit{list}\,\alpha.\texttt{hd}(\texttt{cons}(x, l)) \approx x$$

devient avec la première méthode :

$$\forall a.\forall x.\forall y.\mathtt{typeof}(x) = a \Rightarrow \mathtt{typeof}(l) = \mathtt{list}(a) \Rightarrow \mathtt{hd}(\mathtt{cons}(x, l)) \approx x$$

et avec des décorations :

$$\forall a : \mathsf{T}.\forall x : \mathsf{U}.\forall y : \mathsf{U}.\mathtt{deco}(a, \mathtt{hd}(\mathtt{deco}(\mathtt{list}(a), \mathtt{cons}(\mathtt{deco}(a, x), \mathtt{deco}(\mathtt{list}(a), l))))) \approx \mathtt{deco}(a, x) \quad (6.1)$$

Pour que des listes d'entiers puissent être traitées, Couchot et Lescuyer ajoutent des symboles de pont ($\mathtt{to_I}$,$\mathtt{from_I}$) et des axiomes de bijection entre le type des entiers et le type universel :

$$\forall x : \mathsf{I}.\mathtt{to_I}(\mathtt{deco}(\mathsf{I}, \mathtt{from_I}(x))) = x$$
$$\forall x : \mathsf{U}.\mathtt{from_I}(\mathtt{to_I}(\mathtt{deco}(\mathsf{I}, x))) = x$$

Couchot et Lescuyer ont ainsi montré que l'on pouvait préserver le type des entiers de manière à utiliser la théorie de l'arithmétique prédéfinie des démonstrateurs automatiques. Puisque les démonstrateurs définissent de plus en plus de théories prédéfinies, on peut donc être intéressé par préserver d'autres types ou symboles de fonctions. De plus la protection d'un type réduit les décorations sur les termes de ce type. Elle peut donc être intéressante même sans théories prédéfinies pour des problèmes qui mentionnent de nombreuses fois ce type. Malheureusement, si l'on applique l'encodage de Couchot et Lescuyer au type `unit`, le deuxième axiome de bijection implique que U ne possède qu'un seul élément. Et donc on aura la même incohérence avec le type `bool`.

Nous allons résumer les concepts principaux de ces approches, ensuite nous donnerons un schéma commun qui étend la protection à plusieurs sortes. Nous verrons également comment protéger des symboles de fonctions particuliers.

L'étape de base consiste à *fusionner* les types. En effet, le polymorphisme permet d'appliquer un même lemme à un nombre non borné de types. Comme se restreindre à un nombre borné de types n'est pas possible, il faut garder la possibilité d'appliquer un même lemme à des valeurs de différents types. On va donc fusionner les types en une seule sorte qui sera dite *universelle*. Les valeurs ne doivent cependant pas perdre totalement leur type, car un axiome monomorphe qui est applicable à des booléens ne peut sans doute pas être appliqué à des entiers. Les variables de type deviennent des variables de termes et les types sont conservés sous forme de termes et reliés à leur valeurs par différents moyens :

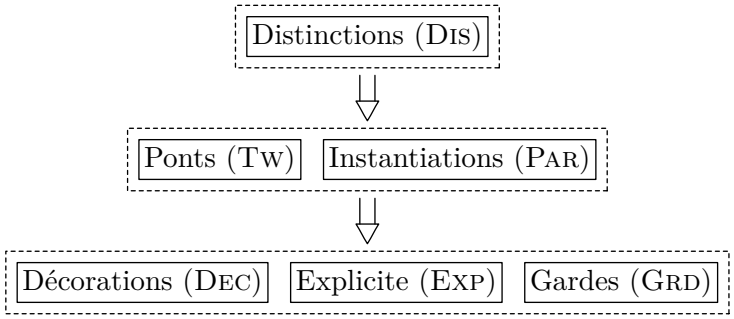
- soit en décorant chaque terme avec son type (Dec) [32, 33] ;
- soit en ajoutant aux symboles de fonctions et prédicats autant d'arguments que leur signature contient de variables de type libres (Exp) [41] ;
- soit en ajoutant des conditions de typage et un axiome pour chaque fonction représentant sa règle de typage (Grd) [42, 41]. Cette technique ne sera pas explicité dans la suite de cette partie. Elle correspond à la première technique correcte décrite.

Les prouveurs multi-sortés définissent des sortes prédéfinies (ex : I) et des fonctions prédéfinies (ex : $+ : \langle \mathsf{I}, \mathsf{I}, \mathsf{I} \rangle$). Ces dernières représentent l'interface avec les théories prédéfinies qui font la puissance des prouveurs *modulo théories*. Il faut donc pouvoir les utiliser et donc empêcher certaines sortes d'être fusionnées. Ces sortes seront dites *protégées*. D'un autre côté il ne faut pas empêcher des valeurs de ces sortes d'être appliquées à des axiomes qui étaient polymorphes. Nous verrons deux manières d'arriver à cela :

- faire des copies (en les instanciant) des axiomes pour chaque sorte que l'on veut protéger (Par) ;
- créer des passages entre les sortes protégées et le type universel (Tw).

On peut ainsi protéger des sortes de départ de manière qu'elles deviennent des sortes particulières à l'arrivée et ainsi des fonctions prédéfinies peuvent être utilisées. Dans le cas où l'on veut protéger des fonctions prédéfinies qui sont surchargées, comme par exemple la fonction de sélection ou de mise à jour de la théorie des tableaux, on ne peut plus simplement protéger une sorte particulière. Dans le texte de départ ces fonctions sont naturellement des fonctions polymorphes. On va donc protéger une instance particulière de cette fonction comme par exemple la sélection sur les tableaux associant des entiers à des booléens. Pour cela on va utiliser une transformation particulière Dis qui ressemble à Par en cela qu'elle instancie les axiomes, mais les axiomes instanciés et les substitutions utilisées dépendent des occurrences des fonctions polymorphes qui apparaissent dans les formules.

Les différentes possibilités d'encodage sont résumées dans le schéma suivant. La protection de sorte est paramétrée par l'ensemble U des sortes à protéger et la protection par distinction est paramétrée par l'ensemble W des instances de fonctions ou prédicats à protéger.



L'interface entre la première étape, distinction, et la seconde, ponts ou instanciation, se fait naturellement. À l'inverse, l'interface entre la seconde et la troisième nécessite de connaître quels termes doivent garder leur type monomorphe lors de l'élimination du polymorphisme. On étend donc notre langage en associant à chaque sorte S dans $\mathcal{T}(\mathtt{FT})$ une *sorte protégée* $\overline{S}$. L'utilisation des sortes protégés est restreinte : une sorte protégée peut apparaître dans la signature d'un symbole ou comme type d'un terme, mais elle ne peut pas apparaître sous un constructeur de type ou dans le codomaine

d'une substitution. En d'autre mots, le seul type qui correspond à une sorte protégée $\overline{S}$ est $\overline{S}$ elle-même.

Par exemple $\mathtt{get}(v : \overline{\mathsf{M}(\mathsf{I},\mathsf{I})}, c : \mathsf{I}) : \mathsf{I}$ est mal formé puisque la signature de type de get est $\langle\ \mathsf{M}(\alpha,\beta), \alpha, \beta\ \rangle$ et $\mathsf{M}(\alpha,\beta)$ ne correspond pas à $\overline{\mathsf{M}(\mathsf{I},\mathsf{I})}$. Similairement $\mathtt{get}(\mathsf{M}(\mathsf{I},\bar{\mathsf{I}}), c : \mathsf{I}) : \bar{\mathsf{I}}$ est mal formé car β ne correspond pas à I et aussi parce que $\mathsf{M}(\mathsf{I},\bar{\mathsf{I}})$ est une expression de type malformé. En revanche, si l'on considère une "spécialisation protégée" de get dénotée $\overline{\mathtt{get}}$, avec la signature de type $\langle\ \overline{\mathsf{M}(\mathsf{I},\mathsf{I})}, \bar{\mathsf{I}} : \bar{\mathsf{I}}\ \rangle$ l'application $\overline{\mathtt{get}}(v : \overline{\mathsf{M}(\mathsf{I},\mathsf{I})}, c : \bar{\mathsf{I}}) : \bar{\mathsf{I}}$ est un terme bien formé.

Quant à l'interprétation, toutes les sortes protégées $\overline{S}$ possèdent leur propre domaine non vide $\mathcal{D}^{\mathfrak{I}}_{\overline{S}}$. De plus, comme pour toute substitution de type, on restreint les valuations de types aux sortes non-protégées. Ainsi l'ensemble $\{\forall(u : \alpha)\forall(v : \alpha)u \approx v, \exists(u : \bar{\mathsf{I}})\exists(v : \bar{\mathsf{I}})u \approx v\}$ est satisfiable. En effet la première formule implique que tous les domaines des sortes non-protégées sont des singletons, mais ne restreint en rien le domaine des sortes protégées.

Nous allons en premier présenter la transformation Dis. Puis nous présenterons deux transformations de protection de sortes, Tw et Par, qui à partir d'un ensemble de formules polymorphes sans sortes protégées donne un ensemble equisatisfiable de formules polymorphes avec protection. Enfin nous présenterons deux transformations de fusion des types non-protégés, Dec et Exp, qui à partir d'un ensemble de formules polymorphes avec des sortes protégées donne un ensemble équisatisfiable de formules monomorphes.

6.1. Distinction

La transformation Dis consiste à produire un nombre suffisant de formules par instanciation de types pour pouvoir ensuite discriminer des symboles de fonctions ou de prédicats sans perdre la complétude.

Soit f un symbole de fonction du problème initial Γ possédant $\mathbf{S}$ comme signature de type. Soit τ une substitution de type. Un symbole de fonction frais f_τ avec la signature de type $\mathbf{S}\tau$ est appelé une *spécialisation* de f. On appelle f_τ une *spécialisation monomorphe* si $\mathbf{S}\tau$ est monomorphe. Les spécialisations de symbole de prédicat sont définies de la même manière.

Soit W un ensemble de spécialisations monomorphes de symboles de fonctions ou de prédicats de Γ. Comme précédemment indiqué, W est l'ensemble des instances que l'on veut faire apparaître. Dis est implicitement paramétré par W.

La transformation Dis modifie la signature de Γ :

1. pour tout symbole de variable u et pour tout type T, on ajoute un nouveau symbole de variable u_T ;
2. on ajoute tous les symboles de fonction et de prédicat de W.

Étant donnée une substitution de type θ, on définit une transformation intermédiaire Dis_θ qui instancie et convertit les termes et formules dans la nouvelle signature :

1. soit une variable $u : T$, $\textsc{Dis}_\theta(u : T) \triangleq u_T : T\theta$;
2. soit un terme $t = f(t_1 : T_1, \ldots, t_n : T_n) : T$. Soit τ la plus petite substitution, c'est-à-dire celle qui est l'identité sur le plus de variable de type possible, qui instancie la signature de type de f à $\langle T_1\theta, \ldots, T_n\theta, T\theta\rangle$. Si f_τ appartient à W, alors $\textsc{Dis}_\theta(t) \triangleq f_\tau(\textsc{Dis}_\theta(t_1) : T_1\theta, \ldots, \textsc{Dis}_\theta(t_n) : T_n\theta) : T\theta$. Sinon on définit $\textsc{Dis}_\theta(t) \triangleq f(\textsc{Dis}_\theta(t_1) : T_1\theta, \ldots, \textsc{Dis}_\theta(t_n) : T_n\theta) : T\theta$;
3. soit une formule atomique $F = p(t_1 : T_1, \ldots, t_n : T_n)$. Soit τ la plus petite substitution qui instancie la signature de type de p à $\langle T_1\theta, \ldots, T_n\theta\rangle$. Si p_τ appartient à W, alors $\textsc{Dis}_\theta(F) \triangleq p_\tau(\textsc{Dis}_\theta(t_1) : T_1\theta, \ldots, \textsc{Dis}_\theta(t_n) : T_n\theta)$. Sinon $\textsc{Dis}_\theta(F) \triangleq p(\textsc{Dis}_\theta(t_1) : T_1\theta, \ldots, \textsc{Dis}_\theta(t_n) : T_n\theta)$.

Les égalités et formules complexes sont converties de manière naturelle :

$$\textsc{Dis}_\theta(t_1 \approx t_2) \triangleq \textsc{Dis}_\theta(t_1) \approx \textsc{Dis}_\theta(t_2) \qquad \textsc{Dis}_\theta(F \wedge G) \triangleq \textsc{Dis}_\theta(F) \wedge \textsc{Dis}_\theta(G)$$
$$\textsc{Dis}_\theta(\neg F) \triangleq \neg\, \textsc{Dis}_\theta(F) \qquad \textsc{Dis}_\theta(\forall x\, F) \triangleq \forall(\textsc{Dis}_\theta(x))\, \textsc{Dis}_\theta(F)$$

Soit F une formule close. On définit l'ensemble des substitutions de types monomorphes $\Theta(F)$ de la manière suivante :

$$\begin{aligned}\Theta(F) \triangleq \{\, \theta \mid\ & F \text{ contient un terme } f(t_1 : T_1, \ldots, t_n : T_n) : T \text{ tel que} \\ & \theta \text{ n'instancie que les variables de } \mathbf{T} = \langle T_1, \ldots, T_n, T\rangle \text{ et} \\ & W \text{ contient une spécialisation de } f \text{ avec la signature de type } \mathbf{T}\theta \,\} \\ \cup \{\, \theta \mid\ & F \text{ contient une formule atomique } p(t_1 : T_1, \ldots, t_n : T_n) \text{ telle que} \\ & \theta \text{ n'instancie que les variables de } \mathbf{T} = \langle T_1, \ldots, T_n\rangle \text{ et} \\ & W \text{ contient une spécialisation de } p \text{ avec la signature de type } \mathbf{T}\theta \,\}\end{aligned}$$

On dit de deux substitutions de types monomorphes qu'elles sont *compatibles* si elles ne substituent pas des sortes différentes pour la même variable de type. *L'union* de deux substitutions de types monomorphes est leur composition (l'ordre n'a ici pas d'importance). On définit $\Theta^\star(F)$ comme la clôture de $\Theta(F)$ par unions finies de substitutions de type compatibles. L'union vide, c'est-à-dire la substitution de type identité, appartient également à $\Theta^\star(F)$.

Finalement Dis convertit une formule close F en un ensemble de formules closes :

$$\textsc{Dis}(F) \triangleq \{\, \textsc{Dis}_\theta(F) \mid \theta \in \Theta^\star(F) \,\}$$

Exemple On choisit de prendre $W = \{\mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}, \mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}\}$. Dis produit alors les deux formules suivantes à partir de la formule 5.1, page 67 :

$$\forall(m_{\mathsf{M}(\alpha,\mathsf{I})} : \mathsf{M}(\alpha,\mathsf{I}))\, \forall(c_\alpha : \alpha)\, \mathtt{get}(\mathtt{set}(m_{\mathsf{M}(\alpha,\mathsf{I})}, c_\alpha, 6), c_\alpha) * 7 \approx 42$$
$$\forall(m_{\mathsf{M}(\alpha,\mathsf{I})} : \mathsf{M}(\mathsf{I},\mathsf{I}))\, \forall(c_\alpha : \mathsf{I})\, \mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(\mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(m_{\mathsf{M}(\alpha,\mathsf{I})}, c_\alpha, 6), c_\alpha) * 7 \approx 42$$

Le nouveau symbole $\mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}$ possède le type monomorphe $\langle \mathsf{M}(\mathsf{I},\mathsf{I}), \mathsf{I}, \mathsf{I}\rangle$.

Lemme 6.1.1. *Soit θ une substitution de type, t un terme de type T et F une formule. Alors* $\textsc{Dis}_\theta(t)$ *est un terme bien formé de type $T\theta$ et* $\textsc{Dis}_\theta(F)$ *est une formule bien formée telle que* $\mathrm{FV}(\textsc{Dis}_\theta(F)) = \{\textsc{Dis}_\theta(x) \mid x \in \mathrm{FV}(F)\}$ *et* $\mathrm{FV_T}(\textsc{Dis}_\theta(F)) = \mathrm{FV_T}(F)\backslash\mathrm{dom}(\theta)$.

Démonstration. La preuve se fait par récurrence sur la structure des termes et des formules. Le seul cas intéressant est l'application de symbole de fonction (l'application de symbole de prédicat se traite de la même façon). Soit t un terme de la forme $f(t_1 : T_1, \ldots, t_n : T_n) : T$. Soit τ la substitution de type qui instancie la signature de type de f vers $\langle T_1\theta, \ldots, T_n\theta, T\theta\rangle$. Pour tous les arguments t_i, $\textsc{Dis}_\theta(t_i)$ est un terme bien formé de type $T_i\theta$ par l'hypothèse de récurrence. Si f_τ appartient à W, alors sa signature est exactement $\langle T_1\theta, \ldots, T_n\theta, T\theta\rangle$. Sinon la signature de type de f s'accorde à $\langle T_1\theta, \ldots, T_n\theta, T\theta\rangle$ à l'aide de τ. □

Théorème 6.1.2 (Correction de Dis)**.** *Si un ensemble de formules closes Γ est satisfiable alors* $\textsc{Dis}(\Gamma)$ *est satisfiable.*

Démonstration. Soit $\mathfrak{M}$ le modèle de Γ. Nous allons construire une interprétation $\mathfrak{I}$ de $\textsc{Dis}(\Gamma)$ telle que, pour toute substitution de type θ, toute valuation de type π, et valuation ψ dans $\mathfrak{I}$, il existe une valuation de type π' et une valuation de variable ψ' dans le modèle initial $\mathfrak{M}$ telles que :

$$\mathfrak{I}_{\pi,\psi}(\textsc{Dis}_\theta(t)) = \mathfrak{M}_{\pi',\psi'}(t) \qquad\qquad \mathfrak{I}_{\pi,\psi}(\textsc{Dis}_\theta(F)) = \mathfrak{M}_{\pi',\psi'}(F) \tag{6.2}$$

pour tout terme t et formule F apparaissant dans Γ. Puisque pour toute formule close $F \in \Gamma$ et pour tout $\theta \in \Theta^\star(F)$, $\mathfrak{I}$ satisfait $\textsc{Dis}_\theta(F)$, $\mathfrak{I}$ est un modèle de $\textsc{Dis}(\Gamma)$.

Pour toute sorte S, on définit le domaine $\mathcal{D}^{\mathfrak{I}}_S$ par $\mathcal{D}^{\mathfrak{M}}_S$. De plus tous les symboles de fonctions et de prédicats dans Γ sont interprétés dans $\mathfrak{I}$ exactement comme dans $\mathfrak{M}$. Tous les nouveaux symboles de fonction f_τ avec la signature $\mathbf{S}$ sont interprétés dans $\textsc{Dis}(\Gamma)$ comme f sur $\mathbf{S}$ dans Γ. Les nouveaux symboles de prédicats sont interprétés de la même façon.

Considérons maintenant une valuation de type π, une valuation ψ de $\mathfrak{I}$, et une substitution de type θ. On définit π' par $\theta \circ \pi$ et $\psi'(u : T)$ par $\psi(u_T : T\theta)$. Remarquons que pour tout type T dans Γ, $\mathcal{D}^{\mathfrak{I}}_{T\theta\pi} = \mathcal{D}^{\mathfrak{M}}_{T\theta\pi} = \mathcal{D}^{\mathfrak{M}}_{T\pi'}$.

En utilisant ces définitions, on prouve les deux égalités précédentes 6.2 par induction sur la structure des termes et des formules. On considère ici trois cas : les variables, les applications de symboles de fonctions, et les formules universellement quantifiées (les autres cas doivent être traités de la même manière).

1. Soit une variable $u : T$ apparaissant dans Γ. On obtient :

$$\mathfrak{I}_{\pi,\psi}(\textsc{Dis}_\theta(u : T)) = \psi(u_T : T\theta) = \psi'(u : T) = \mathfrak{M}_{\pi',\psi'}(u : T).$$

2. Soit un terme $t = f(t_1 : T_1, \ldots, t_n : T_n) : T$, posons $\mathbf{T}$ pour $\langle T_1, \ldots, T_n, T\rangle$. Que W contienne ou non une spécialisation de f avec la signature de type $\mathbf{T}\theta$, l'application de la fonction qui en résulte est interprétée dans $\mathfrak{I}$ exactement comme f sur $\mathbf{T}\theta\pi = \mathbf{T}\pi'$ dans $\mathfrak{M}$. Donc $\mathfrak{I}_{\pi,\psi}(\textsc{Dis}_\theta(t)) = \mathfrak{M}_{\pi',\psi'}(t)$.

3. Soit une formule universellement quantifiée $\forall(u:T)\,F$. Comme nous avons vu précédemment, $\mathcal{D}^{\mathfrak{I}}_{T\theta\pi} = \mathcal{D}^{\mathfrak{M}}_{T\pi'}$. Donc, pour tout élément $a \in \mathcal{D}^{\mathfrak{I}}_{T\theta\pi}$, la valuation $\psi[u_T : T\theta \mapsto a]$ dans $\mathfrak{I}$ devient la valuation $\psi'[u:T \mapsto a]$ dans $\mathfrak{M}$. Donc :

$$\begin{aligned}\mathfrak{I}_{\pi,\psi}(\textsc{Dis}_\theta(\forall(u:T)\,F)) &= \bigwedge_{a\in\mathcal{D}^{\mathfrak{I}}_{T\theta\pi}} \mathfrak{I}_{\pi,\psi[u_T:T\theta\mapsto a]}(\textsc{Dis}_\theta(F)) \\ &= \bigwedge_{a\in\mathcal{D}^{\mathfrak{M}}_{T\pi'}} \mathfrak{M}_{\pi',\psi'[u:T\mapsto a]}(F) = \mathfrak{M}_{\pi',\psi'}(\forall(u:T)\,F)\end{aligned}$$

Ainsi, pour toutes les formules F dans Γ, $\textsc{Dis}(F)$ est satisfait par $\mathfrak{I}$. □

On peut remarquer que cette preuve ne dépend d'aucune propriété sur $\Theta^\star(\cdot)$.

Lemme 6.1.3. *Pour toute formule close F et pour toute valuation de type π, il existe une unique substitution de type maximale (par rapport à l'ensemble des variables de type instanciées) dans $\Theta^\star(F)$ compatible (ici elle est même incluse dans) π.*

Démonstration. Considérons l'ensemble S de toutes les substitutions de type dans $\Theta^\star(F)$ inclus dans π. Cet ensemble est nécessairement fini, puisque F contient un nombre fini de variables de type. Les substitutions de type dans S sont deux à deux compatibles. Donc, S est un semi-treillis complet majoré et son plus grand élément, qui est la substitution que l'on cherche, est inclus dans π. □

Théorème 6.1.4 (Complétude de Dis)**.** *Pour tout ensemble Γ de formules closes, si $\textsc{Dis}(\Gamma)$ est satisfiable alors Γ est satisfiable.*

Démonstration. Soit $\mathfrak{M}$ un modèle de $\textsc{Dis}(\Gamma)$. Nous allons construire une interprétation $\mathfrak{I}$ de Γ telle que pour toute formule close H dans Γ, pour toute valuation de type π, et pour toute valuation ψ dans $\mathfrak{I}$, il existe une substitution de type $\theta \in \Theta^\star(H)$, une valuation de type π', et une valuation ψ' dans le modèle $\mathfrak{M}$ tel que :

$$\mathfrak{I}_{\pi,\psi}(t) = \mathfrak{M}_{\pi',\psi'}(\textsc{Dis}_\theta(t)) \qquad\qquad \mathfrak{I}_{\pi,\psi}(F) = \mathfrak{M}_{\pi',\psi'}(\textsc{Dis}_\theta(F))$$

pour tout terme t et formule F apparaissant dans H. Puisque pour tout $\theta \in \Theta^\star(H)$, le modèle $\mathfrak{M}$ satisfait $\textsc{Dis}_\theta(H)$, la formule initiale H est satisfaite par $\mathfrak{I}$.

Pour toute sorte S dans la signature de Γ, on fixe $\mathcal{D}^{\mathfrak{I}}_S \triangleq \mathcal{D}^{\mathfrak{M}}_S$. Considérons un symbole de fonction f dans Γ, un vecteur de sortes $\mathbf{T} = \langle T_1, \ldots, T_n, T\rangle$, et une substitution de type τ qui instancie la signature de type de f en $\mathbf{T}$. Si la spécialisation monomorphe f_τ appartient à W, on définit $f^{\mathfrak{I}}_{\mathbf{T}} \triangleq (f_\tau)^{\mathfrak{M}}_{\mathbf{T}}$. Sinon $f^{\mathfrak{I}}_{\mathbf{T}} \triangleq f^{\mathfrak{M}}_{\mathbf{T}}$. Les symboles de prédicats sont interprétés de la même manière. Cela conclut la définition de $\mathfrak{I}$.

Maintenant considérons une formule H dans Γ, une valuation de type π et une valuation ψ dans $\mathfrak{I}$. Le lemme 6.1.3 indique que $\Theta^\star(H)$ contient une substitution de type monomorphe maximale θ compatible avec π (et donc inclus dans π). On choisit $\pi' \triangleq \pi$, de telle sorte que $\theta \circ \pi' = \pi$. Pour toute variable $u:T$ apparaissant dans H, on définit $\psi'(u_T : T\theta) \triangleq \psi(u:T)$. En fait, pour tout type T dans H, $\mathcal{D}^{\mathfrak{I}}_{T\pi} = \mathcal{D}^{\mathfrak{M}}_{T\pi} = \mathcal{D}^{\mathfrak{M}}_{T\theta\pi'}$. Par le lemme 5.1.1, il est suffisant de définir ψ' sur les variables qui apparaissent dans $\textsc{Dis}(H)$.

Avec ces définitions, nous allons prouver les deux égalités précédentes par récurrence sur la structure des termes et des formules. On considère ici trois cas : les variables, les applications de symboles de fonctions, et les formules universellement quantifiées. Les autres cas doivent être traités de la même manière.

1. Considérons une variable $u : T$ apparaissant dans H. On a :

$$\mathfrak{I}_{\pi,\psi}(u : T) = \psi(u : T) = \psi'(u_T : T\theta) = \mathfrak{M}_{\pi',\psi'}(\textsc{Dis}_\theta(u : T)).$$

2. Considérons un terme $t = f(t_1 : T_1, \ldots, t_n : T_n) : T$ apparaissant dans H. On pose $\mathbf{T}$ pour $\langle T_1, \ldots, T_n, T\rangle$ et soit τ une substitution de type qui instancie la signature de type de f vers $\mathbf{T}\theta$. Supposons que la spécialisation de f_τ appartiennent à W. Alors $\textsc{Dis}_\theta$ remplace f avec f_τ et $\mathbf{T}\pi = \mathbf{T}\theta$. On a alors :

$$\begin{aligned}
\mathfrak{I}_{\pi,\psi}(t) &= f^{\mathfrak{I}}_{\mathbf{T}\pi}(\mathfrak{I}_{\pi,\psi}(t_1), \ldots, \mathfrak{I}_{\pi,\psi}(t_n)) \\
&= (f_\tau)^{\mathfrak{M}}_{\mathbf{T}\theta\pi'}(\mathfrak{M}_{\pi',\psi'}(\textsc{Dis}_\theta(t_1)), \ldots, \mathfrak{M}_{\pi',\psi'}(\textsc{Dis}_\theta(t_n))) \\
&= \mathfrak{M}_{\pi',\psi'}(f_\tau(\textsc{Dis}_\theta(t_1), \ldots, \textsc{Dis}_\theta(t_n))) \\
&= \mathfrak{M}_{\pi',\psi'}(\textsc{Dis}_\theta(t))
\end{aligned}$$

   Maintenant, considérons le contraire, c'est-à-dire que f_τ n'appartient pas à W, tel que $\textsc{Dis}_\theta$ garde le symbole f dans t. On peut alors montrer qu'il n'y a pas de spécialisation monomorphe de f dans W avec la signature de type $\mathbf{T}\pi$. En fait, s'il y en a une, alors la restriction de π aux variables de types de $\mathbf{T}$ appartient à $\Theta(H)$. Cette restriction est compatible avec θ puisqu'elles sont toutes deux incluses dans π. Comme θ est maximale la restriction est aussi incluse dans θ. Cela implique que $\mathbf{T}\pi = \mathbf{T}\theta$, ce qui conduit à une contradiction. On a alors

$$\begin{aligned}
\mathfrak{I}_{\pi,\psi}(t) &= f^{\mathfrak{I}}_{\mathbf{T}\pi}(\mathfrak{I}_{\pi,\psi}(t_1), \ldots, \mathfrak{I}_{\pi,\psi}(t_n)) \\
&= f^{\mathfrak{M}}_{\mathbf{T}\theta\pi'}(\mathfrak{M}_{\pi',\psi'}(\textsc{Dis}_\theta(t_1)), \ldots, \mathfrak{M}_{\pi',\psi'}(\textsc{Dis}_\theta(t_n))) \\
&= \mathfrak{M}_{\pi',\psi'}(f(\textsc{Dis}_\theta(t_1), \ldots, \textsc{Dis}_\theta(t_n))) \\
&= \mathfrak{M}_{\pi',\psi'}(\textsc{Dis}_\theta(t)).
\end{aligned}$$

3. Considérons une formule universellement quantifiée $\forall(u : T)\, F$. Comme nous l'avons vu précédemment, $\mathcal{D}^{\mathfrak{I}}_{T\pi} = \mathcal{D}^{\mathfrak{M}}_{T\theta\pi'}$. Aussi, pour tout $a \in \mathcal{D}^{\mathfrak{M}}_{T\theta\pi'}$, la valuation $\psi[u : T \mapsto a]$ dans $\mathfrak{I}$ devient la valuation $\psi'[u_T : T\theta \mapsto a]$ dans $\mathfrak{M}$. On obtient :

$$\begin{aligned}
\mathfrak{I}_{\pi,\psi}(\forall(u : T)\, F) &= \bigwedge_{a \in \mathcal{D}^{\mathfrak{I}}_{T\pi}} \mathfrak{I}_{\pi,\psi[u : T \mapsto a]}(F) \\
&= \bigwedge_{a \in \mathcal{D}^{\mathfrak{M}}_{T\theta\pi'}} \mathfrak{M}_{\pi',\psi'[u_T : T\theta \mapsto a]}(\textsc{Dis}_\theta(F)) \\
&= \mathfrak{M}_{\pi',\psi'}(\forall(u_T : T\theta)\, \textsc{Dis}_\theta(F)) = \mathfrak{M}_{\pi',\psi'}(\textsc{Dis}_\theta(\forall(u : T)\, F))
\end{aligned}$$

Ainsi, pour toute formule H dans Γ, H est satisfait par $\mathfrak{I}$. □

La définition de Dis peut être généralisée aux cas où W admet des spécialisations polymorphes. On pourrait ainsi séparer les opérations sur les matrices en général (de manière polymorphe en les éléments), qui sont des tableaux associant des entiers à des tableaux, des opérations sur les autres tableaux. Cela requiert simplement que W soit clos par l'unification des signatures de type, de telle sorte que l'on puisse choisir la spécialisation la plus précise durant la discrimination. Par exemple si l'on choisit de distinguer à la fois les tableaux ayant pour clés des entiers $\mathsf{M}(\mathsf{I}, \alpha)$ et les tableaux représentant la fonction caractéristique d'un ensemble $\mathsf{M}(\alpha, \mathsf{B})$, il faut que l'on distingue également les tableaux représentant des ensembles d'entiers $\mathsf{M}(\mathsf{I}, \mathsf{B})$ car les deux précédents peuvent s'appliquer. Si on choisissait arbitrairement on ne serait plus complet. Il faut également que lors de la transformation en elle-même, l'ensemble $\Theta(F)$ soit considéré modulo renommage des variables de type dans la signature des symboles spécialisés. De plus, l'union de deux substitutions sera alors le raffinement commun le plus précis.

Cependant puisque nos transformations visent des prouveurs monomorphes, nous n'avons pas alourdi les définitions et preuves précédentes avec cette généralisation.

6.2. Encodage par instanciation

Il se trouve que l'approche qui vient d'être présentée peut être poussée jusqu'à protéger des sortes contre l'élimination du polymorphisme. On va simplement discriminer toutes les instanciations de fonctions qui peuvent prendre en argument ou renvoyer un terme d'un type à protéger. Nous allons présenter tout de même toute cette transformation car sa description est au final moins complexe que celle de la discrimination. De plus ici on fait apparaître les types d'arguments et types de retour protégés.

Soit U l'ensemble des sortes que l'on veut protéger. L'ensemble U est fixé pour le reste de la section et la transformation Par est implicitement paramétrée par U. À partir d'un ensemble de types polymorphes M, on définit un ensemble de substitutions monomorphes $\Theta(M)$ comme suit :

$$\begin{aligned}\Theta(M) \triangleq \{\, \theta \mid\ & M \text{ contient un type } T \text{ tel que} \\ & \theta \text{ instancie seulement des variables de type de } T \text{ et} \\ & T\theta \text{ appartient à } U \,\}\end{aligned}$$

On définit $\Theta^\star(M)$ comme la clôture de $\Theta(M)$ par union finie comme défini dans la section précédente. On rappelle que la substitution identité appartient toujours à la clôture.

Lemme 6.2.1. *Soit M un ensemble de types polymorphes et π une substitution de type. Alors il existe une unique substitution de type monomorphe $\theta \in \Theta^\star(M)$ telle que :*
- *θ est inclus dans π (c'est-à-dire pour tout α, soit $\alpha\theta = \alpha$ ou $\alpha\theta = \alpha\pi$) ;*
- *pour tout type $T \in M$, $T\pi \in U$ si et seulement si $T\theta \in U$.*

Démonstration. Soit M' l'ensemble des types de M qui appartiennent à U après instanciation par π. Soit θ la restriction de π aux variables de type qui apparaissent dans M'.

Il est aisé de voir que θ est une union finie de substitutions de $\Theta(M)$ qui instancie les types de M' dans U. En effet toutes ses substitutions sont incluses dans π et donc sont compatibles. Maintenant, soit T un type arbitraire dans M. Si $T\pi$ appartient à U, alors $T \in M'$ et $T\theta = T\pi$. Si $T\theta$ appartient à U, alors $T\pi \in U$, aussi, puisque tous les types dans U sont clos et θ est inclus dans π.

Maintenant supposons qu'il y ait une autre substitution monomorphe θ' dans $\Theta^\star(M)$ qui satisfasse les deux conditions. Puisque θ et θ' sont toutes deux incluses dans π, elles ne peuvent pas substituer deux sortes différentes pour la même variable. Ainsi θ' doit coïncider avec θ (et π) sur toutes les variables de M'. Ainsi θ est inclus dans θ'. Si θ' diffère de θ, alors il existe une variable de type α hors de M' instancié par θ'. Par définition de $\Theta(M)$, il doit y avoir un type $T \in M \backslash M'$ tel que $T\theta' \in U$. Cela contredit la seconde condition. □

Pour un type T donné, le type transformé $[T]$ est défini comme suit : si $T \in U$, on choisit $[T] \triangleq \overline{T}$, sinon $[T] \triangleq T$. Le type transformé étant soit un type protégé soit un type il ne pourra apparaître qu'en tant qu'élément de signature, type de terme ou type de variable.

Maintenant, soit un symbole de fonction f de Γ ayant $\langle S_1, \ldots, S_n, S\rangle$ comme signature. Soit τ une substitution de type de $\Theta^\star(\{S_1, \ldots, S_n, S\})$. Un symbole de fonction frais $\overline{f}_\tau$ avec la signature $\langle [S_1\tau], \ldots, [S_n\tau], [S\tau]\rangle$ sera appelé une *spécialisation protégée* de f. Les spécialisations protégées des symboles de prédicats sont définies de la même manière.

L'idée de la transformation est de produire toutes les spécialisations qui font apparaître une sorte protégée dans la signature de type. Par exemple, si U est $\{\mathsf{M}(\mathsf{I},\mathsf{I})\}$ alors le symbole `get` avec la signature de type $\langle \mathsf{M}(\alpha,\beta), \alpha, \beta\rangle$ introduit cinq spécialisations protégées :

$$\begin{aligned}
\texttt{get}_{[]} &: \langle \mathsf{M}(\alpha,\beta), \alpha, \beta\rangle \\
\texttt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]} &: \langle \overline{\mathsf{M}(\mathsf{I},\mathsf{I})}, \mathsf{I}, \mathsf{I}\rangle \\
\texttt{get}_{[\mathsf{M}(\mathsf{I},\mathsf{I})/\alpha]} &: \langle \mathsf{M}(\mathsf{M}(\mathsf{I},\mathsf{I}),\beta), \overline{\mathsf{M}(\mathsf{I},\mathsf{I})}, \beta\rangle \\
\texttt{get}_{[\mathsf{M}(\mathsf{I},\mathsf{I})/\beta]} &: \langle \mathsf{M}(\alpha,\mathsf{M}(\mathsf{I},\mathsf{I})), \alpha, \overline{\mathsf{M}(\mathsf{I},\mathsf{I})}\rangle \\
\texttt{get}_{[\mathsf{M}(\mathsf{I},\mathsf{I})/\alpha,\mathsf{M}(\mathsf{I},\mathsf{I})/\beta]} &: \langle \mathsf{M}(\mathsf{M}(\mathsf{I},\mathsf{I}),\mathsf{M}(\mathsf{I},\mathsf{I})), \overline{\mathsf{M}(\mathsf{I},\mathsf{I})}, \overline{\mathsf{M}(\mathsf{I},\mathsf{I})}\rangle
\end{aligned}$$

La spécialisation $\texttt{get}_{[]}$ n'ajoutera pas d'instanciation supplémentaire, sa présence ne satisfait qu'un souhait d'uniformité.

La transformation PAR modifie la signature des théories comme suit :

1. Pour tout symbole de variable u et type T, on ajoute un nouveau symbole u_T.
2. On remplace toutes les symboles de fonctions et de prédicats par l'ensemble de toutes leurs spécialisations.

Prenons une substitution arbitraire θ, la transformation PAR_θ instancie et convertit termes et formules dans la nouvelle signature :

1. Soit une variable $u : T$, $\text{PAR}_\theta(u : T) \triangleq u_T : [T\theta]$

2. Soit un terme $t = f(t_1 : T_1, \ldots, t_n : T_n) : T$. Par le lemme 6.2.1, il existe une unique spécialisation protégée $\overline{f}_\tau$ dont la signature correspond à $\langle [T_1\theta], \ldots, [T_n\theta], [T\theta] \rangle$. En effet, si $T_i\theta$ est dans U, alors le seul type qui correspond $[T_i\theta]$ est $[T_i\theta]$ lui-même. Alors $\textsc{Par}_\theta(t) \triangleq \overline{f}_\tau(\textsc{Par}_\theta(t_1), \ldots, \textsc{Par}_\theta(t_n)) : [T\theta]$.
3. Considérons une formule atomique $F = p(t_1 : T_1, \ldots, t_n : T_n)$. Par le lemme 6.2.1 il existe une unique spécialisation $\overline{p}_\tau$ dont la signature correspond à $\langle [T_1\theta], \ldots, [T_n\theta] \rangle$. Alors $\textsc{Par}_\theta(F) \triangleq \overline{p}_\tau(\textsc{Par}_\theta(t_1), \ldots, \textsc{Par}_\theta(t_n))$.

Les égalités et les formules complexes sont converties de manière naturelle :

$$\textsc{Par}_\theta(t_1 \approx t_2) \triangleq \textsc{Par}_\theta(t_1) \approx \textsc{Par}_\theta(t_2) \qquad \textsc{Par}_\theta(F \wedge G) \triangleq \textsc{Par}_\theta(F) \wedge \textsc{Par}_\theta(G)$$
$$\textsc{Par}_\theta(\neg F) \triangleq \neg\, \textsc{Par}_\theta(F) \qquad \textsc{Par}_\theta(\forall x\, F) \triangleq \forall(\textsc{Par}_\theta(x))\, \textsc{Par}_\theta(F)$$

Finalement Par transforme une formule close F est un ensemble de formules :

$$\textsc{Par}(F) \triangleq \{\, \textsc{Par}_\theta(F) \mid \theta \in \Theta^\star(M) \,\}$$

où M est l'ensemble des types de tous les termes qui apparaissent dans F.

Exemple. Supposons que l'on prenne $U = \{\mathsf{I}\}$. Par produit alors les deux formules (par $\textsc{Par}_{[]}$ et $\textsc{Par}_{[\mathsf{I}/\alpha]}$, respectivement) :

$$\forall(m_{\mathsf{M}(\alpha,\mathsf{I})} : \mathsf{M}(\alpha, \mathsf{I}))\, \forall(c_\alpha : \alpha)\, \mathtt{get}_{[\mathsf{I}/\beta]}(\mathtt{set}_{[\mathsf{I}/\beta]}(m_{\mathsf{M}(\alpha,\mathsf{I})}, c_\alpha, 6), c_\alpha) * 7 \approx 42$$
$$\forall(m_{\mathsf{M}(\alpha,\mathsf{I})} : \mathsf{M}(\mathsf{I}, \mathsf{I}))\, \forall(c_\alpha : \bar{\mathsf{I}})\, \mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(\mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(m_{\mathsf{M}(\alpha,\mathsf{I})}, c_\alpha, 6), c_\alpha) * 7 \approx 42$$

Le nouveau symbole $\mathtt{get}_{[\mathsf{I}/\beta]}$ possède la signature $\langle \mathsf{M}(\alpha, \mathsf{I}), \alpha, \bar{\mathsf{I}} \rangle$ et $\mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}$ possède la signature de type monomorphe $\langle \mathsf{M}(\mathsf{I}, \mathsf{I}), \bar{\mathsf{I}}, \bar{\mathsf{I}} \rangle$. Ici vu le U et le W choisis précédemment, il ne sert à rien d'appliquer Dis avant Par ; c'est souvent le cas puisqu'ils sont basés sur le même principe.

Lemme 6.2.2. *Soit θ une substitution de type, t un terme de type T, et F une formule. Alors $\textsc{Par}_\theta(t)$ est un terme bien formé de type $[T\theta]$ et $\textsc{Par}_\theta(F)$ est une formule bien formée telle que $\mathrm{FV}(\textsc{Par}_\theta(F)) = \{\textsc{Par}_\theta(x) \mid x \in \mathrm{FV}(F)\}$ et $\mathrm{FV}_\mathrm{T}(\textsc{Par}_\theta(F)) = \mathrm{FV}_\mathrm{T}(F) \backslash \mathrm{dom}(\theta)$.*

Démonstration. La preuve se fait par induction sur la structure des termes et des formules. Le seul point intéressant est l'application des symboles de fonction (les symboles de prédicats sont traités de la même manière). Soit t un terme de la forme $f(t_1 : T_1, \ldots, t_n : T_n) : T$. Pour tout t_i, $\textsc{Par}_\theta(t_i)$ est un terme bien formé de type $[T_i\theta]$ par l'hypothèse d'induction. Par définition, la signature de type de $\overline{f}_\tau$ correspond à $\langle [T_1\theta], \ldots, [T_n\theta], [T\theta] \rangle$. Donc $\textsc{Par}_\theta(t)$ est un terme bien formé de type $[T\theta]$. □

Théorème 6.2.3 (Correction de Par)**.** *Si un ensemble Γ de formules est satisfiable alors $\textsc{Par}(\Gamma)$ est satisfiable.*

Démonstration. Soit $\mathfrak{M}$ un modèle de Γ. Nous allons construire l'interprétation $\mathfrak{I}$ de $\textsc{Par}(\Gamma)$ telle que, pour toute substitution de type θ, valuation de type π, et valuation ψ dans $\mathfrak{I}$, il existe une valuation ψ' dans le modèle initial $\mathfrak{M}$, telle que :

$$\mathfrak{I}_{\pi,\psi}(\textsc{Par}_\theta(t)) = \mathfrak{M}_{\pi',\psi'}(t) \qquad\qquad \mathfrak{I}_{\pi,\psi}(\textsc{Par}_\theta(F)) = \mathfrak{M}_{\pi',\psi'}(F)$$

pour tout terme t et formule F apparaissant dans Γ. Puisque pour toute formule close $F \in \Gamma$ et tout θ, $\mathfrak{I}$ satisfait $\textsc{Par}_\theta(F)$, $\mathfrak{I}$ est un modèle de $\textsc{Par}(\Gamma)$.

Pour toute sorte S dans Γ, on définit $\mathcal{D}^{\mathfrak{I}}_S \triangleq \mathcal{D}^{\mathfrak{M}}_S$ et $\mathcal{D}^{\mathfrak{I}}_{\bar{S}} \triangleq \mathcal{D}^{\mathfrak{M}}_S$. Ensuite on indique l'interprétation des symboles de fonctions et de prédicats dans $\textsc{Par}(\Gamma)$. Considérons un symbole de fonction $\bar{f}_\tau$ qui est une spécialisation protégée du symbole de fonction f de Γ. On dénote la signature de type de f par $\mathbf{S}$, la signature de type de $\bar{f}_\tau$ est $[\mathbf{S}\tau]$. Soit $\mathbf{T}$ un vecteur de sortes tel que $\mathbf{T} = [\mathbf{S}\tau]\tau'$ pour une substitution de types τ'. On interprète alors $\bar{f}_\tau$ sur $\mathbf{T}$ dans $\mathfrak{I}$ exactement comme f interprété sur $\mathbf{S}\tau\tau'$ dans $\mathfrak{M}$. Cela est correct comme $\mathbf{S}\tau\tau'$ est $\mathbf{T}$ avec les protections enlevées. Les symboles de prédicats sont interprétés de la même manière.

Maintenant considérons une substitution de type θ, une valuation de type π, et une valuation ψ dans $\mathfrak{I}$. On choisit $\pi' \triangleq \theta \circ \pi$ et on choisit $\psi'(u:T) \triangleq \psi(u_T:[T\theta])$. Remarquons que pour tout type T dans Γ, $\mathcal{D}^{\mathfrak{I}}_{[T\theta]\pi} = \mathcal{D}^{\mathfrak{M}}_{T\theta\pi} = \mathcal{D}^{\mathfrak{M}}_{T\pi'}$.

En utilisant ces définitions, on peut prouver les deux égalités. La preuve se déroule par induction sur la structure des termes et des formules. On considère trois cas : les variables, les applications de fonction et les formules universellement quantifiées (les autres cas peuvent être traités de la même manière).

1. Soit une variable $u:T$ apparaissant dans Γ. On obtient :

$$\mathfrak{I}_{\pi,\psi}(\textsc{Par}_\theta(u:T)) = \psi(u_T:[T\theta]) = \psi'(u:T) = \mathfrak{M}_{\pi',\psi'}(u:T)$$

2. Soit un terme $t = f(t_1:T_1,\ldots,t_n:T_n):T$. Soit f_τ la spécialisation protégée de f dont la signature correspond à $[\mathbf{T}\theta]$, quand $\mathbf{T} = \langle T_1,\ldots,T_n,T\rangle$.

$$\begin{aligned}
\mathfrak{I}_{\pi,\psi}(\textsc{Par}_\theta(t)) &= \mathfrak{I}_{\pi,\psi}(\bar{f}_\tau(\textsc{Par}_\theta(t_1),\ldots,\textsc{Par}_\theta(t_n)):[T\theta]) \\
&= (f_\tau)^{\mathfrak{I}}_{[\mathbf{T}\theta]\pi}(\mathfrak{I}_{\pi,\psi}(\textsc{Par}_\theta(t_1)),\ldots,\mathfrak{I}_{\pi,\psi}(\textsc{Par}_\theta(t_n))) \\
&= f^{\mathfrak{M}}_{\mathbf{T}\theta\pi}(\mathfrak{M}_{\pi',\psi'}(t_1),\ldots,\mathfrak{M}_{\pi',\psi'}(t_n)) \\
&= f^{\mathfrak{M}}_{\mathbf{T}\pi'}(\mathfrak{M}_{\pi',\psi'}(t_1),\ldots,\mathfrak{M}_{\pi',\psi'}(t_n)) \\
&= \mathfrak{M}_{\pi',\psi'}(t)
\end{aligned}$$

3. Soit une formule universellement quantifiée $\forall(u:T)\,F$. Comme nous avons vu plus haut, $\mathcal{D}^{\mathfrak{I}}_{[T\theta]\pi} = \mathcal{D}^{\mathfrak{M}}_{T\pi'}$. Donc pour tout élément $a \in \mathcal{D}^{\mathfrak{I}}_{[T\theta]\pi}$, la valuation $\psi[u_T:[T\theta] \mapsto a]$ dans $\mathfrak{I}$ produit la valuation $\psi'[u:T \mapsto a]$ dans $\mathfrak{M}$.

$$\begin{aligned}
\mathfrak{I}_{\pi,\psi}(\textsc{Par}_\theta(\forall(u:T)\,F)) &= \bigwedge_{a\in\mathcal{D}^{\mathfrak{I}}_{[T\theta]\pi}} \mathfrak{I}_{\pi,\psi[u_T:[T\theta]\mapsto a]}(\textsc{Par}_\theta(F)) \\
&= \bigwedge_{a\in\mathcal{D}^{\mathfrak{M}}_{T\pi'}} \mathfrak{M}_{\pi',\psi'[u:T\mapsto a]}(F) = \mathfrak{M}_{\pi',\psi'}(\forall(u:T)\,F)
\end{aligned}$$

Ainsi pour toutes formules F de Γ, $\textsc{Par}(F)$ est satisfait dans $\mathfrak{I}$. □

On peut remarquer que cette preuve utilise simplement le fait que l'on a instancié des formules qui étaient dans le problème initial. On pourrait donc moins instancier ou ne pas toujours utiliser la spécialisation protégée, et on serait toujours corrects. La correction est, comme souvent, très libérale, à la différence de la complétude que l'on va maintenant prouver.

Théorème 6.2.4 (Complétude de Par)**.** *Pour tout ensemble de formules closes* Γ*, si* $\textsc{Par}(\Gamma)$ *est satisfiable alors* Γ *est satisfiable.*

Démonstration. Soit $\mathfrak{M}$ un modèle de $\textsc{Par}(\Gamma)$. Nous allons construire une interprétation $\mathfrak{I}$ de Γ telle que pour toute formule H dans Γ, toute valuation de type π, et valuation ψ dans $\mathfrak{I}$, il existe une substitution de type $\theta \in \Theta^\star(\{T \,|\, T \text{ est le type d'un terme dans } H\,\})$, une valuation de type π', et une valuation ψ' dans le modèle initial $\mathfrak{M}$ telles que :

$$\mathfrak{I}_{\pi,\psi}(t) = \mathfrak{M}_{\pi',\psi'}(\textsc{Par}_\theta(t)) \qquad\qquad \mathfrak{I}_{\pi,\psi}(F) = \mathfrak{M}_{\pi',\psi'}(\textsc{Par}_\theta(F))$$

pour tout terme t et formule F apparaissant dans H.

Pour toute sorte S dans la signature de Γ, on choisit $\mathcal{D}^{\mathfrak{I}}_S \triangleq \mathcal{D}^{\mathfrak{M}}_{[S]}$. Considérons un symbole de fonction F dans Γ et un vecteur de sortes $\mathbf{T} = \langle T_1, \ldots, T_n, T\rangle$ qui correspond à la signature de type de f. Par le lemme 6.2.1, il y a une unique spécialisation protégée $\overline{f}_\tau$ dont la signature correspond au vecteur de sorte $[\mathbf{T}]$. Donc on définit $f^{\mathfrak{I}}_{\mathbf{T}} \triangleq (\bar{f}_\tau)^{\mathfrak{M}}_{[\mathbf{T}]}$. Les symboles de prédicat sont interprétés de la même manière.

Maintenant considérons une formule H de Γ, une valuation de type π, et une valuation ψ dans $\mathfrak{I}$. Alors M est l'ensemble des types de tous les termes apparaissant dans H. Par le lemme 6.2.1, il existe une unique substitution de type $\theta \in \Theta^\star(M)$ telle que θ est incluse dans π et pour tout type $T \in M$, $T\pi \in U$ si et seulement si $T\theta \in U$. On choisit $\pi' \triangleq \pi$, ainsi $\theta \circ \pi' = \pi$. Soit une variable $u : T$ apparaissant dans H. S'il apparaît dans H comme un terme, alors $T \in M$ et on choisit $\psi'(u_T :[T\theta]) \triangleq \psi(u : T)$. En fait pour tout type T dans H, $\mathcal{D}^{\mathfrak{I}}_{T\pi} = \mathcal{D}^{\mathfrak{M}}_{[T\pi]} = \mathcal{D}^{\mathfrak{M}}_{[T\theta]\pi'}$ par définition de θ. Sinon, si $u : T$ est seulement lié par des quantificateurs dans H mais n'apparaît pas sous ces quantificateurs, on peut l'ignorer dans l'évaluation de variable par le lemme 5.1.1.

En utilisant ces définitions on peut prouver les deux égalités. La preuve se déroule par induction structurelle sur les termes et les formules. On considère trois cas : les variables, les applications de fonctions, et les formules universellement quantifiées (les autres cas peuvent être traités de la même manière).

1. Considérons une variable $u : T$ apparaissant dans H comme un terme.

$$\mathfrak{I}_{\pi,\psi}(u : T) = \psi(u : T) = \psi'(u_T :[T\theta]) = \mathfrak{M}_{\pi',\psi'}(\textsc{Par}_\theta(u : T))$$

2. Considérons un terme $t = f(t_1 : T_1, \ldots, t_n : T_n) : T$ apparaissant dans H et posons $\mathbf{T}$ pour $\langle T_1, \ldots, T_n, T\rangle$. Soit $\overline{f}_\tau$ la spécialisation protégée de f dont la signature correspond à $[\mathbf{T}\theta]$. Par définition de θ, $[\mathbf{T}\theta]$ correspond à $[\mathbf{T}\pi]$. Alors $\overline{f}_\tau$ est aussi

l'unique spécialisation protégée de f dont la signature de type correspond à $[\mathbf{T}\pi]$.

$$\begin{aligned}\mathfrak{I}_{\pi,\psi}(t) &= f^{\mathfrak{I}}_{\mathbf{T}\pi}(\mathfrak{I}_{\pi,\psi}(t_1),\ldots,\mathfrak{I}_{\pi,\psi}(t_n))\\ &= (\bar{f}_\tau)^{\mathfrak{M}}_{[\mathbf{T}\pi]}(\mathfrak{M}_{\pi',\psi'}(\textsc{Par}_\theta(t_1)),\ldots,\mathfrak{M}_{\pi',\psi'}(\textsc{Par}_\theta(t_n)))\\ &= (\bar{f}_\tau)^{\mathfrak{M}}_{[\mathbf{T}\theta]\pi'}(\mathfrak{M}_{\pi',\psi'}(\textsc{Par}_\theta(t_1)),\ldots,\mathfrak{M}_{\pi',\psi'}(\textsc{Par}_\theta(t_n)))\\ &= \mathfrak{M}_{\pi',\psi'}(\bar{f}_\tau(\textsc{Par}_\theta(t_1),\ldots,\textsc{Par}_\theta(t_n))) = \mathfrak{M}_{\pi',\psi'}(\textsc{Par}_\theta(t))\end{aligned}$$

3. Considérons une formule universellement quantifiée $\forall(u:T)\,F$ apparaissant dans H. On peux supposer que $u:T$ apparaît dans F et $T \in M$. Alors $\mathcal{D}^{\mathfrak{I}}_{T\pi} = \mathcal{D}^{\mathfrak{M}}_{[T\theta]\pi'}$. De plus pour tout élément de $a \in \mathcal{D}^{\mathfrak{M}}_{[T\theta]\pi'}$, la valuation $\psi[u:T \mapsto a]$ dans $\mathfrak{I}$ produit la valuation $\psi'[u_T:[T\theta] \mapsto a]$ dans $\mathfrak{M}$. On obtient :

$$\begin{aligned}\mathfrak{I}_{\pi,\psi}(\forall(u:T)\,F) &= \bigwedge_{a\in\mathcal{D}^{\mathfrak{I}}_{T\pi}} \mathfrak{I}_{\pi,\psi[u:T\mapsto a]}(F)\\ &= \bigwedge_{a\in\mathcal{D}^{\mathfrak{M}}_{[T\theta]\pi'}} \mathfrak{M}_{\pi',\psi'[u_T:[T\theta]\mapsto a]}(\textsc{Par}_\theta(F))\\ &= \mathfrak{M}_{\pi',\psi'}(\forall(u_T:[T\theta])\,\textsc{Par}_\theta(F)) = \mathfrak{M}_{\pi',\psi'}(\textsc{Par}_\theta(\forall(u:T)\,F))\end{aligned}$$

Donc pour toute formule H dans Γ, H est satisfaite dans $\mathfrak{I}$. □

En pratique, la transformation Par se comporte bien lorsque U reste petit (deux ou trois sortes). Si on essaye d'en protéger plus, par exemple toutes les sortes du problème, la croissance exponentielle dûe aux instanciations de type (par rapport au nombre de variables de type dans la formule) rend la recherche de preuve infaisable. C'est pourquoi, nous ne considérons pas Par comme une alternative pratique à la méthode Tw présentée dans la prochaine section. Cependant, d'un point de vue théorique, elle justifie la séparation entre la protection de type et la fusion des types et elle montre que la notion de sorte protégée donne une bonne caractérisation de la limite entre les deux.

6.3. Encodage par Ponts

La transformation Tw convertit un ensemble de formules en un ensemble equisatisfiable avec sortes protégées. On applique une paire de fonctions de conversion pour passer, lorsque nécessaire, d'une sorte protégée à une sorte non-protégée et vice versa.

Soit U l'ensemble des sortes que l'on veut protéger comme dans la section précédente. Soit un type T, le type transformé $[T]$ est $\overline{T}$ si $T \in U$, et T sinon. Ainsi Tw modifie la signature de la théorie transformée comme suit :

1. On remplace tout symbole de fonction f ayant $\langle S_1,\ldots,S_n,S\rangle$ pour signature par un symbole $\bar{f}$ ayant pour signature de types $\langle[S_1],\ldots,[S_n],[S]\rangle$.
2. On remplace tout symbole de prédicat p ayant $\langle S_1,\ldots,S_n\rangle$ pour signature de types par un symbole $\bar{p}$ ayant pour signature de type $\langle[S_1],\ldots,[S_n]\rangle$.

3. Pour toute sorte $T \in U$, on ajoute une paire de symboles de fonction de "pont" $\mathtt{to}_T : \langle \overline{T}, T \rangle$ et $\mathtt{from}_T : \langle T, \overline{T} \rangle$.

On convertit, alors, les termes et formules atomiques dans la nouvelle signature. Notre but est d'interdire à un type polymorphe d'une signature de type d'être instancié en un type de U. À chaque fois qu'une telle instance apparaît, une fonction de pont est appliquée. Plus précisément :

1. Soit une variable $u : T$, $\mathrm{Tw}(u : T) \triangleq u : [T]$.
2. Soit un terme $t = f(t_1 : T_1, \ldots, t_n : T_n) : T$ et soit $\langle S_1, \ldots, S_n, S \rangle$ sa signature de type.

$$\text{Pour tout } t_i,\ t'_i \triangleq \begin{cases} \mathtt{to}_{T_i}(\mathrm{Tw}(t_i)) : T_i & \text{if } T_i \in U \text{ et } S_i \notin U, \\ \mathrm{Tw}(t_i) & \text{si } T_i \notin U \text{ ou } S_i \in U. \end{cases}$$

$$\text{Alors } \mathrm{Tw}(t) \triangleq \begin{cases} \bar{f}(t'_1, \ldots, t'_n) : [T] & \text{if } T \notin U \text{ où } S \in U, \\ \mathtt{from}_T(\bar{f}(t'_1, \ldots, t'_n) : T) : \overline{T} & \text{si } T \in U \text{ et } S \notin U. \end{cases}$$

3. Soit une formule $p(t_1 : T_1, \ldots, t_n : T_n)$ et soit $\langle S_1, \ldots, S_n \rangle$ la signature de p. Pour tous les arguments t_i, on définit t'_i comme dans le cas précédent. Ainsi, on a $\mathrm{Tw}(p(t_1 : T_1, \ldots, t_n : T_n)) \triangleq \bar{p}(t'_1, \ldots, t'_n)$.

Les égalités et les formules complexes sont converties de manière naturelle :

$$\mathrm{Tw}(t_1 \approx t_2) \triangleq \mathrm{Tw}(t_1) \approx \mathrm{Tw}(t_2) \qquad \mathrm{Tw}(F \wedge G) \triangleq \mathrm{Tw}(F) \wedge \mathrm{Tw}(G)$$
$$\mathrm{Tw}(\neg F) \triangleq \neg\, \mathrm{Tw}(F) \qquad \mathrm{Tw}(\forall x\, F) \triangleq \forall(\mathrm{Tw}(x))\, \mathrm{Tw}(F)$$

Finalement, on convertit les formules dans Γ et on ajoute les axiomes des fonctions de ponts :

$$\begin{aligned} \mathrm{Tw}(\Gamma) \triangleq & \{\, \mathrm{Tw}(F) \mid F \in \Gamma \,\} \\ & \cup \{\, \forall(v : \overline{T})\ \mathtt{from}_T(\mathtt{to}_T(v : \overline{T})) \approx v : \overline{T} \mid T \in U \,\} \\ & \cup \{\, \forall(u : T)\ \mathtt{to}_T(\mathtt{from}_T(u : T)) \approx u : T \mid T \in U \,\} \end{aligned}$$

Exemple. Supposons que $U = \{\mathsf{I}\}$. L'exemple est transformé comme suit. Remarquons que 6, 7, et 42 ont le type $\bar{\mathsf{I}}$ et que la signature de $*$ est $\langle \bar{\mathsf{I}}, \bar{\mathsf{I}}, \bar{\mathsf{I}} \rangle$.

$$\forall(m : \mathsf{M}(\alpha, \mathsf{I}))\, \forall(c : \alpha)\ \mathtt{from}_{\mathsf{I}}(\mathtt{get}(\mathtt{set}(m, c, \mathtt{to}_{\mathsf{I}}(6) : \mathsf{I}), c) : \mathsf{I}) * 7 \approx 42$$

Maintenant si on applique Dis avec $W = \{\mathtt{get}_{[\mathsf{I}/\alpha, \mathsf{I}/\beta]}, \mathtt{set}_{[\mathsf{I}/\alpha, \mathsf{I}/\beta]}\}$ avant d'appliquer Tw. $\mathtt{get}_{[\mathsf{I}/\alpha, \mathsf{I}/\beta]}$ possède maintenant la signature $\langle M(\mathsf{I}, \mathsf{I}), \bar{\mathsf{I}}, \bar{\mathsf{I}} \rangle$ et $\mathtt{set}_{[\mathsf{I}/\alpha, \mathsf{I}/\beta]}$ possède maintenant la signature $\langle M(\mathsf{I}, \mathsf{I}), \bar{\mathsf{I}}, \bar{\mathsf{I}}, M(\mathsf{I}, \mathsf{I}) \rangle$. Et les formules deviennent par Tw :

$$\forall(m_{\mathsf{M}(\alpha,\mathsf{I})} : \mathsf{M}(\alpha, \mathsf{I}))\, \forall(c_\alpha : \alpha)\ \mathtt{from}_{\mathsf{I}}(\mathtt{get}(\mathtt{set}(m_{\mathsf{M}(\alpha,\mathsf{I})}, c_\alpha, \mathtt{to}_{\mathsf{I}}(6)), c_\alpha)) * 7 \approx 42$$
$$\forall(m_{\mathsf{M}(\alpha,\mathsf{I})} : \mathsf{M}(\mathsf{I}, \mathsf{I}))\, \forall(c_\alpha : \bar{\mathsf{I}})\ \mathtt{get}_{[\mathsf{I}/\alpha, \mathsf{I}/\beta]}(\mathtt{set}_{[\mathsf{I}/\alpha, \mathsf{I}/\beta]}(m_{\mathsf{M}(\alpha,\mathsf{I})}, c_\alpha, 6), c_\alpha) * 7 \approx 42$$
$$\forall(v : \bar{\mathsf{I}})\ \mathtt{from}_{\mathsf{I}}(\mathtt{to}_{\mathsf{I}}(v : \bar{\mathsf{I}})) \approx v : \bar{\mathsf{I}}$$
$$\forall(u : \mathsf{I})\ \mathtt{to}_{\mathsf{I}}(\mathtt{from}_{\mathsf{I}}(u : \mathsf{I})) \approx u : \mathsf{I}$$

On peut remarquer que la deuxième formule reste inchangée. De même si l'on choisit $U = \{\mathsf{I}, M(\mathsf{I},\mathsf{I})\}$, on aura alors les mêmes formules à part $M(\mathsf{I},\mathsf{I})$ qui devient $\overline{M(\mathsf{I},\mathsf{I})}$, c'est-à-dire $\mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]} : \langle \overline{M(\mathsf{I},\mathsf{I})}, \bar{\mathsf{I}}, \bar{\mathsf{I}} \rangle$ et $\mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]} : \langle \overline{M(\mathsf{I},\mathsf{I})}, \bar{\mathsf{I}}, \bar{\mathsf{I}}, \overline{M(\mathsf{I},\mathsf{I})} \rangle$.

Lemme 6.3.1. *Pour tout terme t de type T, $\mathrm{Tw}(t)$ est un terme bien formé de type $[T]$, et pour toute formule F, $\mathrm{Tw}(F)$ est une formule bien formée avec les mêmes variables libres (modulo conversion de leur type) et variables de type.*

Démonstration. La preuve se déroule par induction sur la structure des termes et des formules. Les seul cas intéressant est l'application de symbole de fonction (les symboles de prédicats sont traités de la même façon). La conservation des variables libres (de type) est triviale.

Soit t un terme de la forme $f(t_1 : T_1, \ldots, t_n : T_n) : T$. Soit $\langle S_1, \ldots, S_n, S \rangle$ la signature de type de f et τ la substitution de type qui instancie $\langle S_1, \ldots, S_n, S \rangle$ vers $\langle T_1, \ldots, T_n, T \rangle$. Pour tout argument t_i, $\mathrm{Tw}(t_i)$ est un terme bien formé de type $[T_i]$ par hypothèse d'induction, et il y a trois cas à considérer :

1. $T_i \in U$ et $S_i \notin U$. Alors $\mathrm{Tw}(t_i)$ a le type $\overline{T}_i$ et $\mathtt{to}_{T_i}(\mathrm{Tw}(t_i)) : T_i$ est un terme bien formé. Remarquons que le $i$ème élément de la signature de type de $\bar{f}$ est $[S_i] = S_i$, et $S_i\tau = T_i$.
2. $S_i \in U$. Alors S_i est clos, et donc $T_i = S_i$, et $[T_i] = [S_i] = [S_i]\tau$.
3. $S_i \notin U$ et $T_i \notin U$. Alors $[S_i]\tau = S_i\tau = T_i = [T_i]$.

Alors on applique $\bar{f}$ à $t'_1, \ldots, t'_n$ en utilisant τ pour instancier la signature de type de $\bar{f}$. De nouveau on a trois cas possible :

1. $T \in U$ et $S \notin U$. Puisque $S\tau = T$, on obtient $[S]\tau = S\tau = T$. Alors le terme $\mathtt{from}_T(\bar{f}(t'_1, \ldots, t'_n) : T) : \overline{T}$ est bien formé et a le type $[T] = \overline{T}$.
2. $S \in U$. Alors S est close, donc $T = S$ et $[T] = [S] = [S]\tau$.
3. $S \notin U$ et $T \notin U$. Alors $[S]\tau = S\tau = T = [T]$.

Ainsi $\mathrm{Tw}(t)$ est un type bien formé de type $[T]$. □

Théorème 6.3.2 (Correction de Tw)**.** *Si un ensemble de formules closes Γ est satisfiable alors $\mathrm{Tw}(\Gamma)$ est satisfiable.*

Démonstration. Soit $\mathfrak{M}$ un modèle de Γ. Nous allons construire une interprétation $\mathfrak{I}$ de $\mathrm{Tw}(\Gamma)$ telle que, pour toute valuation de type π et pour toute valuation ψ dans $\mathfrak{I}$, il existe une valuation de type π' et une valuation ψ' dans le modèle initial $\mathfrak{M}$, telles que :

$$\mathfrak{I}_{\pi,\psi}(\mathrm{Tw}(t)) = \mathfrak{M}_{\pi',\psi'}(t) \qquad\qquad \mathfrak{I}_{\pi,\psi}(\mathrm{Tw}(F)) = \mathfrak{M}_{\pi',\psi'}(F)$$

pour tout terme t et formule F apparaissant dans Γ.

Premièrement, notons avec $]\cdot[$ *l'élimination de la protection* : pour toute sorte $\overline{S}$, $]\overline{S}[\triangleq S$ et pour tout type non-protégé T,$]\overline{T}[\triangleq T$. Clairement pour tout type de la signature initiale, $][T][= T$. D'un autre coté il n'est généralement pas vrai que $[]T[] = T$.

Pour toute sorte S dans la signature initiale, on définit $\mathcal{D}^{\mathfrak{I}}_S \triangleq \mathcal{D}^{\mathfrak{M}}_S$ et $\mathcal{D}^{\mathfrak{I}}_{\overline{S}} \triangleq \mathcal{D}^{\mathfrak{M}}_S$. On donne maintenant l'interprétation des symboles de fonctions et de prédicats dans $\text{Tw}(\Gamma)$. Excepté les symboles de ponts, tous les symboles de la nouvelle signature proviennent d'un symbole de la signature initiale. Considérons un symbole de fonction f avec la signature $\mathbf{S}$ dans l'ensemble initial Γ. Dans $\text{Tw}(\Gamma)$, il est remplacé par le symbole $\bar{f}$ ayant pour signature de type $[\mathbf{S}]$. Soit $\mathbf{T}$ un vecteur de sorte correspondant à $[\mathbf{S}]$. Il est aisé de voir que le vecteur de sorte "non-protégée" $]\mathbf{T}[$ correspond à $\mathbf{S}$. Ainsi on peut définir $\bar{f}^{\mathfrak{I}}_{\mathbf{T}} \triangleq f^{\mathfrak{M}}_{]\mathbf{T}[}$. Les symboles de prédicat de $\text{Tw}(\Gamma)$ sont définis de la même manière. Finalement pour toute sorte $T \in U$, les symboles de fonction $\texttt{to}_T$ et $\texttt{from}_T$ sont interprétés comme l'identité. Cela termine la définition de $\mathfrak{I}$.

Maintenant prenons une valuation de type π et une valuation ψ dans $\mathfrak{I}$. On choisit $\pi' \triangleq \pi$ et $\psi'(u:T) \triangleq \psi(u:[T])$. Par construction, $\mathcal{D}^{\mathfrak{I}}_{[T]\pi} = \mathcal{D}^{\mathfrak{I}}_{T\pi} = \mathcal{D}^{\mathfrak{M}}_{T\pi'}$.

En utilisant cette définition, on peut prouver les deux égalités par induction sur la structure des termes et des formules. On considère trois cas : une variable, une application de fonction, et une formule universellement quantifiée (les autre cas peuvent être traités de la même manière).

1. Soit $u:T$ une variable apparaissant dans Γ.

$$\mathfrak{I}_{\pi,\psi}(\text{Tw}(u:T)) = \mathfrak{I}_{\pi,\psi}(u:[T]) = \psi(u:[T]) = \psi'(u:T) = \mathfrak{M}_{\pi',\psi'}(u:T)$$

2. Considérons un terme $t = f(t_1:T_1,\ldots,t_n:T_n):T$. Le terme transformé $\text{Tw}(t)$ est l'application du symbole $\bar{f}$ aux arguments transformés $\text{Tw}(t_1),\ldots,\text{Tw}(t_n)$ avec l'ajout, si besoin, des fonctions $\texttt{to}_\square$ et $\texttt{from}_\square$. Par l'hypothèse d'induction on sait que :
   - les symboles de fonctions $\texttt{to}_\square$ et $\texttt{from}_\square$ sont interprétés comme l'identité dans $\mathfrak{I}$, et
   - $\bar{f}$ sur un vecteur de sorte $\mathbf{T}$ est interprété dans $\mathfrak{I}$ comme f sur $]\mathbf{T}[$ dans $\mathfrak{M}$.

   On en conclut que $\mathfrak{I}_{\pi,\psi}(\text{Tw}(t)) = \mathfrak{M}_{\pi',\psi'}(t)$.

3. Considérons une formule universellement quantifiée $\forall(u:T)\,F$ dans Γ. On sait que $\mathcal{D}^{\mathfrak{I}}_{[T]\pi} = \mathcal{D}^{\mathfrak{M}}_{T\pi'}$. De plus pour tout élément $a \in \mathcal{D}^{\mathfrak{I}}_{[T]\pi}$, la valuation $\psi[u:[T] \mapsto a]$ dans $\mathfrak{I}$ produit la valuation $\psi'[u:T \mapsto a]$ dans $\mathfrak{M}$. On obtient ainsi :

$$\begin{aligned}\mathfrak{I}_{\pi,\psi}(\text{Tw}(\forall(u:T)\,F)) &= \bigwedge_{a\in\mathcal{D}^{\mathfrak{I}}_{[T]\pi}} \mathfrak{I}_{\pi,\psi[u:[T]\mapsto a]}(\text{Tw}(F)) \\ &= \bigwedge_{a\in\mathcal{D}^{\mathfrak{M}}_{T\pi'}} \mathfrak{M}_{\pi',\psi'[u:T\mapsto a]}(F) = \mathfrak{M}_{\pi',\psi'}(\forall(u:T)\,F)\end{aligned}$$

Ainsi, pour toutes les formules dans Γ, $\text{Tw}(F)$ est satisfait dans $\mathfrak{I}$. Les axiomes de bijections $\texttt{to}_\square$ et $\texttt{from}_\square$ sont trivialement satisfaits puisque ces deux fonctions sont interprétées par l'identité. □

Théorème 6.3.3 (Complétude de Tw)**.** *Pour tout ensemble de formules closes* Γ*, si* $\text{Tw}(\Gamma)$ *est satisfiable alors* Γ *est satisfiable.*

Démonstration. Soit $\mathfrak{M}$ un modèle de $\mathrm{Tw}(\Gamma)$. Grâce aux axiomes de bijections des fonctions de ponts, on peut supposer sans perte de généralité que, pour toute sorte $T \in U$, $\mathcal{D}_T^{\mathfrak{M}} = \mathcal{D}_{\overline{T}}^{\mathfrak{M}}$ et $\mathtt{to}_T$ et $\mathtt{from}_T$ sont interprétés dans $\mathfrak{M}$ comme l'identité.

Nous allons construire l'interprétation de $\mathfrak{I}$ de Γ telle que, pour toute valuation de type π et pour toute valuation ψ dans $\mathfrak{I}$, il existe une valuation de type π' et une valuation ψ' dans le modèle initial $\mathfrak{M}$, telles que :

$$\mathfrak{I}_{\pi,\psi}(t) = \mathfrak{M}_{\pi',\psi'}(\mathrm{Tw}(t)) \qquad\qquad \mathfrak{I}_{\pi,\psi}(F) = \mathfrak{M}_{\pi',\psi'}(\mathrm{Tw}(F))$$

pour tous les types t et formules F apparaissant dans Γ.

Pour toute sorte S dans la signature de Γ, on définit $\mathcal{D}_S^{\mathfrak{I}} \triangleq \mathcal{D}_S^{\mathfrak{M}}$. Ainsi pour tout type T et pour toutes ses instances de type clos $T\tau$ $\mathcal{D}_{T\tau}^{\mathfrak{I}} = \mathcal{D}_{[T]\tau}^{\mathfrak{M}}$. En effet si $T \in U$ alors $\mathcal{D}_{T\tau}^{\mathfrak{I}} = \mathcal{D}_T^{\mathfrak{I}} = \mathcal{D}_T^{\mathfrak{M}} = \mathcal{D}_{\overline{T}}^{\mathfrak{M}} = \mathcal{D}_{[T]\tau}^{\mathfrak{M}}$, sinon $[T] = T$.

Tout symbole de fonction f (resp. symbole de prédicat p) de signature $\mathbf{S}$ dans Γ interprété sur un vecteur de sorte clos $\mathbf{S}\tau$ dans $\mathfrak{I}$ exactement comme le symbole correspondant $\bar{f}$ (respectivement, $\bar{p}$) sur $[\mathbf{S}]\tau$ dans $\mathfrak{M}$. Cela conclut la définition de $\mathfrak{I}$.

Considérons maintenant une valuation de type π et une valuation ψ dans $\mathfrak{I}$. On choisit $\pi' \triangleq \pi$ et $\psi'(u\!:\![T]) \triangleq \psi(u\!:\!T)$ pour toutes les variables $u\!:\!T$ apparaissant dans Γ. Le lemme 5.1.1 permet de définir ψ' seulement sur les variables qui apparaissent dans la formule à transformer. Rappelons nous également que $\mathcal{D}_{T\pi}^{\mathfrak{I}} = \mathcal{D}_{[T]\pi}^{\mathfrak{M}} = \mathcal{D}_{[T]\pi'}^{\mathfrak{M}}$.

En utilisant cette définition, on peut prouver les deux égalités par induction sur la structure des termes et des formules. On considère trois cas : les variables, les applications de fonctions et les formules universellement quantifiées (les autres cas peuvent être traités de la même manière).

1. Soit $u\!:\!T$ une variable apparaissant dans Γ. On obtient :

$$\mathfrak{I}_{\pi,\psi}(u\!:\!T) = \psi(u\!:\!T) = \psi'(u\!:\![T]) = \mathfrak{M}_{\pi',\psi'}(u\!:\![T]) = \mathfrak{M}_{\pi',\psi'}(\mathrm{Tw}(u\!:\!T))$$

2. Considérons un terme $t = f(t_1\!:\!T_1, \ldots, t_n\!:\!T_n)\!:\!T$. On pose $\mathbf{S} = \langle S_1, \ldots, S_n, S\rangle$ la signature de type de f et τ la substitution de type tel que $\mathbf{S}\tau = \langle T_1, \ldots, T_n, T\rangle$. Le terme $\mathrm{Tw}(t)$ est de la forme $\bar{f}(t'_1\!:\![S_1]\tau, \ldots, t'_n\!:\![S_n]\tau)\!:\![S]\tau$ (avec peut-être un $\mathtt{from}_T$ en plus) où chaque t'_i est $\mathrm{Tw}(t_i)$ (avec peut-être un $\mathtt{to}_{T_i}$ en plus). Sachant que
   – les symboles de fonctions de pont sont interprétés dans $\mathfrak{M}$ comme l'identité ;
   – f est interprété sur $\mathbf{S}\tau$ dans $\mathfrak{I}$ comme $\bar{f}$ sur $[\mathbf{S}]\tau$ dans $\mathfrak{M}$,
   on peut conclure que $\mathfrak{I}_{\pi,\psi}(t) = \mathfrak{M}_{\pi',\psi'}(\mathrm{Tw}(t))$.

3. Considérons une formule universellement quantifiée $\forall(u\!:\!T)\,F$. Pour tout $a \in \mathcal{D}_{T\pi}^{\mathfrak{I}}$, la valuation $\psi[u\!:\!T \mapsto a]$ dans $\mathfrak{I}$ produit la valuation $\psi'[u\!:\![T] \mapsto a]$ dans $\mathfrak{M}$.

$$\begin{aligned}\mathfrak{I}_{\pi,\psi}(\forall(u\!:\!T)\,F) &= \bigwedge_{a\in\mathcal{D}_{T\pi}^{\mathfrak{I}}} \mathfrak{I}_{\pi,\psi[u\,:\,T\mapsto a]}(F)\\ &= \bigwedge_{a\in\mathcal{D}_{[T]\pi'}^{\mathfrak{M}}} \mathfrak{M}_{\pi',\psi'[u\,:\,[T]\mapsto a]}(\mathrm{Tw}(F))\\ &= \mathfrak{M}_{\pi',\psi'}(\forall(u\!:\![T])\,\mathrm{Tw}(F)) = \mathfrak{M}_{\pi',\psi'}(\mathrm{Tw}(\forall(u\!:\!T)\,F))\end{aligned}$$

Ainsi, pour toute formule F dans Γ, F est satisfait dans $\mathfrak{I}$. □

Outre le fait que cette présentation des ponts est plus générale que celle Couchot et Lescuyer dans [20], la présentation des ponts est ici séparée de celle des encodages du polymorphisme. Cela permet de faire naturellement apparaître l'encodage des types dans les deux axiomes de bijection. En particulier dans le cas de la décoration que Couchot et Lescuyer ont présenté en conjonction des ponts, on verra que nos axiomes de bijection sont au final différents des leurs. Cela permet d'appliquer la protection également au type fini. Nous verrons plus précisement les contre-exemples à la forme des axiomes de bijection par Couchot et Lescuyer dans la prochaine section.

6.4. Encodage par décoration

La transformation Dec convertit un problème polymorphe avec des sortes protégées en un problème monomorphe équisatisfiable. Informellement, pour garder les informations de type, elle décore chaque terme avec son type, qui est lui même transformé en un terme d'une sorte spéciale.

La décoration de tous les termes est apparue précédemment dans [32], l'exclusion du type I apparaît après dans [20]. Maintenant nous allons montrer comment on peut éviter à tous les types de U d'être décorés et nous allons montrer la correction et complétude.

Premièrement on introduit trois nouvelles sortes U, D et T. La première est associée aux termes non-décorés, la seconde aux termes décorés et la troisième aux termes représentant les types. Pour transformer la signature des symboles des fonctions et prédicats, on utilise les opérations suivantes sur les types :

$$[T]^- \triangleq \begin{cases} T & \text{si } T \text{ est protégé,} \\ \mathsf{D} & \text{sinon} \end{cases} \qquad [T]^+ \triangleq \begin{cases} T & \text{si } T \text{ est protégé,} \\ \mathsf{U} & \text{sinon} \end{cases}$$

La signature de la théorie transformée est définie comme suit :

1. L'ensemble des constructeurs de type est étendu par U, D, T.
2. On remplace tout symbole de fonction f ayant la signature de type $\langle S_1, \ldots, S_n, S\rangle$ par le symbole $\hat{f}$ ayant la signature monomorphe de type $\langle [S_1]^-, \ldots, [S_n]^-, [S]^+\rangle$.
3. On remplace tout symbole de prédicat p ayant la signature de type $\langle S_1, \ldots, S_n\rangle$ par le symbole $\hat{p}$ ayant la signature monomorphe de type $\langle [S_1]^-, \ldots, [S_n]^-\rangle$.
4. Pour tout symbole de variable u et pour tout type T, on ajoute un nouveau symbole de variable u_T.
5. Pour toute variable de type $\alpha \in \mathbb{V}_{\mathrm{T}}$, on ajoute un nouveau symbole v^α.
6. Pour tous les constructeurs de type $\mathsf{F} \in FT$, on ajoute un nouveau symbole de fonction $\mathtt{F}$ ayant la même arité et la signature $\langle \mathsf{T}, \ldots, \mathsf{T}, \mathsf{T}\rangle$. De cette manière tous les types peuvent être représentés dans T.
7. On ajoute un symbole de "décoration" $\mathtt{deco}{:}\langle \mathsf{T}, \mathsf{U}, \mathsf{D}\rangle$.

La transformation Dec transforme des types non-protégés vers des termes de type T :

$$\textsc{Dec}(\alpha) \triangleq v^{\alpha} : \mathsf{T} \qquad \textsc{Dec}(\mathsf{F}(T_1, \ldots, T_n)) \triangleq \mathtt{F}(\textsc{Dec}(T_1), \ldots, \textsc{Dec}(T_n)) : \mathsf{T}$$

Dans la définition suivante, $\mathbf{t}$ représente un vecteur de termes, $\overline{S}$ une sorte protégée, et T un type non-protégé. La transformation Dec s'applique sur les termes de la manière suivante :

$$\begin{aligned}
\textsc{Dec}(u : \overline{S}) &\triangleq u_{\overline{S}} : \overline{S} \\
\textsc{Dec}(u : T) &\triangleq \mathtt{deco}(\textsc{Dec}(T), u_T : \mathsf{U}) : \mathsf{D} \\
\textsc{Dec}(f(\mathbf{t}) : \overline{S}) &\triangleq \hat{f}(\textsc{Dec}(\mathbf{t})) : \overline{S} \\
\textsc{Dec}(f(\mathbf{t}) : T) &\triangleq \mathtt{deco}(\textsc{Dec}(T), \hat{f}(\textsc{Dec}(\mathbf{t})) : \mathsf{U}) : \mathsf{D}
\end{aligned}$$

et sur les formules ainsi :

$$\begin{aligned}
\textsc{Dec}(p(\mathbf{t})) &\triangleq \hat{p}(\textsc{Dec}(\mathbf{t})) & \textsc{Dec}(\neg F) &\triangleq \neg\, \textsc{Dec}(F) \\
\textsc{Dec}(t_1 \approx t_2) &\triangleq \textsc{Dec}(t_1) \approx \textsc{Dec}(t_2) & \textsc{Dec}(\forall (u : \overline{S}) F) &\triangleq \forall (u_{\overline{S}} : \overline{S})\, \textsc{Dec}(F) \\
\textsc{Dec}(F \wedge G) &\triangleq \textsc{Dec}(F) \wedge \textsc{Dec}(G) & \textsc{Dec}(\forall (u : T) F) &\triangleq \forall (u_T : \mathsf{U})\, \textsc{Dec}(F)
\end{aligned}$$

sur une formule close H avec $\{\alpha_1, \ldots, \alpha_m\} = \mathrm{FV}_{\mathrm{T}}(H)$:

$$\textsc{Dec}^{\circ}(H) \triangleq \forall (v^{\alpha_1} : \mathsf{T}) \ldots \forall (v^{\alpha_m} : \mathsf{T})\, \textsc{Dec}(H)$$

Exemple Supposons $U = \{\mathsf{I}\}$. Les transformations Par et Dec$^{\circ}$ produisent la formule monomorphe suivante (par clarté on omet les indices des variables ajoutées par Par) :

$$\begin{aligned}
&\forall (v^{\alpha} : \mathsf{T})\, \forall (m_{\mathsf{M}(\alpha,\mathsf{I})} : \mathsf{U})\, \forall (c_{\alpha} : \mathsf{U})\, \mathtt{get}_{[\mathsf{I}/\beta]}(\mathtt{deco}(\mathtt{M}(v^{\alpha}, \mathtt{I}), \\
&\qquad \mathtt{set}_{[\mathsf{I}/\beta]}(\mathtt{deco}(\mathtt{M}(v^{\alpha}, \mathtt{I}), m_{\mathsf{M}(\alpha,\mathsf{I})}), \mathtt{deco}(v^{\alpha}, c_{\alpha}), \mathsf{6})), \mathtt{deco}(v^{\alpha}, c_{\alpha})) * \mathsf{7} \approx \mathsf{42}
\end{aligned}$$

$$\begin{aligned}
&\forall (m_{\mathsf{M}(\mathsf{I},\mathsf{I})} : \mathsf{U})\, \forall (c_{\mathsf{i}} : \bar{\mathsf{I}})\, \mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(\mathtt{deco}(\mathtt{M}(\mathtt{I}, \mathtt{I}), \\
&\qquad \mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(\mathtt{deco}(\mathtt{M}(\mathtt{I}, \mathtt{I}), m_{\mathsf{M}(\mathsf{I},\mathsf{I})}), c_{\mathsf{i}}, \mathsf{6})), c_{\mathsf{i}}) * \mathsf{7} \approx \mathsf{42}
\end{aligned}$$

alors que Tw et Dec$^{\circ}$ produisent :

$$\begin{aligned}
&\forall (v^{\alpha} : \mathsf{T})\, \forall (m_{\mathsf{M}(\alpha,\mathsf{I})} : \mathsf{U})\, \forall (c_{\alpha} : \mathsf{U})\, \mathtt{from}_{\mathsf{I}}(\mathtt{deco}(\mathtt{I}, \mathtt{get}(\mathtt{deco}(\mathtt{M}(v^{\alpha}, \mathtt{I}), \\
&\qquad \mathtt{set}(\mathtt{deco}(\mathtt{M}(v^{\alpha}, \mathtt{I}), m_{\mathsf{M}(\alpha,\mathsf{I})}), \mathtt{deco}(v^{\alpha}, c_{\alpha}), \mathtt{deco}(\mathtt{I}, \mathtt{to}_{\mathsf{I}}(\mathsf{6})))), \\
&\qquad\qquad \mathtt{deco}(v^{\alpha}, c_{\alpha})))) * \mathsf{7} \approx \mathsf{42}
\end{aligned}$$

Le deuxième axiome des fonctions de pont $\mathtt{to}_{\mathsf{I}}$ et $\mathtt{from}_{\mathsf{I}}$ devient

$$\forall (u_{\mathsf{I}} : \mathsf{U})\, \mathtt{deco}(\mathtt{I}, \mathtt{to}_{\mathsf{I}}(\mathtt{from}_{\mathsf{I}}(\mathtt{deco}(\mathtt{I}, u_{\mathsf{I}})))) \approx \mathtt{deco}(\mathtt{I}, u_{\mathsf{I}})$$

Grâce à l'application extérieure de `deco` des deux côtés de l'égalité, notre traduction est correcte même si on protège des types finis tels que les booléens. Sans cette décoration additionnelle (comme dans [20, Eq. (8)]), la finitude d'une sorte protégée implique la finitude de toute la sorte U. Les transformations Dis + Tw + Dec$^\circ$, avec $W = \{\mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}, \mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}\}$ et $U = \{\mathsf{I}, M(\mathsf{I},\mathsf{I})\}$, produisent trois formules en plus, deux axiomes de ponts pour $\overline{\mathsf{M}(\mathsf{I},\mathsf{I})}$ et la formule suivante :

$$\forall(m_{\overline{\mathsf{M}(\mathsf{I},\mathsf{I})}} : \overline{\mathsf{M}(\mathsf{I},\mathsf{I})})\, \forall(c_{\bar{\mathsf{I}}} : \bar{\mathsf{I}})\, \mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(\mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(m_{\mathsf{M}(\alpha,\mathsf{I})}, c_\alpha, \mathsf{6}), c_\alpha) * \mathsf{7} \approx \mathsf{42} \tag{6.3}$$

$$\tag{6.4}$$

On peut remarquer que dans la deuxième formule il n'y a pas de décorations puisque tous les types sont protégés. De plus $\mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}$ possède la signature $\langle \overline{M(\mathsf{I},\mathsf{I})}, \bar{\mathsf{I}}, \bar{\mathsf{I}} \rangle$, on peut donc profiter d'une théorie prédéfinie des tableaux du démonstrateur automatique.

Lemme 6.4.1. *Pour tout terme t de type T,* Dec(t) *est un terme bien formé monomorphe de type $[T]^-$. Pour toute formule F,* Dec(F) *est une formule monomorphe bien formée. De plus* $\mathrm{FV}(\textsc{Dec}(t)) = \{u_T : [T]^+ \mid u : T \in \mathrm{FV}(t)\} \cup \{v^\alpha : \mathsf{T} \mid \alpha \in \mathrm{FV}_\mathrm{T}(t)\}$ *et* $\mathrm{FV}(\textsc{Dec}(F)) = \{u_T : [T]^+ \mid u : T \in \mathrm{FV}(F)\} \cup \{v^\alpha : \mathsf{T} \mid \alpha \in \mathrm{FV}_\mathrm{T}(F)\}$. *Pour toute formule close F,* Dec$^\circ(F)$ *est une formule monomorphe close bien formée.*

Démonstration. Le dernier point se déduit aisément des précédents. La preuve des deux premières affirmations se déroule par induction sur la structure des termes et formules. Le seul cas intéressant est l'application d'un symbole de fonction (les symboles de prédicats sont traités de la même manière).

Considérons un terme $t = f(t_1 : T_1, \ldots, t_n : T_n) : T$. On note S_i la i-ème sorte de la signature de sorte de f. Il y a deux cas possibles : soit S_i et T_i sont protégés et $S_i = T_i$, soit S_i et T_i sont à la fois non-protégés, par définition de la correspondance de type protégé. Cependant, $[T_i]^- = [S_i]^-$ pour tout $i \in [1, n]$.

Maintenant, soit S le type des valeurs de retour dans la signature de f. Par le même argument , soit S et T sont protégés et $S = T$, soit S et T ne sont pas protégés. Ainsi $[T]^+ = [S]^+$ et le terme $t' = \hat{f}(\textsc{Dec}(t_1), \ldots, \textsc{Dec}(t_n)) : [T]^+$ est bien formé.

Enfin, si T est protégé, alors $\textsc{Dec}(t) = t'$ et $[T]^- = T = [T]^+$. Remarquez que le type T ne contient pas de variables de type. Sinon, si T est non-protégé, $[T]^+ = \mathsf{U}$ et $\textsc{Dec}(t) = \mathtt{deco}(\textsc{Dec}(T), t') : \mathsf{D}$ est un type bien formé de type $[T]^-$. De plus toutes les variables de type α présentes dans T apparaissent en tant que variable libre $v^\alpha : \mathsf{T}$ dans le premier argument de `deco` dans Dec(t). □

Théorème 6.4.2 (Correction de Dec)**.** *Si un ensemble Γ de formules closes avec des types protégés est satisfiable alors* Dec$^\circ(\Gamma)$ *est satisfiable.*

Démonstration. Soit $\mathfrak{M}$ un modèle de Γ. Nous allons construire une interprétation $\mathfrak{I}$ de Dec$^\circ(\Gamma)$ telle que, pour toute valuation ψ dans $\mathfrak{I}$, il existe un valuation de type π' et une valuation ψ' dans le modèle initial $\mathfrak{M}$, tel que :

$$\mathfrak{I}_\psi(\textsc{Dec}(t : T)) = \langle T\pi', \mathfrak{M}_{\pi',\psi'}(t) \rangle \qquad \mathfrak{I}_\psi(\textsc{Dec}(F)) = \mathfrak{M}_{\pi',\psi'}(F)$$

pour tous les termes t et formules F apparaissant dans Γ. Nous ne considérons pas de valuation de type dans $\mathfrak{I}$, puisque le résultat de $\textsc{Dec}^\circ$ est entièrement monomorphe. Dès que deux égalités sont prouvées, il est facile de montrer que pour toute formule close $H \in \Gamma$, l'interprétation $\mathfrak{I}$ satisfait $\textsc{Dec}^\circ(H)$:

$$\mathfrak{I}(\textsc{Dec}^\circ(H)) = \bigwedge_\psi \mathfrak{I}_\psi(\textsc{Dec}(H)) = \bigwedge_\psi \mathfrak{M}_{\pi',\psi'}(H) = \top$$

Pour toutes les sortes S dans la signature initiale, protégées ou non, nous gardons son domaine en le couplant avec S :

$$\mathcal{D}^{\mathfrak{I}}_S \triangleq \{S\} \times \mathcal{D}^{\mathfrak{M}}_S$$

Le domaine de D est défini comme la somme dépendante sur les sortes non-protégées :

$$\mathcal{D}^{\mathfrak{I}}_{\mathsf{D}} \triangleq \{\, \langle S, c\rangle \mid S \in \mathcal{T}(\mathtt{FT}),\ c \in \mathcal{D}^{\mathfrak{M}}_S \,\}$$

Le domaine de U est l'ensemble de toutes les fonctions qui associent à toutes les sortes non-protégées S dans $\mathcal{T}(\mathtt{FT})$ un élément de $\mathcal{D}^{\mathfrak{M}}_S$:

$$\mathcal{D}^{\mathfrak{I}}_{\mathsf{U}} \triangleq \{\, S \in \mathcal{T}(\mathtt{FT}) \longmapsto c \in \mathcal{D}^{\mathfrak{M}}_S \,\}$$

Finalement le domaine de T est l'ensemble de toutes les sortes non-protégées : $\mathcal{D}^{\mathfrak{I}}_{\mathsf{T}} \triangleq \mathcal{T}(\mathtt{FT})$.

Maintenant nous donnons l'interprétation des symboles de fonctions et de prédicats dans $\textsc{Dec}^\circ(\Gamma)$. Soit f un symbole de fonction dans Γ ayant la signature de type $\mathbf{S} = \langle S_1, \ldots, S_n, S\rangle$. Le symbole $\hat{f}$ à la signature $\mathbf{S}' = \langle [S_1]^-, \ldots, [S_n]^-, [S]^+\rangle$ et c'est le seul vecteur de sorte sur lequel on interprète $\hat{f}$, puisque qu'il est monomorphe. Premièrement supposons que S est protégé. Alors on définit :

$$\hat{f}^{\mathfrak{I}}_{\mathbf{S}'}(\langle T_1, c_1\rangle, \ldots, \langle T_n, c_n\rangle) \triangleq \begin{cases} \langle S, f^{\mathfrak{M}}_{\langle T_1,\ldots,T_n,S\rangle}(c_1,\ldots,c_n)\rangle & \text{si } \mathbf{S} \text{ correspond à } \langle T_1,\ldots,T_n,S\rangle, \\ \langle S, e_S\rangle & \text{sinon,} \end{cases}$$

où e_S est un élément arbitraire mais fixé de $\mathcal{D}^{\mathfrak{M}}_S$. Sinon si S est non-protégé, nous posons :

$$\hat{f}^{\mathfrak{I}}_{\mathbf{S}'}(\langle T_1, c_1\rangle, \ldots, \langle T_n, c_n\rangle) \triangleq \lambda T \in \mathcal{T}(\mathtt{FT}).\begin{cases} f^{\mathfrak{M}}_{\langle T_1,\ldots,T_n,T\rangle}(c_1,\ldots,c_n) & \text{si } \mathbf{S} \text{ correspond à } \langle T_1,\ldots,T_n,T\rangle, \\ e_T & \text{sinon,} \end{cases}$$

où e_T est un élément arbitraire mais fixé de $\mathcal{D}^{\mathfrak{M}}_T$. Similairement, soit p un symbole de prédicat dans Γ ayant la signature de type $\mathbf{S} = \langle S_1, \ldots, S_n\rangle$. Le symbole $\hat{p}$ a la signature de type $\mathbf{S}' = \langle [S_1]^-, \ldots, [S_n]^-\rangle$, et on pose :

$$\hat{p}^{\mathfrak{I}}_{\mathbf{S}'}(\langle T_1, c_1\rangle, \ldots, \langle T_n, c_n\rangle) \triangleq \begin{cases} p^{\mathfrak{M}}_{\langle T_1,\ldots,T_n\rangle}(c_1,\ldots,c_n) & \text{si } \mathbf{S} \text{ correspond à } \langle T_1,\ldots,T_n\rangle, \\ \top & \text{sinon.} \end{cases}$$

Le symbole `deco` est interprété comme suit :

$$\mathtt{deco}^{\mathfrak{J}}_{\langle \mathsf{T},\mathsf{U},\mathsf{D}\rangle}(T,h) \triangleq \langle T, h(T)\rangle$$

Finalement, les symboles de fonctions sont interprétés exactement comme les constructeurs de types initiaux :

$$\mathsf{F}^{\mathfrak{J}}_{\langle \mathsf{T},\ldots,\mathsf{T}\rangle}(T_1,\ldots,T_n) \triangleq \mathsf{F}(T_1,\ldots,T_n)$$

L'interprétation $\mathfrak{I}$ a maintenant été définie. Considérons une valuation ψ dans $\mathfrak{I}$. Pour toutes les variables de type α, on pose $\alpha\pi' \triangleq \psi(v^\alpha : \mathsf{T})$. Tout type dans $\mathcal{D}^{\mathfrak{J}}_{\mathsf{T}}$ est un type non-protégé, donc π' est une valuation de type. De plus pour tous les types non-protégés T, on a $\mathfrak{I}_\psi(\textsc{Dec}(T)) = T\pi'$. Donc, pour toutes les variables $u:T$ dans Γ, on pose :

$$\psi'(u:T) \triangleq \begin{cases} c & \text{si } T \text{ est protégé et } \psi(u_T : T) = \langle T, c\rangle, \\ \psi(u_T : \mathsf{U})(T\pi') & \text{sinon.} \end{cases}$$

On prouve les deux égalités principales encore une fois par induction structurelle sur les termes et les formules. On considère cinq cas : les variables, les applications de fonctions, les égalités, et les formules quantifiés universellement sur une variable de type protégé ou non.

1. Soit $u:T$ une variable de Γ. Si T est protégé, alors $\mathfrak{I}_\psi(\textsc{Dec}(u:T)) = \mathfrak{I}_\psi(u_T:T) = \psi(u_T:T) = \langle T, \psi'(u:T)\rangle = \langle T, \mathfrak{M}_{\pi',\psi'}(u:T)\rangle$. Sinon, si T est un type non-protégé, on a :

$$\begin{aligned}\mathfrak{I}_\psi(\textsc{Dec}(u:T)) &= \mathfrak{I}_\psi(\mathtt{deco}(\textsc{Dec}(T), u_T:\mathsf{U}):\mathsf{D}) = \\ &\mathtt{deco}^{\mathfrak{J}}_{\langle \mathsf{T},\mathsf{U},\mathsf{D}\rangle}(\mathfrak{I}_\psi(\textsc{Dec}(T)), \mathfrak{I}_\psi(u_T:\mathsf{U})) = \langle T\pi', \psi(u_T:\mathsf{U})(T\pi')\rangle = \\ &\langle T\pi', \psi'(u:T)\rangle = \langle T\pi', \mathfrak{M}_{\pi',\psi'}(u:T)\rangle\end{aligned}$$

2. Considérons un terme $t = f(t_1:T_1,\ldots,t_n:T_n):T$. Soit $\mathbf{S} = \langle S_1,\ldots,S_n,S\rangle$ la signature de f et $\mathbf{S}' = \langle [S_1]^-,\ldots,[S_n]^-,[S]^+\rangle$. Si T est protégé, alors

$$\begin{aligned}\mathfrak{I}_\psi(\textsc{Dec}(t)) &= \hat{f}^{\mathfrak{J}}_{\mathbf{S}'}(\mathfrak{I}_\psi(\textsc{Dec}(t_1)),\ldots,\mathfrak{I}_\psi(\textsc{Dec}(t_n))) = \\ &\hat{f}^{\mathfrak{J}}_{\mathbf{S}'}(\langle T_1\pi', \mathfrak{M}_{\pi',\psi'}(t_1)\rangle,\ldots,\langle T_n\pi', \mathfrak{M}_{\pi',\psi'}(t_n)\rangle) = \\ &\langle T, f^{\mathfrak{M}}_{\langle T_1\pi',\ldots,T_n\pi',T\rangle}(\mathfrak{M}_{\pi',\psi'}(t_1),\ldots,\mathfrak{M}_{\pi',\psi'}(t_n))\rangle = \langle T\pi', \mathfrak{M}_{\pi',\psi'}(t)\rangle\end{aligned}$$

   Maintenant, si T n'est pas protégé, on obtient :

$$\begin{aligned}\mathfrak{I}_\psi(\textsc{Dec}(t)) &= \mathfrak{I}_\psi(\mathtt{deco}(\textsc{Dec}(T), \hat{f}(\textsc{Dec}(t_1),\ldots,\textsc{Dec}(t_n)):\mathsf{U}):\mathsf{D}) = \\ &\mathtt{deco}^{\mathfrak{J}}_{\langle \mathsf{T},\mathsf{U},\mathsf{D}\rangle}(T\pi', \hat{f}^{\mathfrak{J}}_{\mathbf{S}'}(\mathfrak{I}_\psi(\textsc{Dec}(t_1)),\ldots,\mathfrak{I}_\psi(\textsc{Dec}(t_n)))) = \\ &\langle T\pi', \hat{f}^{\mathfrak{J}}_{\mathbf{S}'}(\langle T_1\pi', \mathfrak{M}_{\pi',\psi'}(t_1)\rangle,\ldots,\langle T_n\pi', \mathfrak{M}_{\pi',\psi'}(t_n)\rangle)(T\pi')\rangle = \\ &\langle T\pi', f^{\mathfrak{M}}_{\langle T_1\pi',\ldots,T_n\pi',T\pi'\rangle}(\mathfrak{M}_{\pi',\psi'}(t_1),\ldots,\mathfrak{M}_{\pi',\psi'}(t_n))\rangle = \langle T\pi', \mathfrak{M}_{\pi',\psi'}(t)\rangle\end{aligned}$$

3. Considérons une égalité $t_1 \approx t_2$. Soit T le type de t_1 et t_2. Par l'hypothèse d'induction, $\mathfrak{I}_\psi(\textsc{Dec}(t_1)) = \langle T\pi', \mathfrak{M}_{\pi',\psi'}(t_1)\rangle$ et $\mathfrak{I}_\psi(\textsc{Dec}(t_2)) = \langle T\pi', \mathfrak{M}_{\pi',\psi'}(t_2)\rangle$. Ils sont égaux si et seulement si $\mathfrak{M}_{\pi',\psi'}(t_1)$ et $\mathfrak{M}_{\pi',\psi'}(t_2)$ le sont.
4. Considérons une formule universellement quantifiée $\forall(u:S)\,F$, où S est protégée. Par construction, $\mathcal{D}^{\mathfrak{I}}_S = \{S\} \times \mathcal{D}^{\mathfrak{M}}_S$. Prenons un élément arbitraire $\langle S, c\rangle \in \mathcal{D}^{\mathfrak{I}}_S$. D'après les définitions précédentes, la valuation $\psi[u_S : S \mapsto \langle S, c\rangle]$ produit dans $\mathfrak{M}$ la même valuation de type π' et la valuation $\psi'[u : S \mapsto c]$. On obtient :

$$\begin{aligned}\mathfrak{I}_\psi(\textsc{Dec}(\forall(u:S)\,F)) &= \bigwedge_{\langle S,c\rangle \in \mathcal{D}^{\mathfrak{I}}_S} \mathfrak{I}_{\psi[u_S : S \mapsto \langle S,c\rangle]}(\textsc{Dec}(F)) \\ &= \bigwedge_{c \in \mathcal{D}^{\mathfrak{M}}_S} \mathfrak{M}_{\pi',\psi'[u:S\mapsto c]}(F) = \mathfrak{M}_{\pi',\psi'}(\forall(u:S)\,F)\end{aligned}$$

5. Considérons une formule $\forall(u:T)\,F$, où T est un type non-protégé. Après la transformation, le symbole de variable u_T acquière le type U. Tout élément de $\mathcal{D}^{\mathfrak{I}}_{\mathsf{U}}$ est une fonction qui associe à chaque sorte non-protégée S un élément de $\mathcal{D}^{\mathfrak{M}}_S$. Cependant, nous sommes seulement intéressés par la valeur de la fonction sur $T\pi'$, puisque pour toutes fonctions $h, h' \in \mathcal{D}^{\mathfrak{I}}_{\mathsf{U}}$ qui coïncident sur $T\pi'$, $\mathfrak{I}_{\psi[u_T:\mathsf{U}\mapsto h]}(\textsc{Dec}(F)) = \mathfrak{I}_{\psi[u_T:\mathsf{U}\mapsto h']}(\textsc{Dec}(F))$. En effet, $u_T : \mathsf{U}$ peut seulement apparaître dans $\textsc{Dec}(F)$ à l'intérieur d'un terme $\mathtt{deco}(\textsc{Dec}(T), u_T : \mathsf{U}) : \mathsf{D}$. Puisque $\textsc{Dec}(F)$ ne contient pas de quantificateur sur T, le terme $\textsc{Dec}(T)$ sera toujours évalué vers $T\pi'$. Par conséquent, le terme $\mathtt{deco}(\textsc{Dec}(T), u_T : \mathsf{U}) : \mathsf{D}$ sera évalué en $\langle T\pi', h(T\pi')\rangle$ dans les deux cas.

   Pour tout $c \in \mathcal{D}^{\mathfrak{M}}_{T\pi'}$, on pose h_c une fonction arbitraire mais fixé de $\mathcal{D}^{\mathfrak{I}}_{\mathsf{U}}$ qui va de $T\pi'$ vers c. L'évaluation $\psi[u_T : \mathsf{U} \mapsto h_c]$ produit dans $\mathfrak{M}$ la même évaluation de type π' et la valuation $\psi'[u : T \mapsto c]$. On obtient :

$$\begin{aligned}\mathfrak{I}_\psi(\textsc{Dec}(\forall(u:T)\,F)) &= \bigwedge_{h \in \mathcal{D}^{\mathfrak{I}}_{\mathsf{U}}} \mathfrak{I}_{\psi[u_T:\mathsf{U}\mapsto h]}(\textsc{Dec}(F)) \\ &= \bigwedge_{c \in \mathcal{D}^{\mathfrak{M}}_{T\pi'}} \mathfrak{I}_{\psi[u_T:\mathsf{U}\mapsto h_c]}(\textsc{Dec}(F)) \\ &= \bigwedge_{c \in \mathcal{D}^{\mathfrak{M}}_{T\pi'}} \mathfrak{M}_{\pi',\psi'[u:T\mapsto c]}(F) = \mathfrak{M}_{\pi',\psi'}(\forall(u:T)\,F)\end{aligned}$$

Ainsi, pour toutes les formules F dans Γ, $\textsc{Dec}^\circ(F)$ est satisfait dans $\mathfrak{I}$. □

Théorème 6.4.3 (Complétude de Dec)**.** *Pour tout ensemble* Γ *de formules avec sortes protégées, si* $\textsc{Dec}^\circ(\Gamma)$ *est satisfiable alors* Γ *est satisfiable.*

Démonstration. Soit $\mathfrak{M}$ un modèle de $\textsc{Dec}^\circ(\Gamma)$. Nous allons construire une interprétation $\mathfrak{I}$ de Γ telle que, pour toutes les valuations de type π et pour toutes les valuations ψ dans $\mathfrak{I}$, il existe une valuation ψ' dans le modèle original $\mathfrak{M}$ telle que :

$$\mathfrak{I}_{\pi,\psi}(t) = \mathfrak{M}_{\psi'}(\textsc{Dec}(t)) \qquad\qquad \mathfrak{I}_{\pi,\psi}(F) = \mathfrak{M}_{\psi'}(\textsc{Dec}(F))$$

pour tous termes t et formule F apparaissant dans Γ. Puisque $\textsc{Dec}^\circ(F)$ est la clôture par quantification universelle des variables libres de $\textsc{Dec}(F)$, la seconde égalité suffit pour montrer que $\mathfrak{I}_\pi(F)$ est vrai pour tous les $F \in \Gamma$.

Pour toutes les sortes S dans la signature de Γ, on pose :

$$\mathcal{D}_S^{\mathfrak{I}} \triangleq \begin{cases} \mathcal{D}_S^{\mathfrak{M}} & \text{si } S \text{ est protégé,} \\ \mathtt{deco}^{\mathfrak{M}}_{\langle \mathsf{T},\mathsf{U},\mathsf{D}\rangle}(\mathfrak{M}(\textsc{Dec}(S)), \mathcal{D}_{\mathsf{U}}^{\mathfrak{M}}) & \text{sinon,} \end{cases}$$

Dans le deuxième cas, on utilise l'image de $\mathtt{deco}^{\mathfrak{M}}$ sur le domaine de U. On omet les valuations quand on évalue $\textsc{Dec}(S)$, puisque ces termes ne contiennent pas de variables. Remarquons que pour toutes les sortes non-protégées S, $\mathcal{D}_S^{\mathfrak{I}} \subseteq \mathcal{D}_{\mathsf{D}}^{\mathfrak{M}}$.

Considérons un symbole de fonction f de Γ ayant pour signature $\mathbf{S} = \langle S_1, \ldots, S_n, S\rangle$. Soit $\mathbf{S}'$ la signature de $\hat{f}$, $\langle [S_1]^-, \ldots, [S_n]^-, [S]^+\rangle$. Le symbole f est interprété dans $\mathfrak{I}$ sur le vecteur de sorte $\mathbf{S}\tau$ comme suit :

$$f^{\mathfrak{I}}_{\mathbf{S}\tau}(\mathbf{c}) \triangleq \begin{cases} \hat{f}^{\mathfrak{M}}_{\mathbf{S}'}(\mathbf{c}) & \text{si } S \text{ est protégée,} \\ \mathtt{deco}^{\mathfrak{M}}_{\langle \mathsf{T},\mathsf{U},\mathsf{D}\rangle}(\mathfrak{M}(\textsc{Dec}(S\tau)), \hat{f}^{\mathfrak{M}}_{\mathbf{S}'}(\mathbf{c})) & \text{sinon.} \end{cases}$$

Notons que tous les arguments c_i de $\mathbf{c}$ appartiennent soit à $\mathcal{D}_{S_i}^{\mathfrak{M}}$ (si la sorte S_i est protégée) ou à $\mathcal{D}_{\mathsf{D}}^{\mathfrak{M}}$ (si elle ne l'est pas). Et donc cette interprétation est bien définie.

De plus, tous les symboles de prédicat p de type $\mathbf{S} = \langle S_1, \ldots, S_n\rangle$ sont interprétés dans $\mathfrak{I}$ sur $\mathbf{S}\tau$ comme $\hat{p}^{\mathfrak{M}}_{\langle [S_1]^-, \ldots, [S_n]^-\rangle}$ sur les mêmes arguments. Cela conclut la définition de l'interprétation cI.

Maintenant considérons une valuation de type π et une valuation ψ dans $\mathfrak{I}$. Pour toutes les variables apparaissant dans $\textsc{Dec}^\circ(\Gamma)$, on définit (en supposant que S est une sorte protégée et T est un type non-protégé) :

$$\begin{aligned} \psi'(v^\alpha : \mathsf{T}) &\triangleq \mathfrak{M}(\textsc{Dec}(\alpha\pi)) \\ \psi'(u_S : S) &\triangleq \psi(u : S) \\ \psi'(u_T : \mathsf{U}) &\triangleq b_{\psi(u:T)} \end{aligned}$$

où, pour tout $a \in \mathcal{D}_{T\pi}^{\mathfrak{I}}$, b_a est un élément fixé mais arbitraire de $\mathcal{D}_{\mathsf{U}}^{\mathfrak{M}}$ tel que

$$\mathtt{deco}^{\mathfrak{M}}_{\langle \mathsf{T},\mathsf{U},\mathsf{D}\rangle}(\mathfrak{M}(\textsc{Dec}(T\pi)), b_a) = a$$

Par définition de $\mathcal{D}_{T\pi}^{\mathfrak{I}}$, il existe au moins un tel élément. Remarquons que puisque T est non-protégé, $T\pi$ est non-protégé, aussi. Donc $\mathfrak{M}_{\psi'}(\textsc{Dec}(T)) = \mathfrak{M}(\textsc{Dec}(T\pi))$.

La preuve de l'égalité précédente procède par induction sur la structure des termes et formules. On considère trois cas : une variable, une application de fonctions, et une formule universellement quantifiée :

1. Soit $u : T$ une variable apparaissant dans Γ. Si T est une sorte protégée, nous avons $\mathfrak{I}_{\pi,\psi}(u : T) = \psi(u : T) = \psi'(u_T : T) = \mathfrak{M}_{\psi'}(\textsc{Dec}(u : T))$. Si T est un type non

protégé, on obtient :

$$\begin{aligned}\mathfrak{I}_{\pi,\psi}(u:T) = \psi(u:T) &= \mathtt{deco}^{\mathfrak{M}}_{\langle \mathsf{T},\mathsf{U},\mathsf{D}\rangle}(\mathfrak{M}(\textsc{Dec}(T\pi)), b_{\psi(u:T)}) \\ &= \mathtt{deco}^{\mathfrak{M}}_{\langle \mathsf{T},\mathsf{U},\mathsf{D}\rangle}(\mathfrak{M}_{\psi'}(\textsc{Dec}(T)), \psi'(u_T:\mathsf{U})) \\ &= \mathfrak{M}_{\psi'}(\mathtt{deco}(\textsc{Dec}(T), u_T:\mathsf{U}):\mathsf{D}) = \mathfrak{M}_{\psi'}(\textsc{Dec}(u:T))\end{aligned}$$

2. Considérons un terme $t = f(t_1:T_1,\ldots,t_n:T_n):T$ dans Γ. Soit $\mathbf{S} = \langle S_1,\ldots,S_n,S\rangle$ une signature de type de f et $\mathbf{S}'$ la signature de type de $\hat{f}$. Si S est une sorte protégée, alors f est interprété comme $\langle T_1,\ldots,T_n,T\rangle\pi$ dans $\mathfrak{I}$ exactement comme $\hat{f}$ sur $\mathbf{S}'$ dans $\mathfrak{M}$, ainsi $\mathfrak{I}_{\pi,\psi}(t) = \mathfrak{M}_{\psi'}(\textsc{Dec}(t))$. Supposons que S ne soit pas protégée. Si on dénote la liste des arguments par $\mathbf{t}$, on a alors :

$$\begin{aligned}\mathfrak{I}_{\pi,\psi}(t) = f^{\mathfrak{I}}_{\mathbf{T}\pi}(\mathfrak{I}_{\pi,\psi}(\mathbf{t})) = \mathtt{deco}^{\mathfrak{M}}_{\langle \mathsf{T},\mathsf{U},\mathsf{D}\rangle}(\mathfrak{M}(\textsc{Dec}(T\pi)), \hat{f}^{\mathfrak{M}}_{\mathbf{S}'}(\mathfrak{I}_{\pi,\psi}(\mathbf{t}))) = \\ \mathtt{deco}^{\mathfrak{M}}_{\langle \mathsf{T},\mathsf{U},\mathsf{D}\rangle}(\mathfrak{M}_{\psi'}(\textsc{Dec}(T)), \hat{f}^{\mathfrak{M}}_{\mathbf{S}'}(\mathfrak{M}_{\psi'}(\textsc{Dec}(\mathbf{t})))) = \mathfrak{M}_{\psi'}(\textsc{Dec}(t))\end{aligned}$$

3. Considérons une formule universelle $\forall(u:T)\,F$. Le cas avec T protégé est trivial, donc on suppose T est non-protégé. Soit b et b' les deux éléments de $\mathcal{D}^{\mathfrak{M}}_{\mathsf{U}}$ tels que $\mathtt{deco}^{\mathfrak{M}}_{\langle \mathsf{T},\mathsf{U},\mathsf{D}\rangle}(\mathfrak{M}(\textsc{Dec}(T\pi)), b) = \mathtt{deco}^{\mathfrak{M}}_{\langle \mathsf{T},\mathsf{U},\mathsf{D}\rangle}(\mathfrak{M}(\textsc{Dec}(T\pi)), b')$. On peut montrer que $\mathfrak{M}_{\psi'[u_T:\mathsf{U}\mapsto b]}(\textsc{Dec}(F)) = \mathfrak{M}_{\psi'[u_T:\mathsf{U}\mapsto b']}(\textsc{Dec}(F))$. En effet la variable $u_T:\mathsf{U}$ peut apparaître dans $\textsc{Dec}(F)$ seulement dans les termes $\mathtt{deco}(\textsc{Dec}(T), u_T:\mathsf{U}):\mathsf{D}$. Puisque $\textsc{Dec}(F)$ ne contient pas de quantificateurs sur T, le terme $\textsc{Dec}(T)$ s'évaluera toujours en $\mathfrak{M}(\textsc{Dec}(T\pi))$.
Soit a un élément de $\mathcal{D}^{\mathfrak{I}}_{T\pi}$. La valuation de type π et la valuation $\psi[u:T\mapsto a]$ produit dans $\mathfrak{M}$ la valuation $\psi'[u_T:\mathsf{U}\mapsto b_a]$.

$$\begin{aligned}\mathfrak{I}_{\pi,\psi}(\forall(u:T)\,F) &= \bigwedge_{a\in\mathcal{D}^{\mathfrak{I}}_{T\pi}} \mathfrak{I}_{\pi,\psi[u:T\mapsto a]}(F) \\ &= \bigwedge_{a\in\mathcal{D}^{\mathfrak{I}}_{T\pi}} \mathfrak{M}_{\psi'[u_T:\mathsf{U}\mapsto b_a]}(\textsc{Dec}(F)) \\ &= \bigwedge_{b\in\mathcal{D}^{\mathfrak{M}}_{\mathsf{U}}} \mathfrak{M}_{\psi'[u_T:\mathsf{U}\mapsto b]}(\textsc{Dec}(F)) \\ &= \mathfrak{M}_{\psi'}(\forall(u_T:\mathsf{U})\,\textsc{Dec}(F)) = \mathfrak{M}_{\psi'}(\textsc{Dec}(\forall(u:T)\,F))\end{aligned}$$

Donc pour toutes les formules F dans Γ, F est satisfait sous $\mathfrak{I}$. □

6.5. Encodage par explicitation

La transformation Exp est similaire à Dec excepté qu'au lieu d'attacher l'annotation de type à tous les termes, on ajoute des arguments représentant des types aux symboles

polymorphes. Cela permet des modifications plus légères dans le problème original. Cependant cette méthode est correcte seulement sur les problèmes qui admettent un modèle où toutes les sortes non-protégées ont un domaine infini.

On introduit deux constantes fraîches U et T. La première remplace les types non-protégés et la seconde, comme dans Dec, est la sorte des termes représentant les types. Pour tout type T, on définit $[T]$ comme T si T est protégé, et U sinon. Exp modifie ainsi la signature des théories transformées :

1. L'ensemble des constructeurs de type est étendu par U et T.
2. Soit f un symbole de fonction ayant pour signature $\mathbf{S} = \langle S_1, \ldots, S_n, S\rangle$. Soit $\alpha_1, \ldots, \alpha_r$ les variables libres de $\mathbf{S}$. On remplace f par un symbole de fonction $\hat{f}$ d'arité $r+n$ avec la signature de type monomorphe $\langle \mathsf{T}, \ldots, \mathsf{T}, [S_1], \ldots, [S_n], [S]\rangle$.
3. Pour tout symbole de variable u de type T, on ajoute un nouveau symbole de variable u_T.
4. Pour toute variable de type $\alpha \in \mathbb{V}_{\mathrm{T}}$, on ajoute un nouveau symbole de variable v^α.
5. Pour tout constructeur de type $\mathsf{F} \in \mathtt{FT}$, on ajoute un nouveau symbole de fonction $\mathtt{F}$ de la même arité et avec la signature de type $\langle \mathsf{T}, \ldots, \mathsf{T}, \mathsf{T}\rangle$.

La transformation Exp s'applique sur les termes non-protégés, les convertissant en termes de type T, exactement comme Dec :

$$\textsc{Exp}(\alpha) \triangleq v^\alpha : \mathsf{T} \qquad \textsc{Exp}(\mathsf{F}(T_1, \ldots, T_n)) \triangleq \mathtt{F}(\textsc{Exp}(T_1), \ldots, \textsc{Exp}(T_n)) : \mathsf{T}$$

La transformation Exp s'applique sur les termes et les formules. Dans la définition suivante, $\mathbf{t}$ est une liste de termes ; $\alpha_1, \ldots, \alpha_r$ sont les variables de types apparaissant dans la signature de type de f et p ; la signature de type de f (ou de p) est instanciée par la substitution de type τ ; et $\beta_1, \ldots, \beta_m$ sont les variables de type de H :

$$\begin{aligned}
\textsc{Exp}(u : T) &\triangleq u_T : [T] \\
\textsc{Exp}(f(\mathbf{t}) : T) &\triangleq \hat{f}(\textsc{Exp}(\alpha_1\tau), \ldots, \textsc{Exp}(\alpha_r\tau), \textsc{Exp}(\mathbf{t})) : [T] \\
\textsc{Exp}(p(\mathbf{t})) &\triangleq \hat{p}(\textsc{Exp}(\alpha_1\tau), \ldots, \textsc{Exp}(\alpha_r\tau), \textsc{Exp}(\mathbf{t})) \\
\textsc{Exp}(t_1 \approx t_2) &\triangleq \textsc{Exp}(t_1) \approx \textsc{Exp}(t_2) \\
\textsc{Exp}(\neg F) &\triangleq \neg\, \textsc{Exp}(F) \\
\textsc{Exp}(F \wedge G) &\triangleq \textsc{Exp}(F) \wedge \textsc{Exp}(G) \\
\textsc{Exp}(\forall x\, F) &\triangleq \forall(\textsc{Exp}(x))\, \textsc{Exp}(F) \\
\textsc{Exp}^\circ(H) &\triangleq \forall(v^{\beta_1} : \mathsf{T}) \ldots \forall(v^{\beta_m} : \mathsf{T})\, \textsc{Exp}(H)
\end{aligned}$$

Exemple Supposons $U = \{\mathsf{I}\}$. La transformation Par et $\textsc{Exp}^\circ$ produit la formules suivantes (par clarté on omet les indices des variables ajoutés par Par) :

$$\begin{aligned}
&\forall(v^\alpha : \mathsf{T})\, \forall(m_{\mathsf{M}(\alpha,\mathsf{I})} : \mathsf{U})\, \forall(c_\alpha : \mathsf{U})\, \mathtt{get}_{[\mathsf{I}/\beta]}(v^\alpha, \\
&\qquad\qquad \mathtt{set}_{[\mathsf{I}/\beta]}(v^\alpha, m_{\mathsf{M}(\alpha,\mathsf{I})}, c_\alpha, \mathsf{6}), c_\alpha) * \mathsf{7} \approx \mathsf{42} \\
&\forall(m_{\mathsf{M}(\mathsf{I},\mathsf{I})} : \mathsf{U})\, \forall(c_{\bar{\mathsf{I}}} : \bar{\mathsf{I}})\, \mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(\mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(m_{\mathsf{M}(\mathsf{I},\mathsf{I})}, c_{\bar{\mathsf{I}}}, \mathsf{6}), c_{\bar{\mathsf{I}}}) * \mathsf{7} \approx \mathsf{42}
\end{aligned}$$

alors que Tw et Exp° produisent :

$$\forall(v^\alpha : \mathsf{T})\,\forall(m_{\mathsf{M}(\alpha,\mathsf{I})} : \mathsf{U})\,\forall(c_\alpha : \mathsf{U})\,\mathtt{from}_{\mathsf{I}}(\mathtt{get}(v^\alpha, \mathtt{I}, \\ \mathtt{set}(v^\alpha, \mathtt{I}, m_{\mathsf{M}(\alpha,\mathsf{I})}, c_\alpha, \mathtt{to}_{\mathsf{I}}(6)), c_\alpha)) * 7 \approx 42$$

les axiomes de ponts sont inchangés à part au niveau des types non-protégés. Les transformations Dis + Tw + Exp, avec $W = \{\mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}, \mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}\}$ et $U = \{\mathsf{I}, M(\mathsf{I},\mathsf{I})\}$, produisent trois formules en plus : deux axiomes de pont et la même formule monomorphe 6.3 que Dis + Tw + Dec°.

Lemme 6.5.1. *Pour tous les termes t de type T,* Exp(t) *est un terme monomorphe bien formé de type* $[T]$*. Pour toutes les formules F, $EXP(F)$ est une formule monomorphe bien formée. De plus* $\mathrm{FV}(\mathrm{Exp}(t)) = \{u_T : [T] \mid u : T \in \mathrm{FV}(t)\} \cup \{v^\alpha : \mathsf{T} \mid \alpha \in \mathrm{FV}_{\mathrm{T}}(t)\}$ *et* $\mathrm{FV}(\mathrm{Exp}(F)) = \{u_T : [T] \mid u : T \in \mathrm{FV}(F)\} \cup \{v^\alpha : \mathsf{T} \mid \alpha \in \mathrm{FV}_{\mathrm{T}}(F)\}$*. Enfin pour toutes les formules closes F,* Exp°(F) *est une formule monomorphe close bien formée.*

Démonstration. La dernière affirmation se déduit facilement de la seconde. La preuve des deux premières affirmations se déroule par récurrence structurelle sur les termes et les formules. Le seul cas intéressant est l'application des symboles de fonctions (les symboles de prédicats sont traités de la même manière).

Considérons un terme $t = f(t_1 : T_1, \ldots, t_n : T_n) : T$. Soit S_i la $i^{\text{ième}}$ composante dans la signature de f. On a deux cas possibles : soit S_i et T_i sont protégés, et $S_i = T_i$, ou S_i et T_i sont tous deux non-protégés, par définition de la correspondance des types protégés. Ainsi, $[T_i] = [S_i]$ pour tout $i \in [1, n]$.

Maintenant soit S le type de la valeur dans la signature de f. Par les mêmes arguments, soit S et T sont protégés, et $S = T$, soit S et T sont tous deux non-protégés. Ainsi, $[T] = [S]$ et l'application est bien formée.

Finalement toutes les variables de type α présentes dans $\langle T_1, \ldots, T_n, T\rangle$ apparaissent comme variable libre $v^\alpha : \mathsf{T}$ dans les arguments supplémentaires de "type" de $\hat{f}$ dans Exp(t). □

Théorème 6.5.2 (Correction de Exp)**.** *Si un ensemble* Γ *de formules closes avec des sortes protégées est satisfiable de telle manière que toutes les sortes non-protégées ont un domaine infini dans le modèle, alors* Exp°(Γ) *est satisfiable.*

Démonstration. Tout modèle de Γ est un modèle de l'ensemble des instances de type monomorphes MI(Γ) et inversement. Dans cet ensemble les sortes protégées ne requièrent pas de traitement spécial et l'ensemble du problème se réduit à la logique multi-sortée. Soit $\mathfrak{M}$ un modèle de Γ tel que, pour toutes les sortes non-protégées S, le domaine $\mathcal{D}_S^{\mathfrak{M}}$ est infini. Par le théorème de Löwenheim-Skolem pour la logique multi-sortée [42, Sect. 7.6], on peut supposer que le domaine de toutes les sortes non-protégées est simplement $\mathbb{N}$.

Nous allons construire une interprétation $\mathfrak{I}$ de Exp°(Γ) telle que, pour toute valuation ψ dans $\mathfrak{I}$, il existe une valuation de type π' et une valuation ψ' dans le modèle initial $\mathfrak{M}$, tel que :

$$\mathfrak{I}_\psi(\mathrm{Exp}(t)) = \mathfrak{M}_{\pi',\psi'}(t) \qquad\qquad \mathfrak{I}_\psi(\mathrm{Exp}(F)) = \mathfrak{M}_{\pi',\psi'}(F)$$

pour tout terme t et toute formule F apparaissant dans Γ. Nous ne considérons pas une évaluation de type $\mathfrak{I}$, puisque le résultat de $\textsc{Exp}^\circ$ est entièrement monomorphe. Une fois ces deux égalités prouvées il est aisé de montrer que pour toute formule close $H \in \Gamma$, l'interprétation $\mathfrak{I}$ satisfait $\textsc{Exp}^\circ(H)$. En fait,

$$\mathfrak{I}(\textsc{Exp}^\circ(H)) = \bigwedge_\psi \mathfrak{I}_\psi(\textsc{Exp}(H)) = \bigwedge_\psi \mathfrak{M}_{\pi',\psi'}(H) = \top$$

Pour toute sorte S dans la signature initiale, protégée ou pas, on garde son domaine : $\mathcal{D}^{\mathfrak{I}}_S \triangleq \mathcal{D}^{\mathfrak{M}}_S$. Le domaine de U est l'ensemble des entiers naturels : $\mathcal{D}^{\mathfrak{I}}_{\mathsf{U}} \triangleq \mathbb{N}$. Le domaine de T est l'ensemble des sortes non-protégées : $\mathcal{D}^{\mathfrak{I}}_{\mathsf{T}} \triangleq \mathcal{T}(\mathtt{FT})$.

Ensuite on donne l'interprétation des symboles de fonctions et de prédicats dans $\textsc{Exp}^\circ(\Gamma)$. Soit f un symbole de fonction dans Γ de signature de type $\mathbf{S} = \langle S_1, \ldots, S_n, S\rangle$ et $\alpha_1, \ldots, \alpha_r$ sont les variables de type de $\mathbf{S}$. Le symbole $\hat{f}$ possède l'arité $r+n$ et la signature de type $\mathbf{S}' = \langle \mathsf{T}, \ldots, \mathsf{T}, [S_1], \ldots, [S_n], [S]\rangle$ et c'est le seul vecteur de sorte sur lequel $\hat{f}$ est interprété, puisque il est monomorphe. On définit :

$$\hat{f}^{\mathfrak{I}}_{\mathbf{S}'}(T_1, \ldots, T_r, c_1, \ldots, c_n) \triangleq f^{\mathfrak{M}}_{\mathbf{S}\tau}(c_1, \ldots, c_n)$$

où τ est la substitution de type $[T_1/\alpha_1, \ldots, T_r/\alpha_r]$. Remarquons que tout type S dans $\mathbf{S}$ vérifie $\mathcal{D}^{\mathfrak{I}}_{[S]} = \mathcal{D}^{\mathfrak{M}}_{S\tau}$. En fait, soit S est protégé et $[S] = S = S\tau$, ou à la fois S et $S\tau$ sont non protégés et $\mathcal{D}^{\mathfrak{I}}_{[S]} = \mathcal{D}^{\mathfrak{I}}_{\mathsf{U}} = \mathbb{N} = \mathcal{D}^{\mathfrak{M}}_{S\tau}$.

Similairement, soit p un symbole de prédicat dans Γ ayant pour signature de type $\mathbf{S} = \langle S_1, \ldots, S_n\rangle$ et soient $\alpha_1, \ldots, \alpha_r$ les variables de type dans $\mathbf{S}$. Le symbole $\hat{p}$ est d'arité $r+n$ et possède la signature de type $\mathbf{S}' = \langle \mathsf{T}, \ldots, \mathsf{T}, [S_1], \ldots, [S_n]\rangle$. Alors on pose :

$$\hat{p}^{\mathfrak{I}}_{\mathbf{S}'}(T_1, \ldots, T_r, c_1, \ldots, c_n) \triangleq p^{\mathfrak{M}}_{\mathbf{S}[T_1/\alpha_1, \ldots, T_r/\alpha_r]}(c_1, \ldots, c_n)$$

Finalement les symboles de fonctions représentant des types sont interprétés exactement comme les constructeurs originaux :

$$\mathsf{F}^{\mathfrak{I}}_{\langle \mathsf{T}, \ldots, \mathsf{T}\rangle}(T_1, \ldots, T_n) \triangleq \mathsf{F}(T_1, \ldots, T_n)$$

Maintenant on considère une valuation ψ dans $\mathfrak{I}$. Pour toute variable α, on pose $\alpha\pi' \triangleq \psi(v^\alpha : \mathsf{T})$. Pour toutes variables $u : T$ dans Γ, on pose $\psi'(u : T) \triangleq \psi(u_T : [T])$. Comme nous avons vu précédemment, $\mathcal{D}^{\mathfrak{M}}_{T\pi'} = \mathcal{D}^{\mathfrak{I}}_{[T]}$. De plus, pour tout type non-protégé T, on a $\mathfrak{I}_\psi(\textsc{Exp}(T)) = T\pi'$.

La preuve des deux égalités se fait par induction sur la structure des termes et des formules. On considère trois cas : une variable, une application de fonction et une formule universellement quantifiée.

1. Soit $u : T$ une variable de Γ. On obtient :

$$\mathfrak{I}_\psi(\textsc{Exp}(u : T)) = \mathfrak{I}_\psi(u_T : [T]) = \psi(u_T : [T]) = \psi'(u : T) = \mathfrak{M}_{\pi',\psi'}(u : T)$$

2. Considérons un terme $t = f(t_1 : T_1, \ldots, t_n : T_n) : T$. Soit $\mathbf{S} = \langle S_1, \ldots, S_n, S\rangle$ la signature de type de f et $\alpha_1, \ldots, \alpha_r$ les variables de type de $\mathbf{S}$. Soit $\mathbf{S}'$ une signature de type de $\hat{f}$. Finalement soit τ la substitution de type qui instancie $\mathbf{S}$ vers $\langle T_1, \ldots, T_n, T\rangle$. On obtient ainsi :

$$\begin{aligned}\mathfrak{I}_\psi(\textsc{Exp}(t)) &= \mathfrak{I}_\psi(\hat{f}(\textsc{Exp}(\alpha_1\tau), \ldots, \textsc{Exp}(\alpha_r\tau), \textsc{Exp}(t_1), \ldots, \textsc{Exp}(t_n)) : [S])\\ &= \hat{f}^{\mathfrak{I}}_{\mathbf{S}'}(\alpha_1\tau\pi', \ldots, \alpha_r\tau\pi', \mathfrak{M}_{\pi',\psi'}(t_1), \ldots, \mathfrak{M}_{\pi',\psi'}(t_n))\\ &= f^{\mathfrak{M}}_{\mathbf{S}\tau\pi'}(\mathfrak{M}_{\pi',\psi'}(t_1), \ldots, \mathfrak{M}_{\pi',\psi'}(t_n)) = \mathfrak{M}_{\pi',\psi'}(t)\end{aligned}$$

3. On considère une formule universellement quantifiée $\forall(u : T)\, F$. Prenons un élément de $c \in \mathcal{D}^{\mathfrak{I}}_{[T]} = \mathcal{D}^{\mathfrak{M}}_{T\pi'}$. La valuation $\psi[u_T : [T] \mapsto c]$ produit dans $\mathfrak{M}$ la même valuation de type π' et la valuation $\psi'[u : T \mapsto c]$.

$$\begin{aligned}\mathfrak{I}_\psi(\textsc{Exp}(\forall(u : T)\, F)) &= \bigwedge_{c \in \mathcal{D}^{\mathfrak{I}}_{[T]}} \mathfrak{I}_{\psi[u_T : [T] \mapsto c]}(\textsc{Exp}(F))\\ &= \bigwedge_{c \in \mathcal{D}^{\mathfrak{M}}_{T\pi'}} \mathfrak{M}_{\pi',\psi'[u : T \mapsto c]}(F) = \mathfrak{M}_{\pi',\psi'}(\forall(u : T)\, F)\end{aligned}$$

Ainsi pour toute formule F de Γ, $\textsc{Exp}^\circ(F)$ est satisfait dans $\mathfrak{I}$. □

Théorème 6.5.3 (Complétude de Exp)**.** *Pour tout ensemble* Γ *de formules closes avec des sortes protégées, si* $\textsc{Exp}^\circ(\Gamma)$ *est satisfiable alors* Γ *est satisfiable.*

Démonstration. Soit $\mathfrak{M}$ un modèle de $\textsc{Exp}^\circ(\Gamma)$. Nous allons construire une interprétation $\mathfrak{I}$ de Γ telle que, pour toute valuation de type π et toute valuation ψ dans $\mathfrak{I}$ il existe une valuation ψ' dans le modèle initial $\mathfrak{M}$, telle que :

$$\mathfrak{I}_{\pi,\psi}(t) = \mathfrak{M}_{\psi'}(\textsc{Exp}(t)) \qquad\qquad \mathfrak{I}_{\pi,\psi}(F) = \mathfrak{M}_{\psi'}(\textsc{Exp}(F))$$

pour tout terme t et formule F apparaissant dans Γ. Puisque $\textsc{Exp}^\circ(F)$ est la quantification universelle de $\textsc{Exp}(F)$, la seconde égalité suffit à montrer que $\mathfrak{I}_\pi(F)$ pour tout $F \in \Gamma$.

Pour toute sorte S dans la signature de Γ, on choisit $\mathcal{D}^{\mathfrak{I}}_S \triangleq \mathcal{D}^{\mathfrak{M}}_{[S]}$.

Considérons un symbole de fonction f de Γ, ayant la signature de type $\mathbf{S} = \langle S_1, \ldots, S_n, S\rangle$. Soient $\alpha_1, \ldots, \alpha_r$ les variables de type de $\mathbf{S}$. On pose $\mathbf{S}'$, la signature de type de $\hat{f}$, pour $\langle \mathsf{T}, \ldots, \mathsf{T}, [S_1], \ldots, [S_n], [S]\rangle$. Le symbole f est alors interprété par $\mathfrak{I}$ sur le vecteur de sorte $\mathbf{S}\tau$ comme suit :

$$f^{\mathfrak{I}}_{\mathbf{S}\tau}(\mathbf{c}) \triangleq \hat{f}^{\mathfrak{M}}_{\mathbf{S}'}(\mathfrak{M}(\textsc{Exp}(\alpha_1\tau)), \ldots, \mathfrak{M}(\textsc{Exp}(\alpha_r\tau)), \mathbf{c}).$$

Nous omettons la valuation quand on évalue $\textsc{Exp}(\alpha_i\tau)$ puisque les termes ne contiennent pas de variables. Remarquons que tous les arguments c_i de $\mathbf{c}$ appartiennent soit à $\mathcal{D}^{\mathfrak{M}}_{S_i}$ (si la sorte S_i est protégée), soit à $\mathcal{D}^{\mathfrak{M}}_{\mathsf{U}}$ (s'il ne l'est pas), et donc l'interprétation est bien définie.

De même soit p un symbole de prédicat dans Γ ayant la signature de type $\mathbf{S} = \langle S_1, \ldots, S_n \rangle$. Soient $\alpha_1, \ldots, \alpha_r$ les variables de type de $\mathbf{S}$ et $\mathbf{S}'$ la signature de $\hat{p}$. Le symbole p est interprété par $\mathfrak{I}$ sur un vecteur de sortes $\mathbf{S}\tau$ comme suit :

$$p^{\mathfrak{I}}_{\mathbf{S}\tau}(\mathbf{c}) \triangleq \hat{p}^{\mathfrak{M}}_{\mathbf{S}'}(\mathfrak{M}(\textsc{Exp}(\alpha_1\tau)), \ldots, \mathfrak{M}(\textsc{Exp}(\alpha_r\tau)), \mathbf{c}).$$

Cela conclut la définition de $\mathfrak{I}$.

Maintenant considérons une valuation de type π et une valuation ψ dans $\mathfrak{I}$. Pour toutes les variables apparaissant dans $\textsc{Exp}^{\circ}(\Gamma)$ on définit :

$$\psi'(v^{\alpha} : \mathsf{T}) \triangleq \mathfrak{M}(\textsc{Exp}(\alpha\pi)) \qquad\qquad \psi'(u_T : [T]) \triangleq \psi(u : T)$$

Remarquons que $\mathfrak{M}_{\psi'}(\textsc{Exp}(T)) = \mathfrak{M}(\textsc{Exp}(T\pi))$ et $\mathcal{D}^{\mathfrak{I}}_{T\pi} = \mathcal{D}^{\mathfrak{M}}_{[T]}$.

La preuve des égalités se déroule par récurrence sur la structure des termes et des formules. On considère trois cas : les variables, les application de symbole de fonction, et les formules quantifiées universellement.

1. Soit $u : T$ une variable qui apparaît dans Γ. On obtient :

$$\mathfrak{I}_{\pi,\psi}(u : T) = \psi(u : T) = \psi'(u_T : [T]) = \mathfrak{M}_{\psi'}(u_T : [T]) = \mathfrak{M}_{\psi'}(\textsc{Exp}(u : T))$$

2. Considérons un terme $t = f(t_1 : T_1, \ldots, t_n : T_n) : T$ dans Γ. Soit $\mathbf{S} = \langle S_1, \ldots, S_n, S \rangle$ la signature de f, soit $\alpha_1, \ldots, \alpha_r$ une variable de type de $\mathbf{S}$, soit $\mathbf{S}'$ la signature de type de $\hat{f}$, et τ la substitution qui instancie $\mathbf{S}$ en $\langle T_1, \ldots, T_n, T \rangle$. On dénote la liste des arguments avec $\mathbf{t}$, on a :

$$\begin{aligned}
\mathfrak{I}_{\pi,\psi}(t) &= f^{\mathfrak{I}}_{\mathbf{S}\tau\pi}(\mathfrak{I}_{\pi,\psi}(\mathbf{t})) \\
&= \hat{f}^{\mathfrak{M}}_{\mathbf{S}'}(\mathfrak{M}(\textsc{Exp}(\alpha_1\tau\pi)), \ldots, \mathfrak{M}(\textsc{Exp}(\alpha_r\tau\pi)), \mathfrak{I}_{\pi,\psi}(\mathbf{t})) \\
&= \hat{f}^{\mathfrak{M}}_{\mathbf{S}'}(\mathfrak{M}_{\psi'}(\textsc{Exp}(\alpha_1\tau)), \ldots, \mathfrak{M}_{\psi'}(\textsc{Exp}(\alpha_r\tau)), \mathfrak{M}_{\psi'}(\textsc{Exp}(\mathbf{t}))) \\
&= \mathfrak{M}_{\psi'}(\hat{f}(\textsc{Exp}(\alpha_1\tau), \ldots, \textsc{Exp}(\alpha_r\tau), \textsc{Exp}(\mathbf{t})) : [S]) = \mathfrak{M}_{\psi'}(\textsc{Exp}(t))
\end{aligned}$$

3. Considérons une formule universelle $\forall(u : T)\, F$. Soit a un élément de $\mathcal{D}^{\mathfrak{I}}_{T\pi}$. La valuation de type π et la valuation $\psi[u : T \mapsto a]$ produisent dans $\mathfrak{M}$ la valuation $\psi'[u_T : [T] \mapsto a]$. On obtient :

$$\begin{aligned}
\mathfrak{I}_{\pi,\psi}(\forall(u : T)\, F) &= \bigwedge_{a \in \mathcal{D}^{\mathfrak{I}}_{T\pi}} \mathfrak{I}_{\pi,\psi[u : T \mapsto a]}(F) \\
&= \bigwedge_{a \in \mathcal{D}^{\mathfrak{M}}_{[T]}} \mathfrak{M}_{\psi'[u_T : [T] \mapsto a]}(\textsc{Exp}(F)) \\
&= \mathfrak{M}_{\psi'}(\forall(u_T : [T])\, \textsc{Exp}(F)) = \mathfrak{M}_{\psi'}(\textsc{Exp}(\forall(u : T)\, F))
\end{aligned}$$

Ainsi, pour toutes les formules F dans Γ, F est satisfaite dans $\mathfrak{I}$.

□

6.6. Élimination des domaines finis

Comme nous l'avons vu dans la section précédente, l'encodage par explicitation requiert que les types non-protégés aient un domaine infini. Pour s'assurer de cela, on ne peut simplement choisir de protéger tous les types finis. Par exemple, la protection par ponts Tw n'est pas suffisante car la sorte protégée est en bijection avec la sorte non-protégée. De plus, si quelqu'un ajoute au type des booléens un type fantôme de la manière suivante,

```
type bool_poly α

function true : bool_poly α
function false : bool_poly α

axiom finite : ∀ x : bool_poly α. x = true ∨ x = false
```

il faut protéger un nombre infini de sortes, ce qui n'est pas possible.

En outre un axiome de finitude peut se cacher de manière subtile : soit $\texttt{isUnit} : \langle\alpha\rangle$ un prédicat unaire ; alors les formules

$$\forall(x:\alpha)(\texttt{isUnit}(x) \Rightarrow \forall(y:\alpha)(y \approx x))$$
$$\forall(x:\mathsf{A})\,\texttt{isUnit}(x)$$

impliquent que la sorte A possède un seul habitant alors même qu'elle n'apparaît pas dans le premier axiome. En réalité dès qu'il y a une égalité sur une variable [19], il y a un risque que le type soit fini ; ici comme c'est α tous les types peuvent l'être. On doit donc protéger, pour être correct, à la fois les formules monomorphes mais également les formules polymorphes.

Soit un ensemble Δ de formules closes. Soit un ensemble E de types polymorphes, possiblement infini. On dit que c'est un *ensemble de types infinis* pour Δ si, dans le cas où Δ est satisfiable, il existe au moins un modèle où toutes les instances des types de E sont associées à un domaine infini. Si Δ n'est pas satisfiable tout ensemble satisfait cela. Déterminer le plus grand, au sens de l'inclusion, ensemble E à partir de Δ est indécidable. Nous allons faire la supposition qu'un ensemble E nous est donné avec l'ensemble Δ. Nous discuterons à la fin de cette section comment on peut l'obtenir.

Nous allons présenter une transformation Pro qui, à partir de Δ et de E, donnera un ensemble de formules équisatisfiable à Δ mais tel que l'ensemble des types qui ne sont pas dans E est aussi un ensemble de type infinis.

La transformation Pro ajoute autour de tous les termes de Δ l'application du symbole de fonction $\texttt{proj} : \langle\alpha, \alpha\rangle$. Ainsi les contraintes de cardinalité s'appliquent sur l'image des domaines par `proj` au lieu de s'appliquer sur les domaines directement. Pour chaque type infini T appartenant à E, on pose que `proj` est l'identité sur ce type.

La transformation Pro ne modifie pas la signature de Δ, excepté l'ajout de `proj`. La

transformation Pro ne modifie pas le type des termes

$$\textsc{Pro}(u : T) = \texttt{proj}(u : T) \quad \textsc{Pro}(f(t_1, \ldots, t_n)) = \texttt{proj}(f(\textsc{Pro}(t_1), \ldots, PRO(t_1)))$$

et est un morphisme pour les connecteurs logiques :

$$\begin{aligned} \textsc{Pro}(p(t_1, \ldots, t_n)) &= p(\textsc{Pro}(t_1), \ldots, PRO(t_1)) & \textsc{Pro}(\neg F) &= \neg \textsc{Pro}(F) \\ \textsc{Pro}(t_1 \approx t_2) &= \textsc{Pro}(t_1) \approx \textsc{Pro}(t_2) & \textsc{Pro}(\forall(u : T).\, F) &= \forall(u : T).\, \textsc{Pro}(F) \\ \textsc{Pro}(t_1 \wedge t_2) &= \textsc{Pro}(t_1) \wedge \textsc{Pro}(t_2) \end{aligned}$$

Les termes et formules ainsi obtenus sont bien formés puisque l'on insère seulement des applications de `proj` qui a la signature $\langle \alpha, \alpha \rangle$. Enfin on pose que

$$\textsc{Pro}(\Delta) = \{\textsc{Pro}(d) | d \in \Delta\} \cup \{\forall x : T.\, \texttt{proj}(x) = x | T \in E\}$$

Théorème 6.6.1 (Correction de Pro)**.** *Soit un ensemble Δ de formules closes qui possèdent un modèle $\mathfrak{M}$. Soit E un ensemble de types. Alors* $\textsc{Pro}(\Delta)$ *possède un modèle $\mathfrak{I}$ tel que tous les domaines des instances des types de E ont les mêmes cardinalités que dans $\mathfrak{M}$ et tel que toutes les autres sortes possèdent un domaine infini.*

Démonstration. Soit $\mathfrak{M}$ un modèle de Γ. On va montrer qu'il existe un modèle $\mathfrak{I}$ de $\textsc{Pro}(\Gamma)$ tel que pour toute valuation de type π et valuation ψ de $\mathfrak{I}$, il existe une valuation ψ' de $\mathfrak{M}$ telle que :

$$\mathfrak{I}_{\pi,\psi}(\textsc{Pro}(t)) = (\mathfrak{M}_{\pi,\psi'}(t), 0) \qquad \mathfrak{I}_{\pi,\psi}(\textsc{Pro}(F)) = \mathfrak{M}_{\pi,\psi'}(F)$$

avec t et F des termes et formules de Δ.

On va définir $\mathfrak{I}$ à partir de $\mathfrak{M}$. Le domaine $\mathcal{D}_S^{\mathfrak{I}}$ de S, une sorte de $\mathfrak{I}$, est défini par :

$$\mathcal{D}_S^{\mathfrak{I}} = \begin{cases} \mathcal{D}_S^{\mathfrak{M}} \times \{0\} & \text{si il existe } \tau \text{ tel que } T\tau = S \text{ et } T \in E \\ \mathcal{D}_S^{\mathfrak{M}} \times \mathbb{N} & \text{sinon} \end{cases}$$

Soit S une sorte. L'interprétation de `proj` est $\texttt{proj}_S^{\mathfrak{I}}(\ (v, n)\ ) = (v, 0)$ avec $(v, n) \in \mathcal{D}_S^{\mathfrak{I}}$. Soient f un symbole de fonction et $\mathbf{S}$ un vecteur de sortes qui correspond à la signature de f. Soit $\mathbf{v} \in \mathcal{D}_{\mathbf{S}}^{\mathfrak{I}}$. L'interprétation de f dans $\mathfrak{I}$ est $f_{\mathbf{S}}^{\mathfrak{I}}(\mathbf{v}) \triangleq (f_{\mathbf{S}}^{\mathfrak{M}}(\Pi_1(\mathbf{v})), 0)$, avec Π_1 la première projection. On définit similairement l'interprétation des symboles de prédicats.

Soit π une valuation de type et soit ψ une valuation de $\mathfrak{I}$. On définit ψ' par $\psi'(u : T\pi) \triangleq \Pi_1(\psi(u : T\pi))$.

On va prouver les deux égalités par récurrence sur les termes et les formules. On considère quatre cas : les variables, les applications de fonctions, les égalités et les formules.

1. Soit $u : T$ une variable de Γ. On obtient :

$$\begin{aligned} \mathfrak{I}_{\pi,\psi}(\textsc{Pro}(u : T)) &= \mathfrak{I}_{\pi,\psi}(\texttt{proj}(u : T)) \\ &= (\Pi_1(\mathfrak{I}_{\pi,\psi}(u : T)), 0) \\ &= (\Pi_1(\psi(u : T\pi)), 0) \\ &= (\psi'(u : T\pi), 0) \\ &= (\mathfrak{M}_{\pi,\psi'}, 0) \end{aligned}$$

2. Considérons un terme $t = f(\mathbf{t} : \mathbf{T}) : T$.

$$\begin{aligned}\mathfrak{I}_{\pi,\psi}(\textsc{Pro}(f(\mathbf{t})) : T) &= \mathfrak{I}_{\pi,\psi}(\mathtt{proj}(f(\textsc{Pro}(\mathbf{t}))) : T) \\ &= (\Pi_1(f^{\mathfrak{I}}_{\mathbf{T},T}(\mathfrak{I}_{\pi,\psi}(\textsc{Pro}(\mathbf{t})))), 0) \\ &= (\Pi_1(f^{\mathfrak{M}}_{\mathbf{T},T}(\Pi_1(\mathfrak{I}_{\pi,\psi}(\mathbf{t}))), 0), 0) \\ &= (f^{\mathfrak{M}}_{\mathbf{T},T}(\mathfrak{M}_{\pi,\psi'}(\mathbf{t})), 0) \\ &= (\mathfrak{M}_{\pi,\psi'}(f(\mathbf{t} : \mathbf{T})) : T, 0)\end{aligned}$$

3. On considère une formule $F = t_1 \approx t_2$ L'interprétation $\mathfrak{I}_{\pi,\psi}(\textsc{Pro}(t_1 \approx t_2))$ est équivalent à l'égalité entre $\mathfrak{I}_{\pi,\psi}(\textsc{Pro}(t_1))$ et $\mathfrak{I}_{\pi,\psi}(\textsc{Pro}(t_2))$. L'hypothèse de récurrence indique que c'est équivalent à l'égalité entre $(\mathfrak{M}_{\pi,\psi}(t_1), 0)$ et $(\mathfrak{M}_{\pi,\psi}(t_2), 0)$ qui est équivalent à $\mathfrak{M}_{\pi,\psi}(t_1 \approx t_2)$.
4. On considère une formule universellement quantifiée $\forall(u : T)\, F$. Prenons un élément de $c \in \mathcal{D}^{\mathfrak{I}}_{T\pi}$. La valuation $\psi[u : T\pi \mapsto c]$ produit la valuation $\psi'[u : T\pi \mapsto \Pi_1(c)]$. Remarquons que $\Pi_1(\mathcal{D}^{\mathfrak{I}}_{T\pi}) = \mathcal{D}^{\mathfrak{M}}_{T\pi}$.

$$\begin{aligned}\mathfrak{I}_{\pi,\psi}(\textsc{Pro}(\forall(u : T)\, F)) &= \bigwedge_{c \in \mathcal{D}^{\mathfrak{I}}_{T\pi}} \mathfrak{I}_{\pi,\psi[u : T \mapsto c]}(\textsc{Pro}(F)) \\ &= \bigwedge_{c \in \mathcal{D}^{\mathfrak{I}}_{T\pi}} \mathfrak{M}_{\pi,\psi'[u : T \mapsto \Pi_1(c)]}(F) \\ &= \bigwedge_{c \in \mathcal{D}^{\mathfrak{M}}_{T\pi}} \mathfrak{M}_{\pi,\psi'[u : T \mapsto c]}(F) = \mathfrak{M}_{\pi,\psi'}(\forall(u : T)\, F)\end{aligned}$$

Ainsi pour toute formule F de Γ, $\textsc{Pro}(F)$ est satisfiable dans $\mathfrak{I}$. De plus pour tout type T dans E et substitution τ, $\mathcal{D}^{\mathfrak{I}}_{T\tau}$ est en bijection avec $\mathcal{D}^{\mathfrak{M}}_{T\tau}$. Enfin les autres domaines sont infinis puisqu'ils sont en surjection avec $\mathbb{N}$. □

Corollaire 6.6.2. *Soit un ensemble Δ de formules closes satisfiable. Soit E un ensemble composé par les sortes protégées d'une part et par uniquement des types infinis dans Δ d'autres part. Alors $\textsc{Pro}(\Delta)$ est satisfiable et les sortes protégées ont des domaines possédant la même cardinalité que précédemment.*

Théorème 6.6.3 (Complétude de Pro)**.** *Soit un ensemble Δ de formules closes. Soit E un ensemble de type. Si $\textsc{Pro}(\Delta)$ est satisfiable alors Δ l'est aussi.*

Démonstration. Soit $\mathfrak{M}$ un modèle de $\textsc{Pro}(\Delta)$. On va définir $\mathfrak{I}$ un modèle de Δ tel que pour toute valuation de type π et valuation ψ de $\mathfrak{I}$ il existe une valuation ψ' de $\mathfrak{M}$ telle que :

$$\mathfrak{I}_{\pi,\psi}(t) = \mathfrak{M}_{\pi,\psi'}(\textsc{Pro}(t)) \qquad\qquad \mathfrak{I}_{\pi,\psi}(F) = \mathfrak{M}_{\pi,\psi'}(\textsc{Pro}(F))$$

avec t un terme et F une formule de Δ. La seconde égalité suffit à montrer que $\mathfrak{I}_\pi(F)$ pour tout $F \in \Delta$

Le domaine de S dans $\mathfrak{I}$ est l'image de $\mathcal{D}^{\mathfrak{M}}_S$ par $\mathtt{proj}^{\mathfrak{M}}_S : \mathcal{D}^{\mathfrak{I}}_S \triangleq \mathtt{proj}^{\mathfrak{M}}_S(\mathcal{D}^{\mathfrak{M}}_S)$.

Soient un symbole de fonction f, un vecteur de sorte $\mathbf{S} = S_1, \ldots, S_n, S$, et des valeurs $v_i \in \mathcal{D}^{\mathfrak{I}}_S \subseteq \mathcal{D}^{\mathfrak{M}}_S$. L'interprétation de f est $f^{\mathfrak{I}}_{\mathbf{S}}(\mathbf{v}) = \mathtt{proj}^{\mathfrak{M}}_S(f^{\mathfrak{M}}_{\mathbf{S}}(\mathbf{v}))$. L'interprétation des symboles de prédicats est identique dans $\mathfrak{I}$ et dans $\mathfrak{M}$.

Soient une valuation de type π et une valuation ψ de $\mathfrak{I}$. On définit une valuation ψ' de $\mathfrak{M}$ par $\psi'(u:T\pi) = c$ avec $\mathtt{proj}^{\mathfrak{M}}_S(c) = \psi(u:T\pi)$. Une valeur arbitraire mais fixée c existe puisque $\mathcal{D}^{\mathfrak{M}}_{T\pi}$ est l'image par $\mathtt{proj}^{\mathfrak{M}}_{T\pi}$.

On va montrer les deux égalités par récurrence sur les termes et les formules. On considère quatre cas : les variables, les applications de fonctions, les égalités, et les formules.

1. Soit $u:T$ une variable de Γ. On obtient :

$$\begin{aligned}\mathfrak{I}_{\pi,\psi}(u:T) &= \psi(u:T\pi)\\ &= \mathtt{proj}^{\mathfrak{M}}_{T\pi}(\psi'(u:T\pi))\\ &= \mathfrak{M}_{\pi,\psi}(\mathtt{proj}(u:T))\end{aligned}$$

2. Considérons un terme $t = f(\mathbf{t}:\mathbf{T}):T$.

$$\begin{aligned}\mathfrak{I}_{\pi,\psi}(f(\mathbf{t}:\mathbf{T}):T) &= f^{\mathfrak{I}}_{\mathbf{T},T}(\mathfrak{I}_{\pi,\psi}(\mathbf{t}))\\ &= \mathtt{proj}^{\mathfrak{M}}_T(f^{\mathfrak{M}}_{\mathbf{T},T}(\mathfrak{M}_{\pi,\psi'}(\mathbf{t})))\\ &= \mathfrak{M}_{\pi,\psi}(\textsc{Pro}(f(\mathbf{t}:\mathbf{T})):T)\end{aligned}$$

3. On considère une formule $F = t_1 \approx t_2$ L'interprétation $\mathfrak{I}_{\pi,\psi}(t_1 \approx t_2)$ est équivalente à l'égalité entre $\mathfrak{I}_{\pi,\psi}(t_1)$ et $\mathfrak{I}_{\pi,\psi}(t_2)$. L'hypothèse de récurrence indique que c'est équivalent à l'égalité entre $\mathfrak{M}_{\pi,\psi}(t_1)$ et $\mathfrak{M}_{\pi,\psi}(t_2)$ qui est équivalent à $\mathfrak{M}_{\pi,\psi}(t_1 \approx t_2)$.
4. On considère une formule universellement quantifiée $\forall(u:T)\,F$. Prenons un élément de $c \in \mathcal{D}^{\mathfrak{I}}_{T\pi}$. La valuation $\psi[u:T\pi \mapsto c]$ produit une valuation $\psi'[u:T\pi \mapsto c']$, avec $\mathtt{proj}^{\mathfrak{M}}_{T\pi}(c') = c$. D'autre part soit c'' tel que $\mathtt{proj}^{\mathfrak{M}}_{T\pi}(c'') = \mathtt{proj}^{\mathfrak{M}}_{T\pi}(c')$ alors $\mathfrak{M}_{\pi,\psi'[u:T\pi\mapsto c']}(F) = \mathfrak{M}_{\pi,\psi'[u:T\pi\mapsto c']}(F)$ puisque $u:T\pi$ ne peut apparaître que sous `proj`.

$$\begin{aligned}\mathfrak{I}_{\pi,\psi}(\forall(u:T)\,F) &= \bigwedge_{c\in\mathcal{D}^{\mathfrak{I}}_{T\pi}} \mathfrak{I}_{\pi,\psi[u:T\pi\,\mapsto\,c]}(F)\\ &= \bigwedge_{c\in\mathcal{D}^{\mathfrak{M}}_{T\pi}} \mathfrak{M}_{\pi,\psi'[u:T\pi\,\mapsto\,c]}(F) = \mathfrak{M}_{\pi,\psi'}(\forall(u:T)\,F)\end{aligned}$$

Ainsi pour toute formule F de Γ, F est satisfaite dans $\mathfrak{I}$. □

Pour tous les types apparaissant dans E on a défini `proj` comme l'identité. On peut donc, sans changer aucunement la satisfiabilité de $\textsc{Pro}(\Delta)$, enlever toutes les projections dont le type correspond avec un type de E ; on doit malgré tout garder l'axiome. Si tous les types protégés sont dans E, il n'y aura pas d'instanciation de `proj` avec un type protégé. De plus dans les deux preuves précédentes la valuation de type est inchangée

pour interpréter Δ et $\textsc{Pro}(\Delta)$. Ainsi si Δ est un ensemble de formules closes avec des types protégés satisfiable alors $\textsc{Pro}(\Delta)$ l'est également. De cette manière, en appliquant Pro juste avant Exp, on peut appliquer l'explicitation Pro + Exp à tout ensemble de formules avec types protégés.

Utiliser Pro avant Exp donne l'impression de résoudre les problèmes de cardinalité de domaine. Cependant en faisant cela on a ajouté des projections qui peuvent s'apparenter à des décorations. Or Exp modifie peu les formules, uniquement les applications de symboles polymorphes. Ici Pro va ajouter des projections autour de beaucoup de termes, on perd donc un peu de l'intérêt de Exp. Cependant les termes totalement monomorphes de type fini restent inchangés.

L'ensemble E est un ensemble de types, et non pas seulement un ensemble de sorte, car toutes les instances d'un seul type peuvent être infinies et résumer cela avec un seul type simplifie l'axiomatisation. Par exemple, supposons Δ comprenne une axiomatisation des listes polymorphes $\texttt{list}\,\alpha$. Pour toute sorte S, $\texttt{list}\,S$ possède un nombre infini d'éléments puisque l'on a supposé que toutes les sortes sont habitées. En revanche d'autres types possèdent toujours des domaines finis, comme par exemple le type `poly_bool`. Entre ces deux extrêmes, certains types peuvent être infinis sous certaines conditions. Par exemple soit le type $\texttt{set}\,\alpha$ des ensembles polymorphes avec extensionnalité. La sorte $\texttt{set}\,S$ est infinie si et seulement si S est infini. Pour simplifier la description de E par l'utilisateur, on peut lui demander pour chaque constructeur de type dans quelles catégories il se place.

Ces questions de cardinalité des domaines ont déjà été abordées. Meng et Paulson ont proposé de supprimer les axiomes impliquant la finitude d'un type [44, Sect. 2.8] ; le critère qu'ils proposaient n'était cependant pas correcte (leur technique l'était car ils vérifiaient les preuves). Récemment, une technique de détection [19] a été proposée pour détecter une sur-approximation des sortes finies dans un ensemble de formules multi-sortées. Bien que ces techniques peuvent sans doute être étendues à la logique polymorphe, il semble que demander à l'utilisateur (ou au développeur d'une théorie) d'indiquer quelles applications d'un constructeur de type (le constructeur de type est accompagné par une axiomatique) possède uniquement des domaines finis est envisageable [11]. La sémantique peut être, si l'utilisateur indique qu'un type est toujours infini et qu'il ne l'est en fait pas, que l'ensemble de formules est alors insatisfiable (les prémisses sont incohérentes). On peut simplement demander à l'utilisateur d'indiquer pour tout constructeur de type non-interprété s'il se comporte comme `list`, comme `set` ou comme `poly_bool`. Pour les constructeurs de types interprétés, comme les types algébriques, on peut l'inférer.

Une dernière remarque est que Pro + Exp permet également de traduire la logique multi-sortée vers la logique non-sortée si l'ensemble des types protégés est vide. Une telle transformation se rapproche alors de celle présentée par Claessen [19]. On a alors la possibilité d'utiliser des démonstrateurs TPTP comme Vampire [53] ou SPASS [57]. Les démonstrateurs TPTP fonctionnent différemment des démonstrateurs SMT, ce qui est intéressant puisque qu'ils peuvent ainsi réussir à prouver des classes de problèmes très différentes des démonstrateurs SMT.

6.7. Exemple avec des prédicats de séparation

On va illustrer l'intérêt de la distinction à l'aide de la formalisation de liste chaînée munie de prédicats de séparation comme générés dans la section 3.2.

La signature de notre théorie est la suivante :

$$\begin{aligned}
\mathtt{FT} &= \{\, \mathsf{I},\ \mathsf{M}(\cdot,\cdot),\ \mathsf{P} \,\} \\
\mathbb{F} &= \{\, + : \langle \mathsf{I}, \mathsf{I}, \mathsf{I} \rangle,\ \mathtt{0} : \langle \mathsf{I} \rangle,\ \mathtt{1} : \langle \mathsf{I} \rangle \,\} \cup \\
&\quad \{\, \mathtt{get} : \langle \mathsf{M}(\alpha,\beta), \alpha, \beta \rangle,\ \mathtt{set} : \langle \mathsf{M}(\alpha,\beta), \alpha, \beta, \mathsf{M}(\alpha,\beta) \rangle \,\} \cup \\
&\quad \{\, \mathtt{null} : \langle \mathsf{P} \rangle,\ \mathtt{ft}^1_{\mathtt{islist}} : \langle \mathsf{M}(\mathsf{P},\mathsf{P}), \mathsf{P}, ft \rangle,\ \} \\
\mathbb{P} &= \{\, \leq : \langle \mathsf{I}, \mathsf{I} \rangle,\ \mathtt{islist} : \langle \mathsf{M}(\mathsf{P},\mathsf{P}), \mathsf{P} \rangle, \mathtt{sep_islist_islist} : \langle \mathsf{M}(\mathsf{P},\mathsf{P}), \mathsf{P}, \mathsf{P} \,\}
\end{aligned}$$

Le constructeur de type I représente les entiers et P représente les pointeurs, ici les cellules des listes. Le type $\mathsf{M}(\alpha,\beta)$ représente les tables d'association polymorphes du type α vers le type β. Nous utilisons ces tables (avec α et β toutes deux instanciées par P) pour accéder à la cellule suivante. C'est pourquoi les symboles $\mathtt{islist}$, $\mathtt{ft}^1_{\mathtt{islist}}$ et $\mathtt{sep\_islist\_islist}$ prennent une table en argument (voir Section 3.1). Enfin le symbole de constante $\mathtt{null} : \langle \mathsf{P} \rangle$ indique la fin de la liste.

Les axiomes sont donnés dans la figure 6.1. Des annotations de types sont omises pour simplifier la lecture.

À l'aide de cette axiomatique on peut chercher la validité du but suivant qui est un morceau d'une condition de vérification de l'inversion d'une liste en place comme spécifié dans la section 4.3.

$$\begin{aligned}
\forall (m : \mathsf{M}(\mathsf{P},\mathsf{P}))\, \forall (q\ p\ v : \mathsf{P})\, \mathtt{islist}(m,p) \Rightarrow \mathtt{islist}(m,q) \Rightarrow \\
\mathtt{sep_islist_islist}(m,p,q) \Rightarrow \mathtt{islist}(\mathtt{set}(m,p,v),q). \quad (6.5)
\end{aligned}$$

On essaie donc de prouver la satisfiabilité des axiomes avec la négation du but :

$$\begin{aligned}
\exists (m : \mathsf{M}(\mathsf{P},\mathsf{P}))\, \exists (q\ p\ v : \mathsf{P})\, \mathtt{islist}(m,p) \wedge \mathtt{islist}(m,q) \Rightarrow \\
\mathtt{sep_islist_islist}(m,p,q) \wedge \neg \mathtt{islist}(\mathtt{set}(m,p,v),q). \quad (6.6)
\end{aligned}$$

Pour la transformation Dis on choisit de prendre $W = \{\mathtt{get}_{[\mathsf{P}/\alpha,\mathsf{P}/\beta]}, \mathtt{set}_{[\mathsf{P}/\alpha,\mathsf{P}/\beta]}\}$. Dis produit à partir des formules de la figure 6.1 les formules de la figure 6.2 (les indices des variables introduit par Dis sont omis). Le nouveau symbole $\mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}$ à la signature $\langle \mathsf{M}(\mathsf{P},\mathsf{P}), \mathsf{P}, \mathsf{P} \rangle$ et $\mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}$ à la signature $\langle \mathsf{M}(\mathsf{P},\mathsf{P}), \mathsf{P}, \mathsf{P}, \mathsf{M}(\mathsf{P},\mathsf{P}) \rangle$. La transformation Dis a dupliqué les deux axiomes polymorphes et distingué $\mathtt{get}$ et $\mathtt{set}$ dans les autres formules. On peut déjà remarquer que, à part les deux premiers axiomes, toutes les autres formules sont monomorphes et ne contiennent plus de symboles polymorphes.

Si on applique ensuite Tw avec $U = \{\mathsf{I}, \mathsf{M}(\mathsf{I},\mathsf{I})\}$, c'est-à-dire les types qui apparaissent dans les signatures des symboles de W, l'ensemble de formules n'est pas modifié à part I qui devient $\bar{\mathsf{I}}$ et $\mathsf{M}(\mathsf{I},\mathsf{I})$ qui devient $\overline{\mathsf{M}(\mathsf{I},\mathsf{I})}$. En effet aucun symbole polymorphe n'est appliqué à un terme de type protégé. Si on applique maintenant Dec ou Exp seuls les

$$\forall(m:\mathsf{M}(\alpha,\beta))\,\forall(c:\alpha)\,\forall(v:\beta)\;\mathtt{get}(\mathtt{set}(m:\mathsf{M}(\alpha,\beta),c:\alpha,v:\beta):\mathsf{M}(\alpha,\beta),c:\alpha):\beta\approx v:\beta$$

$$\forall(m:\mathsf{M}(\alpha,\beta))\,\forall(c:\alpha)\,\forall(c':\alpha)\,\forall(v:\beta)\;(c\approx c'\vee\mathtt{get}(\mathtt{set}(m,c,v),c')\approx\mathtt{get}(m,c'))$$

$$\forall(m:\mathsf{M}(\mathsf{P},\mathsf{P}))\,\forall(c:\mathsf{P}).\;(\mathtt{islist}(m,c)\Leftrightarrow c\approx\mathtt{null}\vee\mathtt{islist}(m,\mathtt{get}(m:\mathsf{M}(\mathsf{P},\mathsf{P}),c:\mathsf{P}):\mathsf{P}))$$

$$\forall q\forall m\forall p.\;q\in\mathtt{ft}^1_{\mathtt{islist}}(m,p))\Leftrightarrow\begin{pmatrix}p\not\approx\mathtt{null}\wedge\mathtt{islist}(m,\mathtt{get}(m,p))\\ \wedge\,(q\approx p\vee q\in\mathtt{ft}^1_{\mathtt{islist}}(m,\mathtt{get}(m,p)))\end{pmatrix}$$

$$\forall(m:\mathsf{M}(\mathsf{P},\mathsf{P}))\,\forall(c_1,c_2:\mathsf{P})\;\mathtt{sep\_islist\_islist}(m,c_1,c_2))\Leftrightarrow(\forall q:\mathsf{P})\;q\notin\mathtt{ft}^1_{\mathtt{islist}}(m,c_1)\vee q\notin\mathtt{ft}^1_{\mathtt{islist}}(m,c_2)$$

$$\forall(m:\mathsf{M}(\mathsf{P},\mathsf{P}))\,\forall(c,q,v:\mathsf{P})\;q\notin\mathtt{ft}^1_{\mathtt{islist}}(m,c)\Rightarrow\mathtt{ft}^1_{\mathtt{islist}}(m,c)\Rightarrow\mathtt{ft}^1_{\mathtt{islist}}(\mathtt{set}(m,q,v),c)$$

FIGURE 6.1.: Axiomatique de listes simplement chaînées avec prédicat de séparation.

deux premiers axiomes polymorphes sont modifiés. Au final l'axiomatique générée est traduite sans aucun artefact d'encodage. On peut en prime utiliser une théorie prédéfinie des tableaux.

$$\forall(m:\mathsf{M}(\alpha,\beta))\,\forall(c:\alpha)\,\forall(v:\beta)\;\mathtt{get}(\mathtt{set}(m:\mathsf{M}(\alpha,\beta),c:\alpha,v:\beta):\mathsf{M}(\alpha,\beta),c:\alpha):\beta\approx v:\beta$$

$$\forall(m:\mathsf{M}(\alpha,\beta))\,\forall(c:\alpha)\,\forall(c':\alpha)\,\forall(v:\beta)\;(c\approx c'\vee\mathtt{get}(\mathtt{set}(m,c,v),c')\approx\mathtt{get}(m,c'))$$

$$\forall(m:\mathsf{M}(\mathsf{P},\mathsf{P}))\,\forall(c:\mathsf{P})\,\forall(v:\mathsf{P})$$
$$\mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(\mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(m:\mathsf{M}(\mathsf{P},\mathsf{P}),c:\mathsf{P},v:\mathsf{P}):\mathsf{M}(\mathsf{P},\mathsf{P}),c:\mathsf{P}):\mathsf{P}\approx v:\mathsf{P}$$

$$\forall(m:\mathsf{M}(\mathsf{P},\mathsf{P}))\,\forall(c:\mathsf{P})\,\forall(c':\mathsf{P})\,\forall(v:\mathsf{P})$$
$$(c\approx c'\vee\mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(\mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(m,c,v),c')\approx\mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(m,c'))$$

$$\forall(m:\mathsf{M}(\mathsf{P},\mathsf{P}))\,\forall(c:\mathsf{P})\;(\mathtt{islist}(m,c)\Leftrightarrow$$
$$c\approx\mathtt{null}\vee\mathtt{islist}(m,\mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(m:\mathsf{M}(\mathsf{P},\mathsf{P}),c:\mathsf{P}):\mathsf{P}))$$

$$\forall(m:\mathsf{M}(\mathsf{P},\mathsf{P})).\forall(pq:\mathsf{P}).q\in\mathtt{ft}^1_{\mathtt{islist}}(m,p))\Leftrightarrow$$
$$\begin{pmatrix}p\not\approx\mathtt{null}\wedge\mathtt{islist}(m,get_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(m,p))\\ \wedge\,(q\approx p\vee q\in\mathtt{ft}^1_{\mathtt{islist}}(m,get_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(m,p)))\end{pmatrix}$$

$$\forall(m:\mathsf{M}(\mathsf{P},\mathsf{P}))\,\forall(c_1,c_2:\mathsf{P})\;\mathtt{sep\_islist\_islist}(m,c_1,c_2))\Leftrightarrow$$
$$(\forall q:\mathsf{P})\;q\notin\mathtt{ft}^1_{\mathtt{islist}}(m,c_1)\vee q\notin\mathtt{ft}^1_{\mathtt{islist}}(m,c_2)$$

$$\forall(m:\mathsf{M}(\mathsf{P},\mathsf{P}))\,\forall(c,q,v:\mathsf{P})\;q\notin\mathtt{ft}^1_{\mathtt{islist}}(m,c)\Rightarrow$$
$$\mathtt{ft}^1_{\mathtt{islist}}(m,c)\Rightarrow\mathtt{ft}^1_{\mathtt{islist}}(\mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(m,q,v),c)$$

$$\exists(m:\mathsf{M}(\mathsf{P},\mathsf{P}))\,\exists(q\;p\;v:\mathsf{P})\;\mathtt{islist}(m,p)\wedge\mathtt{islist}(m,q)$$
$$\mathtt{sep\_islist\_islist}(m,p,q)\wedge\neg\mathtt{islist}(\mathtt{set}(m,p,v),q).$$

FIGURE 6.2.: Le problème obtenu après application de la transformation DIS avec le paramètre $W = \{\mathtt{get}_{[\mathsf{P}/\alpha,\mathsf{P}/\beta]},\,\mathtt{set}_{[\mathsf{P}/\alpha,\mathsf{P}/\beta]}\}$.

7. Implémentation et résultats

Au cours du temps la plate-forme `Why` s'est dotée de moyens pour traduire des problèmes polymorphes dans la logique d'entrée des différents démonstrateurs. Couchot et Lescuyer ont implémenté au sein de `Why` leur encodage par décoration et pont vers la logique de Simplify [25](uniquement la sorte `I` des entiers) et la logique multi-sortée.

J'ai implémenté la monomorphisation partielle pour tester son efficacité. Des buts avec les axiomatiques produites par la méthode de la première partie étaient prouvés plus facilement grâce à cette méthode. Cependant, `Why` commençait à ne plus pouvoir être étendu dans son architecture d'alors. Les points d'entrée pour définir de nouveaux encodages étaient mal définis, la configuration de l'encodage à utiliser pour chaque démonstrateur était éparpillée et la génération et l'appel des démonstrateurs étaient séparés en deux outils différents. D'autres parties du logiciel `Why` comme le moteur de l'interface utilisateur demandaient également une refonte. L'outil `Why` a donc été totalement repris et j'ai eu le plaisir de participer à cette entreprise. `Why3` [10] est basé sur plusieurs points centraux :

- mise en place d'une API claire à la fois pour une utilisation interne mais également pour que d'autres développements puissent l'utiliser comme bibliothèque externe ;
- utilisation de typage défensif pour s'assurer que l'on ne crée que des termes bien formés avec l'API ;
- définition d'un nouveau langage logique pour permettre de définir aisément des théories (tableaux, ensembles, arithmétique bornée,...). Le langage de programme est un sur-ensemble de celui-là ;
- définition de la notion de *tâches*, qui sont des ensembles d'axiomes suivis d'un but à prouver, et de la notion de *transformations* qui transforment une tâche en une autre ;
- définition de la notion de *driver* qui décrit dans un fichier de configuration pour un démonstrateur donné comment générer le but à envoyer à ce démonstrateur ;
- des fichiers de configurations qui énumèrent les démonstrateurs connus de `Why` et qui indiquent pour chacun d'eux leurs exécutables et leurs drivers.

Chaque point provient d'une difficulté rencontrée dans l'ancien `Why`. L'utilisation d'une API claire et le typage défensif permettent de s'assurer que le reste de `Why` utilise de manière cohérente les formules mais permet également de connaître dans la plupart des cas l'endroit précis où un terme mal formé est construit. Cela facilite grandement l'écriture de transformations complexes. Les transformations sont nées, elles, du constat que les encodages devaient souvent éliminer les mêmes constructions du langage (par exemple les définitions). On a résolu ce problème en permettant de définir des transformations qui peuvent se succéder, de la même manière qu'un compilateur compile un langage source vers un langage cible. Cet enchaînement de transformations est défini dans le dri-

ver spécifique à un démonstrateur automatique. Le driver permet également d'associer à certains symboles de fonctions et de prédicats le symbole prédéfini correspondant du démonstrateur automatique.

Une fois `Why3` développé, nous nous sommes appuyés sur ces nouveaux outils pour implémenter les encodages des chapitres précédents. Chaque chapitre précédent est devenu une transformation particulière. Nous les avons toutes rassemblées au sein de deux transformations "encoding_smt" ou "encoding_tptp" qui apparaissent dans les drivers. En plus des constructions du langage logique dont la sémantique est bien définie, nous avons ajouté des déclarations spéciales, appelé *metas*, qui permettent d'ajouter des informations particulières à des symboles de types, des symboles de fonctions, des symboles de prédicats, des types, ou à l'ensemble de la tâche. Les transformations peuvent utiliser ces informations selon leur sémantique propre ; ainsi ils peuvent servir "d'arguments" données aux transformations. On utilise les metas dans "encoding_smt" et "encoding_tptp" de deux manières. On les utilise si l'on veut choisir un autre chemin d'encodage que celui par défaut. On les utilise également pour choisir les sortes (respectivement les spécialisations monomorphes de symboles), que l'on veut ajouter à U (respectivement W). On peut également choisir la manière de généré automatiquement ces deux ensembles.

Nous avons testé ces encodages sur 4123 conditions de vérification générées par la plate-forme `Why` depuis 166 programmes, qui proviennent de Caduceus [28], Jessie, ou directement de `Why`. Les tâches traduites ont été envoyées vers Z3, CVC3 et Yices avec une limite de temps de 60 secondes. En tout, 3993 obligations de preuves ont été prouvées par au moins l'un de ces démonstrateurs. Les résultats sont accessibles à `http://why3.lri.fr/download/polyfol_encoding.tar.gz`. Dans ces tâches, certains axiomes contiennent des triggers, qui sont des ensembles d'ensembles de termes ou littéraux. On les encode comme tout autre terme et littéral. Ces tâches ne contiennent pas de types finis (pour les booléens, on indiquent simplement que `False` $\not\approx$ `True`) car ces exemples provenaient de `Why2` où les utilisateurs faisaient attention de ne pas en introduire pour permettre des encodage plus simples vers certains prouveurs. Donc Pro n'a pas été utilisé avant Exp.

Nous avons testé les encodages Tw+Dec, Tw+Exp, et Tw+Grd, avec et sans Dis. Sans Dis l'ensemble U de sortes à protéger dans Tw est l'ensemble contenant seulement les entiers et les réels, qui sont prédéfinis dans les trois démonstrateurs. En présence de Dis, on met dans l'ensemble W toutes les spécialisations monomorphes qui apparaissent dans la formule du but. On met de plus les spécialisations des opérations d'accès et de mise à jour sur chaque type monomorphe de tableau présent dans le but. Enfin on protège toutes les sortes présentes dans le but en plus des entiers et des réels. Ces configurations de Dis et Tw donnent de meilleurs résultats comparés aux autres configurations que nous avons essayées, comme par exemple collecter les spécialisations et les sortes protégées de toute la tâche.

Nos résultats sont donnés dans la table 7.1. À la droite du nom du démonstrateur, nous mettons le nombre de buts prouvés par au moins une méthode d'encodage. Dans toutes les cases, on indique le nombre de buts prouvés par un encodage mais pas par l'autre. Par exemple, avec CVC3, l'encodage par Dis+Tw+Dec permet de prouver 84 buts qui ne sont pas prouvés par Tw+Dec. D'un autre coté, avec Tw+Dec, CVC3

Z3 (3809)	Tw+Grd		Tw+Exp		Tw+Dec		Dis+Tw+Grd		Dis+Tw+Exp	
Dis+Tw+Dec	+203	−36	+20	−49	+66	−37	+18	−5	+26	−30
Dis+Tw+Exp	+191	−20	+13	−38	+63	−30	+35	−18		
Dis+Tw+Grd	+195	−41	+11	−53	+59	−43				
Tw+Dec	+157	−19	+15	−73						
Tw+Exp	+211	−15								

CVC3 (3756)	Tw+Grd		Tw+Exp		Tw+Dec		Dis+Tw+Grd		Dis+Tw+Exp	
Dis+Tw+Dec	+269	−20	+0	−26	+84	−19	+66	−4	+0	−6
Dis+Tw+Exp	+272	−17	+0	−20	+88	−17	+69	−1		
Dis+Tw+Grd	+204	−17	+1	−89	+46	−43				
Tw+Dec	+188	−4	+0	−91						
Tw+Exp	+275	−0								

Yices (3717)	Tw+Grd		Tw+Exp		Tw+Dec		Dis+Tw+Grd		Dis+Tw+Exp	
Dis+Tw+Dec	+882	−6	+13	−276	+379	−79	+204	−2	+3	−272
Dis+Tw+Exp	+1149	−4	+39	−33	+574	−5	+472	−1		
Dis+Tw+Grd	+684	−10	+6	−471	+241	−143				
Tw+Dec	+577	−1	+5	−568						
Tw+Exp	+1140	−1								

Table 7.1.: Comparaison de méthodes d'encodage du polymorphisme

prouve 19 buts qu'il ne prouve pas avec Dis+Tw+Dec.

En moyenne, la discrimination de symboles augmente le nombre de but prouvées par un facteur de 1,8 (de 1 à 10 selon les exemples). Ajouter l'étape de Dis permet de prouver plus de buts dans tous les cas sauf avec Z3 et CVC3 avec Exp. En particulier la transformation Grd est remarquablement aidée par Dis. Outre le fait de permettre l'utilisation du support prédéfini des tableaux, l'efficacité de Dis est aussi expliquée par le fait que nous protégeons les sortes qui apparaissent dans les spécialisations monomorphes sélectionnées. Ainsi les prémisses sont, en plus d'être instanciées par des sortes pertinentes, libérées des décorations imposées par la troisième étape de fusion des types. Cela apparaît moins dans le cas de Exp, parce que cette transformation, à la différence de Dec et Grd, ajoute très peu de désordre aux formules encodées.

On peut remarquer que la protection de type est absolument nécessaire : si aucun type n'est protégé, on empêche les démonstrateurs d'utiliser leurs théories prédéfinies, et le nombre total de buts tombe à 1861.

La comparaison entre Exp, Dec, et Grd montre que Exp est généralement plus efficace que Dec qui à son tour est plus efficace que Grd. Ces résultats sont assez différents des résultats obtenus dans [41], où Exp et Grd ont à peu près les mêmes performances. Nous n'avons pas encore identifié si ces différences proviennent de différences dans nos cas d'étude ou dans notre implantation.

8. Conclusion et perspectives

8.1. Conclusion

Nous avons vu comment à partir de la définition d'un prédicat inductif on peut définir de nombreux concepts pour l'utilisateur comme des prédicats de séparation et d'encadrement. Ces outils permettent de prouver des programmes demandant une séparation plus fine que permet un modèle mémoire par champs, tout en gardant une bonne automatisation.

Cependant différents outils ont dû être proposés à l'utilisateur et il semble qu'il faille en introduire encore d'autres pour d'autres problèmes. En revanche comparé à *Implicit Dynamic Frame* ce développement permet une plus grande flexibilité à l'utilisateur : les spécifications peuvent utiliser toute la puissance de la logique du premier ordre alors qu'*Implicit Dynamic Frame* est plus limité. En revanche ces limitations les autorisent à calculer l'empreinte d'un programme directement à partir de sa précondition. Nous pensons que cela n'est pas toujours pratique car il est souvent utile de mentionner des propriétés de structures de données que l'on ne va pas modifier.

L'automatisation étant un point important il a fallu améliorer les techniques d'encodage vers les démonstrateurs automatiques. Pour cela nous avons développé un schéma général d'encodage avec une nouvelle transformation appelée distinction, des protections de types généralisés et des encodages du polymorphisme étendus.

8.2. Génération de définitions alternatives

Nous avons vu comment une définition inductive permettait de générer son empreinte et ensuite de l'utiliser pour prouver un programme. Nous allons voir que l'on peut générer d'autres définitions utiles à partir de la définition d'un inductif. Le problème est présenté puis ce qui pourrait devenir un développement futur est rapidement esquissé sur deux exemples.

Un programme peut utiliser une structure de donnée de différentes manières. Spécifier un tel programme peut donc demander de décrire la structure de donnée de plusieurs manières. Prenons l'exemple d'une fonction qui parcourt une liste d'entiers positifs pour mettre toute la liste à zéro.

```
/*@
inductive pos_list{L}(struct node *p){
  case nil{L} : ∀ struct node *p; p==\null ⇒ ilist(p);
  case cons{L} : ∀ struct node *p;
```

```
    p!=\null ⇒ \valid(p) ⇒ p->val >= 0 ⇒
    pos_list(p->next) ⇒ pos_list(p);
  }
  @*/

  /*@
  requires pos_list(p);
  ensures pos_list(p);
  @*/
  void zeroify (struct node * p){
    struct node *q = p;
    while(q != null){
      q->value = 0;
      q = q->next;
    }
}
```

Pour prouver que le programme suit sa spécification, il faut déterminer un invariant de boucle. On doit exprimer l'idée que ce qui a déjà été parcouru fait partie d'une liste bien formée, et ce qui suit l'est également. Le suffixe de la liste s'exprime simplement à l'aide du prédicat `pos_list(q)`. En revanche le préfixe, qui est ici la liste de `p` à `q`, ne peut s'exprimer ainsi. Cependant, à partir de la définition de `pos_list`, on peut définir une partie de liste de `p` à `q`, c'est-à-dire quelque chose qui, lorsqu'il est complété avec une liste partant de `q`, devient une liste partant de `p`. Cette idée a été étudiée en *shape analysis* [15, 16]. On peut définir un prédicat inductif `pos_list_part` qui exprime exactement cette idée :

```
  /*@ inductive pos_list_part(struct node *p, struct node *q)
```

Le premier cas de l'inductif représente le cas où `p` et `q` sont égaux, qui est bien une liste de `p` à `q`.

```
    case equal{L} : \forall struct node *p, *q;
    p == q ==> pos_list_part(p,q);
```

Ensuite on reprend chaque cas de l'inductif initial et, pour chaque appel récursif en position négative on crée un cas où cet appel récursif est remplacé par un appel à la nouvelle définition. Bien sûr l'appel en position positive est lui aussi remplacé par un appel à la nouvelle définition. Le cas `nil` ne contient pas d'appel récursif ; il est donc supprimé. Le cas `cons` devient :

```
    case cons1{L} : \forall struct node *p, *q;
    p!=\null ==> \valid(p) ==> p->val >= 0 ==>
    pos_list_part(p->next,q) ==> pos_list_part(p,q);
  @*/
```

Le nouvel inductif a ainsi été défini automatiquement. Cette vision ne suffit pas car on ajoute des éléments au préfixe par la fin. On a donc besoin de voir la liste à l'envers, ce

qui correspond à l'inductif suivant, qui peut être également généré automatiquement :

```
inductive pos_list_inv(struct node *p, struct node *q)
   case equal{L} : \forall struct node *p, *q;
    p == q ==> pos_list_inv(p,q);
   case cons1{L} : \forall struct node *p, *q;
     \forall struct node *r. r->next = q ==>
    r!=\null ==> \valid(r) ==> r->val >= 0 ==>
    pos_list_inv(p,r) ==> pos_list_inv(p,q);
```

Ces transformations d'inductifs sont assez similaires aux transformations de types algébriques que sont les zipper [31]. On peut de plus introduire les axiomes de conversion suivants (toujours automatiquement) :

```
axiom a1 : \forall struct node *p, *q;
   pos_list_part (p,q) ==> pos_list(q) ==> pos_list(p);
axiom a2 : \forall struct node *p, *q;
   pos_list_part (p,q) <==> pos_list_inv(p,q);
```

Enfin, comme tout autre inductif on peut définir son empreinte (qui est en relation avec l'empreinte de l'inductif original) et sa séparation avec lui-même ou d'autres inductifs. Par exemple :

```
/*@ #pragma : sep_pos_list_inv_pos_list(pos_list_inv,pos_list) @*/
```

Et ainsi on doit pouvoir prouver le programme précédent avec l'invariant de boucle :

```
pos_list_inv(p,q) && pos_list(q)
&& sep_pos_list_inv_pos_list(p,q,q);
```

La liste est un cas assez simple, il n'y a au plus qu'une application du prédicat inductif en position négative dans sa définition. Si l'on prend le cas des arbres, c'est-à-dire le type :

```
struct tree {
  int key;
  struct tree *left, *right;
}
```

la définition d'un arbre binaire `istree` est :

```
/*@
#pragma : sep_istree_istree(istree,istree)

inductive istree(struct tree * p){
  case leave{L} : ∀ struct tree * p; p == null ⇒ istree(p);
  case node{L} : ∀ struct tree * p;
  p!=null ⇒ \valid(p) ⇒ sep_istree_istree(p->left,p->right) ⇒
  istree(p->left) ⇒ istree(p->right) ⇒ istree(p);
}
@*/
```

La définition alternative, décrivant une partie d'arbre qui devient un arbre complet partant de `p` lorsqu'on le complète avec un autre arbre partant de `q`, est alors :

```
/*@
#pragma : sep_istree_istree_part(istree,istree_part)

inductive istree_part(struct tree * p, struct tree * q){
  case equal{L} : ∀ struct tree *p *q; p == q ⇒ istree_part(p,q);
  case node1{L} : ∀ struct tree *p *q;
    p!=null ⇒ \valid(p) ⇒ sep_istree_istree_part(p->right,p->left,q) ⇒
    istree_part(p->left,q) ⇒ istree(p->right) ⇒ istree_part(p,q);
  case node2{L} : ∀ struct tree *p *q;
    p!=null ⇒ \valid(p) ⇒ sep_istree_istree_part(p->left,p->right,q) ⇒
    istree(p->left) ⇒ istree_part(p->right,q) ⇒ istree_part(p,q);
}
@*/
```

Le prédicat `sep_istree_istree_part` est le prédicat de séparation entre `istree` et `istree_part`.

Cette génération automatique de définition alternative n'a malheureusement pas encore été implémentée. Son étude précise et son implémentation restent des travaux futurs. Ils sont liés aux travaux sur les prédicats de séparation car ils sont liés aux structures de données mutables, mais surtout car l'empreinte associée à un prédicat inductif est liée à l'empreinte du prédicat alternatif. Les générer en même temps permet de décrire automatiquement les relations entre elles.

Dans le cas où l'utilisateur doit avoir plusieurs visions du même prédicat, *Implicit Dynamic Frame* demandera à l'utilisateur de définir plusieurs versions du prédicat et de montrer ensuite leurs correspondances à l'aide d'une preuve sous forme de programme pur [55, section 3.4].

8.3. Extensions de la logique

Nous avons choisi la logique polymorphe par quantification prénexe, car cela correspond à ce que les modèles mémoire utilisent. Quels encodages pourraient être appliqués si on acceptait des quantifications universelles de types et des quantifications existentielles de type à toutes profondeurs ? La sémantique de cette logique changerait car on ne pourrait plus la baser sur une instanciation infinie. L'interprétation devrait en effet modifier la valuation de type π en passant sur un quantificateur comme cela est fait pour la valuation. Tw, Dec, Exp et même Grd doivent pouvoir s'adapter, il y aura simplement des quantifications sur des variables de type T à toutes les profondeurs. La distinction Dis est plus délicate : elle ajoutera au niveau de chaque quantification universelles la conjonction de toutes les instanciations de la sous-formule. Les quantifications existentielles semblent plus difficiles à intégrer avec Dis, car si on skolemise ces quantifications

on peut faire apparaître des types dépendants. En effet dans la formule $\forall x : int.\exists T. \ldots$ pour chaque x différent il existe un type T particulier, c'est-à-dire un "$T(x)$".

Certains démonstrateurs peuvent donner des preuves lorsqu'ils prouvent un résultat. Il y a deux manières d'obtenir, à partir de la preuve pour le problème traduit, une preuve pour le problème initial polymorphe. La première consiste à composer la preuve donnée avec une preuve de correction de la transformation. La preuve de correction est parfois compliquée et cela allonge la longueur de la preuve. La seconde manière consiste à inverser la transformation, en réintroduisant le polymorphisme dans la preuve. Tous les artefacts de l'encodage sont enlevés, diminuant d'autant la longueur de la preuve. Il serait intéressant que `Why3` puisse lire les preuves des différents prouveurs et les transformer dans un langage de preuve commun.

Un autre développement intéressant serait que les démonstrateurs automatiques acceptent nativement le polymorphisme, ce qu'aucun ne fait à l'exception d'`Alt-Ergo` [9]. Cette article défend qu'ajouter dans un démonstrateur automatique du premier ordre basé sur une technique particulière d'instanciation, celle de la majorité des démonstrateurs SMT du premier ordre, ne nécessite pas de modification majeur. Une tentative de normalisation d'un langage polymorphe au sein de la bibliothèque TPTP est en cours.

Liste des tableaux

Table des figures

Bibliographie

[1] Romain Bardou. Invariants de classe et systèmes d'ownership. Master's thesis, 2007.

[2] Clark Barrett and Cesare Tinelli. CVC3. In *CAV'07*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[3] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL : ANSI/ISO C Specification Language*, 2008. `http://frama-c.cea.fr/acsl.html`.

[4] Josh Berdine, Cristiano Calcagno, and Peter O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, *Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68. Springer Berlin / Heidelberg, 2005.

[5] Josh Berdine, Cristiano Calcagno, and Peter W. O'hearn. Smallfoot : Modular automatic assertion checking with separation logic. In *In International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[6] Didier Bert, editor. *B'98 : Recent Advances in the Development and Use of the B Method, Second International B Conference*, volume 1393, Montpellier, France, 1998.

[7] Joshua Bloch. Nearly all binary searches and mergesorts are broken, 2006. `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`.

[8] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover, 2008. `http://alt-ergo.lri.fr/`.

[9] François Bobot, Sylvain Conchon, Évelyne Contejean, and Stéphane Lescuyer. Implementing polymorphism in SMT solvers. In *SMT'08*, volume 367 of *ACM ICPS*, pages 1–5, 2008.

[10] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. *The Why3 platform*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.64 edition, February 2011. `https://gforge.inria.fr/docman/view.php/2990/7411/manual.pdf`.

[11] François Bobot and Andrei Paskevich. Expressing Polymorphic Types in a Many-Sorted Language. In *Frontiers of Combining Systems ('FROCOS), 8th International Symposium, Proceedings*, volume 6989, Saarbrücken, Germany, October 2011.

[12] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[13] Rod Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7 :23–50, 1972.

[14] Sascha Böhme and Michal Moskal. Heaps and data structures : A challenge for automated provers. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Proceedings of the 23rd International Conference on Automated Deduction*, volume 6803, pages 177–191, Wrocław, Poland, 2011.

[15] Bor-Yuh Chang, Xavier Rival, and George Necula. Shape analysis with structural invariant checkers. In Hanne Nielson and Gilberto Filé, editors, *Static Analysis*, volume 4634 of *Lecture Notes in Computer Science*, pages 384–401. Springer Berlin, 2007.

[16] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Proceedings of the 35th annual symposium on Principles of programming languages*, POPL '08, pages 247–260, 2008.

[17] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *SIGPLAN Not.*, 25 :296–310, June 1990.

[18] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58 :345–363, 1936.

[19] Koen Claessen, Ann Lillieström, and Nicholas Smallbone. Sort it out with monotonicity - translating between many-sorted and unsorted first-order logic. In *CADE*, pages 207–221, 2011.

[20] Jean-François Couchot and Stéphane Lescuyer. Handling polymorphism in automated deduction. In *CADE-21*, volume 4603 of *LNAI*, pages 263–278. Springer, 2007.

[21] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL '79 : Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282, New York, NY, USA, 1979. ACM.

[22] H.B. Curry, J.R. Hindley, and J.P. Seldin. *Combinatory Logic II*, volume 65 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1972.

[23] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82 : Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.

[24] Leonardo de Moura and Nikolaj Bjørner. Z3 : An efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[25] David Detlefs, Greg Nelson, and James B. Saxe. Simplify : a theorem prover for program checking. *J. ACM*, 52(3) :365–473, 2005.

[26] Edsger W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. 1976.

[27] Dino Distefano, Peter O Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer Berlin / Heidelberg, 2006.

[28] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In *ICFEM'04*, volume 3308 of *LNCS*, pages 15–29. Springer, 2004.

[29] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580 and 583, October 1969.

[30] Thierry Hubert and Claude Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV'07)*, pages 81–93, Braga, Portugal, March 2007. `http://www.lri.fr/~marche/hubert07hav.pdf`.

[31] Gérard Huet. The zipper. *Journal of Functional Programming*, 7, 1997.

[32] Joe Hurd. An LCF-style interface between HOL and first-order logic. In *CADE-18*, volume 2392 of *LNAI*, pages 134–138. Springer, 2002.

[33] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In *Design and Application of Strategies/Tactics in Higher Order Logics*, pages 56–68. NASA Technical Report NASA/CP-2003-212448, 2003.

[34] Samin Ishtiaq and Peter W. O'hearn. Bi as an assertion language for mutable data structures. In *In POPL*, pages 14–26, 2001.

[35] Bart Jacobs and Frank Piessens. The verifast program verifier. CW Reports CW520, Department of Computer Science, K.U.Leuven, August 2008.

[36] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In *Programming Languages and Systems (APLAS 2010)*, pages 304–311. Springer-Verlag, November 2010.

[37] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, pages 244–256. ACM, 1979.

[38] D. Kaplan. Some Completeness Results in the Mathematical Theory of Computation. *Journal of the ACM*, 15(1) :124–34, Jan 1968.

[39] Ioannis Kassios. Dynamic frames : Support for framing, dependencies and sharing without restrictions. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006 : Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer Berlin / Heidelberg, 2006.

[40] K. Leino. Dafny : An automatic program verifier for functional correctness. In Edmund Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer Berlin / Heidelberg, 2010.

[41] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language : Design and logical encoding. In *TACAS'10*, volume 6015 of *LNCS*, pages 312–327. Springer, 2010.

[42] Maria Manzano. *Extensions of First-Order Logic*, volume 19 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1996.

[43] Claude Marché. Jessie : an intermediate language for Java and C verification. In *Programming Languages meets Program Verification (PLPV)*, pages 1–2, Freiburg, Germany, 2007. ACM.

[44] Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1) :35–60, 2008.

[45] R. Milner. A theory of type polymorphismn programming. *Journal of Computer and System Sciences*, 17, 1978.

[46] Yannick Moy. Union and cast in deductive verification. In *Proceedings of the C/C++ Verification Workshop*, volume Technical Report ICIS-R07015, pages 1–16. Radboud University Nijmegen, July 2007.

[47] Yannick Moy and Claude Marché. Inferring local (non-)aliasing and strings for memory safety. In *Heap Analysis and Verification (HAV'07)*, pages 35–51, Braga, Portugal, March 2007.

[48] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[49] P.W. O'Hearn and D.J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2) :215–244, June 1999.

[50] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du premier Congrès des Mathématiciens des Pays slaves*, Warszawa, 1929.

[51] J. C. Reynolds. Separation logic : a logic for shared mutable data structures. In *17h Annual IEEE Symposium on Logic in Computer Science*, 2002.

[52] John C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science*, pages 303–321. Palgrave, 2000.

[53] Alexandre Riazanov and Andrei Voronkov. Vampire 1.1. In *IJCAR'01*, volume 2083 of *LNCS*, pages 376–380. Springer, 2001.

[54] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames : Combining dynamic frames and separation logic. In Sophia Drossopoulou, editor, *ECOOP 2009 — Object-Oriented Programming*, Lecture Notes in Computer Science, pages 148–172. Springer Berlin / Heidelberg, 2009.

[55] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames : Combining dynamic frames and separation logic (soundness proof). CW Reports CW542, Department of Computer Science, K.U.Leuven, June 2009.

[56] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 43 :544–546, 1937.

[57] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In *CADE-22*, volume 5663 of *LNCS*, pages 140–145. Springer, 2009.