Preuves par raffinement de programmes avec pointeurs

Asma Tafat

ORSAY
N^o d'ordre : 2013PA112141

UNIVERSITÉ DE PARIS-SUD

CENTRE D'ORSAY

THÈSE

présentée
pour obtenir le

Grade de Docteur en Sciences de l'Université Paris-Sud

PAR

Asma TAFAT

—⋊—

SUJET :

Preuves par raffinement de programmes avec pointeurs.

soutenue le 06 Septembre 2013 devant la commission d'examen

Joffroy Beauquier	Président
Marie-Laure Potet	Rapporteur
Catherine Dubois	Rapporteur
Claude Marché	Directeur de thèse
Loïc Correnson	Examinateur
Alain Giorgetti	Examinateur

Remerciements

Je tiens tout d'abord à remercier Claude Marché qui m'a donné l'occasion de faire cette thèse. J'ai eu le plaisir de travailler avec lui pendant mon stage de Master 2, puis durant mes années de thèse et je peux dire en toute honnêteté que ce fut un réel plaisir. Son dynamisme et sa disponibilité sont un exemple.

Je remercie tout particulièrement les rapporteurs (rapporteuses ! ! !), Marie-laure Pôtet et Catherine Dubois qui on pris la peine de lire et de rapporter ma thèse, ainsi que pour leurs remarques. Je remercie également Joffory Beauquier, Alain Giorgetti et Loïc Correnson d'avoir accepté de faire partie de mon jury.

Un grand merci aux membres, actuels et passés, de l'équipe Toccata pour l'ensemble de leur œuvre. Les discussions au coin café représentent une vraie richesse, aussi bien du point de vue scientifique que culinaire, sportif, etc. Merci à Xavier de m'avoir encourager pendant la rédaction et pour sa relecture extrêmement attentive de ce manuscrit surtout sur la forme grammaticale et typographique. Je remercie sincèrement tous ceux qui ont contribué de prés ou de loin à l'accomplissement de ce modeste travail.

Pour finir, mes plus profonds remerciements vont à ma famille, sans qui je n'en serait certainement pas là. À mes frères et sœurs, surtout mes sœurs, qui m'ont supportée ces dernières années, je sais que ce n'était pas facile. À mes trois petits neveux, leur bonne humeur et leur innocence furent une vraie source d'énergie. Enfin, merci à mes parents, tous les remerciements de la terre ne suffiraient pas à leur exprimer ma gratitude pour leurs efforts et leur soutien.

Table des matières

Table des figures

Introduction

Les systèmes informatiques sont confrontés à des enjeux de plus en plus importants. Rendre ces systèmes fiables est l'un des principaux défis pour l'informatique. La fiabilité est importante d'un point de vue économique : coût de développement de logiciels fiables, coût de vérification/validation, et cruciale dans les systèmes de sécurité critiques : les transports (terrestres ou aériens), le domaine médical (télé-chirurgie, imagerie, irradiation, etc.) ou les systèmes de contrôle. Par le passé, des dysfonctionnements dans des systèmes critiques ont eu lieu avec des conséquences plus ou moins graves. En 1962 la sonde *Mariner 1* fut détruite 294,5 secondes après son lancement, à cause d'une mauvaise transcription d'un symbole mathématique dans la spécification du programme. Entre 1985 et 1987, *Therac-25*, une machine de radiothérapie, fut impliquée dans la mort d'au moins cinq personnes pour cause d'irradiation. Cette machine utilisait le code provenant d'autres modèles (*Therac-6* et *Therac-20*) qui, contrairement à la *Therac-25*, possédaient des protections physiques en cas de défaillance logiciels. En 1996, la fusée *Ariane V* a explosé une quarantaine de secondes après le décollage à la suite d'une panne dans le système de navigation, un échec dû à la réutilisation d'un des composants logiciels de la fusée d'*Ariane IV*. Ce sont là des exemples connus, mais cette liste est loin d'être exhaustive, on trouvera dans [7] une liste de plus de cent exemples d'erreurs dans des logiciels critiques.

Intuitivement, on pourrait se dire que tester un programme en couvrant l'ensemble des comportements ou des erreurs possibles suffirait à garantir qu'il fonctionne correctement. Dans le cas des logiciels critiques, des méthodes plus systématiques et rigoureuses sont nécessaires. De nombreuses approches ont été proposées pour assurer une meilleure maîtrise de tels systèmes, insistant par exemple sur la méthodologie de développement, la documentation, les critères de couverture des tests, etc. Les *méthodes formelles* se caractérisent par leur capacité à donner des garanties mathématiques quant à l'absence de certains défauts. Elles sont fondées sur le principe de l'utilisation de logiciels pour la vérification de programmes. On dit qu'un programme est correct s'il effectue sans se tromper la tâche qui lui est confiée et ce pour toutes les valeurs possibles. Pour garantir cela, il est nécessaire de décrire précisément et sans ambiguïté le comportement d'une telle tâche. Une telle spécification peut être écrite dans le langage naturel, des langages dits semi-formels tels que UML [9], ou bien des langages formels tels que JML [32], ACSL [21], etc. Contrairement aux commentaires purement explicatifs qui sont informels et ne sont compréhensibles que par un humain, et par conséquent non exploitables, les langages de spécification sont des langages formels. De même que les programmes écrits dans les langages de programmation peuvent être manipulés par des compilateurs, les spécifications écrites dans un langage de spécification peuvent être manipulés par les générateurs d'obligations de preuve tel que ESC/Java [38] ou Spec# [19].

Les méthodes formelles trouvent des applications naturelles dans la conception de systèmes informatiques critiques. À ce titre, leur utilisation est encouragée ou exigée par certains standards relatifs à la certification de sécurité (ISO/IEC 15408 [2]), c'est-à-dire à la capacité du système à résister aux attaques malveillantes ; ou de sûreté (IEC 61508 : [4]), c'est-à-dire à la fiabilité de fonctionnement du système. En fonction de la façon dont ces méthodes sont utilisées au cours du processus de développement d'un logiciel, elles peuvent être classées en deux sous-catégories :

- **Méthodes formelles dynamiques** : ces méthodes exécutent le programme sur des données d'entrée en générant plusieurs chemins qui couvrent toutes les possibilités des valeurs de l'entrée. Les tests, comme le montrent Woodcock *et al.* en 2009 [98] et le *Runtime assertion checker* de JML[33], sont des exemples de méthodes formelles dynamiques.
- **Méthodes formelles statiques** : contrairement au premier cas, ces méthodes permettent de vérifier le programme sans l'exécuter. Elles sont utilisées lors des phases de spécification, de conception ou d'implantation. On peut citer par exemple, le *Model Checking* qui est, notamment, une technique de vérification de propriétés temporelles sur des systèmes dits réactifs, l'*interprétation abstraite* qui est une théorie permettant l'approximation des sémantiques des systèmes informatiques, ou bien la *vérification déductive*.

Nous nous intéressons dans cette thèse à la *vérification déductive*, dans laquelle la correction d'un code par déduction se ramène à la preuve de formules mathématiques appelées *obligations de preuve*. La méthode B est un exemple de ce type de méthodes. Elle est utilisée aussi bien dans l'industrie que dans le monde académique. Ses applications industrielles sont parmi les précoces succès de la vérification déductive. Elle a notamment été utilisée pour des projets en rapport avec la sûreté ou la sécurité [1] parmi lesquels *KVB* (Alstom) [3], le système de contrôle de vitesse par balises de la *SNCF* en 1993, Meteor (Siemens Transportation Systems) [24] qui équipe la ligne 14 du métro de Parisen 1998 ou *Roissy VAL* (ClearSy) [6], la navette automatique de l'aéroport Roissy Charles de Gaulle depuis 2006. Parmi les autres succès de la vérification déductive, on peut citer le projet *L4.verified* qui propose un micro-noyau d'un système d'exploitation avec des propriétés fortes de sécurité formellement vérifiées [65].

Une notion importante dans le domaine de l'ingénierie, en général, et dans la vérification, en particulier, est celle de la *modularité*. Elle permet de développer (programmation) et de prouver (vérification) un composant indépendemment des autres. Concrètement, cette notion permet de s'assurer que la modification d'une partie du programme n'invalide pas toute la preuve de correction qui lui est associée. Pour ce faire, la preuve est découpée en plusieurs sous-parties indépendantes les unes des autres et qui correspondent chacune à un bout du programme original.

La modularité de la preuve soulève néanmoins des problématiques liées à d'abstraction de données et à la préservation des invariants des données. Ces problématiques forment le sujet de cette thèse. Elles seront détaillées dans le chapitre 1. Ces problématiques ont été mises en avant en particulier en 2007 par Leavens, Leino et Müller [66], qui ont listé des challenges pour la vérification de programmes orientés objets, où la modularité et l'abstraction ont une place prépondérante. En 2010, un ensemble de programmes a été proposé par Leino et Moskal [69] : *VACID benchmarks (Verification of Ample Correctness of Invariants of Data-structures)* disponible dans [10]. Ces exemples, c'est-à-dire ces programmes, illustrent les défis que représentent la gestion du partage de références (aliasing) et des invariants de données pour la vérification déductive modulaire.

Des méthodes variées ont été proposées dans la littérature pour traiter ce genre de programmes. Nous détaillerons ces méthodes à la section 1.3, mais de façon générale on peut dire que ces méthodes sont toutes significativement complexes à mettre en œuvre. Notre objectif est de proposer un langage des constructions de spécifications établissant un compromis entre des spécifications légères et faciles à écrire pour l'utilisateur et des restrictions sur la classe des programmes qu'il est possible de traiter.

Nous considérons les langages dotés d'un langage de spécification tel que ACSL [21] pour C, JML [32] pour Java, Spec# [19] pour C#, etc. Le but de cette thèse est de décrire une méthode pour spécifier et prouver formellement des programmes écrits en C qui manipulent des structures de données avec des invariants. La démarche suivie pour la réalisation de ce travail est présentée dans le chapitre suivant.

Chapitre 1

Préliminaires et état de l'art

Sommaire

Le présent chapitre est structuré comme suit. Nous introduisons dans la section 1.1 la vérification déductive de programmes et les principaux moyens permettant de l'effectuer. Nous nous intéressons ensuite, dans la section 1.2, aux outils de vérification de programmes C. La section 1.3 est un état de l'art, nous y présentons des travaux récents sur des méthodes permettant la gestion des invariants de données et l'abstraction des données en général, et des effets de bord en particulier, dans des programmes pouvant contenir des alias (partage de références sur des pointeurs). Nous résumons enfin dans la section 1.4, les contributions de cette thèse.

1.1 Vérification déductive

La vérification déductive est une méthode de preuve de programmes qui repose sur la génération de formules logiques appelées *obligations de preuves*, dont la validité implique la correction du programme. Les premières contributions significatives dans ce domaine sont celles de Floyd en 1967 [55] puis de Hoare en 1969 [60], introduisant la logique de Floyd-Hoare.

Ces dernières années, des progrès considérables ont été accomplis dans le domaine de la vérification déductive des programmes. La programmation par contrat, introduite par B. Meyer dans son langage *Eiffel* [75] est une méthodologie qui permet de spécifier des programmes en associant des contrats aux fonctions et méthodes. Un contrat est composé de pré et post-conditions et de clauses qui permettent de spécifier les effets de bord de la fonction annotée. Certains langages de programmation tels que Java, C# ou bien C ont été munis de langages de spécification formels. Par exemple, JML pour Java, Spec# pour C# ou bien ACSL pour C. Ces langages permettent d'exprimer des propriétés sur les variables des programmes avec une syntaxe assez proche de celle des langages de programmation associés.

1.1.1 La logique de Floyd-Hoare

La logique de Floyd-Hoare [55, 60], que nous appellerons par la suite logique de Hoare, est un formalisme logique qui permet de raisonner sur la correction des programmes. L'idée est d'annoter un programme S avec une précondition P qui permet de caractériser l'ensemble des états possibles avant l'exécution de S et une post-condition Q qui spécifie les états possibles après l'exécution de S. Un programme et sa spécification sont alors représentés par ce qu'on appelle un *triplet de Hoare* :

$$\{P\}\ S\ \{Q\}$$

où P et Q sont des propriétés exprimées par des formules, typiquement dans la logique du premier ordre.

Exemple 1.1.1 *Considérons l'exemple, présenté dans la figure 1.1 ci-dessous, de la fonction puissance qui prend deux arguments : un réel x et un entier n et calcule x^n.*

```
Power (real x) (integer n) : real =
{P : n ≥ 0}
  real r = 1;
  real t = x;
  integer i = n;
  while(i > 0) {
  {Invariant : i ≥ 0 and x^n == t^i * r}
    if(i % 2 == 1) then r = r * t;
    t = t * t;
    i = i/2;
  }
  return r
{Q : r == x^n}
```

FIGURE 1.1 – Spécification de la fonction $Power$.

Cette fonction est annotée par une spécification formelle qui comporte :

- Une précondition P : la condition sous laquelle la fonction $Power$ peut être invoquée, qui est que le paramètre n doit être positif ($n \geq 0$).
- Une post-condition Q : la condition qui doit être satisfaite à la fin de la fonction, à savoir que la valeur retournée par la fonction $Power$ est égale à x élevé à la puissance n ($Q : r == x^n$).

En plus des pré et post-conditions, on spécifie un invariant de la boucle. C'est une condition qui doit être valide au début, à chaque itération et à la fin de la boucle : $i \geq 0\ and\ x^n == t^i * r$.

On dit que le programme S *satisfait sa spécification* si le triplet de Hoare $\{P\}\ S\{Q\}$ est *valide* dans l'un des deux sens suivants : la *correction partielle*, dans laquelle un triplet de Hoare $\{P\}\ S\ \{Q\}$ est valide, si pour tout état initial vérifiant P, si l'exécution de S se termine, alors Q est vraie après l'exécution de S ; ou la *correction totale*, dans laquelle un triplet de Hoare $\{P\}\ S\ \{Q\}$ est valide, si pour tout état initial qui satisfait la formule P, l'exécution de S se termine dans un état qui vérifie Q.

La *logique de Hoare* est un ensemble de règles d'inférence, qui produisent des triplets de Hoare valides. Nous ne donnons pas les règles dans cette thèse, car nous n'en avons pas besoin.

1.1.2 Calcul de la plus faible précondition

Si le triplet de Hoare $\{P\}\ S\ \{Q\}$ est valide et que, pour toute formule P' telle que le triplet $\{P'\}\ S\ \{Q\}$ est valide, $P' \Rightarrow P$, alors P est la plus faible précondition de S par rapport à Q, que l'on note $\mathrm{WP}(S,\ Q)$. Autrement dit, on appelle plus faible précondition d'un programme S, la condition impliquée par toutes les préconditions admissibles de S. Ainsi, pour prouver qu'un triplet de Hoare $\{P\}\ S\ \{Q\}$ est valide, il suffit, de montrer que $P \Rightarrow \mathrm{WP}(S,\ Q)$.

Dijkstra [51] a montré qu'un telle plus faible précondition existe toujours et peut se calculer de manière systématique. C'est une telle méthode qui est utilisée dans ESC/Java [38], Spec# [19], et la plateforme Why [54], pour générer des obligations de preuve. Dans un contexte légèrement différent mais dans la même optique, elle est utilisée dans la méthode B [13] pour le développement de programmes corrects par construction : la théorie du raffinement est en effet fondée sur une sémantique de plus faible précondition.

Nous présenterons un tel calcul dans le chapitre 3 de cette thèse, spécialisé pour notre langage d'étude où la seule catégorie d'effets de bord est la mise à jour de champs de structure.

Exemple 1.1.2 *Nous montrons ici la plus faible précondition de la fonction* $Power$ *(figure 1.1), calculée (par Why3) à partir de la post-condition* $r == x^n$.

$$
\begin{array}{l}
(0 <= n\ \wedge\ (1 * x^n) = x^n) \wedge \\
(\forall\, e : int, p : int, r : int.\ 0 <= e\ \wedge\ (r * p^e) = x^n \Rightarrow \\
\quad (\texttt{if}\ e > 0\ \texttt{then} \\
\quad\ \ \texttt{if}\ e\,\%\,2 = 1\ \texttt{then} \\
\quad\quad \forall\, r1 : int.\ r1 = (r * p) \Rightarrow (\forall\, p1 : int.\ p1 = (p * p) \Rightarrow \\
\quad\quad (\forall\, e1 : int.\ e1 = e/2 \Rightarrow\ (0 \leq e1\ \wedge\ (r_1 * p_1^{e_1}) = x^n)\ \wedge\ 0 \leq e\ \wedge\ e1 < e)) \\
\quad\ \ \texttt{else} \\
\quad\quad \forall\, p1 : int.\ p1 = (p * p) \Rightarrow \\
\quad\quad (\forall\, e1 : int.\ e1 = e/2 \Rightarrow\ (0 \leq e1\ \wedge\ (r * p_1^{e1}) = x^n)\ \wedge\ \ 0 \leq e\ \wedge\ e1 < e) \\
\quad \texttt{else}\ r = x^n))
\end{array}
$$

Une plus faible précondition telle que celle ci-dessus doit être montrée valide. Une telle tâche peut être confiée à un prouveur automatique, si celui-ci est capable de comprendre les quantificateurs du premier ordre et l'arithmétique entière. Typiquement, ceci peut être traité par des solveurs SMT : Alt-Ergo [28], Simplify[50], Yices[52], Z3[47], CVC3[20], etc.

1.1.3 Raffinement

Le raffinement est une technique utilisée au cours du processus de développement logiciel pour transformer un modèle abstrait d'un système logiciel (la spécification) en un modèle plus concret. C'est un procédé de construction qui consiste à générer graduellement du code, à partir d'une spécification formelle P qui exprime de manière abstraite le comportement du programme qu'on souhaite réaliser. Chaque étape du cycle de développement des logiciels peut être considérée comme un raffinement impliquant l'écriture de preuves mathématiques pour le justifier. L'ensemble de ces preuves garantit que le programme ainsi construit respecte la spécification initiale P, autrement dit : qu'il est correct par construction.

La notion du raffinement a été introduite dans les années 1970 par Dijkstra [51], puis formalisée par Back [15] dans les années 1980. Plusieurs travaux ont ensuite développé cette notion, en particulier Abadi et Lamport [12], Back [15], Morgan [77], Morris [78] et Abrial [13].

Dans la théorie de raffinement, un composant abstrait permet d'introduire des variables abstraites et de définir des opérations par le comportement de ces variables. Un composant qui raffine est défini en introduisant de nouvelles variables concrètes et en réécrivant les opérations à l'aide de ces variables. La relation entre les variables abstraites et concrètes est définie par un *invariant de collage*. Un composant peut ainsi être raffiné en plusieurs étapes, jusqu'à ce que les comportements de toutes les procédures soient des programmes à part entière. Dans la méthode B, un composant est appelé *machine*. Nous présentons dans la figure ci-dessous, les machines utilisées dans cette méthode pour décrire des composants. La machine abstraite M_1 permet de déclarer un ensemble de variables abstraites $x_1, \ldots, x_n$ initialisées

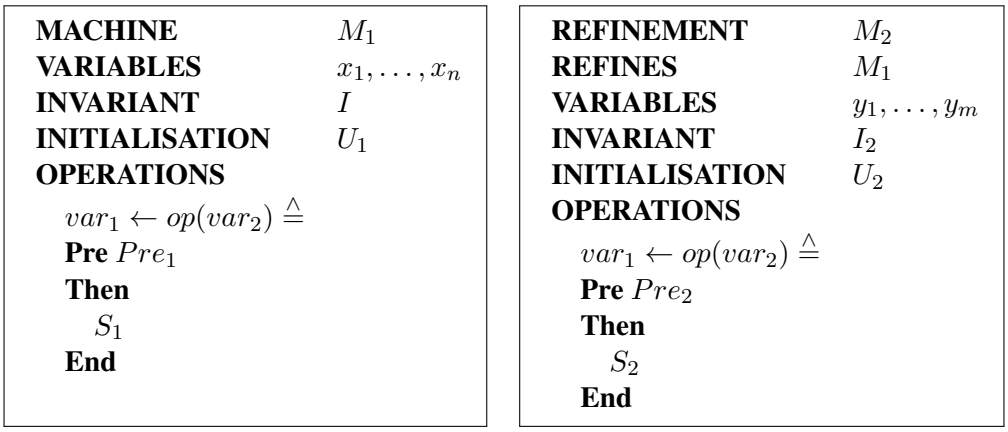


FIGURE 1.2 – Raffinement d'une machine abstraite dans la méthode B.

dans U_1, un invariant de données I, et une opération op dont la précondition est Pre_1. Cette opération prend un paramètre var_2 et stocke le résultat retourné dans var_1.

La machine M_2 est un raffinement de M_1. Elle permet de déclarer des variables concrètes $y_1, \ldots, y_m$ initialisées dans U_2, de définir l'opération op et de définir l'invariant de collage I_2 qui spécifie, entre autres, la relation entre les variables $x_1, \ldots, x_n$ et $y_1, \ldots, y_m$.

Le calculateur de Morgan[77] est un exemple simple et classique dans la théorie du raffinement. Il a pour but de collecter une suite de réels et d'en calculer la moyenne. Le comportement des procédures appliquées à un calculateur de Morgan peut s'exprimer en termes d'opérations sur des multi-ensembles de réels (il est en effet possible d'avoir plusieurs occurrences d'un même élément dans un calculateur).

Exemple 1.1.3 *Décrivons l'exemple du calculateur de Morgan dans une syntaxe à la B.*

MACHINE $Morgan\_abst$
VARIABLES $values$
INITIALISATION $values := \emptyset$
OPERATIONS
$add(x) =$
$values := values \cup \{x\}$
End ;
$result \leftarrow mean =$
Pre $values \neq \emptyset$
Then $result := sumbag(values)/card(values)$
End

L'opération add *permet de rajouter un nouvel élément* x *au calculateur. Le multi-ensemble des éléments du calculateur après l'exécution de cette opération est alors égal à l'union de l'ancienne valeur de ce multi-ensemble et du singleton composé de* x*.*

L'opération $mean$ *retourne la moyenne des éléments du calculateur, c'est-à-dire le résultat de la division de la somme des éléments collectés dans le calculateur par leur nombre.*

Nous raffinons ensuite cette machine en utilisant deux variables concrètes $count$ *et* sum*, qui représentent, respectivement, le nombre d'éléments collectés et leur somme, et en définissant l'invariant de collage qui définit la relation entre ces variables et la variable abstraite* $values$*.*

MACHINE $Morgan$
REFINES $Morgan\_abst$
VARIABLES $sum, count$
INVARIANT $sum = \mathtt{sumbag}(values) \wedge count = \mathtt{card}(values)$
INITIALISATION $sum := 0,\ count := 0$
OPERATIONS
$add(x) =$
Begin
$sum,\ count := sum + x,\ count + 1$
End ;
$result \leftarrow mean =$
$result := sum/count$

L'implémentation de l'opération add *permet de rajouter la valeur du paramètre* x *à* sum*, et d'incrémenter* $count$ *de 1.*

L'opération $mean$ *retourne le résultat de la division de* sum *par* $card$*.*

Le *calcul de raffinement* [77, 14] est une logique de programmes qui favorise l'approche incrémentale du développement formel de programmes : à partir de spécifications abstraites jusqu'à l'implémentation. Techniquement, le calcul de raffinement consiste à vérifier pour chaque opération :

$$\forall c, x, a.\ (P \wedge I) \Rightarrow \exists a'.\mathrm{WP}(S, ((Q \wedge I)[a \mapsto a']))$$

telle que x sont les paramètres en entrée, a les variables abstraites, c les variables concrètes, P la précondition, I l'invariant de collage, Q la post-condition et S le corps de l'opération. Du point de vue du client, cette formule s'interprète comme suit : pour tout état concret c atteignable lors de l'exécution

du code client, il existe une valeur a' de la variable a qui satisfait l'invariant I après l'exécution. Par exemple, dans le code client, on peut, de manière sûre, remplacer l'exécution de S par une mise à jour non déterministe de la valeur de la variable abstraite a en choisissant n'importe quelle valeur a' satisfaisant les deux formules Q et I. L'obligation de preuve de n'importe quelle opération appelée assure que le reste du code client est correct pour tous les autres choix possibles pour cette mise à jour.

Notons que pour l'initialisation d'un composant, une obligation de preuve qui permet d'établir l'invariant est générée :

$$\exists a'; \mathrm{WP}(U, I[a \mapsto a'])$$

Exemple 1.1.4 *L'obligation de preuve de raffinement de l'opération* `add` *est :*

$$\begin{array}{l} \forall count, sum, values, x; \\ \quad (sum = \texttt{sumbag}(values) \wedge count = \texttt{card}(values)) \Rightarrow \\ \quad \exists values'; values' = values \cup \{x\} \wedge \\ \qquad (sum + x = \texttt{sumbag}(values') \wedge count + 1 = \texttt{card}(values')) \end{array}$$

1.1.4 La plateforme Why3

Le système Why3 [29] est un environnement pour la *vérification déductive* de programmes : il permet de coder des programmes en les annotant avec des spécifications logiques, puis de générer les obligations de preuve qui garantissent leur correction fonctionnelle. La plateforme Why3 est composée de deux parties : un langage logique appelé *Why* avec une infrastructure permettant une traduction vers des prouveurs automatiques : Gappa [46], Alt-Ergo [28], Simplify [50], Yices [52], Z3 [47], CVC3 [20], etc. ou des prouveurs interactifs comme Coq [27], PVS [89], Isabelle/HOL [83], etc, Et un langage de programmation *WhyML* muni d'un générateur d'obligations de preuve. Les obligations de preuve générées peuvent être soumises à une large collection de prouveurs automatiques ou interactifs.

Dans le paysage des outils pour la vérification déductive, Why3 se veut à mi-chemin entre les environnements interactifs offrant des langages très expressifs mais un niveau faible d'automatisation des preuves (Coq, PVS, Isabelle/HOL, etc.), et les systèmes plus automatiques mais munis de langages plus pauvres (Spec#, VCC, Frama-C, KeY, etc.). Un objectif naturel est de récupérer le meilleur possible des deux extrêmes : un langage suffisamment expressif qui permet des preuves largement automatiques.

Dans l'approche par raffinement mais également dans le langage *WhyML* de la plateforme Why3, l'aliasing (partage de référence) n'est pas permis [31]. Or nous nous intéressons à des programmes de type C où il est possible de partager des références vers un pointeur. Nous considérons ce cas dans la section suivante.

1.2 Vérification déductive de programmes C

Le besoin de la communauté, dans le domaine de la vérification déductive de programmes C, a incité le développement d'outils qui permettent la gestion de programmes de type Java ou C : ESC/Java [38], Spec# [19], Key [22], VCC [44], Why [54], Frama-C [56], etc.

Cette thèse se place dans le contexte de *Frama-C*.

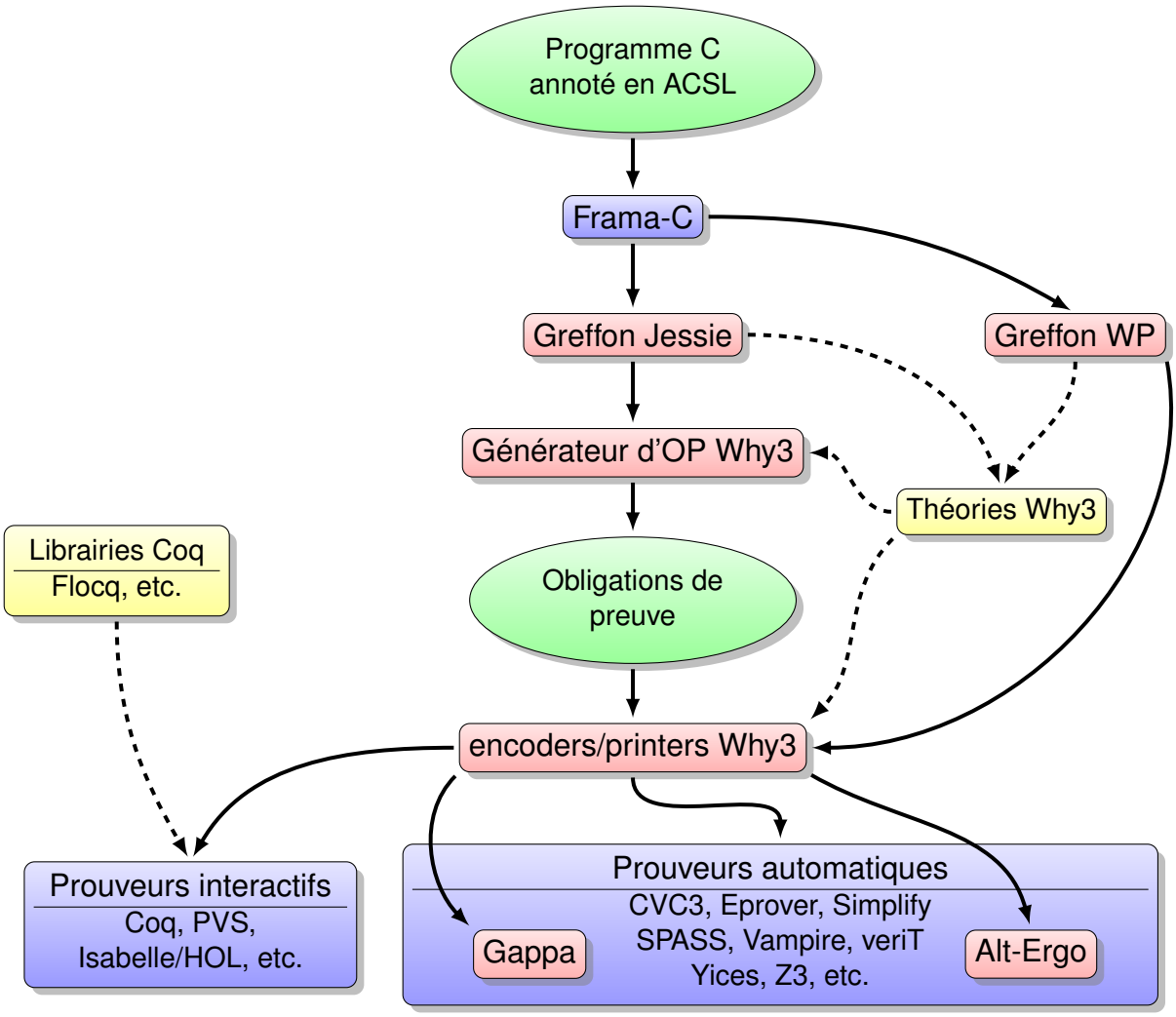


FIGURE 1.3 – Les chaînes de vérification Frama-C/ Jessie / WP/ Why3.

1.2.1 La plateforme de vérification *Frama-C*

Frama-C est une plateforme de vérification modulaire de programmes C, développée en collaboration entre le *CEA-List* et l'équipe *Toccata* (anciennement Proval) commune à INRIA Saclay Île-de-France, le Laboratoire de Recherche en Informatique et l'université Paris-Sud. Concrètement, c'est une suite d'outils dédiés à l'analyse de code C, qui regroupent plusieurs techniques d'analyse statique dans un seul environnement. L'approche collaborative permet aux analyseurs d'exploiter les résultats calculés par d'autres analyseurs dans la plate-forme. Frama-C est composé de greffons (plugins) qui sont destinés à faire des tâches spécifiques d'analyse statique. Nous présenterons dans la suite deux greffons : *Jessie*[80] et *WP* [11], destinés à la vérification déductive.

1.2.2 Le langage de spécification ACSL

Le langage de spécification `ACSL` (*ANSI/ISO C Specification Langage*) est un langage pour la spécification du comportement de programmes écrits dans le langage C, dont la conception est inspirée de

```
typedef struct Purse {
  int balance;
} *purse;

/*@ assigns \nothing;
  @ allocates \result;
  @ ensures \result->balance == 0;
  @*/
purse create_Purse() {
  purse this = (purse) malloc(sizeof(struct Purse));
  this->balance = 0;
  return this
}

/*@ requires s >= 0;
  @ assigns p->balance;
  @ ensures p->balance == \old(p->balance) + s;
  @*/
 void credit(purse p, int s) {
  p->balance += s;
}
```

FIGURE 1.4 – Annotation de fonctions en ACSL

JML. Il permet d'exprimer des propriétés sur le comportement fonctionnel des programmes C, par des annotations dans le code source.

Le concept clé dans ce type de langage est le contrat de fonction. ACSL permet d'écrire des contrats de bas niveau, on peut par exemple spécifier qu'une fonction requiert un pointeur valide, ou des contrats de haut niveau, comme s'assurer qu'une pile n'est pas vide avant de retirer l'élément à son sommet.

Exemple 1.2.1 *Considérons le type structure* `Purse` *(figure 1.4) qui a un champ* `balance` *représentant le contenu de la bourse. La fonction* `credit(purse p, int s)` *permet de créditer la bourse* `p` *d'un montant d'une valeur* `s` *dans la monnaie considérée.*

La modification de l'état de la mémoire par une fonction, lors de la mise à jour de la valeur d'une variable ou d'un pointeur, est indiquée dans le contrat à l'aide de la clause `assigns`*. C'est le cas, par exemple, du champ (*`p` $\rightarrow$ `balance`*) dans la fonction* `credit`*. De même l'allocation de mémoire est spécifiée par la clause* `allocates`*.*

1.2.3 Le greffon Jessie pour la vérification déductive

Le greffon Jessie fait partie des outils qu'offre *Frama-C* pour la vérification de programmes C. Il repose sur la vérification déductive et utilise la logique de Hoare et le calcul de plus faible précondition pour prouver formellement des propriétés ACSL. Concrètement, les programmes C sont annotés en utilisant le langage de spécification ACSL, puis traduits en *WhyML*. Le générateur d'obligations de preuve de Why3 fabrique alors des obligations de preuve, dont la validité implique la correction du programme vis-à-vis de sa spécification (figure 1.3).

Une spécificité du greffon Jessie est sa modélisation du tas mémoire (son *modèle mémoire*) qui sépare statiquement les champs de structure, et également se base sur une analyse statique de séparation pour représenter les portions du tas par le plus grand nombre possible de variables distinctes [79, 61].

1.2.4 Le greffon WP pour la vérification déductive

WP [11] est un autre greffon de *Frama-C* pour la vérification déductive de programmes C, annotés en ACSL. Il utilise un calcul de plus faible précondition pour générer des obligations de preuve.

Ce greffon se distingue du greffon Jessie sur au moins deux aspects. Le premier est le modèle mémoire : ce greffon propose plusieurs modèles mémoire possibles en fonction des caractéristiques du programme source analysé, par exemple s'il y a des cast de pointeurs ou pas. Le second aspect est que le calcul de plus faible précondition y est intégré étroitement avec les modèles mémoire, là où Jessie se contente de générer un programme intermédiaire *WhyML* et délègue le calcul de plus faible précondition à Why3.

Applications existantes de *Frama-C*

L'environnement *Frama-C* est diffusé librement et largement utilisé aussi bien dans le monde académique que dans l'industrie [42].

Parmi les utilisations dans le monde académique, on peut mentionner quelques greffons qui ont été développés par des équipes externes à *Frama-C*. Ceara *et al* [35] ont implémenté un greffon d'analyse de vulnérabilité (*taint analysis*), qui permet d'identifier des chemins d'exécution pouvant mener le système dans un état risqué. Le greffon de Demay *et al* [49] génère des moniteurs de sécurité. Il se base sur l'analyse de valeurs offerte par le greffon *Value* [43] de *Frama-C*. Berthomé *et al* [26] utilisent également l'analyse de valeurs pour détecter des attaques sur des cartes à puces. Bouajjani *et al* [30] proposent un greffon pour synthétiser automatiquement des annotations (en particulier pour des programmes sur des listes simplement chaînées). D'autres groupes de recherche ont développé des études de cas. Ainsi, Burghardt *et al* [34] décrivent des exemples de spécifications, d'implantation et de vérification déductive de programmes C annotés avec ACSL et prouvés en utilisant les greffons *Jessie* et *WP* de *Frama-C*.

Du côté industriel, il existe également des applications de *Frama-C*. Pour des applications chez *Airbus*, Cuoq *et al* [41] développent un greffon qui permet de vérifier les standards de codage spécifiques à cette compagnie. Également, Delmas *et al* [48] utilisent *Value* pour vérifier les diagrammes de contrôle et de flot d'une application développée en interne. Chez *Dassault Aviation*, Pariente et Ledinot [86] vérifient un système de contrôle de vols en utilisant des greffons *Frama-C*, tels que *Value* ou *Slicing*. Ils fournissent une comparaison entre leur approche et les techniques traditionnelles de vérification. Yakobowski *et al* [99] utilisent les greffons Value et WP pour s'assurer de l'absence d'erreurs à l'exécution d'un code utilisé dans le domaine du nucléaire. Dans le cadre du projet *ACCoRD* de la *NASA* [5], Goodloe *et al* [57], analysent la précision numérique de codes pour le contrôle aérien.

1.3 État de l'art

Les programmes avec alias sont significativement plus compliqués que les programmes sans alias. Ainsi, passer d'une approche de vérification déductive sur des programmes sans alias comme c'est le

cas de la logique de Hoare ou de la théorie du raffinement (méthode B), à une vérification déductive sur des programmes avec alias tels que les programmes écrits dans le langage C, est un pas conséquent.

Parmi les problématiques que l'on rencontre lorsqu'on souhaite spécifier et prouver des programmes, celles liées à l'abstraction de données est fondamentale. L'importance de l'abstraction est reconnue depuis longtemps et n'est pas propre aux programmes avec alias. En effet, en 1972, Hoare publie son article *"Proof of Correctness of Data Representations"* dans lequel il soutient qu'un développeur devrait écrire des programmes abstraits, utilisant des structures de données abstraites, à partir desquels des programmes concrets peuvent être dérivés de façon automatique.

La spécification d'un programme a besoin d'être abstraite pour plusieurs raisons. La première est que les spécifications décrivent le comportement fonctionnel des programmes, c'est-à-dire ce que fait le programme et non pas *comment* il le fait. La seconde est que la spécification d'un programme peut être utilisée par un autre programme, c'est-à-dire un programme client. C'est ce qu'on appelle le modèle du *producteur-consommateur*. Exhiber des détails d'implémentation dans la spécification, dans ce cas, réduirait la robustesse de celle-ci. Leavens *et al.* [66] exposent un certain nombre de défis pour la spécification et la vérification de programmes orientés objet. Parmi ces défis, les problématiques liées à l'abstraction des données dans les spécifications prennent une place importante.

La spécification des effets de bord caractérise bien cette problématique. Spécifier les effets de bord d'une fonction donnée consiste à dire quels sont les emplacements mémoire qu'elle est autorisée à modifier lors de son exécution. Dans ACSL, les effets de bord sont spécifiés dans la clause **assigns** du contrat de la fonction. Dans JML, cette clause s'appelle **assignable**. Si une fonction modifie la valeur d'une variable, alors cette variable doit être mentionnée dans la clause **assigns**. C'est le cas, par exemple, de la fonction `credit` présentée dans l'exemple 1.2.1. Le problème avec cette solution est que le champ `balance` est spécifique à l'implémentation du type structure que nous avons proposé. Or, la spécification ne doit contenir que des informations abstraites. Les détails liés à l'implémentation devraient être cachés. Concrètement, les spécifications visibles par un client doivent être exprimées indépendamment de l'implémentation, de façon à permettre l'encapsulation. Le développeur ou le concepteur devraient être libres de changer la représentation des données sans pour autant changer la spécification. L'abstraction des données est un moyen d'y parvenir. Par exemple, une liste chaînée peut être représentée par une séquence mathématique, le comportement d'une liste peut alors être exprimé en terme d'opérations sur les séquences. Le raffinement présenté dans la section précédente est une méthode qui permet de mettre en œuvre une telle abstraction, et qui relie l'état abstrait et l'état concret en utilisant un invariant de collage.

La notion d'invariant est très importante dans les langages de spécification. Un invariant peut être associé à une classe ou à un type. Il peut également établir la relation entre les variables abstraites et concrètes dans une conception par raffinement. Dans le calculateur de Morgan (section 1.1.3) par exemple, l'invariant de collage est une propriété qui permet de relier les variables concrètes `sum` et `count` et la variable abstraite `values`, et que toute instance de type `Calc` doit vérifier. Les invariants peuvent décrire des propriétés telles que : la taille d'un tableau est positive, dans un tas, la valeur d'un nœud est inférieure aux valeurs des nœuds fils. On s'intéresse, dans ce qui suit, aux invariants de données, c'est-à-dire aux invariants permettant de spécifier une propriété devant être observée sur toutes les instances d'un type.

Exemple 1.3.1 *Nous munissons le type* `purse` *introduit dans l'exemple 1.2.1, d'un invariant que toute instance de ce type doit vérifier, qui établit que le champ* `balance` *est toujours positif. Nous utilisons ici la syntaxe des invariants de type existante en ACSL.*

La difficulté de vérifier la préservation d'un invariant de données est connue [66]. Parmi les inter-

```
typedef struct Purse {
  int balance;
} *purse;

//@ type invariant balance_non_negative(purse s) = s->balance >= 0;
```

FIGURE 1.5 – Exemple d'invariant de données

rogations soulevées : Pour quelles instances faut-il vérifier si l'invariant est préservé ou s'il est rompu ? Quand doit-on le faire ? Le spécifier explicitement peut être en contradiction avec le principe d'abstraction de données, car l'invariant est propre à une implémentation et ne doit, par conséquent, pas être visible par les utilisateurs du type auquel l'invariant est associé.

Exemple 1.3.2 *La figure 1.6 illustre un bout de code C comportant un premier type structure* `Sensor` *qui permet de lire le nombre de rotations par minute des roues d'une voiture et un second type structure* `Car` *dont l'un des champs est de type* `Sensor`*. La fonction* `update` *permet d'afficher la vitesse de la voiture. Cette dernière est spécifiée par un invariant de données qui permet d'établir la relation entre la valeur de la sonde et la vitesse de la voiture. Notez que dans le corps de* `update(car c)`*, l'invariant du paramètre* `c` *de la fonction est temporairement rompu.*

```
typedef struct Sensor{
   int rpm;
 }*snesor;

void read(sensor s){ rpm = 2; }

typedef struct Car{
   struct Sensor sensor;
   int display;
 }*car;

//@ predicate Inv_Car(car this) = this->display = K* (this->sensor->rpm);

car create_car(snesor s){
  car this = (car)malloc(sizeof(struct Car));
  this->sens = s;
  this->display = K * (this->sensor->rpm);
  return this;
}

void update(car this){
  read(this->sensor);
  this->display = K * (this->sensor->rpm);
}
```

FIGURE 1.6 – Violation temporaire de l'invariant

Une idée inspirée de la théorie du raffinement est de considérer qu'un invariant de données est vérifié au début et à la fin d'une fonction. C'est également le cas dans JML, où les invariants de toutes

les données accessibles par la fonction doivent être vérifiés au début et à la fin de la fonction.

Nous allons maintenant illustrer, par deux exemples, deux problèmes difficiles qui apparaissent quand on veut maintenir des invariants en présence d'aliasing.

Violation inattendue d'invariant à cause d'aliasing

Exemple 1.3.3 *Dans le code de la figure 1.7, on a un partage de référence (alias) entre le champ* `sensor` *de* `c` *et l'instance* `s` *de* `Sensor`*. Ainsi, l'invocation de la fonction* `read` *avec le paramètre* `s` *modifie la valeur* `rpm` *du champ* `sensor` *de* `c` *et viole, par conséquent, son invariant.*

```
void main(){
  sensor s = (sensor)malloc(sizeof(struct Sensor));
  car c = create_car(s); //alias
  read(s); // violation de l'invariant de c
}
```

FIGURE 1.7 – Violation de l'invariant à cause de l'alias

Abstraction des effets de bord et réentrance

L'exemple ci-dessous montre que cacher les effets de bord *privés*, c'est-à-dire liés à l'implémentation peut s'avérer incorrect en cas de réentrance. En effet, un objet peut, indirectement, être client de lui-même, à cause des principes de réentrance et du typage dynamique. En fait, un tel problème se produirait également dans la méthode B, si la récursion mutuelle entre les composants était autorisée.

Exemple 1.3.4 *La figure 1.8 décrit une première classe abstraite* `A`*, dans la syntaxe Java, qui fournit une méthode* `m()` *sans effets de bord. Une seconde classe* `B` *est définie. Cette classe contient une méthode qui appelle la méthode* `m` *de la classe* `A` *dans son corps.*

Ensuite, la classe `C` *étend (raffine) la classe* `A` *en définissant un champ privé* `x` *que l'implémentation de la fonction* `m` *modifie. Si nous considérons que cet effet de bord est privé et n'est par conséquent pas visible, alors, du point de vue d'un client de la classe* `C`*, la clause* ***assigns*** *est toujours* ***nothing***. *Ainsi, pour la méthode* `f` *de cette classe, nous devrions être en mesure de prouver l'assertion à partir de la spécification de la méthode* `B.g`*, alors que* `x` *a la valeur 1 après l'exécution de l'appel de* `B.g`*.*

Nous venons d'exposer les principales difficultés rencontrées lors de la spécification et la vérification de programmes, en présence d'invariants de données et d'alias. Nous allons dans ce qui suit présenter un état de l'art des solutions existantes dans la littérature, et qui permettent de répondre à au moins un des besoins soulevés précédemment.

1.3.1 Les champs modèle et DataGroups

Un des moyens utilisés pour mettre en œuvre l'abstraction de données est le recours aux champs modèle. Comme les *datagroups* que nous présenterons un peu plus loin dans cette section, ils imitent le raffinement dans les langages à pointeurs.

```
abstract class A {
    //@ assigns \nothing;
    void m();
}

class B{
    //@ assigns \nothing;
    void g(A a){ a.m();}
}

class C extends A{
    private int x;

    void m() { x++; }

    void f(B b){
        x = 0;
        b.g(this);
        //@ assert x == 0;
    }
}
```

FIGURE 1.8 – Exemple de réentrance

Les champs modèle permettent de décrire une représentation abstraite de l'état d'un objet. Syntaxiquement, ces champs ressemblent à des champs classiques, mais ils n'ont pas la même sémantique. Ils sont utilisés dans la spécification uniquement, et donc pas visibles dans le programme. De plus, il n'est pas possible de leur affecter une valeur, ceci ne veut pas dire qu'ils n'ont pas de valeur : un champ modèle peut être associé à une fonction de représentation. Cette fonction exprime la valeur du champ modèle abstrait en termes de représentation concrète. Autrement dit, la valeur de ces champs est déterminée par la spécification et non pas par le programme.

Les clients d'un programme utilisant des champs modèle, peuvent faire référence aux valeurs de ces champs modèle dans les assertions, sans aucune connaissance de la façon dont la représentation abstraite à laquelle ils appartiennent est implémentée.

En JML, un champ modèle est déclaré en utilisant le mot clé **model**, et sa valeur est spécifiée avec la clause **represents**. Par exemple :

```
//@ model int sum;
```

déclare un champ modèle `sum` de type `int` et la clause

```
//@ represents sum <- (\sum int i = 0; i < a.length; a[i]);
```

ou bien

```
//@ represents sum \such_that sum == (\sum int i = 0; i < a.length; a[i]);
```

définit que la valeur de `sum` est égale à la somme des éléments du tableau d'entiers `a`.

La valeur de ce type de champ est alors déterminée en appliquant la fonction ou la relation, introduite

en utilisant la clause « **represents** **such_that** », de représentation sur l'état concret de l'objet.

Exemple 1.3.5 *Dans la figure 1.9, nous déclarons une interface* `IEuros` *qui permet de calculer l'addition et la soustraction en euros. Dans cette interface, le champ modèle* `value` *représente l'état d'une instance de type* `Euros` *par un réel.*

```
interface IEuros {

    //@ model real value;
    //@ invariant this.value >= 0.0;

    /*@ requires a.value >= 0;
      @ assigns this.value;
      @ ensures this.value == \old(this.value + a.value);  */
    void add(Euros a);
}
```

FIGURE 1.9 – Interface de la classe `Euros`

Dans l'implémentation correspondante, donnée dans la figure 1.10, un réel est codé par deux entiers.

Dans cet exemple, la fonction de représentation du champ modèle `value` est définie par l'invariant de collage.

Les champs modèle permettent, ainsi, d'abstraire l'état d'un objet. Mais à notre connaissance, la première approche pour l'abstraction et donc le raisonnement modulaire sur les effets de bord, qui est sûre même en cas de réentrance, a été proposée par Leino [67, 71] et utilise les *datagroups*. Les datagroups permettent de regrouper, dans une même collection, plusieurs emplacements mémoire. Ils sont utilisés dans les clauses **assigns** pour exprimer que tous les emplacements mémoire qui appartiennent au datagroup spécifié sont potentiellement modifiables. L'appartenance d'un champ à un datagroup est spécifiée au moment de la déclaration du champ en question. Un datagroup peut contenir un autre datagroup, et un champ peut appartenir à plusieurs datagroups. La principale caractéristique des datagroups est qu'ils peuvent être étendus dans les sous classes avec de nouveaux champs (privés ou publics).

Exemple 1.3.6 *Reprenons le calculateur de Morgan, que nous écrivons dans une syntaxe Java, dans lequel nous introduisons un datagroup, appelé* `Gvalues`*, qui contient le champ modèle* `values` *dans la classe abstraite (figure 1.11), et qui est étendu avec les champs* `sum` *et* `count` *dans la classe* `SmartCalc` *(figure 1.12).*

Notez que dans cet exemple, il aurait été plus utile de ne pas distinguer le datagroup `Gvalues` et le champs modèle `values`. Mais sur des exemples plus compliqués, il est nécessaire de séparer les deux concepts.

1.3.2 La logique de séparation

La logique de séparation, proposée par Reynolds *et al.* [90, 84], est une approche pour traiter les problèmes d'aliasing, qui a eu beaucoup de succès. La caractéristique principale de cette approche est

```
class Euros implements IEuros{

  private int euros=0;
  private byte cents=0;

  //@ invariant 0 <= euros && 0 <= cents < 100;
  //@ invariant coupling: value == euros + cents / 100.0;

  void add(Euros a) {
      euros += a.euros; cents += a.cents;
      if (cents >= 100) { euros++; cents -= 100; }
  }
}
```

FIGURE 1.10 – Implémentation de la classe `Euros`

```
abstract class Calc {
  //@ datagroup Gvalues;
  //@ model bag<real> values \in Gvalues;

  /*@ assigns Gvalues;
    @ ensures values == union(\old(this.values),singleton(x));*/
  abstract void add(double x);

  /*@ requires values != empty_bag;
    @ assigns \nothing;
    @ ensures \result == sum_bag(values)/card(values);*/
  abstract double mean();
}
```

FIGURE 1.11 – Définition de *Datagoups* dans une classe abstraite.

```
class SmartCalc extends Calc {
    private int count;  //@ \in Gvalues;
    private double sum; //@ \in Gvalues;
    /*@ invariant this.sum == sumbag(this.values)
      @        && this.count == card(this.values); */

    /*@ assigns \nothing;
      @ ensures this.values == empty_bag;*/
    SmartCalc() { ... }

    void add(double x) { ... }

    double mean() { ... }
}
```

FIGURE 1.12 – Extension du *datagroups* dans l'implémentation.

de changer la logique dans laquelle on exprime les spécifications. Nous illustrons cette approche par l'exemple du porte-monnaie électronique présenté dans l'exemple 1.2.1, que nous spécifions dans la figure 1.13 en utilisant la logique de séparation.

```
/*@ requires \emp;
  @ ensures \result->balance == 0;
  @*/
purse create_Purse() { ...}

/*@ requires s >= 0 &*& p->balance == _v;
  @ ensures p->balance == _v + s;
  @*/
void credit(purse p, int s) {...}
```

FIGURE 1.13 – Spécification dans la logique de séparation

La première différence par rapport à la spécification en ACSL (figure 1.4) est l'absence de la clause **allocates**. La clause **requires** indique que la fonction est exécutée dans un tas mémoire vide et la post-condition indique que le tas mémoire contient un pointeur \result . Cette spécification indique implicitement que la fonction `create_Purse` alloue un nouvel emplacement mémoire.

Par ailleurs, syntaxiquement, la logique de séparation introduit un nouveau connecteur `&*&` dont la sémantique est : la formule P `&*&` Q est vraie si et seulement si les formules P et Q sont vraies dans deux parties disjointes de la mémoire. De plus, dans cette logique $x \rightarrow f$, n'est pas un terme. On ne peut parler de la valeur d'un champ qu'avec des formules atomiques de la forme $x \rightarrow f$ == t où t est un terme indépendant du tas mémoire. Dans la spécification de la fonction `credit`, par exemple, nous spécifions qu'à l'appel de fonction, `p` $\rightarrow$ `balance` a une certaine valeur qu'on note `_v` et qu'au retour, il a la valeur `_v + s`. La notation `_v` indique que la variable `v` est liée existentiellement au début du contrat. Le fait d'avoir déplacé le **old**(`p` $\rightarrow$ `balance`) dans la précondition, indique que la cellule mémoire $p \rightarrow balance$ est déjà allouée à l'appel de la fonction. Sa présence dans la post-condition indique qu'elle le reste. Ceci illustre l'idée essentielle de la logique de séparation, qui est que le langage de spécification permet d'exprimer des propriétés sur les valeurs, mais aussi sur la structure du tas mémoire.

La logique de séparation est implémentée dans plusieurs outils tels que Smallfoot [25], VeriFast [62, 63], Ynot [81] et CFML [36]. Comme cette logique n'est pas la logique du premier ordre standard, ces outils ne peuvent utiliser directement des prouveurs automatiques comme les solveurs SMT. Ils utilisent soit des prouveurs dédiés, soit des traductions vers une logique plus standard (Z3 pour VeriFast et Coq pour Ynot et CFML). Les preuves à faire font souvent appel à des inductions (par exemple pour les listes chaînées) et ces inductions doivent être guidées par l'utilisateur (annotations d'ouverture ou de fermeture de prédicat en Smallfoot et VeriFast, preuves interactives en Coq).

D'autre part, comme la logique n'est pas standard, spécifier un programme avec la logique de séparation, demande un apprentissage supplémentaire pour un utilisateur lambda.

La gestion des *programmes concurrents* [53] est une application de la logique de séparation qui a eu du succès. Par exemple, deux threads qui s'exécutent en parallèle dans des mémoires séparées peuvent être facilement prouvés. Un autre application de la logique de séparation consiste à spécifier les invariants de données. En effet, on peut déduire qu'un invariant est préservé, si aucun des emplacements mémoire référencés dans l'invariant n'appartiennent à la partie du tas mémoire modifiée.

Les défauts principaux de la logique de séparation sont donc le manque de naturel des spécifications et son manque d'automatisation.

1.3.3 L'approche d'Ownership

Une autre approche qui permet de vérifier la préservation des invariants, notamment en présence de partage est celle de l'*Ownership*. Proposée par Barnett et. al en 2004 [17], cette approche fournit une technique de vérification des invariants, adaptée à la vérification déductive et implémentée dans *Boogie* [18].

Cette technique est basée sur l'idée qu'un objet est possédé par l'objet dans lequel il est déclaré, ce qui permet de définir une hiérarchie entre les types et de vérifier les invariants de données de façon modulaire. Un objet n'est alors accessible qu'à travers l'objet qui le possède (propriétaire).

De manière générale, les objets sont vus comme des boîtes qui peuvent être ouvertes ou fermées. Un objet fermé vérifie son invariant. Un objet ouvert, peut, lui, être mis à jour. L'utilisateur contrôle, ainsi, les portions de programme où il s'autorise à violer, temporairement, un invariant. Pour déterminer leur état, les objets sont munis d'un champ booléen particulier `inv` semblable à un champ modèle : si `o.inv = true` (la boite est fermée), alors l'objet `o` satisfait son invariant. Si, par contre, `o.inv = false` (la boite est ouverte), alors l'invariant de `o` peut être rompu.

Les objets possèdent un deuxième champ, `committed`, qui permet la gestion des invariants en présence d'objets imbriqués, c'est-à-dire quand les champs sont aussi des objets. En effet, la mise à jour d'un champ, dans ce cas, peut entraîner la violation de l'invariant d'un autre objet fermé, comme le montre l'exemple 1.3.2. Cette propriété est assurée par une discipline stricte. Tout d'abord, l'invariant d'un objet `o` ne peut mentionner que les objets accessibles via des champs précédés par le mot clé **rep**. Plus précisément, l'invariant de `o` ne peut référencer : $o.f_1 \dots f_n.g$ que si les champs $f_1; \dots; f_n$ sont déclarés comme **rep**. Si un champ `f` est déclaré **rep**, alors chaque fois que `o` est fermé, `o.f` doit être fermé, autrement dit, le champ `f` ne peut être modifiable, que si `o` est ouvert. C'est le principe du champ `committed`. Si le champ `committed` d'un objet `f` est à `true`, alors le propriétaire de cet objet vérifie son invariant. Si par contre, il est à `false`, alors l'objet qui possède `f` est ouvert. En termes de boites, on ne peut accéder à une boite que si elle est dans une boite ouverte.

Les champs `inv` et `committed` ne peuvent apparaître que dans la spécification, mais pas dans l'invariant, ni dans l'implémentation. Leurs valeurs ne peuvent être modifiées qu'à l'aide des deux instructions spéciales : *pack* et *unpack*.

Pour toute expression o de type T, on pose $Comp_T(o)$ l'ensemble des expressions $o.f$ pour chaque champ rep `f` dans `T` :

pack $o \equiv$
`assert` $o \neq null \wedge \neg o.inv$;
`assert` $Inv_T(o)$;
`foreach` $p \in Comp_T(o)$ {`assert` $p = null \ \vee \ p.inv$}
`foreach` $p \in Comp_T(o)$ {`assert` $p = null \ \vee (\ p.inv \ \wedge \ \neg\, p.committed\ )$}
`foreach` $p \in Comp_T(o)$ {`if`$(p \neq null)$ { $p.committed := true$ }}
$o.inv := true$;

Cette instruction vérifie que l'invariant de o est établi, que tous les composants p de o, c'est-à-dire les objets possédés par o, non nuls vérifient leurs invariants ($p.inv$), puis affecte la valeur $true$ au champ $p.commited$ de chaque composant p, ainsi qu'au champ $o.inv$.

unpack $o \equiv$
  `assert` $o \neq null \wedge o.inv$;
  $o.inv := false$;
  `foreach` $p \in Comp_T(o)$ {`if`$(p \neq null)$ { $p.committed := false$ }}

La mise à jour d'un champ peut rompre l'invariant de données. Une telle opération requiert, par conséquent, que l'objet soit ouvert. Autrement dit, que le champ `inv` soit à `false`, c'est-à-dire que l'instruction **unpack** soit exécutée.

Exemple 1.3.7 *Reconsidérons l'exemple 1.3.2, dans lequel nous avons montré que l'invariant peut être momentanément rompu.*

```
//@ requires this.inv && !this.committed;
void update(car this){
  //@ unpack this;
  read(this->s);
  this.display = K * (this->(s->rpm));
  //@ pack this;
}
```

La précondition de cette fonction spécifie que l'invariant de `this` *doit être vérifié à l'appel de fonction et que l'objet qui le possède est ouvert.*

Une fois dans le corps de la fonction, la première étape consiste à ouvrir l'objet `this` *à l'aide de l'instruction* ***unpack****. Dans ce cas là l'invariant de celui ci peut être rompu et donc la mise à jour des champs, notamment* `s` *possible. À la sortie de la fonction, l'invariant est rétabli, nous exécutons alors l'instruction* ***pack****.*

Comme on peut le constater sur cet exemple, cette méthode présente le même inconvénient que la logique de séparation dans le sens où elle requiert le rajout de beaucoup d'annotations. Il faut, par exemple, spécifier dans le contrat de la fonction qu'un objet vérifie ou pas son invariant à l'aide du champ inv etc.

Dans cette méthode, la séparation n'est pas écrite dans la logique, mais elle nécessite des annotations spéciales dont l'utilisation peut, là encore, devenir contraignante pour l'utilisateur. On pourrait alors penser qu'une approche dynamique, au sens où les annotations sont générées par des outils, est nettement plus facile du point de vue du client, mais Thierry Hubert montre, dans sa thèse[61], qu'une telle approche n'est pas non plus satisfaisante du fait du nombre important d'annotations générées, ce qui peut s'avérer gênant pour les prouveurs SMT. En effet, plus le contexte est important, moins il est facile pour un prouveur de prouver un but donné.

On aimerait alors exprimer les propriétés de séparation dans le typage, c'est le cas dans les systèmes de types avec régions. Cette façon de faire éviterait la génération d'un nombre important d'obligations de preuve.

1.3.4 Régions et capacités

La notion de *région* a été introduite, initialement, par Mads Tofte et Jean-Pierre Talpin [95, 96], dans le but d'analyser statiquement la mémoire allouée par un programme, en particulier l'aliasing, et

éventuellement de désallouer automatiquement un bloc qui n'est plus utilisé (*Garbage collecting*). Les *capacités* ou plus précisément le calcul de capacités, proposé par Karl Crazy, David Walker et Greg Morriset [40], ajoute aux régions des informations sur les droits d'accès ou de modification des zones de la mémoire. Une des premières implémentations de cette approche est faite dans le langage *Cyclone* [58]. Ce langage est en fait une extension du langage C, dans laquelle l'utilisateur indique explicitement les régions des pointeurs du programme. Le contrôle des régions et des capacités forme un système de typage statique, qui garantit que la mémoire est utilisée correctement.

Un tel système de avec types régions a été utilisé, pour la preuve de programme, par Thierry Hubert et Claude Marché [61] dans le greffon *Jessie*. Les régions sont utilisées dans le calcul de plus faible précondition, où chaque région donne lieu à une variable logique distincte. Cette façon de procéder permet à la génération d'obligations de preuve d'incorporer les informations de séparation. Notons que dans ce cadre, l'utilisateur n'a pas à annoter le programme avec des régions, celles-ci sont inférées automatiquement. Cette approche n'a pas été utilisée pour gérer les invariants de types.

Arthur Charguéraud et François Pottier proposent un système de type fondé sur ces notions de capacités et de régions, pour un langage de programmation avec effets de bord et de l'ordre supérieur [37]. Leur typage permet de prouver de tels programmes par traduction vers des programmes purement fonctionnels. Romain Bardou propose un langage avec un typage également basé sur les régions et les capacités [16], avec pour objectif la gestion des invariants de données. Dans les deux cas, les régions et les capacités doivent être explicitement indiquées dans les programmes. Plus encore, l'évolution des capacités au cours de l'exécution d'un code doit être guidée par l'utilisateur à l'aide d'instructions d'ouverture, de fermeture, d'adoption, etc.

Exemple 1.3.8 *Considérons, dans le langage de Bardou, la classe* `Pair` *qui contient des champs entiers* `fst` *et* `snd` *et un invariant de données.*

```
class Pair {
  fst: int;
  snd: int;
  invariant fst < snd;
}
```

La fonction `incrPair` *incrémente les deux champs d'un pointeur sur un type structure* `Pair`*. Cette fonction prend en paramètre le pointeur* `p` *dont les champs sont incrémentés, mais aussi la région* `r` *à laquelle appartient* `p`*.*

```
fun incrPair [r: Pair] (p: [r]): unit
  consumes r
  produces r
{ unpack p;
  p.fst <— p.fst + 1;
  p.snd <— p.snd + 1;
  pack p;
}
```

De plus, le contrat indique que la région est consommée en entrée et restituée en sortie. Les instructions ***unpack*** *et* ***pack*** *indiquent l'ouverture et la fermeture de la région. L'instruction* ***pack*** *marque que l'invariant doit être rétabli.*

On voit que dans ce type d'approche, on retrouve des aspects analogues à la logique de séparation : consommation et production de régions mémoire ; et des aspects analogues à l'approche *Ownership* :

avec ouverture et fermeture des objets, les invariants devant être vérifiés à la fermeture. Notons que contrairement à la logique de séparation, la gestion d'alias dans un système utilisant des régions se fait au moment du typage et non pas lors de la génération des obligations de preuves. En effet, pour qu'un programme soit bien typé, il ne doit pas partager des références entre différentes régions.

Du fait de la quantité importante d'annotations que l'utilisateur doit ajouter, ces approches s'avèrent fastidieuses.

1.3.5 Les dynamic frames

L'approche par *dynamic frames*, introduite par Ioannis T.Kassios [64], est encore une autre méthode qui permet la gestion de l'utilisation de la mémoire par une fonction. Dans cette approche, le langage de spécification permet de manipuler des ensembles d'emplacements mémoire. La partie de la mémoire à laquelle une fonction accède est typiquement explicitée dans une variable de spécification (la *dynamic frame* de la fonction). Cette méthode permet ainsi d'abstraire les effets de bord d'une fonction et de gérer les alias. Toutes les propriétés liées à l'aliasing sont exprimées par des contraintes d'appartenance aux *frames*, et se retrouvent naturellement dans les obligations de preuve.

Un système de vérification automatique fondé sur la notion de dynamic frames est *Dafny* [68]. Les obligations de preuve générées sont soumises au prouveur automatique Z3 [47]. Le succès de Dafny repose sur la qualité de l'axiomatisation des *frames* fournie à Z3.

Dans la mesure où l'utilisateur doit explicitement indiquer la manière dont les *frames* évoluent au cours de l'exécution, cette approche reste fastidieuse pour l'utilisateur.

Jan Smans, Bart Jacobs et Frank Piessens ont introduit la notion d'*implicit dynamic frames* [91], pour unifier les approches *dynamic frames* et la logique de séparation. Cette notion est utilisée dans l'outil VeriFast, mentionné précédemment. D'une certaine façon, la technique d'*implicit dynamic frames* peut être vue comme une manière de traduire les spécifications originales, écrites en logique de séparation, vers un langage logique plus standard, où la séparation est explicitée par des formules d'appartenance aux *frames*. Le point positif des *implicit dynamic frames* est d'apporter plus d'automatisation à l'approche par logique de séparation. Le point négatif est que, comme pour la logique de séparation, l'utilisateur doit spécifier dans un langage moins naturel que la logique du premier ordre standard.

1.4 Contributions de cette thèse

Le but de ce travail est de spécifier et prouver des programmes avec pointeurs, tels que des programmes C, en utilisant des techniques de raffinement. L'approche proposée permet de faire un compromis entre les techniques complexes qui existent dans la littérature et ce qui est utilisable dans l'industrie, en conciliant légèreté des annotations et restrictions sur les alias.

Nous définissons en premier lieu un langage d'étude, qui s'inspire du langage C, et dans lequel le seul type de données mutable possible est le type des structures, auquel on accède uniquement à travers des pointeurs. De manière non classique, nous incluons, dans le langage de programmation, un langage d'annotation pour spécifier formellement les comportements fonctionnels attendus.

Nous définissons, sur ce langage, une sémantique opérationnelle pour la partie "programmation" du langage, une sémantique dénotationnelle pour la partie logique du langage, un système de typage statique et enfin un mécanisme de génération d'obligations de preuve en proposant un calcul de plus

faible précondition incorporant du raffinement. Nous prouvons ensuite la correction de ce mécanisme de génération d'obligations de preuve par une méthode originale, fondée sur la notion de *sémantique bloquante*, qui s'apparente à une preuve *type soundness* et qui consiste donc, à prouver la préservation puis le progrès de ce calcul.

Nous avons publié aux *JFLA 2013* une formalisation d'un sous-ensemble de ce langage, dans l'outil Why3. Un rapport de recherche [73] (plus détaillé que l'article) décrivant cette formalisation a également été rédigé.

En deuxième lieu, afin de structurer nos programmes, nous munissons notre langage d'une notion de *module*, où un module permet de définir un type de structure et contient un certain nombre de fonctions associées à une même fonctionnalité. Ceci nous permet d'écrire des programmes structurés en composants. Nous étendons également notre langage avec des concepts issus de la théorie du raffinement tels que les variables abstraites que nous formalisons par des champs modèle, et les invariants de collage. Un type structure est alors défini par des champs concrets, des champs modèle et un invariant de données que toute instance de ce type doit vérifier. L'introduction des invariants de données dans notre langage soulève des problématiques liées au partage de pointeurs. En effet, en cas d'alias, on risque de ne plus pouvoir garantir la validité de l'invariant de données d'un pointeur. Nous interdisons alors l'aliasing (le partage de référence) dans notre langage. Nous obtenons certes un langage moins expressif, mais ce n'est pas très restrictif car le partage de référence n'est pas si souvent utilisé dans la programmation. En plus, et tout aussi important, cette hypothèse va dans le sens de la philosophie adoptée par Why, c'est à dire, d'interdire l'aliasing. D'autre part, cette contrainte permet de simplifier les obligations de preuves générées, facilitant ainsi la tâche des prouveurs automatiques. Ainsi, pour contrôler les accès à la mémoire, nous définissons un nouveau système de type, basé sur la notion de régions. Cette contribution s'inspire de la théorie du raffinement et a pour but de rendre les programmes les plus modulaires possibles et leurs preuves les plus automatiques possibles.

Nous étendons à nouveau notre langage, en levant, partiellement, la restriction liée au partage de références. Nous permettons, notamment, le partage de références lorsqu'aucun invariant de données n'est associé au type structure référencé. De plus, nous introduisons le type des tableaux, ainsi que les variables globales et l'affectation qui ne font pas partie du langage noyau.

Pour chacune des extensions citées ci-dessus, nous étendons la définition et la preuve de correction du calcul de plus faible précondition en conséquence.

Nous proposons enfin, une implantation de cette approche sous forme d'un greffon de *Frama-C*. Cette implantation permet de transformer un programme structuré en modules et qui contient des champs modèle et des invariants de données en un ou plusieurs programmes sans champs modèle, ni invariant, afin de pouvoir les passer à d'autres greffons tels que Jessie et WP. Nous expérimentons notre implantation sur des exemples de modules implémentant des structures de données complexes, en particulier des défis issus du challenge vacid0[69], à savoir les tableaux creux (*Sparse Array*) et les tas binaires. Ce deuxième cas d'étude a, dans un premier temps, été traité dans l'environnement de vérification Why3 où les programmes sont écrits dans le langage de programmation *WhyML* et les spécifications dans la logique de Why3. Cette solution est présentée dans le rapport [94].

En résumé, cette thèse s'organise de la manière suivante :

Nous commençons par présenter de manière informelle, dans le chapitre 2, la démarche que nous proposons en l'illustrant avec un premier exemple de structure de données simple, puis avec un exemple de structure de données plus complexe qui contient des tableaux.

Dans le chapitre 3, nous fournissons une présentation formelle de notre approche. Nous définissons

alors un langage d'étude, dont nous présentons la syntaxe, les sémantiques opérationnelle et dénotationnelle associée à chaque partie du langage, ainsi que le système de type. Nous proposons ensuite, un mécanisme de génération d'obligation de preuve dont nous prouvons la correction.

Dans le chapitre 4, nous étendons notre langage en introduisant les notions de *module*, de *champ modèle* et d'*invariant de données*. Ces extensions sont rajoutées dans la syntaxe puis dans la sémantique. Nous définissons ensuite un système de type avec régions, pour contrôler les accès à la mémoire et interdire l'aliasing. Nous étendons enfin le calcul de plus faible précondition ainsi que sa preuve de correction, en conséquence.

Le chapitre 5 permet de décrire de nouvelles extensions : la déclaration de variables globales, l'affectation de ce type de variables et introduction des tableaux de scalaires dans un premier temps, puis des tableaux de structures de données. Comme pour le chapitre 4, nous modifions la syntaxe, la sémantique, le typage et enfin le calcul de plus faible précondition.

Nous présentons dans le chapitre 6 deux études de cas tirés du challenge international Vacid0 [69], qui portent sur des structures de données complexes, à savoir les tableaux creux et les tas binaires. Nous prouvons chacun de ces programmes en utilisant le greffon *Frama-C* que nous avons développé.

Enfin, le chapitre 7 est une conclusion générale de cette thèse. Nous y décrivons les conclusions liés à ces quatre années de travail puis nous présentons quelques perspectives.

Chapitre 2

Présentation informelle de notre approche

Sommaire

Nous présentons, dans ce chapitre, les fondements de l'approche que nous proposons, dans laquelle les programmes sont structurés en modules où chaque module possède une interface publique et une implémentation privée.

La notion de module est essentielle : tous les types de données et les fonctions du programme appartiennent à un module. Un type de données défini dans un module $\mathcal{M}$ est muni d'un ou de plusieurs champs modèle qui ont forcément un type purement logique et qui sont déclarés dans l'interface de $\mathcal{M}$. En effet, les champs modèle permettent d'abstraire un état, ils sont, par conséquent, indépendants de toute implémentation du type auquel ils appartiennent, et peuvent donc apparaître dans l'interface du module. Au contraire les champs concrets ne sont visibles que dans l'implémentation du module dans lequel est défini le type auquel ils appartiennent. Une autre conséquence de cette propriété des champs modèle, est que les contrats des fonctions sont exprimés en utilisant des champs modèle. Par ailleurs, un invariant de collage est associé à un type de structures dans l'implémentation. L'invariant de collage est un prédicat qui permet de relier les champs modèle et les champs concrets de tels types. L'intérêt d'une telle approche est que la correction de l'implémentation d'une fonction est prouvée par des obligations de preuve de raffinement.

Nous illustrons, dans ce qui suit, cette approche de manière informelle sur des exemples utilisant des structures de données complexes : le calculateur de Morgan déjà présenté dans la section 1.1.3 et les structures de pile qui présentent l'intérêt d'utiliser des tableaux.

2.1 Définition de modules

Un programme décrit dans notre langage d'étude est structuré en composants qui sont soit des *modules*, soit des *théories*. La différence est que, contrairement aux modules, les théories ne peuvent contenir que des déclarations logiques.

Un module permet de définir un type structure, ainsi que les fonctions associées à ce type. Si on considère, par exemple, le calculateur de Morgan présenté dans la section 1.1.3, les fonctions associées à ce type sont la fonction `add` qui permet de rajouter un élément au calculateur et la fonction `mean` qui calcule la moyenne de ses éléments.

Un type structure est défini par son nom et un ensemble de champs. Le type structure permettant de formaliser un calculateur de Morgan est défini alors avec un champ modèle `values` qui représente le multi-ensemble des réels collectés et deux champs concrets `sum` et `count`, pour faire l'analogie avec l'approche par raffinement. Car il est bien évidemment possible d'implémenter un tel calculateur en utilisant d'autres champs.

Dans un module, les fonctions sont annotées avec des contrats qui spécifient les pré (**requires**) et post-condition (**ensures**) de la fonction, ainsi que les emplacements mémoire fraîchement alloués (**allocates**), accessibles (**reads**) et potentiellement modifiables (**writes**) par le corps de cette fonction. Les contrats des fonctions d'un module appartiennent à son interface, c'est-à-dire la partie du module qui est visible par les autres modules. Ils ne doivent, par conséquent, référencer que des champs modèle. Pour illustrer ce trait de notre langage, nous faisons correspondre, à chacune des opérations `add` et `mean` déclarées dans la machine abstraite du calculateur de Morgan, des fonctions de même nom dans le module `Morgan`. Nous annotons ces fonctions avec des contrats dont les pré et les post-conditions sont celles spécifiées dans la section 1.1.3.

Contrairement à l'approche par raffinement, notre approche offre la possibilité de créer dynamiquement des instances d'un type structure, alors que dans le raffinement il n'y qu'une seule instance de la structure (la machine) avec une opération d'initialisation. Concrètement, nous définissons une fonction `create_Calc` qui permet de créer une nouvelle instance de type `Calc`, d'initialiser ses champs et de la retourner. La clause **allocates** et la post-condition spécifient donc que le résultat de la fonction est fraîchement alloué et que son champ modèle est vide. Une telle formalisation du calculateur de Morgan, décrite dans notre langage d'étude, est détaillée dans la figure 2.1.

Notez, d'une part, que, par rapport à la méthode B, les fonctions `add` et `mean` ont un paramètre supplémentaire `this` qui est un pointeur sur une structure `Calc`. Nous spécifions, dans la clause **writes** de la fonction `add` que les emplacements mémoire accessibles à partir de `this` sont potentiellement modifiables par cette fonction. Lorsque une clause n'est pas explicitement mentionnée dans le contrat d'une fonction, alors une valeur par défaut lui est affectée. Par exemple, l'absence de la clause **writes** est interprété par l'absence d'effet de bord, ce qui équivalent à **writes** $\backslash nothing$.

D'autre part, l'invariant de collage est un prédicat à un argument `Inv_Calc`, déclaré en même temps que le type.

Enfin, comme dans l'approche par raffinement, nous utilisons un type algébrique `bag` pour représenter le multi-ensemble des réels enregistrés dans le calculateur, c'est-à-dire, le champs modèle `values`. Or dans l'approche que nous proposons, les types logiques sont définis dans des *théories*. Il est alors nécessaire d'importer la théorie `Bag` dans laquelle le type `bag` est défini, à l'aide de la directive `use_T`.

```
module Morgan

 use_T Bag;

 struct Calc{
   real sum;
   int count;
   model bag values;
 } Invariant Inv_Calc(struct Calc this) =
     this->sum == sumbag(this->values) && this->count == card(this->values);

 function struct Calc create_Calc()
   allocates \result;
   ensures \result->values == empty_bag;
 { let this = new Calc in
     this->sum = 0.0;
     this->count = 0;
     this
 }

 function unit add(struct Calc this, real x)
   writes *this;
   ensures this->values == union(\old(this->values),singleton(x));
 { this->sum = this->sum + x;
   this->count = this->count + 1
 }

 function real mean(struct Calc this)
   requires this->values != empty_bag;
   reads *this;
   ensures \result == sumbag(this->values)/card(this->values);
 { this->sum/this->count }

end
```

FIGURE 2.1 – Le module `Morgan`.

2.2 Définition de théories

Les théories permettent, non seulement, de déclarer ou de définir des types logiques, mais également de définir des propriétés sur ces types en posant des axiomes, des lemmes ou des prédicats. Pour illustrer l'utilisation de tels concepts, nous présentons la théorie `Bag` dont la signature est détaillée dans la figure 2.2.

Dans cette théorie, le type `bag` permettant de représenter les multi-ensembles de réels, n'est pas interprété. Il est abstrait et est caractérisé par la fonction `nb_occ` qui associe à chaque élément `x`, dans un multi-ensemble `b`, son nombre d'occurrences ou sa multiplicité. On dit que le nombre d'occurrences caractérise les multi-ensembles dans le sens ou deux multi-ensembles ayant la même multiplicité pour chaque élément, sont égaux.

```
theory Bag
 type bag;
 logic int nb_occ(real x, bag b);
 logic bag empty_bag;
 logic bag singleton (real x);
 logic bag Union (bag a, bag b);
 axiom occ_empty : \forall real x; nb_occ(x, empty_bag) == 0;

 axiom occ_singleton :
   \forall real x, y;
      (x == y && nb_occ(y, singleton(x)) == 1) ||
       (x != y && nb_occ(y, singleton(x)) == 0);

 axiom occ_union: \forall real x, bag a, b;
                   nb_occ(x, Union(a, b)) == nb_occ(x, a) + nb_occ(x, b);

 predicate eq_bag (bag a, bag b) =
     \forall int x; nb_occ(x, a) == nb_occ (x, b);

 axiom bag_extensionality:
     \forall bag a, b; eq_bag(a, b) ==> a == b;

 lemma occ_singleton_eq : \forall real x, y;
                   x == y ==> nb_occ(y, singleton(x)) == 1;
 lemma occ_singleton_neq : \forall real x, y;
                   x != y ==> nb_occ(y, singleton(x)) == 0;
 lemma Union_comm : \forall bag a, b; Union(a, b) == Union(b, a);

 lemma Union_assoc :
  \forall bag a, b, c; Union(a, Union(b, c)) == Union(Union(a, b), c);
...
end
```

FIGURE 2.2 – Théorie des multi-ensembles (`Bag`).

Nous déclarons ensuite les constructeurs standards : `empty_bag`, `singleton` et `union` des multi-ensembles, qu'on axiomatise en termes de multiplicité. Des propriétés classiques telles que l'associativité et la commutativité de l'union sont également énoncées via des lemmes.

Par ailleurs, la spécification du calculateur de Morgan utilise la somme des éléments d'un multi-ensembles et sa cardinalité. Nous introduisons alors deux fonctions `sumbag` et `card`, que nous axiomatisons comme on peut le voir dans la figure 2.3. Une telle axiomatisation implique que l'on considère des multi-ensembles finis.

```
logic real sumbag (bag a);
axiom SumBag_empty : sumbag (empty_bag) == 0.0;
axiom SumBag_singl : \forall real x; sumbag (singleton (x)) == x;
axiom SumBag_union : \forall bag a, b;
  sumbag (union (a, b)) ==  sumbag (a) + sumbag (b);

logic int card (bag a);
axiom Card_empty : card(empty_bag) == 0;
axiom Card_singleton : \forall real x; card(singleton(x)) == 1;
axiom Card_union : \forall bag x, y; card(Union(x, y)) == card(x) + card(y);

lemma card_zero_empty : \forall bag b; card(b) == 0 ==> b == empty_bag;
```

FIGURE 2.3 – Axiomatisation des fonctions `sumbag` et `card` dans la théorie `Bag`.

2.3 Obligations de preuve

Une fois les programmes annotés, des formules logiques qu'on appelle *obligations de preuve*, et dont la validité implique la correction du programme, sont engendrées. Ainsi, prouver l'implémentation d'un module revient à prouver que les obligations de preuve générées à partir de sa spécification sont valides. Or spécifier un module consiste à annoter chacune de ses fonctions avec un contrat décrivant son comportement. Par conséquent, il faut prouver, pour chaque fonction, que si la précondition de la fonction est vraie dans l'état initial, alors la post-condition est vérifiée dans l'état final. En plus, il est nécessaire de prouver que les invariants de collage des types des pointeurs accessibles par la fonction sont vérifiés au début et à la fin de la fonction. On peut alors voir ceci comme une extension du contrat de la fonction, dans lequel on rajoute l'invariant de collage de chaque paramètre et résultat de type structure, en pré et en post-condition de la fonction. En pratique, nous ne rajoutons pas l'invariant de collage associé au type structure S, mais nous définissons un prédicat $Valid\_S$ qui établit qu'un pointeur sur une instance de type structure est valide, c'est-à-dire que l'invariant est vérifié pour cette instance et que le pointeur est bien alloué en mémoire. L'intérêt d'un tel prédicat sera démontré un peu plus loin dans la section. Par ailleurs, tous les champs de type structure sont visibles dans l'implémentation, si ce type appartient au module implémenté, c'est-à-dire que le type structure est défini par ces champs concrets et ces champs modèle. Ainsi, lorsque nous spécifions dans la clause **reads**, respectivement **writes**, que tous les emplacements mémoire accessibles à partir d'un pointeur sur une structure de données, sont visibles, respectivement potentiellement modifiables, par le corps de la fonction, alors c'est interprété, dans l'implémentation, comme tous les champs de la structure, concrets et modèle, le sont.

Le résultat de l'application d'une telle approche sur le module `Morgan`, est décrit dans la figure 2.4. Notez que les clauses qui n'ont pas été spécifiées dans les contrats initiaux des fonctions, sont rajoutées dans les nouveaux contrats avec des valeurs par défaut qui sont $\backslash nothing$ pour les emplacements

```
module Morgan

struct Calc {
  real sum;
  int count;
  bag values;
}

predicate Inv_Calc(struct Calc this) =
  this->sum == sumbag(this->values) && this->count == card(this->values);

predicate Valid_Calc(struct Calc calc) = valid(calc) && Inv_Calc(calc);

function struct Calc create_Calc()
  requires \true;
  reads \nothing;
  writes \nothing;
  allocates \result;
  ensures \result->values == empty_bag;
  ensures Valid_Calc(\result);
{ ... }

function unit add(struct Calc this, real x)
  requires Valid_Calc(this);
  reads this->sum, this->count, this->values;
  writes  this->sum, this->count, this->values;
  allocates \nothing;
  ensures this->values == union(\old(this->values),singleton(x));
  ensures Valid_Calc(this);
{ ... }

function real mean(struct Calc this)
  requires Valid_Calc(this);
  requires this->values != empty_bag;
  reads this->sum, this->count, this->values;
  writes \nothing;
  allocates \nothing;
  ensures \result == sumbag(this->values)/card(this->values);
{ ... }
end
```

FIGURE 2.4 – Spécification "étendue" du module `Morgan`.

mémoire, c'est-à-dire les clauses **reads**, **writes** et **allocates**, et $\backslash true$ pour les prédicats, c'est-à-dire les clauses **requires** et **ensures**.

Le cas de la fonction `mean` est intéressant, dans le sens où si on devait appliquer les transformations citées ci-dessus, il faudrait vérifier dans la pré et la post-condition que l'invariant de `this` est vérifié. Or, la clause **writes** spécifie qu'aucun emplacement mémoire n'est modifié par cette fonction. Par consé-

quent, si l'invariant de `this` est vrai à l'état initial de la fonction `mean` et que cette dernière ne modifie aucun emplacement mémoire, alors cet invariant est toujours vrai à l'état final de la fonction. De manière générale, seuls les invariants de données des pointeurs cités dans la clause **writes** peuvent être rompus lors de l'exécution du corps de la fonction. Nous proposons, alors une solution qui consiste à ne vérifier l'invariant d'un emplacement mémoire est établi, que si ce dernier est spécifié dans la clause **writes**. Les invariants des autres emplacements mémoire le sont par construction. Il faut néanmoins faire attention en cas d'alias. Nous avons, en effet, montré dans la section 1.3 qu'en présence d'alias, une telle hypothèse n'est pas suffisante. Nous traiterons ce point dans la section 2.4.

En outre, la post-condition de la fonction `add` indique que la valeur du champ modèle `values` est modifiée par la fonction. Or les champs modèle ne peuvent pas être affectés dans le code. Nous nous assurons alors, avant l'exécution de l'instruction `return`, qu'il existe des valeurs pour le champ `values` qui établissent la post-condition et l'invariant de collage. Formellement, nous vérifions que :

$$\begin{aligned}\exists\, values',\ this \to sum = sumbag(values') \wedge \\ this \to count = card(values') \wedge \\ values' = union(this \to values,\ singleton(x))\end{aligned}$$

Le champ `values` est alors mis à jour avec une des valeurs vérifiant la condition précédente. Ce qui nous permet de prouver la post-condition de la fonction. La manière dont le champ est modifié est expliquée dans le chapitre 3.

Considérons maintenant une fonction dans un module $\mathcal{M}_1$ qui appelle une fonction d'un autre module $\mathcal{M}_2$. Pour prouver qu'un tel programme est correct, il n'est pas nécessaire de connaître les détails d'implémentation du module $\mathcal{M}_2$, il suffit de connaître le contrat de la fonction appelée. Autrement dit, seule l'interface du module $\mathcal{M}_2$ est nécessaire.

Nous proposons dans ce cas, une solution inspirée de l'approche par raffinement. L'idée est que l'invariant de collage d'un type structure défini dans un module est privé. Il n'est, par conséquent pas visible à partir de l'extérieur du module. Ainsi, la preuve de préservation de l'invariant est propre au module auquel appartient le type structure auquel il est associé. La fonction appelante n'a alors pas besoin de le vérifier.

Supposons que nous avons un programme client qui implémente un cours dans lequel est inscrit un certain nombre d'étudiants. On aimerait alors insérer l'ensemble des notes des étudiants inscrits et calculer la moyenne des notes obtenues. Un moyen de réaliser ce type de programme est de considérer qu'un cours est un calculateur de Morgan comme c'est le cas dans la figure 2.5. Ce programme permet d'insérer quatre notes dans un cours `course` et de vérifier que la moyenne de ces notes est bien égale à 11.25.

Pour prouver ce programme, il suffit de connaître les contrats des fonctions `create_Calc`, `add` et `mean`, qui contiennent assez d'informations pour prouver l'assertion dans la figure 2.5.

En effet, le champ `values` de l'instance `course` du calculateur est vide après l'appel de la fonction `create_Calc`. Il est ensuite modifié à chaque appel de la fonction `add`. Après le dernier appel, le champ `values` est égal à $\{12, 9, 10, 14\}$. La fonction `mean` peut alors être invoquée et elle retourne le résultat de la somme des éléments de `values` divisée par sa cardinalité, c'est-à-dire 11.25.

Notez ici qu'à aucun moment nous n'avons utilisé les champs `sum` et `count`. Du point de vue du client, la fonction `add` ne modifie que le champ modèle `values`. Autrement dit, la fonction `add`, n'accède qu'au champ modèle du type `Calc`. En généralisant cette propriété, on dit que dans l'interface d'un module, seuls les champs modèle appartenant au type structure du pointeur spécifié dans les clauses

```
function unit client(){

  struct Calc course = create_Calc();

  add(course, 12.0);
  add(course, 9.0);
  add(course, 10.0);
  add(course, 14.0);
  real m = mean(course);
  //@ assert m == 11.25;
}
```

FIGURE 2.5 – Programme client du module `Morgan`.

```
module Morgan

struct Calc {
  bag values;
}

predicate Valid_Calc(struct Calc calc) = valid(calc);

function struct Calc create_Calc()
  requires \true;
  reads \nothing;
  writes \nothing;
  allocates \result;
  ensures \result->values == empty_bag;
  ensures Valid_Calc(\result);

function unit add(struct Calc this, real x)
  requires Valid_Calc(this);
  reads this->values;
  writes this->values;
  allocates \nothing;
  ensures this->values == union(\old(this->values),singleton(x));
  ensures Valid_Calc(this);

function real mean(struct Calc this)
  requires Valid_Calc(this);
  requires this->values != empty_bag;
  reads this->values;
  writes \nothing;
  allocates \nothing;
  ensures \result == sumbag(this->values)/card(this->values);
 end
```

FIGURE 2.6 – Interface du module `Morgan`.

reads et **writes** sont visibles ou potentiellement modifiables par la fonction. De plus, même si nous n'avons pas besoin de vérifier les invariants des objets au début et à la fin des fonctions, nous rajoutons tout de même $Valid\_Calc$ en pré et en post-condition de chaque fonction. La différence, par rapport au cas où nous prouvons l'implémentation, est que ici ce prédicat établit que le pointeur est valide (alloué en mémoire), mais à aucun moment nous ne vérifions l'invariant des paramètres des fonctions.

L'interface du module `Morgan` est détaillée dans la figure 2.6.

2.4 Contrôle des alias et correction de l'approche

```
module Course
  struct Course{
    struct Calc notes;
    model real mean;
  } Invariant Inv_Course(struct Course this) =
        this−>mean = mean(this−>notes);

  function struct Course create_Course(struct Calc c){
    let this = new Course
    in this−>notes = c;
    this
  }
end

function unit client() {
  struct Calc calc     = create_Calc();
  struct Course course = create_Course(calc);
  add(calc, 12.0); //violation de l'invariant de course
}
```

FIGURE 2.7 – Violation de l'invariant en présence de partage.

Nous avons dit dans la section précédente que si l'invariant de données d'un pointeur est établi au début d'une fonction qui ne modifie aucun emplacement mémoire accessible à partir de ce pointeur, alors l'invariant est toujours établi à la fin de la fonction. Cette hypothèse n'est pas vraie dans deux cas : soit on modifie un des champs du type structure dans une fonction extérieure au module, soit le pointeur est partagé. Pour le premier cas, nous interdisons toute modification de champ d'un type structure si ce dernier n'appartient pas au module courant. Cette contrainte est une conséquence du fait que les champs concrets ne sont pas visibles à l'extérieur du module auquel appartient le type dans lequel ils sont déclarés.

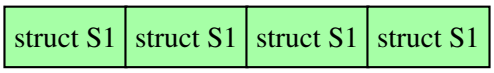
Le deuxième cas de figure est un peu plus compliqué. Modifions le programme client de la section précédente (figure 2.5) en définissant un type structure `Course` qui contient un champ de type `Calc`. Nous obtenons alors le programme décrit dans la figure 2.7, dans lequel nous avons un partage entre `calc` et `course->notes` où `course` est une instance de type `Course`. Dans ce cas, même si l'invariant de `course` est vrai après l'appel de la fonction `create_Course` et que `course` n'est pas modifié, rien ne nous garantit que l'invariant de `course` est toujours vrai à la fin de `client`, car `course` a pu être modifié via `course->notes` et donc l'invariant rompu. En l'occurrence, c'est le cas ici. Car rajouter un élément dans `calc` revient à modifier `course->notes`. L'invariant de `course` n'est alors plus vérifié.

Ainsi, pour nous assurer que ce cas de figure ne se produise pas, nous interdisons le partage de référence sur les types structure avec des invariants. Concrètement, ce programme est mal typé dans l'approche que nous proposons. Cette restriction nous permet de contrôler les accès aux emplacements mémoire et par conséquent de garantir la correction de l'approche que nous proposons.

2.5 Extension avec des types tableaux

Dans la présentation que nous avons faite de notre approche jusque là, le seul type de données mutable que nous avons utilisé est celui des pointeurs sur structures. Ceci nous a permis de traiter la correction de programmes utilisant des types de structure simples tels que le calculateur de Morgan. Un type de structure simple est un type dont les champs des structures sont soit des types de base : **int**, **real**, `bool`, soit des pointeurs sur des types structure. Nous allons, dans ce qui suit, étendre notre approche pour traiter les types des tableaux.

Nous représentons, d'une part, un tableau de structures, de taille n, par un pointeur sur un bloc mémoire qui contient n structures stockées de manière contiguë. Si, par exemple, nous avons un tableau dont les éléments sont de type structure S, alors sa représentation en mémoire est :



Un pointeur sur une structure est alors considéré comme un tableau à un élément. L'accès à un champ de structure est dans ce cas paramétré par la position de la structure dans le tableau auquel elle appartient : si e est un tableau de structures S, alors l'accès au champ f du $i^{\text{ème}}$ élément du tableau e s'écrit : $e + i \rightarrow f$. La notation $(e + i)$ correspond à l'arithmétique des pointeurs telle qu'elle existe dans le langage C.

D'autre part, les tableaux de scalaires sont interprétés comme des tableaux de structures prédéfinies : `Int` pour les entiers, `Real` pour les réels et `Bool` pour les booléens. Chacun de ces types contient un champ `value` qui contient la valeur du scalaire correspondant. Par exemple, le champ `value` du pointeur sur `Int` qui est associé à l'entier 4 contient la valeur 4.

Considérons les structures de piles. Une pile est un type de données qui vérifie la propriété que, si un objet `a` est inséré dans la pile (empilé) avant l'objet `b`, alors `a` ne peut être enlevé de la pile (dépilé) qu'après `b`. Une pile contient donc une suite d'éléments, et peut donc être implémentée par un tableau qui contient ces éléments. Nous définissons alors un type structure `Stack` par un tableau d'entiers `a`, une taille `size` qui est la taille du tableau, et une capacité maximale `cap`. On abstrait ensuite l'état d'une pile en utilisant deux champs modèle : `elements` et `capacity` qui représentent respectivement la liste des éléments de la pile et le nombre maximum d'éléments que peut contenir la pile. On considère seulement les piles qui ont une capacité fixée, c'est-à-dire qui peuvent contenir un nombre maximum d'éléments constant. Cette contrainte permet de définir des piles sans avoir à gérer l'agrandissement dynamique des tableaux.

Un invariant de collage est associé au type `Stack`. Il définit la relation entre les champs modèle et les champs concrets : la valeur du champ concret `cap` est égale à celle du champ modèle `capacity` et le $i^{\text{ème}}$ élément de la liste `elements` est stocké à l'indice `(size - i)` dans le tableau `a` (Voir figure 2.8).

Par ailleurs, le type du champ modèle `elements` est un type logique. Nous définissons alors, dans la figure 2.9, une théorie dans laquelle nous déclarons le type `list` et un ensemble de fonctions logiques

```
struct Stack{
  int a[];
  int cap;
  int size;
  model list elements;
  model int capacity;
}Invariant Inv_Stack(struct Stack this) =
   0 <= this->size && this->size <= this->cap &&
   this->capacity == this->cap && card(this->elements) == this->size &&
   \forall int i; 0 < i && i <= this->size ==>
            nth(this->elements, i) == ((this->a)+(this->size - i))->value;
```

FIGURE 2.8 – Définition du type structure `Stack`.

qui permettent de manipuler des instances de type `list`.

```
theory List

  type list;
  logic list nil;
  logic list cons (int x, list l);   /* constructeur de liste */
  logic int hd(list l);              /* Tête de la liste */
  logic list tl(list l);             /* Queue de la liste */
  logic int card (list l);           /* Taille de la liste */
  logic int nth(list l, int i);      /* i ème élément de la liste */

end
```

FIGURE 2.9 – Déclaration du type logique `list`.

Les seules opérations qu'il est possible d'appliquer sur les piles sont l'empilement et le dépilement. Nous définissons alors, dans le module `Stack`, deux fonctions `push` et `pop`, dont les spécifications sont présentées dans la figure 2.10.

La fonction `push` ajoute un nouvel élément dans le tableau à l'indice `size`. Autrement dit, à l'élément qui se trouve à l'emplacement mémoire accessible en effectuant un décalage de `size`, à partir de l'emplacement mémoire pointé par `a`.

Nous définissons, également, la fonction `create_Stack` qui permet d'allouer une nouvelle instance de type `Stack` et plus particulièrement, le tableau d'entiers `this` $\rightarrow$ `a`.

Pour prouver le module dans lequel le type structure `Stack` et les fonctions `push`, `pop` et `create_Stack` sont définis, nous étendons les pré et les post-conditions avec les invariants des types structures utilisés par les fonctions, c'est-à-dire, Inv_Stack.

2.6 Implémentation

Afin d'expliquer la manière dont nous implémentons notre approche, nous écrivons le module `Calc` présenté dans la figure 2.1, dans le langage C et nous l'annotons avec le langage de spécification ACSL

```
function struct Stack create_Stack(int c)
  requires c >= 0;
  writes \nothing;
  allocates \result;
  ensures \result−>elements == nil && \result−>capacity == c;
{ let this = new Stack in
    this−>a = new int[c];
    this−>size = 0;
    this−>cap = c;
    this
}

function unit push(struct Stack s, int v)
  requires card(s−>elements) < s−>capacity;
  reads ∗s;
  writes ∗s;
  ensures s−>elements == cons (v, \old(s−>elements));
  { ((s−>a) + (s−>size))−>value = v;
  s−>size = s−>size + 1
}

function unit pop(struct Stack s)
  requires s−>elements != nil;
  reads ∗s;
  writes ∗s;
  ensures s−>elements == tl(\old(s−>elements));
{ s−>size = s−>size − 1 }
```

FIGURE 2.10 – Présentation des fonctions du module `Stack`.

(figure 2.11), de façon à le prouver en utilisant les plug-in de vérification déductive de *Frama-C* : *WP* et *Jessie*.

Dans la solution *Frama-C*/ACSL, le concept d'*invariant de type* permet de décrire l'invariant de collage, c'est-à-dire, la relation entre les champs du type structure et les champs modèle qui y sont associés. Dans ACSL, un invariant de type associé au type structure `S` est une propriété qui doit être vérifiée à l'entrée et à la sortie de n'importe quelle fonction. Elle s'applique à toutes les variables globales et tous les paramètres formels de type `S`. Si le résultat de la fonction est de type `S`, alors il doit également satisfaire cet invariant à la sortie de fonction.

Par ailleurs, la notion de champ modèle n'existe pas dans le langage C. L'introduction de ce type de champs se fait alors en annotant le type structure avec la construction *model* du langage ACSL. Notez que ces champs n'ont été introduits que dans la dernière version de *Frama-C*, à notre demande.

Aucun de ces deux concepts n'est traité par les greffons de *Frama-C*, notamment *WP* et *Jessie*. Nous procédons alors à une implémentation par traduction qui consiste à transformer les champs modèle et invariants de collage de telle façon que les programmes résultant de cette traduction puissent être traités par les greffons *WP* et *Jessie*. Concrètement, étant donné un programme composé de plusieurs modules $\mathcal{M}_1 \ldots \mathcal{M}_n$, nous traduisons chaque module $\mathcal{M}_i$ en un programme qui peut être prouvé par les greffons *WP* et *Jessie*. De plus, prouver un module $\mathcal{M}_i$ revient à prouver son implémentation en utilisant

```
typedef struct Calc{
  double sum;
  int count;
} *calc;

//@ model struct Calc { bag values };

/*@ type invariant Inv_Calc(calc this) =
  @   this->sum == sumbag(this->values) && this->count == card(this->values)
  @*/

/*@ allocates \result;
  @ ensures \result->values == empty_bag;
  @*/
calc create_Calc() {
  calc this = (calc) malloc(sizeof(struct Calc));
  this->sum = 0.0;
  this->count = 0;
  return this;
}

/*@ assigns *this;
  @ ensures this->values == union(\old(this->values),singleton(x));
  @*/
unit add(calc this, double x){
  this->sum += x;
  this->count += 1;
}

/*@ requires this->values != empty_bag;
  @ assigns \nothing;
  @ ensures \result == sumbag(this->values)/card(this->values);
  @*/
double mean(calc this){ return this->sum / this->count; }
```

FIGURE 2.11 – Calculateur de Morgan en C et annoté en ACSL.

l'interface des autres modules du programme initial.

Nous définissons alors deux types de traduction. La première permet de transformer la spécification du module que nous souhaitons prouver, la seconde permet de transformer les autres modules en générant l'interface de ces derniers à partir des définitions initiales. Ce sont ces traductions qui étendent les pré et post-conditions des contrats de fonctions, et qui transforment les clauses **reads** et **writes**. Par conséquent, si nous voulons prouver le module `Stack`, nous appliquons le premier type de traduction, qu'on appelle *traduction concrète*, et si nous souhaitons prouver un client de ce module, alors nous appliquons le deuxième type de traduction, qu'on appelle *traduction abstraite*.

En pratique, les champs modèle sont transformés en de simples champs de structure et les invariants de collage en prédicats.

2.7 Résumé des caractéristiques et des limitations du langage

Notre langage est essentiellement un sous ensemble du langage C, dans lequel les seuls types de données modifiables sont les types structure et les tableaux. Ce langage ne permet pas l'arithmétique de pointeurs, ni la prise de l'adresse (opérateur &), ni les types unions, ni les cast de pointeurs. Il n'y a pas non plus d'entiers machines, ni de nombres à virgule flottante, mais ces aspects pourraient être ajoutés de manière orthogonale (cf. thèse de Nguyen [82]).

Au dessus de ce langage, nous introduisons les notions de modules, de champs modèles et d'invariants de données.

Ce langage est muni d'un système de typage qui contraint très fortement les possibilités d'aliasing : le partage de pointeurs sur un type structure n'est autorisé que si ce dernier n'est pas muni d'un invariant de données.

Ce langage présente les limitations suivantes qui seront discutées en détail dans la section 7.2.
- Il n'est pas possible de ré-allouer des blocs de mémoire.
- Les deux branches d'une conditionnelle doivent allouer exactement de la même façon.
- On ne peut pas allouer dans le corps d'une boucle et on ne peut pas faire de fonction récursive qui alloue.
- On ne peut pas définir de types récursifs (Par exemple des listes chaînées ou des arbres), sauf si ceux-ci ne sont pas munis d'invariants.

Chapitre 3

Génération d'obligations de preuve pour un langage à pointeurs

Sommaire

Nous allons définir, dans ce chapitre, un mécanisme de génération d'obligations de preuve par un calcul de plus faible précondition sur un langage à pointeurs avec alias. Dans cette optique, nous définissons un langage d'étude dans lequel le seul type, non primitif, est celui des types structures, dont les effets de bord sont limités à la mise à jour des champs de structures. Par ailleurs, ce langage présente, d'une part, la particularité de contenir des annotations dans sa syntaxe. Autrement dit, nous ne distinguons pas le langage de spécification du langage de programmation. D'autre part, il a la caractéristique de ne pas faire de distinction entre les expressions et les instructions. En effet, une instruction peut être considérée comme une expression de type $unit$ comme c'est le cas dans les langages fonctionnels à la ML.

Nous munissons notre langage d'une sémantique opérationnelle, ainsi que d'un système de types afin de déterminer les programmes bien typés.

Enfin, nous définissons un générateur d'obligations de preuve par un calcul de plus faible précondition, basé sur la notion de *sémantique bloquante*. Nous détaillons, pas à pas, la preuve de correction de ce calcul, qui s'inspire de la méthode *type soundness*, en posant un certain nombre de lemmes intermédiaires, nécessaires pour effectuer cette preuve.

Cette approche par sémantique à petits pas bloquante est originale. Nous l'avons formalisée, dans l'outil Why3, sur un langage impératif simple qui, contrairement au langage que nous allons présenter, ne contient pas de type structure, de pointeurs ni d'appel de fonction. Néanmoins, cette formalisation, publiée aux JFLA en 2013 [74], a permis d'identifier les lemmes nécessaires pour la preuve de correction du calcul de plus faible précondition que nous définissons. L'approche de ce chapitre s'inspire largement de ce travail préliminaire (mais elle n'est pas formalisée en Why3).

Dans la suite de ce document, un objet n'est pas une instance de classe comme c'est le cas dans les langages orientés objets, mais une référence (pointeur) vers une instance de structure.

3.1 Syntaxe

$prog$	::=	$decl^*$	
$decl$	::=	$logic\_decl$	Déclaration logique
		$\mid struct\_decl$	Définition de type structure
		$\mid func\_decl$	Définition de fonction
$const\_decl$	::=	**const** $id\ v$	

FIGURE 3.1 – Grammaire du langage d'entrée.

La grammaire décrite dans la figure 3.1 définit la syntaxe de notre langage qui s'inspire du langage C. Un programme écrit dans un tel langage est une séquence de déclarations logiques, de types structures ou de fonctions. La syntaxe des déclarations logiques est décrite dans la figure 3.2. Nous n'autorisons pas de polymorphisme dans le langage logique. Ceci n'est pas contraignant car on voit bien dans les exemples présentés dans le chapitre 2, que les types définis dans les théories sont toujours instanciés.

Les différents constructeurs, introduits dans cette syntaxe, peuvent être définis ou simplement déclarés. Un type logique peut, par exemple, être non interprété, comme c'est le cas du type `bag` introduit dans la section 2.2. De même, il est possible de ne fournir que la signature d'une fonction logique.

Le langage des termes de la logique (figure 3.3) contient les constantes qui peuvent être entières (n), réelles (r), booléennes (**true**, **false**), ou bien *()*, les variables logiques, les opérations unaires et binaires et l'application de fonctions logiques. Nous considérons, dans notre langage, le type des réels mathématiques. Un terme peut également être un accès vers le champ d'une structure. Il est, par conséquent, nécessaire de pouvoir accéder au champ des instances de ce type. Par ailleurs, nous introduisons, dans notre langage, la notion de label, dans le but de spécifier des propriétés reliant différents états de l'exécution. L'accès à un champ $x \to f$ est paramétré par un label L. Les labels sont introduits dans les expressions. Ils permettent de préciser l'état du programme, labelisé ou étiqueté par L, dans lequel on souhaite accéder à la valeur du champ f de x. Si aucun label n'est mentionné, alors on accède à la valeur du champ f dans l'état courant.

$$
\begin{array}{rcl}
logic\_decl & ::= & logic\_type\_decl \quad | \, logic\_type\_def \\
 & & | \, logic\_const\_decl \quad | \, logic\_const\_def \\
 & & | \, logic\_function\_decl \\
 & & | \, logic\_function\_def \\
 & & | \, logic\_predicate\_decl \\
 & & | \, axiom\_decl \\
 & & | \, predicate\_def \\
 & & | \, lemma\_def \\
logic\_type\_decl & ::= & \textbf{type}\ id; \\
logic\_const\_decl & ::= & \textbf{logic}\ type\ id\ ; \\
logic\_predicate\_decl & ::= & \textbf{predicate}\ id\ (params)^{?}\ ; \\
logic\_function\_decl & ::= & \textbf{logic}\ type\ id\ (params)\ ; \\
axiom\_decl & ::= & \textbf{axiom}\ id\ :\ pred\ ; \\
logic\_type\_def & ::= & \textbf{type}\ id = type; \\
logic\_const\_def & ::= & \textbf{logic}\ type\ id\ =\ term\ ; \\
predicate\_def & ::= & \textbf{predicate}\ id\ (params)^{?}\ =\ pred\ ; \\
logic\_function\_def & ::= & \textbf{logic}\ type\ id\ (params)\ =\ term\ ; \\
lemma\_def & ::= & \textbf{lemma}\ id\ :\ pred\ ;
\end{array}
$$

FIGURE 3.2 – Grammaire des déclarations logiques.

$term$	::=	c	Constante
		$\mid x$	Variable logique
		$\mid un\_op\ term$	
		$\mid term\ bin\_op\ term$	
		$\mid term\ rel\_op\ term$	
		$\mid term \rightarrow id$	
		$\mid term \overset{L}{\rightarrow} id$	
		$\mid id\ (term(,term) *)$	Application de fonctions logiques
c	::=	$n \mid r \mid$ **true** $\mid$ **false** $\mid$ **null** $\mid ()$	
bin_op	::=	$+ \mid - \mid *$	Arithmétique des entiers/réels
		$\mid \&\& \mid \Vert$	Opérations booléennes
rel_op	::=	$\mid < \mid <= \mid > \mid >=$	Comparaison des entiers/réels
		$\mid == \mid !=$	Test d'égalité/ de diségalité
un_op	::=	$-$	Moins unaire
		$\mid !$	Négation booléenne

FIGURE 3.3 – Grammaire des termes.

L'abréviation $\backslash\textbf{at}(t, L)$ désigne la valeur du terme t au moment où l'exécution atteint le label L. Plus précisément, c'est une abréviation qui veut dire que le label L est attaché aux sous termes de t qui sont des accès vers un champ et qui ne sont pas labelisés. Si un terme est labelisé par deux labels, on choisit le plus ancien, c'est-à-dire le plus profond dans l'expression. De même l'absence de label correspond au label par défaut *Here* désignant l'état courant. Par exemple,

$$\backslash\textbf{at}(x \rightarrow f + y \overset{K}{\rightarrow} g,\ L) + x \rightarrow f$$

est une abréviation pour

$$x \overset{L}{\rightarrow} f + y \overset{K}{\rightarrow} g + x \overset{Here}{\rightarrow} f$$

La syntaxe des formules logiques (figure 3.4) contient les expressions atomiques (termes booléens), les connecteurs classiques de conjonction, disjonction, négation, implication et équivalence, et les quantifications universelle et existentielle. Nous ajoutons la liaison locale par `let`, pour permettre une défi-

$pred$	::=	t	Formules atomiques
		$\mid id\ (\ term\ (,\ term)^*\ )$	Application de prédicats
		$\mid pred\ log\_op\ pred$	Opérations logiques
		$\mid\ \neg pred$	
		$\mid$ `let` $id = term$ `in` $pred$	
		$\mid$ **forall** $params\ ;\ pred$	
		$\mid$ **exists** $params\ ;\ pred$	
		$\mid$ **valid**$(term,\ id)$	
		$\mid$ **valid**$(term)$	
log_op	::=	$\wedge\ \mid\ \vee\ \mid \Rightarrow \mid \Leftrightarrow$	

FIGURE 3.4 – Grammaire des formules logiques.

nition plus élégante du calcul de plus faible précondition. Dans cette grammaire, un prédicat est donc défini par une formule du premier ordre.

La formule logique **valid** est un prédicat particulier qui indique que l'emplacement mémoire t auquel il est appliqué est alloué. Il existe deux variantes de ce prédicat. La première variante, **valid**$(t,\ L)$, dénote que le terme t est alloué dans la partie de la mémoire correspondant à l'état du programme labelisé par L. La deuxième variante, **valid**(t), est un cas particulier dans lequel le label prend la valeur prédéfinie *Here*, qui correspond à l'état courant du programme. Enfin, comme pour les termes, nous définissons une notation **at**$(p,\ L)$ qui permet d'appliquer le label L aux termes qui composent p. Par exemple,

$$\backslash\textbf{at}(t_1 \Rightarrow t_2,\ L)$$

est une abréviation pour

$$\backslash\textbf{at}(t_1,\ L) \Rightarrow \backslash\textbf{at}(t_2,\ L)$$

La figure 3.5 décrit le second type de déclarations, à savoir le type structure. Comme dans le langage C, un type structure est défini par un nom et des champs. Un champ est, à son tour, défini par un type et un identifiant. Les types de données sont soit des types primitifs ($int,\ bool,\ real,\ unit$) soit des types structure.

$struct\_decl$	::=	**struct** $id\{(field\ ;)^*\};$
$field$	::=	$type\ id$
$type$	::=	**int** | **real** | **bool** | **unit**
		| **struct** id

FIGURE 3.5 – Grammaire des déclarations de structures.

Le troisième et dernier type de déclaration, dont la grammaire est détaillée dans la figure 3.6, porte sur les définitions de fonction. Une fonction est définie par son nom, son type de retour, son contrat et enfin son corps.

$$
\begin{array}{rcl}
func\_decl & ::= & \textbf{function}\ type\ id\ (params)\ contrat\ \{body\}; \\
params & ::= & \epsilon \\
 & & \mid param(, param)^* \\
param & ::= & type\ id \\
body & ::= & expr
\end{array}
$$

FIGURE 3.6 – Grammaire des déclarations de fonctions.

Comme l'indique la syntaxe décrite dans la figure 3.7, le contrat d'une fonction est composé de quatre clauses : **requires**, **allocates**, **writes** et **ensures**. Les clauses **requires** et **ensures** permettent de

$$
\begin{array}{rcl}
contrat & ::= & requires\_clause^* \\
 & & writes\_clause^*\ allocates\_clause^* \\
 & & ensures\_clause^* \\
requires\_clauses & ::= & \textbf{requires}\ pred\ ; \\
writes\_clauses & ::= & \textbf{writes}\ locations\ ; \\
allocates\_clauses & ::= & \textbf{allocates}\ term\ (,\ term)^*; \\
 & & \mid \textbf{allocates}\ \backslash\textbf{nothing}; \\
ensures\_clauses & ::= & \textbf{ensures}\ pred\ ; \\
locations & ::= & location\ (,\ location)^* \mid \backslash\textbf{nothing} \\
location & ::= & term \rightarrow id \mid *term
\end{array}
$$

FIGURE 3.7 – Grammaire pour les contrats de fonctions.

stipuler les pré et post-conditions de la fonction annotée. Si aucune clause **requires** n'est donnée, alors cette fonction peut toujours être invoquée. Ce qui peut se formuler par une clause **requires** égale à **true**. De même pour la post-condition, où l'absence de clause **ensures** est interprétée par un **ensures true**.

Par ailleurs, la post-condition a généralement besoin de faire référence au résultat retourné par la fonction ou bien à la valeur d'un terme au moment où la fonction est appelée. Nous introduisons une variable logique notée \result qui dénote la valeur retournée par la fonction. Nous introduisons aussi l'abréviation $\backslash\textbf{old}(t)$ pour un terme t, respectivement $\backslash\textbf{old}(p)$ pour une formule p, pour $\backslash\textbf{at}(t,\ \mathit{Old})$, respectivement $\backslash\textbf{at}(p,\ \mathit{Old})$, où *Old* est un label prédéfini qui indique l'état initial de la fonction annotée par le contrat. Ce type d'abréviations ne peut être utilisé que dans une post-condition. Ça n'a, en effet, pas de sens de parler du résultat de la fonction ou de la valeur initiale d'un terme ou d'un prédicat, ailleurs que dans la post-condition.

La clause **writes** spécifie les emplacements mémoire qui sont potentiellement modifiés (**writes**) par la fonction. Si cette clause n'est pas spécifiée, alors la fonction annotée n'a pas d'effets de bord. Ce que nous pouvons également indiquer par **writes** **nothing**.

La dernière clause, **allocates**, spécifie les emplacements mémoire alloués par la fonction annotée. Dans ce cas, une fonction qui n'alloue aucun emplacement mémoire est annotée avec **allocates** **nothing**.

Dans la syntaxe des contrats de fonction, l'emplacement mémoire de la forme $*t$ représente l'ensemble des emplacements mémoire accessibles à partir de l'emplacement mémoire t.

Enfin, le corps d'une fonction est une expression. La syntaxe des expressions est décrite dans la figure 3.8. Nous considérons, dans cette partie du langage, les opérations unaires et binaires, la séquence d'expressions, l'instruction conditionnelle standard, la boucle while, la liaison locale, l'accès et la mise à jour d'un champ d'un type structure, l'allocation, l'appel de fonction et l'expression labelisée qui permet de déclarer un nouveau label. De manière non classique, nous ajoutons directement dans la syntaxe des annotations logiques : un invariant explicite pour les boucles, ainsi que l'expression `assert` .

$expr$	::=	c	Constante
		$\mid x$	Variable logique
		$\mid expr\ bin\_op\ expr$	
		$\mid expr\ rel\_op\ expr$	
		$\mid un\_op\ expr$	
		$\mid expr\ ;\ expr$	Séquence d'expressions
		$\mid$ `if` $expr$ `then` $expr$ `else` $expr$	Expression conditionnelle
		$\mid$ `while` $expr$ `invariant` $pred$ `do` $expr$ `end`	Expression itérative
		$\mid$ `let` $id = expr$ `in` $expr$	liaison locale
		$\mid expr \rightarrow id$	Accès au champ
		$\mid expr \rightarrow id\ =\ expr$	Mise à jour du champ
		$\mid$ `new` id	Allocation
		$\mid id\ (expr(, expr)^*)$	Appel de fonction
		$\mid$ `assert` $pred$	Assertion
		$\mid id : expr$	Expression labelisée

FIGURE 3.8 – Grammaire des expressions.

Le langage décrit par cette syntaxe ne contient pas de variables globales de programme. L'expression *let* permet d'introduire des variables locales uniquement. De plus, le seul type d'affectation que nous avons défini, permet d'affecter une valeur à un champ de structure. Ceci nous permet de limiter les effets de bord, dans le langage, à la mise à jour des champs de structures.

3.2 Typage

Maintenant que nous avons défini notre langage, nous avons besoin de préciser quels sont les programmes bien typés. Nous introduisons pour cela un ensemble de règles de typage. Ces règles déterminent les expressions, les termes, les formules, les listes de déclarations bien formées.

La première étape, de ce processus, consiste à définir un environnement de typage. Dans le but d'avoir des règles de typage concises, nous définissons un seul environnement de typage pour toutes les déclarations. Ainsi, un environnement de typage Γ contient à la fois le type des variables libres, les signatures des fonctions et prédicats, et les signatures des structures de données et même les labels. En procédant de la sorte, si la déclaration **logic** $\tau\ f(\tau_1\ x_1, \ldots, \tau_n\ x_n) = t$ est bien formée, alors on insère l'association $(f\ :\ lfun\ \tau_1 \ldots \tau_n, \tau)$ dans Γ. On fait de même avec les prédicats, en insérant $(p : lfun\ \tau_1 \ldots \tau_n, \texttt{Prop})$ dans Γ. Enfin, pour une structure de données **struct** $id\{\tau_1\ f_1; \ldots \tau_n\ f_n\}$, on

ajoute $(id : struct\ f_1\ \tau_1, \ldots, f_n\ \tau_n)$ dans Γ.

On note Δ une séquence de déclarations, nous définissons alors les jugements de typage suivants :
- $\Gamma \vdash \Delta$ signifie que dans l'environnement Γ, les déclarations dans Δ sont bien formées,
- $\Gamma \vdash t : \tau$ signifie que dans l'environnement Γ, le terme t est bien formé et a le type τ,
- $\Gamma \vdash f$: `Prop` signifie que dans l'environnement Γ, la formule f est bien formée (a le type `Prop`),
- $\Gamma \vdash e : \tau$ signifie que dans l'environnement Γ, l'expression e est bien formée et a le type τ.
- $\Gamma \vdash_{loc} l$) désigne que dans l'environnement Γ, l'ensemble des emplacements mémoire l sont valides pour la clause **writes**.
- $\Gamma \vdash_{alloc} t$ désigne que dans l'environnement Γ, l'ensemble des termes t sont valides pour la clause **allocates**.

Nous allons, dans ce qui suit, définir l'ensemble des règles correspondant à chacun de ces jugements. Nous commencerons par définir les règles de typage des termes puis des formules, car ces dernières sont nécessaires pour le typage des déclarations. Nous finirons cette section en présentant les règles associées aux expressions du langage.

3.2.1 Typage des termes du langage

Les règles de typage des termes, qui correspondent au jugement de typage $\Gamma \vdash t : \tau$ sont présentées ci-dessous.

Constantes

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash r : \texttt{real}} \qquad \frac{}{\Gamma \vdash () : \texttt{unit}}$$

$$\frac{}{\Gamma \vdash \textbf{null} : \textbf{struct}\ S}$$

$$\frac{}{\Gamma \vdash \textbf{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \textbf{false} : \texttt{bool}}$$

$$\frac{\Gamma \vdash t : \texttt{int}}{\Gamma \vdash t : \texttt{real}}$$

Variables logiques

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Opérations unaires

$$\frac{\Gamma \vdash t : \texttt{int}}{\Gamma \vdash op\ t : \texttt{int}}\ op \in \{-\} \qquad \frac{\Gamma \vdash t : \texttt{bool}}{\Gamma \vdash op\ t : \texttt{bool}}\ op \in \{!\}$$

Opérations binaires

$$\frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau \qquad \tau \in \{\texttt{int}, \texttt{real}\}}{\Gamma \vdash t_1\ op\ t_2 : \tau}\ op \in \{+, -, *\}$$

$$\frac{\Gamma \vdash t_1 : \texttt{bool} \qquad \Gamma \vdash t_2 : \texttt{bool}}{\Gamma \vdash t_1\ op\ t_2 : \texttt{bool}}\ op \in \{\&\&, ||\} \qquad \frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1\ op\ t_2 : \texttt{bool}}\ op \in \{=, != \}$$

$$\frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau \qquad \tau \in \{\texttt{int},\ \texttt{real}\}}{\Gamma \vdash t_1\ op\ t_2 : \texttt{bool}} \quad op\ \in \{<, <=, >, >=\}$$

Accès à un champ

$$\frac{l \in \Gamma \qquad (S : struct \ldots) \in \Gamma \qquad \Gamma \vdash t : \textbf{struct}\ S \qquad f \in \mathsf{Fields}(S) \qquad \tau = type(f)}{\Gamma \vdash t \xrightarrow{l} f : \tau}$$

Pour que le terme $t \xrightarrow{l} f$ soit bien typé et ait le type τ, il faut que le label l soit déclaré, que le terme t soit de type structure S déjà défini, et que f appartienne à l'ensemble $\mathsf{Fields}(S)$ des champs de ce type structure S et soit de type τ. La première règle de typage dans la section 3.2.5 (expression labelisée) montre comment les labels sont introduits dans l'environnement Γ.

Application de fonctions logiques

$$\frac{(f : lfun\ \tau_1, \ldots \tau_n, \tau) \in \Gamma \qquad \Gamma \vdash t_i : \tau_i}{\Gamma \vdash f(t_1, \ldots, t_n) : \tau}$$

3.2.2 Typage des formules logiques du langage

Les règles de typage des formules sont décrites en utilisant le jugement de typage : $\Gamma \vdash f : \texttt{Prop}$

Formules atomiques

$$\frac{\Gamma \vdash t : \texttt{bool}}{\Gamma \vdash t : \texttt{Prop}}$$

Négation

$$\frac{\Gamma \vdash p : \texttt{Prop}}{\Gamma \vdash \neg p : \texttt{Prop}}$$

Opérations logiques

$$\frac{\Gamma \vdash p_1 : \texttt{Prop} \qquad \Gamma \vdash p_2 : \texttt{Prop}}{\Gamma \vdash p_1\ op\ p_2 : \texttt{Prop}} \qquad op \in \{\vee,\ \wedge,\ \Rightarrow,\ \Leftrightarrow\}$$

Let

$$\frac{\Gamma \vdash t : \tau \qquad \{x : \tau\} \cdot \Gamma \vdash p : \texttt{Prop}}{\Gamma \vdash \texttt{let}\ x = t\ \texttt{in}\ p : \texttt{Prop}}$$

Quantificateur universel

$$\frac{\{x_i : \tau_i | 1 \leq i \leq n\}.\ \Gamma \vdash p : \texttt{Prop}}{\Gamma \vdash \backslash forall\ \tau_1\ x_1 \ldots \tau_n\ x_n\ ;\ p : \texttt{Prop}}$$

Quantificateur existentiel

$$\frac{\{x_i : \tau_i | 1 \leq i \leq n\}.\Gamma \vdash p : \texttt{Prop}}{\Gamma \vdash \backslash exists\ \tau_1\ x_1\ \ldots \tau_n x_n\ ;\ p : \texttt{Prop}}$$

Pointeur valide

$$\frac{L \in \Gamma \qquad (S : struct \ldots) \in \Gamma \qquad \Gamma \vdash t : \textbf{struct}\ S}{\Gamma \vdash \textbf{valid}(t,\ L) : \texttt{Prop}}$$

$$\frac{(S : struct \ldots) \in \Gamma \qquad \Gamma \vdash t : \textbf{struct}\ S}{\Gamma \vdash \textbf{valid}(t) : \texttt{Prop}}$$

Application de prédicat

$$\frac{\Gamma \vdash t_i : \tau_i \qquad (p : lfun\ \tau_1 \ldots \tau_n\ ,\ \texttt{Prop}) \in \Gamma}{\Gamma \vdash p(t_1, \ldots, t_n) : \texttt{Prop}}$$

3.2.3 Typage des clauses writes et allocates

Les emplacements mémoire sont des données particulières. Nous définissons alors un nouveau type noté loc dont les valeurs correspondent concrètement à des adresses mémoire. En d'autres termes, les pointeurs dans le programme sont des variables de type loc. Le jugement $\Gamma \vdash_{loc} l$ permet de décrire les règles d'inférence de ce type loc.

$$\frac{}{\emptyset \vdash_{loc} \backslash \textbf{nothing}} \qquad \frac{\Gamma \vdash_{loc} l_1 \qquad \Gamma \vdash_{loc} l_2}{\Gamma \vdash_{loc} l_1, l_2}$$

$$\frac{(S : struct \ldots) \in \Gamma \qquad f \in \mathsf{Fields}(S) \qquad \Gamma \vdash t : \textbf{struct}\ S}{\Gamma \vdash_{loc} t \to f}$$

$$\frac{(S : struct \ldots) \in \Gamma \qquad \Gamma \vdash t : \textbf{struct}\ S}{\Gamma \vdash_{loc} *t}$$

Nous décrivons ensuite les règles d'inférence pour la clause **allocates**.

$$\frac{}{\emptyset \vdash_{alloc} \backslash \textbf{nothing}} \qquad \frac{\Gamma \vdash_{alloc} t_1 \qquad \Gamma \vdash_{alloc} t_2}{\Gamma \vdash_{alloc} t_1, t_2}$$

$$\frac{(S : struct \ldots) \in \Gamma \qquad \Gamma \vdash t : \textbf{struct}\ S}{\Gamma \vdash_{alloc} t}$$

3.2.4 Typage des déclarations du langage

Les règles de typage des déclarations spécifient que si une déclaration est bien formée, alors elle est insérée dans l'environnement de typage Γ.

Pour ne pas alourdir les règles, nous n'écrivons pas les hypothèses de bonne formation des types. En réalité, pour chaque type apparaissant dans la syntaxe, on doit vérifier qu'il est bien déclaré (si c'est un type structure).

Déclaration de type logique

$$\frac{\{id : type\} \cdot \Gamma \vdash \Delta}{\Gamma \vdash \textbf{type}\ id; \Delta}$$

Déclaration de constantes logiques

$$\frac{\{id : const\ \tau\} \cdot \Gamma \vdash \Delta}{\Gamma \vdash \textbf{logic}\ \tau\ id\ ; \Delta}$$

Définition de constante logique

$$\frac{\{id : const\ \tau\} \cdot \Gamma \vdash \Delta \qquad \Gamma \vdash t : \tau}{\Gamma \vdash \textbf{logic}\ \tau\ id = t; \Delta}$$

Déclaration de prédicat

$$\frac{\{id : lfun\ \tau_1 \ldots \tau_n,\ \texttt{Prop}\} \cdot \Gamma \vdash \Delta}{\Gamma \vdash \textbf{predicate}\ id(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ ; \Delta}$$

Déclaration de fonction logique

$$\frac{\{id : lfun\ \tau_1 \ldots \tau_n, \tau\} \cdot \Gamma \vdash \Delta}{\Gamma \vdash \textbf{logic}\ \tau\ id(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ ; \Delta}$$

Définition de prédicat

$$\frac{\{x_i : \tau_i | 1 \leq i \leq n\} \cdot \Gamma \vdash p \qquad \{id : lfun\ \tau_1 \ldots \tau_n,\ \texttt{Prop}\} \cdot \Gamma \vdash \Delta}{\Gamma \vdash \textbf{predicate}\ id(\tau_1\ x_1, \ldots, \tau_n\ x_n) = p\ ; \Delta}$$

Définition de fonction logique

$$\frac{\{x_i : \tau_i | 1 \leq i \leq n\} \cdot \Gamma \vdash t : \tau \qquad \Gamma \cdot \{id : lfun\ \tau_1 \ldots \tau_n, \tau\} \vdash \Delta}{\Gamma \vdash \textbf{logic}\ \tau\ id(\tau_1\ x_1, \ldots, \tau_n\ x_n) = t; \Delta}$$

Déclaration d'axiome

$$\frac{\Gamma \vdash p : \texttt{Prop} \qquad \Gamma \vdash \Delta}{\Gamma \vdash \textbf{axiom}\ id : p; \Delta}$$

Déclaration de lemme

$$\frac{\Gamma \vdash p : \texttt{Prop} \qquad \Gamma \vdash \Delta}{\Gamma \vdash \textbf{lemma}\ id : p; \Delta}$$

Définition de structures

$$\frac{\{S : \textbf{struct}\ \tau_1 \ldots \tau_n\} \cdot \Gamma \vdash \Delta \qquad \text{Pour tout } i, j.\ i \neq j, f_i \neq f_j}{\Gamma \vdash \textbf{struct}\ S\{\tau_1 f_1; \ldots; \tau_n f_n;\};\ \Delta}$$

Une déclaration de structure est bien formée si tous les champs de la structure ont des noms différents.

Définition de fonctions

$$\frac{\begin{array}{c}\Gamma' = \{x_i : \tau_i | 1 \leq i \leq n\} \cdot \Gamma \qquad \{\backslash\mathsf{result} : \tau\} \cdot \Gamma' \vdash_{alloc} A \qquad \Gamma' \vdash_{loc} W \\ \{fun\ f : \tau_1 \ldots \tau_n, \tau\} \cdot \Gamma' \vdash Body : \tau \qquad \Gamma' \vdash Pre : \texttt{Prop} \\ \{Old\} \cdot \{\backslash\mathsf{result} : \tau\} \cdot \Gamma' \vdash Post : \texttt{Prop} \qquad \{fun\ f : \tau_1 \ldots \tau_n, \tau\} \cdot \Gamma \vdash \Delta\end{array}}{\Gamma \vdash \textbf{function}\ \tau\ f\ (\tau_1\ x_1, \ldots \tau_n\ x_n)\ contrat\{\ Body\ \}\ ;\ \Delta}$$

où f est déclarée comme suit :

function τ $f(\tau_1\ x_1, \ldots, \tau_n\ x_n)$
requires *Pre*
allocates A
writes W
ensures *Post*
{ Body }

Pour qu'une déclaration de fonction soit bien formée, si Γ' est l'environnement obtenu en étendant Γ avec les paramètres formels de la fonction, alors il faut s'assurer que : les ensembles d'emplacements mémoire W sont bien typés dans Γ', le corps de la fonction est bien typé dans Γ', la précondition Pre est également bien formée dans Γ' et enfin que la post-condition $Post$ est bien formée dans l'environnement Γ' que nous étendons à son tour avec une association entre $\backslash\backslash\mathsf{result}$ et τ le type de retour de la fonction et le label *Old*.

Notez que nous autorisons les fonctions récursives comme dans le langage C.

3.2.5 Typage des expressions du langage

Le dernier ensemble de règles de typage permet de typer les expressions du langage.

Les constantes et les variables logiques sont typées comme dans les termes.

Expression labelisée

$$\frac{\Gamma \cdot \{L\} \vdash e : \tau}{\Gamma \vdash\ L : e : \tau}$$

Cette règle spécifie que l'expression $L : e$ est bien typée de type τ si et seulement si l'expression e est bien typée de type τ dans l'environnement Γ étendu avec le label L .

Opérations unaires

$$\frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash\ op\ e : \texttt{int}}\ op \in \{-\} \qquad \frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash\ op\ e : \texttt{bool}}\ op \in \{!\}$$

Opérations binaires

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \in \{\texttt{int}, \texttt{real}\}}{\Gamma \vdash e_1\ op\ e_2 : \tau}\ op\ \in \{+, -, *\}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1\ op\ e_2 : \texttt{bool}}\ op\ \in \{\&\&, ||\}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\texttt{int}, \texttt{real}\}}{\Gamma \vdash e_1 \; op \; e_2 : \texttt{bool}} \quad op \in \{<, <=, >, >=\}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \; op \; e_2 : \texttt{bool}} \quad op \in \{=, !=\}$$

Séquence

$$\frac{\Gamma \vdash e_1 : \texttt{unit} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1; e_2 : \tau}$$

Liaison locale

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \{x : \tau_1\} \cdot \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

Expression conditionnelle

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}$$

Expression itérative

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash inv : \texttt{Prop} \qquad \Gamma \vdash e_1 : \texttt{unit}}{\Gamma \vdash \texttt{while}\ e\ \texttt{invariant}\ inv\ \texttt{do}\ e_1\ \texttt{end} : \texttt{unit}}$$

Allocation

$$\frac{(struct\ S : \dots) \in \Gamma}{\Gamma \vdash \texttt{new}\ \ S : \textbf{struct}\ S}$$

Accès à un champ

$$\frac{(S : struct \dots) \in \Gamma \qquad f \in \mathsf{Fields}(S) \qquad type(f) = \tau \qquad \Gamma \vdash e : \textbf{struct}\ S}{\Gamma \vdash e \to f : \tau}$$

Mise à jour d'un champ

$$\frac{(S : struct\ \dots) \in \Gamma \qquad f \in \mathsf{Fields}(S) \qquad type(f) = \tau \qquad \Gamma \vdash e_1 : \textbf{struct}\ S \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \to f = e_2 : \texttt{unit}}$$

Assertion

$$\frac{\Gamma \vdash p : \texttt{Prop}}{\Gamma \vdash \texttt{assert}\ \ p : \texttt{unit}}$$

Appel de fonction

$$\frac{(fun\ f : \tau_1 \dots \tau_n,\ \tau) \in \Gamma \quad \Gamma \vdash e_i : \tau_i\ 1 \leq i \leq n}{\Gamma \vdash f(e_1, \dots, e_n) : \tau}$$

3.3 Sémantique du langage

La troisième étape dans la présentation de notre langage consiste à munir ce dernier d'une sémantique. Les programmes considérés dans cette section sont bien typés.

3.3.1 Notations pour les fonctions partielles

Pour exprimer la sémantique opérationnelle de notre langage, nous utilisons des fonctions partielles. Nous allons utiliser les notations suivantes qui sont plus ou moins celles de la théorie des ensembles telles qu'utilisées dans la méthode B par exemple.

Si F est une fonction partielle, on note

- $\texttt{dom}(F)$ le domaine de définition de F.
- $F(x)$ l'application de F à x pour $x \in \texttt{dom}(F)$. Dans certains cas, nous utiliserons une notation alternative $F@x$.
- $F[x := y]$ la mise à jour de F au point x par la valeur y.
- La notation $F = F_1 \oplus F_2$ indique que F est l'union disjointe de F_1 et F_2, c'est-à-dire :
  - $\texttt{dom}(F_1) \cap \texttt{dom}(F_2) = \emptyset$,
  - $\texttt{dom}(F_1) \cup \texttt{dom}(F_2) = \texttt{dom}(F)$,
  - $F(x) = F_1(x)$ si $x \in \texttt{dom}(F_1)$,
  - $F(x) = F_2(x)$ si $x \in \texttt{dom}(F_2)$.

3.3.2 Valeurs et états mémoire

Dans notre langage, chaque terme et chaque expression vont s'évaluer en une *valeur*. L'ensemble des valeurs est formé des constantes entières, réelles, booléennes, etc. et l'ensemble des *emplacements mémoire*, typiquement notés $loc, loc_1, \ldots$ y compris la constante `null`.

Un programme P s'exécute dans un état mémoire défini par une paire d'environnements $(\Pi,\ \mathcal{H})$. Le premier, Π, est un environnement local. C'est une fonction partielle qui associe à chaque variable locale x sa valeur. Un *tas mémoire* est une fonction partielle qui à une paire (loc, f) composée d'un emplacement mémoire et d'un nom de champ, associe la valeur qui est stockée à cet endroit de la mémoire. Le second environnement, $\mathcal{H}$, est alors une collection de tas mémoire indexée par les labels, c'est-à-dire une fonction partielle des labels vers les tas mémoire. Autrement dit, un tas mémoire *complet* est un historique des états de la mémoire à chaque label rencontré.

Nous utiliserons la notation $\mathcal{H}@L$ pour désigner l'application de $\mathcal{H}$ à L, c'est-à-dire :
- $\mathcal{H}@L$ retourne le tas mémoire dans l'état L du programme,
- $\mathcal{H}@L(v, f)$ est la valeur associée à la paire (v, f) dans $\mathcal{H}@L$,
- $\mathcal{H}@L[v_1, f := v_2]$ est la mise à jour de la valeur v_2 à la paire (v_1, f) dans $\mathcal{H}@L$.

Une remarque générale est que les tas mémoire que l'on considère seront toujours bien formés vis-à-vis des définitions de structure au sens où si $(v, f) \in \texttt{dom}(\mathcal{H}@L)$ pour un certain champ f d'un type structure S, alors $(v, g) \in \texttt{dom}(\mathcal{H}@L)$ pour tous les autres champs g de S.

Nous utiliserons deux autres notations :

– $\texttt{fresh}(H)$ dénote n'importe quel emplacement mémoire loc tel que $(loc, f) \notin \texttt{dom}(H@L)$ (pour tout f),
– $default(S, f)$ dénote la valeur par défaut du champ f de la structure S, c'est-à-dire : 0 pour les entiers et les réels, **false** pour les booléens et `null` pour les emplacements mémoire.

3.3.3 Sémantique des annotations logiques

Nous munissons la logique de notre langage d'une sémantique dénotationnelle. Nous commençons par définir, dans la figure 3.9, l'évaluation des types primitifs du langage, où le type `int` s'évalue en l'ensemble des entiers relatifs, le type `real` en l'ensemble des réels, le type `bool` en l'ensemble composé des deux valeurs **true** et **false** et enfin le type `unit` en le singleton qui contient la valeur $()$.

$$\begin{array}{rcl}
[\![int]\!]_{\Pi,\,\mathcal{H}} & = & \mathbb{Z} \\
[\![real]\!]_{\Pi,\,\mathcal{H}} & = & \mathbb{R} \\
[\![bool]\!]_{\Pi,\,\mathcal{H}} & = & \mathbb{B} = \{true, false\} \\
[\![unit]\!]_{\Pi,\,\mathcal{H}} & = & \mathbb{U} = \{()\}
\end{array}$$

FIGURE 3.9 – Évaluation des types primitifs

Ensuite, la valeur d'un terme t, dans un état $(\Pi,\ \mathcal{H})$, est déterminée par une fonction récursive $[\![\_]\!]_{\Pi,\,\mathcal{H}}$, comme indiqué dans la figure 3.10.

$$\begin{array}{rcl}
[\![v]\!]_{\Pi,\,\mathcal{H}} & = & v \ (v \text{ est une constante entière, réelle ou booléenne}) \\
[\![x]\!]_{\Pi,\,\mathcal{H}} & = & \Pi[x] \ (x \text{ est une variable logique}) \\
[\![t \xrightarrow{l} f]\!]_{\Pi,\,\mathcal{H}} & = & \mathcal{H}@l([\![t]\!]_{\Pi,\,\mathcal{H}},\ f) \\
[\![\ un_op\ t]\!]_{\Pi,\,\mathcal{H}} & = & [\![un_op]\!]\ [\![t]\!]_{\Pi,\,\mathcal{H}},\ \ un_op \in \{-,\ !\} \\
[\![t_1\ rel_op\ t_2]\!]_{\Pi,\,\mathcal{H}} & = & [\![t_1]\!]_{\Pi,\mathcal{H}}\ [\![rel_op]\!]\ [\![t_2]\!]_{\Pi,\,\mathcal{H}},\ \ rel_op \in \{<,\ >,\ <=,\ >=,\ ==,\ !=\} \\
[\![t_1\ bin_op\ t_2]\!]_{\Pi,\,\mathcal{H}} & = & [\![t_1]\!]_{\Pi,\,\mathcal{H}}\ [\![bin_op]\!]\ [\![t_2]\!]_{\Pi,\mathcal{H}},\ \ bin_op \in \{+,\ -,\ *\} \\
[\![f(t_1, \ldots, t_n)]\!]_{\Pi,\,\mathcal{H}} & = & [\![f]\!]([\![t_1]\!]_{\Pi,\,\mathcal{H}}, \ldots, [\![t_n)]\!]_{\Pi,\,\mathcal{H}})
\end{array}$$

FIGURE 3.10 – Sémantique dénotationnelle des termes

Le cas de l'accès à un champ de structure est intéressant. En effet, l'évaluation d'un terme $t \xrightarrow{l} f$ est la valeur associée à la paire $([\![t]\!]_{\Pi,\,\mathcal{H}},\ f)$ dans le tas $\mathcal{H}@l$.

Puis, nous définissons l'évaluation des formules par une nouvelle fonction récursive $[\![\_]\!]_{\Pi,\,\mathcal{H}}$, décrite dans la figure 3.11.

Les opérateurs $[\![bin\_op]\!]$, $[\![un\_op]\!]$, $[\![rel\_op]\!]$, $[\![log\_op]\!]$ et $[\![\neg]\!]$ sont, respectivement, les sémantiques naturelles des opérateurs bin_op, un_op, rel_op, log_op et $\neg$ sur les entiers, les réels ou sur les booléens.

Par ailleurs, les sémantiques $[\![f]\!]$ des fonctions logiques et $[\![p]\!]$ des prédicats sont supposées données

$$
\begin{array}{rcl}
[\![t]\!]_{\Pi,\,\mathcal{H}} & = & [\![t]\!]_{\Pi,\,\mathcal{H}} = true \\
[\![\neg p]\!]_{\Pi,\,\mathcal{H}} & = & [\![\neg]\!]\ [\![p]\!]_{\Pi,\,\mathcal{H}} \\
[\![\textbf{valid}(t,\ L)]\!]_{\Pi,\,\mathcal{H}} & = & ([\![t]\!]_{\Pi,\,\mathcal{H}}, f) \in \mathtt{dom}(\mathcal{H}@L), \\
 & & \qquad \text{Si } type(t) = \textbf{struct}\ S \text{ pour n'importe quel } f \in \mathsf{Fields}(S) \\
[\![p_1\ log\_op\ p_2]\!]_{\Pi,\,\mathcal{H}} & = & [\![p_1]\!]_{\Pi,\mathcal{H}}\ [\![log\_op]\!]\ [\![p_2]\!]_{\Pi,\,\mathcal{H}} \\
[\![p(t_1,\ldots,t_n)]\!]_{\Pi,\,\mathcal{H}} & = & [\![p]\!]([\![t_1]\!]_{\Pi,\,\mathcal{H}},\ldots,[\![t_n)]\!]_{\Pi,\,\mathcal{H}}) \\
[\![\mathtt{let}\ x = t\ \mathtt{in}\ p]\!]_{\Pi,\,\mathcal{H}} & = & [\![p]\!]_{\Pi[x:=[\![t]\!]_{\Pi,\ \mathcal{H}}],\mathcal{H}} \\
[\![\backslash forall\ \tau_1 x_1 \ldots \tau_n x_n\ ;\ p]\!]_{\Pi,\,\mathcal{H}} & = & \forall\ v_1 : [\![\tau_1]\!], \ldots, v_n : [\![\tau_n]\!].\ [\![p]\!]_{\Pi\{x_i:=v_i\},\mathcal{H}} \\
[\![\backslash exists\ \tau_1 x_1 \ldots \tau_n x_n\ ;\ p]\!]_{\Pi,\,\mathcal{H}} & = & \exists\ v_1 : [\![\tau_1]\!], \ldots, v_n : [\![\tau_n]\!].\ [\![p]\!]_{\Pi\{x_i:=v_i\},\mathcal{H}}
\end{array}
$$

FIGURE 3.11 – Sémantique dénotationnelle des prédicats

par n'importe quel modèle des théories dans lesquelles ils ont été définis. En d'autres termes, notre sémantique dénotationnelle de la logique est paramétrée par un tel choix des modèles des théories.

L'évaluation du prédicat **valid** s'explique par le fait que, par construction, si l'emplacement mémoire loc est alloué dans le tas mémoire, alors toutes les paires de la forme $(loc,\ f)$ telles que loc est de type **struct** S et $f \in \mathsf{Fields}(S)$, appartiennent au domaine du tas mémoire.

3.3.4 Sémantique des programmes

Nous munissons le langage de programmation d'une sémantique opérationnelle à petits pas telle que le jugement $\Pi,\ \mathcal{H}, e \rightsquigarrow \Pi', \mathcal{H}', e'$ exprime que, si dans l'état $(\Pi,\ \mathcal{H})$ on exécute l'expression e, alors on se retrouve dans le nouvel état $(\Pi', \mathcal{H}')$ et e' est la prochaine expression à exécuter. Cette sémantique suit le modèle connu sous le nom de *SOS* (*Structural Operational Semantics*) et proposé par Plotkin en 1980 [88]. Nous considérons ici que les programmes sont bien typés.

Expression labelisée

$$\frac{\mathcal{H}' = \mathcal{H}[L := \mathcal{H}@\mathit{Here}]}{\Pi,\ \mathcal{H}, L :\ e \rightsquigarrow \Pi,\ \mathcal{H}',\ e}$$

Rappelons que le tas mémoire $\mathcal{H}$ est indexé par les labels, cette règle consiste donc à associer au nouveau label L une copie de ce qui est actuellement associé au label *Here*, c'est-à-dire une copie de l'état courant.

Opérations unaires

$$\frac{\Pi,\ \mathcal{H}, e \rightsquigarrow \Pi', \mathcal{H}',\ e'}{\Pi,\ \mathcal{H},\ op\ e \rightsquigarrow \Pi',\ \mathcal{H}',\ op\ e'} \quad op \in \{-,\ !\}$$

$$\frac{v' = op\ v}{\Pi,\ \mathcal{H},\ op\ v \rightsquigarrow \Pi,\ \mathcal{H},\ v'} \quad op \in \{-,\ !\}$$

La première règle est une règle de contexte : elle permet d'exprimer le fait qu'il faut d'abord évaluer l'opérande e de l'expression avant de pourvoir évaluer $un\_op\ e$.

Opérations binaires

$$\frac{\Pi,\ \mathcal{H}, e_1 \rightsquigarrow \Pi', \mathcal{H}', e_1'}{\Pi,\ \mathcal{H},\ e_1\ op\ e_2 \rightsquigarrow \Pi', \mathcal{H}', e_1'\ op\ e_2} \quad op \in \{bin\_op,\ rel\_op,\ log\_op\}$$

$$\frac{\Pi,\ \mathcal{H}, e_2 \rightsquigarrow \Pi', \mathcal{H}', e_2'}{\Pi,\ \mathcal{H}, v_1\ op\ e_2 \rightsquigarrow \Pi', \mathcal{H}', v_1\ op\ e_2'} \quad op \in \{bin\_op,\ rel\_op,\ log\_op\}$$

$$\frac{v = v_1\ op\ v_2}{\Pi,\ \mathcal{H}, v_1\ op\ v_2 \rightsquigarrow \Pi,\ \mathcal{H}, v} \quad op \in \{bin\_op,\ rel\_op,\ log\_op\}$$

Les deux premières règles sont également des règles de contexte. Elles déterminent l'ordre d'évaluation des expressions : de gauche à droite, c'est-à-dire qu'il faut commencer par évaluer l'opérande gauche de l'opération, et ce n'est que lorsque celui-ci est une valeur, que nous pouvons évaluer le deuxième opérande.

Dans la suite, nous définirons des règles de contexte, à chaque fois que l'expression est composée de sous-expressions, afin de spécifier l'ordre d'évaluation.

Séquence

$$\frac{\Pi,\ \mathcal{H}, e_1 \rightsquigarrow \Pi', \mathcal{H}', e_1'}{\Pi,\ \mathcal{H}, e_1; e_2 \rightsquigarrow \Pi', \mathcal{H}', e_1'; e_2}$$

$$\frac{}{\Pi,\ \mathcal{H}, (); e_2 \rightsquigarrow \Pi,\ \mathcal{H}, e_2}$$

Expression conditionnelle

$$\frac{\Pi,\ \mathcal{H}, e \rightsquigarrow \Pi', \mathcal{H}', e'}{\Pi,\ \mathcal{H}, \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow \Pi', \mathcal{H}', \texttt{if}\ e'\ \texttt{then}\ e_1\ \texttt{else}\ e_2}$$

$$\frac{}{\Pi,\ \mathcal{H}, \texttt{if}\ true\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow \Pi,\ \mathcal{H}, e_1}$$

$$\frac{}{\Pi,\ \mathcal{H}, \texttt{if}\ false\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow \Pi,\ \mathcal{H}, e_2}$$

Liaison locale

$$\frac{\Pi,\ \mathcal{H}, e_1 \rightsquigarrow \Pi', \mathcal{H}', e_1'}{\Pi,\ \mathcal{H}, \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightsquigarrow \Pi', \mathcal{H}', \texttt{let}\ x = e_1'\ \texttt{in}\ e_2}$$

$$\frac{}{\Pi,\ \mathcal{H}, \texttt{let}\ x = v_1\ \texttt{in}\ e_2 \rightsquigarrow \Pi[x := v_1], \mathcal{H}, e_2}$$

Allocation

$$\frac{loc = \texttt{fresh}(\mathit{Here}) \qquad \mathcal{H}' = \mathcal{H}[\mathit{Here} := \mathit{Here} \oplus \{(loc,\ f) \leftarrow default(S,\ f) \mid f \in \mathsf{Fields}(S)\}]}{\Pi,\ \mathcal{H}, \texttt{new}\ S \rightsquigarrow \Pi, \mathcal{H}',\ loc}$$

Accès à un champ

$$\frac{\Pi,\ \mathcal{H},\ e \rightsquigarrow \Pi', \mathcal{H}',\ e'}{\Pi,\ \mathcal{H},\ e \to f \rightsquigarrow \Pi', \mathcal{H}',\ e' \to f}$$

$$\frac{(v, f) \in \mathtt{dom}(\mathcal{H}@\mathit{Here}) \qquad v' = \mathcal{H}@\mathit{Here}(v,\ f)}{\Pi,\ \mathcal{H}, v \to f \rightsquigarrow \Pi,\ \mathcal{H},\ v'}$$

L'évaluation de la valeur d'un champ s'effectue en lisant l'état courant $\mathcal{H}@\mathit{Here}$. Notez que l'expression $v \to f$ ne s'exécute pas si v n'est pas allouée.

Mise à jour de champ

$$\frac{\Pi,\ \mathcal{H}, e_1 \rightsquigarrow \Pi', \mathcal{H}', e_1'}{\Pi,\ \mathcal{H}, e_1 \to f = e_2 \rightsquigarrow \Pi', \mathcal{H}', e_1' \to f = e_2}$$

$$\frac{\Pi,\ \mathcal{H}, e_2 \rightsquigarrow \Pi', \mathcal{H}', e_2'}{\Pi,\ \mathcal{H}, v \to f = e_2 \rightsquigarrow \Pi', \mathcal{H}', v \to f = e_2'}$$

$$\frac{(v_1,\ f) \in \mathtt{dom}(\mathcal{H}@\mathit{Here}) \qquad \mathcal{H}' = \mathcal{H}[\mathit{Here} := \mathcal{H}@\mathit{Here}[(v_1,\ f) :=\ v_2]]}{\Pi,\ \mathcal{H}, v_1 \to f = v_2 \rightsquigarrow \Pi,\ \mathcal{H}',\ ()}$$

Dans ce cas aussi, l'exécution requiert que v_1 soit allouée.

Expression itérative

$$\frac{[\![Inv]\!]_{\Pi,\ \mathcal{H}} \text{ est valide}}{\begin{array}{c}\Pi,\ \mathcal{H}, \mathtt{while}\ e_1\ \mathtt{invariant}\ Inv\ \mathtt{do}\ e_2\ \mathtt{end} \rightsquigarrow \\ \Pi,\ \mathcal{H},\ \mathtt{if}\ e_1\ \mathtt{then}\ (e_2; \mathtt{while}\ e_1\ \mathtt{invariant}\ Inv\ \mathtt{do}\ e_2\ \mathtt{end})\ \mathtt{else}\ ()\end{array}}$$

Nous spécifions ici de dire que l'expression `while` ne s'exécute que si son invariant est vérifié au moment où nous testons la condition de la boucle.

Assertion

$$\frac{[\![p]\!]_{\Pi,\ \mathcal{H}} \text{ est vraie}}{\Pi,\ \mathcal{H}, \mathtt{assert}\ \ p \rightsquigarrow \Pi,\ \mathcal{H}, ()}$$

Enfin, l'expression `assert` ne s'exécute que si la formule p est vérifiée.

Appel de fonction En premier lieu, nous avons besoin d'une règle de contexte pour évaluer les arguments.

$$\frac{\Pi,\ \mathcal{H}, e_{i+1} \rightsquigarrow \Pi', \mathcal{H}', e_{i+1}' \quad 0 \leq i < n-1}{\Pi,\ \mathcal{H},\ f(v_1, \ldots, v_i, e_{i+1}, \ldots, e_n) \rightsquigarrow \Pi', \mathcal{H}',\ f(v_1, \ldots, v_i, e_{i+1}', \ldots, e_n)}$$

Afin de définir une sémantique opérationnelle à petits pas pour l'appel de fonction, nous avons besoin d'une pseudo-expression « return » qui permet de sauvegarder l'état local de l'appelant et de le restaurer à la fin.

$$\frac{\Pi_{prm} = \{x_i \leftarrow v_i \mid 1 \leq i \leq n\} \qquad [\![Pre]\!]_{\Pi_{prm},\mathcal{H}} \text{ est vraie}}{\Pi,\ \mathcal{H}, f(v_1, \ldots, v_n) \rightsquigarrow \Pi_{prm},\ \mathcal{H},\ \mathit{Old} : \texttt{let}\ v = \mathit{Body}\ \texttt{in}\ \mathit{return}(v,\ \mathit{Post},\ W,\ A,\ \Pi,\ \Pi_{prm})}$$

où v est une variable fraîche. Notez l'introduction du label *Old* qui permettra de faire une copie de l'état mémoire, lors de l'appel, dans le tas mémoire $\mathcal{H}$.

Fondamentalement, l'exécution d'un appel de fonction commence par vérifier que la précondition est vraie. Le cas échéant, l'expression qui lie le corps de la fonction à une variable fraîche v, dans une expression « return », est exécutée dans un état mémoire dans lequel l'environnement local, Π_{prm}, associe les paramètres effectifs aux paramètres formels uniquement. Le tas mémoire $\mathcal{H}$, lui, ne change pas.

L'expression « return » est construite à partir des paramètres suivants : la post-condition $Post$ de la fonction appelée, les ensembles W et A qui sont respectivement spécifiés dans les clauses **writes** et **allocates**, l'environnement local Π tel qu'il est au moment de l'appel de la fonction et l'environnement Π_{prm}.

Pseudo-expression return La sémantique de cette expression est décrite ci-dessous.

$$\frac{[\![\texttt{let}\ \backslash\mathsf{result} = v\ \texttt{in}\ P]\!]_{\Pi',\mathcal{H}} \qquad \mathit{Assigns}(\mathcal{H},\ [\![W]\!]^{loc}_{\Pi',\mathcal{H}}) \qquad \mathit{Allocates}(\mathcal{H},\ [\![A]\!]^{alloc}_{\Pi',\mathcal{H}})}{\Pi'', \mathcal{H}, \mathit{return}(v,\ P,\ W,\ A,\ \Pi,\ \Pi') \rightsquigarrow \Pi,\ \mathcal{H},\ v}$$

Cette règle spécifie que l'expression $return$ bloque si la post-condition P n'est pas vérifiée ou si l'une des deux propriétés mathématiques $\mathit{Assigns}(\mathcal{H},\ [\![W]\!]^{loc}_{\Pi',\mathcal{H}})$ ou $\mathit{Allocates}(\mathcal{H},\ [\![A]\!]^{alloc}_{\Pi',\mathcal{H}})$ correspondant respectivement aux ensembles W et A et dont les définitions sont présentées ci-dessous, n'est pas valide.

$$\begin{array}{l}\mathit{Assigns}(\mathcal{H},\ S) \stackrel{\text{def}}{=} \\ \quad \texttt{dom}(\mathcal{H}@\mathit{Old}) \subseteq \texttt{dom}(\mathcal{H}@\mathit{Here}) \wedge \\ \quad\quad \forall\ \mathit{loc},\ f.\ (\mathit{loc},\ f) \in \texttt{dom}(\mathcal{H}@\mathit{Old}) \wedge (\mathit{loc},\ f) \notin S \Rightarrow \mathcal{H}@\mathit{Old}(\mathit{loc}, f) = \mathcal{H}@\mathit{Here}(\mathit{loc}, f)\end{array}$$

La première propriété $\mathit{Assigns}(\mathcal{H},\ S)$ permet de spécifier que seuls les emplacements mémoire qui appartiennent à l'ensemble S sont modifiés lors de l'exécution du corps de la fonction.

$$\begin{array}{l}\mathit{Allocates}(\mathcal{H},\ S) \stackrel{\text{def}}{=} \\ \quad \forall\ \mathit{loc}.\ \mathit{loc} \notin S \Rightarrow \forall\ f.\ (\mathit{loc},\ f) \in \texttt{dom}(\mathcal{H}@\mathit{Here}) \Leftrightarrow (\mathit{loc},\ f) \in \texttt{dom}(\mathcal{H}@\mathit{Old})\end{array}$$

La seconde propriété, $\mathit{Allocates}(\mathcal{H}, S)$, permet d'établir que les seules allocations effectuées en mémoire sont celles des emplacements qui appartiennent à S.

Les fonctions sémantiques $[\![\_]\!]^{loc}_{\Pi,\ \mathcal{H}}$ et $[\![\_]\!]^{alloc}_{\Pi,\ \mathcal{H}}$ sont définies récursivement dans les figures 3.12 et 3.13. $[\![\_]\!]^{loc}_{\Pi,\ \mathcal{H}}$ associe à une expression syntaxique Loc de locations, un ensemble d'emplacements mémoire qui est l'ensemble des paires $(loc,\ f)$ telles que l'emplacement mémoire loc est accessible à partir de Loc. $[\![\_]\!]^{alloc}_{\Pi,\ \mathcal{H}}$ associe à une expression syntaxique $t_1, \ldots t_n$ d'emplacements alloués, un ensemble de locations loc tel que loc est une adresse accessible à partir de $t_1, \ldots t_n$.

$$
\begin{aligned}
[\![loc_1, \ldots loc_n]\!]^{loc}_{\Pi,\,\mathcal{H}} &= [\![loc_1]\!]^{loc}_{\Pi,\,\mathcal{H}} \cup \cdots \cup [\![loc_n]\!]^{loc}_{\Pi,\,\mathcal{H}} \\
[\![t \to f]\!]^{loc}_{\Pi,\,\mathcal{H}} &= \{[\![t]\!]_{\Pi,\,\mathcal{H}}, f\} \\
[\![\backslash\textbf{nothing}]\!]^{loc}_{\Pi,\,\mathcal{H}} &= \emptyset
\end{aligned}
$$

Et si t a le type **struct** S et $l = [\![t]\!]_{\Pi,\,\mathcal{H}}$ alors,

$$
\begin{aligned}
[\![*t]\!]^{loc}_{\Pi,\,\mathcal{H}} &= \{(l, f) \mid f \in \mathsf{Fields}(S)\} \cup \\
&\qquad \bigcup_{f \in \mathsf{Fields}(S) \text{ de type } \textbf{struct}} \{[\![*l']\!]^{loc}_{\Pi,\,\mathcal{H}} \mid l' = \mathcal{H}@\mathit{Here}(l,\ f)\}
\end{aligned}
$$

FIGURE 3.12 – Évaluation des emplacements mémoire

$$
\begin{aligned}
[\![t_1, \ldots t_n]\!]^{alloc}_{\Pi,\,\mathcal{H}} &= [\![t_1]\!]^{alloc}_{\Pi,\,\mathcal{H}} \cup \cdots \cup [\![t_n]\!]^{alloc}_{\Pi,\,\mathcal{H}} \\
[\![\backslash\textbf{nothing}]\!]^{alloc}_{\Pi,\,\mathcal{H}} &= \emptyset
\end{aligned}
$$

Et si t a le type **struct** S et $l = [\![t]\!]_{\Pi,\,\mathcal{H}}$ alors,

$$
[\![t]\!]^{alloc}_{\Pi,\,\mathcal{H}} = \{l\} \cup \bigcup_{f \in \mathsf{Fields}(S) \text{ de type } \textbf{struct}} \{[\![l \to f]\!]^{alloc}_{\Pi,\,\mathcal{H}}\}
$$

FIGURE 3.13 – Évaluation des emplacements alloués

En résumé, nous présentons une sémantique opérationnelle qui a la particularité d'être bloquante au sens proposé par Herms *et al* [59]. L'exécution d'une expression donnée peut donner lieu à trois comportements possibles. Le premier cas est l'exécution d'un nombre fini d'étapes jusqu'à l'obtention d'une *valeur* v, c'est-à-dire une constante. On parle alors d'une exécution finie. Le deuxième cas est une exécution infinie, autrement dit, l'exécution ne se termine pas. Le troisième cas est une exécution qui bloque si nous essayons d'accéder à un emplacement mémoire qui n'est pas alloué, si une assertion invalide est rencontrée, si l'invariant de boucle n'est pas vérifié au moment où l'on teste la condition de la boucle ou bien si la post-condition de la fonction ou l'une des propriétés $Assigns$ et $Allocates$ construites à partir des clause **writes** et **allocates** n'est pas vérifiée à la fin de l'exécution du corps de la fonction.

L'intérêt d'une telle sémantique est que l'on peut poser la définition suivante :

Définition 3.3.1 Un programme *respecte sa spécification* s'il *s'exécute sans bloquer,* et ceci est valable en particulier pour un programme qui ne se termine pas.

3.4 Calcul de la plus faible précondition

L'étape suivante dans la présentation de notre langage consiste à définir un calcul de plus faible précondition WP. $\mathrm{WP}(e,\ Q)$ associe à une expression e et à une formule Q du langage d'étude, une formule logique d'un langage cible qui est proche mais différent du langage logique source. Ce langage

cible est la logique du premier ordre avec égalité, arithmétique, types polymorphes, etc. Mais ne contient pas de notion de type structure ni d'état mémoire. Grosso modo, on peut voir ce langage comme étant le langage logique implémenté dans Why3. Les accès mémoire sont interprétés dans ce langage par des constructions exprimées dans une théorie qui s'appelle le *modèle mémoire*.

Nous supposerons que cette logique cible connaît déjà la notion d'ensemble polymorphe (type $Set\ \alpha$) et qu'elle est munie des fonctions suivantes :

- $\texttt{mem}(\alpha, Set\ \alpha)$
- $(==)\ (s_1\ s_2 :\ Set\ \alpha) = forall\ x : \alpha.\ \texttt{mem}(x, s_1) \leftrightarrow \texttt{mem}(x, s_2)$
- $\texttt{subset}\ \ (s_1\ s_2 :\ Set\ \alpha) = forall\ x : \alpha.\ \texttt{mem}(x, s_1) \rightarrow \texttt{mem}(x, s_2)$
- $\texttt{empty} :\ Set\ \alpha$
- $\texttt{union} :\ Set\ \alpha,\ Set\ \alpha \rightarrow Set\ \alpha$
- $\texttt{inter} :\ Set\ \alpha,\ Set\ \alpha \rightarrow Set\ \alpha$

3.4.1 Modèle mémoire

Le modèle mémoire que nous choisissons pour notre calcul de plus faible précondition n'est pas celui des greffons Jessie ou WP. C'est un modèle qui est proche de notre sémantique opérationnelle.

Nous commençons par déclarer trois types logiques : $location$ pour les emplacements mémoire, $field$ pour les champs des types structure et $value$ pour les valeurs, ainsi que les constantes : $null$ et f_i pour les noms des champs des types structure.

$$\begin{array}{l}
\textbf{type}\ location \\
\textbf{type}\ field \\
\textbf{type}\ value = Int\ int \mid Bool\ bool \mid Void \mid Loc\ of\ location \\
null : location \\
f_1,\ f_2, \ldots : field
\end{array}$$

Nous pouvons maintenant définir le type $heap$ des tas mémoire ainsi que les fonctions qui permettent de gérer la mémoire. Chaque label du langage source va être interprété par une variable logique de type $heap$ dans le langage logique cible. Le type $heap$ est abstrait et nous axiomatisons son comportement comme celui d'une fonction partielle de domaine fini allant des paires de type $(location,\ field)$ vers les valeurs.

$$\begin{array}{l}
\textbf{type}\ heap \\
\texttt{dom} :\ heap \rightarrow Set\ location \\
\texttt{select} :\ heap, location, field \rightarrow value \\
\texttt{store} :\ heap, location, field, value \rightarrow heap \\
\texttt{valid} :\ heap, location \rightarrow Prop \\
\texttt{alloc} :\ heap \rightarrow location * heap
\end{array}$$

Le comportement de ces fonctions est axiomatisé par les axiomes suivants, où h une variable de type $heap$, $l,\ l'$ des emplacements mémoire ($location$), f un champ défini dans un type structure ($field$) et

v une valeur ($value$) :

$$\begin{array}{ll}
\textbf{Axiome 1}: & \texttt{mem}(l, \texttt{dom}(h)) \Rightarrow \texttt{select}(\texttt{store}(h, l, f, v), l, f) = v \\
\textbf{Axiome 2}: & \texttt{mem}(l, \texttt{dom}(h)) \Rightarrow l \neq l' \vee f \neq f' \Rightarrow \\
 & \qquad \texttt{select}(\texttt{store}(h, l', f', v), l, f) = \texttt{select}(h, l, f) \\
\textbf{Axiome 3}: & \texttt{dom}(\texttt{store}(h, l, f, v)) = union(\texttt{dom}(h), \{l\}) \\
\textbf{Axiome 4}: & \texttt{valid}(\texttt{store}(h, l, f, v), l') = l = l' \ \vee \ \texttt{valid}(h, l') \\
\textbf{Axiome 5}: & \textbf{fst}(\texttt{alloc}(h)) \notin \texttt{dom}(h) \\
\textbf{Axiome 6}: & \texttt{dom}(\textbf{snd}(\texttt{alloc}(h))) = union(\texttt{dom}(h), \textbf{fst}(\texttt{alloc}(h))) \\
\textbf{Axiome 7}: & \texttt{mem}(l, \texttt{dom}(h)) \Rightarrow \texttt{select}(\textbf{snd}(\texttt{alloc}(h)), l, f) = \texttt{select}(h, l, f) \\
\textbf{Axiome 8}: & l = \textbf{fst}(\texttt{alloc}(h)) \Rightarrow \texttt{select}(\textbf{snd}(\texttt{alloc}(h)), l, f) = default(type(f))
\end{array}$$

Nous modélisons également les ensembles de locations apparaissant dans la clause **writes** par un autre type abstrait $locSet$ muni d'une fonction spécifique de test d'appartenance, car ces ensembles ne sont pas forcément finis.

$$\begin{array}{ll}
\textbf{type}\ locSet = & Union\ locSet\ locSet \mid Field\ location\ field \mid Star\ location \\
reachable: & heap,\ location,\ field,\ locSet \rightarrow Prop
\end{array}$$

Les axiomes qui définissent le comportement du prédicat $reachable$ sont :

$$\begin{array}{ll}
\textbf{Axiome 1}: & reachable(h, l, f, Union(s_1, s_2)) \Leftrightarrow \\
 & \qquad reachable(h, l, f, s_1) \vee reachable(h, l, f, s_2) \\
\textbf{Axiome 2}: & reachable(h, l, f, Field(l_1, f_1)) \Leftrightarrow l = l_1 \wedge f = f_1 \\
\textbf{Axiome 3}: & reachable(h, l, f, Star(l_1)) \Leftrightarrow \\
 & \qquad l = l_1 \vee \bigvee_{g \text{ de type structure}} reachable(h, l, f, Star(\texttt{select}(h, l_1, g)))
\end{array}$$

De même, nous modélisons également les ensembles de locations apparaissant dans la clause **allocates** par un autre type abstrait $allocSet$ muni d'une fonction spécifique de test d'appartenance.

$$\begin{array}{ll}
\textbf{type}\ allocSet = & Union\ allocSet\ allocSet \mid Term\ location \\
reachableAlloc: & heap,\ location,\ allocSet \rightarrow Prop
\end{array}$$

Les axiomes qui définissent le comportement du prédicat $reachableAlloc$ sont :

$$\begin{array}{ll}
\textbf{Axiome 1}: & reachableAlloc(h, l, Union(s_1, s_2)) \Leftrightarrow \\
 & \qquad reachableAlloc(h, l, s_1) \vee reachableAlloc(h, l, s_2) \\
\textbf{Axiome 2}: & reachableAlloc(h, l, Term(l_1)) \Leftrightarrow \\
 & \qquad l = l_1 \vee \bigvee_{g \text{ de type structure}} reachableAlloc(h, l, Term(\texttt{select}(h, l_1, g)))
\end{array}$$

3.4.2 Traduction des formules vers le langage logique cible

Les formules de notre langage source doivent être interprétées dans notre langage logique cible, dans lequel les labels du langage source deviennent des variables. Nous définissons alors deux variables logiques particulières *Here* et *Old* qui correspondent aux labels du même nom. Autrement dit, étant

donné un tas mémoire $\mathcal{H}$, les variables *Here* et *Old* correspondent à $\mathcal{H}@Here$ et $\mathcal{H}@Old$. Nous notons $\overline{p}$ la traduction de la formule p écrite dans la logique de notre langage d'étude. Cette fonction est définie récursivement comme indiqué ci-dessous :

$$
\begin{array}{lcl}
\overline{\neg p} & = & \neg\overline{p} \\
\overline{p_1\ log\_op\ p_2} & = & \overline{p_1}\ log\_op\ \overline{p_2} \\
\overline{\texttt{let}\ x = t\ \texttt{in}\ p} & = & \texttt{let}\ x = \overline{t}\ \texttt{in}\ \overline{p} \\
\overline{\backslash\textbf{forall}\ \tau_1 x_1 \ldots \tau_n x_n\ ;\ p} & = & \backslash forall\ \tau_1 x_1 \ldots \tau_n x_n\ ;\ \overline{p} \\
\overline{\backslash\textbf{exists}\ \tau_1 x_1 \ldots \tau_n x_n\ ;\ p} & = & \backslash exists\ \tau_1 x_1 \ldots \tau_n x_n\ ;\ \overline{p} \\
\overline{\textbf{valid}(t,\ L)} & = & \texttt{valid}(L, \overline{t}) \\
\overline{\textbf{valid}(t)} & = & \texttt{valid}(\textit{Here},\ \overline{t}) \\
\overline{p(t_1, \ldots, t_n)} & = & p(\overline{t_1}, \ldots, \overline{t_n})
\end{array}
$$

et les termes sont traduits de la façon suivante :

$$
\begin{array}{lcl}
\overline{v} & = & v \\
\overline{x} & = & x \\
\overline{un\_op\ t} & = & un\_op\ \overline{t} \\
\overline{t_1\ bin\_op\ t_2} & = & \overline{t_1}\ bin\_op\ \overline{t_2} \\
\overline{t_1\ rel\_op\ t_2} & = & \overline{t_1}\ rel\_op\ \overline{t_2} \\
\overline{t \overset{L}{\rightarrow} f} & = & \texttt{select}(L,\ \overline{t},\ f)
\end{array}
$$

Les hypothèses $Assigns(\mathcal{H},\ S)$ et $Allocates(\mathcal{H},\ S)$ utilisées dans la sémantique opérationnelle de l'expression « return » (section 3.3.4) vont être interprétées par les prédicats explicites suivants :

$$
\begin{array}{l}
assigns : heap,\ heap, locSet \\
assigns(h_1, h_2, S) \overset{\text{def}}{=} \\
\quad \texttt{subset}(\texttt{dom}(h_1),\ \texttt{dom}(h_2)) \wedge \\
\quad\quad \forall\ loc : location,\ f : field,\ \texttt{valid}(h_1,\ loc)\ \wedge \textbf{not}\ reachable(h_1, loc, f,\ S) \Rightarrow \\
\quad\quad\quad \texttt{select}(h_1, loc, f) = \texttt{select}(h_2, loc, f)
\end{array}
$$

$$
\begin{array}{l}
allocates : heap,\ heap, AllocSet \\
allocates(h_1, h_2, S) \overset{\text{def}}{=} \forall\ loc : location \\
\quad \textbf{not}\ reachableAlloc(h_2, loc,\ S) \Rightarrow (\texttt{valid}(h_1,\ loc) \Leftrightarrow\ \texttt{valid}(h_2,\ loc))
\end{array}
$$

Nous posons alors deux lemmes qui établissent que l'évaluation d'un terme t dans la logique de notre langage se ramène à l'évaluation de la traduction $\overline{t}$ du terme t dans la logique dans lequel le calcul de plus faible précondition est défini.

Lemme 3.4.1 *Soit t un terme écrit dans la logique de notre langage et $\overline{t}$ le résultat de la traduction de t dans la logique utilisée pour la définition du calcul de plus faible précondition. On a alors la propriété suivante :*

$$[\![t]\!]_{\Pi,\ \mathcal{H}} = [\![\ \overline{t}\ ]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$$

Preuve. La preuve de ce lemme se fait par récurrence sur la structure du terme. Seul le cas d'accès à un champ est modifié par la traduction. L'évaluation d'un tel terme dans un état $(\Pi,\ \mathcal{H})$ est égale à $\mathcal{H}@l([\![t]\!]_{\Pi,\ \mathcal{H}},\ f)$, c'est-à-dire la valeur associée à la paire $([\![t]\!]_{\Pi,\ \mathcal{H}},\ f)$ dans $\mathcal{H}@L$. Ainsi, si on étend

l'environnement Π en associant à chaque variable logique L, le contenu de $\mathcal{H}@L$ alors l'évaluation de $\bar{t}$ dans l'environnement étendu est égale à la valeur associée à la variable ($[\![t]\!]_{\Pi,\ \mathcal{H}},\ f$), autrement dit, $\Pi'([\![t]\!]_{\Pi,\ \mathcal{H}},\ f)$ où $\Pi' = \{L \leftarrow \mathcal{H}@L\} \cdot \Pi$

Dans les autres cas, la preuve du lemme est triviale.

□

Lemme 3.4.2 *Soit P une formule écrite dans la logique de notre langage et $\overline{P}$ le résultat de la traduction de P dans la logique utilisée pour la définition du calcul de plus faible précondition. On a alors la propriété suivante :*

$$[\![P]\!]_{\Pi,\ \mathcal{H}} \Leftrightarrow [\![\ \overline{P}\ ]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}$$

Preuve. Comme pour les termes, il suffit de faire une récurrence sur la structure des formules.

Le seul cas non trivial est celui du prédicat $valid$. L'évaluation du prédicat $valid(t,\ L)$ dans l'état mémoire $(\Pi,\ \mathcal{H})$ est vraie si et seulement si $[\![t]\!]_{\Pi,\ \mathcal{H}}$ est alloué dans le tas mémoire $\mathcal{H}$ au point L. Ainsi, si L est la variable logique à laquelle est associé le contenu de $\mathcal{H}@L$, dans l'environnement Π, alors $[\![\bar{t}]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}$ appartient au domaine de L, autrement dit $[\![\texttt{valid}(\bar{t},\ L)]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}$ est vraie. □

3.4.3 Définition du calcul de plus faible précondition

Définition 3.4.3 *Le calcul de plus faible précondition est une fonction qui à une expression e et une formule Q associe une nouvelle formule $\mathrm{WP}(e,\ Q)$. Il s'agit d'une définition par récurrence sur e. La propriété habituelle attendue est que si l'on exécute e dans un état qui vérifie $\mathrm{WP}(e,\ Q)$, alors on atteint un état final qui vérifie Q.*

Notre définition du calcul de plus faible précondition suit le schéma classique d'un tel calcul dont les règles sont énumérées ci-dessous. Ces règles sont regroupées en quatre parties, ce découpage permet une meilleure structuration des preuves des lemmes et des théorèmes que nous allons énoncer, dans le sens où il permet d'éviter les redondances.

Partie 1 : Pour toute expression pure, c'est à dire sans effets de bord, autre que l'accès à un champ.

$$\mathrm{WP}(e,\ Q) = Q[\backslash\mathsf{result} \leftarrow e]$$

On rappelle (section 3.1, page 52) que $\backslash$result est une variable logique spéciale qui désigne le résultat d'une expression.

Partie 2 : L'expression *let* :

$$\mathrm{WP}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2,\ Q) = \mathrm{WP}(e_1,\ \mathrm{WP}(e_2,\ Q)[x \leftarrow \backslash\mathsf{result}])$$

Partie 3 : Les constructions du langage :

– **Expression labelisée** :

$$\mathrm{WP}(L:\ e,\ Q) = \mathrm{WP}(e,\ Q)[L \leftarrow \mathit{Here}]$$

La notation $L \leftarrow \mathit{Here}$ est une abréviation pour $\mathtt{store}(\mathit{Here}, v, f, \mathtt{select}(L, v, f))$ pour toute paire (v, f) telle que $(v, f) \in \mathtt{dom}(L)$

– **Séquence d'expressions** :

$$\mathrm{WP}(e_1; e_2,\ Q) = \mathrm{WP}(e_1, \mathrm{WP}(e_2,\ Q))$$

– **Expression conditionnelle** :

$$\begin{aligned}\mathrm{WP}(&\mathtt{if}\ v_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3,\ Q) = \\ &(v_1 = \mathit{true} \Rightarrow \mathrm{WP}(e_2,\ Q))\ \wedge (v_1 = \mathit{false} \Rightarrow \mathrm{WP}(e_3,\ Q))\end{aligned}$$

– **Expression itérative** :

$$\begin{aligned}\mathrm{WP}(&\mathtt{while}\ v_1\ \mathtt{invariant}\ I\ \mathtt{do}\ e_2\ \mathtt{end},\ Q) = \\ &\overline{I} \wedge \forall \mathit{Here}.(\overline{I} \Rightarrow (v_1 = \mathit{true} \Rightarrow \mathrm{WP}(e_2, \overline{I})) \wedge (v_1 = \mathit{false} \Rightarrow Q))[\mathit{Old} \leftarrow \mathit{Here}]\end{aligned}$$

Le cas de l'expression itérative introduit une quantification universelle sur les effets de bord du corps de la boucle e_2. Ce qui revient, dans notre cas, à quantifier universellement sur la variable logique *Here* et de substituer la variable *Old* par *Here* car c'est le seul état de la mémoire sur lequel les effets de bord d'une mise à jour sont visibles.

– **Accès à un champ** :

$$\mathrm{WP}(v \rightarrow f,\ Q) = \mathtt{valid}(\mathit{Here},\ v) \wedge Q[\backslash\mathsf{result} \leftarrow \mathtt{select}(\mathit{Here}, v, f)]$$

– **Mise à jour d'un champ** :

$$\mathrm{WP}(v_1 \rightarrow f = v_2,\ Q) = \mathtt{valid}(\mathit{Here},\ v_1) \wedge Q[\backslash\mathsf{result} \leftarrow (), \mathit{Here} \leftarrow \mathtt{store}(\mathit{Here}, v_1, f, v_2)]$$

– **Allocation** :

$$\mathrm{WP}(\mathtt{new}\ \ S,\ Q) = \mathtt{let}\ l = \mathtt{alloc}(\mathit{Here})\ \mathtt{in}\ Q[\backslash\mathsf{result} \leftarrow \mathbf{fst}(l),\ \mathit{Here} \leftarrow \mathbf{snd}(l)]$$

– **Assertion** :

$$\mathrm{WP}(\mathtt{assert}\ \ P,\ Q) = \overline{P} \wedge Q = \overline{P} \wedge (\overline{P} \Rightarrow Q)$$

La seconde version est utilisée dans les outils en pratique. Ceci permet que P soit en hypothèse des autres obligations de preuve.

– **Appel de fonction** :

$$\begin{aligned}\mathrm{WP}(&f(v_1, \ldots, v_n),\ Q) = \overline{\mathit{Pre}}[x_i \leftarrow v_i] \wedge \\ &(\forall\, \mathit{Here}.\ \overline{\mathit{Post}}[x_i \leftarrow v_i] \wedge \mathit{assigns}(W) \wedge \mathit{allocates}(A) \Rightarrow Q)[\mathit{Old} \leftarrow \mathit{Here}]\end{aligned}$$

– **L'expression** $return$:

$$\begin{aligned}\mathrm{WP}(&\mathit{return}(v,\ P,\ W,\ A,\ \Pi,\ \{x_i \leftarrow v_i\}),\ Q) = \\ &(\overline{P}[x_i \leftarrow v_i] \wedge \mathit{assigns}(W) \wedge \mathit{allocates}(A) \wedge Q[l_i \leftarrow \Pi(l_i)])[\backslash\mathsf{result} \leftarrow v]\end{aligned}$$

Partie 4 : Enfin les expressions dont le calcul de plus faible précondition se ramène à celui du *let* :

– **Opérations binaires** :

$$\begin{aligned}\mathrm{WP}(e_1\ op\ e_2,\ Q) &= \mathrm{WP}(\texttt{let}\ v_1 = e_1\ \texttt{in let}\ v_2 = e_2\ \texttt{in}\ v_1 op\ v_2,\ Q)\\ \mathrm{WP}(v_1\ op\ e_2,\ Q) &= \mathrm{WP}(\texttt{let}\ v_2 = e_2\ \texttt{in}\ v_1 op\ v_2,\ Q)\end{aligned}$$

– **Opérations unaires** :

$$\mathrm{WP}(un\_op\ e,\ Q) = \mathrm{WP}(\texttt{let}\ v = e\ \texttt{in}\ un\_op\ v,\ Q)$$

– **Expression conditionnelle** :

$$\mathrm{WP}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ Q) = \mathrm{WP}(\texttt{let}\ v_1 = e_1\ \texttt{in if}\ v_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ Q)$$

– **Expression itérative** :

$$\mathrm{WP}(\texttt{while}\ e_1\ \texttt{invariant}\ I\ \texttt{do}\ e_2\ \texttt{end},\ Q) = \mathrm{WP}(\texttt{let}\ v_1 = e_1\ \texttt{in while}\ v_1\ \texttt{invariant}\ I\ \texttt{do}\ e_2\ \texttt{end},\ Q)$$

– **Accès à un champ** :

$$\mathrm{WP}(e \to f,\ Q) = \mathrm{WP}(\texttt{let}\ v = e\ \texttt{in}\ v \to f,\ Q)$$

– **Mise à jour d'un champ** :

$$\begin{aligned}\mathrm{WP}(e_1 \to f = e_2,\ Q) &= \mathrm{WP}(\texttt{let}\ v_1 = e_1\ \texttt{in let}\ v_2 = e_2\ \texttt{in}\ v_1 \to f = v_2,\ Q)\\ \mathrm{WP}(v_1 \to f = e_2,\ Q) &= \mathrm{WP}(\texttt{let}\ v_2 = e_2\ \texttt{in}\ v_1 \to f = v_2,\ Q)\end{aligned}$$

– **Appel de fonction** :

$$\begin{aligned}&\mathrm{WP}(f(v_1, \ldots, v_i, e_{i+1}, \ldots, e_n),\ Q) =\\ &\quad \mathrm{WP}(\texttt{let}\ v_{i+1} = e_{i+1}\ \texttt{in}\ \ldots \texttt{let}\ v_n = e_n\ \texttt{in}\ f(v_1, \ldots, v_i, v_{i+1}, \ldots, v_n),\ Q)\end{aligned}$$

3.4.4 Lemmes auxiliaires sur le calcul de plus faible précondition

Il existe dans la littérature des études théoriques avancées sur la famille des transformateurs de prédicats en général, dont le calcul de plus faible précondition fait partie [14]. Des propriétés sont énoncées souvent sans forcément montrer leur utilité. Pour notre étude, nous allons énoncer et prouver des propriétés qui servent ensuite à la preuve de correction.

La première propriété est classiquement nommée *monotonie* du calcul.

Lemme 3.4.4 (Monotonie) *Soit une expression e et deux formules P et Q, si P implique Q, alors la plus faible précondition de e calculée à partir de P implique la plus faible précondition calculée à partir de Q, plus précisément, si pour tout environnement Π, $[\![(P \Rightarrow Q)]\!]_\Pi$ est valide, alors pour tout environnement Π, $[\![\mathrm{WP}(e,\ P) \Rightarrow \mathrm{WP}(e,\ Q)]\!]_\Pi$ est valide.*

Avant de détailler cette preuve, notez que la quantification sur tous les Π est essentielle : il n'est pas vrai que $[\![P \Rightarrow Q]\!]_\Pi$ entraîne $[\![\mathrm{WP}(e, P) \Rightarrow \mathrm{WP}(e, Q)]\!]_\Pi$ pour un Π fixé. Voici un contre-exemple : dans un état où $v \overset{\mathit{Here}}{\to} f = 42$, c'est-à-dire si Π est tel que $\texttt{select}(\Pi[\mathit{Here}],\ v,\ f) = 42$, alors ($\mathbf{true} \Rightarrow v \overset{\mathit{Here}}{\to} f = 42$) est valide dans Π ; mais $\mathrm{WP}(v \to f = 7, \mathbf{true}) = \mathbf{true}$ n'implique pas la formule $\mathrm{WP}(v \to f = 7, v \overset{\mathit{Here}}{\to} f = 42)$ qui est équivalente à $(7 = 42)$.

Preuve. La preuve de cette propriété de monotonie se fait par récurrence sur la structure de e. Cette preuve est décomposée en quatre parties. Cette décomposition correspond à celle faite lors de la définition du calcul de plus faible précondition.

Nous avons l'hypothèse qui dit que pour toutes formules P et Q et pour tout état Π,

$$[\![P \Rightarrow Q]\!]_\Pi \text{ est valide} \tag{3.1}$$

Partie 1 : e est une expression pure.

La propriété (3.1) reste vraie si on substitue dans chacune des formules P et Q les instances de $\backslash$result par la même expression e. Ainsi $[\![(P[\backslash\mathsf{result} \leftarrow e]) \Rightarrow (Q[\backslash\mathsf{result} \leftarrow e])]\!]_\Pi$, c'est-à-dire $[\![\mathrm{WP}(e, P) \Rightarrow \mathrm{WP}(e, Q)]\!]_\Pi$, est valide.

Partie 2 : e est une expression «let» de la forme $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$

Dans ce cas, étant donnée la propriété (3.1), l'hypothèse de récurrence sur e_2 implique que

$$[\![(\mathrm{WP}(e_2,\ P)[x \leftarrow \backslash\mathsf{result}]) \Rightarrow (\mathrm{WP}(e_2,\ Q)[x \leftarrow \backslash\mathsf{result}])]\!]_\Pi$$

est valide. Ensuite, l'hypothèse de récurrence sur e_1, nous permet de déduire que

$$[\![\mathrm{WP}(e_1,\ \mathrm{WP}(e_2,\ P)[x \leftarrow \backslash\mathsf{result}]) \Rightarrow \mathrm{WP}(e_1,\ \mathrm{WP}(e_2,\ Q)[x \leftarrow \backslash\mathsf{result}])]\!]_\Pi$$

qui est équivalente, par définition, à $[\![\mathrm{WP}(e,\ P) \Rightarrow \mathrm{WP}(e,\ Q)]\!]_\Pi$, est valide.

Partie 3 : Les constructions du langage.

- e est une séquence d'expressions de la forme $e = e_1; e_2$.
  D'après la propriété (3.1) et l'hypothèse de récurrence sur la sous-expression e_2, on sait que $[\![\mathrm{WP}(e_2,\ P) \Rightarrow \mathrm{WP}(e_2,\ Q)]\!]_\Pi$ est valide. Dans ce cas, l'hypothèse de récurrence sur e_1 nous garantit alors que $[\![\mathrm{WP}(e_1, \mathrm{WP}(e_2,\ P)) \Rightarrow \mathrm{WP}(e_1, \mathrm{WP}(e_2,\ Q))]\!]_\Pi$, c'est-à-dire

  $$[\![\mathrm{WP}(e_1; e_2,\ P) \Rightarrow \mathrm{WP}(e_1; e_2,\ Q)]\!]_\Pi$$

  est également valide.

- e est une expression conditionnelle de la forme $e = \texttt{if}\ v_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$.
  D'après les hypothèses de récurrence sur les sous expressions e_2 et e_3 et la propriété (3.1), les évaluations suivantes sont valides.

  $$[\![\mathrm{WP}(e_2,\ P) \Rightarrow \mathrm{WP}(e_2,\ Q)]\!]_\Pi \tag{3.2}$$
  $$[\![\mathrm{WP}(e_3,\ P) \Rightarrow \mathrm{WP}(e_3,\ Q)]\!]_\Pi \tag{3.3}$$

  On procède ensuite par cas sur la valeur de v_1. Étant de type bool, cette dernière ne peut donc prendre que deux valeurs.
  - $[\![v_1]\!]_\Pi =$ **true**. Alors, pour toute formule P

    $$\mathrm{WP}(\texttt{if}\ v_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ P) = \mathrm{WP}(e_2,\ P)$$

  - $[\![v_1]\!]_\Pi =$ **false**. Alors, pour toute formule P

    $$\mathrm{WP}(\texttt{if}\ v_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ P) = \mathrm{WP}(e_3,\ P)$$

Ainsi, d'après (3.2) et (3.3), on peut déduire que

$$[\![\mathrm{WP}(\texttt{if } v_1 \texttt{ then } e_2 \texttt{ else } e_3,\ P) \Rightarrow \mathrm{WP}(\texttt{if } v_1 \texttt{ then } e_2 \texttt{ else } e_3,\ Q)]\!]$$

est valide dans Π.

– e est une expression itérative de la forme $e = \texttt{while } v_1 \texttt{ invariant } I \texttt{ do } e_2 \texttt{ end}$.
Étant donnée l'hypothèse (3.1), si

$$F_1 = (v_1 = \textbf{true} \Rightarrow \mathrm{WP}(e_2,\ I) \wedge v_1 = \textbf{false} \Rightarrow P)$$

et

$$F_2 = (v_1 = \textbf{true} \Rightarrow \mathrm{WP}(e_2,\ I) \wedge v_1 = \textbf{false} \Rightarrow Q)$$

alors, $[\![F_1 \Rightarrow F_2]\!]$ est valide dans n'importe quel environnement Π. Ce qui est également le cas de $[\![(I \Rightarrow F_1) \Rightarrow (I \Rightarrow F_2)]\!]$. Nous pouvons ensuite appliquer la propriété qui dit que

$$\text{Si } [\![P \Rightarrow Q]\!] \text{ alors } [\![(\forall\ \vec{w}, P) \Rightarrow (\forall\ \vec{w}, Q)]\!]$$

Cette propriété peut être vue comme une conséquence de deux propriétés : d'une part si $[\![f]\!]$ alors $[\![\forall\ \vec{w}, f]\!]$, et d'autre part si $\forall\ \vec{w}, (P \Rightarrow Q)$ alors $(\forall\ \vec{w}, P) \Rightarrow (\forall\ \vec{w}, Q)$. Ainsi, la formule

$$[\![I \wedge \forall\ \mathit{Here}, (I \Rightarrow F_1)[\mathit{Old} \leftarrow \mathit{Here}] \Rightarrow I \wedge \forall\ \mathit{Here}, (I \Rightarrow F_2)[\mathit{Old} \leftarrow \mathit{Here}]]\!]$$

est valide dans tout environnement Π.

– e est un accès à un champ de la forme $e = v \xrightarrow{l} f$.
La propriété (3.1) est toujours vérifiée si on remplace la même variable logique $\backslash$result, dans chacune des deux formules P et Q, par la même valeur $\texttt{select}(\Pi(l), v, f)$. Autrement dit,

$$[\![(P[\backslash\text{result} \leftarrow \texttt{select}(\Pi(l), v, f)]) \Rightarrow (Q[\backslash\text{result} \leftarrow \texttt{select}(\Pi(l), v, f)])]\!]$$

ou par définition $[\![\mathrm{WP}(e,\ P) \Rightarrow \mathrm{WP}(e,\ Q)]\!]_\Pi$ est valide.

– e est une mise à jour de la valeur d'un champ de la forme $v_1 \rightarrow f = v_2$.
La preuve est similaire à celle du cas de l'accès à un champ. Sauf qu'en plus de substituer la variable $\backslash$result par $()$ dans P et Q, on met-à-jour la variable *Old* avec *Here* et l'emplacement mémoire $\texttt{select}(\Pi(l),\ v_1,\ f)$ avec la valeur v_2.

– e est une allocation de la forme $e = \texttt{new } S$.
On a, d'après la définition du calcul de plus faible précondition ;

$$\mathrm{WP}(\texttt{new } S,\ P) \quad = \quad \texttt{let } l = \texttt{alloc}(\mathit{Here}) \texttt{ in } P[\backslash\text{result} \leftarrow \textbf{fst}(l),\ \mathit{Here} \leftarrow \textbf{snd}(l)]$$

On veut alors prouver que l'évaluation de la formule

$$\begin{aligned} &\texttt{let } l_1 = \texttt{alloc}(\mathit{Here}) \texttt{ in } P[\backslash\text{result} \leftarrow \textbf{fst}(l_1),\ \mathit{Here} \leftarrow \textbf{snd}(l_1)] \Rightarrow \\ &\quad \texttt{let } l_2 = \texttt{alloc}(\mathit{Here}) \texttt{ in } Q[\backslash\text{result} \leftarrow \textbf{fst}(l_2),\ \mathit{Here} \leftarrow \textbf{snd}(l_2)] \end{aligned} \tag{3.4}$$

est valide dans n'importe quel environnement Π.
`alloc` est une fonction logique, elle retourne alors la même valeur lorsqu'on l'appelle avec le même paramètre *Here*. Ainsi, l_1 et l_2 sont égales. La formule (3.4) s'écrit alors,

$$\texttt{let } l_1 = \texttt{alloc}(\mathit{Here}) \texttt{ in } (P \Rightarrow Q)[\backslash\text{result} \leftarrow \textbf{fst}(l_1),\ \mathit{Here} \leftarrow \textbf{snd}(l_1)] \tag{3.5}$$

D'après l'hypothèse, quelle que soit l'environnement Π, $[\![P \Rightarrow Q]\!]_\Pi$ est valide. En particulier, lorsque les variables logiques $\backslash$result et *Here* sont, respectivement, substituées par $\textbf{fst}(l_1)$ un emplacement mémoire fraîchement alloué et $\textbf{snd}(l_1)$ la nouvelle valeur de type $heap$ associée à *Here*. Par conséquent, l'évaluation de la formule (3.5) dans un tel environnement, qui est équivalente à $[\![\mathrm{WP}(\texttt{new } S,\ P) \Rightarrow \mathrm{WP}(\texttt{new } S,\ Q)]\!]_\Pi$ est valide.

– e est une assertion de la forme $e =$ `assert` F.
Ce cas est trivial. En effet, si $[\![P \Rightarrow Q]\!]_\Pi$ est valide, alors si $\overline{F}$ est la traduction de la formule F, alors

$$[\![(\overline{F} \wedge (\overline{F} \Rightarrow P)) \Rightarrow (\overline{F} \wedge (\overline{F} \Rightarrow Q))]\!]_\Pi$$

c'est-à-dire

$$[\![\mathrm{WP}(\texttt{assert}\ F,\ P) \Rightarrow \mathrm{WP}(\texttt{assert}\ F,\ Q)]\!]_\Pi$$

l'est aussi.

– e est un appel de fonction $e = f(v_1, \ldots, v_n)$.
Comme dans le cas du `assert` , à partir de la propriété (3.1), on peut déduire que

$$\begin{array}{l}[\![(\overline{Post_f}[x_i \leftarrow v_i] \wedge assigns(W) \wedge allocates(A) \Rightarrow P) \Rightarrow \\ \qquad (\overline{Post_f}[x_i \leftarrow v_i] \wedge assigns(W) \wedge allocates(A) \Rightarrow Q)]\!]_\Pi\end{array}$$

est valide, où $Post_f$ est la post-condition de la fonction f, W et A les ensembles des emplacements mémoire, modifiables et fraîchement alloués par f et les x_i (respectivement v_i) pour $1 \leq i \leq n$ ses paramètres formels (respectivement effectifs). Cette propriété est vérifiée quels que soient les effets de bord de la fonction. Autrement dit, quelque soit la nouvelle valeur de la variable logique *Here* dans l'environnement Π. Ainsi, la formule

$$\begin{array}{l}[\![\forall\ \mathit{Here}.(\overline{Post_f}[x_i \leftarrow v_i] \wedge assigns(W) \wedge allocates(A) \Rightarrow P)[\mathit{Old} \leftarrow \mathit{Here}] \Rightarrow \\ \quad \forall\ \mathit{Here}.(\overline{Post_f}[x_i \leftarrow v_i] \wedge assigns(W) \wedge allocates(A) \Rightarrow Q)[\mathit{Old} \leftarrow \mathit{Here}]]\!]\end{array}$$

reste valide dans Π. Ce qui est encore le cas lorsqu'on "ajoute à chaque partie de l'implication" la conjonction avec la formule $Pre_f[x_i \leftarrow v_i]$. On a alors prouvé que

$$[\![\mathrm{WP}(f(v_1, \ldots, v_n),\ P) \Rightarrow \mathrm{WP}(f(v_1, \ldots, v_n),\ Q)]\!]$$

est valide dans n'importe quel environnement local Π.

– e est une expression de la forme $e = return(v,\ F,\ W,\ A,\ \Pi,\ \Pi_{prm})$.
Ce cas est également trivial. Comme pour l'expression `assert` , si $[\![P \Rightarrow Q]\!]$ est valide dans l'environnement Π, alors pour toute formule G, $[\![G \wedge P \Rightarrow G \wedge Q]\!]$ est également valide dans ce même environnement. En particulier, pour

$$G = ((\overline{F}[x_i \leftarrow v_i]\ \wedge\ assigns(W)\ \wedge\ allocates(A))[\backslash\mathsf{result} \leftarrow v])$$

Cette propriété reste vraie, si on substitue dans chacune des formules P et Q les variables libres par la valeur qui leur est associée dans l'environnement $Stack$, c'est-à-dire : $P[l_i \leftarrow [\![l_i]\!]_{\Pi,\ \mathcal{H}}]$ et $Q[m_j \leftarrow [\![m_j]\!]_{\Pi,\ \mathcal{H}}]$. Par consé, $[\![\mathrm{WP}(e,\ P) \Rightarrow \mathrm{WP}(e,\ Q)]\!]_\Pi$ est valide.

Partie 4 : Ce cas regroupe :
– Les opérations unaires de la forme $op\ e_1$
– Les opérations binaires de la forme $e_1 op\ e_2$
– Les opérations binaires de la forme $v_1 op\ e_2$ où v_1 est une valeur.
– Les expressions conditionnelles de la forme `if` e_1 `then` e_2 `else` e_3
– Les expressions itératives de la forme `while` e_1 `invariant` I `do` e_2 `end`
– Les accès à un champ de la forme $e_1 \rightarrow f$
– Les mises à jour de la valeur d'un champ de la forme $e_1 \rightarrow f = e_2$
– Les appels de fonctions de la forme $f(v_1, \ldots, v_i, e_{i+1}, \ldots, e_n)$

Les règles de calcul de la plus faible précondition de ces expressions sont dérivées de celle du «let». Par conséquent, la preuve de la monotonie dans ces cas se rapporte à celle du «let». □

Lemme 3.4.5 (Monotonie généralisée) *Si pour tout environnement* Π*, les deux formules*

$$[\![\forall\, \mathit{Here}, (P \Rightarrow Q)]\!]_\Pi$$

et

$$[\![\mathrm{WP}(e,\ P)]\!]_\Pi$$

sont valides, alors pour tout environnement Π*,*

$$[\![\mathrm{WP}(e,\ Q)]\!]_\Pi$$

est à son tour valide.

Preuve. La preuve de cette propriété se fait par récurrence structurelle sur e. Elle est similaire à celle du lemme précédent de monotonie, sauf pour le cas de la mise à jour du champ. Nous savons que $[\![\mathrm{WP}(v_1 \leftarrow f = v_2,\ P)]\!]$ et donc

$$[\![\mathbf{valid}(v_1) \wedge P[\backslash\mathsf{result} \leftarrow (), \mathit{Here} \leftarrow \mathtt{store}(\mathit{Here}, v_1, f, v_2)]]\!]$$

est valide dans l'environnement Π, c'est-à-dire que $[\![P]\!]$ est valide dans l'environnement Π dans lequel *Here* est substituée par $\mathtt{store}(\mathit{Here}, v_1, f, v_2)$. Or d'après l'hypothèse du lemme, $P \Rightarrow Q$ pour toutes les valeurs possibles de *Here*. En particulier, pour $\mathtt{store}(\mathit{Here}, v_1, f, v_2)$ Par conséquent,

$$[\![Q[\backslash\mathsf{result} \leftarrow (), \mathit{Here} \leftarrow \mathtt{store}(\mathit{Here}, v_1, f, v_2)]]\!]$$

et donc $[\![\mathrm{WP}(v_1 \leftarrow f = v_2,\ Q)]\!]$ est valide dans l'environnement Π.

La seconde propriété qui va nous servir par la suite est la *distributivité de la conjonction*.

Lemme 3.4.6 (Distributivité de la conjonction) *Pour toute expression* e *et pour toutes formules* P *et* Q*, la plus faible précondition de* e *calculée à partir de la conjonction des formules* P *et* Q *est équivalente à la conjonction des plus faibles préconditions calculées à partir de* P *et de* Q*, plus précisément, pour tout environnement* Π*,*

$$[\![\mathrm{WP}(e,\ P)]\!]_\Pi \wedge [\![\mathrm{WP}(e,\ Q)]\!]_\Pi \Rightarrow [\![\mathrm{WP}(e,\ P \wedge Q)]\!]_\Pi$$

Preuve. Comme pour le lemme de monotonie, la preuve de ce lemme se fait par récurrence sur la structure de e et est décomposée en quatre parties.

On sait que pour toute expression e, pour toutes formules P et Q et pour tout état Π, l'évaluation suivante est valide :

$$[\![\mathrm{WP}(e,\ P)]\!]_\Pi \wedge [\![\mathrm{WP}(e,\ Q)]\!]_\Pi \tag{3.6}$$

Partie 1 : Les expressions pures.

D'après la définition du calcul de plus faible précondition, la propriété (3.6) est équivalente à

$$([\![P[\backslash\mathsf{result} \leftarrow e]]\!]_\Pi \wedge [\![Q[\backslash\mathsf{result} \leftarrow e]]\!]_\Pi)$$

Par conséquent, $[\![(P \wedge Q)[\backslash\mathsf{result} \leftarrow e]]\!]_\Pi$ et donc, $[\![\mathrm{WP}(e,\ P \wedge Q)]\!]_\Pi$ est valide.

Partie 2 : La liaison locale. L'expression e est de la forme : $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$

Étant donnée la propriété (3.6) pour l'expression e, d'après l'hypothèse de récurrence sur e_1, d'une part,

$$[\![\mathrm{WP}(e_1,\ \mathrm{WP}(e_2,\ P)[x \leftarrow \backslash\mathsf{result}] \wedge \mathrm{WP}(e_2,\ Q)[x \leftarrow \backslash\mathsf{result}]]\!]_\Pi$$

est valide. D'autre part, d'après l'hypothèse de récurrence sur e_2,

$$[\![\mathrm{WP}(e_2,\ P)]\!]_\Pi \wedge [\![\mathrm{WP}(e_2,\ Q)]\!]_\Pi \Rightarrow [\![\mathrm{WP}(e_2,\ P \wedge Q)]\!]_\Pi$$

Ce qu'on peut également écrire,

$$[\![\mathrm{WP}(e_2,\ P)[x \leftarrow \backslash\mathsf{result}]]\!]_\Pi \wedge [\![\mathrm{WP}(e_2,\ Q)[x \leftarrow \backslash\mathsf{result}]]\!]_\Pi \Rightarrow [\![\mathrm{WP}(e_2,\ P \wedge Q)[x \leftarrow \backslash\mathsf{result}]]\!]_\Pi$$

Ainsi, d'après la propriété de monotonie $[\![\mathrm{WP}(e_1,\ \mathrm{WP}(e_2,\ P \wedge Q)[x \leftarrow \backslash\mathsf{result}])]\!]_\Pi$, autrement dit $[\![\mathrm{WP}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2, P \wedge Q)]\!]_\Pi$ est aussi valide.

Partie 3 : Les constructions du langage.

- e est une séquence d'expressions de la forme $e = e_1; e_2$
  La preuve dans ce cas est similaire à celle de *let*. À partir de la propriété (3.6) qui s'écrit,

  $$[\![\mathrm{WP}(e_1,\ \mathrm{WP}(e_2,\ P))]\!]_\Pi \wedge [\![\mathrm{WP}(e_1,\ \mathrm{WP}(e_2,\ Q))]\!]_\Pi$$

  En appliquant l'hypothèse de récurrence sur l'expression e_1, on en déduit que

  $$[\![\mathrm{WP}(e_1,\ \mathrm{WP}(e_2,\ P) \wedge\ \mathrm{WP}(e_2,\ Q)]\!]_\Pi$$

  est valide. On conclut ensuite, la preuve en appliquant l'hypothèse de récurrence sur e_2 puis le lemme de monotonie, pour déduire que $[\![\mathrm{WP}(e_1, \mathrm{WP}(e_2,\ P \wedge Q))]\!]$ est valide dans n'importe quel environnement Π.

- e est une expression conditionnelle de la forme $e = \texttt{if}\ v_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$.
  A partir de l'hypothèse (3.6), on peut déduire que

  $$\begin{aligned}[\![&(v_1 = \textbf{true} \Rightarrow \mathrm{WP}(e_2,\ P) \wedge \mathrm{WP}(e_2,\ Q)) \wedge \\ &(v_1 = \textbf{false} \Rightarrow \mathrm{WP}(e_3,\ P) \wedge \mathrm{WP}(e_3,\ Q))]\!]_\Pi\end{aligned} \tag{3.7}$$

  Les hypothèse de récurrence sur e_2 et e_3 nous permettent de déduire

  $$[\![\mathrm{WP}(e_2,\ P) \wedge \mathrm{WP}(e_2,\ Q)]\!]_\Pi \Rightarrow [\![\mathrm{WP}(e_2,\ P \wedge Q)]\!]_\Pi$$

  et

  $$[\![\mathrm{WP}(e_3,\ P) \wedge \mathrm{WP}(e_3,\ Q)]\!]_\Pi \Rightarrow [\![\mathrm{WP}(e_3,\ P \wedge Q)]\!]_\Pi$$

  La formule (3.7), s'écrit alors

  $$[\![(v_1 = \textbf{true} \Rightarrow \mathrm{WP}(e_2,\ P \wedge Q)) \wedge (v_1 = \textbf{false} \Rightarrow \mathrm{WP}(e_3,\ P \wedge Q))]\!]_\Pi$$

  Donc, $[\![\mathrm{WP}(\texttt{if}\ v_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ P \wedge Q)]\!]_\Pi$ est valide.

– e est une expression itérative de la forme $e = \texttt{while}\ v_1\ \texttt{invariant}\ I\ \texttt{do}\ e_2\ \texttt{end}$
La formule (3.6) s'écrit dans le cas d'une expression itérative comme suit,

$$\begin{aligned}[\![\overline{I} \wedge \forall\, \mathit{Here}, \overline{I} \Rightarrow ((v_1 = \textbf{true} \Rightarrow \mathrm{WP}(e_2, \overline{I})) \wedge (v_1 = \textbf{false} \Rightarrow P))[\mathit{Old} \leftarrow \mathit{Here}] \wedge \\ \overline{I} \wedge \forall\, \mathit{Here}, \overline{I} \Rightarrow ((v_1 = \textbf{true} \Rightarrow \mathrm{WP}(e_2, \overline{I})) \wedge (v_1 = \textbf{false} \Rightarrow Q))[\mathit{Old} \leftarrow \mathit{Here}]]\!]_\Pi\end{aligned}$$

On applique ici la propriété ; si $[\![(\forall \vec{w}, P) \wedge (\forall \vec{w}, Q)]\!]$, alors $[\![\forall \vec{w}, P \wedge Q]\!]$. Dans notre cas, on quantifie sur les effets de bord du corps de la boucle dans chacune des formules. Autrement dit, sur les mêmes variables dans les deux calculs de plus faible précondition.On obtient ainsi,

$$\begin{aligned}[\![\overline{I} \wedge \forall\, \mathit{Here}, \overline{I} \Rightarrow ((v_1 = \textbf{true} \Rightarrow \mathrm{WP}(e_2, \overline{I})) \wedge \\ (v_1 = \textbf{false} \Rightarrow P) \wedge (v_1 = \textbf{false} \Rightarrow Q))[\mathit{Old} \leftarrow \mathit{Here}]]\!]_\Pi\end{aligned}$$

Donc, $[\![\overline{I} \wedge \forall\, \mathit{Here}, \overline{I} \Rightarrow (v_1 = \textbf{true} \Rightarrow \mathrm{WP}(e_2, \overline{I}) \wedge v_1 = \textbf{false} \Rightarrow P \wedge Q)[\mathit{Old} \leftarrow \mathit{Here}]]\!]_\Pi$ qui est égale à $[\![\mathrm{WP}(\texttt{while}\ v_1\ \texttt{invariant}\ I\ \texttt{do}\ e_2\ \texttt{end}, P \wedge Q)]\!]_\Pi$ par définition, est valide.

– e est un accès à un champ de la forme $e = (v \xrightarrow{l} f)$.
La preuve dans ce cas est triviale. En effet, il suffit de substituer dans l'hypothèse du lemme, les formules $\mathrm{WP}(e,\ P)$ et $\mathrm{WP}(e,\ Q)$ par leur calcul respectif : $P[\backslash\mathsf{result} \leftarrow \texttt{select}(\Pi(l), v, f)]$ et $Q[\backslash\mathsf{result} \leftarrow \texttt{select}(\Pi(l), v, f)]$. On peut alors en déduire que

$$[\![(P \wedge Q)[\backslash\mathsf{result} \leftarrow \texttt{select}(\Pi(l), v, f)]]\!]_\Pi$$

ce qui représente exactement $[\![\mathrm{WP}(e,\ P \wedge Q)]\!]_\Pi$.

– e est une mise à jour de la valeur d'un champ de la forme $e = (v_1 \rightarrow f = v_2)$.
La preuve est similaire à celle de l'accès à un champ.

– e est une allocation de la forme $e = \texttt{new}\ S$
D'après l'hypothèse, la conjonction $[\![\mathrm{WP}(\texttt{new}\ S,\ P)]\!]_\Pi \wedge [\![\mathrm{WP}(\texttt{new}\ S,\ Q)]\!]_\Pi$ est valide. Ce qui, d'après la définition du calcul de plus faible précondition, revient à dire que la formule suivante est valide.

$$\begin{aligned}[\![\texttt{let}\ l_1 = \texttt{alloc}(\mathit{Here})\ \texttt{in}\ P[\backslash\mathsf{result} \leftarrow \textbf{fst}(l_1),\ \mathit{Here} \leftarrow \textbf{snd}(l_1)]\ \wedge \\ \texttt{let}\ l_2 = \texttt{alloc}(\mathit{Here})\ \texttt{in}\ Q[\backslash\mathsf{result} \leftarrow \textbf{fst}(l_2),\ \mathit{Here} \leftarrow \textbf{snd}(l_2)]]\!]_\Pi\end{aligned} \tag{3.8}$$

l_1 et l_2 sont égales. Ainsi, la formule (3.8) s'écrit

$$[\![\texttt{let}\ l = \texttt{alloc}(\mathit{Here})\ \texttt{in}\ P \wedge Q[\backslash\mathsf{result} \leftarrow \textbf{fst}(l),\ \mathit{Here} \leftarrow \textbf{snd}(l)]]\!]_\Pi$$

Autrement dit, $[\![\mathrm{WP}(\texttt{new}\ S,\ P \wedge Q)]\!]_\Pi$ est valide.

– e est une assertion de la forme $e = \texttt{assert}\ F$.
On sait que $([\![\mathrm{WP}(e,\ P)]\!]_\Pi \wedge [\![\mathrm{WP}(e,\ Q)]\!]_\Pi)$, ou encore

$$([\![\overline{F} \wedge (\overline{F} \Rightarrow P)]\!]_\Pi\ \wedge [\![\overline{F} \wedge (\overline{F} \Rightarrow Q)]\!]_\Pi)$$

est valide dans tout environnement Π. Ainsi, la formule $[\![\overline{F} \wedge (\overline{F} \Rightarrow (P \wedge Q))]\!]_\Pi$ ou par définition $[\![\mathrm{WP}(\texttt{assert}\ F,\ P \wedge Q)]\!]_\Pi$ est valide.

– e est un appel de fonction de la forme $e = f(v_1, \ldots, v_n)$.
On veut prouver la validité de $[\![\mathrm{WP}(e,\ P \wedge Q)]\!]_\Pi$, sachant que $[\![\mathrm{WP}(e,\ P)]\!]_\Pi$ et $[\![\mathrm{WP}(e,\ Q)]\!]_\Pi$ sont valides pour toutes formules P et Q et tout environnement Π.

On sait donc que

$$\begin{array}{l}[\![Pre_f[x_i \leftarrow t_i] \wedge \\ \quad \forall\, \mathit{Here}.\ (Post_f[x_i \leftarrow t_i] \wedge assigns(W) \wedge allocates(A) \Rightarrow P)[\mathit{Old} \leftarrow \mathit{Here}] \wedge \\ \quad \forall\, \mathit{Here}.\ (Post_f[x_i \leftarrow t_i] \wedge assigns(W) \wedge allocates(A) \Rightarrow Q)[\mathit{Old} \leftarrow \mathit{Here}]]\!]_\Pi\end{array}$$

qu'on peut écrire :

$$\begin{array}{l}[\![Pre_f[x_i \leftarrow t_i] \wedge \\ \quad \forall \mathit{Here}.\ (Post_f[x_i \leftarrow t_i]\ \wedge\ assigns(W)\ \wedge\ allocates(A) \Rightarrow P \wedge Q)[\mathit{Old} \leftarrow \mathit{Here}]]\!]_\Pi\end{array}$$

est valide. Autrement dit, $[\![\mathrm{WP}(f(v_1, \ldots, v_n), P \wedge Q)]\!]_\Pi$ l'est.

– e est un expression de la forme $return$, $e = return(v,\ F,\ W,\ A,\ \Pi,\ \Pi_{prm})$.
D'après la formule (3.6),

$$\begin{array}{l}[\![(F[\backslash\mathsf{result} \leftarrow v]\ \wedge\ assigns(W)\ \wedge\ allocates(A)\ \wedge P[l_i \leftarrow \Pi(l_i)]) \\ \quad \wedge (F[\backslash\mathsf{result} \leftarrow v]\ \wedge\ assigns(W)\ \wedge\ allocates(A)\ \wedge Q[k_j \leftarrow \Pi(k_j)])]\!]_\Pi\end{array}$$

et par conséquent,

$$[\![(F[\backslash\mathsf{result} \leftarrow v])\ \wedge\ assigns(W)\ \wedge\ allocates(A)\ \wedge (P \wedge Q)[l_i \leftarrow \Pi(l_i)][k_j \leftarrow \Pi(k_j)]]\!]_\Pi$$

est valide. Autrement dit, pour tout environnement Π,

$$[\![\mathrm{WP}(return(v,\ F,\ W,\ A, \Pi,\ \Pi_{prm}),\ (P \wedge Q)]\!]$$

est vérifiée.

Partie 4 : Tout comme pour la preuve de monotonie (lemme 3.4.4), ce cas regroupe les expressions dont le calcul de la plus faible précondition est dérivé de celui du «let». Par conséquent, la preuve de la distributivité de la conjonction dans ces cas se rapportent à celle du «let». □

3.4.5 Correction du calcul de plus faible précondition

Notre objectif est d'établir qu'un programme respecte sa spécification au sens de la définition 3.3.1, en calculant des obligations de preuve que l'on prouve valides. Dans cette section, nous montrons que cette démarche est correcte. Le théorème principal est le théorème 3.4.13.

Définition 3.4.7 (Obligations de preuves d'un programme) *Supposons que toute fonction f d'un programme source $prog$ est munie d'un contrat,*

requires	Pre_f
allocates	A_f
writes	W_f
ensures	$Post_f$

et d'un corps $Body_f$, et que $\overline{Pre_f}$ et $\overline{Post_f}$ sont respectivement les traductions dans le langage logique cible des formules Pre_f et $Post_f$, alors l'ensemble des obligations de preuves extraites à partir de $prog$ est la conjonction des obligations de preuves générées pour chacune des fonctions du programme.

$$\begin{array}{l}OP(prog) \stackrel{def}{=} \bigwedge_{f \in prog}\ \forall\, \mathit{Old},\ \mathit{Here}.\ \forall \vec{x}.\ \overline{Pre_f} \Rightarrow \\ \qquad \mathrm{WP}(Body_f,\ \overline{Post_f} \wedge assigns(W_f) \wedge allocates(A_f))[\mathit{Old} \leftarrow \mathit{Here}]\end{array}$$

Nous définissons les théorèmes de préservation et de progrès du calcul de plus faible précondition, par analogie avec les théorèmes de *Subject reduction* et *Progress* du typage fort. Cette méthode, appelée *type soundness* ou *type safety*, consiste à prouver, dans un premier temps (*Subject reduction*), que si un programme p est bien typé, de type τ, et si p se réduit en un pas en un programme p', alors p' est bien typé de type τ. Puis, dans un second temps (*progress*), qu'un programme p bien typé est soit une valeur (une constante), soit il peut se réduire en un pas. Par une récurrence immédiate, on déduit alors que l'exécution d'un programme bien typé est soit finie, soit infinie, mais ne peut pas se bloquer.

Mais avant ça, on commence par poser quelques lemmes nécessaires dans les preuves des théorèmes ultérieurement énoncés.

Lemme 3.4.8 ($pure\_reduction$**)** *Si à partir d'un état mémoire* $(\Pi,\ \mathcal{H})$*, une expression pure* e *se réduit, en un pas d'exécution, vers une autre expression* e' *dans un nouvel état mémoire* $(\Pi',\ \mathcal{H}')$ *et que la plus faible précondition de* e*, calculée à partir d'une formule* Q *quelconque, est valide dans l'environnement* $(\{L \leftarrow \mathcal{H}@L\} \cdot \Pi)$*, alors la plus faible précondition de* e'*, calculée à partir de la même formule* Q *est valide dans l'environnement* $(\{L \leftarrow \mathcal{H}'@L\} \cdot \Pi')$*. Autrement dit, si* e *est une expression pure telle que* $\Pi,\ \mathcal{H},\ e \rightsquigarrow \Pi', \mathcal{H}',\ e'$ *et* $[\![\mathrm{WP}(e, Q)]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}$ *est valide, alors* $[\![\mathrm{WP}(e', Q)]\!]_{\{L \leftarrow \mathcal{H}'@L\} \cdot \Pi'}$ *est valide.*

Preuve. Avant de détailler la preuve de ce lemme, on rappelle que la plus faible précondition d'une expression pure e calculée à partir de n'importe quelle formule Q, est égale à la substitution de toutes les instances de $\backslash\mathsf{result}$ dans Q par e.

$$\mathrm{WP}(e,\ Q) = Q[\backslash\mathsf{result} \leftarrow e] \tag{3.9}$$

Supposons, maintenant, que e est une expression pure qui se réduit, en un pas d'exécution, en une autre expression e' comme suit : $\Pi,\ \mathcal{H},\ e \rightsquigarrow \Pi', \mathcal{H}',\ e'$ et que $[\![\mathrm{WP}(e, Q)]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}$ est valide pour toute formule Q. On veut alors prouver que $[\![\mathrm{WP}(e', Q)]\!]_{\{L \leftarrow \mathcal{H}'@L\} \cdot \Pi'}$ est valide. Cette preuve se fait par induction sur l'étape de réduction $\rightsquigarrow$.

Deux réductions sont possibles. Elles correspondent aux réductions des opérations unaires (respectivement binaires). Dans chacun de ces cas, e se réduit en la valeur v telle que $v = un\_op\ v_1$ (resp. $v = v_1\ bin\_op\ v_2$). Or d'après (3.9), $\mathrm{WP}(e,\ Q) = Q[\backslash\mathsf{result} \leftarrow e]$ qui est égale à $\mathrm{WP}(v,\ Q)$, une fois qu'on a substitué e par sa valeur v. Par conséquent la validité de $[\![\mathrm{WP}(e, Q)]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}$ implique la validité de $\mathrm{WP}(v,\ Q)$ dans l'environnement $\{L \leftarrow \mathcal{H}@L\} \cdot \Pi$.

□

Lemme 3.4.9 ($let\_reduction$**)** *Si à partir d'un état* $(\Pi,\ \mathcal{H})$*, l'expression* $e = (\texttt{let}\ t = e_1\ \texttt{in}\ e_2)$ *se réduit en un pas d'exécution en une autre expression* e' *dans un état* $(\Pi',\ \mathcal{H}')$*, et que la plus faible précondition de* e*, calculée à partir d'une formule* Q *quelconque, est valide dans l'environnement* $(\{L \leftarrow \mathcal{H}@L\} \cdot \Pi)$*, alors la plus faible précondition de* e'*, calculée à partir de la même formule* Q *est valide dans l'environnement* $(\{L \leftarrow \mathcal{H}'@L\} \cdot \Pi')$*. Autrement dit, si*

$$\Pi,\ \mathcal{H},\ e \rightsquigarrow \Pi', \mathcal{H}',\ e'$$

et

$$[\![\mathrm{WP}(e,\ Q)]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}$$

est vraie dans l'état $(\Pi,\ \mathcal{H})$ *alors*

$$[\![\mathrm{WP}(e', Q)]\!]_{\{L \leftarrow \mathcal{H}'@L\} \cdot \Pi'}$$

est vraie.

Preuve. Cette preuve se fait également par induction sur l'étape de réduction $\leadsto$. On suppose que e se réduit en un pas d'exécution en une autre expression e' comme suit : $\Pi,\ \mathcal{H},\ e \leadsto \Pi', \mathcal{H}',\ e'$, et que pour toute formule Q,

$$[\![(\mathrm{WP}(\texttt{let}\ t = e_1\ \texttt{in}\ e_2, Q)]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$$

est valide, et on veut prouver que

$$[\![\mathrm{WP}(e', Q)]\!]_{\{L \leftarrow \mathcal{H}'@L\}\cdot\Pi'}$$

l'est aussi.

Il existe deux règles de réduction pour une expression *let*.

– $\Pi,\ \mathcal{H},\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \leadsto \Pi[x := e_1], \mathcal{H},\ e_2$, si e_1 est une valeur.
Dans ce cas,

$$\mathrm{WP}(e_1,\ \mathrm{WP}(e_2,\ Q)[x \leftarrow \backslash\mathsf{result}])$$

est équivalent à $\mathrm{WP}(e_2,\ Q)[x \leftarrow e_1]$. Ainsi,

$$[\![\mathrm{WP}(e_2,\ Q)[x \leftarrow e_1]]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$$

est valide. En d'autres termes, $[\![\mathrm{WP}(e_2,\ Q)]\!]$ est valide dans l'environnement $(\{L \leftarrow \mathcal{H}@L\} \cdot \Pi)$ où x a pour valeur e_1. C'est-à-dire dans l'environnement $(\{L \leftarrow \mathcal{H}@L\} \cdot \Pi[x := e_1])$.

– $\Pi,\ \mathcal{H},\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \leadsto \Pi', \mathcal{H}',\ \texttt{let}\ x = e_1'\ \texttt{in}\ e_2$, si e_1 n'est pas une valeur et se réduit comme suit : $\Pi,\ \mathcal{H},\ e_1 \leadsto \Pi', \mathcal{H}',\ e_1'$.
Étant donné que

$$[\![\mathrm{WP}(e_1,\ \mathrm{WP}(e_2,\ Q)[x \leftarrow \backslash\mathsf{result}])]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$$

est valide et que

$$\Pi,\ \mathcal{H},\ e_1 \leadsto \Pi', \mathcal{H}',\ e_1'$$

alors d'après l'hypothèse de récurrence sur e_1,

$$[\![\mathrm{WP}(e_1',\ \mathrm{WP}(e_2,\ Q)[x \leftarrow \backslash\mathsf{result}])]\!]_{\{L \leftarrow \mathcal{H}'@L\}\cdot\Pi'}$$

qui est exactement la définition de $[\![\mathrm{WP}(\texttt{let}\ x = e_1'\ \texttt{in}\ e_2,\ Q)]\!]_{\{L \leftarrow \mathcal{H}'@L\}\cdot\Pi'}$, est valide.

□

Théorème 3.4.10 (Préservation par réduction) *Si à partir d'un état mémoire* $(\Pi,\ \mathcal{H})$*, une expression* e *se réduit, en un pas d'exécution, vers une autre expression* e' *dans un nouvel état mémoire* $(\Pi',\ \mathcal{H}')$ *et que la plus faible précondition de* e *calculée à partir d'une formule* Q *quelconque est valide dans l'environnement* $(\{L \leftarrow \mathcal{H}@L\} \cdot \Pi)$*, alors la plus faible précondition de* e' *calculée à partir de la même formule* Q *est valide dans l'environnement* $(\{L \leftarrow \mathcal{H}'@L\} \cdot \Pi')$*. Autrement dit, si* e *est une expression telle que*

$$\Pi,\ \mathcal{H}, e \leadsto \Pi', \mathcal{H}',\ e'$$

et

$$[\![\mathrm{WP}(e, Q)]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$$

est valide, alors

$$[\![\mathrm{WP}(e', Q)]\!]_{\{L \leftarrow \mathcal{H}'@L\}\cdot\Pi'}$$

est valide.

Preuve. Comme pour les deux lemmes précédents, qui sont en fait des cas particuliers de ce théorème, la preuve se fait par induction sur une étape de réduction $\leadsto$. On suppose donc que

$$\Pi,\ \mathcal{H},\ e \leadsto \Pi', \mathcal{H}', e'$$

et que

$$[\![\mathrm{WP}(e,\ Q)]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$$

est valide pour toute formule Q, et on prouve que

$$[\![\mathrm{WP}(e', Q)]\!]_{\{L \leftarrow \mathcal{H}'@L\}\cdot\Pi'}$$

est valide.

La preuve ainsi construite se décompose en quatre parties.

Partie 1 : Réduction d'une expression pure. Cette partie concerne les réductions suivantes.
- $\Pi,\ \mathcal{H}, op\ v \leadsto \Pi,\ \mathcal{H},\ v$ avec $op \in \{un\_op\}$.
- $\Pi,\ \mathcal{H}, v_1 op\ v_2 \leadsto \Pi,\ \mathcal{H},\ v$ avec $v = v_1 op\ v_2$ et $op \in \{bin\_op,\ rel\_op,\ log\_op\}$.

D'après le lemme *pure_reduction* (lemme 3.4.8), $[\![\mathrm{WP}(v, Q)]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$ est valide.

Partie 2 : Réduction d'une expression *let*. On a alors deux réductions possibles.
- $\Pi,\ \mathcal{H},\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \leadsto \Pi', \mathcal{H}',\ \texttt{let}\ x = e_1'\ \texttt{in}\ e_2$
- $\Pi,\ \mathcal{H}, \texttt{let}\ x = v_1\ \texttt{in}\ e_2 \leadsto \Pi[x := v_1], \mathcal{H},\ e_2$

Ces deux cas de figure sont prouvés par le lemme *let_reduction* (lemme 3.4.9).

Partie 3 : Réduction des constructions de notre langage.
- Réduction d'une séquence d'expressions $e = e_1; e_2$.
  - $\Pi,\ \mathcal{H},\ e_1; e_2 \leadsto \Pi', \mathcal{H}',\ e_1'; e_2$.
    D'une part,
    $$[\![\mathrm{WP}(e_1, \mathrm{WP}(e_2,\ Q))]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$$
    est valide. D'autre part,
    $$\Pi,\ \mathcal{H}, e_1 \leadsto \Pi', \mathcal{H}', e_1'$$
    Ainsi, d'après l'hypothèse de récurrence sur e_1,
    $$[\![\mathrm{WP}(e_1', \mathrm{WP}(e_2,\ Q))]\!]_{\{L \leftarrow \mathcal{H}'@L\}\cdot\Pi'}$$
    c'est-à-dire $[\![\mathrm{WP}(e_1'; e_2,\ Q))]\!]_{\{L \leftarrow \mathcal{H}'@L\}\cdot\Pi'}$ est valide.
  - $\Pi,\ \mathcal{H},\ (); e_2 \leadsto \Pi,\ \mathcal{H}, e_2$.
    Ce cas est trivial car $\mathrm{WP}((); e_2,\ Q) = \mathrm{WP}(e_2,\ Q)$. Ainsi, la validité de $\mathrm{WP}(e_2,\ Q)$ dans un environnement donné est une conséquence directe de la validité de $\mathrm{WP}((); e_2,\ Q)$ dans ce même environnement.
- Réduction d'une expression conditionnelle $e = \texttt{if}\ v_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$.
  - $\Pi,\ \mathcal{H}, \texttt{if}\ \textbf{true}\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \leadsto \Pi,\ \mathcal{H},\ e_2$.
    Ce cas est trivial. En effet, on sait que
    $$\mathrm{WP}(\texttt{if}\ \textbf{true}\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ Q) = \mathrm{WP}(e_2,\ Q)$$

Or,

$$[\![\mathrm{WP}(\texttt{if}\ \textbf{true}\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ Q)]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi}$$

est valide d'après l'hypothèse. Donc, $[\![\mathrm{WP}(e_2,\ Q)]\!]$ est valide dans ce même environnement.

– $\Pi,\ \mathcal{H}, \texttt{if}\ \textbf{false}\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow \Pi,\ \mathcal{H},\ e_3$.
La preuve pour cette réduction est également triviale pour la même raison que le cas précédent. La seule différence est que $\mathrm{WP}(\texttt{if}\ \textbf{false}\ \texttt{then}\ e_2\ \texttt{else}\ e_3,\ Q) = \mathrm{WP}(e_3,\ Q)$.

– Réduction d'une expression itérative de la forme $e = \texttt{while}\ e_1\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end}$.
Cette expression se réduit comme suit :

$$\begin{array}{l}\Pi,\ \mathcal{H}, \texttt{while}\ e_1\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end} \rightsquigarrow \\ \quad \Pi,\ \mathcal{H},\ \texttt{if}\ e_1\ \texttt{then}\ (e_2; \texttt{while}\ e_1\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end})\ \texttt{else}\ ()\end{array}$$

On sait que $[\![\mathrm{WP}(\texttt{while}\ e_1\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end},\ Q)]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi}$ est valide. D'après le calcul de plus faible précondition, on distingue deux cas de figures :
– Soit e_1 est une valeur.
– Soit e_1 est une expression qui peut se réduire, alors $\mathrm{WP}(\texttt{while}\ e_1\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end},\ Q)$ se réduit en $\mathrm{WP}(\texttt{let}\ v_1 = e_1\ \texttt{in while}\ v_1\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end},\ Q)$. Ce cas sera traité dans la quatrième partie de preuve.
On s'intéresse ici au cas où e_1 est une valeur qu'on note v_1. Ainsi, si

$$[\![\mathrm{WP}(\texttt{while}\ v_1\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end},\ Q)]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi}$$

est valide, alors chacune des deux formules suivantes est valide

$$\begin{cases} [\![\overline{I}]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi} \\ [\![\forall\ \mathit{Here}.\ \overline{I} \Rightarrow (v_1 = \textbf{true} \Rightarrow \mathrm{WP}(e_2,\ \overline{I})) \wedge (v_1 = \textbf{false} \Rightarrow Q)[\mathit{Old} \leftarrow \mathit{Here}]]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi} \end{cases} \tag{3.10}$$

et on veut prouver que $[\![\mathrm{WP}(\texttt{if}\ v_1\ \texttt{then}\ (e_2;\ \texttt{while}\ e_1\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end})\ \texttt{else}\ (),\ Q)]\!]$, c'est-à-dire $[\![(v_1 = \textbf{true} \Rightarrow \mathrm{WP}(e_2; e,\ Q)) \wedge (v_1 = \textbf{false} \Rightarrow \mathrm{WP}((),\ Q))]\!]$ est également valide dans le même environnement.
– Si $[\![v_1]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi} = \textbf{true}$, alors les formules (3.10) s'écrivent :

$$[\![\overline{I}]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi} \tag{3.11}$$

$$[\![\forall\ \mathit{Here}.\ \overline{I} \Rightarrow \mathrm{WP}(e_2, \overline{I})[\mathit{Old} \leftarrow \mathit{Here}]]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi} \tag{3.12}$$

Par ailleurs,

$$\begin{array}{rl} & \mathrm{WP}(\texttt{if}\ v_1\ \texttt{then}\ (e_2; \texttt{while}\ e_1\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end})\ \texttt{else}\ (),\ Q) \\ = & \mathrm{WP}(e_2; \texttt{while}\ e_1\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end},\ Q) \\ = & \mathrm{WP}(e_2, \mathrm{WP}(\texttt{while}\ e_1\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end},\ Q)) \\ = & \mathrm{WP}(e_2, \overline{I} \wedge \forall\ \mathit{Here}.\ \overline{I} \Rightarrow \mathrm{WP}(e_2, \overline{I})[\mathit{Old} \leftarrow \mathit{Here}]) \end{array}$$

Il suffit alors de montrer que $\mathrm{WP}(e_2,\ \overline{I})$ et $\mathrm{WP}(e_2,\ \forall\ \mathit{Here}.\ \overline{I} \Rightarrow \mathrm{WP}(e_2,\ \overline{I})[\mathit{Old} \leftarrow \mathit{Here}])$ sont valides et puis d'appliquer le lemme de distributivité de la conjonction (lemme 3.4.6) pour en déduire

$$\mathrm{WP}(\texttt{if}\ v_1\ \texttt{then}\ (e_2; \texttt{while}\ e_1\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end})\ \texttt{else}\ (),\ Q)$$

est à son tour valide.

La première formule est valide d'après l'hypothèse qui dit que initialement WP est vraie. Par ailleurs, vu que WP de la boucle est supposée vrai initialement, alors la formule 3.10 est valide, et vu que $\mathrm{WP}(e_2,\ \overline{I})$ est vraie, alors le lemme de monotonie (lemma 3.4.5) généralisée nous permet de déduire exactement la formule que nous voulons prouver.

– Sinon $[\![v_1]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi} = \textbf{false}$. Dans ce cas,

$$\mathrm{WP}(\texttt{while}\ \textbf{false}\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end},\ Q) \quad = \quad \overline{I} \wedge \forall \mathit{Here}.\ \overline{I} \Rightarrow Q$$

$\overline{I}$ et Q sont des formules. Elles n'ont de ce fait pas d'effets de bord. La formule $\forall \vec{y}.\ \overline{I} \Rightarrow Q$ est donc équivalente à $\overline{I} \Rightarrow Q$ et $\mathrm{WP}(e,\ Q)$ s'écrit alors $\overline{I} \wedge Q$. Par conséquent, $[\![Q]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi}$ ($[\![\mathrm{WP}((),\ Q)]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi}$) est valide, car $[\![\mathrm{WP}(e,\ Q)]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi}$ l'est par hypothèse.

– Réduction d'un accès à un champ d'une structure de données qui correspond à la règle de réduction suivante :

$$\Pi,\ \mathcal{H},\ v \to f \rightsquigarrow \Pi, \mathcal{H},\ \mathcal{H}@\mathit{Here}(v,\ f)$$

Ce cas ressemble à la première partie de la preuve. En effet, $\mathrm{WP}(v \to f,\ Q)$ est, par définition, égale à $Q[\backslash\mathsf{result} \leftarrow \texttt{select}(\mathcal{H},\ v,\ f)]$. La formule $\mathrm{WP}(\mathcal{H}@\mathit{Here}(v,\ f),\ Q)$ est de ce fait valide dans Π.

– Réduction de la mise à jour d'un champ,

$$\Pi,\ \mathcal{H}, v_1 \to f = v_2 \rightsquigarrow \Pi, \mathcal{H}[\mathit{Here} := \mathcal{H}@\mathit{Here}[(v_1,\ f) :=\ v_2],\ ()$$

Pour toute formule Q :

$$\begin{array}{rcl} \mathrm{WP}(v_1 \to f = v_2,\ Q) & = & Q[\backslash\mathsf{result} \leftarrow (),\ \mathcal{H} \leftarrow \texttt{store}(\mathcal{H},\ v_1,\ f,\ v_2)] \\ & = & \mathrm{WP}((),\ Q[\mathcal{H} \leftarrow \texttt{store}(\mathcal{H},\ v_1,\ f,\ v_2)]) \end{array}$$

$[\![\mathrm{WP}(e,\ Q)]\!]_\Pi$ est valide. Par conséquent,

$$\mathrm{WP}((),\ Q[\mathcal{H} \leftarrow \texttt{store}(\mathcal{H},\ v_1,\ f,\ v_2)])$$

est valide dans l'environnement Π. Autrement dit, $\mathrm{WP}((),\ Q)$ est valide dans Π où seule la valeur du champ f de la variable v_1 est mise à jour avec la valeur v_2 dans $\mathcal{H}$. C'est-à-dire dans l'état mémoire $(\Pi,\ \mathcal{H}[\mathit{Here} := \mathcal{H}@\mathit{Here}[(v_1,\ f) :=\ v_2])$.

– Réduction d'une allocation, $\Pi,\ \mathcal{H},\ \texttt{new}\ S \rightsquigarrow \Pi,\ \mathcal{H}',\ loc$.
Étant donné que

$$[\![\mathrm{WP}(\texttt{new}\ S,\ Q)]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi}$$

est valide, alors l'évaluation de la formule

$$[\![\texttt{let}\ l = \texttt{alloc}(\mathit{Here})\ \texttt{in}\ Q[\backslash\mathsf{result} \leftarrow \textbf{fst}(l),\ \mathit{Here} \leftarrow \textbf{snd}(l)]]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi}$$

et donc

$$[\![Q[\backslash\mathsf{result} \leftarrow \textbf{fst}(l),\ \mathit{Here} \leftarrow \textbf{snd}(l)]]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi[l\leftarrow[\![\texttt{alloc}(\mathit{Here})]\!]_{\Pi,\ \mathcal{H}}]}$$

l'est.
D'une part, la fonction logique $alloc$ a le même comportement que la fonction $fresh$. On peut donc déduire que

$$[\![Q[\backslash\mathsf{result} \leftarrow loc,\ \mathit{Here} \leftarrow \textbf{snd}(l)]]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi[l\leftarrow[\![\texttt{alloc}(\mathit{Here})]\!]_{\Pi,\ \mathcal{H}}]}$$

où $loc = fresh(\mathit{Here})$, est valide. D'autre part, la notation $l \leftarrow [\![\texttt{alloc}(\mathit{Here})]\!]_{\Pi,\ \mathcal{H}}$ implique que si $l = (loc',\ L')$, alors la valeur associée à loc', respectivement L', dans l'environnement local est loc, respectivement *Here*. Par ailleurs, évaluer $(Q[\backslash\mathsf{result} \leftarrow loc,\ \mathit{Here} \leftarrow \mathbf{snd}(l)])$ dans l'environnement $(\{L \leftarrow \mathcal{H}@L\} \cdot \Pi')$ revient à évaluer $(Q[\backslash\mathsf{result} \leftarrow loc])$ dans cet environnement mais où on associe $\mathbf{snd}(\mathit{Here})$ à la variable *Here*. Ainsi, si on définit un nouveau tas $\mathcal{H}'$ tel que $\mathcal{H}'@\mathit{Here} = \mathbf{snd}(l)$ et $\mathcal{H}'@L = \mathcal{H}@L$ pour toute variable L différente de *Here*, alors la formule $[\![Q[\backslash\mathsf{result} \leftarrow loc]]\!]_{\{L \leftarrow \mathcal{H}'@L\} \cdot \Pi'}$ est valide, et donc dans l'environnement $(\{L \leftarrow \mathcal{H}'@L\} \cdot \Pi)$.

– Réduction d'une assertion, $\Pi,\ \mathcal{H},\ \texttt{assert}\ F \rightsquigarrow \Pi,\ \mathcal{H},\ ()$.
D'après l'hypothèse, $[\![\mathrm{WP}(\texttt{assert}\ F,\ Q)]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}$ est valide. Par conséquent, la formule $[\![\overline{F} \wedge Q]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}$ et donc $[\![Q]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}$, qui est équivalent à $[\![\mathrm{WP}((),\ Q)]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}$, est aussi valide.

– Réduction d'un appel de fonction, ce qui correspond à la règle de réduction suivante :

$$\begin{array}{l}\Pi,\ \mathcal{H},\ f(v_1, \ldots, v_n) \rightsquigarrow \{x_i \leftarrow v_i\},\ \mathcal{H}',\\ \qquad \texttt{let}\ v = \mathit{Body}\ \texttt{in}\ \mathit{return}(v,\ \mathit{Post},\ W,\ A,\ \Pi,\ \{x_i \leftarrow v_i\})\end{array}$$

On sait, d'une part, que pour toute formule Q :

$$\begin{array}{l}\mathrm{WP}(f(v_1, \ldots, v_n),\ Q) = \overline{\mathit{Pre}}[x_i \leftarrow v_i] \wedge\\ \quad \forall\ \mathit{Here}\ \mathit{Old}.\ (\overline{\mathit{Post}}[x_i \leftarrow v_i] \wedge \mathit{assigns}(W)\ \wedge\ \mathit{allocates}(A) \Rightarrow Q)[\mathit{Old} \leftarrow \mathit{Here}]\end{array}$$

et que :

$$\forall \vec{x}.\ \forall\ \mathit{Here}.\ \overline{\mathit{Pre}} \Rightarrow \mathrm{WP}(\mathit{Body},\ \overline{\mathit{Post}} \wedge \mathit{assigns}(W) \wedge \mathit{allocates}(A))[\mathit{Old} \leftarrow \mathit{Here}] \quad (3.13)$$

D'une part, cette formule est valide pour toutes les valeurs possibles des paramètres x_i de la fonction f. En particulier pour les v_i. On peut donc en déduire que

$$[\![\overline{\mathit{Pre}} \Rightarrow \mathrm{WP}(\mathit{Body},\ \overline{\mathit{Post}} \wedge\ \mathit{assigns}(W)\ \wedge\ \mathit{allocates}(A))]\!] \quad (3.14)$$

est valide dans l'environnement qui associe aux paramètres formels x_i de la fonction f les paramètres effectifs v_i.
D'autre part, d'après l'hypothèse, $[\![\mathrm{WP}(f(v_1, \ldots, v_n))]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}$ est valide. Par conséquent, chacune des deux évaluations suivantes est valide :

$$\begin{array}{l}[\![\overline{\mathit{Pre}}[x_i \leftarrow v_i]]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}\\ [\![\forall\ \mathit{Here}.\ (\overline{\mathit{Post}}[x_i \leftarrow v_i] \wedge \mathit{assigns}(W) \wedge \mathit{allocates}(A) \Rightarrow Q)[\mathit{Old} \leftarrow \mathit{Here}]]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}\end{array}$$

En raisonnant de la même façon que précédemment sur ces deux formules, on obtient que les évaluations suivantes sont valides.

$$[\![\overline{\mathit{Pre}}]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \{x_i \leftarrow v_i\}} \quad (3.15)$$

$$[\![\overline{\mathit{Post}} \wedge \mathit{assigns}(W) \wedge \mathit{allocates}(A) \Rightarrow Q]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \{x_i \leftarrow v_i\}} \quad (3.16)$$

Rappelons qu'on veut prouver que :

$$\begin{array}{l}[\![\mathrm{WP}(\texttt{let}\ v = \mathit{Body}\ \texttt{in}\ \mathit{return}(v,\\ \quad \overline{\mathit{Post}} \wedge \mathit{assigns}(W) \wedge \mathit{allocates}(A),\ \Pi,\ \{x_i \leftarrow v_i\}),\ Q)]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \{x_i \leftarrow v_i\}}\end{array}$$

est valide. Or,

$$\begin{aligned} & \mathrm{WP}(\texttt{let}\ v = Body\ \texttt{in}\ return(v, \\ & \qquad \overline{Post} \wedge assigns(W) \wedge allocates(A),\ \Pi,\ \{x_i \leftarrow v_i\}),\ Q) \\ = \ & \mathrm{WP}(Body, \mathrm{WP}(return(v, \\ & \qquad \overline{Post} \wedge assigns(W) \wedge allocates(A),\ \{x_i \leftarrow v_i\},\ \Pi),\ Q)[v \leftarrow \backslash\mathsf{result}]) \\ = \ & \mathrm{WP}(Body,\ \overline{Post} \wedge assigns(W) \wedge allocates(A) \wedge Q) \end{aligned}$$

est valide. Il suffit alors de montrer que :

$$[\![\mathrm{WP}(Body, \overline{Post} \wedge assigns(W) \wedge allocates(A)) \wedge \mathrm{WP}(Body,\ Q)]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \{x_i \leftarrow v_i\}}$$

est valide. Autrement dit, que les évaluations suivantes le sont :

$$[\![\mathrm{WP}(Body,\ \overline{Post} \wedge assigns(W) \wedge allocates(A))]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \{x_i \leftarrow v_i\}} \tag{3.17}$$

$$[\![\mathrm{WP}(Body,\ Q)]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \{x_i \leftarrow v_i\}} \tag{3.18}$$

À partir des formules (3.14) et (3.15), on déduit que

$$[\![\mathrm{WP}(Body,\ \overline{Post} \wedge assigns(W) \wedge allocates(A))]\!]$$

est valide dans l'environnement $(\{L \leftarrow \mathcal{H}'@L\} \cdot \{x_i \leftarrow v_i\})$. Enfin, on applique la propriété de monotonie (lemme 3.4.4) qui, sachant (3.16), implique que

$$[\![\mathrm{WP}(Body,\ Q)]\!]_{\{L \leftarrow \mathcal{H}'@L\} \cdot \{x_i \leftarrow v_i\}}$$

est valide.

– Réduction de l'expression *return*,

$$\Pi',\ \mathcal{H}',\ return(v,\ F,\ W,\ A,\ \Pi,\ \{x_i \leftarrow v_i\}) \rightsquigarrow \Pi,\ \mathcal{H}',\ v$$

On veut alors prouver que $[\![\mathrm{WP}(v,\ Q)]\!]_{\{L \leftarrow \mathcal{H}'@L\} \cdot \Pi}$, c'est-à-dire $[\![Q[\backslash\mathsf{result} \leftarrow v]]\!]_{\{L \leftarrow \mathcal{H}'@L\} \cdot \Pi}$, est valide. D'après l'hypothèse du lemme, la formule

$$[\![(\overline{F}[x_i \leftarrow v_i] \wedge assigns(W) \wedge allocates(A) \wedge Q[l_i \leftarrow \Pi(l_i)])[\backslash\mathsf{result} \leftarrow v]]\!]$$

est valide dans l'environnement $(\Pi'.\{L \leftarrow \mathcal{H}'@L\})$. Par conséquent, $[\![Q[\backslash\mathsf{result} \leftarrow v]]\!]_{\Pi,\mathcal{H}'}$ et donc, $[\![\mathrm{WP}(v,\ Q)]\!]_{\Pi,\mathcal{H}'}$ est valide.

Partie 4 : Réduction des expressions dont le calcul de la plus faible précondition est dérivé de celui de l'expression *let*.

Pour chacune des réductions suivantes, $WP(e,\ Q)$ s'écrit sous la forme

$$\mathrm{WP}(\texttt{let}\ t = e_1\ \texttt{in}\ e[e_1 \leftarrow t],\ Q)$$

La preuve se ramène ainsi, à chaque fois, au cas du *let* et donc au lemme *let_reduction* (lemme 3.4.9).

– $\Pi,\ \mathcal{H},\ op\ e_1 \rightsquigarrow \Pi', \mathcal{H}',\ op\ e_1'$ avec $op \in \{un\_op\}$
– $\Pi,\ \mathcal{H}, e_1\ op\ e_2 \rightsquigarrow \Pi', \mathcal{H}',\ e_1'\ op\ e_2$ avec $op \in \{bin\_op,\ rel\_op,\ log\_op\}$
– $\Pi,\ \mathcal{H},\ v_1\ op\ e_2 \rightsquigarrow \Pi', \mathcal{H}',\ v_1\ op\ e_2'$ avec $op \in \{bin\_op,\ rel\_op,\ log\_op\}$
– $\Pi,\ \mathcal{H},\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow \Pi',\ \mathcal{H}',\ \texttt{if}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3$

- $\Pi,\ \mathcal{H},\ \texttt{while}\ e_1\ \texttt{invariant}\ I\ \texttt{do}\ e_2\ \texttt{end} \rightsquigarrow$
  $\Pi,\ \mathcal{H},\ \texttt{if}\ e_1\ \texttt{then}\ (e_2; \texttt{while}\ e_1\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end})\ \texttt{else}\ ()$
- $\Pi,\ \mathcal{H},\ e_1 \rightarrow f \rightsquigarrow \Pi', \mathcal{H}',\ e_1' \rightarrow f$
- $\Pi,\ \mathcal{H},\ e_1 \rightarrow f = e_2 \rightsquigarrow \Pi', \mathcal{H}',\ e_1' \rightarrow f = e_2$
- $\Pi,\ \mathcal{H},\ v_1 \rightarrow f = e_2 \rightsquigarrow \Pi', \mathcal{H}',\ v_1 \rightarrow f = e_2'$
- $\Pi,\ \mathcal{H},\ f(v_1, \ldots, v_i, e_{i+1}, \ldots, e_n) \rightsquigarrow \Pi', \mathcal{H}',\ f(v_1, \ldots, v_i, e_{i+1}', \ldots, e_n)$

□

Théorème 3.4.11 (Progrès) *Si la plus faible précondition d'une expression* e *calculée à partir de n'importe quelle formule* Q*, est valide dans un environnement donné, alors soit* e *est une valeur, soit il existe un état* $(\Pi', \mathcal{H}')$ *dans lequel* e *se réduit, en un pas d'exécution, en* e'*. En d'autres termes, si* $[\![\mathrm{WP}(e, Q)]\!]$ *est valide dans un environnement de la forme* $(\{L \leftarrow \mathcal{H}@L\} \cdot \Pi)$*, alors soit* e *est une valeur, soit il existe un état* $(\Pi', \mathcal{H}')$ *tel que* $\Pi,\ \mathcal{H},\ e \rightsquigarrow \Pi', \mathcal{H}',\ e'$*.*

Preuve. La preuve de ce théorème se fait par récurrence structurelle sur l'expression e. On a ainsi l'hypothèse de récurrence suivante sur chaque sous expression e_i de e : si $[\![\mathrm{WP}(e_i, Q)]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}$, alors soit e_i est une valeur, soit il existe un état $(\Pi', \mathcal{H}')$ tel que $\Pi,\ \mathcal{H},\ e_i \rightsquigarrow \Pi', \mathcal{H}',\ e_i'$.

Partie 1 : e est une expression pure. Trois cas de figures sont alors possibles :
- Soit e est une constante $e = v$ et donc une valeur.
- Soit e est une opération unaire de la forme $e = un\_op\ v_1$. Dans ce cas, l'expression e se réduit, en un pas d'exécution, en v telle que $v = un\_op\ v_1$.
- e est une opération binaire de la forme $e = v_1\ op\ v_2$, avec $op \in \{bin\_op,\ rel\_op,\ log\_op\}$. De même que pour le cas précédent, l'expression e se réduit, en un pas d'exécution, en v telle que $v = v_1\ op\ v_2$.

Dans ces cas, l'expression e se réduit comme suit : $\Pi,\ \mathcal{H},\ e \rightsquigarrow \Pi,\ \mathcal{H},\ v$

Partie 2 : e est une expression *let* de la forme $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$.

Pour toute formule Q et pour tout état $(\Pi,\ \mathcal{H})$, $[\![\mathrm{WP}(e_1,\ \mathrm{WP}(e_2,\ Q)[x \leftarrow \backslash\mathsf{result}])]\!]$ est valide dans l'environnement $(\{L \leftarrow \mathcal{H}@L\} \cdot \Pi)$. D'après l'hypothèse de récurrence sur la sous expression e_1, soit cette dernière est une valeur et alors l'expression e se réduit en e_2 dans l'état $(\Pi[x := e_1],\ \mathcal{H})$:

$$\Pi,\ \mathcal{H}, \texttt{let}\ x = v_1\ \texttt{in}\ e_2 \rightsquigarrow \Pi[x := v_1], \mathcal{H}, e_2$$

Soit e_1 n'est pas une valeur et dans ce cas, il existe un nouvel état $(\Pi', \mathcal{H}')$ tel que :

$$\Pi,\ \mathcal{H},\ e_1 \rightsquigarrow \Pi', \mathcal{H}',\ e_1'$$

Par conséquent, e se réduit, à son tour, dans $(\Pi', \mathcal{H}')$ comme suit :

$$\Pi,\ \mathcal{H},\ \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \rightsquigarrow \Pi', \mathcal{H}',\ \texttt{let}\ x = e_1'\ \texttt{in}\ e_2$$

Partie 3 : Les constructions de notre langage.
- e est une séquence d'expressions $e = e_1; e_2$.
  On sait que pour toute formule Q et pour tout état $(\Pi,\ \mathcal{H})$, $[\![\mathrm{WP}(e_1,\ \mathrm{WP}(e_2,\ Q))]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}$ est valide. Alors, d'après l'hypothèse de récurrence, deux cas sont possibles : Soit e_1 est une valeur

et le typage nous garantit qu'elle ne peut être que $()$, dans ce cas l'expression e se réduit en e_2 sans que l'état ne soit modifié :

$$\Pi,\ \mathcal{H},\ ();e_2 \rightsquigarrow \Pi,\ \mathcal{H}, e_2$$

Soit e_1 n'est pas une valeur, alors il existe un état $(\Pi', \mathcal{H}')$ tel que : $\Pi,\ \mathcal{H},\ e_1 \rightsquigarrow \Pi', \mathcal{H}',\ e_1'$ et donc

$$\Pi,\ \mathcal{H},\ e_1; e_2 \rightsquigarrow \Pi', \mathcal{H}',\ e_1'; e_2$$

– e est une expression conditionnelle de la forme $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_1$ où e_1 est une valeur. Dans ce cas, e_1 ne peut prendre que deux valeurs possibles. En effet, le typage nous assure que e_1 est une expression booléenne.
  – Soit $[\![e_1]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi} = \textbf{true}$, alors e se réduit en $e_2 : \Pi,\ \mathcal{H},\ e \rightsquigarrow \Pi,\ \mathcal{H},\ e_2$.
  – Soit $[\![e_1]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi} = \textbf{false}$, alors e se réduit en $e_3 : \Pi,\ \mathcal{H},\ e \rightsquigarrow \Pi,\ \mathcal{H},\ e_3$.

– e est une expression itérative de la forme $e = \texttt{while}\ e_1\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end}$ où e_1 est une valeur.
D'après l'hypothèse, $[\![\mathrm{WP}(e,\ Q)]\!]$ est valide dans l'environnement $(\{L \leftarrow \mathcal{H}@L\} \cdot \Pi)$. Ainsi,

$$[\![\overline{I} \wedge \forall Here, (\overline{I} \Rightarrow (v_1 = \textbf{true} \Rightarrow \mathrm{WP}(e_2, \overline{I})) \wedge (v_1 = \textbf{false} \Rightarrow Q))[\mathit{Old} \leftarrow \mathit{Here}]]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$$

et donc $[\![I]\!]_{\Pi,\ \mathcal{H}}$ est valide. Par conséquent, l'expression e se réduit en un pas :

$$\Pi,\ \mathcal{H},\ \texttt{while}\ e_1\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end} \rightsquigarrow \Pi,\ \mathcal{H},\ \texttt{if}\ e_1\ \texttt{then}\ (e_2; \texttt{while}\ e_1\ \texttt{invariant}\ Inv\ \texttt{do}\ e_2\ \texttt{end})\ \texttt{else}\ ()$$

– e est un accès à un champ de la forme $e = e_1 \xrightarrow{l} f$ où e_1 est une valeur.
Si e_1 est une valeur, alors d'après l'hypothèse

$$\texttt{valid}(e_1,\ l)\ \wedge\ [\![Q[\backslash\mathsf{result} \leftarrow \texttt{select}(\mathcal{H}@l,\ e_1,\ f)]]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$$

est valide. Ceci nous permet alors d'affirmer que la paire $(e_1,\ f)$ appartient à $\texttt{dom}(\mathcal{H}@l)$. L'expression e se réduit, par conséquent, en la valeur associée à la paire $(e_1,\ f)$ dans la table correspondant au label l dans $\mathcal{H}$:

$$\Pi,\ \mathcal{H},\ e_1 \xrightarrow{l} f \rightsquigarrow \Pi,\ \mathcal{H},\ \mathcal{H}@l(e_1,\ f)$$

– e est une mise à jour d'un champ de la forme $e = (e_1 \rightarrow f = e_2)$ où les sous expressions e_1 et e_2 sont toutes les deux des valeurs. La preuve dans ce cas est similaire au cas précédent. En effet, la validité de $[\![Q[\backslash\mathsf{result} \leftarrow (), \mathit{Here} \leftarrow \texttt{store}(\mathit{Here},\ e_1,\ f,\ e_2)]]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$ nous assure que $(e_1,\ f)$ appartient à $\texttt{dom}(\mathit{Here})$ et donc e se réduit comme suit :

$$\Pi,\ \mathcal{H}, v \rightarrow f = e_2 \rightsquigarrow \Pi,\ \mathcal{H}[\mathit{Here} := \mathcal{H}@\mathit{Here}[(e_1,\ f) :=\ e_2],\ ()$$

– e est une allocation, $e = \texttt{new}\ S$.
Une expression $\texttt{new}$ se réduit toujours. Il n'y a donc rien à prouver dans ce cas.

– e est une assertion, $e = \texttt{assert}\ P$.
Si $[\![\mathrm{WP}(\texttt{assert}\ P,\ Q)]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$ et donc par définition $[\![\overline{P} \wedge (\overline{P} \Rightarrow Q)]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$ est valide, alors $[\![\overline{P}]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$ est aussi valide. Ainsi, e se réduit comme indiqué dans la règle suivante :

$$\Pi,\ \mathcal{H}, \texttt{assert}\ P \rightsquigarrow \Pi,\ \mathcal{H}, ()$$

– e est un appel de fonction de la forme $e = f(v_1, \ldots, v_n)$.
D'après l'hypothèse, $[\![\mathrm{WP}(f(v_1, \ldots, v_n),\ Q)]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$ est valide. Autrement dit, la formule

$$[\![\overline{Pre}[x_i \leftarrow v_i] \wedge \forall\ \textit{Here Old}.(\overline{Post}[x_i \leftarrow v_i] \wedge assigns(W) \wedge allocates(A) \Rightarrow Q)[\textit{Old} \leftarrow\ \textit{Here}]]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$$

est valide. Ainsi, $[\![\overline{Pre}[x_i \leftarrow v_i]]\!]$ est valide dans $(\{L \leftarrow \mathcal{H}@L\} \cdot \Pi)$.
Par ailleurs, d'après le lemma 3.4.2, $[\![Pre[x_i \leftarrow v_i]]\!]$ est valide dans $(\Pi, \mathcal{H})$ et vu que les seules variables locales que contient Pre sont les paramètres formels x_i, alors $[\![Pre]\!]$ est valide dans l'état dont l'environnement local est $\{x_i \leftarrow r_i\}$. L'appel de fonction se réduit alors ;

$$\Pi,\ \mathcal{H}, f(v_1, \ldots, v_n) \rightsquigarrow \{x_i \leftarrow r_i\},\ \mathcal{H},\ \texttt{let}\ v = Body\ \texttt{in}\ return(v,\ Post,\ W,\ A,\ \Pi,\ \{x_i \leftarrow r_i\})$$

– e est une expression *return*, $e = return(v,\ Post,\ W,\ A,\ \Pi,\ \{x_i \leftarrow r_i\})$.
D'après l'hypothèse du lemme $[\![\mathrm{WP}(e,\ Q)]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$ est valide. Ainsi,

$$[\![(\overline{Post}[x_i \leftarrow v_i] \wedge assigns(W) \wedge allocates(A) \wedge Q[l_i \leftarrow \Pi(l_i)])[\backslash\mathsf{result} \leftarrow v]]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$$

est aussi valide. Par conséquent, les trois formules suivantes :

$$[\![Post[x_i \leftarrow v_i][\backslash\mathsf{result} \leftarrow v]]\!] \text{ et donc } [\![\texttt{let}\ \backslash\mathsf{result} = v\ \texttt{in}\ Post]\!]$$

$$[\![Assigns(\mathcal{H}, [\![W]\!]^{loc}_{\Pi_{prm},\ \mathcal{H}-2})]\!]$$

$$[\![Allocates(\mathcal{H},\ [\![A]\!]^{loc}_{\Pi_{prm},\ \mathcal{H}-2})]\!]$$

sont valides dans l'environnement $(\mathcal{H}, \{x_i \leftarrow r_i\})$. L'expression e se réduit alors en v :

$$\Pi', \mathcal{H}_2, return(v,\ Post,\ W,\ A,\ \Pi,\ \{x_i \leftarrow r_i\}) \rightsquigarrow \Pi, \mathcal{H}_2,\ v$$

Partie 4 : De même que pour les lemmes de monotonie et de distributivité de la conjonction, cette partie de la preuve regroupe les expressions dont le calcul de plus faible précondition se réduit à celui d'une expression *let*.

3.4.6 Théorème final de correction

Définition 3.4.12 *On dit qu'un programme vérifie ses annotations si pour toute fonction f dont le corps est donné, à partir d'un état vérifiant sa précondition, elle peut soit s'exécuter indéfiniment, soit terminer sur une valeur et un état qui vérifie sa post-condition.*

Le résultat principal de ce chapitre est maintenant le suivant :

Théorème 3.4.13 *Pour tout programme P, bien typé, ou plus simplement une expression e, tout état $(\Pi,\ \mathcal{H})$ et toute formule Q, si la plus faible précondition de e calculée à partir de Q est vérifiée dans l'état intial $(\Pi,\ \mathcal{H})$, alors soit le programme s'exécute indéfiniment, soit il se réduit, en un nombre fini d'étapes, dans un état final $(\Pi',\ \mathcal{H}')$, vers une valeur v et alors dans ce cas Q est vraie dans l'état final.*

Preuve. Ce résultat se prouve par récurrence sur le nombre d'étapes de réduction effectuées. Mais comme il n'est pas pratique d'énoncer le fait qu'une exécution est infinie, on formule cela en disant que si e s'exécute en un nombre fini d'étapes jusqu'à e' et que e' ne peut plus se réduire, alors e' est une valeur.

La preuve du théorème ainsi énoncé devient alors assez facile. En effet, si pour toute expression e, pour toute formule Q et pour tout état $(\Pi,\ \mathcal{H})\ \mathrm{WP}(e,\ Q)_{\Pi,\ \mathcal{H}}$ est valide, alors : d'après la propriété de progrès (lemme 3.4.11) démontrée auparavant, si l'expression initiale e est une valeur, alors elle ne se réduit pas et donc le programme termine. Par contre si e n'est pas une valeur, alors e se réduit en un pas d'exécution vers une nouvelle expression e' comme suit : $\Pi,\ \mathcal{H},\ e \rightsquigarrow \Pi', \mathcal{H}',\ e'$. En appliquant la propriété de préservation par réduction (lemme 3.4.10), on peut déduire que $\mathrm{WP}(e',\ Q)_{\Pi',\mathcal{H}'}$ est valide. Cette récurrence ne s'arrête que si e est une valeur.

On a ainsi montré que si une expression e n'est pas une valeur, alors elle peut toujours faire au moins un pas d'exécution. C'est-à-dire qu'un programme P peut soit terminer sur une valeur, soit s'exécuter indéfiniment.

3.5 Conclusion

Nous avons présenté dans ce chapitre un langage avec des annotations. Nous avons muni ce langage d'un système de type et d'une sémantique opérationnelle qui présente la particularité d'être bloquante. Enfin, nous avons défini un calcul de plus faible précondition pour les constructions de ce langage, que nous avons prouvé en nous inspirant de la méthode de *type soundness*, inspiré du travail préliminaire que nous avions publié au JFLA [74].

Chapitre 4

Invariants de données, champs modèles et raffinement

Sommaire

Nous présentons dans ce qui suit, une formalisation de l'approche détaillée dans le chapitre 2. Nous définissons, en premier lieu, le langage source qui étend le langage que nous avons décrit dans le chapitre précédent, en y incluant trois notions : les *modules*, les *champs modèles* et les *invariants de données*. Nous décrivons ensuite l'impact de ces extensions sur le système de types et la sémantique opérationnelle. Plus précisément, nous définissons un système de type avec régions afin de contrôler les accès à la mémoire et interdire l'aliasing. Nous étendons ensuite le calcul de plus faible précondition ainsi que sa preuve de correction, en conséquence.

4.1 Syntaxe du langage source

Le langage source est une extension du langage proposé dans le chapitre précédent que nous allons appeler dans la suite de ce document langage noyau. Cette extension consiste à rajouter une couche au dessus des déclarations pour plus de modularité. Un programme est alors structuré en modules. Ce n'est plus une suite de déclarations, mais une séquence de théories suivie d'une séquence de modules, comme indiqué dans la figure 4.1. En d'autres termes, nous n'avons plus un programme à plat mais des composants reliés entre eux, où chaque composant est défini par un identifiant unique et une suite de déclarations. On dit que les déclarations *appartiennent* au composant dans lequel elles sont déclarées ou définies.

Nous n'autorisons pas de récursion entre les composants. Les dépendances entre composants forment un graphe acyclique.

$prog$	::=	$theory^*\ module^*$	
$theory$	::=	**theory** $id\ use\_th\_decl^*\ logic\_decl^*$ **end**	Définition de théories
$module$	::=	**module** $id\ use\_decl^*\ decl^*$ **end**	Définition de modules

FIGURE 4.1 – Syntaxe du langage source.

Il existe trois types de déclarations. Le premier type est celui des déclarations « **use** » (figure 4.2). Celles-ci permettent de spécifier qu'un composant utilise un autre composant. C'est ce qui définit les dépendances entre les différents composants d'un programme. Le mot clé **use**_T permet d'importer une théorie et **use**_M importe un module.

use_decl	::=	use_th_decl
		$\mid use\_mod\_decl$
use_th_decl	::=	**use**_$T\ id$;
use_mod_decl	::=	**use**_$M\ id$;

FIGURE 4.2 – Syntaxe des déclarations « use ».

Le deuxième type de déclarations concerne les déclarations logiques dont la grammaire est la même que celle présentée dans le langage noyau (figure 3.2). Enfin le dernier type de déclarations regroupe les déclarations de types structure et les définitions de fonctions.

Les deux premiers types de déclarations peuvent être contenus dans des théories ou des modules. Par contre les déclarations de programmes, c'est-à-dire les déclarations de types structure et les définitions de fonctions ne peuvent appartenir qu'à un module. De plus, un module peut ne pas contenir de fonctions. Un tel module permet de décrire un ensemble de propriétés sur un type structure en énonçant des lemmes (propriétés logiques) qui dépendent de l'implémentation de ce type.

Contrairement à la grammaire des déclarations logiques, celle des déclarations de programmes est étendue. Ainsi, en plus du nom et des champs concrets, un type structure est défini par des champs modèles. Dans la syntaxe ces champs sont précédés par le mot clés **model** (figure 4.2). Un invariant de collage, établissant la relation entre les champs concrets et les champs modèles, peut également être associé au type structure. Le prédicat qui définit cet invariant a un paramètre unique, qui est de type structure.

Nous faisons le choix d'autoriser le référencement de tous les champs d'un type structure dans

l'invariant de collage. Si on compare avec l'approche *Ownership*, ceci signifie que tous les champs d'un type structure sont des champs **rep**.

$$
\begin{array}{rcl}
struct\_decl & ::= & \textbf{struct}\ id\{(field\ ;)^*\ \textbf{Invariant}\ id\ (\ (\textbf{struct}\ id)\ id\ )=\ pred\ \} \\
field & ::= & type\ id \mid \textbf{model}\ type\ id \\
type & ::= & \textbf{int} \mid \textbf{real} \mid \textbf{bool} \mid \textbf{unit} \\
 & & \mid type_s \\
type_s & ::= & \textbf{struct}\ id
\end{array}
$$

FIGURE 4.3 – Syntaxe des déclarations de type structure.

Exemple 4.1.1 *Le type structure qui décrit un calculateur de Morgan est défini comme suit :*

```
struct Calc{
  real sum;
  int count;
  model bag values;
} Invariant Inv_Calc(struct Calc this) =
    this−>sum == sumbag(this−>values) && this−>count == card(this−>values);
```

4.2 Typage

Comme on l'a vu dans la section 1.3, garantir la préservation des invariants de données, et par conséquent spécifier les programmes par raffinement, est difficile dans le cas des programmes à pointeurs. Nous avons besoin de contrôler fortement l'aliasing ainsi que les effets de bord cachés (dans les champs privés).

La solution que nous proposons s'appuie sur un contrôle statique au sens où il est effectué par des règles de typage. Ce typage garantit des propriétés fortes sur la structure du tas mémoire lors de l'exécution.

Dans ce système de types, les pointeurs sont paramétrés par des *régions*. Premièrement, les régions partitionnent la mémoire au sens où deux pointeurs de régions différentes sont différents. Deuxièmement, le tas mémoire n'est plus une collection d'emplacements mémoire, mais forme une structure d'arbre. Cette structure d'arbre implique une hiérarchie entre objets, analogue à celle de l'approche Ownership. Troisièmement, l'invariant d'un objet ne pourra dépendre que des éléments de la structure d'arbre dont il est la racine. Notons que dans une région donnée, tous les objets ont le même type structure.

Lors d'un appel de fonction, les régions des paramètres de type structure ne sont pas fixées a priori, il est naturel de considérer que les régions sont également des paramètres implicites de la fonction. C'est pourquoi dans les systèmes de type avec régions, on est naturellement amené à considérer du *polymorphisme de régions* où les fonctions sont paramétrées par des *variables de régions*.

Dans cette section, nous commençons par présenter le système de type avec régions et toutes les règles qui le définissent, et seulement dans second temps (section 4.2.7), l'inférence de ces types et des régions associées. L'algorithme d'inférence de types que nous proposons imite celui décrit dans la thèse de Thierry Hubert[61], qui s'inspire de l'algorithme W de Milner[76] pour les systèmes de types polymorphes.

4.2.1 Typage avec régions

Les types de notre langage sont modifiés par rapport au chapitre précédent : les types **struct** sont indexés avec une région comme le montre la grammaire ci-dessous.

$\widehat{type}$	::=	$type \mid \mathbf{struct}_{region}\ id$	
$region$	::=	ρ	Variable de région
		$\mid \mathcal{R}$	Région constante

Les objets de la grammaire $\widehat{type}$ sont appelés les *types avec régions*.

L'environnement de typage Γ est modifié en conséquence : tous les types structure ont maintenant un indice.

La nouveauté est que nous ajoutons au contexte de typage une notion de *shape* qu'on notera $\widehat{R}$. Il faut comprendre une telle *shape* comme une abstraction du tas mémoire. Il s'agit d'une fonction partielle qui associe à chaque région ρ et chaque champ f, une paire (b, τ) où b est un booléen qui indique si le champ f est modifiable ou pas et τ est le type avec région de la valeur de f. De plus, une $shape$ n'est pas n'importe quelle fonction partielle, mais un arbre fini étiqueté dans le sens suivant : le graphe $G(\widehat{R})$ est le graphe dont les nœuds sont les régions et un arc existe entre ρ et ρ' étiqueté par f si $\mathbf{snd}(\widehat{R}(\rho, f)) = \mathbf{struct}_{\rho'}\ S$ pour une certaine structure S. La $shape\ \widehat{R}$ est bien formée de racine ρ si $G(\widehat{R})$ est un arbre de racine ρ.

Plus généralement, de même que les tas mémoire sont indexés par les labels, le contexte de typage contient une collection de $shape$ indexée par des labels.

4.2.2 Jugements de typage

Les programmes écrits dans le langage dont la syntaxe est présentée dans la section 4.1 sont structurés en modules. De ce fait, un tel programme est bien typé si tous les modules (respectivement théories) qui le composent sont bien typés. Nous enrichissons, ainsi, notre environnement d'évaluation avec deux environnements supplémentaires, Γ_M et Γ_T qui associent respectivement à un module $\mathcal{M}$ et à une théorie $\mathcal{T}$ sa signature. Nous pouvons maintenant introduire les jugements de typage suivants :

- $\Gamma_M,\ \Gamma_T,\ \Gamma \vdash \Delta$ signifie que dans l'environnement $\Gamma_M,\ \Gamma_T,\ \Gamma$, les déclarations Δ sont bien formées,
- $\widehat{R},\ \Gamma \vdash t : \tau$ signifie que dans l'environnement Γ et la $shape\ \widehat{R}$, le terme t est bien formé et a le type τ,
- $\widehat{R},\ \Gamma \vdash f$: `Prop` signifie que dans l'environnement Γ et la $shape\ \widehat{R}$, la formule f est bien formée,
- $\widehat{R},\ \Gamma \vdash e : \tau,\ \widehat{R'}$ signifie que dans l'environnement Γ et la $shape\ \widehat{R}$, l'expression e est bien formée, qu'elle a le type τ et qu'elle modifie $\widehat{R}$ en $\widehat{R'}$.

Nous définissons alors les règles de typage correspondant à ces jugements en commençant par les règles de typage des termes, puis celles des formules logiques, ensuite celles des déclarations et enfin les règles de typage des expressions du langage.

4.2.3 Typage des termes

Une conséquence de la distinction entre champs modèles et champs concrets est que nous avons deux règles de typage pour les accès aux champs de type structure. En effet, les champs concrets d'un type structure ne peuvent être visibles qu'à l'intérieur du module dans lequel le type structure est déclaré, c'est-à-dire qu'ils ne peuvent être référencés que dans le corps d'une fonction qui appartient au même module. Au contraire des champs modèles qui peuvent être accessibles à partir de n'importe quel module, comme le montrent les règles suivantes :

$$\frac{(struct\ S : \ldots) \in \Gamma \qquad l \in \Gamma \qquad f \in \mathsf{Models}(S) \qquad \widehat{R},\ \Gamma \vdash t : \mathbf{struct}_\rho\ S \qquad \mathbf{snd}(\widehat{R}@l(\rho,\ f)) = \tau}{\widehat{R},\ \Gamma \vdash t \xrightarrow{l} f : \tau}$$

$$\frac{\begin{array}{c}(struct\ S : \ldots) \in \Gamma \qquad l \in \Gamma \qquad f \in \mathsf{Fields}(S) \\ \widehat{R},\ \Gamma \vdash t : \mathbf{struct}_\rho\ S \qquad \mathbf{struct}_\rho\ S \in \mathcal{M} \qquad \mathbf{snd}(\widehat{R}@l(\rho,\ f)) = \tau\end{array}}{\widehat{R},\ \Gamma \vdash t \xrightarrow{l} f : \tau}$$

où $\mathcal{M}$ est le module courant, c'est-à-dire le module que nous sommes en train de typer, la fonction Fields est celle définie dans le chapitre 3. Celle-ci retourne l'ensemble des champs concrets d'un type structure. Par analogie, la fonction Models retourne l'ensemble des champs modèles.

Les autres règles d'inférence pour les termes sont similaires à celles définies pour langage noyau avec simplement l'ajout de la $shape$ dans le contexte.

4.2.4 Typage des déclarations

Nous supposons que nous disposons d'une opération implicite sig qui permet de récupérer ce qui est visible dans un module ou une théorie, c'est-à-dire les signatures des déclarations. Nous pouvons alors introduire les règles d'inférence suivantes :

$$\frac{\Gamma_M,\ \Gamma_T,\ \Gamma \vdash \Delta \qquad \Gamma_M \cdot \{M : sig(M)\},\ \Gamma_T,\ \Gamma \vdash \Delta'}{\Gamma_M,\ \Gamma_T,\ \Gamma \vdash \mathbf{module}\ M\ \Delta\ \mathbf{end};\ \Delta'}$$

$$\frac{\Gamma_M,\ \Gamma_T,\ \Gamma \vdash \Delta \qquad \Gamma,\ \Gamma_M,\ \Gamma_T \cdot \{T : sig(T)\} \vdash \Delta'}{\Gamma_M,\ \Gamma_T,\ \Gamma \vdash \mathbf{theory}\ T\ \Delta\ \mathbf{end};\ \Delta'}$$

Si les déclarations d'un composant sont bien typées, alors une association entre le nom du composant et sa signature est ajoutée dans l'environnement correspondant : Γ_M pour les modules et Γ_T pour les théories.

$$\frac{T \in \Gamma_T \qquad \Gamma_M,\ \Gamma_T,\ \Gamma \cdot \{T : sig(T)\} \vdash \Delta}{\Gamma_M,\ \Gamma_T,\ \Gamma \vdash \mathbf{use\_}T\ T; \Delta}$$

$$\frac{M \in \Gamma_M \qquad \Gamma_M,\ \Gamma_T,\ \Gamma \cdot \{M : sig(M)\} \vdash \Delta}{\Gamma_M,\ \Gamma_T,\ \Gamma \vdash \mathbf{use\_}M\ M; \Delta}$$

Pour les déclarations «**use**», si le composant appartient bien à l'environnement correspondant, c'est-à-dire qu'il est déjà défini, alors on ajoute une association entre le nom du composant et sa signature dans l'environnement de typage. Le graphe de dépendance, entre les composants, ne contient, de ce fait, pas de cycle. En particulier, une conséquence est qu'on n'autorise pas d'appel récursif mutuel entre deux fonctions de deux modules différents.

Déclaration de structures

Nous nous intéressons ensuite au typage des déclarations de types structure. La différence avec la règle de typage dans le langage noyau porte essentiellement sur la signature du type structure typé. Ainsi, une fois que nous nous sommes assuré que le prédicat qui définit l'invariant de collage est bien typé, on insère dans l'environnement de typage Γ, l'association entre le nom de la structure et sa signature. Cette signature est composée des types de champs concrets, des types des champs modèles.

$$\frac{\rho = \mathsf{racine}(\widehat{R}) \quad \widehat{R},\ \{x : \mathbf{struct}_{\rho}\ S\} \vdash p : \texttt{Prop} \qquad \Gamma \cdot \{struct\ S : \tau_1 \dots \tau_n\ \mu_1 \dots \mu_m\} \vdash \Delta}{\begin{array}{l}\Gamma \vdash \mathbf{struct}\ S = \{\tau_1\ f_1; \dots; \tau_n\ f_n; \mathbf{model}\ \mu_1\ g_1; \dots; \mathbf{model}\ \mu_m\ g_m;\} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{Invariant}\ I\ (\mathbf{struct}\ S\ x) = p;\ \Delta\end{array}}$$

Le point important de cette nouvelle règle de typage est le typage de l'invariant p. Ce typage doit être effectué dans le contexte d'une certaine $shape$ $\widehat{R}$ qui doit être déterminée statiquement.

Exemple 4.2.1 *Considérons un premier exemple qui est celui du calculateur de Morgan.*

```
struct Calc{
  int count;
  real sum;
  model bag values;
} Invariant Inv_Calc(struct Calc c) =
     sumbag(c->values) == c->sum && card(c->values) == c->card;
```

L'invariant `Inv_Calc` *est typé dans l'environnement :*

$$\Gamma = \{c : \mathbf{struct}_{\rho}\ Calc\}$$
$$\widehat{R} = \{\ (\rho, sum) \rightarrow (\mathbf{false}, \texttt{real});\ (\rho, count) \rightarrow (\mathbf{false}, \texttt{int});\ (\rho, values) \rightarrow (\mathbf{false},\ bag);\ \}$$

Exemple 4.2.2 *Nous présentons ici un deuxième exemple dans lequel nous définissons un type structure Exam dont l'un des champs est un calculateur de Morgan supposé défini dans un autre module.*

```
struct Exam {
  Calc calc;
  bool passed;
} Invariant Inv_Exam (struct Exam e) =
  e->passed<=>sum_bag((e->calc)->values) >= card((e->calc)->values) * 10;
```

Alors, l'invariant `Inv_Exam` *est typé dans l'environnement :*

$$\Gamma = \{e : \mathbf{struct}_{\rho} Exam\}$$
$$\widehat{R} = \{\ (\rho, calc) \rightarrow (\mathbf{false}, \mathbf{struct}_{\rho_1} Calc);\ (\rho, passed) \rightarrow (\mathbf{false}, bool);$$
$$(\rho_1, values) \rightarrow (\mathbf{false}, bag);\ \}$$

Notons que cette définition de $shape$ n'autorise pas les types récursifs. Ces derniers seront abordés dans le chapitre 5.

Définitions de fonctions

Lorsque nous voulons typer une définition de fonction f, nous associons à tout paramètre formel x_i de f de type structure, une variable de région ρ_j. Seuls les types structures sont indexés par des régions, le nombre de régions m est donc inférieur ou égal au nombre de paramètres n de la fonction. Ces variables de régions font alors partie de la signature de la fonction, au même titre que les types.

La règle de typage d'une définition de fonction s'écrit :

$$\frac{\begin{array}{c}\Gamma' = \{x_i : \widehat{\tau_i} | 1 \leq i \leq n\} \cdot \Gamma \qquad \rho_j = \mathsf{racine}(\widehat{R_j}) \qquad 1 \leq j \leq m \leq n \qquad \widehat{R}_{in} = \widehat{R_1} \oplus \cdots \oplus \widehat{R_m} \\ \widehat{R}_{in},\ \Gamma' \vdash_{loc} W \qquad \widehat{R}_{in},\ \Gamma' \vdash Pre : \texttt{Prop} \qquad \widehat{R}_{out} = \widehat{R}_{in} \oplus \widehat{R}_{alloc} \qquad \widehat{R}_{in},\ \Gamma' \vdash Body : \widehat{\tau},\ \widehat{R}_{out} \\ \widehat{R}_{out},\ \Gamma' \vdash_{alloc} A \qquad \widehat{R}_{out}, \{Old\} \cdot \{\backslash\mathsf{result} : \widehat{\tau}\} \cdot \Gamma' \vdash Post : \texttt{Prop} \qquad \Gamma \cdot \{f : sig(f)\} \vdash \Delta\end{array}}{\Gamma \vdash \textbf{function}\ \tau\ f\ (\tau_1\ x_1, \ldots \tau_n\ x_n)\ contrat\{\ Body\ \}\ ; \Delta}$$

où les types $\widehat{\tau_i}$ sont indexés par les régions $\rho_1 \ldots \rho_m$. Chaque type $\widehat{\tau_i}$ est soit juste τ_i si c'est un scalaire, soit $\textbf{struct}_{\rho_j}\ S_i$ si $\tau_i = \textbf{struct}\ S_i$ structure, où ρ_j est l'une des régions paramètre, l'association entre ρ_j et τ_i devant être injective. Par ailleurs, chaque $shape\ \widehat{R_i}$ est calculée à partir de la région ρ_i associée au type $\widehat{\tau_i}$ et $\widehat{R}_{in}$ est l'union disjointe de ces $shape$. $\widehat{R}_{out}$ est à son tour une union disjointe de la shape à l'appel de la fonction $\widehat{R}_{in}$ et $\widehat{R}_{out}$ de la $shape$ calculée à partir de la clause **allocates**. Enfin, si ρ est la région associée au type de retour de la fonction f, si celui-ci est de la forme **struct** S, alors la signature de la fonction f s'écrit $sig(f) = fun\ \rho_1, \widehat{R_1}, \ldots, \rho_m, \widehat{R_m}, \rho, \widehat{\tau_1}, \ldots, \widehat{\tau_n}, \widehat{\tau}, \widehat{R}_{alloc}$.

Cette règle de typage n'est pas déterministe au sens où il faut "deviner" les $shapes$. La construction de ces $shapes$ sera effectuée par l'inférence de type.

Typage des emplacements mémoire

Les règles de typage correspondant aux emplacements mémoire des clauses **writes** sont :

$$\frac{(\textbf{struct}\ S : \ldots) \in \Gamma \qquad \widehat{R},\ \Gamma \vdash t : \textbf{struct}_\rho\ S}{\widehat{R},\ \Gamma \vdash_{loc} *t}$$

$$\frac{(\textbf{struct}\ S : \ldots) \in \Gamma \qquad f \in \mathsf{Fields}(S) \qquad \widehat{R},\ \Gamma \vdash t : \textbf{struct}_\rho\ S \qquad \textbf{struct}_\rho\ S \in \mathcal{M}}{\widehat{R},\ \Gamma \vdash_{loc} t \to f}$$

$$\frac{(\textbf{struct}\ S : \ldots) \in \Gamma \qquad f \in \mathsf{Models}(S) \qquad \widehat{R},\ \Gamma \vdash t : \textbf{struct}_\rho\ S}{\widehat{R},\ \Gamma \vdash_{loc} t \to f}$$

Les règles permettant de typer les clauses **allocates** sont :

$$\frac{(\textbf{struct}\ S : \ldots) \in \Gamma \qquad \widehat{R},\ \Gamma \vdash t : \textbf{struct}_\rho\ S}{R,\ \Gamma \vdash_{alloc} t}$$

Exemple 4.2.3 *Étant donnée la fonction suivante* `add_Exam`*, qui appartient au module* `Exam`*, en supposant que la fonction* `create_Calc` *est définie dans le module* `Calc`*.*

```
function unit add_Exam(struct Exam e, real note)
  writes *e;
{ add(e->calc, note) }
```

Nous avons $\Gamma = \{\texttt{e} : \textbf{\textit{struct}}_\rho\ \texttt{Exam}\}$. *Nous commençons alors par calculer* $\widehat{R_1}$ *dont la racine est* ρ *:*

$$\widehat{R_1} = \{(\rho, \texttt{calc}) \to (\textbf{\textit{true}}, \textbf{\textit{struct}}_{\rho_1}\texttt{Calc}); (\rho, \texttt{passed}) \to (\textbf{\textit{true}}, \texttt{bool}); (\rho_1, \texttt{values}) \to (\textbf{\textit{true}}, \texttt{bag})\}$$

puis $\widehat{R}_{alloc} = \emptyset$ *(la fonction n'effectue pas d'allocation)*

Nous ne référençons pas les champs concrets sum *et* $count$ *car nous supposons que le type structure* `Calc` *est défini dans un module autre que celui dans lequel la fonction* `add_Exam` *est définie (*`Exam`*).*

Notez, par ailleurs, que comme le contrat de la fonction spécifie que `e` *est modifiable, le premier élément de la paire associée à* (ρ, f) *où* f *est un champ de* `Exam`*, est à* ***true****. Ce qui veut dire que tous les champs de* $e \to f$ *sont modifiables.*

La fonction `add_Exam` *ainsi définie est bien typée et sa signature est :*

$$\texttt{add\_Exam} : fun\ \rho,\ \widehat{R}_1,\ \textbf{\textit{struct}}_\rho\texttt{Exam},\ \textit{real},\ \emptyset$$

Exemple 4.2.4 *Considérons ici une deuxième fonction,* `create_Exam`*, du module* `Exam`*. Cette fonction alloue une instance* `c` *de type* `Calc` *et une autre instance* `e` *de type* `Exam`*, puis initialise le champ* `calc` *de* `e` *avec* `c`*.*

```
function struct Exam create_Exam()
  allocates \result;
{
  let c = create_Calc() in
  let e = new Exam in
    e->calc = c;
    e->passed = mean(c) >= 10;
    e
}
```

Dans ce cas, c'est la shape $\widehat{R}_{in}$ *qui est vide et*

$$\widehat{R}_{alloc} = \{(\rho, \texttt{calc}) \to (\textbf{\textit{false}}, \textbf{\textit{struct}}_{\rho_1}\texttt{Calc}); (\rho, \texttt{passed}) \to (\textbf{\textit{false}}, \texttt{bool}); \\ (\rho_1, \texttt{values}) \to (\textbf{\textit{true}}, \texttt{bag})\}$$

Dans l'environnement de typage Γ*, nous ajoutons donc :*

$$\texttt{create\_Exam} : fun\ \rho,\ \textbf{\textit{struct}}_\rho\texttt{Exam},\ \widehat{R}_{alloc}$$

Exemple 4.2.5 *Étant donnée une variante de la fonction* `create_Exam` *présentée dans l'exemple précédent, dans laquelle le calculateur est passé en paramètre.*

```
function struct Exam create_Exam(struct Calc c)
  allocates \result;
 {
   let e = new Exam in
   e -> calc = c;
   e -> passed = mean(c) >= 10;
   e
 }
```

Dans ce cas,

$$\widehat{R}_{in} = \widehat{R_1} = \{(\rho_1, \texttt{values}) \rightarrow (\textbf{\textit{false}}, \texttt{bag})\}$$

et

$$\widehat{R}_{alloc} = \{(\rho_2, \texttt{calc}) \rightarrow (\textbf{\textit{false}}, \textbf{\textit{struct}}_{\rho_1}\texttt{Calc});\ (\rho_2, \texttt{passed}) \rightarrow (\textbf{\textit{false}}, \texttt{bool});\ (\rho_1, \texttt{values}) \rightarrow (\textbf{\textit{true}}, \texttt{bag})\}$$

Cette fonction est mal typée car on ne peut pas trouver $\widehat{R}_{in}$ et $\widehat{R}_{alloc}$ tels que $\widehat{R}_{in} \cap \widehat{R}_{alloc} = \emptyset$, or la règle de typage exige une union disjointe. Autrement dit, la shape $\widehat{R}_{alloc}$ référence une région qui appartient à $\widehat{R}_{in}$, la shape calculée à partir des paramètres de la fonction. Ce qui crée un alias. Concrètement, le programme affecte `e` $\rightarrow$ `calc` *à* `c` *et impose donc que la région de* `c` *appartienne à $\widehat{R}_{alloc}$. Rejeter cette fonction est bien ce que l'on souhaite, car si l'on acceptait la modification de l'objet* `c`*, alors on pourrait rompre l'invariant de* `e`*.*

Exemple 4.2.6 *Nous allons présenter dans cet exemple une deuxième variante de la fonction* `create_Exam`*, dans laquelle le calculateur est passé en paramètre.*

Définissons d'abord la fonction `duplicate` *qui crée une copie d'une instance de calculateur donnée.*

```
function struct Calc duplicate(struct Calc c)
  allocates \result;
{ let c' = new Calc in
   c'->sum = c->sum;
   c'->card = c->card;
   c'
}
```

Cette fonction `duplicate` *est bien typée et sa signature est :*

`duplicate` : fun
$\rho_1,\ \{(\rho_1, \texttt{values}) \rightarrow (\textbf{\textit{false}}, \texttt{bag});\ (\rho_1, \texttt{sum}) \rightarrow (\textbf{\textit{false}}, \textit{real});\ (\rho_1, \texttt{count}) \rightarrow (\textbf{\textit{false}}, \textit{int})\},$
$\rho_2,\ \textbf{\textit{struct}}_{\rho_1}\texttt{Calc},\ \textbf{\textit{struct}}_{\rho_2}\texttt{Calc},$
$\{(\rho_2, \texttt{values}) \rightarrow (\textbf{\textit{false}}, \texttt{bag});\ (\rho_2, \texttt{sum}) \rightarrow (\textbf{\textit{false}}, \textit{real});\ (\rho_2, \texttt{count}) \rightarrow (\textbf{\textit{false}}, \textit{int})\}$

La fonction `create_Exam_ok` *ci-dessous est à son tour bien typée.*

```
function struct Exam create_Exam_ok(struct Calc c)
  allocates \result;
{
  let e = new Exam in
    e->calc = duplicate(c);
    e->passed = mean(c) >= 10;
    e
}
```

En effet, dans ce cas de figure :

$$\widehat{R}_{in} = \widehat{R_1} = \{(\rho_1, \texttt{values}) \rightarrow (\textbf{\textit{false}}, \texttt{bag})\}$$

et

$$\widehat{R}_{alloc} = \{(\rho_2, \texttt{calc}) \to (\textbf{\textit{false}}, \textbf{\textit{struct}}_{\rho_3}\texttt{Calc});\ (\rho_2, \texttt{passed}) \to (\textbf{\textit{false}}, \texttt{bool});\\ (\rho_3, \texttt{values}) \to (\textbf{\textit{true}}, \texttt{bag})$$

Contrairement à l'exemple précédent, nous avons bien $\widehat{R_{in}} \cap \widehat{R}_{alloc} = \emptyset$.

4.2.5 Typage des formules logiques

La syntaxe des formules est la même que dans le chapitre précédent. Ainsi, les règles de typage sont pratiquement inchangées. La seule différence est que le contexte dans lequel les formules sont typées est composé de l'environnement Γ et de la $shape$ $\widehat{R}$. L'ensemble des ces règles est détaillé dans la figure 4.4.

$$\frac{\widehat{R},\ \Gamma \vdash t : \texttt{bool}}{\widehat{R},\ \Gamma \vdash t : \texttt{Prop}} \qquad \frac{\widehat{R},\ \Gamma \vdash p : \texttt{Prop}}{\widehat{R},\ \Gamma \vdash \neg p : \texttt{Prop}}$$

$$\frac{\widehat{R},\ \Gamma \vdash p_1 : \texttt{Prop} \qquad \widehat{R},\ \Gamma \vdash p_2 : \texttt{Prop}}{\widehat{R},\ \Gamma \vdash p_1\ op\ p_2 : \texttt{Prop}} \qquad op \in \{\vee,\ \wedge,\ \Rightarrow,\ \Leftrightarrow\}$$

$$\frac{\widehat{R},\ \Gamma \vdash t : \tau \qquad \widehat{R},\ \{x : \tau\} \cdot \Gamma \vdash p : \texttt{Prop}}{\widehat{R},\ \Gamma \vdash \texttt{let}\ x = t\ \texttt{in}\ p : \texttt{Prop}}$$

$$\frac{\widehat{R},\ \{x_i : \tau_i \mid 1 \leq i \leq n\} \cdot \Gamma \vdash p : \texttt{Prop}}{\widehat{R},\ \Gamma \vdash \backslash forall\ \tau_1 x_1 \ldots \tau_n x_n\ ;\ p : \texttt{Prop}}$$

$$\frac{\widehat{R},\ \{x_i : \tau_i \mid 1 \leq i \leq n\} \cdot \Gamma \vdash p : \texttt{Prop}}{\widehat{R},\ \Gamma \vdash \backslash exists\ \tau_1 x_1 \ldots \tau_n x_n\ ;\ p : \texttt{Prop}}$$

$$\frac{L \in \Gamma \qquad (S : struct \ldots) \in \Gamma \qquad \widehat{R},\ \Gamma \vdash t : \textbf{struct}_\rho\ S}{\widehat{R},\ \Gamma \vdash \textbf{valid}(t,\ L) : \texttt{Prop}}$$

$$\frac{(S : struct \ldots) \in \Gamma \qquad \widehat{R},\ \Gamma \vdash t : \textbf{struct}_\rho\ S}{\widehat{R},\ \Gamma \vdash \textbf{valid}(t) : \texttt{Prop}}$$

$$\frac{\widehat{R},\ \Gamma \vdash t_i : \tau_i \qquad (p : lfun\ \rho_1 \ldots \rho_k\ \tau_1 \ldots \tau_n\ ,\ \texttt{Prop}) \in \Gamma}{\widehat{R},\ \Gamma \vdash p(t_1, \ldots, t_n) : \texttt{Prop}}$$

FIGURE 4.4 – Règles de typage des formules logiques.

Remarquons que dans le cas des quantifications, universelles ou existentielles, les types des variables liées sont des scalaires, autrement dit, nous n'autorisons pas la quantification sur les types structure.

De plus, de même que pour les définitions de fonctions de programmes, les prédicats et les fonctions logiques sont paramétrés par les régions associées à leurs paramètres de type structure.

4.2.6 Typage des expressions

Le jugement de typage $\widehat{R},\ \Gamma \vdash e : \tau,\ \widehat{R'}$ dénote que dans l'environnement Γ, l'expression e est bien formée, qu'elle a le type τ et qu'elle modifie $\widehat{R}$ en $\widehat{R'}$. Les règles de typage que nous décrivons dans cette partie, montrent comment $\widehat{R}$ est modifié. Mais avant de présenter ces règles, nous définissons les deux fonctions Rdom et Fdom qui permettent d'accéder au domaine de $\widehat{R}$.

La fonction Rdom retourne l'ensemble des régions qui appartiennent au domaine d'une $shape$. Cet ensemble est obtenu en appliquant la projection **fst** sur chaque paire du domaine de la $shape$. La deuxième fonction Fdom retourne, étant données une $shape$ $\widehat{R}$ et une région ρ, l'ensemble des champs f tels que $(\rho, f) \in \mathtt{dom}(\widehat{R})$.

Allocation Nous commençons par présenter la règle de typage de l'allocation, car c'est la seule expression qui modifie $\widehat{R}$. Une expression de la forme $new\ S$ est bien formée et est de type $\mathbf{struct}_\rho S$ si la région ρ n'appartient pas à l'ensemble $\mathsf{Rdom}(\widehat{R})$. De plus, cette expression étend $\widehat{R}$ avec une entrée $(\rho,\ f_i) \rightarrow (\mathbf{true},\ \widehat{\tau})$ pour tout champ f_i appartenant au type structure S. Le type $\widehat{\tau}$ est le type de f_i dans S si ce n'est pas un type structure, ou bien, si le type de f_i est **struct** S', alors $\widehat{\tau} = \mathbf{struct}_{\rho'}\ S'$ où ρ' est une variable de région fraîche. Notez que les champs sont initialement modifiables, pour permettre l'initialisation de ces champs après l'allocation.

$$\frac{(struct\ S : \dots) \in \Gamma \qquad S \in \mathcal{M} \qquad \rho \notin \mathsf{Rdom}(\widehat{R})}{\widehat{R},\ \Gamma \vdash \mathtt{new}\ \ S : \mathbf{struct}_\rho S,\ \widehat{R} \oplus \{(\rho,\ f_i) \rightarrow (\mathbf{true},\ \widehat{type}(f_i, S))\}}$$

Expression labelisée

$$\frac{\widehat{R}, \{L\} \cdot \Gamma \vdash e : \tau,\ \widehat{R'}}{\widehat{R},\ \Gamma \vdash\ L : e : \tau,\ \widehat{R'}}$$

Opération unaire

$$\frac{\widehat{R},\ \Gamma \vdash e : \mathtt{int},\ \widehat{R'}}{\widehat{R},\ \Gamma \vdash op\ e : \mathtt{int},\ \widehat{R'}}\ \ op \in \{-\} \qquad \frac{\widehat{R},\ \Gamma \vdash e : \mathtt{bool},\ \widehat{R'}}{\widehat{R},\ \Gamma \vdash op\ e : \mathtt{bool},\ \widehat{R'}}\ \ op \in \{!\}$$

Opération binaire L'opérande gauche d'une opération binaire est évalué avant l'opérande droit. Par conséquent, si e_1 transforme $\widehat{R}$ en $\widehat{R_1}$, alors l'expression e_2 est typée dans l'environnement $(\widehat{R_1},\ \Gamma)$. En

outre, si e_2 transforme $\widehat{R_1}$ en $\widehat{R_2}$, alors l'expression $e_1\ op\ e_2$ transforme $\widehat{R}$ en $\widehat{R_2}$.

$$\frac{\widehat{R},\ \Gamma \vdash e_1 : \tau,\ \widehat{R_1} \qquad \widehat{R_1},\ \Gamma \vdash e_2 : \tau,\ \widehat{R_2} \qquad \tau \in \{\texttt{int},\ \texttt{real}\}}{\widehat{R},\ \Gamma \vdash e_1\ op\ e_2 : \tau,\ \widehat{R_2}} \quad op\ \in \{+,\ -,\ *\}$$

$$\frac{\widehat{R},\ \Gamma \vdash e_1 : \texttt{bool},\ \widehat{R_1} \qquad \widehat{R_1},\ \Gamma \vdash e_2 : \texttt{bool},\ \widehat{R_2}}{\widehat{R},\ \Gamma \vdash e_1\ op\ e_2 : \texttt{bool},\ \widehat{R_2}} \quad op\ \in \{\&\&, ||\}$$

$$\frac{\widehat{R},\ \Gamma \vdash e_1 : \tau,\ \widehat{R_1} \qquad \widehat{R_1},\ \Gamma \vdash e_2 : \tau,\ \widehat{R_2} \qquad \tau \in \{\texttt{int},\ \texttt{real}\}}{\widehat{R},\ \Gamma \vdash e_1\ op\ e_2 : \texttt{bool},\ \widehat{R_2}} \quad op\ \in \{<,\ <=,\ >,\ >=\}$$

$$\frac{\widehat{R},\ \Gamma \vdash e_1 : \tau,\ \widehat{R_1} \qquad \widehat{R_1},\ \Gamma \vdash e_2 : \tau,\ \widehat{R_2}}{\widehat{R},\ \Gamma \vdash e_1\ op\ e_2 : \texttt{bool},\ \widehat{R_2}} \quad op\ \in \{=,\ !=\}$$

Séquence d'expressions

$$\frac{\widehat{R},\ \Gamma \vdash e_1 : \texttt{unit}, \widehat{R_1} \quad \widehat{R_1},\ \Gamma \vdash e_2 : \tau, \widehat{R_2}}{R,\ \Gamma \vdash e_1; e_2 : \tau, \widehat{R_2}}$$

Liaison locale

$$\frac{\widehat{R},\ \Gamma \vdash e_1 : \tau_1,\ \widehat{R_1} \quad \widehat{R_1},\ \{x : \tau_1\} \cdot \Gamma \vdash e_2 : \tau_2,\ \widehat{R_2}}{\widehat{R},\ \Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2,\ \widehat{R_2}}$$

Même chose pour les cas de la séquence et la liaison locale, où l'expression e_1 transforme d'abord $\widehat{R}$ en $\widehat{R_1}$ puis l'expression e_2 transforme $\widehat{R_1}$ en $\widehat{R_2}$.

Expression conditionnelle

$$\frac{\widehat{R},\ \Gamma \vdash e : \texttt{bool},\ \widehat{R_1} \qquad \widehat{R_1},\ \Gamma \vdash e_1 : \tau,\ \widehat{R_2} \qquad \widehat{R_1},\ \Gamma \vdash e_2 : \tau,\ \widehat{R_2}}{\widehat{R},\ \Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau,\ \widehat{R_2}}$$

Dans cette règle, les sous expressions e_1 et e_2 doivent obligatoirement modifier $\widehat{R}$ de la même manière pour que l'expression e soit bien typée. Par conséquent, il n'est pas autorisé d'effectuer une allocation dans l'une des sous expressions sans faire de même dans la seconde.

Expression itérative

$$\frac{\widehat{R},\ \Gamma \vdash e : \texttt{bool},\ \widehat{R} \qquad \widehat{R},\ \Gamma \vdash inv : \texttt{Prop} \qquad \widehat{R},\ \Gamma \vdash e_1 : \texttt{unit},\ \widehat{R}}{\widehat{R},\ \Gamma \vdash \texttt{while}\ e\ \texttt{invariant}\ inv\ \texttt{do}\ e_1\ \texttt{end} : \texttt{unit},\ \widehat{R}}$$

Cette règle spécifie qu'il est interdit d'effectuer une allocation dans le corps d'une boucle. En effet, l'expression e_1 ne modifie pas la $shape$ $\widehat{R}$. Or, nous avons dit précédemment que l'allocation modifie la $shape$. L'expression e_1 n'est donc pas une allocation.

Accès aux champs La distinction entre les champs modèles et les champs concrets faite dans les termes n'a pas de sens dans les expressions, car contrairement aux champs concrets, les champs modèles n'ont d'existence que du point de vue de la spécification et de la preuve. Ils ne peuvent, par conséquent, pas être référencés dans les programmes et donc les expressions.

$$\frac{(struct\ S\ldots) \in \Gamma \qquad f \in \mathsf{Fields}(S) \qquad \textbf{struct}\ S \in \mathcal{M} \qquad \widehat{R},\ \Gamma \vdash e : \textbf{struct}_\rho\ S,\ \widehat{R_1} \qquad \textbf{snd}(\widehat{R}(\rho,\ f)) = \tau}{\widehat{R},\ \Gamma \vdash e \to f : \tau,\ \widehat{R_1}}$$

Mise à jour d'un champ Dans le cas de la mise à jour de la valeur d'un champ $e_1 \to f$, en plus de s'assurer qu'on peut bien accèder au champ $e_1 \to f$, il faut que celui-ci soit bien modifiable, c'est-à-dire que le booléen associé à la paire composée de la région à laquelle appartient le pointeur e_1 et le champ f dans $\widehat{R}$ est à **true**.

$$\frac{(struct\ S : \ldots) \in \Gamma \qquad f \in \mathsf{Fields}(S) \qquad \widehat{R},\ \Gamma \vdash e_1 : \textbf{struct}_\rho\ S,\ \widehat{R_1} \qquad \textbf{struct}_\rho\ S \in \mathcal{M} \qquad \widehat{R_1}(\rho,\ f) = (\textbf{true},\ \widehat{\tau}) \qquad \widehat{R_1},\ \Gamma \vdash e_2 : \widehat{\tau},\ \widehat{R_2}}{\widehat{R},\ \Gamma \vdash e_1 \to f = e_2 : \texttt{unit}}$$

Appel de fonction Le typage de l'appel de fonction est la règle la plus cruciale de notre système de type. Lors du typage de la déclaration de la fonction, nous avons supposé que les régions paramètre et les $shape$ associées étaient disjointes deux à deux. Lors de l'appel ces régions paramètre sont instanciées par des régions effectives qu'il faut vérifier être disjointes.

La règle de typage de l'appel de fonction s'écrit alors :

$$\frac{\begin{array}{c}(fun\ f : \rho_1\widehat{R}_1 \ldots \rho_m\widehat{R}_m\ \widehat{\tau_1} \ldots \widehat{\tau_n},\ \widehat{\tau}, \widehat{R}_{alloc}) \in \Gamma \\ \forall\ 1 \leq i \leq m,\ 1 \leq j \leq m.\ i \neq j \Rightarrow \sigma(\rho_i) \neq \sigma(\rho_j) \\ \widehat{R}_{in} = \sigma(\widehat{R}_1) \oplus \cdots \oplus \sigma(\widehat{R}_m) \qquad \widehat{R}_{in},\ \Gamma \vdash e_i : \widehat{\tau_i},\ \widehat{R}_{in}\end{array}}{\widehat{R},\ \Gamma \vdash f(e_1, \ldots, e_n) : \widehat{\tau},\ \widehat{R}_{in} \oplus \sigma(\widehat{R}_{alloc})}$$

Dans cette règle la substitution σ sera inférée lors de l'inférence de types. La deuxième condition assure la séparation que l'on souhaite entre les $shape$.

Notons que les expressions e_i passées en argument ne peuvent pas allouer. Nous pouvons, cependant, toujours s'en sortir en introduisant des liaisons locales (`let`).

Exemple 4.2.7 *Soit la fonction `f` suivante qui prend en paramètre deux instances de type `Calc`.*

```
function () f(struct Calc c1, struct Calc c2)
  writes *c1, *c2;
{ add(c1, 10);
  add(c2, 12)
}
```

Cette fonction est bien typée.

$$\begin{array}{l}\texttt{f} : fun \\ \quad \rho_1,\ \{(\rho_1, \texttt{values}) \to (\textbf{\textit{false}}, \texttt{bag})\},\ \rho_2,\ \{(\rho_2, \texttt{values}) \to (\textbf{\textit{false}}, \texttt{bag})\} \\ \quad \textbf{\textit{struct}}_{\rho_1}\texttt{Calc},\ \textbf{\textit{struct}}_{\rho_2}\texttt{Calc},\ \emptyset\end{array}$$

Si cette fonction est appelée avec la même instance de `Calc` *pour les deux paramètres :*

```
function () g(struct Calc c)
{ ...
  f(c, c);
  ...
}
```

alors cet appel est mal typé. En effet, la signature de la fonction `g` *est :*

$$\texttt{g}:\ fun\ \rho,\ \textbf{\textit{struct}}_\rho\texttt{Calc},\ \{(\rho,\texttt{values}) \rightarrow (\textbf{\textit{false}},\texttt{bag});\},\ \emptyset$$

alors à l'appel de la fonction `f`*, nous avons* $\sigma(\rho_1) = \rho$ *et* $\sigma(\rho_2) = \rho$*. La condition qui dit que les régions substituées sont disjointes deux à deux n'est donc pas vérifiée.*

4.2.7 Inférence des régions

Comme nous l'avons dit auparavant, l'algorithme d'inférence que nous définissons imite celui défini par Thierry Hubert dans sa thèse. La nouveauté de notre inférence est d'inférer des $shape$ également.

Nous commençons par construire le graphe de dépendance entre les fonctions (du module courant). Ensuite, pour chaque fonction f, dans l'ordre du graphe d'appel :

1. Construire une signature temporaire pour f où pour chaque paramètre de type structure et éventuellement pour le type de retour, on introduit une région fraîche et une $shape$ calculée par la fonction décrite ci-dessous.
2. Effectuer le typage du corps de la fonction (et aussi son contrat) où chaque affectation ou test d'égalité va introduire une contrainte d'égalité entres les régions fraîches introduites précédemment. Il s'agit ici d'un algorithme d'unification de régions.
3. Si l'emplacement mémoire $x \rightarrow f$ est mentionné dans la clause **writes** et que x est de type $\textbf{struct}_\rho S$, alors remplacer la variable b_i par la valeur **true** dans $(\rho, f) \rightarrow (b_i, \tau)$.
4. Vérifier que les $shape$ après unification sont encore disjointes deux à deux. Remplacer tous les booléens non instanciés par **false**.
5. Introduire la signature définitive de la fonction f dans le contexte, où toutes les variables de régions fraîches sont généralisées et deviennent les paramètres $\rho_1 \ldots \rho_m$ de la signature de fonction.

$$shape(\textbf{struct}_\rho S) = \bigcup_{\substack{f:\tau \in fields(S) \cup \mathsf{Models}(S) \\ \tau \text{ n'est pas un type structure}}} \{(\rho, f) \rightarrow (b_1, \tau)\}$$

$$\cup \bigcup_{\substack{f:\tau \in fields(S) \\ \tau = \textbf{struct}_{\rho'}\ S' \\ \rho' \text{fraîche}, S' \in \mathcal{M}}} \{(\rho, f) \rightarrow (b_2, \textbf{struct}_{\rho'} S')\} \cup shape(\textbf{struct}_{\rho'} S')$$

$$\cup \bigcup_{\substack{f:\tau \in fields(S) \\ \tau = \textbf{struct}_{\rho'}\ S', \rho' \text{fraîche} \\ S' \notin \mathcal{M}}} \{(\rho, f) \rightarrow (b_3, \textbf{struct}_{\rho'} S')\} \cup \bigcup_{g:\tau' \in \mathsf{Models}(S')} \{(\rho, g) \rightarrow (b_4, \tau')\}$$

où $b_1 \ldots b_4$ sont des variables booléennes fraîches.

Le calcul de cette *shape* construit l'arbre de ce qui est accessible à partir de la structure. À partir d'une structure de type S, tous les champs concrets et modèle sont accessibles, et pour les champs concrets qui sont de type structure, si celle-ci est dans le même module que le type S, alors toute la structure est dans la *shape* récursivement, sinon seuls les champs modèle sont dans la *shape*.

Exemple 4.2.8 *Reprenons la fonction* `create_Exam_ok` *présentée dans l'exemple 4.2.6.*

L'inférence est schématisée comme suit :

```
function struct Exam create_Exam_ok(struct Calc c)
1.   c : struct_ρ1 Calc
2.   R̂1 = {(ρ1, values) → (b1, bag)}
3.   \result : struct_ρ2 Exam
4.   R̂alloc = { (ρ2, passed) → (b1, bool); (ρ2, calc) → (b2, struct_ρ3 Calc); (ρ2, values) →
(b3, bag)}
5.  allocates \result;
     {
6.   let e = new Exam in
7.   e : struct_ρ4 Exam
8.   R̂ = R̂ ⊕ {(ρ4, calc) → (b4, struct_ρ5 Calc); (ρ4, passed) → (b5, bool); }
9.   let c' = duplicate(c) in
10.   c' : struct_ρ6 Calc
11.   R̂ = R̂ ⊕ {(ρ6, values) → (b6, bag); }
12.   e –> calc = c';
13.   ρ6 = ρ5
14.   e –> passed = mean(c) >= 10;
15.   e
16.   ρ4 = ρ2, ρ5 = ρ3
 }
```

Les lignes 1 à 4 introduisent les shape temporaires associées aux paramètres `c` *et* `\result`*. La ligne* 6 *introduit la variable* `e` *avec un type dans une région fraîche* ρ_4 *et la shape allouée correspondante. L'appel à la fonction* `duplicate` *alloue une nouvelle région* ρ_6 *et sa shape. L'affectation de la ligne 12 unifie les régions* ρ_6 *et* ρ_5*. Enfin, le retour de* `e` *unifie les types de* `e` *et de* $result$*, donc les régions* ρ_4 *et* ρ_2*. Cette unification se propage dans les shape et donc* $\rho_5 = \rho_3$*.*

La signature finale de la fonction `create_Exam` *est :*

$$\begin{aligned}\texttt{create\_Exam} : {} & \rho_1,\ \rho_2,\ \rho_3,\ \textbf{\textit{struct}}_{\rho_1}\texttt{Calc},\ \textbf{\textit{struct}}_{\rho_2}\texttt{Exam},\\ & \{(\rho_1, \texttt{values}) \rightarrow (\textbf{\textit{false}}, \texttt{bag})\},\\ & \{(\rho_2, \texttt{calc}) \rightarrow (\textbf{\textit{false}},\ \textbf{\textit{struct}}_{\rho_3}\texttt{Calc});\ (\rho_2, \texttt{passed}) \rightarrow (\textbf{\textit{false}}, \texttt{bool});\\ & (\rho_3, \texttt{values}) \rightarrow (\textbf{\textit{false}}, \texttt{bag})\}\end{aligned}$$

4.3 Sémantique opérationnelle

Nous allons modifier la sémantique définie au chapitre précédent de façon à expliciter à quel moment de l'exécution quels objets vérifient leurs invariants. Informellement, nous voulons dire que les objets

accessibles par une fonction vérifient leurs invariants à l'appel et au retour de l'appel. Une difficulté est de définir précisément quels sont les objets accessibles. Nous allons utiliser le typage pour déterminer ces objets. Notons donc que notre sémantique ne sera définie que sur les programmes bien typés par le typage avec régions.

Les modifications de la sémantique sont alors :
- À l'appel de fonction, le tas mémoire passé sera non pas la mémoire complète, mais seulement la partie utile indiquée par les $shape$.
- Les objets qui doivent vérifier leurs invariants sont précisément ceux du domaine de la partie du tas mémoire passée à l'appel.
- Au retour de l'appel, les champs modèle des objets modifiés sont mis à jour de façon non déterministe, de manière à rétablir les invariants de ces objets.

Une étape préalable à ces définitions est de définir une relation de correspondance entre un contexte de typage et un tas mémoire.

4.3.1 Correspondance entre le contexte de typage et le tas mémoire

Telle qu'elle est définie dans l'étape de typage, une $shape$ $\widehat{R}$ est une abstraction du tas mémoire. On dira que $\widehat{R}$ abstrait un tas $\mathcal{H}$ à travers une fonction d'abstraction ϕ, si ϕ associe à chaque emplacement mémoire une région de telle façon que :

$$\text{Si } \phi(l) = \rho \text{ et } \widehat{R}@L(\rho, f) = \rho' \text{ alors } \phi(\mathcal{H}@L(l, f)) = \rho'$$

Autrement dit, la forme du tas mémoire à un point donné du programme est compatible avec Γ et $\widehat{R}$ au même point.

Comme les $shape$ ont une structure d'arbre, l'existence d'une correspondance entre un tas mémoire et une union disjointe de $shape$ garantit que le tas a une structure de forêt.

Exemple 4.3.1 *Prenons un exemple pour illustrer la correspondance entre l'environnement de typage Γ, $\widehat{R}$ et le tas mémoire.*

```
function unit f() {
let c1 = create_Calc() in
let c2 = create_Calc() in
let e1 = create_Exam(c1) in
let e2 = create_Exam(c2) in ...}
```

où `f` est une fonction définie dans le module auquel appartient le type `Exam`, `create_Calc()` retourne une instance fraîche de `Calc` et `create_Exam()` est la fonction définie dans l'exemple 4.2.5.

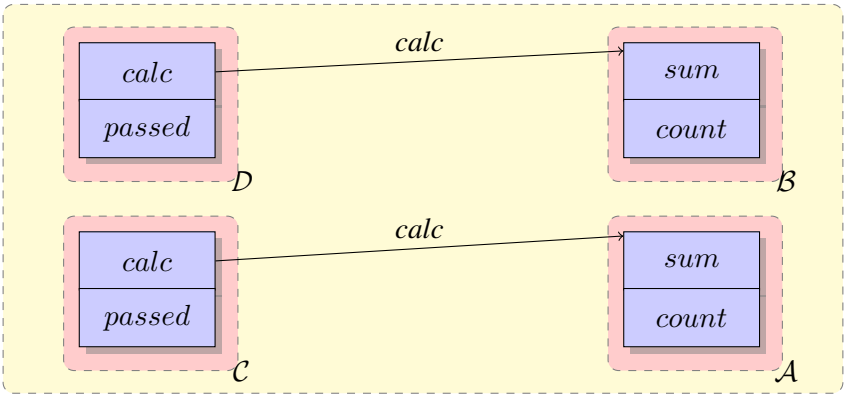
Après le typage du deuxième appel de la fonction `create_Exam`,

$$\Gamma = \{\mathtt{e}_2 : \textbf{\textit{struct}}_{\mathcal{D}}\ \mathtt{Exam},\ \mathtt{e}_1 : \textbf{\textit{struct}}_{\mathcal{C}}\ \mathtt{Exam},\ \mathtt{c}_2 : \textbf{\textit{struct}}_{\mathcal{B}}\ \mathtt{Calc},\ \mathtt{c}_1 : \textbf{\textit{struct}}_{\mathcal{A}}\ \mathtt{Calc}\}$$

et

$$\begin{aligned}\widehat{R} = \{&(\mathcal{D},\ \mathtt{calc}) \to (\textbf{\textit{true}},\ \textbf{\textit{struct}}_{\mathcal{B}}\ \mathtt{Calc});\ (\mathcal{D},\ \mathtt{passed}) \to (\textbf{\textit{true}},\ \mathit{bool});\\ &(\mathcal{C},\ \mathtt{calc}) \to (\textbf{\textit{true}},\ \textbf{\textit{struct}}_{\mathcal{A}}\ \mathtt{Calc});\ (\mathcal{C},\ \mathtt{passed}) \to (\textbf{\textit{true}},\ \mathit{bool});\\ &(\mathcal{B},\ \mathtt{sum}) \to (\textbf{\textit{true}},\ \mathit{real});\ (\mathcal{B},\ \mathtt{count}) \to (\textbf{\textit{true}},\ \mathit{int});\\ &(\mathcal{A},\ \mathtt{sum}) \to (\textbf{\textit{true}},\ \mathit{real});\ (\mathcal{A},\ \mathtt{count}) \to (\textbf{\textit{true}},\ \mathit{int})\}\end{aligned}$$

Ce qui correspond dans le tas mémoire au schéma suivant :



4.3.2 Sémantique opérationnelle

Nous ne reprendrons que les règles qui sont différentes de celles définies pour le langage noyau. En l'occurrence, celle de l'appel de fonction et de l'expression « return ». De même que pour le langage noyau (section 3.3.4), nous ne considérons les programmes bien typés. Nous disposons alors des informations collectées dans le typage, notamment la $shape\ \widehat{R}_{in}$ pour l'appel de fonction et $\widehat{R}_{out}$ pour le retour.

Appel de fonction

$$\frac{\begin{array}{c}\Pi_{prm} = \{x_i \leftarrow v_i\} \qquad [\![Pre]\!]_{\Pi_{prm},\mathcal{H}} \text{ est valide} \\ h_1 = \{(l,f) \rightarrow \mathcal{H}@\mathit{Here}(l,f) \mid \phi(l) \in \mathtt{dom}(\widehat{R}_{in})\} \qquad \mathcal{H}@\mathit{Here} = h_0 \oplus h_1 \qquad \mathcal{H}' = \{\mathit{Here} \leftarrow h_1\} \\ \forall\, l : \textbf{struct}\ S,\ l \in \mathtt{dom}(h_1) \Rightarrow [\![Inv_S(this)]\!]_{\{this \leftarrow l\},\mathcal{H}}\end{array}}{\Pi,\ \mathcal{H}, f(v_1,\ldots,v_n) \rightsquigarrow \Pi_{prm},\ \mathcal{H}',\ \mathtt{let}\ v = Body\ \mathtt{in}\ return(v,\ Post,\ W,\ A,\ \Pi,\ \Pi_{prm},\ h_0,\ \mathcal{H})}$$

Détaillons la signification de cette règle. En premier lieu nous vérifions la validité de la précondition de la fonction appelée, dans l'état mémoire $(\Pi_{prm},\ \mathcal{H})$, comme dans le chapitre précédent. Une première nouveauté est que nous calculons une partie h_1 du tas mémoire, celle qui correspond aux emplacements mémoire appartenant à la $shape\ \widehat{R}_{in}$ calculée dans le typage de cet appel. Le tas mémoire $\mathcal{H}'$ dans lequel on exécute le corps de la fonction est alors défini uniquement pour le label *Here* et $\mathcal{H}'(\mathit{Here}) = h_1$. La deuxième nouveauté est que nous exigeons que tous les objets de h_1 vérifient leurs invariants. Dans la pseudo-expression «return», nous ajoutons deux paramètres supplémentaires : h_0 qui est le complémentaire de h_1 dans $\mathcal{H}@\mathit{Here}$ comme indiqué dans la figure 4.5, et aussi le tas mémoire $\mathcal{H}$ en entier qui sert à "mémoriser" les labels autres que *Here*.

La pseudo-expression «return»

$$\frac{\begin{array}{c}h_1 = \mathcal{H}@Here[(l,\ m) := u_{(l,m)} \mid \phi(l) \in \mathtt{dom}(\widehat{R}_{out}) \ \wedge\ \textbf{fst}(\widehat{R}_{out}(\phi(l),\ m)) = \textbf{true}] \\ \mathcal{H}_1 = \mathcal{H}[\mathit{Here} := h_1] \qquad [\![\mathtt{let}\ \backslash\mathsf{result} = v\ \mathtt{in}\ Post]\!]_{\Pi_{prm},\mathcal{H}_1} \\ Assigns(\mathcal{H}_1,\ [\![W]\!]^{loc}_{\Pi_{prm},\mathcal{H}_1}) \qquad Allocates(\mathcal{H}_1,\ [\![A]\!]^{alloc}_{\Pi_{prm},\mathcal{H}_1}) \\ \forall\, l : \textbf{struct}\ S,\ l \in \mathtt{dom}(h_1) \Rightarrow [\![Inv_S(this)]\!]_{\{this \leftarrow l\},\mathcal{H}_1} \qquad \mathcal{H}_2 = \mathcal{H}'[\mathit{Here} := h_0 \oplus h_1]\end{array}}{\Pi, \mathcal{H}, return(v,\ Post,\ W,\ A,\ \Pi',\ \Pi_{prm},\ h_0,\ \mathcal{H}') \rightsquigarrow \Pi',\ \mathcal{H}_2,\ v}$$

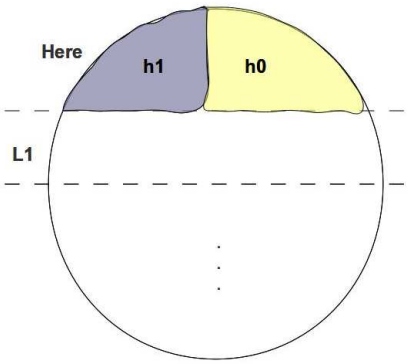


FIGURE 4.5 – Tas mémoire.

À la fin de l'exécution de la fonction, une nouveauté importante est la mise à jour des champs modèle. Dans cette règle, cela s'exprime par la modification des champs modèle m de tous les emplacements mémoire l dont la région est dans la $shape$ de sortie, et le champ m est modifiable. La nouvelle valeur $u_{l,m}$ de ce champ n'est pas déterminée de manière unique. Au contraire, toute valeur qui permet de rendre valide la post-condition, les clauses **writes** et **allocates** et les invariants des objets accessibles, donne lieu à une exécution possible du «return».

Cette règle non déterministe de mise à jour des champs modèle, est analogue à la mise à jour non déterministe des variables abstraites dans l'approche par raffinement.

4.3.3 Invariant global sur les tas mémoire

Maintenant que nous avons bien formalisé la sémantique opérationnelle en présence des invariants de données, notre objectif est de proposer un calcul de plus faible précondition correct, c'est-à-dire que la validité des obligations de preuve garantira la propriété de progrès pour notre sémantique bloquante.

Pour parvenir à cet objectif, un préalable est de poser une propriété générale INV sur un tas mémoire et un ensemble de $shape$ donnés, dont on montrera la préservation par réduction. Autrement dit, ce sera un invariant global de l'exécution.

1. Les fils de deux noeuds de régions différentes sont de régions différentes. Récursivement, cela garantit que les sous-arbres de deux noeuds de régions différentes sont séparés.
2. Si dans $\widehat{R}$ un champ f, de type $\textbf{struct}_{\rho_1}$, n'est pas modifiable $(\widehat{R}(\rho,\ f) = (false, \textbf{struct}_{\rho_1}))$, alors :
   (a) Aucun des emplacements mémoire accessibles à partir de ρ n'est modifiable. Autrement dit, si un emplacement mémoire n'est pas modifiable, alors aucun des emplacements mémoire de ses sous arbres ne l'est (Noeud $C_{\rho 3}$ dans la figure 4.6). Une conséquence directe de cette propriété est que si un champ d'un type structure n'est pas modifiable, alors aucun de ses sous-champs ne l'est.
   (b) Pour tout p tel que $\phi(p) = \rho_1$, l'invariant de données $Inv_S[this \leftarrow p]$ est vérifié.

Une manière d'exprimer l'invariant (2.a) est de dire que dans le tas mémoire, il n'y a jamais de nœud non modifiable ancêtre d'un nœud modifiable.

Exemple 4.3.2 *Étant donné,*

$$\begin{aligned}\widehat{R} = \quad & \{(\rho_1, a_1) \to (\textbf{\textit{true}}, \textbf{\textit{struct}}_{\rho_2} B);\ (\rho_1, a_2) \to (false, \textbf{\textit{struct}}_{\rho_3} C);\\ & (\rho_1, a_3) \to (\textbf{\textit{true}}, \textbf{\textit{struct}}_{\rho_4} D);\ (\rho_2, b) \to (\textbf{\textit{true}}, \dots);\\ & (\rho_3, c_1) \to (false, \dots);\ (\rho_3, c_2) \to (false, \dots);\\ & (\rho_4, d_1) \to (false, \dots);\ (\rho_4, d_2) \to (\textbf{\textit{true}}, \dots);\}\end{aligned}$$

Le tas mémoire correspondant est schématisé dans la figure 4.6.

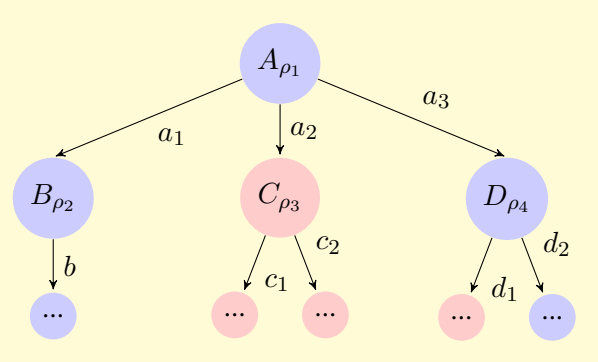


FIGURE 4.6 – Invariant global sur le tas mémoire.

Nous représentons en bleu les emplacements mémoire modifiables et en rouge ceux qui ne le sont pas. Cette figure schématise l'invariant global sur les tas mémoire.

L'objectif sera de montrer que INV est un invariant de l'exécution d'un programme si ce programme vérifie ses obligations de preuve. Comme premier résultat, nous montrons que INV est préservé par toutes les expressions autres que l'appel de fonction.

Lemme 4.3.3 *Si l'expression e est bien typée, de type τ telle que $\widehat{R},\ \Gamma \vdash e : \tau,\ \widehat{R'}$ et $\mathcal{H}$ est un tas mémoire qui vérifie l'invariant global $INV(\widehat{R}, \mathcal{H})$, alors si $\Pi,\ \mathcal{H},\ e \rightsquigarrow \Pi',\ \mathcal{H}',\ e'$, alors $INV(\widehat{R'}, \mathcal{H}')$.*

Preuve. La preuve se fait par récurrence structurelle sur l'expression e. Seules la mise à jour des champs et l'allocation ont des effets de bord.

L'expression exécutée e est une mise à jour de la forme $e_1 \to f = e_2$:

- Invariant 1 : La règle de typage d'une telle expression nous assure que $e_1 \to f$ et e_2 sont dans la même région. Par conséquent, aucun chemin supplémentaire n'est créé entre les régions.
- Invariant 2.a : Le typage assure que $e_1 \to f$ est modifiable. L'invariant impose que tous les nœuds ancêtres de e_1 sont modifiables. Par conséquent, cette affectation ne peut pas placer un nœud non modifiable au dessus d'un nœud modifiable.
- Invariant 2.b : Les seuls invariants qui peuvent être invalidés par l'affectation sont ceux de e_1 et de ses ancêtres. Mais ceux-ci étant modifiables, on n'exige pas que leurs invariants soient valides.

L'expression exécutée e est une allocation, `new` S.

- **Invariant 1 :** L'allocation se fait dans une région fraîche. Donc la propriété est trivialement préservée.
- **Invariant 2.a :** Aucun objet ne pointe sur le nouvel emplacement mémoire. Donc la propriété est trivialement préservée.
- **Invariant 2.b :** L'objet alloué est initialement modifiable, donc n'a pas besoin de vérifier son invariant.

4.4 Calcul de plus faible précondition

Comme nos règles de sémantique opérationnelle ont changé dans les cas de l'appel de fonction ainsi que pour la pseudo-expression «return», nous devons modifier le calcul de plus faible précondition pour ces mêmes cas.

La difficulté est que la sémantique opérationnelle exige la validité des invariants pour tous les objets du tas accessibles par la fonction appelée. Notre objectif est alors d'introduire, dans les obligations de preuve, uniquement les invariants des objets pour lesquels ceci est nécessaire, c'est-à-dire, les objets dont les champs ont été modifiés.

Appel de fonction :

$$\mathrm{WP}(f(v_1,\ldots,v_n),\ Q) = \overline{Pre}[x_i \leftarrow v_i]\ \wedge \bigwedge_{\substack{t\ \in\ Shape(v_i,\ldots v_n,\widehat{R}_{in})\\ t:\mathbf{struct}_\rho\ S}} \overline{Inv_S}[this \leftarrow t] \wedge$$

$$\forall \textit{Here Old}.\ (\overline{Post}[x_i \leftarrow v_i]\ \wedge\ assigns(W)\ \wedge\ allocates(A)$$

$$\bigwedge_{\substack{t\in Shape(v_1,\ldots v_n,\backslash\mathsf{result},\widehat{R}_{out})\\ t:\mathbf{struct}_\rho\ S\ \wedge\\ \exists f,\mathbf{fst}(\widehat{R}_{out}(\rho,f))=\mathbf{true}}} \overline{Inv_S}[this \leftarrow t] \Rightarrow Q[l_i \leftarrow \Pi(l_i)][\ Old \leftarrow Here]$$

où $Shape$ est un ensemble de termes calculé de la façon suivante :

$$Shape(v_1,\ldots v_n,\widehat{R}) = Shape(v_1,\widehat{R}) \cup \ldots Shape(v_n,\widehat{R})$$

$$Shape(v,\widehat{R}) = \emptyset \text{ si } v \text{ n'est pas de type structure}$$

et si v a le type $\mathbf{struct}_\rho S$, alors :

$$Shape(v,\widehat{R}) = \emptyset \text{ s'il n'y a pas de champ concret } f \text{ tel quel } (\rho,f) \in \widehat{R}$$

et sinon

$$Shape(v,\widehat{R}) = \{v\} \cup \bigcup_{f\in \mathsf{Fields}(S)} Shape(v \rightarrow f,\widehat{R})$$

La pseudo-expression $return$

$$\begin{array}{l}\mathrm{WP}(return(v,\ Post,\ W,\ A,\ \Pi,\ \{x_i \leftarrow v_i\},\ h_0,\ \mathcal{H}),\ Q) = \\ \quad \exists\, u_1 \ldots u_k.\ (\overline{Post}[x_i \leftarrow v_i]\ \wedge \bigwedge_{\substack{t \in Shape(v_1,\ldots v_n,\backslash\mathsf{result},\widehat{R}_{out}) \\ t:\mathbf{struct}_\rho\ S\ \wedge \\ \exists f,\mathbf{fst}(\widehat{R}_{out}(\rho,f))=\mathbf{true}}} \overline{Inv_S}[this \leftarrow t] \wedge \\ \qquad assigns(W)\ \wedge\ allocates(A)\ \wedge Q[l_i \leftarrow \Pi(l_i)] \\ \quad)[\sigma,\ \backslash\mathsf{result} \leftarrow v]\end{array}$$

où σ est une substitution qui remplace, pour chaque t dans $Shape(v_1, \ldots v_n, \backslash\mathsf{result}, \widehat{R}_{out})$ les occurrences de `select`(Here, t, m) par une variable fraîche u_j et `select`(Old, t, m) par `select`(Here, t, m).

Exemple 4.4.1 *Soit la fonction* `wp_exam` *qui appartient au même module que le type* `Exam` *:*

```
function unit wp_exam(struct Exam e)
  writes *e;
{
  add(e->calc, 10);
  e -> passed = ...
}
```

La signature de cette fonction `f` *est alors :*

$$\begin{array}{l}\texttt{wp\_exam} : \rho,\ \rho_1, \mathbf{struct}_\rho Exam, \\ \qquad \widehat{R}_{in} = \{(\rho, calc) \rightarrow (\mathbf{true}, \mathbf{struct}_{\rho_1} Calc); \\ \qquad\qquad (\rho, passed) \rightarrow (\mathbf{true}, bool);\ (\rho_1, values) \rightarrow (\mathbf{true}, bag)\},\ \emptyset\end{array}$$

Rappelons que le type `Calc` *est défini dans un module différent. Ainsi,*

$$Shape(\texttt{e} : \mathbf{struct}_\rho, \widehat{R}_{in}) = \{e\}$$

Il suffit, dans ce cas de figure, de vérifier l'invariant de `e`*. L'invariant de* `calc` *est établi par la fonction* `add`*.*

Exemple 4.4.2 *Nous supposons maintenant, dans cet exemple, que les types* `Calc` *et* `Exam` *appartiennent au même module. Les champs* `sum` *et* `count` *sont, par conséquent, visibles par la fonction* `wp_exam2` *définie ci-dessous.*

```
function unit wp_exam2(struct Exam e)
  writes *e;
{
  e->calc->sum = e->calc->sum + 10;
  e->calc->count = e->calc->count + 1;
  e -> passed = ...
}
```

La signature de cette fonction `wp_exam2` *est alors :*

$$\begin{array}{l}\texttt{wp\_exam2} : \rho,\ \rho_1, \textbf{\textit{struct}}_\rho Exam,\\ \qquad \widehat{R}_{in} = \{(\rho, calc) \rightarrow (\textbf{\textit{true}}, \textbf{\textit{struct}}_{\rho_1} Calc);\ (\rho, passed) \rightarrow (\textbf{\textit{true}}, bool);\\ \qquad\qquad (\rho_1, sum) \rightarrow (\textbf{\textit{true}}, \texttt{real});\ (\rho_1, count) \rightarrow (\textbf{\textit{true}}, \texttt{int});\\ \qquad\qquad (\rho_1, values) \rightarrow (\textbf{\textit{true}}, \texttt{bag})\},\ \emptyset\end{array}$$

Alors,

$$Shape(\texttt{e} : \textbf{\textit{struct}}_\rho, \widehat{R}_{in}) = \{e,\ e \rightarrow calc\}$$

et donc, il faut vérifier les invariant de `e` *et de* `e->calc`*.*

Le point important illustré par ces exemples est que même si l'invariant d'un objet "privé" (au sens dont le type n'appartient pas au module courant) n'est pas visible par le programme client, c'est-à-dire qu'il n'apparaît jamais dans les obligations de preuve du client (ni en hypothèse ni en conclusion), cet invariant sera préservé au cours de l'exécution. C'est en ce sens que notre approche est réellement modulaire.

4.4.1 Lemmes auxiliaires du calcul de plus faible précondition

La définition du calcul de plus faible précondition ayant changé, il est nécessaire de prouver les propriétés de monotonie et de distributivité de la conjonction pour cette nouvelle définition. Mais comme les preuves sont identiques, nous n'allons pas les reprendre dans ce chapitre.

4.4.2 Correction du calcul de plus faible précondition

Comme pour le calcul de plus faible précondition défini dans le langage noyau, on montre que ce calcul est correct en prouvant la préservation par réduction ainsi que le progrès.

$$\begin{array}{rcl} OP(prog) & \stackrel{def}{=} & \bigwedge_{f \in prog} \forall \vec{x}.\, \forall\, \textit{Here Old}.\, (\overline{Pre_f} \wedge \bigwedge_{\substack{t \in Shape(x_1,\ldots x_n, \backslash\mathsf{result}, \widehat{R}_{in}) \\ t:\textbf{struct}_\rho\ S}} \overline{Inv_S}[this \leftarrow t]) \Rightarrow \\ & & \exists\, u_1 \ldots u_k.\, \mathrm{WP}(Body_f,\ \overline{Post_f} \wedge assigns(W_f)\ \wedge\ allocates(A_f)\ \wedge \\ & & \qquad \bigwedge_{\substack{t \in Shape(x_1,\ldots x_n, \backslash\mathsf{result}, \widehat{R}_{out}) \\ t:\textbf{struct}_\rho\ S\ \wedge \\ \exists f, \textbf{fst}(\widehat{R}_{out}(\rho,f)) = \textbf{true}}} \overline{Inv_S}[this \leftarrow t])[\sigma,\ \textit{Old} \leftarrow \textit{Here}\,] \end{array} \tag{4.1}$$

où σ est la même substitution que dans la règle ci-dessus pour l'expression «return».

Nous prouvons maintenant la correction du calcul de plus faible précondition en énonçant les propriétés de progrès et de préservation. Comme dans le chapitre précédent, ces lemmes sont prouvés, respectivement, par récurrence sur la structure de l'expression e et par induction sur une étape de réduction $\rightsquigarrow$. Or, vu que nous n'avons modifié la sémantique opérationnelle et le calcul de plus faible précondition que pour l'appel de fonction et le $return$, nous ne reprendrons ainsi, que les preuves de ces deux expressions.

Lemme 4.4.3 (Préservation) *Si à partir d'un état mémoire* $(\Pi,\ \mathcal{H})$ *tel que* $\mathcal{H}$ *vérifie son invariant, une expression* e *se réduit, en un pas d'exécution, vers une autre expression* e' *dans un nouvel état mémoire* $(\Pi',\ \mathcal{H}')$ *et que la plus faible précondition de* e *calculée à partir d'une formule* Q *quelconque est valide dans l'environnement* $(\{L \leftarrow \mathcal{H}@L\} \cdot \Pi)$*, alors la plus faible précondition de* e' *calculée à partir de la même formule* Q *est valide dans l'environnement* $(\{L \leftarrow \mathcal{H}'@L\} \cdot \Pi')$ *et* $\mathcal{H}'$ *vérifie également son invariant. Autrement dit, si* e *est une expression telle que* $\widehat{R}, \Gamma \vdash e : \tau,\ \widehat{R'}$ *et* $\Pi,\ \mathcal{H}, e \rightsquigarrow \Pi', \mathcal{H}',\ e'$*,* $INV(\widehat{R}, \mathcal{H})$ *est établi et* $[\![\mathrm{WP}(e, Q)]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}$ *est valide, alors* $[\![\mathrm{WP}(e', Q)]\!]_{\{L \leftarrow \mathcal{H}'@L\} \cdot \Pi'}$ *est valide et* $INV(\widehat{R'}, \mathcal{H}')$ *est établi.*

Preuve.

– Réduction d'un appel de fonction, ce qui correspond à la règle de réduction suivante :

$$\Pi,\ \mathcal{H},\ f(v_1, \ldots, v_n) \rightsquigarrow \\ [x_i \leftarrow v_i], \mathcal{H}',\ \texttt{let}\ v = Body\ \texttt{in}\ return(v,\ Post,\ W,\ A,\ \Pi_{prm},\ \Pi,\ h_0,\ \mathcal{H})$$

Nous savons que $[\![(f(v_1, \ldots, v_n),\ Q)]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \Pi}$ est valide, alors, d'après la définition de la fonction WP, les évaluations suivantes sont valides dans l'environnement $(\{L \leftarrow \mathcal{H}@L\} \cdot \Pi)$ et en particulier dans $(\{L \leftarrow \mathcal{H}@L\} \cdot \{x_i \leftarrow v_i\})$:

$$[\![\overline{Pre} \wedge \bigwedge_{\substack{t \in Shape(v_i, \ldots v_n, \widehat{R}_{in}) \\ t{:}\mathbf{struct}_\rho\ S}} \overline{Inv_S}[this \leftarrow t])]\!] \tag{4.2}$$

$$\forall\ \textit{Here Old}.\ [\![(\overline{Post} \wedge assigns(W) \wedge allocates(A) \wedge \bigwedge_{\substack{t \in Shape(v_1, \ldots v_n, \backslash\mathsf{result}, \widehat{R}_{out}) \\ t{:}\mathbf{struct}_\rho\ S \\ \exists f, \mathbf{fst}(\widehat{R}_{out}(\rho, f)) = \mathbf{true}}} \overline{Inv_S}[this \leftarrow t]) \Rightarrow Q[l_i \leftarrow \Pi(l_i)][\ Old \leftarrow Here]]\!] \tag{4.3}$$

La formule 4.1 est aussi valide. Par conséquent, étant donnée (4.2), la formule :

$$\forall\ \textit{Old},\ \textit{Here}.\ \exists\ u_1 \ldots u_k.\ [\![\mathrm{WP}(Body_f,\ \overline{Post_f} \wedge assigns(W_f) \wedge allocates(A_f) \wedge \bigwedge_{\substack{t \in Shape(x_1, \ldots x_n, \backslash\mathsf{result}, \widehat{R}_{out}) \\ t{:}\mathbf{struct}_\rho\ S \\ \exists f, \mathbf{fst}(\widehat{R}_{out}(\rho, f)) = \mathbf{true}}} \overline{Inv_S}[this \leftarrow t])[\sigma,\ \textit{Old} \leftarrow \textit{Here}\ ]]\!]$$

où σ est la substitution définie pour le calcul de plus faible précondition de l'expression «return» (4.4), est vérifiée dans l'environnement $(\{L \leftarrow \mathcal{H}@L\} \cdot \{x_i \leftarrow v_i\})$.
Sachant (4.3), on applique alors le lemme de monotonie pour déduire la validité de :

$$\forall\ \textit{Here Old}.\ \exists\ u_1 \ldots u_k.\ [\![\mathrm{WP}(Body_f,\ Q[l_i \leftarrow \Pi(l_i)])[\sigma,\ \textit{Old} \leftarrow \textit{Here}\ ]]\!]_{\{L \leftarrow \mathcal{H}@L\} \cdot \{x_i \leftarrow v_i\}}$$

Ainsi, d'après la propriété de distributivité de la conjonction, on conclut que :

$$\forall\ \textit{Here Old}.\ \exists\ u_1 \ldots u_k.\ [\![\mathrm{WP}(Body_f,\ \overline{Post_f} \wedge assigns(W) \wedge allocates(A) \wedge \bigwedge_{\substack{t \in Shape(x_1, \ldots x_n, \backslash\mathsf{result}, \widehat{R}_{out}) \\ t{:}\ \mathbf{struct}_\rho S \\ \exists f, \mathbf{fst}(\widehat{R}_{out}(\rho, f)) = \mathbf{true}}} \overline{Inv_S}[this \leftarrow t]) \wedge Q[l_i \leftarrow \Pi(l_i)])[\sigma,\ \textit{Old} \leftarrow \textit{Here}\ ]]\!] \tag{4.4}$$

est valide dans $\{L \leftarrow \mathcal{H}@L\} \cdot \{x_i \leftarrow v_i\}$.

Les champs modèle appartiennent obligatoirement à un type structure, les m_j sont donc de la forme $e \rightarrow m_j$. Substituer chaque m_j par u_j consiste alors à mettre à jour sa valeur dans le tas mémoire. Nous obtenons alors $h = \texttt{store}(\mathcal{H}@\mathit{Here}, e, m_j, u_j)$. On construit donc un nouveau tas mémoire $\mathcal{H}' = \mathcal{H}'[L := h]$ pour tout label L.

En outre, l'appel de fonction n'a pas d'effets de bord et $\mathcal{H}'$ est construite à partir d'une partie du tas initial $\mathcal{H}$. Sachant que $\mathcal{H}$ vérifie l'invariant global sur les tas mémoire, ce dernier est établi aussi pour $\mathcal{H}'$.

– Réduction de l'expression *return*

$$\Pi, \mathcal{H}, return(v,\ Post,\ W,\ A,\ \Pi',\ \Pi_{prm},\ h_0,\ \mathcal{H}') \rightsquigarrow \Pi',\ \mathcal{H}_2,\ v$$

D'après l'hypothèse,

$$[\![\mathrm{WP}(return(v,\ Post,\ \Pi',\ \Pi_{prm},\ h_0,\ \mathcal{H}'),\ Q)]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$$

et donc

$$[\![\exists\, u_1 \ldots u_k.\ Q[l_i \leftarrow \Pi(l_i)][\sigma,\ \backslash\mathsf{result} \leftarrow v]]\!]_{\Pi,\ \mathcal{H}}$$

est valide pour le même σ que pour l'appel de fonction. Autrement dit,

$$\exists\, u_1 \ldots u_k.\ [\![Q[l_i \leftarrow \Pi(l_i)][\backslash\mathsf{result} \leftarrow v]]\!]$$

est valide dans l'état $(\Pi,\ \mathcal{H}_1)$ où $\mathcal{H}_1 = \mathcal{H}[\mathit{Here} := \texttt{store}(\mathcal{H}@Here,\ e,\ m_j,\ u_j)]$. En effet, seule la partie $\mathcal{H}@\mathit{Here}$ du tas mémoire est mise à jour avec les nouvelles valeurs des champs modèles $(e \rightarrow m_j)$. C'est l'invariant global sur le tas mémoire qui nous garantit que le reste du tas mémoire n'est pas modifié. Nous pouvons alors déduire que dans l'état mémoire $(\Pi,\ \mathcal{H}_2)$ tel que $\mathcal{H}_2[\mathit{Here} := \mathcal{H}_1@\mathit{Here},\ L := \mathcal{H}'@L]\ L \neq \mathit{Here}$, la formule

$$\exists\, u_1 \ldots u_k.\ [\![Q[l_i \leftarrow \Pi(l_i),\ \backslash\mathsf{result} \leftarrow v]]\!]$$

est vraie et le tas mémoire ainsi construit, vérifie également l'invariant global.

□

Lemme 4.4.4 (Progrès) *Si la plus faible précondition d'un expression e calculée à partir de n'importe quelle formule Q, est valide dans un environnement de la forme $(\{L \leftarrow \mathcal{H}@L\} \cdot \Pi)$ et que l'invariant global sur les tas mémoire est établi pour $\mathcal{H}$, alors soit e est une valeur, soit il existe un état $(\Pi', \mathcal{H}')$ dans lequel e se réduit, en un pas d'exécution, en e' et $\mathcal{H}'$ vérifie l'invariant global. En d'autres termes, si $\widehat{R}, \Gamma \vdash e : \tau,\ \widehat{R'},\ [\![\mathrm{WP}(e, Q)]\!]$ est valide dans un environnement $(\{L \leftarrow \mathcal{H}@L\} \cdot \Pi)$ et $INV(\widehat{R}, \mathcal{H})$ est vrai, alors soit e est une valeur, soit il existe un état $(\Pi', \mathcal{H}')$ tel que $\Pi,\ \mathcal{H},\ e \rightsquigarrow \Pi', \mathcal{H}',\ e'$ et $INV(\widehat{R'}, \mathcal{H}')$ est vrai.*

Preuve.

– e est un appel de fonction, $e = f(e_1, \ldots e_n)$.
Nous savons que

$$[\![\mathrm{WP}(f(v_1, \ldots, v_n),\ Q)]\!]_{\{L \leftarrow \mathcal{H}@L\}\cdot\Pi}$$

est valide, alors, $[\![\bigwedge_{\substack{t \in Shape(v_1, \ldots v_n, \widehat{R}_{in}) \\ t : \mathbf{struct}_\rho\ S}} \overline{Inv_S}[this \leftarrow t]]\!]_{\Pi,\ \mathcal{H}}$ est vraie. Cette conjonction peut s'écrire sous forme d'une quantification universelle sur les emplacements mémoire :

$$\forall\, l,\ \phi(l)\ \in\ \mathsf{Rdom}(\widehat{R}_{in})\ \wedge\ l : \mathbf{struct}_\rho S \Rightarrow [\![Inv_S(this)]\!]_{\{this \leftarrow l\}, \mathcal{H}}$$

Ce qui est, par construction, équivalent à

$$\forall\, l,\ l\ \in\ \mathtt{dom}(h_1)\ \wedge\ l : \mathbf{struct}_\rho S \Rightarrow [\![Inv_S(this)]\!]_{\{this\leftarrow l\},\mathcal{H}}$$

où h_1 est la partie du tas mémoire qui contient les emplacements mémoire correspondant aux régions qui appartiennent au domaine de $\widehat{R}_{in}$. Formellement,

$$h_1 = \{(l,f) \rightarrow \mathcal{H}@\mathit{Here}(l,f) \mid \phi(l) \in \mathsf{Rdom}(\widehat{R}_{in})\}$$

De plus, $[\![Pre[x_i \leftarrow v_i]]\!]_{\Pi,\ \mathcal{H}}$ est vraie aussi. Nous pouvons alors appliquer à ces deux formules la propriété qui dit qu'évaluer une formule dans un état mémoire donné se ramène à son évaluation dans un état réduit aux variables qui apparaissent libres dans cette formule. Ainsi, La formule $[\![Pre[x_i \leftarrow v_i]]\!]_{\{x_i\leftarrow v_i\},\ \mathcal{H}}$ est vraie. Il existe donc un nouvel état mémoire $(\{x_i \leftarrow v_i\},\ \mathcal{H}')$, où pour tout label L, $\mathcal{H}'@L = h_1$, tel que :

$$\Pi,\ \mathcal{H}, f(v_1,\ldots,v_n) \rightsquigarrow \Pi_{prm},\ \mathcal{H}',\ \mathtt{let}\ v = \mathit{Body}\ \mathtt{in}\ \mathit{return}(v,\ \mathit{Post},\ W,\ A,\ \Pi_{prm},\ \Pi,\ h_0,\ \mathcal{H})$$

où h_0 est le complément de h_1 dans $\mathcal{H}$.

– e est une expression $return$.
Étant donné que

$$[\![\mathrm{WP}(\mathit{return}(v,\ \mathit{Post},\ W,\ A,\ \Pi',\ \{x_i \leftarrow v_i\},\ h_0,\ \mathcal{H}'),\ Q)]\!]_{\{L\leftarrow\mathcal{H}@L\}\cdot\Pi}$$

est valide, alors, par définition,

$$[\![\exists\, u_1 \ldots u_k.\ (\mathit{Post}[x_i \leftarrow v_i]\ \wedge\ \mathit{assigns}(W)\ \wedge\ \mathit{allocates}(A))[\sigma,\ \backslash\mathsf{result} \leftarrow v]]\!]_{\Pi,\ \mathcal{H}}$$

est valide. Ce qui revient-à-dire que les trois évaluations suivantes sont valides.

$$[\![\exists\, u_1 \ldots u_k.\ \mathtt{let}\ \backslash\mathsf{result} = v\ \mathtt{in}\ \mathit{Post}[\sigma]]\!]_{\{x_i\leftarrow v_i\},\mathcal{H}}$$
$$[\![\mathit{assigns}(W)]\!]_{\{x_i\leftarrow v_i\},\mathcal{H}}$$
$$[\![\mathit{allocates}(A)]\!]_{\{x_i\leftarrow v_i\},\mathcal{H}}$$

Nous mettons ensuite à jour les champs modèle comme c'est spécifié dans la substitution σ pour obtenir un nouveau tas $\mathcal{H}_1$. Par conséquent,

$$\exists\, u_1 \ldots u_k.\ [\![\mathtt{let}\ \backslash\mathsf{result} = v\ \mathtt{in}\ \mathit{Post}]\!]_{\{x_i\leftarrow v_i\},\mathcal{H}_1}$$

est valide.
En raisonnant de la même façon que pour l'appel de fonction, nous déduisons à partir de la validité de

$$[\![\bigwedge_{\substack{t\in \mathit{Shape}(v_1,\ldots v_n,\backslash\mathsf{result},\widehat{R}_{out})\\ t:\mathbf{struct}_\rho\ S\\ \exists f,\mathbf{fst}(\widehat{R}_{out}(\rho,f))=\mathbf{true}}} \overline{Inv_S}[this \leftarrow t]]\!]_{\{x_i\leftarrow v_i\},\mathcal{H}_1}$$

que

$$\forall\, l : \mathbf{struct}\ S,\ l \in \mathtt{dom}(\mathcal{H}_1@\mathit{Here}) \Rightarrow [\![Inv_S(this)]\!]_{\{this\leftarrow l\},\mathcal{H}_1}$$

est valide. Par conséquent, il existe un nouvel état mémoire $(\Pi', \mathcal{H}_2)$ où

$$\mathcal{H}_2 = \mathcal{H}'[\mathit{Here} := h_0 \oplus h_1]$$

dans lequel $\mathit{return}(v,\ \mathit{Post},\ W,\ A,\ \Pi',\ \Pi_{prm},\ h_0,\ \mathcal{H}')$ se réduit en la valeur v.

□

4.5 Exemple : Mémoïsation

Avant de conclure ce chapitre, nous montrons un exemple qui illustre des capacités intéressantes de notre approche par raffinement. Cet exemple sera prolongé au chapitre suivant.

Le but de cet exemple est d'utiliser la technique de mémoïsation pour améliorer l'efficacité d'un algorithme donné. Pour l'exemple, nous allons considérer le calcul de la fonction de *Fibonacci* par l'algorithme le plus naïf, qui est exponentiel.

4.5.1 Premier niveau d'abstraction

Au premier niveau d'abstraction, nous pouvons spécifier ce calcul par le module de la figure 4.7 où la fonction logique `math_fibo` est axiomatisée dans la figure 4.8.

```
module Fibonacci

function int fib(int n)
 requires n >= 0;
 ensures \result == math_fibo(n);

end
```

FIGURE 4.7 – Interface simple du module `Fibonacci`

```
theory Fibonacci

 logic int math_fibo(int n);
 axiom fib0 : math_fibo(0) == 0;
 axiom fib1 : math_fibo(1) == 1;
 axiom fibn :
   \forall int n; n >= 2 ==> math_fibo(n) == math_fibo(n−1) + math_fibo(n−2);
end
```

FIGURE 4.8 – Axiomatisation de la fonction $math\_fibo$

L'interface ainsi écrite est trop simple pour permettre une implémentation avec mémoïsation. Nous la modifions comme indiqué dans la figure 4.9, où nous rajoutons une structure `Fibo` passée en paramètre supplémentaire. Cette structure ne contient aucun champ visible, c'est-à-dire aucun champ modèle. Par contre, la clause **writes** indique que ce paramètre est modifié à chaque appel.

4.5.2 Raffinement

L'objectif de ce raffinement est d'implémenter la structure `Fibo` à l'aide d'une table de hachage.

Nous commençons par proposer une interface pour une telle table de hachage donnée dans la figure 4.10.

La structure `Memo` est abstraitement définie par un champ modèle `mapping`. Le type `map` de ce champ est un type abstrait logique représentant une table d'association. La théorie de ce type est donnée dans

```
module Fibonacci

struct Fibo { }

function struct Fibo create_Fibo()
  allocates \result;

function int fib(struct Fibo this, int n)
 requires n >= 0;
 writes *this;
 ensures \result == math_fibo(n);

end
```

FIGURE 4.9 – Interface du module `Fibonacci`

```
struct Memo { model map mapping; }

function struct Memo Memo_create(int n)
   requires n >= 1;
   allocates \result;
   ensures \forall int i; get(\result->mapping , i) == None;

function unit Memo_set(struct Memo this, int k, int v)
   writes *this;
   ensures get(this->mapping, k) == Some(v);
   ensures  \forall int k', v'; k != k' ==>
               get(this->mapping, k') == Some(v') ==>
               get(\old(this)->mapping, k') == Some(v');

function int Memo_get(struct Memo this, int k)
   ensures \result = match get(this->mapping, k) with
                     | None -> -1
                     | Some(v) -> v;
```

FIGURE 4.10 – Interface du module `Memo`

la figure 4.11.

Ce type logique `map` représente une table d'association partielle, ce que nous définissons comme une table associant des entiers à des options d'entiers. Le résultat `None` exprime que cette table n'est pas définie à l'indice considéré.

La fonction `Memo_create` alloue une nouvelle table. La post-condition indique que le `mapping` n'est défini nulle part.

La fonction `Memo_set` ajoute dans la table la valeur `v` à la clé `k`. La spécification indique que toutes les autres associations encore présentes étaient dans la table avant l'appel. Notons que cette spécification autorise l'effacement d'une partie des associations présentes initialement.

```
/*@ axiomatic Map {
  @   type map;
  @   logic integer get_map(map m, integer key);
  @   logic map set_map(map m, integer key, integer value);
  @   logic map constr(integer value);
  @
  @   axiom Const :
  @     \forall integer value, integer key; get_map(constr(value), key) == value;
  @
  @   axiom Select_eq :
  @     \forall map m, integer key1, key2, integer value;
  @    key1 == key2 ==> get_map(set_map(m, key1, value), key2) == value;
  @
  @   axiom Select_neq :
  @     \forall map m, integer key1, key2, integer value;
  @     key1 != key2 ==> get_map(set_map(m, key1, value), key2) == get_map(m, key2);
  @ }
  @*/
```

FIGURE 4.11 – La théorie `Map`.

La fonction `Memo_get` récupère la valeur associée à une clé dans la table. Le résultat `-1` est retourné par défaut.

Nous pouvons maintenant implémenter le calcul de Fibonacci mémoïsé comme indiqué sur la figure 4.12. L'invariant indique que toutes les associations présentes dans la table de mémoïsation sont bien de la forme $(x \to \texttt{math\_fibo}(x))$.

Pour prouver que cette implémentation vérifie sa spécification, il faut vérifier que l'invariant de `Fibo` est rétabli à la fin de l'appel à la fonction `fib`. La structure `Memo` est abstraite, on ne sait pas si elle a des invariants. Cela n'empêche pas de prouver notre module `Fibonacci`. Autrement dit, quelque soit l'implémentation future de `Memo`, on sait que notre module ne pourra pas violer les invariants d'une telle structure.

Nous compléterons cet exemple au chapitre suivant.

4.6 Conclusion et limitations

Nous avons présenté dans ce chapitre une extension du langage noyau en y introduisant les notions de modules, de champs modèles et d'invariants. Comme dans le chapitre 3, nous avons d'abord présenté la syntaxe du langage, puis nous lui avons associé un système de type et une sémantique opérationnelle. L'aspect de notre langage qui consiste à interdire le partage de références est garanti par la définition d'un système de type avec régions.

Pour finir, nous avons redéfini le calcul de plus faible précondition, pour y inclure le concept d'invariants et puis prouvé la correction de ce nouveau calcul en appliquant la même méthode, c'est-à-dire prouver la préservation par réduction et le progrès.

Ce langage présente plusieurs limitations. Le chapitre 5 sera consacré à lever une partie de ses limitations, les quelques limitations restantes seront discutées dans le chapitre de conclusion.

```
module Fibonacci

 use_T Fibonacci;
 use_M Memo;

 struct Fibo{
   struct Memo memo;
 } Invariant Inv_Fibo(struct Fibo this) =
      \forall int x, int y;
         get(this->memo->mapping, x) == Some y ==> y == math_fibo(x);

 function struct Fibo create_Fibo()
  allocates \result;
  { let this = new Fibo in
      this->memo = Memo_create(1000);
      this
  }

function int fib(struct Fibo this, int n)
 requires n >= 0;
 writes *this;
 ensures \result == math_fibo(n);
 { if(n <= 1) then n
   else
     let x = Memo_get(this->memo, n) in
     if (x == -1) then let y = fib(n-1) + fib(n-2) in
                       Memo_set(this->memo, n, y);
                            y
     else x
 }
end
```

FIGURE 4.12 – Implémentation du module `Fibonacci`

Chapitre 5

Extensions du langage

Sommaire

Nous proposons dans ce chapitre trois extensions de notre langage : la première consiste à rajouter les variables mutables, ce qui permettra de définir l'affectation par exemple. En effet, jusqu'à présent, notre langage ne permet de définir des variables immuables qu'à travers l'expression `let` et le seul effet de bord possible est la mise à jour des champs des types structure.

La deuxième extension permet de lever certaines restrictions sur le partage des références qui, rappelons-le, est strictement interdit dans les chapitres précédents.

Enfin, la troisième extension permet d'introduire les tableaux. En effet, tel qu'il est défini dans les deux chapitres précédents, notre langage ne permet pas la gestion de telles structures de données. Il n'est, dans ce cas, pas possible d'écrire des programmes tels que celui des piles, présenté dans la section 2.5.

5.1 Les variables globales

5.1.1 Syntaxe

La syntaxe du langage est modifiée à quatre endroits. Tout d'abord, au niveau des déclarations où nous rajoutons une nouvelle déclaration pour les variables globales. Ensuite, au niveau des termes où

nous pouvons référencer une variable de programme. Au niveau des expressions où nous introduisons l'affectation d'une variable. Enfin, nous rajoutons une nouvelle clause **reads** dans les contrats pour indiquer les variables globales accédées par la fonction.

$decl$	::=	...	
		| **var** $type\ id$	Déclaration de variables
$term$	::=	...	
		| x	Accès à une variable de programme
		| $x@L$	Accès labelisé à une variable
$expr$	::=	...	
		| x	Accès à une variable de programme
		| $x = expr$	Affectation
$contrat$	::=	$requires\_clause^* reads\_clause^*$	
		$assigns\_clause^*\ allocates\_clause^*$	
		$ensures\_clause^*$	
$reads\_clauses$	::=	**reads** $locations$;	

Dans la logique, l'accès à une variable peut être paramétré par un label $x@L$. En effet, on lit la valeur d'une variable dans un état précis du programme. Par défaut, cet état est l'état courant du programme. La notation x est alors une abréviation de $x@\mathit{Here}$.

Remarquons que les variables globales peuvent aussi apparaître dans les clauses **reads**, **writes** et **allocates**.

Si aucune clause **reads** n'est mentionnée dans le contrat, alors l'ensemble des emplacements mémoire visibles par le corps est égal à celui des emplacements mémoire potentiellement modifiables par cette fonction. En effet, pour qu'une fonction puisse modifier la valeur d'un champ, il faut déjà qu'elle puisse y accéder, c'est-à-dire que le champ en question soit visible. Plus généralement, la sémantique que nous allons définir fera que tous les emplacements mémoire mentionnés dans la clause **writes** appartiennent, implicitement, à l'ensemble des emplacements mémoire lus.

Nous nous limitons ici aux variables globales, nous pourrions aussi ajouter des variables mutables locales à une fonction, sans nouvelle difficulté.

5.1.2 Typage

Nous étendons alors le système de type avec les règles d'inférence suivantes :

– Déclaration d'une variable

$$\frac{\Gamma \cdot \{x :\ \mathbf{var}\ \widehat{\tau}\} \vdash \Delta}{\Gamma \vdash \mathbf{var}\ \tau\ x;\ \Delta}$$

où $\widehat{\tau}$ est le type avec région de x, c'est-à-dire que si τ est un type **struct** S, alors $\widehat{\tau} = \mathbf{struct}_r\ S$ où r est une région *fraîche constante*. Le mot clé **var** dans Γ indique que la variable est mutable.

– Accès à une variable dans les termes

$$\frac{L \in \Gamma \qquad x :\ \mathbf{var}\ \tau \in \Gamma}{\widehat{R},\ \Gamma \vdash x@L : \tau}$$

– Accès à une variable dans les expressions

$$\frac{x : \textbf{var}\ \tau \in \Gamma}{\widehat{R},\ \Gamma \vdash x : \tau,\ \widehat{R}}$$

– Affectation

$$\frac{x : \textbf{var}\ \tau \in \Gamma \qquad \widehat{R},\ \Gamma \vdash e : \tau,\ \widehat{R_1}}{\widehat{R},\ \Gamma \vdash x = e : \texttt{unit},\ \widehat{R_1}}$$

Par ailleurs, les clauses **reads** interviennent maintenant dans le typage des déclarations de fonction et des appels de fonctions. Les variables globales accédées ou modifiées sont traitées de la même façon que les paramètres. Ainsi la $shape$ d'entrée $\widehat{R}_{in}$ est l'union disjointe des $shape$ associées aux paramètres et aux variables globales lues. À l'appel de fonction, on doit étendre la règle de typage de la page 107 afin de vérifier, en plus, que les $shape$ instanciées associées aux paramètres sont disjointes des $shape$ associées aux variables globales accédées.

Exemple 5.1.1 *Considérons les fonctions suivantes :*

```
var struct Calc calc_global;

function unit f(struct Calc c)
  reads *calc_global;
  writes *c;
{
  if(mean(calc_global) >= 10) add(c, 10)
}

function unit bad() {
  f(calc_global)
}
```

La déclaration de `f` *est bien typée, mais celle de* `bad` *ne l'est pas car elle identifie la région constante de* `calc_global` *avec la région paramètre associée à* `c`*.*

5.1.3 Sémantique opérationnelle

Un état mémoire est maintenant défini par trois environnements $(\Sigma,\ \Pi,\ \mathcal{H})$, où l'environnement d'évaluation Σ permet d'associer une valeur à chaque variable globale du programme. Dans les assertions, il est souhaitable de parler des valeurs des variables globales dans les états passés, à l'aide de la construction $\backslash$**at**. Ainsi, comme pour le tas mémoire, l'environnement Σ est une collection de tables d'associations indexées par des labels. La valeur d'une variable dans un état L est alors notée $\Sigma@L(x)$.

Sémantique des annotations logiques

Une conséquence de cette extension est qu'il faut distinguer les variables locales stockées dans Π des variables locales stockées dans Σ. L'évaluation d'une variable est dans ce cas :

$$[\![x]\!]_{\Sigma,\,\Pi,\,\mathcal{H}} = \begin{cases} \Sigma@\mathit{Here}(x) & \text{si } x \text{ est mutable} \\ \Pi(x) & \text{sinon} \end{cases}$$

$$[\![x@L]\!]_{\Sigma,\,\Pi,\,\mathcal{H}} = \Sigma@L(x)$$

Si le label est explicite lors de l'accès à une variable x, alors la variable est obligatoirement une variable de programme. L'évaluation d'un tel terme est alors égale à la valeur associée à la variable x dans $\Sigma@L$. Si par contre, le label n'est pas mentionné explicitement, alors l'évaluation d'une variable est égale soit à la valeur qui lui est associée dans l'environnement local Π si c'est une variable logique, ou bien à la valeur qui lui est associée dans $\Sigma@\mathit{Here}$.

Les autres règles restent essentiellement inchangées. L'environnement Σ est simplement rajouté dans l'état dans lequel le terme ou la formule est évalué.

Sémantique des programmes

Dans la sémantique opérationnelle, nous avons, classiquement, deux règles pour l'affectation. En effet, il faut d'abord évaluer la sous expression à droite de l'affectation, et ce n'est que si c'est une valeur que la mise à jour de la variable peut être effectuée.

$$\frac{\Sigma,\ \Pi,\ \mathcal{H},\ e \rightsquigarrow \Sigma',\ \Pi',\ \mathcal{H}',\ e'}{\Sigma,\ \Pi,\ \mathcal{H},\ x = e \rightsquigarrow \Sigma',\ \Pi',\ \mathcal{H}',\ x = e'} \qquad \frac{}{\Sigma,\ \Pi,\ \mathcal{H},\ x = v \rightsquigarrow \Sigma[x \leftarrow v],\ \Pi,\ \mathcal{H},\ ()}$$

Comme pour le typage, la sémantique de l'appel de fonction doit prendre en compte les variables globales accédées comme si elles étaient des paramètres. Autrement dit, les variables globales accédées ou modifiées doivent vérifier leurs invariants à l'entrée et à la sortie de l'appel.

5.1.4 Calcul de la plus faible précondition

Nous étendons le calcul de plus faible précondition, en définissant la plus faible précondition de l'affectation d'une variable de programme.

$$\begin{aligned} \mathrm{WP}(x = v,\ Q) &= Q[\backslash\mathsf{result} \leftarrow (), x \leftarrow v] \\ \mathrm{WP}(x = e,\ Q) &= \mathrm{WP}(\texttt{let}\ v = e\ \texttt{in}\ x = v,\ Q) \end{aligned}$$

La règle de l'appel de fonction doit également être modifiée pour intégrer la clause **reads** de la même façon que les paramètres.

5.1.5 Correction du calcul de plus faible précondition

Nous devons étendre notre preuve de correction du calcul de plus faible précondition aux nouvelles constructions pour les variables globales. Notez que ce langage avec variables globales, auquel on retire les types structure et les fonction, correspond alors au fragment que nous avons formalisé en Why3 [74].

La première étape, dans la preuve de correction du calcul que nous avons défini, consiste à prouver les propriétés de monotonie (lemme 3.4.4) et de distributivité de la conjonction (lemme 3.4.6) pour l'affectation. Les preuves de ces lemmes sont faites par récurrence sur la structure de l'expression. Il est donc impératif de montrer que ces lemmes sont vrais dans le cas de l'affectation. Par ailleurs, l'environnemt d'évaluation a été étendu. Les plus faibles préconditions calculées par la fonction WP sont évaluées dans un environnement composé de Σ et Π, mais ce changement ne remet pas en cause les preuves précédentes.

Preuve de monotonie

- L'affectation d'une variable par une valeur $x = v$.
  Si la formule $[\![P \Rightarrow Q]\!]_{\Sigma,\,\Pi}$ est vraie, alors elle l'est encore si on substitue, dans les formules P et Q, les mêmes variables par les même valeurs. Ainsi,

  $$[\![P[\backslash\mathsf{result} \leftarrow (), x \leftarrow v] \Rightarrow Q[\backslash\mathsf{result} \leftarrow (), x \leftarrow v]]\!]$$

  est vraie, et ce dans n'importe quel environnement d'évaluation. Autrement dit,

  $$[\![\mathrm{WP}(x = v,\ P) \Rightarrow \mathrm{WP}(x = v,\ Q)]\!]_{\Sigma,\,\Pi}$$

  est vraie quelque soient Σ et Π.

- L'affectation d'une variable par une expression quelconque $x = e$.
  Ce cas se ramène à celui d'une expression `let` (Voir 3.4.4).

Preuve de la distributivité de la conjonction

- L'affectation d'une variable par un terme $x = t$.
  Si les formules $[\![\mathrm{WP}(x = t,\ P)]\!]$ et $[\![\mathrm{WP}(x = t,\ Q)]\!]$ sont vraies dans un état $(\Sigma,\ \Pi)$, alors, par définition du calcul de plus faible précondition, $[\![P[\backslash\mathsf{result} \leftarrow (), x \leftarrow t] \wedge Q[\backslash\mathsf{result} \leftarrow (), x \leftarrow t]]\!]$ l'est. Ce qui implique que $[\![\mathrm{WP}(x = t,\ P \wedge Q)]\!]$ est vraie dans $\Sigma,\ \Pi$.

- Le cas de l'affectation d'une variable par une expression quelconque se ramène, dans ce cas aussi, à celui d'une expression `let` (Voir 3.4.4).

Preuve de préservation par réduction

- Réduction d'une affectation de la forme $\Sigma,\ \Pi,\ \mathcal{H},\ x = t \rightsquigarrow \Sigma[x \leftarrow t],\ \Pi,\ \mathcal{H},\ ()$.
  Si $[\![\mathrm{WP}(x = t,\ Q)]\!]_{\Sigma,\,\Pi}$ est valide, alors $[\![Q[\backslash\mathsf{result} \leftarrow (), x \leftarrow t]]\!]_{\Sigma,\,\Pi}$ l'est (par définition). Par conséquent, $[\![Q[\backslash\mathsf{result} \leftarrow ()]]\!]$ est valide dans l'état $(\Sigma,\ \Pi)$ dans lequel la variable de programme x est substituée par t. En d'autres termes, $[\![Q[\backslash\mathsf{result} \leftarrow ()]]\!]_{\Sigma[x\leftarrow t],\,\Pi}$, qui est la définition de $[\![\mathrm{WP}((),\ Q)]\!]_{\Sigma[x\leftarrow t],\,\Pi}$, est valide.

- Réduction d'une affectation de la forme $\Sigma,\ \Pi,\ \mathcal{H},\ x = e \rightsquigarrow \Sigma',\ \Pi',\ \mathcal{H}',\ x = e'$.
  Se ramène au cas d'un `let` (Voir 3.4.5).

Preuve de progrès

Si e est une affectation d'une valeur dans une variable de programme, alors e progresse toujours. Et si e est une affectation d'un expression quelconque, alors on se ramène encore une fois au cas du `let` (Voir 3.4.5).

5.2 Structures de données partageables

Dans la section 4.6, nous avons dit que les restrictions de non aliasing, sur des instances de types structure auquel aucun invariant n'est associé, pouvaient être levées. C'est à cette question que nous allons répondre dans ce qui suit.

Exemple 5.2.1 *Soit les modules* `Car` *et* `Sensor` *correspondant à l'exemple 1.3.2.*

```
module Sensor
 struct Sensor{
   real rpm;
 };

 function unit read(struct Sensor this){
   this->rpm = 2;
 }
end

module Car
 struct Car{
   struct Sensor sensor;
   model int display;
 } Invariant inv(struct Car this) =
      this->display = round(K*(this->sensor->rpm));

unit update(struct Car this){
  f(this->sensor);
  this.display = K * (this->sensor->rpm);
}
end
```

Le bout de programme permettant de partager une instance de $Sensor$ et le champ $sensor$ d'une instance c de `Car`*, et mettant en évidence les problèmes liés à l'aliasing, est repris ci-dessous.*

```
let s = new Sensor in
let c = new Car in
  c->sensor = s;
...
read(s);
```

Supposons maintenant que le champ modifié, par la fonction `read` du module `Sensor` ci-dessus, n'est pas référencé dans l'invariant du type structure `Car`, alors, il pourrait être partagé. Si la structure `Car` contenait, par exemple, un autre champ, alors celui là peut être partagé, car il n'apparaît pas dans l'invariant de `Car`.

Exemple 5.2.2 *Soit* `tempr` *un champ du type structure* `Car`*, de type* ***struct*** `Tempr`*, où* `Tempr` *est un type structure similaire à* `Sensor` *et qui permet de mesurer la température externe.*

Considérons alors le code suivant :

```
let s = new Sensor in
let t = new Temp in
let c = new Car in
  c—>sensor = s;
  c—>tempr = t;
...
reads(t);
```

Contrairement au cas précédent, l'appel de la fonction `read` *du module* `Tempr` *ne viole pas l'invariant de* `c`*.*

On voit bien dans cet exemple, que seul le partage des champs qui apparaissent dans l'invariant associé à un type structure peut entraîner la violation de cet invariant. Autrement dit, un champ qui n'apparaît pas dans l'invariant peut être partagé.

Nous proposons alors de spécifier qu'un champ est partageable de manière explicite, en précédant la déclaration d'un tel champ par le mot clé ***sharable***. Nous autorisons ainsi le partage des pointeurs qui ne sont pas mentionnés dans l'invariant de données du type structure auquel ils appartiennent.

Exemple 5.2.3 *L'exemple du Car s'écrit :*

```
module Car
 struct Car{
   struct Sensor sensor;
   sharable struct Tempr tempr;
   model int display;
 }
```

Nous ne détaillons pas les changements dans le typage, la sémantique ou le calcul de plus faible précondition. Il suffit de considérer que les champs partageables ne sont pas dans la structure dès qu'il s'agit de parler de $shape$ ou d'invariants.

5.3 Les tableaux

Nous introduisons, dans cette section, la structure de tableaux dans notre langage, en distinguant les tableaux dont les éléments sont de type structure, de ceux dont les éléments sont des scalaires.

5.3.1 Les tableaux de structures

Jusqu'à présent, nous avions soit des types scalaires, soit des types structure. Un type **struct** S désigne un pointeur sur un bloc mémoire contenant les champs de S. Nous modifions ce choix en considérant maintenant que **struct** S désigne un pointeur sur un bloc mémoire qui peut contenir un

nombre arbitraire d'instances de cette structure. C'est un tel bloc qu'on appelle *tableau*. Ce n'est pas réellement une modification, mais une extension du cas précédent, où tous les pointeurs pointaient sur des tableaux de taille 1.

Syntaxe

Nous commençons par ajouter dans la syntaxe un moyen de déclarer qu'un champ est un tableau de structures. C'est-à-dire que c'est un pointeur vers un bloc qui contient plus qu'une structure. Nous modifions alors la syntaxe de déclaration des types structures présentée dans le chapitre précédent (figure 4.3) comme suit :

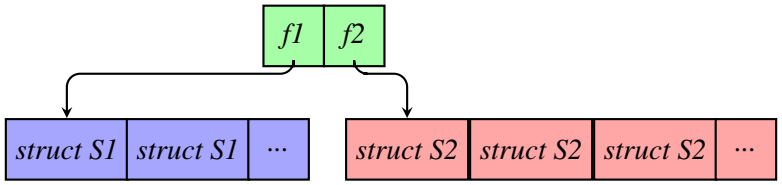
$$\begin{array}{rcl} field & ::= & \ldots \\ & & \mid type\ id \\ & & \mid type_s\ id[\,] \end{array}$$

Notez que cette syntaxe permet de définir de deux manières différentes qu'un champ f est un pointeur vers un bloc de structures S. La forme simple **struct** S impose que le bloc est de taille 1.

Exemple 5.3.1 *La déclaration ci-dessous*

```
struct S{
  struct S1 f1[];
  struct S2 f2[];
}
```

correspond à la représentation mémoire suivante :



Notons que nous ne rajoutons, ainsi, pas d'indirection, comme c'est le cas en C mais pas en Java par exemple.

Maintenant que nous pouvons déclarer un tableau de structures, on doit pouvoir allouer un tel bloc et accéder à ses éléments. Pour l'allocation, nous introduisons une nouvelle expression `new` qui prend en argument le nombre d'emplacements mémoire alloués, c'est-à-dire, la taille du bloc. Les tableaux ont donc une taille fixe établie au moment de l'allocation.

$$\begin{array}{rcll} expr & ::= & \ldots & \\ & & \mid \texttt{new}\ id[n] & \text{Allocation d'un tableau de structures} \\ & & \mid \texttt{new}\ id & \end{array}$$

L'expression `new` S précédente est alors un raccourci pour l'expression `new` $S[1]$.

L'accès et la mise à jour d'un champ d'une structure sont maintenant paramétrés par l'indice de la structure dans le tableau. Ainsi, pour accéder au $i^{\text{ème}}$ élément d'un bloc de structures, il suffit de rajouter

un décalage i au pointeur t sur un tel bloc. Concrètement, on remplace, dans la syntaxe du langage, tous les pointeurs t par $t + i$. Le terme $t + i \rightarrow f$ permet alors d'accéder au champ f du $i^{\text{ème}}$ élément du tableau t. Si le tableau ne contient qu'un seul élément, ce dernier est à l'indice 0. L'accès au champ f d'une telle structure s'écrit $t + 0 \rightarrow f$. On retrouve, alors, bien le cas du chapitre 3.

$$
\begin{array}{rcl}
term & ::= & \ldots \\
 & & \mid term + i \xrightarrow{l} id \\
 & & \mid length(term,\ L) \\
 & & \mid length(term) \\
expr & ::= & \ldots \\
 & & \mid expr + i \xrightarrow{l} id \\
 & & \mid expr + i \rightarrow id\ =\ expr \\
pred & ::= & \ldots \\
 & & \mid \textbf{valid}(term + i,\ L) \\
 & & \mid \textbf{valid}(term + i)
\end{array}
$$

Enfin, on définit une nouvelle fonction logique $length(t)$ qui retourne la longueur du bloc référencé par le pointeur t.

Contrairement au langage C, la syntaxe $t + i$ n'a pas de sens ailleurs qu'à gauche d'une flèche d'accès à un champ ou bien en paramètre du prédicat **valid**. Autrement dit, notre langage ne propose pas d'arithmétique de pointeurs. Les pointeurs pointent toujours au début d'un bloc de mémoire, jamais au milieu.

Typage

Nous présentons dans ce qui suit les règles de typage sur lesquelles les changements introduits ci-dessus ont une conséquence. Le point important dans ces nouvelles règles est que toutes les structures d'un même tableau (bloc mémoire) sont dans la même région. On ne peut pas être plus fin au niveau du typage, car on ne peut pas distinguer statiquement les indices d'un même tableau. Une conséquence est que lorsqu'on doit vérifier un invariant, il faudra le faire pour toutes les cases d'un même tableau en même temps.

Typage des déclarations Lors du typage d'une déclaration d'un type structure, les champs qui sont déclarés comme étant des tableaux de structures sont insérés dans l'environnement de typage en tant que pointeurs sur une structure.

$$
\frac{\widehat{R},\ \{x : \textbf{struct}_\rho\ S\} \vdash p : \texttt{Prop} \qquad \Gamma \cdot \{S : struct\ \tau'_1 \ldots \tau'_n\ \mu_1 \ldots \mu_m,\ p\} \vdash \Delta}{\begin{array}{l}\Gamma \vdash \textbf{struct}\ S = \{\tau_1\ f_1; \ldots; \tau_n\ f_n; \textbf{model}\ \mu_1\ g_1; \ldots; \textbf{model}\ \mu_m\ g_m;\} \\ \qquad\qquad \textbf{Invariant}\ I\ (\textbf{struct}\ S\ x) = p;\ \Delta\end{array}}
$$

où $\tau'_i = \tau_i$, sauf dans le cas où le champ f_i est un tableau, c'est-à-dire qu'on a **struct** $S'\ f_i[]$, alors $\tau'_i = \textbf{struct}\ S'$.

Typage des termes Les règles de typage qui changent par rapport à celles définies dans la section 4.2 sont celles des accès à un champ (modèle ou concret).

$$\frac{\begin{array}{c} l \in \Gamma \qquad (S : struct \dots) \in \Gamma \qquad f \in \mathsf{Fields}(S) \\ \textbf{struct}\ S \in \mathcal{M} \qquad \widehat{R},\ \Gamma \vdash t : \textbf{struct}_\rho\ S \qquad \textbf{snd}(\widehat{R}(\rho,\ f)) = \tau \qquad \widehat{R},\ \Gamma \vdash i : int \end{array}}{\widehat{R},\ \Gamma \vdash (t + i) \xrightarrow{l} f : \tau}$$

$$\frac{\begin{array}{c} l \in \Gamma \qquad (S : struct \dots) \in \Gamma \qquad f \in \mathsf{Models}(S) \\ \widehat{R},\ \Gamma \vdash t : \textbf{struct}_\rho\ S \qquad \textbf{snd}(\widehat{R}(\rho,\ f)) = \tau \qquad \widehat{R},\ \Gamma \vdash i : int \end{array}}{\widehat{R},\ \Gamma \vdash (t + i) \xrightarrow{l} f : \tau}$$

Nous introduisons ensuite deux nouvelles règles de typage associées à la fonction logique $length$

$$\frac{L \in \Gamma \quad (S : struct \dots) \in \Gamma \quad \widehat{R},\ \Gamma \vdash t : \textbf{struct}_\rho\ S}{\widehat{R},\ \Gamma \vdash length(t,\ L) : \texttt{int}}$$

$$\frac{(S : struct \dots) \in \Gamma \quad \widehat{R},\ \Gamma \vdash t : \textbf{struct}_\rho\ S}{\widehat{R},\ \Gamma \vdash length(t) : \texttt{int}}$$

Typage des formules logiques Les seules formules dont la syntaxe a changé sont les deux variantes de **valid**.

$$\frac{L \in \Gamma \quad (S : struct \dots) \in \Gamma \quad \widehat{R},\ \Gamma \vdash t : \textbf{struct}_\rho\ S \qquad \widehat{R},\ \Gamma \vdash i : int}{\widehat{R},\ \Gamma \vdash \textbf{valid}(t + i,\ L) : \texttt{Prop}}$$

$$\frac{(S : struct \dots) \in \Gamma \quad \widehat{R},\ \Gamma \vdash t : \textbf{struct}_\rho\ S \qquad \widehat{R},\ \Gamma \vdash i : int}{\widehat{R},\ \Gamma \vdash \textbf{valid}(t + i) : \texttt{Prop}}$$

Typage des expressions Les règles de typage des expressions qui diffèrent de celles définies dans le chapitre précédent sont :

$$\frac{\begin{array}{c} (S : struct \dots) \in \Gamma \qquad f \in \mathsf{Fields}(S) \\ \textbf{struct}\ S \in \mathcal{M} \qquad \widehat{R},\ \Gamma \vdash e : \textbf{struct}_\rho\ S,\ \widehat{R'} \qquad \textbf{snd}(\widehat{R}(\rho,\ f)) = \tau \qquad \widehat{R},\ \Gamma \vdash i : int \end{array}}{\widehat{R},\ \Gamma \vdash (e + i) \xrightarrow{l} f : \tau,\ \widehat{R'}}$$

$$\frac{\begin{array}{c} (S : struct \dots) \in \Gamma \qquad f \in \mathsf{Fields}(S) \qquad \textbf{struct}\ S \in \mathcal{M} \\ \widehat{R},\ \Gamma \vdash e_1 : \textbf{struct}_\rho S,\ \widehat{R_1} \qquad \widehat{R_1}(\rho,\ f) = (true,\ \tau) \qquad \widehat{R_1},\ \Gamma \vdash e_2 : \tau,\ \widehat{R_2} \qquad \widehat{R},\ \Gamma \vdash i : int \end{array}}{\widehat{R},\ \Gamma \vdash (e_1 + i) \rightarrow f = e_2 : \texttt{unit},\ \widehat{R_2}}$$

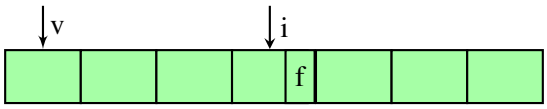
Nous présentons ensuite la règle de typage de la nouvelle expression `new` $S[n]$.

$$\frac{(S : struct \dots) \in \Gamma \qquad \rho \notin \mathsf{Rdom}(\widehat{R})}{\widehat{R},\ \Gamma \vdash \texttt{new}\ \ S[n] : \textbf{struct}_\rho S,\ \widehat{R} \oplus \{(\rho,\ f_i) \rightarrow (true,\ \widehat{type}(f_i, S)) \mid f_i \in \mathsf{Fields}(S)\}}$$

Cette expression modifie $\widehat{R}$ en ajoutant une association entre la région fraîchement allouée ρ et le champ f_i, et le type du champ f_i dans la structure S ($type(f_i,\ S)$). Comme remarqué précédemment, tous les éléments du tableau appartiennent ainsi à une seule et même région.

Sémantique

Les changements appliqués dans la sémantique du langage sont plus conséquents. Le premier changement à effectuer porte sur la structure du tas mémoire qui est non seulement indexé par des pointeurs (chapitres 3 et 4), mais également par un entier qui représente le décalage dans une région. De plus, il est nécessaire de sauvegarder les longueurs des blocs.



Nous adaptons alors les notations précédentes (section 3.3.1) comme suit :

- $\mathcal{H}@L(v, i, f)$ désigne la valeur stockée dans le bloc pointé par v à l'indice i et au champ f, dans $\mathcal{H}@L$,
- $\mathcal{H}@L[v_1,\ i,\ f := v_2]$ désigne la mise à jour de la valeur du champ f à l'indice i dans le bloc pointé par v_1, dans $\mathcal{H}@L$.

Un invariant des fonctions partielles représentant les tas mémoire est que pour tout emplacement mémoire v, l'ensemble des indices i tels que $(v, i, f) \in \mathtt{dom}(H)$ est un intervalle commençant à zéro $0 \ldots l-1$, l s'appelant alors la longueur du bloc et est noté $length(H, v)$.

Sémantique dénotationnelle Nous étendons la sémantique des annotations logiques de notre langage avec l'évaluation de la fonction $length$.

$$\begin{aligned}
[\![length(t,\ L)]\!]_{\Pi,\,\mathcal{H}} &= 1 + max\{i \mid ([\![t]\!]_{\Pi,\,\mathcal{H}},\ i,\ f) \in \mathtt{dom}(\mathcal{H}@L), \\
&\qquad type(t) = \textbf{struct}\ S,\ f \in \mathsf{Fields}(S)\} \\
[\![length(t)]\!]_{\Pi,\,\mathcal{H}} &= 1 + max\{i \mid ([\![t]\!]_{\Pi,\,\mathcal{H}},\ i,\ f) \in \mathtt{dom}(\mathcal{H}@\mathit{Here}), \\
&\qquad type(t) = \textbf{struct}\ S,\ f \in \mathsf{Fields}(S)
\end{aligned}$$

max est la fonction qui retourne la plus grande valeur de i pour laquelle $([\![t]\!]_{\Pi,\,\mathcal{H}},\ i,\ f)$ appartient bien au domaine du tas mémoire.

Sémantique opérationnelle Les sémantiques opérationnelles de l'accès et de la mise à jour d'un champ sont similaires à celles définies dans le chapitre précédent. La seule différence est qu'il faut vérifier que le décalage par rapport au pointeur sur le bloc de structures est bien compris entre 0 et la taille de ce bloc.

$$\frac{(v, i, f) \in \mathtt{dom}(\mathcal{H}@L) \qquad v' = \mathcal{H}@L(v,\ i,\ f)}{\Pi,\ \mathcal{H},\ (v+i) \xrightarrow{L} f \rightsquigarrow \Pi,\ \mathcal{H},\ v'}$$

$$\frac{(v_1,\ i) \in \mathtt{dom}(\mathcal{H}@\mathit{Here}) \qquad \mathcal{H}' = \mathcal{H}[\mathit{Here} := \mathcal{H}@\mathit{Here}(v_1,\ i,\ f,\ v_2)]}{\Pi,\ \mathcal{H},\ (v_1+i) \to f = v_2 \rightsquigarrow \Pi,\ \mathcal{H}',\ ()}$$

L'expression `new` alloue un nouvel emplacement mémoire loc et étend le tas mémoire en initialisant

les champs de chaque structure du bloc.

$$\frac{\begin{array}{c} n > 0 \qquad loc = \texttt{fresh}(\mathit{Here}) \qquad \mathcal{H}' = \mathcal{H}[\mathit{Here} := \mathit{Here}'] \\ \text{où } \mathit{Here}' = \{(loc,\ i,\ f) \leftarrow d \mid 0 \leq i < n,\ f \in \mathsf{Fields}(S),\ d = default(S,\ f)\} \oplus \mathcal{H}@\mathit{Here} \end{array}}{\Pi,\ \mathcal{H}, \texttt{new}\ S[n] \rightsquigarrow \Pi, \mathcal{H}',\ loc}$$

La dernière expression affectée par l'introduction des tableaux est l'appel de fonction. En effet, dans le chapitre précédent, nous nous assurons, avant chaque appel et retour de fonction f, que les invariants des objets qui sont dans des régions accessibles par le corps de f, sont établis. Cette propriété est toujours requise, mais contrairement au cas précédent, une région ne contient plus une unique instance d'un certain type structure, mais un bloc d'instances. Par conséquent, dans chaque région, il faut vérifier autant d'invariants que d'objets dans le tableau et ce même si un seul objet a été mis à jour. Ainsi, à chaque fois que la valeur d'un des champs de $(t+i)$ est mise à jour, nous vérifions que tous les invariants dans la région qui contient $(t + i)$ sont vrais. D'où la quantification universelle sur i dans la prémisse permettant d'établir la validité des invariants de tous les pointeurs qui appartiennent au domaine de h_1. La sémantique opérationnelle de l'appel de fonction et du $return$ décrivant un tel comportement est décrit ci-dessous.

$$\frac{\begin{array}{c} \Pi_{prm} = \{x_i \leftarrow v_i\} \qquad [\![Pre]\!]_{\Pi_{prm},\mathcal{H}} \text{ est valide} \\ h_1 = \{(l,f) \rightarrow \mathcal{H}@\mathit{Here}(l,f) \mid \phi(l) \in \mathsf{Rdom}(\widehat{R}_{in})\} \qquad \mathcal{H}@\mathit{Here} = h_0 \oplus h_1 \qquad \mathcal{H}' = \{\mathit{Here} \leftarrow h_1\} \\ \forall\, l : \textbf{struct}\ S,\ l \in \texttt{dom}(h_1) \Rightarrow \forall\, i, 0 \leq i < length(l) \Rightarrow [\![Inv_S(this)]\!]_{\{this \leftarrow (loc\ +\ i)\},\mathcal{H}} \end{array}}{\Pi,\ \mathcal{H}, f(v_1, \ldots, v_n) \rightsquigarrow \Pi_{prm},\ \mathcal{H}', \mathit{Old} :\ \texttt{let}\ v = Body\ \texttt{in}\ return(v,\ Post, W,\ A,\ \Pi,\ \Pi_{prm},\ h_0,\ \mathcal{H})}$$

$$\frac{\begin{array}{c} h_1 = \mathcal{H}@Here[(l,i,m) := u_{(l,i,m)} \mid \phi(l) \in \mathsf{Rdom}(\widehat{R}_{out})\ \wedge\ \textbf{fst}(\widehat{R}_{out}(\phi(l),\ m)) = true] \\ \mathcal{H}_1 = \mathcal{H}[\mathit{Here} := h_1] \qquad [\![\texttt{let}\ \backslash\mathsf{result} = v\ \texttt{in}\ Post]\!]_{\Pi_{prm},\mathcal{H}_1} \\ Assigns(\mathcal{H}_1,\ [\![W]\!]^{loc}_{\Pi_{prm},\mathcal{H}_1}) \qquad Allocates(\mathcal{H}_1,\ [\![A]\!]^{alloc}_{\Pi_{prm},\mathcal{H}_1}) \\ \forall\, l : \textbf{struct}\ S,\ l \in \texttt{dom}(h_1) \Rightarrow \forall\, i, 0 \leq i < length(\mathcal{H}@\mathit{Here},\ l)[\![Inv_S(this)]\!]_{\{this \leftarrow (loc\ +\ i)\},\mathcal{H}} \end{array}}{\Pi,\ \mathcal{H},\ return(v,\ Post,\ W,\ A,\ \Pi',\ \Pi_{prm},\ h_0,\ \mathcal{H}') \rightsquigarrow \Pi',\ \mathcal{H}'[\mathit{Here} := h_0 \oplus h_1],\ v}$$

Calcul de la plus faible précondition

Afin de pouvoir étendre notre calcul de plus faible précondition, nous adaptons notre modèle mémoire, en respectant la structure du tas mémoire décrite dans la section 5.3.1

Modèle mémoire Nous modifions l'axiomatisation présentée dans la section 3.4.1 de la façon suivante :

$$\begin{array}{l} \texttt{select} :\ heap \rightarrow location \rightarrow int \rightarrow field \rightarrow value \\ \texttt{store} :\ heap \rightarrow location \rightarrow int \rightarrow field \rightarrow value \rightarrow heap \\ \texttt{alloc} :\ heap \rightarrow int \rightarrow location * heap \\ \texttt{length} : heap \rightarrow location \rightarrow int \end{array}$$

Les fonctions `select` et `store` prennent un argument supplémentaire i qui est l'indice auquel il faut accéder dans le bloc mémoire. La fonction `alloc` prend en argument la taille du bloc à allouer. Enfin, une nouvelle fonction `length` remplace l'utilisation de `valid` : à la place de $\texttt{valid}(L,t)$ qui représentait $\textbf{valid}(t, L)$, nous représentons $\textbf{valid}(t + i, L)$ par $0 \leq i < \texttt{length}(L,\ t)$.

Accès à un champ :

$$\mathrm{WP}((v+i) \to f,\ Q) = 0 \leq i < \mathit{length}(\mathit{Here},\ v) \wedge Q[\backslash\mathsf{result} \leftarrow \mathtt{select}(\mathit{Here},\ i,\ v,\ f)]$$

Mise à jour d'un champ :

$$\begin{array}{l}\mathrm{WP}(v_1 + i \to f = v_2,\ Q) = \\ \qquad 0 \leq i < \mathit{length}(\mathit{Here},\ v_1) \wedge Q[\backslash\mathsf{result} \leftarrow (), \mathit{Here} \leftarrow \mathtt{store}(\mathit{Here},\ i,\ v_1,\ f,\ v_2)]\end{array}$$

Allocation : La plus faible précondition de l'allocation n'est pas modifiée. C'est le comportement de la fonction logique `alloc` qui est adapté. En effet, au lieu d'allouer un seul emplacement mémoire, cette fonction alloue un bloc de taille n fois la taille du type structure.

$$\mathrm{WP}(\mathtt{new}\ S[n],\ Q) = n > 0\ \wedge\ \mathtt{let}\ l = \mathtt{alloc}(\mathit{Here}, n)\ \mathtt{in}\ Q[\backslash\mathsf{result} \leftarrow \mathbf{fst}(l),\ \mathit{Here} \leftarrow \mathbf{snd}(l)]$$

Appel de fonction : Dans le chapitre précédent, une région contenait un seul et unique pointeur. Il suffisait de vérifier l'invariant de l'objet contenu dans chacune des régions qui appartiennent au domaine de $\widehat{R}$. Avec cette extension, pour chaque région accessible, il faut parcourir l'ensemble des objets et vérifier leurs invariants.

$$\begin{array}{l}\mathrm{WP}(f(v_1, \ldots, v_n),\ Q) = \\ \quad \overline{\mathit{Pre}}[x_i \leftarrow v_i]\ \wedge \bigwedge_{\substack{t \in \mathit{Shape}(v_1,\ldots,v_n,\ \widehat{R}_{in}) \\ t:\mathbf{struct}_\rho S}} (\forall\ i,\ 0 \leq i < \mathtt{length}(\mathit{Here}, t) \Rightarrow \overline{\mathit{Inv}_S}[\mathit{this} \leftarrow (t+i)]) \wedge \\ \quad \forall \mathit{Here}\ \mathit{Old},\ (\overline{\mathit{Post}}[x_i \leftarrow v_i]\ \wedge\ \mathit{assigns}(W)\ \wedge\ \mathit{allocates}(A) \wedge \\ \qquad \bigwedge_{\substack{t \in \mathit{Shape}(v_1,\ldots,v_n,\ \backslash\mathsf{result},\ \widehat{R}_{out}) \\ t:\ \mathbf{struct}_\rho S \wedge \\ \exists f, \mathbf{fst}(\widehat{R}_{out}(\rho,f))=\mathit{true}}} (\forall\ i,\ 0 \leq i < \mathtt{length}(\mathit{Here}, t) \Rightarrow \overline{\mathit{Inv}_S}[\mathit{this} \leftarrow (t+i)]) \\ \qquad\qquad \Rightarrow Q)[\mathit{Old} \leftarrow \mathit{Here}]\end{array}$$

où $\widehat{R}_{in}$ et $\widehat{R}_{out}$ sont les *shape* calculées lors du typage de la définition de la fonction f.

La pseudo-expression «return»

$$\begin{array}{l}\mathrm{WP}(\mathit{return}(v,\ \mathit{Post},\ W,\ A,\ \Pi,\ \{x_i \leftarrow v_i\},\ h_0,\ \mathcal{H}),\ Q) = \\ \quad \exists\ u_1 \ldots u_k.\ (\overline{\mathit{Post}}[x_i \leftarrow v_i]\ \wedge \bigwedge_{\substack{t \in \mathit{Shape}(v_1,\ldots v_n, \backslash\mathsf{result}, \widehat{R}_{out}) \\ t:\mathbf{struct}_\rho\ S \wedge \\ \exists f, \mathbf{fst}(\widehat{R}_{out}(\rho,f))=\mathit{true}}} \bigwedge_{i=0}^{\mathit{length}(\mathit{Here},t)-1} \overline{\mathit{Inv}_S}[\mathit{this} \leftarrow (t+i)]) \wedge \\ \qquad\qquad \mathit{assigns}(W)\ \wedge\ \mathit{allocates}(A)\ \wedge Q[l_i \leftarrow \Pi(l_i)] \\ \qquad)[\sigma,\ \backslash\mathsf{result} \leftarrow v]\end{array}$$

Correction du calcul de plus faible précondition

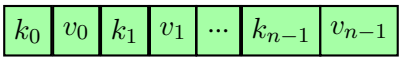
$$
\begin{array}{l}
OP(prog) \stackrel{def}{=} \bigwedge_{f \in prog} \forall \vec{x}.\, \forall\, \mathit{Here}\ \mathit{Old}.\, (\overline{Pre_f} \wedge \\
\qquad \bigwedge_{\substack{t \in Shape(x_1,\ldots x_n,\backslash\mathsf{result},\widehat{R}_{in}) \\ t:\mathbf{struct}_\rho\ S}} (\forall\, i,\ 0 \leq i < \mathtt{length}(\mathit{Here}, t) \Rightarrow \overline{Inv_S}[this \leftarrow (t+i)])) \Rightarrow \\
\qquad \exists\, u_1 \ldots u_k.\, \mathrm{WP}(Body_f,\ \overline{Post_f} \wedge assigns(W_f)\ \wedge\ allocates(A_f)\ \wedge \\
\qquad\qquad \bigwedge_{\substack{t \in Shape(x_1,\ldots x_n,\backslash\mathsf{result},\widehat{R}_{out}) \\ t:\mathbf{struct}_\rho\ S\ \wedge \\ \exists f, \mathbf{fst}(\widehat{R}_{out}(\rho,f))=true}} (\forall\, i,\ 0 \leq i < \mathtt{length}(\mathit{Here}, t) \Rightarrow \overline{Inv_S}[this \leftarrow (t+i)])) \\
\qquad [\sigma,\ \mathit{Old} \leftarrow \mathit{Here}\ ]
\end{array}
$$

La correction de ce calcul de plus faible précondition (préservation par réduction et progrès) est directement adaptée de la preuve du chapitre précédent. Il suffit de remplacer ce qui concerne une structure donnée par ce qui concerne l'ensemble des cellules d'un tableau.

5.3.2 Suite de l'exemple de mémoïsation

Nous illustrons les tableaux de structures en fournissant une implémentation du module `Memo` présenté dans la section 4.5.

La figure 5.1 présente une implémentation de la structure `Memo` sous la forme d'un tableau de paires d'entiers :



L'invariant de collage exprime que pour tout i, la valeur de k_i modulo la taille du tableau est i, et que si v_i est positif ou nulle, alors la paire (k_i, v_i) est une association valide de cette table.

L'implémentation des trois fonctions de ce module est donnée dans la figure 5.2.

Notons que les obligations de preuve associées à `Memo_set` et `Memo_create` requièrent d'établir l'invariant de `Memo` qui porte sur le tableau complet.

5.3.3 Tableaux de scalaires

Le deuxième type de tableaux que nous introduisons sont les tableaux de scalaires, c'est-à-dire des tableaux dont les éléments sont des int, des $bool$ ou bien des $real$.

La syntaxe des champs de structures est encore une fois modifiée pour permettre la déclaration de tableaux de n'importe quelle sorte : tableaux de scalaires ou tableaux de structures

$$
\begin{array}{rcl}
field & ::= & \ldots \\
 & & \mid type\ id([\,])^?
\end{array}
$$

```
module Memo

 struct Data {
  int key;
  int value;
 }

 struct Memo {
   struct Data data[];
   int size;
   model map mapping;
 } Invariant Inv_Memo(struct Memo this) =
     this->size == length(this->data) && this->size >= 1 &&
     \forall int i; 0 <= i < this->size =>
        let k = data+i->key in k % size == i &&
        let v = data+i->value in
          get(this->mapping, i) == if v >= 0 then Some(v) else None;
```

FIGURE 5.1 – Déclaration du type structure `Memo`

```
function struct Memo Memo_create(int n){
  var int i;
  let hash = new Memo in
    hash->data = new Data[n];
    hash->size = n;
    i = 0;
    while (i < n) do
      hash->(data+i -> key) = i;
      hash->(data+i -> value) = -1;
      i = i + 1;
    end;
    hash
}

function unit Memo_set(struct Memo this, int k, int v) {
  let i = k % this->size in
     (this->data)+i->key = k;
     (this->data)+i->value = v
}

function int Memo_get(struct Memo this, int k) {
  let i = k % this->size in
  let k' = (this->data)+i->key in
     if k == k' then (this->data)+i->value
     else -1
}
```

FIGURE 5.2 – Définitions des fonctions du module `Memo`

Nous définissons, par ailleurs, trois types structure Int, $Bool$ et $Real$ correspondant aux types primitifs int, $bool$ et $real$ respectivement. Ces types structures ne contiennent qu'un seul champ qui représente la valeur du scalaire associé.

$$\begin{array}{lll} \textbf{struct} & Int & \{\ int\ value;\ \} \\ \textbf{struct} & Bool & \{\ bool\ value;\ \} \\ \textbf{struct} & Real & \{\ real\ value;\ \} \end{array}$$

L'entier 3 est, par exemple, représenté par une instance de type Int dont le champ value contient la valeur 3.

Avec une représentation des scalaires, un tableau de scalaires est considéré comme un tableau de structures Int, $Bool$ ou $Real$. Nous nous ramenons ainsi au cas précédent. Du point de vu de l'utilisateur, un tableau de scalaire est déclaré comme suit : $scalaire\ t$ [ ], ce n'est qu'au moment de typer une telle déclaration, que le type est transformé en pointeur vers la structure de données adéquate.

$$\frac{\widehat{R},\ \{x : \textbf{struct}_\rho\ S\} \vdash p : \texttt{Prop} \qquad \tau_i \in \{\textbf{int}, \textbf{real}, \textbf{bool}\} \qquad \Gamma \cdot \{S : struct\ \tau'_1 \ldots \tau'_n\ \mu_1 \ldots \mu_m,\ p\} \vdash \Delta}{\Gamma \vdash \textbf{struct}\ S = \{\tau_1\ f_1; \ldots \tau_i\ f_i[\,]; \ldots; \tau_n\ f_n; \textbf{model}\ \mu_1\ g_1; \ldots; \textbf{model}\ \mu_m\ g_m;\} \quad \textbf{Invariant}\ I\ (\textbf{struct}\ S\ x) = p;\ \Delta}$$

avec $\tau'_i = \tau_i$, si c'est un scalaire, $\tau'_i = \textbf{struct}\ S'$ si f_i est un tableau de structures et $\tau'_i = \varphi(\tau_i)$ si f_i est un tableau de scalaires où φ est une fonction qui associe à un type scalaire, la structure de données correspondante.

$$\begin{array}{lcl} \varphi(int) & = & \textbf{struct}\ Int \\ \varphi(real) & = & \textbf{struct}\ Real \\ \varphi(bool) & = & \textbf{struct}\ Bool \end{array}$$

Exemple 5.3.2 *Soit la structure de données suivante :*

***struct** S { int t[] ; }*

*Alors l'environnement de typage contient : (S :struct (**struct** Int))*

5.3.4 Passage en paramètre d'une cellule de tableau

Une limitation de notre extension aux tableaux est qu'il n'est pas possible d'appeler une fonction directement sur l'une des structures d'un tableau de structures. Par exemple, supposons l'existence d'une fonction dont le profil est :

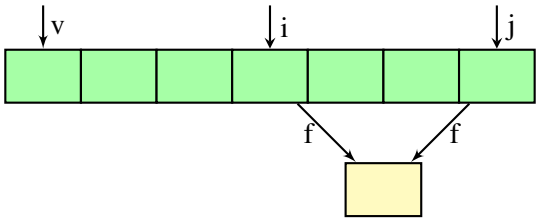
```
function unit f(struct Data d);
```

où `Data` est la structure définie dans notre exemple de mémoïsation, alors on ne peut pas appeler `f` sur la i^{me} cellule de `memo->data`. La syntaxe du langage ne le permet pas.

Supposons donc que l'on étende la syntaxe pour autoriser la forme $e + i$ en paramètre de fonction, ce qui nous permettrait d'écrire : `f((memo->data)+i)`.

La difficulté principale de cette extension est qu'il n'est pas évident d'assurer que les invariants des autres cellules du tableau sont préservés. De façon générale, si `f` prend un paramètre `x` de type **`struct S`**, et si l'on appelle `f(e+i)`, alors `f` peut modifier la i^{me} cellule du tableau `e`. Si la structure `S` a un invariant,

alors `f` rétablira l'invariant de `e + i`, mais rien ne garantit que les invariants des autres cases sont encore valides. Ce cas peut se produire si les cases du tableau partagent des données, ce qui est possible car toutes les cases sont dans la même région.



Une solution possible serait de prouver à nouveau les invariants de toutes les structures du tableau après l'appel à f. Mais une telle solution n'est pas possible pour des questions de modularité : l'invariant de la structure S n'est pas forcément visible par f.

Pour éviter la situation précédente, on impose la règle suivante lorsque l'on appelle une fonction sur un paramètre de la forme $e + i$ où e a le type **struct** S : tous les champs non scalaires de S sont partageables. Autrement dit, l'invariant de S ne peut mentionner que des champs scalaires de S. Dans ces conditions, le partage ne peut pas conduire à la violation de l'invariant.

Variante de l'implémentation de `Memo`

Nous proposons ici une variante de l'implémentation de `Memo` dans laquelle la table de hachage est également implémentée à l'aide d'un tableau de structures, mais où les types `Data` et `Memo` sont définis dans deux modules différents (Voir les figures 5.3 et 5.4).

FIGURE 5.3 – Définition du module `Data`

Nous définissons, par ailleurs, dans le module `Data` les fonctions `get_key` et `get_value` qui retournent respectivement le champ `key` et `value` d'une instance de type `Data`, et la fonction `set` qui affecte les deux champs.

FIGURE 5.4 – Nouvelle déclaration du type structure `Memo`

La définition du module `Memo` est alors modifiée, car les champs concrets du type `Data` ne sont pas visibles. D'abord au niveau de l'invariant du type `Memo`, où il n'est plus possible de référencer les champs `key` et `value` (figure 5.4), puis au niveau des définitions des fonctions `Memo_create`, `Memo_set` et `Memo_get`, comme indiqué dans la figure 5.5

Notons que les fonctions du module `Data` sont, à chaque fois, appelées sur une cellule `i` du tableau. Mais comme l'invariant de `Data` ne référence que des champs scalaires, alors il n'y pas de risque que les invariants des autres structures `data+j` pour `j` $\neq$ `i`, soient rompus.

```
function struct Memo Memo_create(int n){
  var int i;
  let hash = new Memo in
    hash->data = new Data[n];
    hash->size = n;
    i = 0;
    while (i < n) do
      set(hash->data+i, i, -1);
      i = i + 1;
    end;
    hash
}

function unit Memo_set(struct Memo this, int k, int v) {
  let i = k % this->size in set(hash->data+i, k, v)
}

function int Memo_get(struct Memo this, int k) {
  let i = k % this->size in
  let k' = get_key(this->data+i) in
     if k == k' then get_value(this->data+i)
     else -1
}
```

FIGURE 5.5 – Définitions des fonctions du module `Memo`.

5.4 Conclusion

Nous avons présenté dans ce chapitre des extensions de notre langage qui permettent de gérer les variables de programmes ainsi que les tableaux. Nous avons, par ailleurs, partiellement levé la restriction qui consiste à interdire tout aliasing, en autorisant le partage de pointeurs sur des types structure si ces dernier ne sont pas munis d'un invariant. Nous obtenons ainsi un langage assez riche pour pouvoir écrire des programmes intéressants, ce que nous allons voir dans le chapitre suivant.

Chapitre 6

Implémentation et études de cas

Sommaire

Nous présentons dans ce chapitre une façon d'implémenter notre approche puis une réalisation de cette implémentation, qui consiste à développer un greffon *Frama-C*.

Nous illustrons ensuite cette réalisation sur deux études de cas concrètes issues du challenge Vacid0 [70] : les tableaux creux (en Anglais Sparse Array) et les tas binaires.

Rappelons que les benchmarks Vacid-0 forment une collection de petits programmes, dont les énoncés sont présentés dans une syntaxe proche de celle de Java, qui posent des défis pour la vérification formelle de leur comportement fonctionnel, en particulier : invariants de données et abstraction des données.

Pour chaque structure, nous commençons par présenter, dans le langage source, le programme qui décrit la structure que nous souhaitons étudier. Nous prouvons ensuite l'implantation de la structure, dans un premier temps, puis un programme client du module, dans un deuxième temps.

6.1 Implémentation par traduction dans le langage cible

La présentation théorique de notre approche développée dans les deux chapitres précédents devrait nous amener à implémenter un nouveau calcul de plus faible précondition pour des programmes C. Néanmoins, le calcul de plus faible précondition de référence pour un langage à pointeurs que nous introduisons dans le chapitre 3 est naïf, car il modélise un tas mémoire par une seule variable logique. L'efficacité, en pratique, des greffons comme Jessie et WP, vient de leurs modèles mémoire plus fins, qui utilisent plusieurs variables logiques pour modéliser les parties du tas mémoire que l'on sait être forcément disjointes (par exemple, une variable par champ de structure). Afin de pouvoir réutiliser toutes l'expérience acquise dans le développement de ces greffons, nous avons choisi de ne pas implémenter notre calcul de plus faible précondition directement, mais de procéder par traduction du code source annoté vers un code C intermédiaire, qui ne contient plus de champs modèles, ni d'invariants, et peut donc être traité par Jessie ou WP. Cette traduction est présentée dans la suite de cette section. L'objectif est que : si les obligations de preuves générées par le calcul de plus faible précondition (Jessie ou WP) sur le code cible sont valides, alors les obligations de preuve générées par notre propre calcul de plus faible précondition le sont aussi. Plus précisément, l'obligation de preuve associée à une fonction f du programme cible devra correspondre à la formule (4.1) de la page 116. Cette affirmation n'est pas formellement prouvée, mais nous pensons que la traduction présentée ci-dessous est suffisamment simple pour s'en convaincre.

Le but de cette implémentation est donc de traduire un programme écrit dans le langage décrit dans la section 4.1(langage source) dans un autre langage (langage cible). Concrètement, il s'agit de transformer un programme qui contient n modules et m théories, en n programmes cibles. À chaque module $\mathcal{M}$, dans le programme source, nous associons un programme cible composé des théories du programme, du module $\mathcal{M}$ et des interfaces des $(n-1)$ autres modules. Si P est le programme source suivant :

$$\textbf{theory}_1 \ \ldots \ \textbf{theory}_m \ \textbf{module}_1 \ \ldots \ \textbf{module}_i \ \ldots \ \textbf{module}_n$$

alors le i^{me} programme cible P_i est de la forme :

$$\begin{array}{rcl} \mathcal{TRD}_{\mathcal{P}}(\textbf{module}_i,\ P) & = & \textbf{theory}_1 \ \ldots \ \textbf{theory}_m \\ & & \text{interface}(\textbf{module}_1) \ \ldots \ \text{interface}(\textbf{module}_{i-1}) \\ & & \textbf{module}_i \\ & & \text{interface}(\textbf{module}_{i+1}) \ \ldots \ \text{interface}(\textbf{module}_n) \end{array}$$

où $\mathcal{TRD}_{\mathcal{P}}$ est la fonction de traduction du programme source vers un programme cible. Ainsi, traduire un tel programme P consiste à traduire chacun de ses composants. Notez que nous avons trois sortes de traductions :

- La traduction des théories,
- La traduction du module **module**$_i$ que nous appellerons *traduction concrète*,
- La traduction des $(n-1)$ autres modules que nous appellerons *traduction abstraite*.

Nous commençons par présenter la fonction de traduction des théories, puis celle des modules. Pour éviter les redondances, nous regrouperons dans la même section les deux traductions abstraite et concrète.

6.1.1 Traduction des théories

La fonction qui permet de traduire les théories est en fait la fonction identité. En effet, les théories sont recopiées telles quelles dans le programme cible.

$$\mathcal{TRD}_{\mathcal{TH}}(\textbf{theory}\ id\ logic\_decl^*\ \textbf{end}) \quad = \quad \textbf{theory}\ id\ logic\_decl^*\ \textbf{end}$$

6.1.2 Traduction des modules

Traduire un module revient à traduire les déclarations qui le composent. Mais, la manière dont ces déclarations sont transformées dépend du fait qu'elles appartiennent ou pas au module courant. Le module courant, qu'on note $\mathcal{M}$, est le module en paramètre de la fonction de traduction.

Nous définissons alors deux fonctions de traduction $\mathcal{ABS}$ et $\mathcal{CCR}$ qui effectuent, respectivement, la traduction abstraite et concrète des déclarations d'un module. La transformation d'un module s'écrit alors :

$$\mathcal{TRD}_{\mathcal{M}}(\textbf{module } M\ decl^*\ \textbf{end}) \quad = \quad \begin{cases} \textbf{module } M'\ \mathcal{ABS}(decl)^*\ \textbf{end} = \text{interface}(M') & \text{Si } M \neq M' \\ \textbf{module } M\ \mathcal{CCR}(decl)^*\ \textbf{end} & \text{Sinon} \end{cases}$$

Comme son nom l'indique, une traduction abstraite permet d'abstraire une déclaration donnée, c'est-à-dire que, contrairement à une traduction concrète, le résultat d'une telle transformation ne contient que des données abstraites ou indépendantes de l'implémentation. Le résultat de ce type de transformation est une interface du module traduit.

Nous allons, dans ce qui suit, décrire la traduction de chaque type de déclaration.

Traduction des déclarations des types structure

La traduction d'un type structure S se fait en deux étapes. La première étape consiste à générer deux prédicats Inv_S et $Valid\_S$ dont les définitions dépendent du type de la traduction. Le premier, Inv_S, est l'invariant de collage associé au type structure S. Le second prédicat, $Valid\_S$, permet d'établir qu'une instance de type structure est valide. Or, tester la validité d'un pointeur sur un type structure qui n'appartient pas au module courant revient à tester simplement si ce pointeur est bien alloué en mémoire. Dans ce cas, le prédicat Inv_S n'a pas besoin d'être généré, car l'invariant de collage d'un type structure n'est pas visible de l'extérieur du module dans lequel il est défini. Par contre, si le type structure appartient au module courant, alors il faut non seulement s'assurer que le pointeur est alloué mais en plus que l'objet pointé vérifie bien son invariant de collage.

La deuxième étape de traduction porte sur les champs d'un type. Pour les champs modèles, nous appliquons les mêmes transformations dans les deux traductions. Ces champs sont transformés en de simples champs (en supprimant le mot clé **model**), mais dans le cas des champs concrets, nous distinguons bien les deux traductions : les champs concrets sont maintenus lors d'une traduction concrète et supprimés au cours d'une traduction abstraite. En effet, ces champs sont propres à une implémentation donnée. Ils ne peuvent de ce fait pas apparaître dans l'interface du module.

$$\mathcal{ABS}\left(\begin{array}{l} \textbf{struct } S = \{\tau_1 f_1; \ldots; \tau_n f_n; \\ \qquad\qquad \textbf{model } \rho_1 m_1; \ldots; \\ \qquad\qquad \textbf{model } \rho_m m_m; \} \\ \textbf{Invariant } Inv\_S\ (\textbf{struct } S\ t) = p \end{array}\right) \quad = \quad \begin{array}{l} \textbf{struct } S = \{\rho_1 m_1; \ldots; \rho_m m_m; \}; \\ \textbf{predicate } Valid\_S(\textbf{struct } S\ t) = \\ \qquad\qquad\qquad\qquad \textbf{valid}(t); \end{array}$$

$$\mathcal{CCR}\left(\begin{array}{l}\textbf{struct } S = \{\tau_1 f_1; \ldots; \tau_n f_n; \\ \qquad\qquad \textbf{model } \rho_1 m_1; \ldots; \\ \qquad\qquad \textbf{model } \rho_m m_m; \} \\ \textbf{Invariant } Inv\_S\ (\textbf{struct } S\ t) = p\end{array}\right) = \begin{array}{l}\textbf{struct } S = \{\tau_1 f_1; \ldots; \tau_n f_n;\ \rho_1 m_1; \ldots;\ \rho_m m_m; \}; \\ \textbf{predicate } Inv\_S(\textbf{struct } S\ t) = p; \\ \textbf{predicate } Valid\_S(\textbf{struct } S\ t)\ = \\ \qquad\qquad \textbf{valid}(t) \wedge Inv\_S(t);\end{array}$$

Traduction des définitions de fonctions

Traduire une définition de fonction consiste à traduire, en premier lieu, son contrat. Cette traduction réside dans la transformation de chacune des clauses qui composent le contrat.

$$\mathcal{TRD}_{\mathcal{C}}\left(\begin{array}{ll}\textbf{requires} & Pre \\ \textbf{reads} & R_1, \ldots, R_n \\ \textbf{allocates} & A_1, \ldots A_m \\ \textbf{writes} & W_1, \ldots, W_k \\ \textbf{ensures} & Post\end{array}\right) = \begin{array}{ll}\textbf{requires} & \mathcal{TRD}_{\mathcal{R}}(Pre) \\ \textbf{reads} & \mathcal{TRD}_{\mathcal{L}}(R_1), \ldots, \mathcal{TRD}_{\mathcal{L}}(R_n) \\ \textbf{allocates} & \mathcal{TRD}_{\mathcal{L}}(A_1), \ldots, \mathcal{TRD}_{\mathcal{L}}(A_m) \\ \textbf{writes} & \mathcal{TRD}_{\mathcal{L}}(W_1), \ldots, \mathcal{TRD}_{\mathcal{L}}(W_l) \\ \textbf{ensures} & \mathcal{TRD}_{\mathcal{R}}(Post)\end{array}$$

De plus, si une clause n'est pas spécifiée dans le contrat original, alors elle est rajoutée dans la traduction en lui attribuant une valeur par défaut : $\backslash true$ pour les clauses **requires** et **ensures** et $\backslash$**nothing** pour les clauses **reads**, **writes** et **allocates**.

Les transformations appliquées aux clauses **requires** et **ensures** sont identiques pour les deux traductions. Les deux fonctions $\mathcal{ABS}$ et $\mathcal{CCR}$ se comportent exactement de la même façon.

En général, à l'appel d'une fonction, on doit s'assurer que tous les objets dans la mémoire vérifient leurs invariants de données. Mais la clé de notre approche est qu'il n'est, justement, pas nécessaire de tous les vérifier, mais uniquement ceux des objets qui sont accessibles par le corps de la fonction f. Nous proposons, dans cette implémentation, de simuler ces invariants par des pré et des post-conditions. En pratique, il s'agit de rajouter une précondition et une post-condition $Inv_s[this \leftarrow l]$ pour tout emplacement mémoire l tel que $\phi(l) \in \mathsf{Rdom}(\widehat{R}_{in})$ pour la précondition et $\phi(l) \in \mathsf{Rdom}(\widehat{R}_{out})$ pour la post-condition.

L'idée est que les invariants des objets accessibles par le corps de la fonction sont vérifiés à l'entrée de la fonction. Autrement dit, tout paramètre x_i de type **struct** S_i de la fonction doit vérifier le prédicat $Valid\_S_i$.

$$\mathcal{TRD}_{\mathcal{R}}(Pre) = Pre \wedge \bigwedge_{i=1}^{n} Valid_{S_i}(x_i)$$

Les invariants doivent être rétablis à la fin de la fonction. Nous pourrions alors appliquer la même transformation que pour la précondition.

$$\mathcal{TRD}_{\mathcal{R}}(Post) = Post \wedge \bigwedge_{i=1}^{n} Valid\_S_i(x_i)$$

Cependant, si un objet n'a pas été modifié au cours de l'exécution du corps de la fonction, il n'est pas nécessaire de vérifier que son invariant est vrai. En effet, comme nous avons interdit tout partage de références, le seul moyen de rompre un invariant est de modifier les champs qu'il référence. Nous

proposons alors, de ne vérifier que les invariants des objets qui sont potentiellement modifiables par la fonction, c'est-à-dire de ceux spécifiés dans la clause **writes**.

Par ailleurs, les instances de types structure fraîchement allouées dans une fonction et qui sont mentionnées dans la clause **allocates** doivent vérifier leurs invariants. Enfin, dans le cas où la fonction retourne une valeur \result de type (**struct** R), il faudra également vérifier que \result vérifie son invariant. La post-condition de la fonction traduite est alors :

$$\mathcal{TRD}_{\mathcal{R}}(Post) = Post \wedge (\bigwedge_{i=1}^{n} Valid\_S_i(w_i)) \wedge (\bigwedge_{j=1}^{m} Valid\_S_j(a_j)) \wedge Valid_R(\backslash\mathsf{result})$$

où w_i et a_j sont, respectivement, les objets modifiés et alloués par la fonction.

Contrairement aux clauses **requires** et **ensures**, le résultat de la traduction des clauses **writes** et **reads** dépend du type de la traduction appliquée. Dans le cas d'une traduction abstraite, seuls les champs modèles peuvent faire partie du résultat, mais dans une traduction concrète, tous les champs, concrets et modèles, sont spécifiés. Si c est de type **struct** S, alors

$$\begin{array}{rcl} \mathcal{ABS}(*c) & = & \{c \rightarrow f \mid f \in \mathsf{Models}(S)\} \\ \mathcal{CCR}(*c) & = & \{c \rightarrow f \mid f \in \mathsf{Models}(S) \vee f \in \mathsf{Fields}(S))\} \end{array}$$

Exemple 6.1.1 *Considérons l'exemple de la fonction* `add` *du Calculateur de Morgan présentée dans l'exemple 1.1.3. Cette fonction modifie les deux champs concrets du type structure* `Calc` *:* `sum` *et* `count`*. Rappelons que le contrat de la fonction* `add` *est :*

Le résultat de la traduction abstraite de ce contrat :

function struct $Calc\ add($***struct*** $Calc\ c,\ real\ x)$
reads x ;
writes $*c$;
ensures $c \rightarrow values = \backslash$***old***$(c \rightarrow values) \cup \{x\}$;
ensures $Valid\_Calc(c)$;

est :

function struct $Calc\ add($***struct*** $Calc\ c,\ real\ x)$
requires $Valid\_Calc(c)$;
allocates $\backslash$***nothing***
reads x ;
writes $c \rightarrow values$;
ensures $c \rightarrow values = \backslash$***old***$(c \rightarrow values) \cup \{x\}$;
ensures $Valid\_Calc(c)$;

Alors que le résultat de la traduction concrète de ce même contrat est :

function struct $Calc\ add($***struct*** $Calc\ c,\ real\ x)$
requires $Valid\_Calc(c)$;
allocates $\backslash$***nothing***
reads $x,\ *c$;
writes $c \rightarrow values, c \rightarrow sum, c \rightarrow count$;
ensures $c \rightarrow values = \backslash$***old***$(c \rightarrow values) \cup \{x\}$;
ensures $Valid\_Calc(c)$;

Enfin, la fonction de traduction de la clause **allocates** est la fonction identité. Cette étape termine la traduction du contrat de la fonction.

En second lieu, nous traduisons le corps de la fonction. Une interface ne contient que les signatures des déclarations. Dans le cas des fonctions, ça voudrait dire les déclarations des fonctions, c'est-à-dire, pas de corps. Ainsi, la traduction abstraite du corps de la fonction consiste, tout simplement, à le supprimer.

$$\mathcal{ABS}(\textbf{function}\ \tau f(\tau_1\ x_1, \ldots \tau_n\ x_n)\ C\ \{\ body\ \}) = \textbf{function}\ \tau\ f\ (\tau_1\ x_1, \ldots \tau_n\ x_n)\ \mathcal{ABS}(C)$$

De son coté, la traduction concrète, non seulement maintient le corps de la fonction, mais en plus, elle le modifie.

$$\mathcal{CCR}(\textbf{function}\ \tau(\tau_1\ x_1, \ldots \tau_n\ x_n)\ C\{body\}) = \textbf{function}\ \tau\ f\ (\tau_1\ x_1, \ldots \tau_n\ x_n)\ \mathcal{CCR}(C)\{\mathcal{CCR}(body)\}$$

Cette modification consiste à rajouter une instruction à la fin du corps qui permet de mettre à jour les valeurs des champs modèle, et de vérifier les invariants nécessaires. Concrètement, il s'agit de réaliser l'équivalent d'une mise à jour non déterministe des champs modèle. Ceci sera réalisé par un appel de fonction fictive, qui sera détaillé dans la section suivante.

6.2 Réalisation

La réalisation consiste à développer un greffon *Frama-C*, que nous appelons `Abstr`, qui met en œuvre la traduction présentée ci-dessus. Ainsi, ce greffon traduit un programme C qui contient les notions de champs modèle et d'invariant de données en un programme qui ne fait référence à aucun des ces concepts. Le programme cible peut alors être traité par les autres greffons de vérification déductive : Jessie et WP. En effet, nous avons dit dans la section 1.2.3 que les programmes annotés en ACSL sont traduits en Why3 dont le générateur d'obligations de preuve fabrique des obligations de preuve envoyées vers les prouveurs automatiques ou interactifs. Le greffon peut être vu comme une étape préliminaire dans ce processus. Le schéma présenté dans la figure 6.1 montre la manière dont le greffon `Abstr` est intégré dans un tel processus.

6.2.1 Les théories

Une théorie dans notre langage d'étude est une suite de déclarations logiques. Le langage de spécification ACSL, offre la possibilité de définir ces mêmes constructions : définition de types logiques, de constantes, de fonctions logiques et de prédicats, mais également des axiomatisations de types, prédicats ou fonctions logiques.

6.2.2 Appartenance à un module

La notion de module n'existe pas dans le langage C. Or nous aimerions pouvoir dire qu'un type de données τ ou une fonction f appartiennent à un module M. Pour ce faire, nous utilisons l'ensemble des *pragmas* suivant :

– *#pragma TypeDecl(Module, Type)* dénote que le type *Type* appartient au module *Module*,

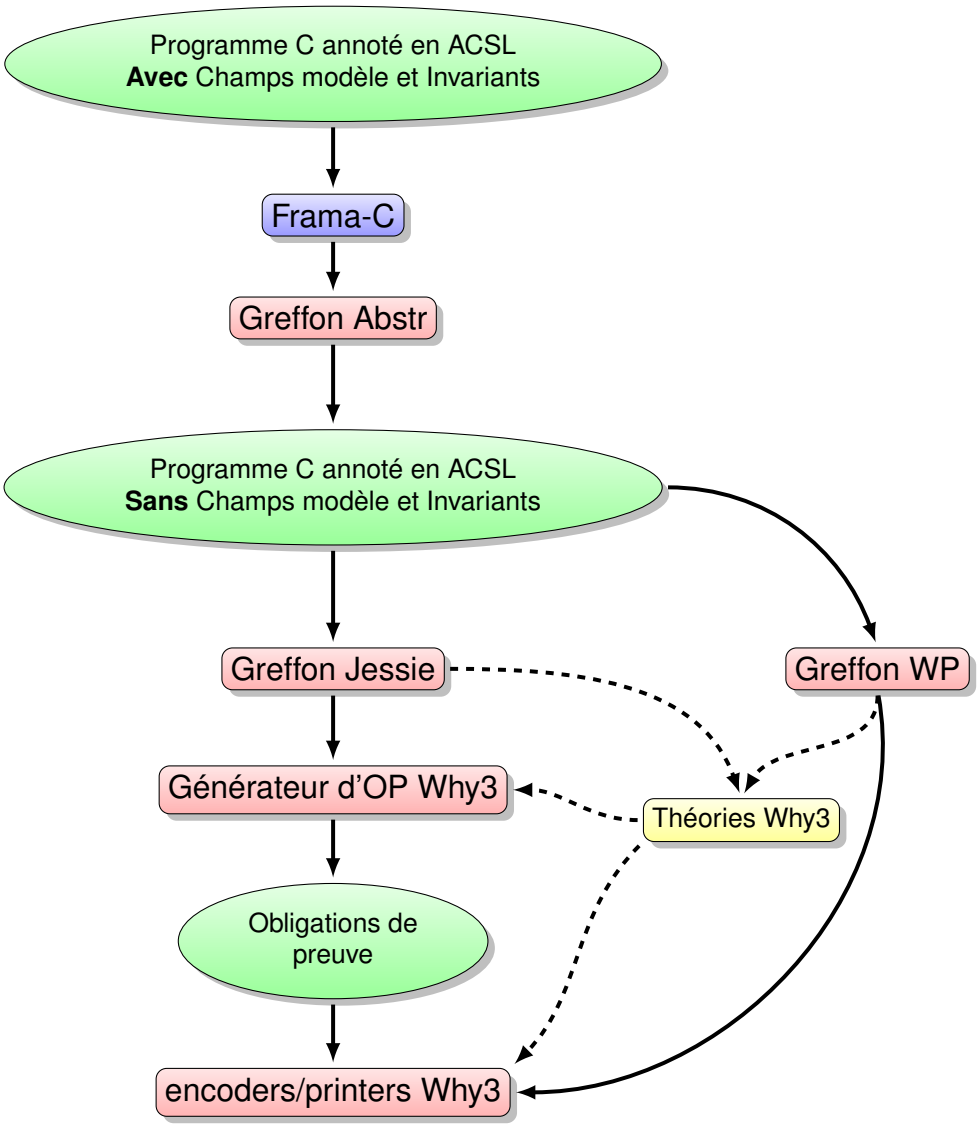


FIGURE 6.1 – Greffon `Abstr`.

- *#pragma Model(Module, Model)* indique le champ modèle *Model* appartient au module *Module*,
- *#pragma Coupling_invariant(Type, Inv)* spécifie que l'invariant *Inv* est associé au type *Type*,
- *#pragma Method(Module, Method)* mentionne que la méthode *Method* appartient au module *Module*.

Exemple 6.2.1 *Déclarer que le type* `Calc` *appartient au module* `Mod_Calc` *s'écrit sous la forme :*

```
#pragma TypeDecl(Mod_Calc, Calc)
typedef struct Calc *calc;
```

Le greffon `Abstr` permet à l'utilisateur de préciser quel module il souhaite vérifier, c'est ce que nous appelons *module courant*. Si aucun module n'est spécifié, alors le module courant est le programme client. En pratique, il s'agit de prouver les fonctions qui appartiennent au module. Concrètement, le

greffon applique une traduction concrète aux types structure et aux fonctions qui appartiennent au module courant et une traduction abstraite aux types structure et aux fonctions restantes.

6.2.3 Traduction des définitions de types structure

Les définitions des types structures dans un module écrit dans notre langage source peuvent contenir des champs modèle et des invariants de collage. Il nous faudrait alors pouvoir formaliser ces concepts dans le programme C en entrée du greffon `Abstr`.

La notion de champ modèle a été introduite, à notre demande, dans le langage ACSL dans la version *Oxygen* de *Frama-C*. La définition du champ modèle `m` de type `T` dans un type structure `S` s'écrit :

```
typedef struct S { T1 f1; ... Tn fn;} *s;
//@ model struct S { T m };
```

En ce qui concerne les invariants de collage, nous utilisons les *invariants de type* d'ACSL qui sont des propriétés que toute instance du type, auquel ils sont associés, vérifie. Concrètement, c'est un prédicat qui prend, en argument, un unique paramètre qui est une instance du type sur lequel l'invariant est défini.

```
//@ type invariant Inv_S(S this) = p;
```

où `p` est la formule qui définit la relation entre les champs modèle et les champs concrets du type structure `S`.

La traduction d'une définition d'un type structure S munie d'un invariant de type `Inv_S` consiste à générer un nouveau prédicat $Valid\_S$ tel que : Si `S` appartient au module courant, alors

```
//@ type invariant Inv_S(S this) = p;
//@ predicate Valid_S(S this) = valid(this) && Inv_S(this);
```

sinon (le type n'appartient pas au module courant)

```
//@ predicate Valid_S(struct S *s) = valid(s);
```

Notez que dans le deuxième cas, qui correspond à une traduction abstraite, l'invariant de type est supprimé. De même que les champs concrets. La définition de type présentée ci-dessus est alors transformée en une déclaration :

```
//@ typedef struct S *s;
```

6.2.4 Traduction des définitions de fonctions

Dans ACSL, un contrat de fonction est composé des clauses suivantes :

- **requires** permet de spécifier la précondition,
- **ensures** permet de spécifier la post-condition,
- **allocates** permet de spécifier les emplacements mémoire alloués par le fonction annotée,
- **assigns** permet de spécifier les emplacements mémoire dont la valeur est potentiellement modifiable par la fonction. C'est l'équivalent de la clause **writes** dans notre langage d'étude.

La traduction des fonctions consiste, en premier lieu, à étendre leurs contrats afin d'exprimer la propriété qui dit que les invariants de données des objets accessibles dans le corps de la fonction doivent être établis au début et à la fin de la fonction. Vu qu'il n'existe pas de clause **reads** dans ACSL, nous calculons l'ensemble de ces objets à partir de la liste des paramètres effectifs de la fonction. Pour tout paramètre `xi` de type **`struct`** `Ti`, rajouter dans le contrat de la fonction :

```
requires Valid_Ti(xi);
ensures Valid_Ti(xi);
```

Une optimisation consiste à ne pas rétablir `Valid_Ti(xi)` en post-condition si `xi` n'est pas mentionné dans la clause **assigns**.

Par ailleurs, de nouveaux emplacements mémoire peuvent être alloués par la fonction, ceux là sont spécifiés dans la clause **allocates**. Ces instances de type structure doivent à leur tour vérifier leur invariant à la fin de la fonction. Pour tout emplacement mémoire `ai` de type `Ti` spécifié dans la clause **allocates**, rajouter dans le contrat :

```
ensures Valid_Ti(ai);
```

Enfin, si la fonction retourne un pointeur sur un type structure, alors il faut, là encore, s'assurer que l'instance pointée vérifie son invariant.

```
ensures Valid_Ti(\result);
```

Nous spécifions, dans le programme source, qu'une instance `t` de type structure est modifiée, c'est-à-dire que les champs de `t` sont modifiés :

```
assigns *t;
```

Lors de la traduction une telle clause est transformée en explicitant l'ensemble des champs du type de `t`. Dans le cas d'une traduction abstraite, seuls les champs modèle sont mentionnés et dans le cas d'une traduction concrète tous les champs (modèle et concrets) sont spécifiés. Concrètement, si `f_i` pour $0 \leq i \leq n$ sont les champs concrets du type de `t` et `g_j` pour $0 \leq j \leq m$ les champs modèle de ce type, alors la clause **assigns** ci-dessus est transformée en :

```
assigns t→g_0, ... t→g_m, t→f_0, ... t→f_n;
```

si la fonction traduite appartient au module courant. et

```
assigns t→g_0, ... t→g_m;
```

dans le cas contraire.

Mise à jour des champs modèles

En deuxième lieu, la traduction de la définition d'une fonction `f` transforme le corps de cette dernière. Il s'agit plus précisément de mettre à jour les champs modèle des objets dont l'invariant doit être rétabli à la fin de la fonction. En effet, les valeurs de ces champs doivent être modifiées de telle façon que la relation entre les champs modèle et les champs concrets définie dans l'invariant de collage soit vérifiée. Pour ce faire, nous déclarons la fonction `close_f` suivante :

```
/*@ requires  \exists v_0, ..., v_k, ...  v'_0, ..., v'_k',
              Inv_T0(x0)[(x0->m_i) <- v_i] && ... &&
              Inv_T(xn)[(xn->m'_j) <- v'_j] &&
              (\old(Post_f))[v_i <- m_i];
  @ allocates \nothing;
  @ assigns   x->m_0, ... x->m_0, ...x_n->m'_0, ... x_n->m'_k';
  @ ensures   Inv_T0(x0) && ... &&  Inv_T(xn) && Post_f;
  @*/
function close_f(struct T_0 x_0, ..., struct T_n x_n, type result);
```

où $x_0 \dots x_n$ sont les variables de type structure qui doivent vérifier leurs invariants, m_i sont les champs modèle qui sont mis à jour, v_i des variables fraîches qui dénotent les nouvelles valeurs des m_i et $Post_f$ la post-condition de la fonction f.

La fonction `close_f` a un paramètre supplémentaire `result`. Ce dernier a le type de retour de la fonction. Il est nécessaire, car la post-condition de la fonction peut faire référence au résultat de celle-ci à l'aide de la variable logique $\backslash result$. Cependant, cette variable ne peut apparaître que dans une post-condition, il faut donc la substituer par `result`.

Le contrat de cette fonction spécifie que pour chaque champ modèle `m_i` qui appartient au type structure `T_i`, il existe une valeur `v_i` telle que l'invariant `Inv_Ti` et la post condition de la fonction sont vérifiés. La valeur `v_i` est alors affectée au champ `m_i`.

Les invariants et la post-condition doivent être établis à la fin de la fonction. Par conséquent, nous ajoutons avant l'expression *return*, un appel vers la fonction `close_f` pour mettre à jour les champs modèle.

Exemple 6.2.2 *La traduction du corps de la fonction* `add` *du calculateur de Morgan est :*

```
void add(calc c, double x) {
  count++;
  sum+=x;
  close_add(c);
}

/*@ requires \exists bag v; sumbag(v)==c->sum && card(v)==c->count &&
  @                        v ==union(c->values, singleton(x));
  @ assigns c->values;
  @ ensures sumbag(c->values)==c->sum && card(c->values)==c->count &&
  @         c->values ==union(\old(c->values), singleton(x));
  @*/
void close_add(calc c);
```

6.3 Les tableaux creux

6.3.1 Challenge initial

Le code initial de ce challenge est présenté dans la figure 6.2. Il décrit une classe `SparseArray` qui fournit des fonctions pour : la création d'un tableau creux (`create`), l'écriture dans un tableau (`set`) et la lecture du contenu d'un élément d'un tableau (`get`). Chacune de ces opérations est effectuée en temps constant. De plus, comme indiqué dans l'article décrivant ces benchmarks[69], les tableaux sont alloués à l'aide de l'instruction `new` et ne sont pas initialisés. La fonction `get` retourne alors une valeur par défaut, dans ce cas 0, en cas de lecture d'une case d'un tableau avant son initialisation.

```
class SparseArray {
  int val[MAXLEN];
  uint idx[MAXLEN], back[MAXLEN];
  uint sz;
  uint max_cap;

 static SparseArray create(uint cap) {
  SparseArray t = new SparseArray();
  val = new int[sz];
  back = new uint[sz];
  idx = new uint[sz];
  sz = 0;
  max_cap = cap;
  return t;
 }

 int get(uint i) {
  if (idx[i] < sz && back[idx[i]] == i) return val[i];
  else return 0;
 }

 void set(uint i, int v) {
  val[i] = v;
  if (!(idx[i] < sz && back[idx[i]] == i)) {
    //@ assert(sz < MAXLEN);
    idx[i] = sz;
    back[sz] = i;
    sz = sz + 1;
  }
 }
}
```

FIGURE 6.2 – Programme initial `SparseArray`.

L'implémentation proposée pour cette structure utilise trois tableaux `val`, `back`, et `idx` et deux entiers `sz` et `max_cap`. Le premier tableau, `val`, contient les valeurs effectives du tableau creux. L'entier `sz` représente le nombre de cases dans `val` auxquelles une valeur a été affectée. Le second tableau, `back`, contient dans ses `(sz - 1)` premières cellules les indices, dans `val`, des cases affectées. Le troisième et dernier tableau, `idx`, permet de savoir où ces indices apparaissent dans le tableau `back`. Enfin, `max_cap` est la capacité du tableau, c'est-à-dire le nombre maximal d'éléments que peut contenir le tableau.

La figure 6.3 illustre l'état d'un objet `SparseArray` après l'insertion, dans l'ordre, des éléments `a`, `b`,

c, et d, respectivement aux indices 3, 7, 5 et 1. Les cellules qui contiennent des "-" n'ont pas encore été initialisées.

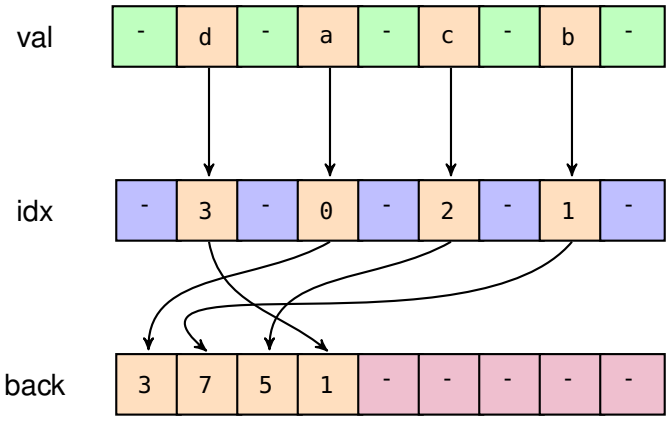


FIGURE 6.3 – Exemple d'une instance de `SparseArray`.

La première méthode définie dans la classe `SparseArray` est `create`. Elle permet de créer une nouvelle instance de `SparseArray`, d'allouer les trois tableaux (sans les initialiser) et d'initialiser les entiers `max_cap` et `size` avec le paramètre `cap` et la valeur 0, respectivement.

La deuxième méthode fournie, `get`, teste si le contenu du $i^{\text{ème}}$ élément du tableau `idx` est inférieur à `size`, ce qui signifie qu'une valeur a bien été stockée dans cette cellule. Le cas échéant, elle vérifie si à l'indice `idx[i]`, le tableau `back` contient bien la valeur `i`. Dans ce cas, la fonction `get` retourne `val[i]`, sinon elle retourne 0.

La troisième méthode, `set`, met le contenu de `val` à l'indice `i` à `v`. Ensuite, si c'est la première écriture à cet indice, les tableaux `idx` et `back` sont mis à jour, puis le nombre d'éléments `sz` insérés dans `val` est incrémenté de 1.

Par ailleurs, une méthode `sparseArrayTestHarness`, dont le code est détaillé dans la figure 6.4 est fournie. Cette méthode teste l'implantation sur des données concrètes. Elle crée, pour cela, deux instances de type `SparseArray`, `a` et `b`, insère dans chacune d'elle un élément et s'assure que seules les cellules modifiées par les appels de la fonction `set` sont mises à jour en vérifiant que les autres cellules contiennent la valeur par défaut, c'est-à-dire 0.

Elle contient des assertions permettant de vérifier statiquement et de façon modulaire, que le programme est correct, c'est-à-dire que les annotations des fonctions (pré et post-conditions), permettent d'établir ces assertions.

Nous allons maintenant montrer que l'approche que nous proposons permet de prouver cette implantation des tableaux creux, ainsi que la fonction `sparseArrayTestHarness`, de façon modulaire. Nous commençons alors par écrire le programme source décrivant la structure `sparseArray`, puis en fonction de ce que nous voulons prouver : implantation ou programme client, nous transformons notre programme en appliquant la traduction adéquate.

6.3.2 Formalisation des tableaux creux dans notre approche

Nous décrivons, dans un premier temps, une implantation de la structure de `SparseArray`, présentée dans la section précédente, dans notre langage. Puis, dans un deuxième temps, nous définirons un

```
void sparseArrayTestHarness() {
  SparseArray a = create(10);
  SparseArray  b = create(20);
  assert(a.get(5) == 0);
  assert (b.get(7) == 0);
  a.set(5, 1);
  b.set(7, 2);
  assert(a.get(5) == 1);
  assert(b.get(7) == 2);
  assert(a.get(0) == 0);
  assert(b.get(0) == 0);
}
```

FIGURE 6.4 – Programme client `sparseArrayTestHarness`.

programme client qui correspond à la fonction `sparseArrayTestHarness`.

Implantation de la structure `SparseArray`

Nous définissons, en premier lieu, un module `SparseArray` dans lequel le type structure `SparseArray` est défini, comme dans la présentation du challenge, par trois champs de type tableau : `val`, `idx` et `back` et deux entiers `sz` et `max_cap`. La définition du type structure `SparseArray` est détaillée dans la figure 6.5. Nous étendons la définition du type structure `SparseArray` avec trois champs modèle. Le troisième champ, `mapping`, est une représentation abstraite du tableau `val`.

Le type `map` est un type abstrait dont le comportement est axiomatisé dans la figure 6.6. Les deux autres champs modèle `size` et `capacity` représentent la taille du tableau `val` et sa capacité maximale. Ces deux champs sont identiques aux champs concrets `sz` et `max_cap` (dans l'invariant de collage). Ils sont néanmoins nécessaires pour décrire les contrats des fonctions du module, car contrairement aux champs concrets, ces champs sont utilisés dans les spécifications des fonctions.

Nous munissons, par ailleurs, le type structure `SparseArray` d'un invariant de collage qui permet d'établir la relation entre les éléments des trois tableaux, la relation entre les champs modèle `size` et `capacity` et les champs concrets `sz` et `max_cap`, c'est-à-dire que la valeur de `size`, (respectivement `capacity`) est égale à celle de `sz` (respectivement `max_cap`) et la relation entre les éléments du tableau `val` et le champ modèle `mapping`, à savoir que le $i^{\text{ème}}$ élément dans `mapping` correspond à l'élément stocké à l'indice i dans le tableau `val`. Si cette case n'est pas initialisée dans le tableau `val`, alors elle contient la valeur 0 qui correspond à la valeur par défaut pour les éléments d'une variable de type `map`.

Nous spécifions ensuite les fonctions `create`, `get` et `set` comme suit :

La fonction `create` alloue une nouvelle instance de type `SparseArray` dont la capacité est initialisée au paramètre `max` et qui ne contient aucun élément. Nous formulons ces propriétés dans la post-condition en spécifiant que les champs modèle `size` et `capacity` valent `0` et `max`. La post-condition de la fonction stipule également que tous les éléments du champ modèle `mapping` du résultat contiennent la valeur 0 (figure 6.7).

Le contrat de la fonction `get`, décrit dans la figure 6.8, indique que pour que cette fonction puisse être invoquée, la valeur de son paramètre `i` doit être comprise entre 0 et `a->capacity`. En d'autres termes, nous n'accédons pas à un élément en dehors des bornes du tableau. La clause **assigns** indique qu'aucun emplacement mémoire n'est modifié, c'est-à-dire que la fonction n'a pas d'effets de bord. Enfin, la

```
#pragma TypeDecl(Mod_Sparse, SparseArray)
typedef struct SparseArray {
   int *val;
   int *idx;
   int *back;
   int max_cap;
   int sz;
}*sparse;

#pragma Model(Mod_Sparse, size)
//@ model struct SparseArray { integer size };

#pragma Model(Mod_Sparse, capacity)
//@ model struct SparseArray { integer capacity };

#pragma Model(Mod_Sparse, mapping)
//@ model struct SparseArray { map mapping };

#pragma Coupling_invariant(SparseArray, Inv_Sparse)
/*@ type invariant Inv_Sparse(sparse this) =
  @  0 <= this->sz && this->sz <= this->max_cap &&
  @   this->capacity == this->max_cap && this->size == this->sz &&
  @   \valid(this->val+ (0..this->capacity- 1)) &&
  @   \valid(this->idx+(0..this->capacity- 1)) &&
  @   \valid(this->back+(0.. this->capacity- 1))
  @    && (\forall integer i;
  @          0 <= i < this->size ==>
  @          0 <= this->back[i] < this->capacity &&
  @          this->idx[this->back[i]] == i )
  @   && (\forall integer i; 0 <= i < this->capacity ==>
  @        ((0 <= this->idx[i] < this->size && this->back[this->idx[i]] == i) ==>
  @               get_map(this->mapping, i)  == this->val[i]) &&
  @        (!(0 <= this->idx[i] < this->size && this->back[this->idx[i]] == i) ==>
  @               get_map(this->mapping, i)  == 0))
  @;
  @*/
```

FIGURE 6.5 – Définition du type structure SparseArray.

fonction retourne le $i^{\text{ème}}$ élément de `mapping`.

La troisième et dernière fonction du module `SparseArray`, dont le code est présenté dans la figure 6.9, est la fonction `set`. Cette fonction affecte la valeur `v` au $i^{\text{ème}}$ élément du tableau `val`, met à jour les tableaux `idx` et `back` si la cellule en question n'a pas encore été initialisée et incrémente le champ `sz`. Par conséquent, avant l'appel de la fonction, en plus de vérifier que `i` est bien dans les bornes du tableau, il faut s'assurer que nous n'avons pas atteint la capacité maximale de ce dernier, c'est-à-dire que le nombre d'éléments insérés est strictement inférieur à la capacité du tableau. De plus, nous spécifions le fait que la fonction `set` modifie les champs de `a` dans la clause **writes**, en déclarant que tous les emplacements mémoire atteignables à partir de `a` sont potentiellement modifiables. Enfin, la post-condition indique que le $i^{\text{ème}}$ élément de `val` vaut `v` et que tous les autres éléments sont inchangés.

```
/*@ axiomatic Map {
  @   type map;
  @   logic integer get_map(map m, integer key);
  @   logic map set_map(map m, integer key, integer value);
  @   logic map constr(integer value);
  @
  @   axiom Const :
  @     \forall integer value, integer key; get_map(constr(value), key) == value;
  @
  @   axiom Select_eq :
  @     \forall map m, integer key1, key2, integer value;
  @    key1 == key2 ==> get_map(set_map(m, key1, value), key2) == value;
  @
  @   axiom Select_neq :
  @     \forall map m, integer key1, key2, integer value;
  @     key1 != key2 ==> get_map(set_map(m, key1, value), key2) == get_map(m, key2);
  @ }
  @*/
```

FIGURE 6.6 – La théorie `Map`.

```
#pragma Method(Mod_Sparse, create_Sparse)
/*@ allocates \result;
  @ ensures \result->capacity == max && \result->size == 0;
  @ ensures \result->mapping == constr(0);
  @*/
sparse create_Sparse(int max)
{
  sparse a = (sparse)malloc(sizeof(struct SparseArray));
  a->val = (int *)calloc (max, sizeof(int));;
  a->idx = (int *)calloc (max, sizeof(int));;
  a->back = (int *)calloc (max, sizeof(int));;
  a->sz = 0;
  a->max_cap = max;
  return a;
 }
```

FIGURE 6.7 – Implantation de la fonction `create_Sparse` du module `SparseArray`.

Programme client

La programme client correspondant à la fonction `sparseArrayTestHarness` (figure 6.4) appartient au module `Harness`, comme indiqué dans la figure 6.10.

6.3.3 Preuve de l'implantation de la structure SparseArray

Afin de prouver une telle implantation de la structure de tableaux creux, nous appliquons une traduction concrète sur le module `SparseArray`. Cette traduction se fait en trois étapes : la première étape consiste à transformer la déclaration du type structure, la deuxième étape étend les contrats des fonctions définies dans le module et la troisième étape modifie le corps de chaque fonction.

La figure 6.11 décrit le résultat de la traduction du type structure `SparseArray` dans lequel un prédi-

```
#pragma Method(Mod_Sparse, get)
/*@ requires 0 <= i && i < a->capacity;
  @ assigns \nothing;
  @ ensures \result == get_map(a->mapping, i);
  @*/
int get(sparse a, int i){
  if ((0 <= a->idx[i]) && (a->idx[i] < a->sz) && (a->back[a->idx[i]] == i))
  return a->val[i];
  else return 0;
}
```

FIGURE 6.8 – Implantation de la fonction `get` du module `SparseArray`.

```
#pragma Method(Mod_Sparse, set)
/*@ requires 0 <= i < a->capacity;
  @ assigns *a;
  @ ensures a->mapping == set_map(\old(a->mapping), i, v);
  @ ensures a->capacity == \old(a->capacity);
  @*/
void set(sparse a, int i, int v){
  this->val[i] = v;
  int tmp = this->idx[i];
  if (!((0 <= tmp) && (tmp < this->sz) && (this->back[tmp] == i))) {
    //@ assert this->sz < this->max_cap;
    this->idx[i] = this->sz;
    this->back[this->sz] = i;
    this->sz ++;
  }
}
```

FIGURE 6.9 – Implantation de la fonction `set` du module `SparseArray`.

cat `Valid_Sparse` est défini. Ce prédicat établit qu'une instance de type `SparseArray` est valide si elle est allouée en mémoire et qu'elle vérifie l'invariant de type `Inv_Sparse`.

La deuxième étape de la traduction porte sur les contrats des fonctions du module traduit. La modification appliquée à ces derniers consiste à rajouter les clauses qui n'y sont pas spécifiées. Par exemple, les clauses **requires** et **assigns** ne sont pas mentionnées dans le contrat de la fonction `create` présenté dans la figure 6.7. Par contre, ces clauses sont explicitement mentionnées dans le résultat de la traduction.

En outre, nous avons fait le choix de vérifier les invariants des objets au début et à la fin des fonctions. Concrètement, le prédicat `Valid_Sparse` doit être établi en pré et en post-condition de chaque fonction, pour les objets accessibles dans le corps de la fonction annotée. C'est le cas, pour les fonctions `set` et `get`, dont les pré et post-conditions sont étendues avec `Valid_Sparse(a)`. En regardant de plus près, nous ne vérifions pas que le prédicat `Valid_Sparse` est vrai à la fin de la fonction `get`. En effet, cette fonction ne modifie aucun emplacement mémoire, alors si `Valid_Sparse(a)` est vrai au début de la fonction, alors il l'est obligatoirement à la fin. Ceci est bien sûr garanti par l'absence de partage dans notre approche. On montre, dans la figure 6.12, le résultat de cette étape de traduction sur la fonction `set`.

La fonction `create` ne lit, ni modifie, aucun emplacement mémoire, il n'y a alors aucun invariant à vérifier à l'appel de la fonction. Par contre, elle alloue bien un nouvel emplacement mémoire. Or, les objets alloués doivent vérifier leurs invariants. Ainsi, nous spécifions dans la post-condition de cette

```
#pragma Method(Harness, sparseArrayTestHarness)
void sparseArrayTestHarness() {
  sparse a = create_Sparse(10);
  sparse b = create_Sparse(20);
  int get_a_5 = get(a, 5);
  //@ assert get_a_5 == 0;
  int get_b_7 = get(b, 7);
  //@ assert get_b_7 == 0;
  set(a, 5, 1);
  set(b, 7, 2);
  int get_a_5 = get(a, 5);
  //@ assert { get_a_5 == 1 };
  int get_b_7 = get(b, 7);
  //@ assert { get_b_7 == 2 };
  int get_a_0 = get(a, 0);
  //@ assert { get_a_0 == 0 };
  int get_b_0 = get(b, 0);
  //@ assert { get_b_0 == 0 }
 }
```

FIGURE 6.10 – Programme client du module `SparseArray`.

```
#pragma TypeDecl(Mod_Sparse, SparseArray)
typedef struct SparseArray {
   int *val;
   int *idx;
   int *back;
   int max_cap;
   int sz;
}*sparse;

//@ model struct SparseArray { integer size };
//@ model struct SparseArray { integer capacity };
//@ model struct SparseArray { map mapping };

/*@ type invariant Inv_Sparse(sparse this) =
  @  0 <= this->sz && this->sz <= this->max_cap &&
  @  this->capacity == this->max_cap && this->size == this->sz &&
  @  \forall integer i; 0 <= i < this->sz ==>
  @     0 <= this->back[i] < this->max_cap &&
  @     this->idx[this->back[i]] == i &&
  @     get_map(this->mapping, i)  == this->val[i];
  @*/

/*@ predicate Valid_Sparse(sparse this) = valid(this) && Inv_Sparse(this);
  @*/
```

FIGURE 6.11 – Traduction concrète de la définition du type structure `SparseArray`.

fonction que `Valid_Sparse(\result)` est vrai.

La troisième étape de la traduction consiste à modifier les corps des fonctions. Le but de cette étape est de mettre à jour les champs modèle afin d'établir les invariants des objets modifiés dans le corps de la fonction, en associant à chaque fonction `f` une nouvelle déclaration `close_f`, puis en rajoutant, à la fin du corps de la fonction `f`, un appel vers la fonction `close_f` comme indiqué dans la figure 6.12.

```
/*@ requires Valid_Sparse(a);
  @ requires 0 <= i && i < a->capacity;
  @ requires a->size < a->capacity - 1;
  @ allocates \nothing;
  @ assigns a->mapping, a->size, a->capacity,
            a->val, a->idx, a->back, a->sz, a->max_cap;
  @ ensures get_map(a->mapping, i) == v;
  @ ensures \forall int j; j != i ==>
  @                 get_map(a->mapping, i) == get_map(\old(a->mapping), j);
  @ ensures Valid_Sparse(a);
  @*/
void set(sparse a, int i, int v)
{ ...
  close_set(this);
}
```

FIGURE 6.12 – Traduction concrète de la fonction `set` du module `SparseArray`.

```
function close_set(sparse a)
  requires \exists capacity_1, size_1, mapping_1;
     0 <= a->sz <= a->max_cap && capacity_1 == a->max_cap && size_1 == a->sz &&
     \forall integer i; 0 <= i < a->sz ==> 0 <= a->back[i] < a->max_cap &&
       a->idx[a->back[i]] == i && get(mapping_1, i) == a->val[i] &&
      get(mapping_1, i) == v &&
      \forall integer j; j != i ==> get(mapping_1, i) == get(a->mapping, j);
  allocates \nothing;
  assigns  a->capacity, a->size, a->mapping;
  ensures 0 <= a->sz <= a->max_cap &&
      a->capacity == a->max_cap && a->size == a->sz &&
      \forall integer i; 0 <= i < a->sz ==> 0 <= a->back[i] < a->max_cap &&
        a->idx[a->back[i]] == i && get (mapping, i)  == a->val[i] &&
      get(a->mapping, i) == v &&
      \forall integer j; j != i ==> get(a->mapping, i) == get(\old(a->mapping), j);
```

FIGURE 6.13 – Génération de la fonction `close_set`.

L'ensemble de ces fonctions est présenté dans la figure 6.13.

6.4 Les tas binaires

Un tas binaire est une arbre binaire dont les nœuds préservent la propriété de tas, c'est-à-dire que chaque nœud est plus petit que chacun de ses enfants.

6.4.1 Présentation du challenge

Le défi des tas binaires tel qu'il est présenté dans l'article [70] est composé de deux parties distinctes. La première partie du challenge, la classe abstraite, contient les méthodes suivantes :

- **create** : permet de créer une nouvelle instance de $Heap$ de capacité maximale sz,
- **insert** : permet d'insérer un nouvel élément dans le tas,
- **extractMin** : retourne le plus petit élément du tas, c'est-à-dire la racine, et reconstruit le tas avec

les éléments restants.

```
class Heap {
    static Heap create(int sz);
    void insert(int e);
    int extractMin();
}
```

FIGURE 6.14 – Interface d'un tas binaire.

La deuxième partie du challenge, qui représente le programme client, contient :

1. Une implantation simple pour le tri par tas **heapSort**. Cette fonction trie un tableau d'entiers en insérant dans un premier temps un à un ses éléments dans un tas et dans un second temps en les extrayant dans un ordre croissant,
2. Un cas d'utilisation **heapSortTestHarness** permettant de tester le tri d'un tableau d'entiers particulier. On ajoute, explicitement, deux assertions pour spécifier les propriétés qui doivent être vérifiées après le tri.

```
void heapSort(int[] arr, int len) {
    int i;
    Heap h = create(len);
    for (i = 0; i < len; ++i) h.insert(arr[i]);
    for (i = 0; i < len; ++i) arr[i] = h.extractMin();
}

void heapSortTestHarness() {
    int[] arr = { 42, 13, 42 };
    heapSort(arr, 3);
    assert(arr[0] <= arr[1] && arr[1] <= arr[2]);
    assert(arr[0] == 13 && arr[1] == 42 && arr[2] == 42);
}
```

FIGURE 6.15 – Implantation d'un tri par tas.

Nous allons, dans ce qui suit, traiter ce challenge en appliquant notre approche.

6.4.2 Formalisation d'une structure de tas binaire dans notre approche

L'architecture de la solution que nous proposons, détaillée dans la figure 6.16 où les théories sont en rouge et les modules en bleu, représente le graphe de dépendance entre les différents composants qui vont nous permettre de spécifier puis prouver le module `Binary_Heap`, dans un premier temps, et les fonctions `heapSort` et `heapSortTestHarness`, dans un deuxième temps. La première partie de cette architecture formalise les notions de multi-ensemble, définit le multi-ensemble des éléments d'un tableau et enfin définit un certain nombre de propriétés arithmétiques. La seconde partie fournit une implantation des tas binaires. Enfin, la troisième partie de l'architecture fournit une interface des tas binaires conformément à l'énoncé du challenge décrit dans la section 6.4.1 et les programmes C correspondant aux fonctions `heapSort` et `heapSortTestHarness`.

Nous détaillons ci-dessous, chacune de ces parties.

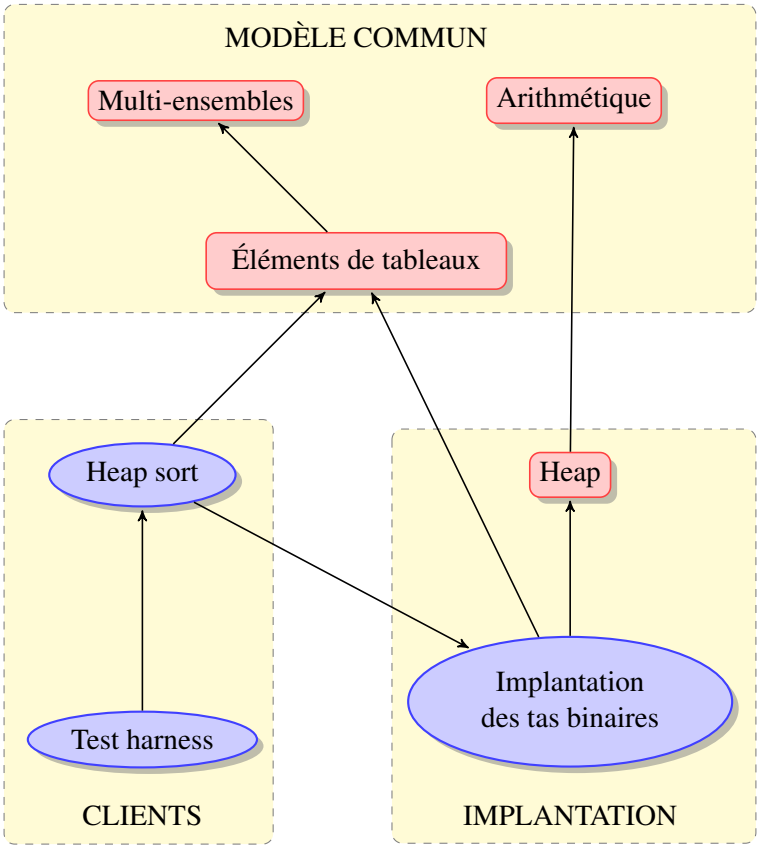


FIGURE 6.16 – Architecture de la solution.

Modèle commun

La théorie des multi-ensembles : Nous utilisons la théorie `Bag`, présentée dans la section 2.2 pour la définition du calculateur de *Morgan*, que nous étendons dans la figure 6.17 en déclarant une nouvelle fonction `diff` qui dénote la différence entre deux multi-ensembles. Cette fonction est axiomatisée en termes de nombre d'occurrences `nb_occ`. Nous posons ensuite un certain nombre de lemmes qui nous seront utiles par la suite. Ces lemmes permettent d'énoncer des propriétés sur la fonction `diff` telles que la différence entre un multi-ensemble `b` et le multi-ensemble vide (`empty_bag`), ou plus généralement, des propriétés reliant les différentes fonctions définies dans la théorie `Bag`.

Nous étendons à nouveau la théorie `Bag` avec la fonction logique, non interprétée, `min_bag` qui retourne le plus petit élément d'un multi-ensemble. Cette fonction est axiomatisée avec deux axiomes établissant que le plus petit élément dans un singleton est l'élément contenu dans ce singleton, et le plus petit élément dans une union de deux multi-ensembles `x` et `y` est le minimum entre `min_bag(x)` et `min_bag(y)`. Nous énonçons en plus deux lemmes : le premier spécifie que le minimum d'un ensemble résultant de l'ajout de l'élément `x` au multi-ensemble `b` est le minimum entre `x` et `min_bag` de `b` et le second lemme dit que si un entier `a` est inférieur à `min_bag` d'un bag `b`, alors il est également inférieur à `min_bag` du bag résultant de l'ajout de `a` à `b`.

Les résultats de l'application des prouveurs automatiques, Alt-ergo, CVC3, Vampire et Z3 sur ces lemmes sont résumés dans la figure 6.19. Chaque colonne, dans le tableau, correspond à un prouveur et chaque ligne correspond à une obligation de preuve. Ainsi, chaque cellule contient le temps mis par le prouveur pour prouver l'obligation de preuve. Les cellules vides correspondent aux cas où le prouveur

```
/*@ axiomatic Diff {
  @  logic bag diff (bag a, bag b);
  @
  @  axiom Diff_occ: \forall bag b1, b2, integer x;
  @    nb_occ(x, diff(b1, b2)) == max(0, nb_occ(x, b1) - nb_occ(x, b2));
  @ }
  @*/

//@ lemma Diff_empty_right: \forall bag b; diff(b, empty_bag) == b;

//@ lemma Diff_empty_left: \forall bag b; diff(empty_bag, b) == empty_bag;

//@ lemma Diff_add: \forall bag b, integer x; diff(add(x, b), singleton(x)) == b;

/*@ lemma Diff_comm:
  @   \forall bag b, b1, b2; diff(diff(b, b1), b2) == diff(diff(b, b2), b1);
  @*/

/*@ lemma Add_diff:
  @   \forall bag b, integer x; nb_occ(x, b) > 0 ==> add(x, diff(b, singleton(x))) == b;
  @*/
```

FIGURE 6.17 – Extension de la théorie `Bag`

```
/*@ axiomatic Min_bag {
  @  logic integer min_bag(bag a);
  @
  @  axiom Min_bag_singleton : \forall integer x; min_bag (singleton (x)) == x;
  @
  @  axiom Min_bag_union :
  @    \forall bag a, b; min_bag (Union (a, b)) == min(min_bag (a), min_bag (b));
  @ }
  @*/

/*@ lemma Min_bag_union1 :
  @   \forall bag x, y, integer a; x == add(a, y) ==> min_bag(x) == min(a, min_bag(y));
  @*/

/*@ lemma Min_bag_union2 :
  @   \forall bag x, integer a; a <= min_bag(x) ==> a <= min_bag(add(a, x));
  @*/
```

FIGURE 6.18 – Nouvelle extension de la théorie `Bag`

concerné n'a pas été testé sur l'obligation de preuve associée.

Notons qu'aucun des prouveurs automatiques n'est capable de prouver les lemmes `Union_comm`, `Diff_add`, etc. Nous utilisons dans ce cas là, le prouveur interactif Coq. Ces preuves sont simples (pas plus de 10 lignes de Coq) et peuvent être rejouées rapidement comme indiqué dans la figure 6.19. Elles appliquent l'axiome `bag_extensionality`, ce qui est probablement la raison pour laquelle les prouveurs automatiques ne peuvent pas les prouver.

Proof obligations	Alt-Ergo 0.93	CVC3 2.2	Coq 8.3pl2	Eprover 1.4 Namring	Simplify 1.5.4	Vampire 0.6	Yices 1.0.25	Z3 2.19
is_empty		0.00		0.00		0.04		0.24
occ_singleton_eq	0.01	0.00		0.00	0.00	0.00	0.00	0.01
occ_singleton_neq	0.01	0.00		0.00	0.00	0.01	0.00	0.00
Union_comm			0.48					
Union_identity			0.50			0.18		
Union_assoc			0.50					
bag_simpl			0.50					
bag_simpl_left	0.01	0.00	0.51	0.09	0.02		0.00	0.01
occ_add_eq	0.01			0.24	0.10	0.05		1.15
occ_add_neq	0.01			0.37	0.07	0.85		0.17
Diff_empty_right			0.52			1.06		
Diff_empty_left			0.52			0.81		
Diff_add			0.54					
Diff_comm			1.00					
Add_diff			0.56					
Min_bag_union1	0.02			0.06	0.01	1.12		0.02
Min_bag_union2	0.02	0.01		0.66		0.08		0.29

FIGURE 6.19 – Résultats des preuves des lemmes de la théorie `Bag`.

Propriétés sur les éléments d'un tableau : Nous avons fait le choix d'implanter les tas binaires en utilisant les tableaux. Nous avons alors besoin de décrire des propriétés sur ces structures de données. Nous commençons alors par déclarer et axiomatiser la fonction logique `elements` tel que présenté dans la figure 6.20. Cette fonction associe à un tableau `a`, et deux entiers `i` et `j`, le multi-ensemble des éléments stockés entre les indices `i` et `(j-1)` dans `a`. L'axiomatisation de cette fonction utilise deux axiomes. Le premier `Elements_empty` établit que `elements(a, i, j)`, pour un tableau `a` lorsque `i` est supérieur ou égal à `j`, est un multi-ensemble vide vu qu'il n'y a aucun élément entre les deux indices. Le deuxième axiome `Elements_add` dit que si `i < j`, alors le multi-ensemble des éléments de `a` stockés entre les indices `i` et `j` est construit en rajoutant l'élément `a[j-1]` au multi-ensemble contenant les éléments dont l'indice est compris entre `i` et `(j-1)`.

Nous posons ensuite un ensemble de lemmes et, comme pour la théorie précédente, nous résumons dans la figure 6.22 quels sont les prouveurs qui permettent de prouver ces lemmes.

Le lemme `Elements_singleton` indique que l'ensemble des éléments stockés entre deux indices successifs `i` et `(i+1)` d'un tableau `a` correspond au singleton contenant `a[i]`. Ce lemme est une conséquence directe de l'axiome `Elements_add` et est automatiquement prouvé par Z3.

Le lemme `Elements_union` dit que l'union des deux multi-ensembles contenant les éléments dont les indices sont compris, respectivement, entre `i` et `j` et entre `j` et `k` est égale au multi-ensemble contenant les éléments du tableau stockés entre les indices `i` et `k`, pour tout entier `i`, `j` et `k` tels que $\texttt{i} \leq \texttt{j} \leq \texttt{k}$. Pour prouver ce lemme, nous avons besoin de faire une induction sur `k` et donc d'utiliser Coq.

Le lemme `Elements_add1` établit que pour tout entiers `i` et `j` tels que `i < j`, le multi-ensemble qui contient les éléments du tableau `a` dont l'indice est entre `i` et `j` résulte de l'ajout de l'élément `a[i]` au

```
/*@ axiomatic Bag_arrays {
  @  logic bag elements{L}(int *a, integer i, integer j);
  @
  @  axiom Elements_empty{L}: \forall int *a, integer i, j;
  @    i >= j ==> elements{L}(a, i, j) == empty_bag;
  @
  @  axiom Elements_add: \forall int *a, integer i, j;
  @     i < j ==> elements(a, i, j) == add(a[j-1], elements(a, i, (j-1)));
  @ }
  @*/

/*@ lemma Elements_singleton{L}: \forall int *a, integer i, j;
  @    j == (i + 1) ==> elements{L}(a, i, j) == singleton(a[i]);
  @*/

/*@ lemma Elements_union{L}: \forall int *a, integer i, j, k;
  @    i <= j <= k ==> elements{L}(a, i, k) == Union(elements(a, i, j), elements(a, j, k));
  @*/

/*@ lemma Elements_add1 : \forall int *a, integer i, j;
  @    i < j ==> elements(a, i, j) == add(a[i], elements(a,(i+1), j));
  @*/

/*@ lemma Elements_remove_last: \forall int *a, integer i, j;
  @    i < j-1 ==> elements(a, i, (j-1)) == diff(elements(a, i, j), singleton(a[j-1]));
  @*/

/*@ lemma Occ_elements: \forall int *a, integer i, j, n;
  @    i <= j < n ==> nb_occ(a[j], elements(a, i, n)) > 0;
  @*/

/*@ lemma Elements_set_outside {L1, L2}:
  @   \forall int *a, *new_a, integer i, j;
  @     0 <= i ==> i <= j ==> \forall integer k;
  @      (k < i || k >= j) ==> \forall int e;
  @      A_set{L1, L2}(a, new_a, k, e) ==>
  @      elements{L2}(new_a, i, j) == elements{L1}(a, i, j);
  @*/

/*@ lemma Elements_set_inside {L1, L2}:
  @   \forall int *a, *new_a, integer i, j, n, int e, bag b;
  @     i <= j < n ==> A_set{L1, L2}(a, new_a, j, e) ==>
  @      elements{L1}(a, i, n) == add(\at(a[j], L1), b) ==>
  @      elements{L2}(new_a, i, n) == add(e, b);
  @*/

/*@ lemma Elements_set_inside2 {L1, L2}:
  @   \forall int *a, *new_a, integer i, j, n, int e;
  @     i <= j < n ==> A_set{L1, L2}(a, new_a, j, e) ==>
  @      elements{L2}(new_a, i, n) ==
  @        add(e, diff(elements{L1}(a, i, n), singleton(\at(a[j], L1))));
  @*/
```

FIGURE 6.20 – Axiomatisation de la fonction `elements`.

multi-ensemble composé des éléments du tableau compris entre les indices `(i+1)` et `j`. Ce lemme est prouvé en Coq en réécrivant le terme à l'aide du lemme précédent.

Le lemme `Elements_remove_last` permet d'exprimer que si `i < (j-1)` pour `i` et `j` des entiers quelconques, alors le multi-ensemble des éléments du tableau `a` entre `i` et `j` est construit en ajoutant `a[j-1]` au multi-ensemble des éléments de `a` entre `i` et `(j-1)`. Les prouveurs automatiques Alt-ergo, CVC3 et Z3 permettent de prouver automatiquement ce lemme.

Le lemme `Occ_element` établit que chaque élément du tableau entre les indices `i` et `n` apparaît au moins une fois dans `elements(a, i, n)`, pour tout entier `i` et `n`. La preuve d'un tel lemme est faite en Coq.

Le lemme `Elements_set_outside` dit que la modification d'un tableau à l'indice `k` ne modifie pas le multi-ensemble des éléments de ce tableau entre les indices `i` et `j` si `k` est à l'extérieur de l'intervalle `[i, j[`. La preuve de ce lemme est également faite en Coq.

Pour finir, deux lemmes supplémentaires permettent d'exprimer quel sera le multi-ensemble des éléments du tableau `a` dont l'indice est compris entre `i` et `n` lorsque `a` est modifié à l'indice `j` tel que $\texttt{i} \leq \texttt{j} < \texttt{n}$. Ainsi, le lemme `Elements_set_inside2` établit que : pour `i`, `j`, et `n`, insérer une valeur `e` dans `a[j]` a pour conséquence que le multi-ensemble des éléments de `a` entre `i` et `n` contient une nouvelle occurrence de `e`, et `a[j]` est supprimé. Le second lemme est une autre formulation de la même propriété : si le multi-ensemble qui contient les éléments du tableau a est construit en rajoutant l'élément `a[j]` au multi-ensemble `b`, alors après l'insertion de la valeur `e` à l'indice `j` du tableau, le multi-ensemble des éléments de `a` entre les indices `i` et `n` est égal au résultat de l'ajout de `e` dans `b`. Le premier lemme est prouvé en Coq, mais comme le second est une reformulation du premier, les prouveurs automatiques Alt-ergo et Z3 le prouvent.

Nous notons dans les trois derniers lemmes l'utilisation d'un prédicat `A_set`. Ce dernier est défini dans la théorie `Set`(Voir figure 6.21) et spécifie que les deux tableaux `a1` et `a2` contiennent les mêmes éléments, sauf à l'indice `i`, où `a2` contient la valeur `v`.

```
/*@ predicate A_set{L1, L2}(int *a1, int *a2, integer i, int v) =
  @   \at (a2[i], L2) == v && \forall integer j; j >= 0 ==>
  @   j != i ==> \at (a1[j], L1) == \at (a2[j], L2);
  @*/

/*@ lemma A_set_eq {L1, L2}: \forall int *a1, *a2, integer i, j, int v;
  @   i == j ==> A_set{L1, L2}(a1, a2, i, v) ==> \at(a2[i], L2) == v;
  @*/

/*@ lemma A_set_neq {L1, L2}: \forall int *a1, *a2, integer i, j, int v;
  @   j >= 0 ==>
  @   i != j ==> A_set{L1, L2}(a1, a2, i, v) ==> \at (a1[j], L1) == \at (a2[j], L2);
  @*/
```

FIGURE 6.21 – La théorie `Set`.

Propriétés arithmétiques : la théorie `Arithmetic` : La théorie `Arithmetic` introduit des fonctions logiques qui sont du sucre syntaxique permettant de simplifier les formules par la suite.

Les fonctions de cette théorie définissent pour un nœud dont l'indice est i, les indices des fils droit et gauche et du nœud parent. Nous montrons ensuite dans la figure 6.23, un ensemble de lemmes qui

Proof obligations	Alt-Ergo 0.93	CVC3 2.2	Coq 8.3pl2	Eprover 1.4 Namring	Yices 1.0.25	Z3 2.19
Elements_singleton						0.02
Elements_union			0.58			
Elements_add1			0.51		0.48	
Elements_remove_last	0.02	0.02		4.80		0.02
Occ_elements			0.56			
Elements_set_outside			0.66			
Elements_set_inside			0.60			
Elements_set_inside2	0.02					0.12

FIGURE 6.22 – Résultats des preuves des lemmes sur les multi-ensembles d'éléments de tableau.

```
//@ logic integer right(integer i) = 2*i+2;
//@ logic integer left(integer i) = 2*i+1;
//@ logic integer parent(integer i) = (i-1) / 2;

//@ lemma Parent_inf: \forall integer i;  0 < i ==> parent(i) < i;

//@ lemma Left_sup: \forall integer i; 0 <= i ==> i < left(i);

//@ lemma Right_sup: \forall integer i; 0 <= i ==> i < right(i);

//@ lemma Parent_right: \forall integer i; 0 <= i ==> parent(right(i)) == i;

//@ lemma Parent_left: \forall integer i; 0 <= i ==>  parent(left(i)) == i;

//@ lemma Inf_parent: \forall integer i, j; 0 < j <= right(i) ==> parent(j) <= i;

//@ lemma Parent_pos: \forall integer j; 0 < j ==> 0 <= parent(j);

/*@ lemma Child_parent: \forall integer i;
  @   0 < i ==> left(parent(i)) == i ||  right(parent(i)) == i;
  @*/

/*@ predicate parentChild (integer i, integer j) =
  @  0 <= i < j ==> (j == left(i)) || (j == right(i));
  @*/
```

FIGURE 6.23 – La théorie `Arithmetic`.

Proof obligations	Alt-Ergo 0.93	CVC3 2.2	Eprover 1.4 Namring	Simplify 1.5.4	Yices 1.0.25	Z3 2.19
Parent_inf	0.01	0.00		0.00	0.00	0.01
Left_sup	0.01	0.00		0.00	0.00	0.01
Right_sup	0.01	0.00		0.00	0.00	0.01
Parent_right	0.02	0.00		0.00	0.00	0.01
Parent_left	0.01	0.00		0.06	0.00	0.01
Inf_parent	0.02	0.01		0.00	0.00	0.01
Child_parent	0.12	1.81		0.24	0.00	0.01
Parent_pos	0.01	0.00	5.22	0.00	0.00	0.01

FIGURE 6.24 – Résultats des preuves des lemmes de la théorie `Arithmetic`.

énoncent des propriétés sur ces fonctions. Ce sont des propriétés arithmétiques qui sont toutes prouvées automatiquement (figure 6.24).

Implantation des tas binaires

Nous définissons dans cette partie une implantation des tas binaires. Les tas sont des arbres binaires complets où la valeur de chaque nœud est supérieure ou égale à celles de ses nœuds fils. Nous proposons une implantation efficace dans laquelle un tas est un arbre binaire stocké dans un tableau, tel que la racine du tas est à l'indice 0 du tableau et pour tout indice i, le fils gauche du nœud stocké à l'indice i est à l'indice $2 * i + 1$ et son fils droit est à l'indice $2 * i + 2$.

Le module `Heap` : Nous définissons d'abord le module `Heap`, dont le code détaillé dans la figure 6.25, permet de décrire des propriétés liées à notre implantation des tas binaires.

Nous commençons par définir le prédicat `is_heap_array(a, idx, sz)` qui est vrai lorsque dans le tableau $a[0 \ldots sz-1]$, le sous arbre dont la racine est à l'indice idx vérifie la propriété de tas c'est-à-dire : la valeur de chaque nœud est inférieure ou égale aux valeurs stockées dans le sous arbre.

Nous posons ensuite des lemmes qui permettent d'énoncer les propriétés que toute instance de type `Binary_Heap` vérifie. Par exemple, nous avons besoin d'indiquer (`Is_heap_when_node_modified`) que : si on rajoute un élément `a` à un tas `h` en respectant les propriétés que la valeur d'un nœud est supérieure à celle de du nœud parent (`Parent_inf_el`), et qu'elle est inférieure à celle de ses fils (`Left_sup_el` et `Right_sup_el`), alors `h` est toujours un tas.

Comme nous pouvons le constater dans la figure 6.26, ces lemmes sont prouvés automatiquement par Alt-ergo et CVC3, excepté le dernier, `Is_heap_relation` qui établit que la racine d'un tas binaire est plus petite que tous les autres éléments du tas. La preuve de ce lemme nécessite une induction sur les entiers et est faite en Coq.

```
/*@ predicate is_heap_array (int *a, integer idx, integer sz) =
  @   0 <= idx ==> \forall integer i, j;
  @     idx <= i < j < sz ==> parentChild (i, j) ==> a[i] <= a[j];
  @*/

//@ predicate Full(heap h) =  h->size == h->capacity;

/*@ lemma Is_heap_when_no_element : \forall int *a, integer idx, n;
  @   0 <= n <= idx ==> is_heap_array(a, idx, n);
  @*/

/*@ lemma Is_heap_sub :
  @   \forall int *a, integer i, n; is_heap_array(a, i, n) ==>
  @    \forall integer j;  i <= j <= n ==> is_heap_array(a, i, j);
  @*/

/*@ lemma Is_heap_sub2 :
  @   \forall int *a, integer n; is_heap_array(a, 0, n) ==>
  @   \forall integer j; 0 <= j <= n ==> is_heap_array(a, j, n);
  @*/

/*@ lemma Is_heap_when_node_modified {L1, L2}:
  @   \forall int *a, *new_a , integer n, idx, int e;
  @     is_heap_array{L1}(a, idx, n) ==>
  @     \forall integer i; 0 <= i < n ==>
  @        (i > 0 ==> \at(a[parent(i)], L1) <= e ) ==>
  @        (left(i)< n ==> e <= \at(a[left(i)], L1)) ==>
  @        (right(i) < n ==> e <= \at(a[right(i)], L1)) ==>
  @        A_set{L1, L2}(a, new_a, i, e) ==> is_heap_array{L2}(new_a, idx, n);
  @*/

/*@ lemma Is_heap_add_last {L1, L2}:
  @   \forall int *a, *new_a, integer n, int e; n > 0 ==>
  @    is_heap_array{L1}(a, 0, n)  &&  (e >= \at(a[parent(n)], L1)) ==>
  @    A_set{L1, L2}(a, new_a, n, e) ==> is_heap_array{L2}(new_a, 0, (n + 1));
  @*/

/*@ lemma Parent_inf_el:
  @   \forall int *a, integer n; is_heap_array(a, 0, n) ==>
  @   \forall integer j; 0 < j < n ==> a[parent(j)] <= a[j];
  @*/

/*@ lemma Left_sup_el:
  @   \forall int *a, integer n; is_heap_array(a, 0, n) ==>
  @   \forall integer j; 0 <= j < n ==> left(j) < n ==> a[j] <=  a[left(j)];
  @*/

/*@ lemma Right_sup_el:
  @   \forall int *a, integer n; is_heap_array(a, 0, n) ==>
  @   \forall integer j; 0 <= j < n ==> right(j) < n ==> a[j] <= a[right(j)];
  @*/

/*@ lemma Is_heap_relation:
  @   \forall int *a, integer n; n > 0 ==> is_heap_array(a, 0, n) ==>
  @   \forall integer j; 0 <= j ==> j < n ==>  a[0] <= a[j];
  @*/
```

FIGURE 6.25 – Le module auxiliaire Heap.

Proof obligations	Alt-Ergo 0.93	CVC3 2.2	Coq 8.3pl2	Eprover 1.4 Namring	Simplify 1.5.4	Vampire 0.6	Yices 1.0.25	Z3 2.19
Is_heap_when_no_element	0.01	0.00			0.00	1.14	0.01	0.01
Is_heap_sub	0.01	0.00			0.00		0.01	0.02
Is_heap_sub2	0.02	0.00			0.00			0.02
Is_heap_when_node_modified	0.42	0.31					0.02	0.05
Is_heap_add_last		0.05					0.01	0.02
Parent_inf_el		0.01		3.95			0.01	0.02
Left_sup_el	0.02	0.01		4.74	0.01	4.85	0.01	0.02
Right_sup_el	0.02	0.01		4.85	0.00	4.76	0.00	0.02
Is_heap_relation			0.59					

FIGURE 6.26 – Résultats des preuves des lemmes du module `Heap`.

```
#define MAX_INT 2147483647

#pragma TypeDecl(Mod_Heap, Heap)
typedef struct Heap{
  int elements[];
  int max_cap;
  int sz;
}*heap;

#pragma Model(Mod_Heap, size)
//@ model struct Heap { integer size };

#pragma Model(Mod_Heap, capacity)
//@ model struct Heap { integer capacity };

#pragma Model(Mod_Heap, values)
//@ model struct Heap { bag values };

#pragma Coupling_invariant(Heap, Inv_Heap)
/*@ type invariant Inv_Heap(heap this) =
  @  this->max_cap == this->capacity && this->sz == this->size &&
  @  0 <= this->size <= this->capacity &&
  @  this->values == elements(this->elements, 0, this->size) &&
  @  is_heap_array(this->elements, 0, this->size);
  @*/
```

FIGURE 6.27 – Définition d'un type structure `Heap`.

Implantation des tas binaires Nous pouvons maintenant définir, dans le module `Binary_Heap`, le type structure `Heap` et les fonctions présentées dans l'interface 6.14.

Le type structure `Heap` définit un tas binaire par trois champs concrets : un tableau d'entiers `elements` qui contient les éléments du tas et deux entiers `sz` et `max_cap` qui représentent respectivement le nombre

d'éléments dans le tas et sa capacité maximale, et trois champs modèles : le multi-ensemble des éléments du tas `values` et `capacity` et `size`. Comme dans le premier défi, présenté dans la section 6.3, les deux derniers champs modèles permettent d'écrire des spécifications abstraites, c'est-à-dire des spécifications qui ne font référence à aucun champ concret.

Un invariant de type est également associé à ce type structure. Cet invariant établit que les champs `cap_max` et `capacity` sont égaux, de même que pour les champs `sz` et `size`, que la taille du tas ne peut pas dépasser sa capacité et enfin que les éléments du tableau `elements` compris entre les indices 0 et `size` vérifient la propriété de tas définie par le prédicat `is_heap_array` dans le module `Heap` défini ci-dessus.

La première fonction, `create_Heap`, dont le code est présenté dans la figure 6.28 crée une nouvelle instance fraîche et vide de type `Heap`, de capacité maximale `sz` et la retourne, comme indiqué dans la figure. Le fait que l'instance retournée par la fonction est fraîchement allouée est mentionné dans la clause **allocates** du contrat. La post-condition de la fonction spécifie que d'une part, la capacité du tas retourné est égale à `sz` et que, d'autre part, ce tas ne contient aucun élément.

```
typedef unsigned int uint;

/*@ requires sz <= MAX_INT <= 0;
  @ allocates \result;
  @ ensures \result->values == empty_bag;
  @ ensures \result->capacity == sz;
  @*/
heap create (uint sz) {
  heap this = (heap)malloc(sizeof(struct Heap));
  this->elements = (int *)calloc (sz, sizeof(int));
  this->size = 0;
  this->capacity = sz;
  return this;
  }
```

FIGURE 6.28 – Spécification de la fonction `create_Heap` du module `Binary_Heap`.

La deuxième fonction du module, `insert`, implante l'algorithme classique d'insertion dans un tas qui rajoute le nouvel élément dans la première case vide. Cette insertion risque de casser la propriété sur les tas binaires, il est nécessaire alors de déplacer cet élément en le remontant, jusqu'à ce que la valeur du nœud parent soit inférieure à celle du nœud inséré. Cet algorithme est illustré dans la figure 6.29 et le code correspondant est présenté dans la figure 6.30.

Nous ne pouvons pas insérer un nouvel élément dans le tas si la taille de celui-ci est égale à sa capacité maximale. C'est cette propriété qui est exprimée dans la précondition de la fonction `insert`. Nous spécifions par ailleurs dans la clause **assigns** du contrat de la fonction, que cette dernière modifie le paramètre `this`. Ensuite, la post-condition permet de décrire la relation entre l'ancienne (avant l'appel) et la nouvelle (à la fin du corps) valeur de `this` : le multi-ensemble des éléments du tas `this` à l'état final est le résultat de l'addition du paramètre `e` au multi-ensemble contenant les éléments de `this` à l'état initial.

Notez que nous ajoutons, dans le corps de la fonction, des assertions et des invariants de boucles, dans le but d'aider les prouveurs automatiques.

La troisième et dernière fonction définie dans le module `Binary_Heap` est la fonction `extract_Min`. Cette fonction retourne le plus petit élément du tas qui est, par construction, la racine de l'arbre repré-

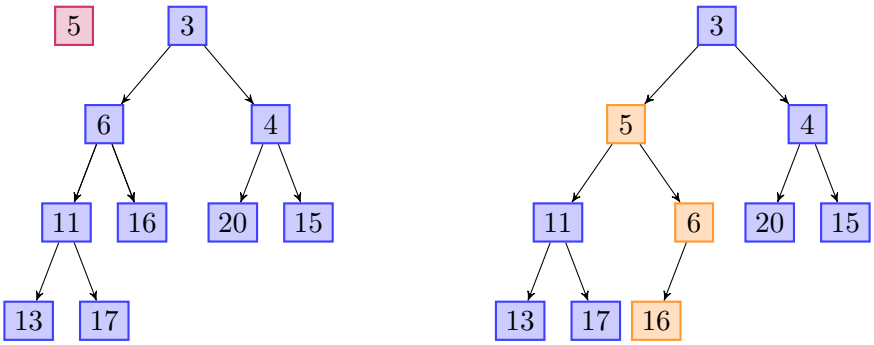


FIGURE 6.29 – Insertion de l'élément 5 dans le tas

sentant le tas. On réinsère ensuite le dernier élément du tableau dans le tas. Les modifications appliquées au tas sont illustrées dans la figure 6.31 et le code de la fonction est donné dans la figure 6.32.

Comme pour la fonction précédente, la clause **assigns** spécifie que le paramètre `this` est modifié. La précondition établit que le multi-ensemble des éléments de `this` n'est pas vide. Enfin la post-condition assure que la valeur retournée, par la fonction, est le plus petit élément dans le tas et donne la relation entre l'état de ce dernier avant et après l'exécution de la fonction, à savoir que le multi-ensemble qui représente le tas contient un élément en moins, qui est le minimum, et que la taille du tas a diminué de 1.

De même que pour la fonction `insert`, les assertions dans le corps de la fonction permettent d'avoir des preuves les plus automatiques possibles.

Programmes clients

Dans la troisième partie de l'architecture (figure 6.16), nous nous intéressons aux programmes clients.

La fonction `heapSort` Elle est d'abord annotée comme on peut le voir dans la figure 6.33. Nous utilisons les fonctions logiques `elements` et `min_bag`, nous avons ainsi besoin d'importer les théories `Elements` et `Bag`.

Le code proposé pour cette fonction dans l'énoncé du challenge contient deux boucles. La première permet de créer un tas binaire à partir des éléments d'un tableau d'entiers `a` en invoquant la fonction `insert`. L'invariant associé à cette boucle dit qu'à la $i^{\text{ème}}$ itération, le tas construit contient exactement les `i` premiers éléments du tableau `a`.

La deuxième boucle effectue le processus inverse, c'est-à-dire extraire les éléments un à un du tas et les insérer dans le tableau. Étant donné que la fonction `extractMin` retourne le plus petit élément du tas, alors le tableau ainsi obtenu est trié. L'invariant de la boucle n'est pas plus compliqué que le premier. À chaque itération, nous avons besoin de préciser la cardinalité du multi-ensemble des éléments du tas, et que ce dernier contient un élément de moins qu'à l'itération précédente. Il est également nécessaire de spécifier que les `i` premiers éléments de `a` sont triés et que les `i-1` éléments extraits sont plus petits que le minimum du tas.

```
/*@ requires size(this) <= MAX_INT - 1;
  @ requires !Full(this);
  @ ensures this->values == Union(singleton(e), \old(this)->values);
  @*/
void insert (heap this,int e) {
  int *arr = this->elements;
  int i = this->size;
  int size = i;
  /*@ loop invariant
    @  0 <= i <= size &&
    @  (i == size ==> is_heap_array(arr, 0, size) &&
    @     elements{Here}(arr, 0, size) == elements{Pre}(arr, 0, size)) &&
    @  (i < size ==> is_heap_array(arr, 0, size + 1) && \at(arr[i], Here) > e &&
    @     (elements{Here}(arr, 0, size + 1) ==
    @     Union (singleton(\at(arr[i], Here)), elements{Pre}(arr, 0, size))));
    @ loop variant i;
    @*/
  while (i > 0) {
    int parent = (i - 1) / 2;
    int p = arr[parent];
    if (e >= p) break;
    L3: arr[i] = p;
    //@ assert A_set{L3, Here}(arr, arr, i, p);
    i = parent;
    //@ assert is_heap_array(arr, 0, size + 1);
  }
  L2: arr[i] = e;
  //@ assert A_set{L2, Here}(arr, arr, i, e);
  this->size += 1;
  /*@ assert i == size ==>
    @   elements{Here}(arr, 0, size + 1) == Union(singleton(e), elements{Pre}(arr, 0, size));
    @*/
  /*@ assert i < size ==>
    @   elements{Here}(arr, 0, size + 1) == Union(singleton(e), elements{Pre}(arr, 0, size));
    @*/
  //@ assert elements{Here}(arr, 0, size + 1) == this->values;
  /*@ assert Union(singleton(e), elements{Pre}(arr, 0, size)) ==
    @                                                  Union(singleton(e), \at(Pre, this->values));
    @*/
  //@ assert i == size  ==> Is_heap{Here}(this);
  //@ assert i < size ==> Is_heap{Here}(this);
  }
```

FIGURE 6.30 – Spécification de la fonction `insert` du module `Binary_Heap`.

La fonction `TestHarness` La deuxième fonction définie dans cette partie permet de tester l'implantation de la fonction `heapSort`. Là encore, nous ajoutons des assertions pour aider les prouveurs automatiques comme indiqué dans la figure 6.34.

Comme on peut le constater dans cette interface, le prédicat `Valid_Heap` spécifié dans les contrats des fonctions permet seulement de vérifier que le pointeur `this` est alloué. En effet, lorsqu'une fonction du module `Heap` est invoquée de l'extérieur du module, il n'est pas nécessaire de vérifier les invariants de données des objets dont le type appartient à ce module.

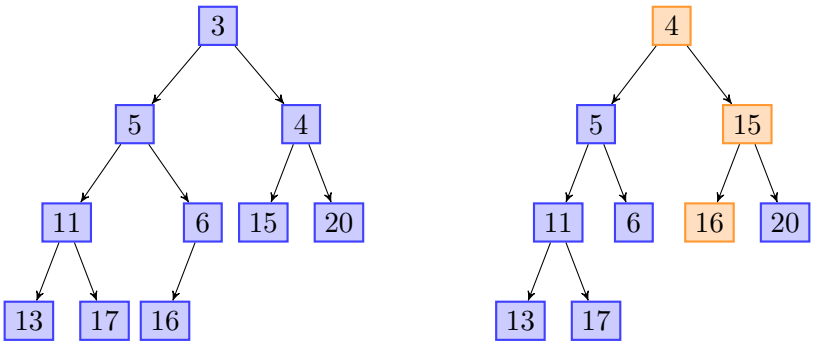


FIGURE 6.31 – Extraction du plus petit élément du tas.

Pour le code annoté qui est présenté dans la figure 6.33, Why3 génère 16 obligations de preuves qui sont, toutes, prouvées automatiquement.

6.5 Conclusion

Nous avons validé notre approche en effectuant la preuve de deux des exemples fournis par le benchmark *VACID-0*, donné comme représentatif des difficultés liées aux invariants de données.

Ces exemples ont été traités aussi par d'autres outils : Dafny et VCC. Vis-à-vis de ces outils, un point fort de notre approche est que lorsque l'on prouve les programmes client, les obligations de preuve sont très simples, car elles ne parlent que des champs modèle et n'exigent jamais de reprouver les invariants de données.

Nous n'avons pas traité les trois autres exemples de *VACID-0*, à savoir les arbres rouges et noirs, la structure d'*union-find* et le pattern *composite*.

Concernant les arbres rouges et noirs, *VACID-0* n'impose pas quelle implémentation utiliser. Classiquement, il existe deux grandes approches : soit on utilise une structure non mutable et l'insertion d'un élément effectue des copies, soit on utilise des arbres modifiables en place. Le premier cas n'a pas d'intérêt pour nous, car il n'y a pas d'effets de bord, c'est plus une version purement fonctionnelle de cette structure [85]. Le deuxième cas ne peut pas être traité par notre approche, car il faudrait poser un invariant sur un type récursif.

Concernant *union-find*, il s'agit d'un exemple où la structure contient des pointeurs permettant du partage arbitraire. Néanmoins, il n'y a à priori pas d'invariant à poser sur les structures en question. Il y a seulement un invariant sur la structure globale. Un tel exemple serait à priori faisable.

Le cas du pattern *composite* est connu comme étant particulièrement ardu. Il s'agit d'une structure d'arbre où chaque nœud contient un pointeur sur son père, il y a donc du partage dans tous les sens. En particulier, on sait qu'il n'est pas traitable par l'approche *Ownership* ou la logique de séparation et certains auteurs ont proposé une approche encore plus complexe pour le traiter [92]. Un cas similaire est celui du design pattern *Subject-Observer* [87]. Là encore, notre approche ne permet pas de traiter ces exemples, du fait du partage trop libéral.

```
/*@ requires this->values != empty_bag;
  @ requires this->size <= (MAX_INT - 2) / 2;
  @ ensures && \result == min_bag(this->values);
  @ ensures this->values == Union (singleton(\result), \old(this)->values);
  @*/
int extractMin (heap this) {
  int *arr = this->elements;
  int min = arr[0];
  int size = (this->size);
  //@ assert elements{Here}(arr, 0, size) == this->values;
  (this->size)--;
  size--;
  //@ assert size >= 0;
  int i = 0;
  int last = arr[size];
  /*@ loop invariant
    @  i >= 0 && is_heap_array(arr, 0, size) &&
    @  (i == 0 ==> elements{Pre}(arr, 0, size) == elements{Here}(arr, 0, size) &&
    @   elements{Pre}(arr, 0, size+1) ==
    @                Union (singleton(last), elements{Pre}(arr, 0, size)))  &&
    @  (0 < i < size ==>
    @         Union (singleton(\at(arr[i], Here)), elements{Pre}(arr, 0, size+1)) ==
    @    Union (singleton(last), Union (singleton(min), elements{Here}(arr, 0, size)))) &&
    @  (i >= size ==> elements{Pre}(arr, 0,  size+1) ==
    @                   Union (singleton(min), elements{Here}(arr, 0, size)))  &&
    @        (i > 0 ==> \at(arr[parent(i)], Here) < last);
    @ loop variant size - i;
    @*/
  while ( i < size) {
    int left = (2 * i) + 1;
    int right = (2 * i) + 2;
    if (left >= size) break;
    int smaller = left;
    if (right < size)
      if (arr[left] > arr[right]) smaller = right;
    if (last <= arr[smaller]) break;
    //@ assert last > arr[smaller];
  L: arr[i]= arr[smaller];
    //@ assert A_set{L, Here}(arr, arr, i, arr[smaller]);
    i = smaller;
  }
  if (i < size) {
  L1: arr[i] = last;
      //@ assert A_set{L1, Here}(arr, arr, i, last);
  }
  //@ assert elements{Here}(arr, 0, size) == this->values;
  return min;
}
```

FIGURE 6.32 – Spécification de la fonction `extract_Min` du module `Binary_Heap`.

```
/*@ predicate Sorted{L}(int *a, integer l, integer h) =
  @   \forall integer i; l <= i < h ==> \at(a[i],L) <= \at(a[i+1],L);
  @*/

/*@ requires \valid_range(arr, 0, len - 1);
  @ ensures \valid_range(arr, 0, len - 1);
  @ ensures  Sorted{Here}(arr, 0, len);
  @ ensures elements(arr, 0, len) == elements(\old(arr), 0, len);
  @*/
void heapSort(int *arr, uint len){
  uint i;
  heap h = create_Heap(len);

/*@ loop invariant
  @   (0 <= i) && (i <= len) &&
  @   (card (h->values) == i) &&
  @   h->values == elements(arr, 0, i);
  @ loop variant len - i;
  @*/
  for(i = 0; i < len; ++i) insert(h, arr[i]);

 /*@ loop invariant
   @   (0 <= i) && (i <= len) &&
   @   (card (h->values) == len - i) &&
   @   elements (\at(arr, Pre), 0, len) == Union (h->values, elements (arr, 0, i)) &&
   @        Sorted{Here}(arr, 0, i) &&
   @    \forall integer j; 0 <= j < i ==> arr[j] <= min_bag (h->values);
   @ loop variant len - i;
   @*/
  for(i = 0; i < len; ++i) arr[i] = extractMin(h);
}
```

FIGURE 6.33 – Le module client `HeapSort`.

```
void heapSortTestHarness() {
  int arr[3] = {42, 13, 42};
  heapSort(arr, 3);
  //@ assert arr[0] <= arr[1] <= arr[2];
  /*@ assert elements{Here}(arr, 0, 3) ==
    @        Unions(singleton(13), Union(singleton(42), singleton(42)));
    @*/
  //@ assert arr[0] == min_bag(elements{Here}(arr, 0, 3));
  //@ assert arr[0] == 13;
  //@ assert arr[1] == min_bag(elements{Here}(arr, 1, 3));
  //@ assert arr[1] == 42;
  //@ assert arr[2] == 42;
}
```

FIGURE 6.34 – Le module client `TestHarness`.

Chapitre 7

Conclusion

7.1 Résumé des contributions

Nous avons présenté, dans un premier temps, une technique de preuve de correction d'un calcul de plus faible précondition, qui imite la technique classique de preuve de *type soundness* d'un langage, en montrant à la fois que la plus faible précondition est préservée par réduction et que la validité de la plus faible précondition entraîne la réductibilité. Comme la satisfaction des annotations d'un programme est par définition l'absence de blocage (sémantique bloquante), nous obtenons une méthode de vérification qui est garantie correcte, y compris sur des programmes non terminants. La définition par sémantique bloquante est proposée par Herms *et al* [59] pour le cas d'une sémantique à grands pas. Nous utilisons une sémantique bloquante à petit pas : notre approche est donc originale à notre connaissance. C'est un langage impératif, dans lequel le seul type non primitif est celui des types structure. Nous avons formalisé cette approche sur un sous-ensemble du langage qui ne contient pas de type structure, de pointeurs ni d'appel de fonction. Cette formalisation a donné lieu à une publication au JFLA [74]. Une démarche de preuve similaire a été utilisée par Conchon et Filliâtre [39] dans un contexte différent : preuve de correction d'une procédure de décision de la sûreté d'utilisation de structures de données semi-persistantes.

Nous avons, dans un second temps, étendu notre langage d'étude afin qu'il contienne les notions de modules, de champs modèles et d'invariants. Ces concepts permettent notamment de structurer les programmes en composants et de les abstraire, ce qui permet de faire des preuves modulaires. Pour que cette approche soit correcte, nous introduisons un système de typage avec régions qui contraint très fortement les possibilités d'aliasing. En effet, ce dernier rend difficile la préservation des invariants de données. Ce système de typage est caché à l'utilisateur. Ceci permet d'écrire des spécifications sans notions compliquées. La sémantique opérationnelle et le calcul de plus faible précondition sont, ensuite étendus pour permettre la gestion des invariants de données qui doivent être vrais au début et à la fin de chaque appel de fonction. La preuve de ce nouveau calcul est également effectuée par la méthode de *type soundness*.

Nous avons étendu, à nouveau, notre langage d'étude en introduisant les variables globales et les tableaux. Nous avons, par ailleurs, partiellement levé la restriction qui consiste à interdire tout aliasing, en autorisant le partage de pointeurs sur un type structure si ce dernier n'est pas muni d'un invariant de données.

Enfin, nous présentons deux études de cas qui permettent de prouver l'implantation, dans un premier

temps, puis l'utilisation des structures de données qui sont les tableaux creux et les tas binaires. Ces études de cas sont tirées du challenge Vacid0 [69]. Le but de ces études est de montrer que l'approche que nous proposons permet de traiter des exemples concrets en utilisant le greffon *Frama-C* que nous avons développé et qui met en œuvre cette approche.

7.2 Limitations

Le langage que nous avons présenté présente des limitations que nous discutons ci-dessous.

Ré-allocation

Il est parfois nécessaire de *ré-allouer* des blocs de mémoire. C'est le cas par exemple si l'on veut implémenter des tableaux redimensionnables (par exemple la classe **Vector** de Java). Dans un tel code, si le tableau est plein, on va typiquement allouer un nouveau tableau de taille double et copier l'ancien dans le nouveau. L'ancien tableau ne sera plus utilisé.

```
function unit add(struct Vector v, int x){
  if (v->size >= v->max_size) {
   v->data = new int[2*v->size];
  } ...
}
```

Le problème est que l'on souhaite cacher l'allocation au programme client, autrement dit, la clause **allocates** devrait être à **nothing**. On souhaiterait aussi que la région du nouveau tableau alloué ne soit pas fraîche, mais celle de l'ancien tableau. Ce type de problème est appelé également *strong update* dans la littérature et est analogue au transfert d'Ownership[93].

Le danger d'un tel code serait que l'ancien tableau `v->data` soit encore référencé dans le reste du code, auquel cas certains invariants pourraient ne pas être préservés. Néanmoins, notre approche interdit ce genre d'alias. Par conséquent, étendre notre approche à ce type de ré-allocation semble naturellement faisable. Il suffirait d'être capable d'identifier par le typage qu'il s'agit d'une ré-allocation. Autrement dit, il faudrait probablement une règle de typage spécifique aux expressions de la forme $e \rightarrow f = \texttt{new}$ ..., qui indiquerait que globalement cette expression n'alloue rien.

Allocation mémoire conditionnelle

Comme nous avons vu dans la section 4.2.6, le système de typage avec régions oblige à effectuer exactement les mêmes allocations dans les deux branches d'une conditionnelle. De manière similaire, on ne peut pas allouer dans le corps d'une boucle, et également on ne peut pas faire de fonction récursive qui alloue. Cette limitation ne devient réellement contraignante que dans les cas des types récursifs dont nous parlons ci-dessous.

Types récursifs

Notre langage n'autorise pas la définition de types récursifs. Par exemple, on ne peut pas travailler avec des listes chaînées ou des arbres. La raison fondamentale est que notre typage impose qu'une région ne contient qu'un seul bloc mémoire (une structure ou un tableau de structures). Intuitivement, une liste chaînée devrait être formée de plusieurs blocs appartenant tous à la même région. Dans une telle situation, il semble impossible, pour notre approche statique par typage, d'autoriser de poser un invariant sur chaque nœud d'une liste individuellement. En effet, le partage possible entre les nœuds d'une liste empêchera de pouvoir contrôler quels invariants seront à rétablir.

Par contre, il nous semble possible d'étendre notre approche si l'on respecte une discipline restreinte d'utilisation de telles listes : il faut introduire une structure particulière en tête de liste. C'est par exemple, la façon dont les listes chaînées sont implémentées dans Java. En C, on écrirait :

```
 struct Node {
  int data;
  struct Node succ;
 }

struct List{
  struct Node list;
}
```

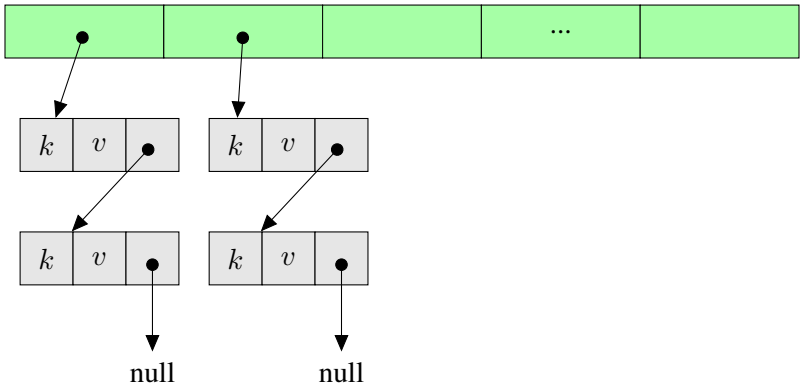
On peut alors typer une telle structure en associant une unique région pour tous les nœuds d'une liste. La $shape$ est alors une abstraction grossière de la structure

$$\{(\rho, list) \rightarrow (true, \textbf{struct}_{\rho_1} Node); (\rho_1, succ) \rightarrow (true, \textbf{struct}_{\rho_1} Node)\}$$

Le typage impose alors que deux listes distinctes ne peuvent pas partager de nœuds. Dans de telles conditions, on peut autoriser le programmeur à poser un invariant sur la structure de liste pouvant mentionner tous les nœuds de sa région. Par contre, il n'est pas autorisé de poser un invariant sur les nœuds eux-mêmes.

Notons que pour poser un invariant sur tous les nœuds d'une liste, il faut probablement utiliser un prédicat inductif.

Une telle extension permettrait d'implémenter notre exemple de mémoïsation à l'aide d'une véritable implémentation des tables de hachage formées d'un tableau où chaque cellule est la tête d'une liste chaînée.



7.3 Comparaisons avec les travaux antérieurs

Dans notre approche, nous avons voulu nous placer à mi-chemin entre deux extrêmes :
- les possibilités offertes à un utilisateur d'un outil comme *Frama-C*, non spécialiste des difficultés posées par les invariants de données dans un langage à pointeurs,
- les méthodes très avancées existantes dans la littérature : logique de séparation, *Ownership*, typage avec régions et capacités, *dynamic frames*, et implémentées dans des outils prototypes.

Apports dans *Frama-C*

Vis-à-vis de l'existant en *Frama-C* et des greffons Jessie et WP, les nouveautés que nous apportons sont le support des invariants de type, de l'allocation dynamique et de l'abstraction avec des champs modèle. Ces extensions devraient répondre à des besoins concrets d'utilisateurs de *Frama-C* pour spécifier des bibliothèques (cf *ACSL by Example* [34]).

Positionnement par rapport aux approches avancées

En comparaison avec les méthodologies avancées de la littérature, notre approche a des limitations, qui ont été discutées dans la section précédente. Elle apporte, néanmoins, des fonctionnalités non fournies par ces techniques avancées : l'approche par raffinement à base de champs modèle et de mise à jour non déterministe de ces champs au retour de chaque appel de fonction. De plus, les spécifications que l'utilisateur doit écrire restent simples : les régions sont implicites, l'utilisateur n'a pas besoin d'indiquer quand l'invariant est temporairement violé (contrairement à l'approche *Ownership*), le langage de spécification est la logique du premier ordre standard, plus naturelle pour un utilisateur moyen qu'une logique comme la logique de séparation. L'utilisateur n'a pas besoin d'indiquer explicitement, dans les spécifications, l'empreinte mémoire d'une structure ou d'une fonction, comme dans l'approche *dynamic frames*.

7.4 Perspectives

Nous présentons ici quelques perspectives.

Une première famille de perspectives concerne les extensions du langage supporté par notre approche. Une première piste est de permettre à l'utilisateur d'annoter explicitement quelles sont les parties publiques et les parties privées de son code :
- permettre des fonctions privées qui ne sont visibles que dans le module auquel elles appartiennent et des fonctions publiques qui sont dans l'interface du module.
- permettre de poser des invariants publics.
- permettre de déclarer des champs concrets publics.

Enfin, il serait intéressant de mettre en œuvre les idées présentées précédemment, pour autoriser les types récursifs.

Un aspect important pour qu'un tel langage permette de développer des modules réutilisables, est de proposer du polymorphisme de type et également de l'ordre supérieur. Par exemple, une fonction de tri générique devrait être paramétrée aussi bien par le type des éléments à trier que par la fonction de

comparaison à utiliser [72, 97]. Ajouter le polymorphisme et l'ordre supérieur à notre langage est un pas supplémentaire non trivial.

Une autre perspective concerne, de manière pratique, le prototype qui implémente notre approche. Le principe de traduire le programme source vers un code C intermédiaire, sans invariants de type, et de donner le résultat à Jessie ou WP est une idée intéressante car elle réutilise les outils existants relativement matures. Il y a deux inconvénients majeurs : d'une part, cette implémentation ne correspond pas strictement au calcul de plus faible précondition formalisé dans cette thèse (voir discussion au début de la section 6.1), d'autre part, cette approche n'est pas idéale car l'information calculée par le système de type est perdue. On pourrait imaginer d'implémenter notre calcul de plus faible précondition, directement sur le code source, mais on perdrait les optimisations existantes des greffons comme WP et Jessie. Nous pensons qu'une meilleure idée serait d'étendre ces greffons pour qu'ils supportent les champs modèles, les invariants de type et globalement l'approche par raffinement proposée dans cette thèse. Techniquement, le greffon Jessie possède déjà un calcul de régions, qui permet d'introduire une variable par région dans les obligations de preuve, il faudrait alors fusionner ces deux calculs.

Une autre perspective intéressante serait de faire un développement formel de notre approche, qui étendrait le travail que nous avons fait sur un langage jouet, publié aux JFLA'13.

Du point de vue des études de cas, il serait intéressant de traiter les exemples présentés dans *"ACSL by example"*[34], en particulier en appliquant la méthodologie que nous avons définie. L'objectif à long terme, qui devrait être poursuivi, est de développer des bibliothèques prouvées, réutilisables. Il s'agit d'une étape cruciale pour permettre l'applicabilité de la vérification déductive sur des programmes de taille conséquente.

Le travail présenté dans cette thèse n'est pas spécifique au langage C, il peut naturellement être utilisé pour des langages similaires. Par exemple, le langage *Spark2014* [8] est un sous-ensemble de *Ada*, dédié au développement d'applications critiques. Ce langage est muni de contrats, mais pas encore d'invariants de type. Or, ce langage impose des restrictions fortes sur les alias possibles. C'est donc un candidat idéal pour appliquer notre approche.

fra

Bibliographie

[1] Atelier B projects. http://www.methode-b.com/documentation_b/ClearSy-Industrial_Use_of_B.pdf.

[2] Common criteria for information technology security evaluation. http://www.commoncriteriaportal.org.

[3] Controle de vitesse par balises. http://www.metro-pole.net/expl/signal/kvb.html.

[4] Functional safety of electrical, electronic, programmable electronic safety-related systems. http://www.iec.ch/zone/fsafety.

[5] Langley fomal methods. http://shemesh.larc.nasa.gov/people/cam/ACCoRD/.

[6] Roissy cdg val. http://www.cdgfacile.com/?id=121.

[7] Software horror stories. http://www.cs.tau.ac.il/~nachumd/horror.html.

[8] Spark 2014. expanding the boundaries of safe and secure programming. http://www.spark-2014.org/.

[9] Uml. http://www.uml.org/.

[10] Vacid solutions. http://vacid.codeplex.com/.

[11] Wp plug-in. http://frama-c.com/wp.html.

[12] M. Abadi and L. Lamport. The existence of refinement mappings. 1988. Technical report, Digital Systems Research Center, Palo Alto, California.

[13] J.-R. Abrial. *The B-Book, assigning programs to meaning*. Cambridge University Press, 1996.

[14] R.-J. Back and J. von Wright. *Refinement Calculus : A Systematic Introduction*. Springer-Verlag, 1998.

[15] R.-J. J. Back. *On the correctness of refinement in program development*. PhD thesis, Department of Computer Science, University of Helsinki, 1978.

[16] R. Bardou. *Verification of Pointer Programs Using Regions and Permissions*. Thèse de doctorat, Université Paris-Sud, Oct. 2011. http://proval.lri.fr/publications/bardou11phd.pdf.

[17] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6) :27–56, June 2004.

[18] M. Barnett, R. DeLine, B. Jacobs, B.-Y. E. Chang, and K. R. M. Leino. Boogie : A Modular Reusable Verifier for Object-Oriented Programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects : 4th International Symposium*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, 2005.

[19] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System : An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[20] C. Barrett and C. Tinelli. CVC3. In Damm and Hermanns [45], pages 298–302.

[21] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL : ANSI/ISO C Specification Language*, 2008. `http://frama-c.cea.fr/acsl.html`.

[22] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software : The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.

[23] B. Beckert and C. Marché, editors. *Formal Verification of Object-Oriented Software, Papers Presented at the International Conference*, Karlsruhe Reports in Informatics, Paris, France, June 2010. `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000019083`.

[24] P. Behm, P. Desforges, and J.-M. Meynadier. Météor : An industrial success in formal development. In *B*, page 26, 1998.

[25] J. Berdine, C. Calcagno, and P. 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.

[26] P. Berthomé, K. Heydemann, X. Kauffmann-Tourkestansky, and J.-F. Lalande. Attack model for verification of interval security properties for smart card c codes. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '10, pages 2 :1–2 :12, New York, NY, USA, 2010. ACM.

[27] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

[28] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The Alt-Ergo automated theorem prover, 2008. `http://alt-ergo.lri.fr/`.

[29] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3 : Shepherd your herd of provers. In *Boogie 2011 : First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[30] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *PLDI*, pages 578–589, 2011.

[31] S. Boulmé and M.-L. Potet. Interpreting invariant composition in the B method using the Spec# ownership relation : a way to explain and relax B restrictions. In J. Julliand and O. Kouchnarenko, editors, *B 2007*, volume 4355 of *Lecture Notes in Computer Science*. Springer, 2007.

[32] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2004.

[33] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3) :212–232, June 2005.

[34] J. Burghardt, J. Gerlach, L. Gu, K. Hartig, H. Pohl, J. Soto, and K. Völlinger. ACSL by example, towards a verified C standard library. Technical report, Fraunhofer First, 2011. `http://www.first.fraunhofer.de/fileadmin/FIRST/ACSL-by-Example.pdf`.

[35] D. Ceara, L. Mounier, and M.-L. Potet. Taint dependency sequences : A characterization of insecure execution paths based on input-sensitive cause sequences. In *ICST Workshops*, pages 371–380, 2010.

[36] A. Charguéraud. Characteristic formulae for the verification of imperative programs. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming (ICFP)*, pages 418–430, Tokyo, Japan, September 2011. ACM.

[37] A. Charguéraud and F. Pottier. Functional translation of a calculus of capabilities. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 213–224, Sept. 2008.

[38] D. R. Cok and J. Kiniry. ESC/Java2 : Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.

[39] S. Conchon and J.-C. Filliâtre. Semi-Persistent Data Structures. In *17th European Symposium on Programming (ESOP'08)*, Budapest, Hungary, Apr. 2008.

[40] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 262–275. ACM Press, 1999.

[41] P. Cuoq, D. Delmas, V. M. Lamiel, and S. Duprat. Fan-C, a Frama-C plug-in for data flow verification. In *Embedded Real Time Software and Systems*, 2012.

[42] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C : a software analysis perspective. In *Proceedings of the 10th international conference on Software Engineering and Formal Methods*, SEFM12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[43] P. Cuoq and V. Prevosto. *Value Plugin Documentation,* Carbon *version*. CEA-List, 2011. `http://frama-c.com/download/frama-c-value-analysis.pdf`.

[44] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC : Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 429–430. IEEE Comp. Soc. Press, 2009.

[45] W. Damm and H. Hermanns, editors. *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, Berlin, Germany, July 2007. Springer.

[46] M. Daumas and G. Melquiond. Certification of bounds on expressions involving rounded operators. *Transactions on Mathematical Software*, 37(1), 2010.

[47] L. de Moura and N. Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[48] D. Delmas, S. Duprat, V. M. Lamiel, and J. Signoles. Taster, a Frama-C plug-in to enforce coding standards. In *Embedded Real Time Software and Systems*, 2010.

[49] J.-C. Demay, E. Totel, and F. Tronel. Sidan : A tool dedicated to software instrumentation for detecting attacks on non-control-data. In *CRiSIS*, pages 51–58, 2009.

[50] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify : a theorem prover for program checking. *J. ACM*, 52(3) :365–473, 2005.

[51] E. W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.

[52] B. Dutertre and L. de Moura. The Yices SMT solver. available at `http://yices.csl.sri.com/tool-paper.pdf`, 2006.

[53] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In R. D. Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2007.

[54] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Damm and Hermanns [45], pages 173–177.

[55] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[56] The Frama-C platform for static analysis of C programs, 2008. `http://www.frama-c.cea.fr/`.

[57] A. Goodloe, C. Muñoz, F. Kirchner, and L. Correnson. Verification of floating point programs : From real numbers to floating point numbers. In *NASA Formal Methods Symposium*, 2013.

[58] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Programming language design and implementation (PLDI)*, pages 282–293, 2002.

[59] P. Herms, C. Marché, and B. Monate. A certified multi-prover verification condition generator. In R. Joshi, P. Müller, and A. Podelski, editors, *Verified Software : Theories, Tools, Experiments (4th International Conference VSTTE)*, volume 7152 of *Lecture Notes in Computer Science*, pages 2–17, Philadelphia, USA, Jan. 2012. Springer.

[60] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580 and 583, Oct. 1969.

[61] T. Hubert. *Analyse Statique et preuve de Programmes Industriels Critiques*. Thèse de doctorat, Université Paris-Sud, June 2008.

[62] B. Jacobs and F. Piessens. The VeriFast program verifier. CW Reports CW520, Department of Computer Science, K.U.Leuven, August 2008.

[63] B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *Programming Languages and Systems (APLAS 2010)*, pages 304–311. Springer-Verlag, November 2010.

[64] I. T. Kassios. Dynamic frames : Support for framing, dependencies and sharing without restrictions. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *14th International Symposium on Formal Methods (FM'06)*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283, Hamilton, Canada, 2006.

[65] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4 : Formal verification of an OS kernel. *Communications of the ACM*, 53(6) :107–115, June 2010.

[66] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 2007.

[67] K. R. M. Leino. Data groups : Specifying the modification of extended state. In *OOPSLA*, pages 144–153, 1998.

[68] K. R. M. Leino. Dafny : An Automatic Program Verifier for Functional Correctness. In Springer, editor, *LPAR-16*, volume 6355, pages 348–370, 2010.

[69] K. R. M. Leino and M. Moskal. VACID-0 : Verification of ample correctness of invariants of data-structures, edition 0. In *Proceedings of Tools and Experiments Workshop at VSTTE*, 2010.

[70] K. R. M. Leino and M. Moskal. VACID-0 : Verification of ample correctness of invariants of data-structures, edition 0. In *VSTTE*, 2010.

[71] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *PLDI*. ACM, 2002.

[72] C. Marché. Towards modular algebraic specifications for pointer programs : a case study. In *Rewriting, Computation and Proof*, volume 4600 of *Lecture Notes in Computer Science*, pages 235–258. Springer, 2007.

[73] C. Marché and A. Tafat. Weakest precondition calculus, revisited using Why3. Research Report RR-8185, INRIA, Dec. 2012.

[74] C. Marché and A. Tafat. Calcul de plus faible précondition, revisité en Why3. In *Vingt-quatrièmes Journées Francophones des Langages Applicatifs*, Aussois, France, Feb. 2013.

[75] B. Meyer. *Eiffel : The Language*. Prentice Hall, Hemel Hempstead, 1992.

[76] R. Milner. A theory of type polymorphismn in programming. *J. Comput. Syst. Sci.*, 17, 1978.

[77] C. Morgan. *Programming from specifications (2nd ed.)*. Prentice Hall International (UK) Ltd., 1994.

[78] J. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9 :287–306, 1987.

[79] Y. Moy and C. Marché. Modular inference of subprogram contracts for safety checking. *Journal of Symbolic Computation*, 45 :1184–1211, 2010.

[80] Y. Moy and C. Marché. *The Jessie plugin for Deduction Verification in Frama-C — Tutorial and Reference Manual*. INRIA & LRI, 2011. `http://krakatoa.lri.fr/`.

[81] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare type theory. In J. H. Reppy and J. L. Lawall, editors, *11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006*, pages 62–73, Portland, Oregon, USA, 2006. ACM Press.

[82] T. M. T. Nguyen. *Taking architecture and compiler into account in formal proofs of numerical programs*. Thèse de doctorat, Université Paris-Sud, June 2012.

[83] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[84] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL '01 : Proceedings of the 15th International Workshop on Computer Science Logic*, pages 1–19, London, UK, 2001. Springer-Verlag.

[85] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[86] D. Pariente and E. Ledinot. Formal verification of industrial C code using Frama-C : a case study. In Beckert and Marché [23], pages 205–219. `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000019083`.

[87] M. Parkinson. Class invariants : The end of the road ? In T. Wrigstad, editor, *3rd International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO), in conjunction with ECOOP 2007*, Berlin, Germany, July 2007. `http://www.cs.purdue.edu/homes/wrigstad/iwaco/`.

[88] G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61 :3–15, 2004.

[89] The PVS system. `http://pvs.csl.sri.com/`.

[90] J. C. Reynolds. Separation logic : a logic for shared mutable data structures. In *17h Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.

[91] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames : Combining dynamic frames and separation logic. In S. Drossopoulou, editor, *ECOOP 2009 — Object-Oriented Programming*, Lecture Notes in Computer Science, pages 148–172. Springer Berlin / Heidelberg, 2009.

[92] A. J. Summers and S. Drossopoulou. Considerate reasoning and the composite design pattern. In G. Barthe and M. V. Hermenegildo, editors, *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 328–344. Springer, 2010.

[93] A. Tafat, S. Boulmé, and C. Marché. A refinement methodology for object-oriented programs. In Beckert and Marché [23], pages 143–159. `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000019083`.

[94] A. Tafat and C. Marché. Binary heaps formally verified in Why3. Research Report 7780, INRIA, Oct. 2011. http://hal.inria.fr/inria-00636083/en/.

[95] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3) :245–271, 1992.

[96] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 1997.

[97] E. Tushkanova, A. Giorgetti, C. Marché, and O. Kouchnarenko. Specifying generic Java programs : two case studies. In *PreProceedings of LDTA'2010*, pages 92–106, 2010. http://ldta.info/preproceedings2010.pdf.

[98] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. S. Fitzgerald. Formal methods : Practice and experience. *ACM Comput. Surv.*, 41(4), 2009.

[99] B. Yakobowski, P. Cuoq, P. Hilsenkopf, F. Kirchner, S. Labbé, and N. Thuy. Formal verification of software important to safety using the frama-c tool suite. In *NPIC*, 2012.