Formalisation d'un vérificateur dynamique de propriétés mémoire pour programmes C

Dara Ly



UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE MATHÉMATIQUES, INFORMATIQUE, PHYSIQUE THÉORIQUE ET INGÉNIERIE DES SYSTÈMES

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLÉANS

LABORATOIRE DE SÛRETÉ ET SÉCURITÉ DES LOGICIELS, CEA LIST

THÈSE présentée par :

Dara LY

soutenue le : **5 décembre 2022**

pour obtenir le grade de : **Docteur de l'Université d'Orléans**

Discipline : **Informatique**

Formalisation d'un vérificateur dynamique de propriétés mémoire pour programmes C

THÈSE dirigée par :

M. Jean-Michel COUVREUR	Professeur, Université d'Orléans
M. Frédéric LOULERGUE	Professeur, Université d'Orléans

RAPPORTEURS :

M. Claude MARCHÉ	Directeur de Recherche, INRIA
M. Alan SCHMITT	Directeur de Recherche, INRIA

JURY :

M. Jean-Christophe FILLIÂTRE	Directeur de recherche, CNRS, Président du jury
M. Nikolai KOSMATOV	Ingénieur-chercheur, Thales
M. François PESSAUX	Professeur associé, ENSTA Paris
M. Julien SIGNOLES	Ingénieur-chercheur, CEA LIST

Remerciements

Si ce manuscrit représente l'aboutissement d'un travail de plusieurs années, alors on peut légitimement considérer que les différentes personnes qui sont intervenues dans ce long processus ont, chacune à leur façon, contribué au résultat final. Un certain nombre de remerciements sont par conséquent de mise.

Tout d'abord, je souhaite remercier les membres du jury d'avoir accepté d'examiner mes travaux, et notamment les rapporteurs Alan Schmitt et Claude Marché pour leur relecture et leurs commentaires. Je remercie par ailleurs spécialement François Pessaux, dont la responsabilité dans l'existence de cette thèse n'est pas des moindres puisque c'est lui qui m'a initié, à l'ENSTA, aux joies des langages de programmation.

Merci à Jean-Michel Couvreur et Frédéric Loulergue d'avoir assuré à quatre mains le rôle de directeur de thèse dans tous ses aspects. Merci en particulier à Frédéric de m'avoir accueilli en Arizona du Nord chaque fois que l'occasion se présentait.

Au CEA LIST, où se déroulait mon quotidien de doctorant, j'ai bénéficié de l'encadrement de Nikolai Kosmatov et Julien Signoles. Merci, donc, à Nikolai pour sa redoutable efficacité en tant que co-auteur, et ses remarques toujours si précises sur les aspects techniques. Et merci à Julien d'avoir été disponible au quotidien pour examiner mes idées, répondre à mes questions, et plus généralement de m'avoir guidé avec une rare bienveillance tout le long du cheminement tortueux qu'a été cette thèse.

Je voudrais également remercier mes collègues du Laboratoire de Sécurité et de Sûreté du Logiciel. Les discussions avec les uns et les autres, sur des sujets scientifiques ou tout autres, ont largement contribué à faire du laboratoire un cadre de travail à la fois stimulant et agréable.

Parmi mes collègues, je remercie tout particulièrement le groupe protéiforme et informellement défini des « jeunes du DILS ». Merci, donc, à celles et ceux avec qui j'ai eu le plaisir de partager qui l'organisation d'un séminaire au vert (ou au verre…), qui des TD à donner à l'ENSTA, qui une d'école d'été, qui des séances d'escalade, de potin ou de pâtisserie, qui des ateliers d'autohébergement ou d'installation de logiciels libres… Et, en arrière-plan de tous ces moments, une réconfortante solidarité dans les affres partagées du doctorat.

Ces quelques mots seraient incomplets s'ils ne mentionnaient pas mes parents. Bien que ma gratitude à leur égard englobe bien plus que ce seul motif, je voudrais les remercier ici pour m'avoir soutenu durant ces années de thèse, ainsi que toutes les précédentes.

Enfin, du fond du cœur, merci Eugénie. Merci pour tout.

Table des matières

Chapitre 1

Introduction

Du fait d'une numérisation dont l'extension ne semble pas connaître de limite, les systèmes techniques conçus aujourd'hui tendent à inclure presque systématiquement une composante informatique, et bien souvent à être pilotés par elle. En conséquence, une part toujours plus importante des activités de nos sociétés, bâties sur des infrastructures technologiques complexes, se trouve dépendre du bon fonctionnement de systèmes informatiques, et est ainsi proportionnellement affectée par leurs dysfonctionnements.

Or, les défaillances de systèmes informatiques sont monnaie courante, tout particulièrement en ce qui concerne les logiciels : on les appelle alors *bugs* [1], et ils sont pour ainsi dire omniprésents. Si les plus bénins (et fréquents) sont sans conséquence notable, des bugs graves ont par le passé occasionné toutes sortes de dégâts, allant jusqu'à la perte de vies humaines. Ce fut par exemple le cas dans la seconde moitié des années 1980, lorsqu'un bug du logiciel de calibration du Therac-25, un appareil de radiothérapie, entraîna l'exposition de patients à d'importantes doses de rayonnements ionisants, et le décès de plusieurs d'entre eux [LT93].

En sus des dysfonctionnements accidentels, les bugs ouvrent également la voie aux attaques informatiques, des altérations (détournement, mise hors-service, infiltration, prise de contrôle…) des systèmes informatiques résultant d'une volonté délibérée. Là encore, les conséquences sont proportionnelles aux enjeux confiés à ces systèmes.

On comprend donc l'intérêt, voire la nécessité, de développer des méthodes et techniques permettant de garantir le bon fonctionnement des programmes. Selon leur approche du logiciel et de son développement, on les situera dans deux grandes catégories. La première résulte de l'application de la méthode expérimentale au phénomène du développement logiciel, et est caractéristique du champ du génie logiciel empirique. À l'inverse, la seconde consiste à considérer les programmes comme des objets mathématiques, et à raisonner dessus en conséquence. Cette approche est prédominante dans le champ des méthodes formelles, dans lequel se placent les travaux exposés ici.

1.1 Vérification logicielle

On peut définir la vérification de programmes comme le problème consistant à déterminer, pour un programme donné et une *spécification* de ce programme, si le programme est conforme à sa spécification. Notons qu'en méthodes formelles, le terme de spécification désigne généralement un objet défini mathématiquement, à l'instar des programmes. Autrement dit, on s'intéresse principalement aux *spécifications formelles*, à distinguer des spécifications en langue naturelle, qui sont les plus courantes en développement logiciel.

1. ou *bogues* par francisation

La vérification de programmes a constitué un sujet d'étude important dès les tout débuts de l'histoire de la programmation (voir [Sig18] pour une chronologie détaillée). L'effort de recherche résultant a engendré un large éventail de techniques, avec chacune ses forces et limitations propres.

1.1.1 Techniques statiques

Les techniques dites *statiques* permettent, en étudiant le seul code source d'un programme, de raisonner sur toutes les exécutions possibles de celui-ci. Il s'agit là d'une caractéristique hautement désirable, étant donné qu'un même programme peut bien souvent admettre un grand nombre d'exécutions potentielles différentes, voire un nombre infini. Les analyses statiques sont cependant confrontées à une difficulté majeure, à savoir l'impossibilité de déterminer algorithmiquement (et en temps fini) si un programme donné satisfait une propriété sémantique donnée. Cette impossibilité est la conséquence d'un résultat de calculabilité, le théorème de RICE [Ric53] : elle est donc sans appel.

Les analyses statiques résultent donc de compromis autour de l'analyse « idéale » interdite par le théorème de Rice. Par exemple, la *vérification déductive*, une famille de techniques fondée par les travaux de HOARE [Hoa69], FLOYD [Flo67] et DIJKSTRA [Dij75], abandonne partiellement le caractère automatique, de sorte que le processus de vérification nécessite une intervention humaine, de plus ou moins grande envergure selon le cas.

Une autre approche possible est de renoncer au caractère conclusif de l'analyse. Ainsi, l'*interprétation abstraite* [CC77] permet de calculer automatiquement un sur-ensemble de l'ensemble des exécutions possibles d'un programme. Si tous les éléments de ce sur-ensemble sont des exécutions correctes, le programme est alors nécessairement correct ; si toutefois ce n'est pas le cas, l'analyse ne permet pas de conclure, ni dans un sens ni dans l'autre.

1.1.2 Techniques dynamiques

À l'inverse, les techniques dites *dynamiques* analysent le programme étudié durant son exécution. Elles ne peuvent donc pas prétendre à l'exhaustivité, mais bénéficient en contrepartie d'un accès direct, par observation, à des informations sur l'exécution du programme potentiellement difficiles (ou impossibles) à inférer statiquement.

Le *test* constitue de loin la méthode de vérification dynamique la plus répandue. Nous adoptons ici la définition de MEYER : *tester un programme, c'est tenter de le faire échouer* [Mey08]. Pour tester un programme vis-à-vis de sa spécification, on exécute le programme sur une entrée donnée (un *cas de test*), et l'on contrôle par le biais d'un *oracle* la conformité du résultat obtenu à la spécification. La démarche est empirique par nature : chaque cas de test constitue une tentative de réfutation de l'hypothèse de conformité du programme à la spécification. Un échec du cas de test montre la présence d'un bug dans le programme, tandis qu'un déroulement du test sans échec corrobore l'hypothèse de conformité.

Une autre technique dynamique est la vérification d'assertions à l'exécution (*Runtime Assertion Checking* en anglais, parfois abrégé en RAC). Comme son nom le laisse entendre, cette technique repose sur une spécification donnée sous forme d'annotations insérées dans le code : les *assertions*. Une assertion est une annotation qui sert de point de contrôle : placée en un endroit bien choisi du code source, elle garantit que l'état de l'exécution à cet endroit satisfait une certaine propriété. Si c'est bien le cas, l'exécution se poursuit normalement ; sinon, elle est immédiatement interrompue. L'usage judicieux des assertions permet d'éviter les cas de figures, particulièrement ardus à déboguer, où une erreur fausse subtilement le comportement du programme, sans que cela ne se manifeste de prime abord (par exemple par un message

d'erreur du programme ou de son environnement d'exécution), pour ensuite provoquer des dysfonctionnements bien plus loin au cours de l'exécution.

1.2 Outils de vérification pour le langage C

La mise en œuvre pratique des techniques présentées dans la section précédente nécessite la conception et le développement d'outils dédiés. Les travaux présentés dans cette thèse sont étroitement liés au développement d'un tel outil : il s'agit du greffon E-ACSL de la plateforme Frama-C, que l'on présente dans les lignes qui suivent.

1.2.1 Analyses de programmes C : l'outil Frama-C

Parmi les nombreux et divers langages qui tissent l'histoire de la programmation, le C occupe assurément une place à part. Apparu dans les années 1970 pour le développement du système d'exploitation Unix, ce langage a non seulement fortement influencé l'évolution des langages de programmation, mais il demeure encore aujourd'hui le langage de référence pour les programmes dits de bas niveau (c'est-à-dire situés à un niveau d'abstraction proche du matériel). En tant que tel, il s'agit du langage utilisé pour la plupart des noyaux de système d'exploitation, des pilotes de périphériques, et des systèmes embarqués. Ainsi, le constat fait plus haut de l'omniprésence des systèmes logiciels demeure largement valable une fois restreint aux seuls programmes C.

Or, la programmation en C est un exercice délicat. Le langage, de bas niveau, conçu pour permettre au programmeur d'exploiter toutes les possibilités de la machine sous-jacente, ne pose en revanche aucun garde-fou (ou presque) pour l'empêcher de commettre des erreurs. En particulier, le C nécessite le plus souvent de gérer explicitement la mémoire utilisée pour les données du programme. Les mécanismes prévus pour cela dans le langage, notamment les *pointeurs*, objets désignant un emplacement en mémoire, sont à la fois indispensables et notoirement difficiles à utiliser sans commettre d'erreur.

La vérification de programmes C présente par conséquent un intérêt considérable. Néanmoins, elle constitue également un défi de taille. En effet, en raison d'une multitude de facteurs aussi bien théoriques que pratiques, l'analyse de programmes C est, en toute généralité, un problème difficile à traiter.

La plateforme Frama-C [Kir+15] propose en réponse à ce défi une infrastructure logicielle destinée à faciliter la mise en œuvre de diverses analyses de programmes C, et à permettre l'interaction et la collaboration entre celles-ci. Elle repose, pour la spécification des programmes, sur un langage d'annotations dédié, nommé ACSL (acronyme de *ANSI/ISO C Specification Language*, soit « langage de spécification du C norme ANSI/ISO »).

Conçu, comme son nom l'indique, pour le langage C, ACSL est par ailleurs doté d'une sémantique suffisamment précise pour y appuyer des analyses de programme. La syntaxe des annotations ACSL, dont un exemple apparaît dans la figure 1.1, fait qu'elles sont considérées comme des commentaires en C, donc ignorées par les outils comme les compilateurs C. Elles sont en revanche interprétées par les outils de Frama-C.

L'objectif de Frama-C se reflète dans son architecture logicielle : la plateforme est composée d'un *noyau*, autour duquel sont articulés un certain nombre de *greffons*. Un greffon est un composant logiciel assurant une fonction bien précise, en général une analyse ou une transformation de programme. Le noyau, quant à lui, fournit aux greffons un ensemble de services de base et, surtout, coordonne leur action conjointe.

```
1  int search(int *t, int len, int x) {
2    int lo = 0;
3    int hi = len - 1;
4    while (lo <= hi) {
5      int mid = lo + (hi - lo) / 2;
6      /*@ assert(\valid(t + mid)); */
7      if (t[mid] == x)
8        return mid;
9      else if (t[mid] < x)
10       lo = mid + 1;
11     else
12       hi = mid - 1;
13   }
14   return -1;
15 }
16
17 int main(void) {
18   int t[5] = { -3, 2, 4, 7, 10 };
19   return search(t, 5, 7);
20 }
```

FIGURE 1.1 – Un programme de recherche dichotomique dans un tableau, écrit en C et contenant une annotation en ACSL.

1.2.2 Vérification dynamique en C : l'outil E-ACSL

Si Frama-C a au départ été pensé pour le développement d'analyses statiques, la plateforme comprend aussi des greffons d'analyse dynamique. Parmi eux se trouve E-ACSL, un outil de vérification dynamique qui possède notamment des fonctionnalités de vérification d'assertions à l'exécution. Il tire son nom du langage E-ACSL, pour *Executable ACSL* : il s'agit en effet d'un sous-ensemble d'ACSL, choisi pour rendre possible la transformation d'annotations en code C, donc exécutable. Les possibilités de vérification offertes par E-ACSL sont assez étendues [Sig21], mais dans notre cadre, on le considèrera exclusivement sous l'angle de la vérification d'assertions à l'exécution.

Étant donné que le langage C contient déjà, dans sa bibliothèque standard, une construction d'assertion, le bénéfice d'un outil dédié tel qu'E-ACSL provient principalement de ce qu'il permet de vérifier des propriétés qui ne peuvent être exprimées à l'aide d'assertions C classiques. En particulier, les assertions en E-ACSL peuvent exprimer des propriétés sur l'état de la mémoire du programme (ou *propriétés mémoire*) : c'est par exemple le cas de l'assertion à la ligne 6 de la figure 1.1. La vérification à l'exécution de ce type de propriétés constitue cependant un problème technique complexe, ce dont on peut se faire une idée sommaire en comparant la version d'origine de notre programme d'exemple au résultat de son traitement par E-ACSL (figure 1.2).

E-ACSL munit le programme à analyser d'un *moniteur* en ligne, c'est-à-dire d'un programme secondaire, intégré dans celui d'origine, et chargé de contrôler son exécution. Concrètement, la génération d'un moniteur pour le programme de la figure 1.1, aussi appelée *instrumentation*, produit en sortie un code tel que celui de la figure 1.2.

Le rôle primaire du moniteur est de contrôler la conformité de l'exécution du programme à la spécification définie par les annotations (en langage E-ACSL), et d'interrompre l'exécution en cas de non-conformité. Mais pour être en mesure de déterminer la validité des assertions, il lui faut également collecter des informations tout au long de l'exécution. Au rôle primaire de *contrôle* se superpose donc un rôle secondaire d'*observation*.

Cette dualité des rôles transparaît dans le code généré pour notre programme d'exemple,

```
1  int search(int *t, int len, int x)
2  {
3    int __retres;
4    __e_acsl_store_block((void *)(& t),8UL);
5    int lo = 0;
6    int hi = len - 1;
7    while (lo <= hi) {
8      int mid = lo + (hi - lo) / 2;
9      {
10       int __gen_e_acsl_valid;
11       __gen_e_acsl_valid = __e_acsl_valid((void *)(t + mid),sizeof(int),
12                                           (void *)t,(void *)(& t));
13       __e_acsl_assert(__gen_e_acsl_valid,"Assertion","search",
14                       "\\valid(t + mid)","ex.c",7);
15     }
16     /*@ assert \valid(t + mid); */ ;
17     if (*(t + mid) == x) {
18       __retres = mid;
19       goto return_label;
20     }
21     else
22       if (*(t + mid) < x) lo = mid + 1; else hi = mid - 1;
23   }
24   __retres = -1;
25   return_label: {
26                   __e_acsl_delete_block((void *)(& t));
27                   return __retres;
28                 }
29 }
30
31 int main(void)
32 {
33   int tmp;
34   __e_acsl_memory_init((int *)0,(char ***)0,8UL);
35   int t[5] = {-3, 2, 4, 7, 10};
36   __e_acsl_store_block((void *)(t),20UL);
37   __e_acsl_full_init((void *)(& t));
38   tmp = search(t,5,7);
39   __e_acsl_delete_block((void *)(t));
40   __e_acsl_memory_clean();
41   return tmp;
42 }
```

FIGURE 1.2 – Le programme de la figure 1.1, après génération de code C à partir des annotations, sur une architecture 64 bits.

lequel comporte, par rapport au programme d'origine, un certain nombre de lignes additionnelles. Sans chercher à comprendre précisément chacune d'entre elles, on peut remarquer que du code a été généré au niveau de l'assertion mentionnée précédemment, sous la forme d'un bloc délimité par des accolades (lignes 9 à 15) : ce code correspond à l'évaluation de l'assertion, c'est-à-dire au rôle de contrôle du moniteur.

La différence ne s'arrête toutefois pas là : des lignes ont également été ajoutées à divers autres endroits du programme (qui a d'ailleurs approximativement doublé en taille, sur cet exemple). Ces lignes correspondent au rôle d'observation du moniteur. Si l'on examine succinctement leur contenu, on s'aperçoit que beaucoup font appel à des fonctions, dont le nom a pour préfixe `__e_acsl`. Ces fonctions sont les primitives sur lesquelles s'appuie le moniteur

pour observer et contrôler le programme instrumenté, et constituent la bibliothèque d'exécution (en anglais *Run Time Library*), qui est liée avec tout programme instrumenté par E-ACSL. L'ensemble du processus d'instrumentation est schématisé par la figure 1.3.

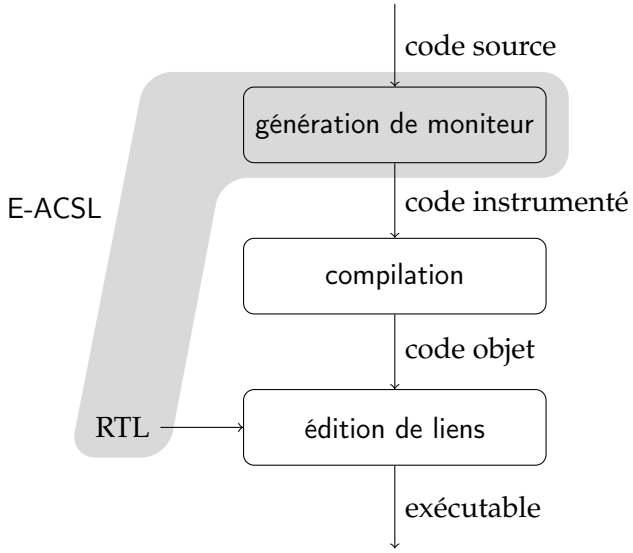

FIGURE 1.3 – Les étapes de traitement d'un programme avec E-ACSL. L'abréviation RTL désigne la bibliothèque d'exécution (*Run Time Library*) d'E-ACSL. La zone grisée indique les composants d'E-ACSL.

1.3 Contributions

Dans cette thèse, on se propose de modéliser formellement la transformation de programme au cœur de l'outil E-ACSL. L'objectif global de cette étude est d'affiner la compréhension de la transformation, et notamment des propriétés sémantiques conférées aux programmes instrumentés.

Les bénéfices d'une telle démarche sont multiples. Par exemple, un modèle formel peut constituer un argument fort en faveur de la *sûreté* de l'outil, que l'on définit ici comme sa capacité à détecter de manière exacte les assertions invalides.

Il permet d'identifier et d'expliciter les hypothèses utilisées pour garantir les propriétés recherchées, et de cartographier l'espace des compromis (performance, expressivité, sûreté…) dans lequel se situe l'outil.

Enfin, un modèle formel peut permettre d'identifier des axes d'amélioration pour la mise en œuvre. Il peut par exemple permettre d'effectuer des optimisations de performance tout en conservant la sûreté.

Afin d'étudier l'instrumentation de programmes à des fins de vérification à l'exécution, nous commençons par définir dans le chapitre 2 un langage représentatif du problème à l'étude, c'est-à-dire un langage impératif permettant d'une part de manipuler explicitement la mémoire, et d'autre part d'exprimer des assertions sur des propriétés relatives à celle-ci. Un travail d'adaptation est réalisé à partir de l'état de l'art en matière de sémantiques formelles du langage C, à savoir la sémantique utilisée pour le compilateur certifié CompCert [Ler09], et en particulier son *modèle mémoire*. On définit ainsi ce que nous appellerons notre langage *source*.

Le chapitre 3 est consacré à la définition de la transformation de programmes qui constitue le cœur de nos travaux. On y introduit un second modèle de la mémoire, dit *mémoire d'observa-*

tion, qui représente les données connues du moniteur de vérification à l'exécution. L'interface entre le moniteur de vérification à l'exécution et la mémoire d'observation est modélisée par un langage dérivé du langage source : le langage *cible*. Ces éléments étant posés, on définit la transformation comme une fonction associant à tout programme en langage source un programme en langage cible.

Le chapitre 4 porte sur l'étude des propriétés de la transformation précédemment définie. Il s'agit de caractériser formellement le comportement d'un programme transformé, par rapport au programme d'origine. Le chapitre est en majeure partie composé des théorèmes formant cette caractérisation, ainsi que de leurs interprétations et preuves respectives.

Les chapitres 2 à 4 constituent un ensemble cohérent, dont les principales contributions ont été publiées (avec moins de détails techniques) dans l'article « Verified Runtime Assertion Checking for Memory Properties » [Ly+20].

Les travaux présentés dans le chapitre 5 sont motivés par une question d'ordre pratique : celle des performances des programmes instrumentés, en termes de temps d'exécution et d'empreinte mémoire. Le sujet de ce chapitre est une analyse de flot de données, conçue pour servir de support à une optimisation de l'instrumentation. L'analyse est conçue avec la sûreté comme objectif prioritaire, c'est-à-dire que les optimisations qu'elle engendre ne doivent pas introduire de comportements incorrects dans les programmes optimisés. Dans cette optique, on démontre un théorème exprimant la sûreté de l'analyse présentée. Le contenu de ce chapitre a fait l'objet de l'article « Soundness of a Dataflow Analysis for Memory Monitoring » [Ly+19], et dépend des chapitres 2 et 3.

Enfin, le chapitre 6 synthétise les contributions et présente les différentes perspectives qui en découlent.

Chapitre 2

Langage d'étude

Ce chapitre présente le langage central de notre étude, qui sert de support aux différentes contributions exposées dans les chapitres 3 à 5. À partir des principales caractéristiques du problème à étudier, on définit dans la section 2.1 la syntaxe de ce langage. La notion de mémoire jouant un rôle central tout le long de cette thèse, la section 2.2 présente la modélisation formelle que nous avons adoptée pour la représenter. On peut dès lors s'appuyer dessus pour définir une sémantique formelle du langage d'étude, ce qui est fait dans la section 2.3. La dernière section du chapitre présente une discussion sur les différents choix de conception du langage, et le situe par rapport aux travaux existants (section 2.4).

2.1 Motivations et périmètre de la formalisation

Étant donné un programme écrit en langage C et muni d'une spécification formelle sous forme d'annotations, le générateur de code E-ACSL réalise son instrumentation, c'est-à-dire l'insertion d'instructions mettant en œuvre un moniteur en ligne, dont le rôle est de contrôler à l'exécution la validité des annotations présentes dans le programme source. On peut donc considérer qu'il s'agit d'une transformation d'un programme C en un autre programme *dans le même langage*.

Toutefois, les fonctionnalités du moniteur lui-même sont déterminées par les annotations adjointes au programme, qui constituent une forme de spécification. En ce sens, on peut voir l'action de l'outil E-ACSL comme une traduction d'un langage (dit *source*) vers un autre langage (dit *cible*) distinct. Suivant ce point de vue, le langage C étendu par le langage de spécification exécutable E-ACSL constitue le langage de programmation source pour le programmeur. Celui-ci peut par exemple écrire un programme tel que celui de la figure 2.1, qui comprend une annotation formelle en E-ACSL pour vérifier l'accès à une case de tableau. Les programmes sources sont ensuite traduits en C « pur » (au sens où les programmes sont intégralement en C), qui est de ce fait le langage cible du processus de traduction. Ainsi, la traduction du programme source de la figure 2.1 donne un programme en C, où l'annotation `assert` est remplacée par une section de code C ayant pour effet de vérifier la propriété indiquée. L'annotation elle-même est usuellement laissée dans le code dans un but de documentation, mais elle est désormais considérée comme un simple commentaire, sans incidence sur l'exécution du programme.

Une telle approche offre deux avantages. D'une part, en matérialisant la spécification du programme sous la forme d'un langage d'annotations, elle facilite l'étude sémantique de la transformation.

D'autre part, elle permet de s'abstraire de certains aspects de l'instrumentation que nous ne souhaitons pas traiter ici. En effet, la mise en œuvre du moniteur en ligne fait appel à une bibliothèque auxiliaire, écrite elle aussi en langage C, qui lui fournit une certaine structure

```
1  int search(int *t, int length, int x) {
2    int lo = 0, hi = length - 1;
3    while (lo <= hi) {
4      int mid = lo + (hi - lo) / 2;
5      /*@ assert(\valid(t + mid)); */
6      if (t[mid] == x) return mid;
7      else if (t[mid] < x) lo = mid + 1;
8      else hi = mid - 1;
9    }
10   return -1;
11 }
12
13 int main(void) {
14   int t[5] = { -3, 2, 4, 7, 10 };
15   return search(t, 5, 7);
16 }
```

FIGURE 2.1 – Un exemple de programme C annoté en ACSL.

de données indispensable à son fonctionnement (nous y reviendrons à la section 3.2). Tandis qu'une modélisation du problème avec un unique langage nous imposerait de raisonner directement sur le code de cette bibliothèque, c'est-à-dire *in fine* en effectuer la spécification et la vérification, l'emploi d'un langage cible distinct nous permet d'intégrer directement dans notre langage cible la bibliothèque auxiliaire et ses propriétés.

À ce stade, la modélisation du problème se compose de trois parties : un langage cible représentant le langage C, un langage source formé par l'adjonction au premier d'un langage de spécification, et une traduction du second vers le premier. Il s'agit maintenant de délimiter les fragments de C (pour la programmation) et d'E-ACSL (pour la spécification) qui seront l'objet de notre attention.

Le point focal de cette thèse est la vérification d'assertions portant sur des propriétés liées à l'état de la mémoire (on abrègera parfois ce terme en *propriétés mémoire*). La composante « programmation » commune aux deux langages devra donc refléter les capacités du langage C en termes de manipulation de la mémoire ; quant à la composante « spécification », elle devra permettre d'exprimer les propriétés jugées utiles à cet égard.

2.1.1 Un noyau de langage impératif

L'essence d'un langage impératif tel que C tient en une poignée de constructions syntaxiques, communément appelée langage While [NNH99], et présentées dans la figure 2.2.

Les expressions de ce langage sont composées de constantes entières, de variables et d'opérateurs (arithmétiques par exemple) unaires et binaires. Ces derniers ne jouent pas un rôle très important dans notre problème, aussi ils ne sont pas explicités mais seulement désignés par les symboles génériques $\dagger$ et $\ddagger$. On les supposera suffisants pour pourvoir à nos besoins. Une instruction peut être :

— l'instruction nulle `skip`, qui n'a aucun effet ;
— l'affectation $e_1 := e_2$; de la valeur de l'expression e_2 à l'expression e_1 ; à ce stade, cela n'a de sens que dans le cas où e_1 est une variable, mais nous allons rapidement étendre cette construction ;
— une séquence d'instructions $s_1 s_2$, exécutées l'une après l'autre ;
— une instruction conditionnelle `if`(e) `then` s_1 `else` s_2, exécutant la branche s_1 si la condition définie par e est vraie, et la branche s_2 sinon ;

$e ::=$		expressions
$\mid$	n	entier constant
$\mid$	x	variable
$\mid$	$\dagger\, e$	opérateur unaire
$\mid$	$e \ddagger e$	opérateur binaire
$s ::=$		instructions
$\mid$	`skip;`	instruction nulle
$\mid$	$e := e$;	affectation
$\mid$	$s\ s$	séquence
$\mid$	`if`(e) `then` s `else` s	instruction conditionnelle
$\mid$	`while`(e) s	boucle

FIGURE 2.2 – Grammaire d'un noyau de langage impératif.

— une boucle `while`(e) s, exécutant l'instruction s si la condition définie par e est vraie, et répétant l'exécution tant qu'elle le demeure.

2.1.2 Gestion de la mémoire

Ajoutons maintenant les mécanismes de gestion de la mémoire présents en C. Pour ce faire, nous étendons la grammaire de notre langage avec les constructions syntaxiques données dans la figure 2.3.

$e ::=$	...	
$\mid$	$*e$	déréférencement
$\mid$	$\&e$	adresse
$s ::=$	...	
$\mid$	$e :=$ `malloc`(e);	allocation
$\mid$	`free`(e);	déallocation
$\mid$	`let` $x : \tau$ `in` s `end`	variable locale

FIGURE 2.3 – Adjonction de pointeurs et de primitives de gestion de la mémoire.

Les expressions sont enrichies de deux constructions : la prise d'adresse, qui permet d'obtenir un pointeur vers un emplacement en mémoire, et le déréférencement d'un pointeur, par lequel on accède aux données résidant à l'emplacement pointé.

Pour pouvoir utiliser l'arithmétique de pointeurs, c'est-à-dire faire des calculs sur des valeurs de pointeurs, il est nécessaire que les opérateurs binaires du langage incluent un opérateur $+$ qui gère l'addition d'un entier à un pointeur.

Conjointement aux pointeurs, le mécanisme clé de la gestion de la mémoire par le programmeur est l'allocation dynamique d'espace en mémoire. Ainsi, la fonction `malloc` alloue un espace dont la taille en octets est déterminée par son argument, et renvoie normalement un pointeur vers la zone allouée. Son complémentaire, la fonction de déallocation `free`, prend en argument un tel pointeur et libère l'espace précédemment alloué. Notre langage ne dispose pas

de fonctions, mais nous modélisons ces deux fonctions particulières de la bibliothèque standard de C par des instructions dédiées.

Par ailleurs, outre l'allocation manuelle de mémoire sur le tas, le langage C présente également une forme rudimentaire de gestion automatique de la mémoire : les déclarations de variables locales à des blocs. Nous les modélisons par une instruction `let` : l'instruction `let` x : τ `in` s `end` alloue, sans l'initialiser, un espace en mémoire pour une variable x de type τ (voir la section suivante), exécute l'instruction s, puis libère la mémoire allouée pour x.

2.1.3 Système de types

Avec l'introduction des pointeurs, notre langage ne manipule plus seulement des nombres entiers mais aussi les valeurs abstraites que sont les pointeurs. En outre, notre formalisation comprend des entiers dont le codage en binaire peut se faire sur plusieurs tailles différentes, donnant ainsi lieu à autant de types différents.

$$\begin{array}{rcll} \tau ::= & \texttt{int}k,\ k \in \{8, 16, 32, 64\} & & \text{type entier} \\ \mid & \tau* & & \text{type pointeur} \end{array}$$

Dans un tel langage, un système de types régit habituellement les interactions possibles entre les objets intervenant dans le programme en fonction de leurs types : ainsi on pourra considérer que l'addition de deux entiers d'une taille donnée résulte en un entier de même taille, que l'addition d'un entier à un pointeur donne un pointeur, mais que la multiplication de deux pointeurs n'a pas de sens et juger mal formé tout programme qui en inclurait une.

La définition d'un système de types statique ne présente pas d'intérêt vis-à-vis de notre problème, et n'est pas indispensable à la formalisation, aussi nous avons fait le choix de nous en passer. À la place, nous supposons toute expression de notre programme munie d'une annotation de type, et définissons l'évaluation des programmes (section 2.3) de manière à tenir compte de ces annotations.

L'annotation explicite de chaque nœud de l'arbre de syntaxe des expressions est susceptible d'alourdir le texte au point d'entraver considérablement la lisibilité ; à titre d'exemple, une expression aussi simple que $p + 17$ pourrait donner une fois annotée le résultat suivant :

$$((p : \texttt{int64}*) \ + \ (17 : \texttt{int32})) : \texttt{int64}*.$$

On préférera par conséquent laisser le plus souvent *implicites* les annotations de type des expressions, et ne faire apparaître les annotations de type que lorsqu'elles sont nécessaires. Comme dans l'exemple précédent, on notera alors $e : \tau$ pour signifier que l'expression e est annotée avec le type τ.

2.1.4 Spécification

Il ne reste désormais qu'à doter notre langage source de capacités de vérification dynamique, sous forme d'assertions. Il s'agit là d'une addition assez conséquente, comme le montre la figure 2.4. Une assertion est une instruction comprenant un prédicat d'une certaine logique, lequel exprime une propriété sur l'exécution du programme en cours. Si le prédicat est faux, le programme s'interrompt immédiatement. Dans le cas contraire, l'exécution se poursuit normalement (l'assertion est sans effet).

La définition du langage de prédicats, autrement dit de la *logique* dans laquelle sont exprimées les propriétés à vérifier, mérite une attention particulière. En effet, le pouvoir d'expression de notre langage — la richesse des propriétés qu'il permettra d'exprimer — en dépend principalement. Dans cette thèse, nous nous concentrons sur les propriétés liées à la mémoire, et plus particulièrement sur les propriétés dites *spatiales* [Sze+13], c'est-à-dire relatives à l'état de

$s ::=$	`...`	
$\mid$	`logical_assert`(t);	assertion
$p ::=$		prédicats
$\mid$	`\true`	vrai
$\mid$	`\false`	faux
$\mid$	$t \bowtie t$	comparaison
$\mid$	$p \wedge p$	conjonction
$\mid$	$\neg\, p$	négation
$\mid$	`\valid`(t)	validité
$\mid$	`\initialized`(t)	initialisation
$t ::=$		termes
$\mid$	e	expression
$\mid$	`\base_address`(t)	adresse de base
$\mid$	`\offset`(t)	décalage d'un pointeur
$\mid$	`\block_length`(t)	longueur d'un bloc
$\mid$	$\circledast t$	déréférencement
$\mid$	$\bar{+}\, t$	opérateur unaire
$\mid$	$t \,\bar{+}\, t$	opérateur binaire
$\bowtie ::=$	$=$	égalité
$\mid$	$\leq$	inégalité

FIGURE 2.4 – Adjonction d'assertions logiques.

la mémoire à un instant donné, par exemple le fait que l'accès à une case d'un tableau se fasse bien à un indice compris entre les bornes du tableau. On ne traitera donc pas les propriétés dites *temporelles* [CR06], utilisées notamment pour caractériser l'absence d'erreurs telles que l'utilisation d'un pointeur après déallocation de la zone mémoire vers laquelle il pointait, alors qu'un autre bloc a pu être alloué dans cette zone mémoire depuis (« *use after free* ») ou la déallocation d'une zone mémoire déjà déallouée auparavant (« *double free* »).

Notre langage logique est une logique propositionnelle comportant les connecteurs logiques de conjonction et négation (disjonction et implication pouvant être encodées à partir de ceux-ci) et les constantes `\true` et `\false`. On y ajoute des prédicats portants sur des *termes* qui représentent des objets liés à l'exécution du programme (notamment à sa mémoire), au sens large. Un prédicat peut ainsi être une comparaison de termes, à l'aide d'opérateurs relationnels : égalité et relations d'ordre sur les entiers. Enfin, deux prédicats spécifiques, `\valid`() et `\initialized`(), permettent d'exprimer des propriétés de pointeurs. Étant donné un terme t de type pointeur, le prédicat `\valid`(t) exprime la validité de ce pointeur, c'est-à-dire la possibilité de le déréférencer (en lecture ou en écriture) sans provoquer d'erreur. Le prédicat `\initialized`(t), quant à lui, signifie que les données pointées par un pointeur t ont été initialisées, c'est-à-dire ont été écrites au moins une fois.

Les termes sont une généralisation des expressions du langage de programmation, permettant de représenter les valeurs manipulées par le programme. On les obtient par adjonction aux expressions de trois constructions servant à décrire l'état de la mémoire utilisée par le programme, et illustrées dans la figure 2.5 : `\block_length`(), `\base_address`() et `\offset`(). Étant donné un terme t évalué en un pointeur vers un certain bloc de mémoire (c'est-à-dire

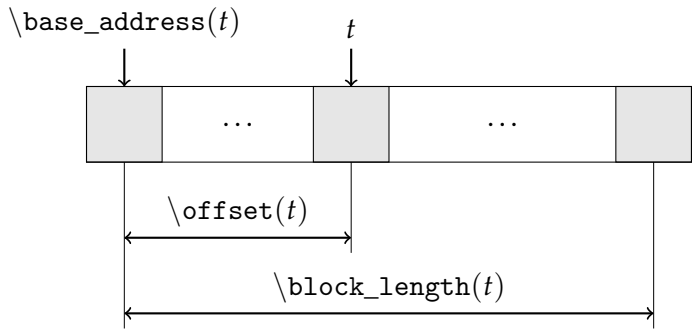

FIGURE 2.5 – Propriétés liées aux blocs.

un ensemble d'emplacements contigüs, résultant d'une unique allocation), elles expriment des propriétés relatives à ce bloc. Si par exemple, t pointe vers la case d'indice i d'un tableau, $\texttt{\textbackslash block\_length}(t)$ désigne la taille (en octets) du tableau, c'est-à-dire le nombre de cases du tableau multiplié par la taille en octets d'une case ; $\texttt{\textbackslash base\_address}(t)$ est un pointeur vers la case d'indice 0 de ce même tableau ; enfin $\texttt{\textbackslash offset}(t)$ est le décalage (en octets) de t par rapport au début du tableau, c'est-à-dire i multiplié par la taille en octets de chaque case du tableau.

Puisque $\texttt{\textbackslash base\_address}(t)$ est un pointeur, il nous faut une construction syntaxique pour le déréférencer : en effet, l'opérateur de déréférencement $*$ n'est applicable qu'aux expressions. Nous définissons donc le déréférencement *pour les termes*, noté $\bar{*}$. De même, afin de permettre l'arithmétique de pointeur avec des termes, nous définissons $\bar{+}$ et $\bar{-}$, les équivalents des opérateurs unaires et binaires des expressions.

Les termes sont, de même que les expressions, munis d'un système de type, qui dans le cas de notre étude restera suffisamment simple pour pouvoir être omis. Précisons seulement que les termes relatifs à la mémoire ont les types naturellement associés à leur signification : type entier pour $\texttt{\textbackslash offset}(t)$ et $\texttt{\textbackslash block\_length}(t)$, pointeur pour $\texttt{\textbackslash base\_address}(t)$.

Nous obtenons finalement le langage source présenté par la figure 2.6 : un noyau de langage C, centré sur la gestion de la mémoire, et étendu par des assertions sur des propriétés, elles aussi centrées sur la mémoire. Le code d'exemple de la figure 2.1 peut être transcrit dans ce langage d'étude, à quelques adaptations près. Le code présenté dans figure 2.7 donne un exemple d'une telle transcription, illustrant ainsi un usage pratique du langage.

2.2 Modèle mémoire

La définition de notre langage est étroitement liée à la notion de mémoire. Cette caractéristique, déjà apparente dans la syntaxe du langage, passe au tout premier plan lorsqu'il s'agit d'en définir la sémantique. Puisque nos programmes sont en grande partie composés d'instructions sur l'utilisation de la mémoire, définir le sens de ces programmes implique nécessairement de décrire leurs interactions avec elle, interactions qui nécessitent à leur tour une définition de ce qu'est la mémoire. Une telle définition formelle est appelée un *modèle mémoire*. Dans cette section, nous présentons celui adopté pour notre langage d'étude.

Une mémoire est conceptuellement un contenant structuré dans lequel on peut entreposer des données à un certain *emplacement* par une opération d'écriture. Elles y sont conservées de manière à pouvoir les y retrouver ultérieurement par une opération de lecture. D'autre part, l'espace mémoire est une ressource : avant de pouvoir l'utiliser il faut en requérir une certaine quantité, par une opération d'allocation. Un espace mémoire qui n'est plus utilisé peut être restitué, par une opération de déallocation.

expr	$e ::=$	n	entier constant
	$\mid$	x	variable
	$\mid$	$*e$	déréférencement
	$\mid$	$\&e$	adresse
	$\mid$	$\dagger\, e$	opérateur unaire
	$\mid$	$e \ddagger e$	opérateur binaire
term	$t ::=$	e	expression
	$\mid$	$\bar{*}t$	déréférencement
	$\mid$	$\bar{\dagger}\, t$	opérateur unaire
	$\mid$	$t \bar{\ddagger} t$	opérateur binaire
	$\mid$	$\backslash\texttt{base\_address}(t)$	adresse de base
	$\mid$	$\backslash\texttt{offset}(t)$	décalage d'un pointeur
	$\mid$	$\backslash\texttt{block\_length}(t)$	longueur d'un bloc
pred	$p ::=$	$\backslash\texttt{true} \mid \backslash\texttt{false}$	vrai, faux
	$\mid$	$t \bowtie t$	comparaison
	$\mid$	$p \wedge p$	conjonction
	$\mid$	$\neg\, p$	négation
	$\mid$	$\backslash\texttt{valid}(t)$	validité d'un pointeur
	$\mid$	$\backslash\texttt{initialized}(t)$	initialisation
stmt	$s ::=$	$\texttt{skip};$	instruction nulle
	$\mid$	$e = e;$	affectation
	$\mid$	$e = \texttt{malloc}(e);$	allocation
	$\mid$	$\texttt{free}(e);$	déallocation
	$\mid$	$\texttt{logical\_assert}(p);$	assertion logique
	$\mid$	$s\ s$	séquence
	$\mid$	$\texttt{if}(e)\ \texttt{then}\ s\ \texttt{else}\ s$	instruction conditionnelle
	$\mid$	$\texttt{while}(e)\ s$	boucle
	$\mid$	$\texttt{let}\ x : \tau\ \texttt{in}\ s\ \texttt{end}$	variable locale
ctyp	$\tau ::=$	$\texttt{int}k,\ k \in \{8, 16, 32, 64\}$	type entier
	$\mid$	$\tau*$	type pointeur

FIGURE 2.6 – Syntaxe du langage source. Le type (au niveau formel) associé à chaque catégorie syntaxique est donné en gras.

```
let t: int64* in
let length: int64 in
let x: int64 in
  length = 17;
  t = malloc (length * sizeof(int64));
  x = 0;
  while (x < 17)
    *(t + x) = 2 * x + 1;
    x = x + 1;

  x = 5;
  let idx: int64 in
  let lo: int64 in
  let hi: int64 in
    idx = -1;
    lo = 0;
    hi = length - 1 ;
    while (lo <= hi)
    let mid: int64 in
      mid = lo + (hi - lo) / 2;
      logical_assert(\valid(t + mid));
      if (*(t + mid) == x) then
        idx = mid;
        lo = hi + 1;
      else if (*(t + mid) < x) then
        lo = mid + 1;
      else
        hi = mid - 1;
    end
  end
  end
  end
end
end
end
```

FIGURE 2.7 – Un programme de recherche dichotomique en langage source.

Ce qui nous importe ici est le comportement de la mémoire vu de l'extérieur, et non son fonctionnement interne. C'est pourquoi nous allons la définir sous la forme d'une *spécification algébrique* [ST12], c'est-à-dire d'un type abstrait accompagné d'un ensemble de fonctions (ou *opérations*) permettant de manipuler des valeurs de ce type, ainsi que d'une théorie axiomatique décrivant ces opérations.

2.2.1 Le modèle mémoire de CompCert : présentation générale

Le modèle mémoire que nous employons ici est une adaptation de celui du compilateur certifié CompCert [Ler09], dans sa première version [LB08]. Le choix de ce modèle particulier tient notamment à sa structuration en blocs, qui correspond relativement bien à la notion de bloc intervenant dans les spécifications en E-ACSL. Par ailleurs, s'agissant d'un modèle mémoire conçu pour un compilateur C, il est également un bon choix lorsqu'il s'agit de définir des sous-parties de ce langage. Les modifications apportées au modèle mémoire original sont détaillées dans la section 2.4.

On peut classifier les modèles mémoire selon leur degré d'abstraction. Ceux dits de bas niveau exposent une représentation proche du matériel, où la mémoire est typiquement vue

comme un tableau d'octets (voire de bits), linéaire et uniforme, sans plus de structure. *A contrario*, les modèles dits de haut niveau présentent une vision plus abstraite et plus structurée, incluant des notions qui traduisent directement certaines entités sémantiques du langage d'étude. Par exemple, un modèle mémoire pour un langage dans lequel les listes jouent un rôle prépondérant peut inclure une représentation directe de celles-ci, sans passer par un encodage de plus bas niveau.

On peut considérer que le modèle mémoire de CompCert se situe à un niveau intermédiaire, car il présente la mémoire de manière légèrement plus structurée qu'un simple tableau d'octets. Dans ce modèle, la mémoire est constituée d'un ensemble de *blocs*. Un bloc peut être vu comme un tableau d'octets de taille fixe, possédant un identifiant unique (abstrait). L'adresse d'un octet consiste donc en la paire de l'identifiant du bloc auquel il appartient, et de l'indice de l'octet dans le tableau — aussi appelé *décalage*.

Les données sont enregistrées en mémoire sous forme de *valeurs*, et leur accès paramétré par un type, que l'on appellera *type mémoire* pour le distinguer des types du langage. Les valeurs (type **value**), et les types mémoire (type **mtyp**) sont définis dans la figure 2.8. Une valeur peut être un entier, un pointeur (composé d'un bloc et d'un entier indiquant un décalage d'un certain nombre d'octets par rapport au début de ce bloc), ou la valeur spéciale Undef indiquant une donnée sans signification (par exemple, présente à un emplacement mémoire non initialisé).

On note **mem** le type des états mémoire, et **block** celui des (identifiants de) blocs. L'un comme l'autre sont laissés abstraits, et ils sont accompagnés de quatre opérations : allocation (alloc) d'un nouveau bloc de mémoire, déallocation (free) d'un bloc auparavant alloué, écriture (store) d'une valeur à une position donnée, et lecture (load) de la valeur présente à une certaine position. Ces opérations sont accompagnées de deux fonctions auxiliaires, qui font également l'objet d'une axiomatisation. La première est le prédicat définissant la *validité* d'un bloc (que l'on note $\vDash$ de manière infixe), tandis que la seconde (notée length) en donne la *taille*. Ces opérations possèdent les signatures de type présentées dans la figure 2.9.

Chacune des quatre opérations principales prend en argument l'état courant de la mémoire et, à l'exception de la lecture (qui ne le modifie pas), renvoie l'état résultant de l'application de l'opération. Par ailleurs, les opérations de déallocation, d'écriture et de lecture sont susceptibles d'échouer si elles sont appelées avec des arguments invalides (la notion de validité fait elle-même l'objet d'une définition spécifique dans la section 2.2.2). Cela se traduit par l'usage d'un type option dans leur type de retour : une opération ayant le type de retour **option t** renverra ε en cas d'échec, et $\lfloor v \rfloor$ (où v est de type **t**) en cas de succès.

Allocation Étant donné une taille n en octets, $\mathsf{alloc}(M_1, n)$ renvoie un couple (b, M_2) où b est un bloc nouvellement alloué, de taille n. On ne s'intéresse pas ici au cas où l'allocation peut

$$
\begin{array}{lrll}
\textbf{value} & v ::= & \mathsf{Int}(n) & \text{entier} \\
& \mid & \mathsf{Ptr}(b, \delta) & \text{pointeur} \\
& \mid & \mathsf{Undef} & \text{indéfini} \\
\\
\textbf{mtyp} & \kappa ::= & \mathrm{i}\,8 & \\
& \mid & \mathrm{i}\,16 & \\
& \mid & \mathrm{i}\,32 & \\
& \mid & \mathrm{i}\,64 &
\end{array}
$$

FIGURE 2.8 – Valeurs du langage et types en mémoire.

$$
\begin{aligned}
\mathsf{alloc}: &\quad \mathbf{mem} \times \mathbb{N} \to \mathbf{block} \times \mathbf{mem} \\
\mathsf{free}: &\quad \mathbf{mem} \times \mathbf{block} \to \mathbf{option\ mem} \\
\mathsf{store}: &\quad \mathbf{mtyp} \times \mathbf{mem} \times \mathbf{block} \times \mathbb{Z} \times \mathbf{value} \to \mathbf{option\ mem} \\
\mathsf{load}: &\quad \mathbf{mtyp} \times \mathbf{mem} \times \mathbf{block} \times \mathbb{Z} \to \mathbf{option\ value} \\
\cdot \vDash \cdot: &\quad \mathbf{mem} \times \mathbf{block} \to \mathbf{bool} \\
\mathsf{length}: &\quad \mathbf{mem} \times \mathbf{block} \to \mathbb{N}
\end{aligned}
$$

FIGURE 2.9 – Signatures des opérations mémoire.

échouer à cause d'une insuffisance de la mémoire disponible dans le système, et on considère par conséquent que l'opération d'allocation réussit inconditionnellement, ce qui revient à supposer un système pourvu d'une mémoire infinie.

Déallocation Étant donné, dans l'état M_1, un bloc alloué b, la déallocation $\mathsf{free}(M_1, b)$ renvoie un état $\lfloor M_2 \rfloor$ dans lequel b n'est plus alloué. Si ce n'est pas le cas, soit parce que b n'a jamais été alloué, soit parce sa déallocation a déjà eu lieu, l'opération renvoie ε (autrement dit, elle échoue).

Écriture. $\mathsf{store}(\kappa, M_1, b, \delta, v)$ dénote l'écriture d'une valeur v de type κ à une position (b, δ) dans l'état M_1. Son résultat, si elle réussit, est un nouvel état mémoire M_2 dont le contenu est identique à celui de M_1 sauf en ce qui concerne l'emplacement (b, δ), qui contient la valeur v. L'écriture échoue si le type mémoire κ ne permet pas de représenter la valeur v, ou bien si (b, δ) n'est pas un emplacement valide pour une écriture de type κ.

Lecture Si (b, δ) est dans M_1 un emplacement valide contenant une valeur v de type κ, l'opération $\mathsf{load}(\kappa, M_1, b, \delta)$ renvoie $\lfloor v \rfloor$. Sinon, elle échoue.

Validité La validité d'un bloc b dans un état mémoire M, notée $M \vDash b$, représente la possibilité d'accéder au bloc b pour y écrire ou y lire des données. Elle est acquise par allocation, et perdue par déallocation.

Longueur $length(M, b)$ dénote la longueur (ou *taille*) en octets du bloc b dans M, c'est-à-dire le nombre d'octets contigus disponibles pour y enregistrer des données. La longueur d'un bloc est définie lors de son allocation, et ne varie pas par la suite.

2.2.2 Axiomatisation

Le comportement des opérations mémoire est déterminé par un jeu d'axiomes, qui constituent une théorie utilisable pour définir et raisonner sur des sémantiques du langage. L'organisation de ces axiomes correspond, pour la majeure partie d'entre eux, à la combinatoire des séquences de deux opérations mémoire, illustrée dans la figure 2.10 : il s'agit en somme de décrire l'évolution des fonctions de « lecture » (au sens large : validité, longueur, lecture) après application des fonctions « d'écriture » (allocation, déallocation, écriture).

Dans la suite de cette section, on présente les axiomes du modèle mémoire en suivant cette classification. On expose d'abord ceux qui définissent la *validité* des blocs, ensuite ceux traitant de leur *taille*, et enfin ceux ayant trait au *contenu* de la mémoire. Un quatrième groupe est formé de divers axiomes qui échappent à la classification ci-dessus.

après	alloc	free	store
$\models$	2.1, 2.2	2.3, 2.4, 2.5	2.6
length	2.7, 2.8	2.9	2.10
load	2.11, 2.12	2.13	2.14, 2.15, 2.16

FIGURE 2.10 – Typologie des principaux axiomes du modèle mémoire.

Beaucoup de ces axiomes ont une formulation comportant grand nombre de variables quantifiées universellement. Afin d'alléger la présentation, on considérera, dans ce contexte, que les variables libres sont implicitement universellement quantifiées.

Validité des blocs La première série d'axiomes spécifie la notion de validité d'un bloc : étant donné un bloc b et un état mémoire M, on note $M \models b$ la proposition « b est valide dans M ». Celle-ci peut être comprise intuitivement en considérant M comme le résultat d'une succession d'opérations (écritures, allocations, déallocations) depuis un état initial : on dira alors que b est valide dans M lorsque, parmi les opérations ayant abouti à M, figure une allocation dont le résultat comprend b, et aucune déallocation de b depuis cette allocation. La négation de cette proposition est notée $M \not\models b$.

Les axiomes 2.1 et 2.2 spécifient le comportement de l'allocation, qui est de générer un nouveau bloc valide en laissant inchangés les blocs déjà valides. L'axiome 2.1 définit une exigence de « fraîcheur » pour les blocs issus d'une allocation : celle-ci ne peut fournir un bloc *déjà valide* dans l'état courant. Hormis le nouveau bloc, l'allocation préserve la validité (et l'invalidité) des blocs existants : cette invariance est exprimée par l'axiome 2.2.

Axiome 2.1 (invalidité avant allocation)**.**
Si $\mathsf{alloc}(M_1, n) = (b, M_2)$, alors $M_1 \not\models b$.

Axiome 2.2 (validité après allocation)**.**
Si $\mathsf{alloc}(M_1, n) = (b, M_2)$, alors $M_2 \models b' \iff b' = b \vee M_1 \models b'$.

La déallocation, en tant que complémentaire de l'allocation, a pour effet de rendre invalide un bloc, la validité des autres étant inchangée. L'axiome 2.4 assure ainsi qu'un bloc est invalide dans l'état résultant de sa déallocation, tandis que l'axiome 2.3 garantit que les blocs différents de celui déalloué conservent leur validité.

Axiome 2.3 (validité après déallocation)**.**
Si $\mathsf{free}(M_1, b) = \lfloor M_2 \rfloor$ et $b' \neq b$, alors $M_2 \models b' \iff M_1 \models b'$.

Axiome 2.4 (invalidité après déallocation)**.**
Si $\mathsf{free}(M_1, b) = \lfloor M_2 \rfloor$, alors $M_2 \not\models b$.

Par ailleurs, tout bloc valide doit pouvoir être déalloué, c'est-à-dire que sa déallocation ne doit pas échouer mais avoir pour résultat un nouvel état mémoire. Réciproquement, une déallocation ne réussit que si elle est effectuée sur un bloc valide. Ces propriétés sont exprimées par l'axiome 2.5.

Axiome 2.5 (valide équivaut à déallouable)**.**
$M_1 \models b$ si, et seulement si, $\exists M_2, \mathsf{free}(M_1, b) = \lfloor M_2 \rfloor$.

Enfin, l'axiome 2.6 traduit le fait que seules les allocations et déallocations affectent la validité des blocs, qui est insensible aux écritures.

Axiome 2.6 (validité après écriture)**.**
Si $\mathsf{store}(\kappa, M_1, b, \delta, v) = \lfloor M_2 \rfloor$, alors $M_2 \vDash b' \iff M_1 \vDash b'$.

Longueur des blocs La deuxième série d'axiomes décrit le comportement de la fonction length, qui donne la taille d'un bloc. L'axiomatisation a essentiellement pour rôle d'assurer que la taille d'un bloc est déterminée à l'allocation, et reste ensuite inchangée au travers des autres opérations.

La taille d'un bloc venant d'être alloué est définie par l'axiome 2.7 comme celle donnée en argument de l'allocation. Pour un bloc donné, allouer un nouveau bloc (axiome 2.8), effectuer une écriture (axiome 2.10), ou déallouer un autre bloc (axiome 2.9) est sans effet sur sa taille.

Axiome 2.7 (longueur après allocation – même bloc)**.**
Si $\mathsf{alloc}(M_1, n) = (b, M_2)$, alors $\mathsf{length}(M_2, b) = n$.

Axiome 2.8 (longueur après allocation – autre bloc)**.**
Si $\mathsf{alloc}(M_1, n) = (b, M_2)$ et $b' \neq b$, alors $\mathsf{length}(M_2, b') = \mathsf{length}(M_1, b')$.

Axiome 2.9 (longueur après déallocation)**.**
Si $\mathsf{free}(M_1, b) = \lfloor M_2 \rfloor$ et $b' \neq b$, alors $\mathsf{length}(M_2, b') = \mathsf{length}(M_1, b')$.

Axiome 2.10 (longueur après écriture)**.**
Si $\mathsf{store}(\kappa, M_1, b, \delta, v) = \lfloor M_2 \rfloor$, alors $\mathsf{length}(M_2, b') = \mathsf{length}(M_1, b')$.

Lecture La troisième série d'axiomes décrit l'effet sur le contenu de la mémoire, observable via l'opération load. Les opérations de lecture et d'écriture sont conditionnées par un critère dit d'*accès valide*, défini ci-après. Un accès (en lecture ou en écriture) à un emplacement est valide lorsque le bloc de cet emplacement est valide, et la plage d'indices concernée par l'écriture est comprise dans les bornes du bloc (bornes qui sont 0 inclus et longueur du bloc exclue).

Définition 1 (accès valide)**.**
On dit que l'accès à l'emplacement (b, δ) avec le type κ est valide dans l'état mémoire M, et on note $M \vDash \kappa @ b, \delta$, si :

$$M \vDash b \quad \text{et} \quad \delta \geq 0 \quad \text{et} \quad \delta + \mathsf{sizeof}(\kappa) \leq \mathsf{length}(M, b).$$

Propriété 2.1 (accès valide après allocation)**.**
L'allocation d'un bloc n'affecte pas la validité des accès à d'autres blocs que celui-ci :

$$\mathsf{alloc}(M_1, n) = (b, M_2) \ \wedge \ b' \neq b \implies M_1 \vDash \kappa @ b', \delta \iff M_2 \vDash \kappa @ b', \delta.$$

Démonstration. Il s'agit d'une conséquence des axiomes 2.2 et 2.8. □

Propriété 2.2 (accès valide après déallocation)**.**
La déallocation d'un bloc n'affecte pas la validité des accès à d'autres blocs que celui-ci :

$$\mathsf{free}(M_1, b) = \lfloor M_2 \rfloor \ \wedge \ b' \neq b \implies M_1 \vDash \kappa @ b', \delta \iff M_2 \vDash \kappa @ b', \delta.$$

Démonstration. Il s'agit d'une conséquence des axiomes 2.3 et 2.9. □

Propriété 2.3 (accès valide après écriture)**.**
La validité des accès est invariante par écriture :

$$\mathsf{store}(\kappa, M_1, b, \delta, v) = \lfloor M_2 \rfloor \implies M_1 \vDash \kappa' @ b', \delta' \iff M_2 \vDash \kappa' @ b', \delta'.$$

Démonstration. Il s'agit d'une conséquence des axiomes 2.6 et 2.10. □

L'axiome 2.11 spécifie le caractère indéfini des données d'un bloc nouvellement alloué. L'axiome 2.12 exprime l'invariance des opérations de lecture vis-à-vis des allocations, dès lors que la lecture n'a pas lieu dans le bloc nouvellement alloué. De la même façon, l'axiome 2.13 exprime l'invariance des lectures vis-à-vis des déallocations, tant que ces lectures ont lieu en dehors du bloc déalloué.

Axiome 2.11 (lecture après allocation – même bloc)**.**
Si $\mathsf{alloc}(M_1, n) = (b, M_2)$ et $M_2 \vDash \kappa \,@\, b, \delta$, alors $\mathsf{load}(\kappa, M_2, b, \delta) = \lfloor \mathsf{Undef} \rfloor$.

Axiome 2.12 (lecture après allocation – autre bloc)**.**
Si $\mathsf{alloc}(M_1, n) = (b, M_2)$ et $b \neq b'$, alors $\mathsf{load}(\kappa, M_2, b', \delta) = \mathsf{load}(\kappa, M_1, b', \delta)$.

Axiome 2.13 (lecture après déallocation)**.**
Si $\mathsf{free}(M_1, b) = \lfloor M_2 \rfloor$ et $b \neq b'$, alors $\mathsf{load}(\kappa, M_2, b', \delta) = \mathsf{load}(\kappa, M_1, b', \delta)$.

Trois axiomes sont nécessaires pour spécifier les interactions entre lecture et écriture, correspondant aux trois configurations relatives possibles des deux zones mémoire concernées : identiques, disjointes, ou partiellement superposées.

Si lecture et écriture ont lieu au même emplacement, (axiome 2.14), c'est-à-dire ont lieu dans le même bloc, avec le même décalage, et un type de même longueur, alors la valeur résultant de la lecture est celle qui a été écrite, sous réserve de certaines contraintes de compatibilités. Ces contraintes, données par les définitions 2 et 3, concernent la valeur écrite, le type avec lequel elle a été écrite, et celui avec lequel elle est lue. Si la lecture a lieu sur une zone disjointe de celle couverte par l'écriture (axiome 2.15), alors le résultat est le même que dans l'état mémoire précédant l'écriture. Enfin, si les zones se recouvrent partiellement (axiome 2.16), la lecture renvoie une valeur indéfinie.

Définition 2 (représentabilité)**.**
La représentabilité d'une valeur sur un type est définie par cas :

- une valeur $\mathsf{Int}(n)$ est représentable par le type $\mathrm{i}\,k$, où $k \in \{8, 16, 32, 64\}$, si l'entier n vérifie $-2^{k-1} \leq n \leq 2^{k-1} - 1$
- une valeur $\mathsf{Ptr}(b, \delta)$ est représentable sur le type $\mathrm{i}\,w$ uniquement, où w est la taille d'un mot machine.
- la valeur Undef n'est représentable sur aucun type.

Nous avons adopté ici une notion restrictive de compatibilité entre types, qui ne permet de lire une valeur qu'avec le type utilisé lors de son écriture (dans le cas contraire, la valeur lue est indéfinie). En cela, nous avons dévié du traitement des types dans le modèle mémoire de CompCert [LB08], où la fonction convert assure la conversion entre entiers (selon leur taille et leur caractère signé ou non), ainsi que la conversion entre nombres à virgule flottante (selon leur taille). Dans notre système de types simplifié, convert n'opére plus vraiment de conversion, et se borne à contrôler l'égalité des types employés pour la lecture et l'écriture d'une valeur, renvoyant Undef le cas échéant.

Définition 3 (conversion de type)**.**
La conversion d'une valeur v du type κ au type κ' est définie par :

$$\mathsf{convert}(v, \kappa, \kappa') = \begin{cases} \mathsf{Undef} \text{ si } \kappa \neq \kappa' \text{ ou si } v \text{ n'est pas représentable sur } \kappa \\ v \text{ si } \kappa = \kappa' \text{ et } v \text{ est représentable sur } \kappa. \end{cases}$$

Axiome 2.14 (lecture après écriture – même emplacement)**.**
Si $\mathsf{store}(\kappa, M_1, b, \delta, v) = \lfloor M_2 \rfloor$ et $M_2 \vDash \kappa' \,@\, b, \delta$, alors $\mathsf{load}(\kappa', M_2, b, \delta) = \lfloor \mathsf{convert}(v, \kappa, \kappa') \rfloor$.

Axiome 2.15 (lecture après écriture – zones disjointes)**.**
Si $\mathsf{store}(\kappa, M_1, b, \delta, v) = \lfloor M_2 \rfloor$ et $b' \neq b \ \vee \ \delta + \mathsf{sizeof}(\kappa) \leq \delta' \ \vee \ \delta' + \mathsf{sizeof}(\kappa') \leq \delta$, alors $\mathsf{load}(\kappa', M_2, b', \delta') = \mathsf{load}(\kappa', M_1, b', \delta')$.

Axiome 2.16 (lecture après écriture – superposition partielle)**.**
Si $\mathsf{store}(\kappa, M_1, b, \delta, v) = \lfloor M_2 \rfloor$, et si en outre les conditions suivantes sont vérifiées :

$$\begin{cases} \delta \neq \delta' \\ \delta + \mathsf{sizeof}(\kappa) > \delta' \\ \delta' + \mathsf{sizeof}(\kappa') > \delta \\ \delta' \geq 0 \\ \delta' + \mathsf{sizeof}(\kappa') \leq \mathsf{length}(M_2, b) \end{cases}$$

alors $\mathsf{load}(\kappa', M_2, b, \delta') = \lfloor \mathsf{Undef} \rfloor$.

Axiomes complémentaires Les axiomes introduits jusqu'ici décrivent les propriétés d'états mémoires résultant d'une écriture, ou des valeurs résultant d'une lecture, mais ne donnent pas de conditions suffisantes pour pouvoir effectuer ces opérations sans erreur. Les deux axiomes suivants définissent l'accès valide comme condition suffisante à la fois pour écrire dans un emplacement en mémoire, ou pour le lire.

Axiome 2.17 (accès valide et écriture)**.**
L'accès valide est une condition nécessaire et suffisante pour écrire à un emplacement donné :

$$M_1 \vDash \kappa \mathbin{@} b, \delta \iff \exists M_2, \ \mathsf{store}(\kappa, M_1, b, \delta, v) = \lfloor M_2 \rfloor.$$

Axiome 2.18 (accès valide et lecture)**.**
L'accès valide est une condition nécessaire et suffisante pour lire à un emplacement donné :

$$M \vDash \kappa \mathbin{@} b, \delta \iff \exists v, \ \mathsf{load}(\kappa, M, b, \delta) = \lfloor v \rfloor.$$

Le dernier axiome du modèle définit le caractère nouveau des blocs issus de l'allocation, en s'appuyant sur la notion de *support*. Le support d'un état mémoire correspond à l'ensemble des blocs ayant un rôle à jouer dans cet état, soit parce qu'ils sont utilisés pour enregistrer des données (cas des blocs valides), soit parce qu'il existe en mémoire des pointeurs qui les référencent.

En spécifiant qu'un bloc nouvellement alloué ne peut pas être compris dans le support de l'état précédant l'allocation, l'axiome 2.19 empêche qu'une allocation rende à nouveau valide un pointeur dit *pendant* (c'est-à-dire pointant vers un bloc déalloué).

Définition 4 (support)**.**
On appelle support de M, et on note $\mathsf{supp}(M)$, l'ensemble des blocs qui sont soit valides dans M, soit référencés par un pointeur présent dans M :

$$\begin{aligned} \mathsf{supp}(M) \triangleq \ & \{b \in \mathbf{block} \mid M \vDash b\} \\ & \cup \{b \in \mathbf{block} \mid \exists \kappa, b', \delta', \delta, \mathsf{load}(\kappa, M, b', \delta') = \lfloor \mathsf{Ptr}(b, \delta) \rfloor\}. \end{aligned}$$

Axiome 2.19 (fraîcheur des blocs alloués)**.**
Si $\mathsf{alloc}(M_1, n) = (b, M_2)$, alors $b \notin \mathsf{supp}(M_1)$.

2.3 Sémantique de référence

Nous pouvons désormais munir notre langage d'une sémantique, ce qui nous permettra par la suite de raisonner sur des exécutions de programmes. Parmi les différents styles de sémantique possibles [NN07], nous avons retenu pour ce langage une sémantique opérationnelle

à grand pas [Kah87]. Elle est adaptée, comme le modèle mémoire, d'une sémantique élaborée pour CompCert [BL09].

2.3.1 Présentation d'ensemble

Notre sémantique est constituée d'un ensemble de relations liant des objets syntaxiques du programme (expressions, instructions, termes, prédicats…), le *contexte* dans lequel ils sont évalués, et le résultat de leur évaluation dans ce contexte. Dans notre langage source, le contexte est un couple formé d'un état mémoire et d'un *environnement* (généralement noté E) associant à chaque variable de la portée lexicale courante le bloc de mémoire qu'elle désigne. Formellement, un tel environnement aura le type **map ident block**, où **map** est défini comme un type dictionnaire avec valeur de retour optionnelle (c'est-à-dire que le résultat de la recherche de la clé k est $\lfloor v \rfloor$ s'il existe une clé v est associée à k, et ε sinon).

Nous définissons une relation d'évaluation pour chaque catégorie syntaxique de notre langage, sauf pour les expressions qui en nécessitent deux. Cette nécessité vient de ce que certaines expressions peuvent désigner deux concepts différents selon la position à laquelle elles figurent dans le programme.

Si l'on considère par exemple $x = x + 1;$, soit l'incrémentation de la variable x, cette dernière ne joue pas le même rôle selon qu'elle figure à gauche du symbole d'affectation ou à droite : le x de droite désigne la *valeur* des données correspondant à la variable, utilisée pour calculer arithmétiquement l'incrément, tandis que celui de gauche désigne l'*emplacement* en mémoire de ces données, auquel sera écrit le résultat du calcul de l'incrément. C'est pourquoi on parlera par la suite d'évaluation en tant que *valeur gauche* lorsqu'une expression est évaluée pour obtenir un emplacement en mémoire.

Au total, il y a donc cinq « modes » d'évaluation dans notre langage, engendrant cinq relations que nous notons sous la forme générale :

$$\text{contexte} \vDash_{\text{mode}} \text{objet syntaxique} \Downarrow \text{résultat}.$$

Le contexte d'évaluation est toujours un couple formé d'un environnement et d'une mémoire, tandis que les types de l'objet syntaxique évalué et du résultat de l'évaluation sont déterminés par le mode d'évaluation suivant le tableau de la figure 2.11 (où **expr**, **term**, **pred** et **stmt** sont respectivement les types des expressions, des termes, des prédicats et des instructions).

mode	type de l'objet évalué	type du résultat
expression	**expr**	**value**
valeur gauche	**expr**	**block** $\times \mathbb{Z}$
terme	**term**	**value**
prédicat	**pred**	**bool**
instruction	**stmt**	**mem**

FIGURE 2.11 – Modes d'évaluation de la sémantique source.

Chaque relation d'évaluation est donnée sous la forme d'un ensemble de règles qui, interprétées inductivement, définissent l'ensemble des jugements d'évaluation valides (c'est-à-dire la relation d'évaluation elle-même). Un programme peut être évalué dans un contexte donné s'il est possible de construire un jugement d'évaluation pour ce programme (dans ce contexte) à partir des règles sémantiques ; dans le cas contraire, le programme est considéré comme n'ayant pas de sémantique dans notre langage. Autrement dit, une erreur à l'exécution d'un programme se traduit dans notre sémantique par l'absence de jugement d'évaluation pour ce

programme. En particulier, les erreurs ne font pas l'objet de règles ou plus généralement d'un traitement spécifique (nous reviendrons sur ce choix en section 2.3).

La suite de cette section est consacrée à la présentation des différentes relations d'évaluation.

2.3.2 Évaluation des expressions

L'évaluation d'une expression e en une valeur v dans un contexte formé d'un environnement E et d'une mémoire M est notée $E, M \vDash_{\mathrm{e}} e \Downarrow v$. Son évaluation dans le même contexte mais en tant que valeur gauche, résultant en un emplacement mémoire (b, δ), est quant à elle notée $E, M \vDash_{\mathrm{lv}} e \Downarrow b, \delta$. Ces deux relations sont définies par induction mutuelle par le système de règles de la figure 2.12, que nous détaillons à présent.

Une constante entière est directement évaluée en la valeur correspondante (règle E-INT).

L'évaluation des variables en valeur gauche est définie par la règle LV-VAR : dans notre langage, chaque variable est associée, via l'environnement, à un bloc de mémoire qui lui est propre. Les données qu'elle désigne résident dans ce bloc à l'indice 0.

La seconde règle d'évaluation en valeur gauche est le déréférencement de pointeur (règle E-DEREF) qui, étant donné une expression évaluée en un pointeur, permet d'obtenir l'emplacement mémoire désigné par ce pointeur.

Inversement, la prise d'adresse (règle E-ADDR) d'une valeur gauche construit un pointeur vers l'emplacement de celle-ci.

La règle E-LVAL est d'un intérêt particulier, en ce qu'elle illustre l'application sémantique du modèle mémoire décrit dans la section précédente. Étant donné une expression e évaluée comme valeur gauche en un emplacement (b, δ), on effectue une lecture en mémoire à l'emplacement en question : si la lecture réussit, renvoyant une valeur v, *et que cette valeur n'est pas indéfinie*, alors e est évaluée correctement en v. Le type mémoire utilisé pour la lecture est obtenu à partir du type de e, via la fonction mtype, qui associe à un type du langage le type mémoire correspondant. Dans notre système de types rudimentaire, elle peut être définie comme suit :

$$\begin{array}{lcll} \mathsf{mtype}(\mathtt{int}k) & = & \mathtt{i}k & \text{pour } k \in \{8, 16, 32, 64\} \\ \mathsf{mtype}(*\tau) & = & \mathtt{i}w & \end{array}$$

où w est la taille d'un mot machine.

Concernant les opérateurs, unaires et binaires, nous ne nous intéressons pas ici à leur définition précise. C'est pourquoi les règles E-UNOP et E-BINOP sont simplement paramétrées par les fonctions $\mathsf{sem\_unop}$ et $\mathsf{sem\_binop}$, que nous laissons indéfinies. On demandera seulement à ce que $\mathsf{sem\_binop}$ gère le cas de l'addition d'un entier à un pointeur, de façon à pouvoir utiliser dans notre langage de l'arithmétique de pointeurs. On aura ainsi :

$$\mathsf{sem\_binop}(+, \mathsf{Ptr}(b, \delta) : \tau*, \mathsf{Int}(n)) \triangleq \mathsf{Ptr}(b, \delta + n * \mathsf{sizeof}(\mathsf{mtype}(\tau))).$$

On peut noter que l'évaluation d'une expression a lieu à contexte constant, c'est-à-dire que le même environnement et la même mémoire servent à évaluer toutes les sous-expressions d'une expression donnée. Cette invariance est due au fait que nos expressions n'ont pas de mécanisme d'introduction de variables locales – notre langage possède un `let` mais il s'agit d'une instruction – ni d'effets de bord.

2.3.3 Évaluation des termes

Les termes étant une généralisation des expressions, leur sémantique est très semblable à celle de ces dernières. Ainsi l'évaluation d'une expression en tant que terme (règle T-EXPR) a simplement pour résultat la valeur de l'expression.

$$\text{E-INT}\ \frac{}{E, M \vDash_{\mathrm{e}} n \Downarrow \mathsf{Int}(n)} \qquad \text{LV-VAR}\ \frac{E(x) = b}{E, M \vDash_{\mathrm{lv}} x \Downarrow b, 0} \qquad \text{LV-DEREF}\ \frac{E, M \vDash_{\mathrm{e}} e \Downarrow \mathsf{Ptr}(b, \delta)}{E, M \vDash_{\mathrm{lv}} *e \Downarrow b, \delta} \qquad \text{E-ADDR}\ \frac{E, M \vDash_{\mathrm{lv}} e \Downarrow b, \delta}{E, M \vDash_{\mathrm{e}} \&e \Downarrow \mathsf{Ptr}(b, \delta)}$$

$$\text{E-LVAL}\ \frac{E, M \vDash_{\mathrm{lv}} e : \tau \Downarrow b, \delta \qquad \mathsf{load}(\mathsf{mtype}(\tau), M, b, \delta) = \lfloor v \rfloor \qquad v \neq \mathsf{Undef}}{E, M \vDash_{\mathrm{e}} e \Downarrow v}$$

$$\text{E-UNOP}\ \frac{E, M \vDash_{\mathrm{e}} e \Downarrow v_1 \qquad \mathsf{sem\_unop}(\dagger, v_1) = \lfloor v_2 \rfloor}{E, M \vDash_{\mathrm{e}} \dagger e \Downarrow v_2}$$

$$\text{E-BINOP}\ \frac{E, M \vDash_{\mathrm{e}} e_1 \Downarrow v_1 \qquad E, M \vDash_{\mathrm{e}} e_2 \Downarrow v_2 \qquad \mathsf{sem\_binop}(\ddagger, v_1, v_2) = \lfloor v_3 \rfloor}{E, M \vDash_{\mathrm{e}} e_1 \ \ddagger\ e_2 \Downarrow v_3}$$

FIGURE 2.12 – Évaluation des expressions.

$$\text{T-BASEADDR}\ \frac{E, M \vDash_{\mathrm{t}} t \Downarrow \mathsf{Ptr}(b, \delta)}{E, M \vDash_{\mathrm{t}} \backslash\texttt{base\_address}(t) \Downarrow \mathsf{Ptr}(b, 0)} \qquad \text{T-OFS}\ \frac{E, M \vDash_{\mathrm{t}} t \Downarrow \mathsf{Ptr}(b, \delta)}{E, M \vDash_{\mathrm{t}} \backslash\texttt{offset}(t) \Downarrow \mathsf{Int}(\delta)}$$

$$\text{T-BLOCKLEN}\ \frac{E, M \vDash_{\mathrm{t}} t \Downarrow \mathsf{Ptr}(b, \delta) \qquad \mathsf{length}(M, b) = n}{E, M \vDash_{\mathrm{t}} \backslash\texttt{block\_length}(t) \Downarrow \mathsf{Int}(n)} \qquad \text{T-EXPR}\ \frac{E, M \vDash_{\mathrm{e}} e \Downarrow v}{E, M \vDash_{\mathrm{t}} e \Downarrow v}$$

$$\text{T-DEREF}\ \frac{E, M \vDash_{\mathrm{t}} t : \tau* \Downarrow \mathsf{Ptr}(b, \delta) \qquad \mathsf{load}(\mathsf{mtype}(\tau), M, b, \delta) = \lfloor v \rfloor \qquad v \neq \mathsf{Undef}}{E, M \vDash_{\mathrm{t}} \bar{*}t \Downarrow v}$$

$$\text{T-UNOP}\ \frac{E, M \vDash_{\mathrm{t}} t \Downarrow v_1 \qquad \mathsf{sem\_unop}(\bar{\dagger}, v_1) = \lfloor v_2 \rfloor}{E, M \vDash_{\mathrm{t}} \bar{\dagger}\, t \Downarrow v_2}$$

$$\text{T-BINOP}\ \frac{E, M \vDash_{\mathrm{t}} t_1 \Downarrow v_1 \qquad E, M \vDash_{\mathrm{t}} t_2 \Downarrow v_2 \qquad \mathsf{sem\_binop}(\bar{\ddagger}, v_1, v_2) = \lfloor v_3 \rfloor}{E, M \vDash_{\mathrm{t}} t_1 \,\bar{\ddagger}\, t_1 \Downarrow v_3}$$

FIGURE 2.13 – Évaluation des termes.

Comme la sémantique des opérateurs unaires et binaires est définie sur les valeurs, et que les termes sont évalués en valeurs, on peut définir les règles T-UNOP et T-BINOP à partir des mêmes fonctions sémantiques sem_unop et sem_binop que pour les expressions. Ce traitement relativement simple des opérateurs est permis par le choix d'utiliser les mêmes valeurs dans le langage de programmation et dans le langage de spécification. Si nous avons fait ce choix dans un souci de simplicité, il est généralement appréciable de disposer dans les annotations logiques d'autres types de valeurs, notamment des entiers non bornés. E-ACSL possède de tels

entiers, mais leur traitement est non-trivial et largement orthogonal au sujet des propriétés mémoire, qui nous préoccupe ici.

Le déréférencement nécessite une construction plus spécifique, puisque celui des expressions utilise l'évaluation en valeur gauche, qui n'a pas équivalent sur les termes. La règle T-DEREF combine ainsi l'évaluation d'un terme en un pointeur, et la lecture à l'emplacement désigné par ce pointeur.

Prendre l'adresse de base d'un pointeur (règle T-BASEADDR) revient simplement à mettre à zéro la composante décalage de ce pointeur, tandis qu'au contraire on obtient le décalage (règle T-OFFSET) en « oubliant » la composante relative au bloc. Quant à la longueur d'un bloc (règle T-DEREF), elle est obtenue par un appel à l'opération correspondante du modèle mémoire.

2.3.4 Évaluation des prédicats

Les prédicats sont chacun évalués en une valeur de vérité représentée ici par une valeur booléenne : $\top$ (vrai) ou $\bot$ (faux). Ceci est rendu possible par le caractère *exécutable* des spécifications E-ACSL, et par le choix de ne pas couvrir dans notre formalisation le cas des erreurs à l'exécution, et notamment celles introduites par les annotations.

$$\text{P-VALID}\quad \frac{E, M \vDash_{\mathrm{t}} t : \tau* \Downarrow \mathsf{Ptr}(b,\delta) \qquad M \vDash \mathsf{mtype}(\tau)\,@\,b,\delta}{E, M \vDash_{\mathrm{p}} \backslash\mathtt{valid}(t) \Downarrow \top}$$

$$\text{P-INVALID}\quad \frac{E, M \vDash_{\mathrm{t}} t : \tau* \Downarrow \mathsf{Ptr}(b,\delta) \qquad M \nvDash \mathsf{mtype}(\tau)\,@\,b,\delta}{E, M \vDash_{\mathrm{p}} \backslash\mathtt{valid}(t) \Downarrow \bot}$$

$$\text{P-INITIALIZED}\quad \frac{E, M \vDash_{\mathrm{t}} t : \tau* \Downarrow \mathsf{Ptr}(b,\delta) \qquad \mathsf{load}(\mathsf{mtype}(\tau), M, b, \delta) = \lfloor v \rfloor \qquad v \neq \mathsf{Undef}}{E, M \vDash_{\mathrm{p}} \backslash\mathtt{initialized}(t) \Downarrow \top}$$

$$\text{P-UNINITIALIZED}\quad \frac{E, M \vDash_{\mathrm{t}} t : \tau* \Downarrow \mathsf{Ptr}(b,\delta) \qquad \mathsf{load}(\mathsf{mtype}(\tau), M, b, \delta) = \lfloor \mathsf{Undef} \rfloor}{E, M \vDash_{\mathrm{p}} \backslash\mathtt{initialized}(t) \Downarrow \bot}$$

$$\text{P-CMP}\quad \frac{E, M \vDash_{\mathrm{t}} t_1 \Downarrow v_1 \qquad E, M \vDash_{\mathrm{t}} t_2 \Downarrow v_2 \qquad \mathsf{sem\_cmp}(\bowtie, v_1, v_2) = \beta}{E, M \vDash_{\mathrm{p}} t_1 \bowtie t_2 \Downarrow \beta}$$

FIGURE 2.14 – Évaluation des prédicats élémentaires.

Les règles régissant l'évaluation des prédicats élémentaires, c'est-à-dire formés à partir de termes, sont présentés dans la figure 2.14. L'évaluation de la validité d'un pointeur (règles P-VALID et P-INVALID) repose sur la primitive correspondante du modèle mémoire. L'initialisation des données référencées par un pointeur (règles P-INITIALIZED et P-UNINITIALIZED) est définie par le fait que la lecture des données en mémoire est possible et que la valeur résultante est différente de Undef (valeur qui caractérise les données non initialisées). Les autres prédicats élémentaires sont les comparaisons de termes. Les entiers peuvent faire l'objet des comparaisons usuelles, tandis que pour les pointeurs seule l'égalité peut (dans notre langage) être testée.

La sémantique des prédicats restants est constituée des règles d'évaluation des connecteurs logiques, exposées dans la figue 2.15. Elles forment un calcul propositionnel dont la seule particularité est le caractère *paresseux* de la conjonction : lors de l'évaluation d'une conjonction, la fausseté du membre gauche implique celle conjonction dans son ensemble, quelle que soit la valeur du membre droit ; dans ce cas, ce dernier n'est pas évalué. La règle P-CONJ-LEFT traduit

$$\text{P-CONJ-LEFT}\ \frac{E, M \vDash_{\mathrm{p}} p_1 \Downarrow \bot}{E, M \vDash_{\mathrm{p}} p_1 \wedge p_2 \Downarrow \bot} \qquad \text{P-CONJ-RIGHT}\ \frac{E, M \vDash_{\mathrm{p}} p_1 \Downarrow \top \qquad E, M \vDash_{\mathrm{p}} p_2 \Downarrow \beta}{E, M \vDash_{\mathrm{p}} p_1 \wedge p_2 \Downarrow \beta}$$

$$\text{P-NEG-FALSE}\ \frac{E, M \vDash_{\mathrm{p}} p \Downarrow \top}{E, M \vDash_{\mathrm{p}} \neg p \Downarrow \bot} \qquad \text{P-NEG-TRUE}\ \frac{E, M \vDash_{\mathrm{p}} p \Downarrow \bot}{E, M \vDash_{\mathrm{p}} \neg p \Downarrow \top}$$

FIGURE 2.15 – Évaluation des prédicats composés.

ce comportement : dans le cas d'une conjonction $p_1 \wedge p_2$, si p_1 est faux alors on peut conclure à la fausseté de $p_1 \wedge p_2$ sans qu'il soit besoin d'évaluer p_2. Si p_1 est vrai en revanche (règle P-CONJ-RIGHT), l'évaluation de p_2 en une valeur de vérité V est requise et donne le résultat de la conjonction.

Nous avons choisi d'inclure ce trait sémantique dans notre formalisation principalement par cohérence avec la logique d'E-ACSL, où il joue un rôle important en raison de la possibilité pour certains prédicats d'être indéfinis [DKS13]. La paresse n'a pas cette importance dans notre cadre, puisque nous ne nous préoccupons pas des prédicats indéfinis, mais elle offre l'avantage de la cohérence du modèle avec E-ACSL, sans présenter d'inconvénient particulier.

Par ailleurs, de même que pour les expressions et les termes, les prédicats sont évalués à environnement et mémoire constants, comme on est en droit de l'attendre étant donné leur rôle de spécification : il semble cohérent que l'évaluation d'une propriété du programme sous observation n'altère pas son contexte d'exécution.

2.3.5 Évaluation des instructions

Si l'évaluation d'une expression ou d'un terme a pour résultat une valeur du langage, et celle d'un prédicat une valeur de vérité, les instructions quant à elles ont comme résultat d'évaluation un état mémoire, possiblement différent de celui dans lequel elles ont été initialement évaluées.

Tout comme les prédicats, les instructions peuvent être séparées en instructions élémentaires (correspondant aux productions non récursives de la grammaire des instructions) et instructions composées (correspondant aux productions récursives).

Instructions élémentaires. Outre l'instruction `skip` (règle S-SKIP), dont le résultat est simplement l'état mémoire initial dans lequel l'instruction est exécutée, on trouve dans la première catégorie les instructions modifiant la mémoire (affectation, allocation, déallocation) ainsi que l'assertion d'un prédicat. Leur évaluation est décrite dans la figure 2.16.

L'affectation (règle S-ASSIGN) a pour effet d'écrire en mémoire la valeur du membre droit à l'emplacement désigné par le membre gauche ; l'état mémoire obtenu constitue le résultat de l'évaluation.

Dans notre langage, l'allocation est la contraction de deux opérations mémoire, illustrée par la règle S-MALLOC : un nouveau bloc de la taille demandée est d'abord alloué, ce qui porte la mémoire dans un état intermédiaire (M_2 dans la figure) ; après quoi, un pointeur vers la tête (c'est-à-dire le premier octet) du nouveau bloc est écrit à l'emplacement défini par le membre gauche de l'allocation.

$$\text{S-SKIP}\quad \frac{}{E, M \vDash_{\mathrm{s}} \texttt{skip};\Downarrow M}$$

$$\text{S-ASSIGN}\quad \frac{E, M_1 \vDash_{\mathrm{e}} e_2 : \tau \Downarrow v \qquad E, M_1 \vDash_{\mathrm{lv}} e_1 : \tau \Downarrow b, \delta \qquad \mathsf{store}(\mathsf{mtype}(\tau), M_1, b, \delta, v) = \lfloor M_2 \rfloor}{E, M_1 \vDash_{\mathrm{s}} e_1 = e_2;\Downarrow M_2}$$

$$\text{S-MALLOC}\quad \frac{E, M_1 \vDash_{\mathrm{e}} e_2 \Downarrow \mathsf{Int}(n) \qquad \mathsf{alloc}(M_1, n) = (b', M_2) \qquad E, M_1 \vDash_{\mathrm{lv}} e_1 : \tau* \Downarrow b, \delta \qquad \mathsf{store}(\mathsf{mtype}(\tau*), M_2, b, \delta, \mathsf{Ptr}(b', 0)) = \lfloor M_3 \rfloor}{E, M_1 \vDash_{\mathrm{s}} e_1 = \texttt{malloc}(e_2);\Downarrow M_3}$$

$$\text{S-FREE}\quad \frac{E, M_1 \vDash_{\mathrm{e}} e \Downarrow \mathsf{Ptr}(b, 0) \qquad b \notin \mathsf{Image}(E) \qquad \mathsf{free}(M_1, b) = \lfloor M_2 \rfloor}{E, M_1 \vDash_{\mathrm{s}} \texttt{free}(e);\Downarrow M_2}$$

$$\text{S-LOGICAL-ASSERT}\quad \frac{E, M \vDash_{\mathrm{p}} p \Downarrow \top}{E, M \vDash_{\mathrm{s}} \texttt{logical\_assert}(p);\Downarrow M}$$

FIGURE 2.16 – Évaluation des instructions élémentaires.

L'instruction `free`, quant à elle, prend en argument un pointeur vers la tête d'un bloc, et procède à sa déallocation par l'opération mémoire free correspondant (S-FREE) ; à noter que l'instruction n'opère pas sur les blocs de mémoire associés aux variables de l'environnement E, c'est-à-dire ceux appartenant à l'ensemble $\mathsf{Image}(E)$.

Enfin, la vérification d'assertion (règle S-LOGICAL-ASSERT) se résume à l'évaluation du prédicat à vérifier : l'exécution du programme ne se poursuit que si celui-ci est évalué à $\top$, autrement le programme n'a pas de sémantique (on parle de *sémantique bloquante* [CS12]).

Instructions composées. Les instructions élémentaires peuvent être combinées à l'aide des instructions composées, dont la sémantique est donnée par la figure 2.17.

La séquence (règle S-SEQ) évalue une première instruction, et utilise l'état mémoire résultant (M_2 dans la figure) dans le contexte d'évaluation de la seconde instruction.

Le branchement conditionnel consiste à évaluer une expression en tant que condition – c'est-à-dire à considérer la condition valide si l'expression est évaluée à une valeur entière non nulle, et fausse si elle est évaluée à l'entier zéro – et à évaluer la première branche (règle S-IF-TRUE) ou la seconde (règle S-IF-FALSE) selon le cas.

L'évaluation en boucle d'une instruction peut se dérouler de deux façons. Soit la condition de la boucle n'est pas satisfaite (règle S-WHILE-FALSE), auquel cas l'évaluation est sans effet ; soit, dans le cas contraire, le corps de la boucle est évalué une fois, produisant un état mémoire intermédiaire, puis la boucle elle-même est à nouveau évaluée depuis cet état (règle S-WHILE-TRUE). Notons que l'interprétation inductive de cette règle implique que la sémantique ne couvre que les exécutions qui terminent : les boucles infinies ne sont donc pas définies.

Enfin, l'introduction d'une variable locale (instruction `let`) alloue une variable en mémoire, et lie le nom de la variable au bloc alloué dans l'environnement (ces deux opérations sont représentées par la fonction alloc_var, définie ci-après, dans la règle S-LET). Il en résulte un nouveau contexte (E_2, M_2 dans la figure) utilisé pour évalué le corps de l'instruction. À l'issue de cette évaluation, le bloc correspondant à la variable locale est déalloué, et l'on obtient l'état mémoire final de l'évaluation. Similairement à l'allocation de variable, la déallocation est effec-

$$\text{S-SEQ}\ \frac{E, M_1 \vDash_{\mathrm{s}} s_1 \Downarrow M_2 \qquad E, M_2 \vDash_{\mathrm{s}} s_2 \Downarrow M_3}{E, M_1 \vDash_{\mathrm{s}} s_1\ s_2 \Downarrow M_3} \qquad \text{S-IF-FALSE}\ \frac{E, M_1 \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(0) \qquad E, M_1 \vDash_{\mathrm{s}} s_2 \Downarrow M_2}{E, M_1 \vDash_{\mathrm{s}} \texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow M_2}$$

$$\text{S-IF-TRUE}\ \frac{E, M_1 \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(n) \qquad n \neq 0 \qquad E, M_1 \vDash_{\mathrm{s}} s_1 \Downarrow M_2}{E, M_1 \vDash_{\mathrm{s}} \texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow M_2} \qquad \text{S-WHILE-FALSE}\ \frac{E, M \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(0)}{E, M \vDash_{\mathrm{s}} \texttt{while}\ (e)\ s \Downarrow M}$$

$$\text{S-WHILE-TRUE}\ \frac{E, M_1 \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(n) \qquad n \neq 0 \qquad E, M_1 \vDash_{\mathrm{s}} s \Downarrow M_2 \qquad E, M_2 \vDash_{\mathrm{s}} \texttt{while}\ (e)\ s \Downarrow M_3}{E, M_1 \vDash_{\mathrm{s}} \texttt{while}\ (e)\ s \Downarrow M_3}$$

$$\text{S-LET}\ \frac{E_2, M_2 = \mathsf{alloc\_var}(E_1, M_1, x, \tau) \qquad E_2, M_2 \vDash_{\mathrm{s}} s \Downarrow M_3 \qquad M_4 = \mathsf{dealloc\_var}(E_2, M_3, x)}{E_1, M_1 \vDash_{\mathrm{s}} \texttt{let}\ x : \tau\ \texttt{in}\ s\ \texttt{end} \Downarrow M_4}$$

FIGURE 2.17 – Évaluation des instructions composées.

tuée par une fonction auxiliaire dealloc_var, définie ci-dessous, qui consulte l'environnement pour déterminer le bloc associé à la variable à déallouer, et effectue l'opération de déallocation en mémoire. À noter que contrairement à la fonction d'allocation, cette fonction ne renvoie pas de nouvel environnement, puisque celui-ci serait simplement l'environnement initial avant allocation.

$$\begin{aligned}
\mathsf{alloc_var}(E_1, M_1, x, \tau) &\triangleq (E_2, M_2) \\
\text{où } b, M_2 &\triangleq \mathsf{alloc}(M_1, \mathsf{sizeof}(\mathsf{mtype}(\tau))) \\
\text{et } E_2 &\triangleq \mathsf{add}(E_1, x, b) \\
\mathsf{dealloc_var}(E_1, M_1, x) &\triangleq \begin{cases} \lfloor M_2 \rfloor & \text{si } \mathsf{lookup}(E_1, x) = \lfloor b \rfloor \\ & \text{et } \mathsf{free}(M_1, b) = \lfloor M_2 \rfloor \\ \varepsilon & \text{sinon} \end{cases}
\end{aligned}$$

2.4 Discussion et travaux connexes

Les principales propriétés du langage sont : modèle de C centré sur les pointeurs et gestion de la mémoire, et annotation formelles sur les propriétés mémoire aux fins de vérification d'assertions à l'exécution.

De nombreux travaux ont été consacrés à l'étude sémantique du langage C. Si ces travaux ont d'abord été limités à des formalisations partielles (NORRISH donne dans sa thèse [Nor99] une vue d'ensemble de cette partie de la littérature), plus récemment les avancées successives dans le domaine, et en particulier les travaux autour du compilateur certifié CompCert [BL09 ; Ler09], ont considérablement rapproché la perspective d'une formalisation mécanisée de l'intégralité du standard [Kre15].

Concernant les annotations formelles pour la vérification à l'exécution, notre langage modélise une partie des assertions d'E-ACSL, qui est une variante d'ACSL. Dans sa thèse [Her13], HERMS a formalisé en Coq une extension du langage Clight de CompCert avec des annotations

logiques. Ces dernières recouvrent une partie significative d'ACSL, qui ne comprend toutefois pas les propriétés liées aux blocs.

Un autre pan de la littérature à considérer est constitué des travaux autour du langage de spécification JML [LBR06] (*Java Modeling Language*), qui a inspiré l'élaboration d'ACSL [Kir+15], et qui intègre dès la conception la perspective de vérifier les assertions à l'exécution. JML a notamment fait l'objet d'une formalisation en Coq par LEHNER [Leh11].

Chapitre 3

Transformation de programme

On définit dans ce chapitre une modélisation formelle de la transformation de programme opérée par le greffon E-ACSL.

La section 3.1 présente au travers d'un exemple une vue d'ensemble de la transformation, et met en lumière le besoin de définir deux éléments techniques : une structure de données, la *mémoire d'observation*, et un langage, le *langage cible* de la transformation. Ces deux éléments sont respectivement présentés dans les sections 3.2 et 3.3, et permettent de définir formellement, dans la section 3.4, la transformation étudiée.

3.1 Aperçu et principe général

L'objectif principal de notre transformation de programme est de générer, pour chaque assertion logique du programme source, un code effectuant l'évaluation du prédicat concerné : la transformation opère une *traduction* de prédicats logiques en code exécutable.

3.1.1 Traduction des prédicats

Considérons le programme de la figure 2.7, qui comporte une unique assertion (à la ligne 21), et tentons de caractériser le programme résultant de la traduction de cette assertion. Une première idée à envisager est d'avoir un programme identique au programme source, sauf en ce qui concerne l'assertion logique, qui est remplacée par une section de code de nature programmatique. Schématiquement, ce processus de traduction pourrait être représenté par la figure 3.1. Plus concrètement, le code obtenu pourrait ressembler à celui de la figure 3.2 : par rapport au programme source, l'instruction `logical_assert` (laissée dans le code sous forme de commentaire, mais désormais sans effet) a été remplacée par une instruction `assert` précédée d'une section de code (lignes 5 à 10), dont le contenu ne nécessite pas d'être examiné en détail pour l'instant.

La description formelle d'une telle transformation serait déterminée par une certaine fonction de traduction $[\![\cdot]\!]_\Pi$: **pred** $\rightarrow$ **stmt**, et l'on définirait la transformation sous la forme d'une fonction sur les instructions $[\![\cdot]\!]_\Sigma$: **stmt** $\rightarrow$ **stmt** consistant en une simple descente récursive ne modifiant que les assertions. Ce schéma correspond à la définition par cas ci-dessous, dans laquelle la dernière ligne correspond aux cas non traités par les lignes précédentes :

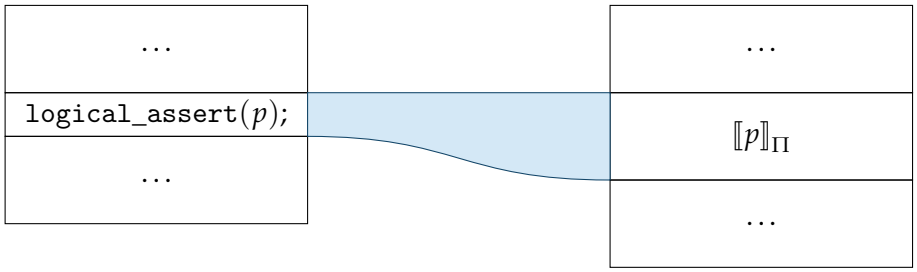

FIGURE 3.1 – Traduction des prédicats. En bleu, la correspondance entre une assertion sur un prédicat logique et sa traduction en code.

$$
\begin{aligned}
\llbracket \texttt{logical\_assert}(p); \rrbracket_\Sigma &\triangleq \llbracket p \rrbracket_\Pi \\
\llbracket s_1\ s_2 \rrbracket_\Sigma &\triangleq \llbracket s_1 \rrbracket_\Sigma\ \llbracket s_2 \rrbracket_\Sigma \\
\llbracket \texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \rrbracket_\Sigma &\triangleq \texttt{if}\ (e)\ \texttt{then}\ \llbracket s_1 \rrbracket_\Sigma\ \texttt{else}\ \llbracket s_2 \rrbracket_\Sigma \\
\llbracket \texttt{while}\ (e)\ s \rrbracket_\Sigma &\triangleq \texttt{while}\ (e)\ \llbracket s \rrbracket_\Sigma \\
\llbracket s \rrbracket_\Sigma &\triangleq s.
\end{aligned}
$$

Cependant, une difficulté apparaît dès que l'on tente de préciser la traduction des prédicats : comment, par exemple, traduire un prédicat tel que $\backslash\texttt{valid}(p)$? Celui-ci dénote une propriété de l'état de la mémoire du programme, propriété qui n'est pas exprimable dans la partie programmatique (c'est-à-dire hors prédicats et termes) de notre langage source.

Par conséquent, afin d'être en mesure de traduire les prédicats correspondant à des propriétés mémoire, nous allons matérialiser l'état de la mémoire dans une structure de donnée dédiée, que nous appellerons *mémoire d'observation*. Cette structure de données sera, elle, utilisable dans le code généré par la transformation, par le biais d'une interface de programmation. La définition formelle de la mémoire d'observation se trouve dans la section 3.2, et celle de

```
1   while (lo <= hi)
2   let mid: int64 in
3     mid = lo + (hi - lo) / 2;
4     // logical_assert(\valid(t + mid));
5     let res_0: int8 in
6       let res_1: int64* in
7         res_1 = t + mid;
8         res_0 = is_valid(res_1);
9       end
10      assert(res_0);
11    end
12    if (*(t + mid) == x) then
13      idx = mid;
14      lo = hi + 1;
15    else if (*(t + mid) < x) then
16      lo = mid + 1;
17    else
18      hi = mid - 1;
```

FIGURE 3.2 – Traduction de prédicat à partir du programme de la figure 2.7.

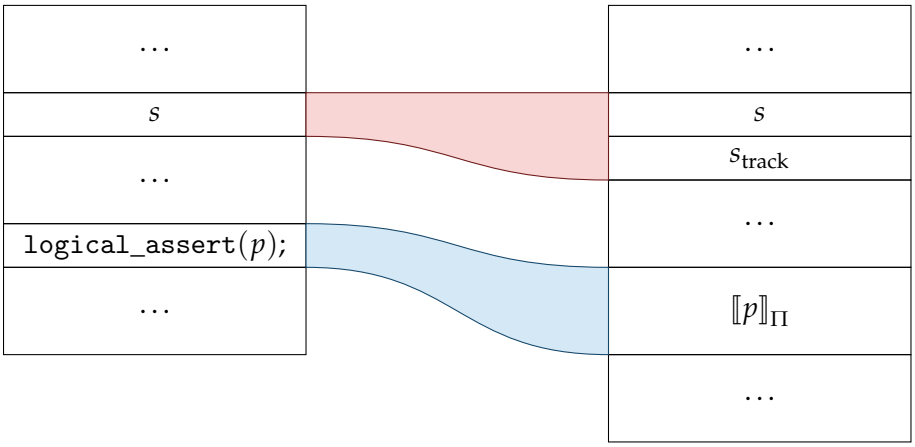

FIGURE 3.3 – Traduction des prédicats (en bleu) et suivi des opérations mémoire (en rouge).

son interface de programmation (qui détermine le *langage cible* de la transformation) dans la section 3.3, mais elles ne sont pas indispensables pour comprendre le reste de cette section.

3.1.2 Suivi des opérations mémoire

Supposons donc que nous disposons, pour les besoins de la transformation, d'une structure de données à même de décrire l'état de la mémoire. Le code issu de la traduction d'un prédicat pourra, par exemple, y consulter le statut de validité d'un pointeur, comme à la ligne 8 de la figure 3.2.

Il ne suffit toutefois pas de disposer d'une telle structure de données : encore faut-il que celle-ci reflète fidèlement la mémoire d'exécution, qui évolue au gré de l'exécution du programme. Cette exigence implique de suivre toutes les opérations sur la mémoire d'exécution et de les reporter au fur et à mesure dans la mémoire d'observation. Outre la traduction des prédicats, la transformation a donc un second rôle : la mise à jour de la mémoire d'observation au cours de l'exécution du programme observé. Celle-ci peut être réalisée, au niveau du code source, en accompagnant toute instruction modifiant la mémoire (d'exécution) par un code effectuant le report de cette modification dans la mémoire d'observation.

Une telle instrumentation additionnelle peut être visualisée schématiquement comme dans la figure 3.3. Si on l'applique à l'exemple de code de la figure 2.7, on obtient le code présenté dans la figure 3.4 . Ici aussi, nous ne nous soucions pas pour l'instant du sens précis des instructions insérées (en rouge dans le code).

Syntaxiquement, ce nouveau schéma peut être écrit sous la forme suivante, où s_{track} représente une instruction de suivi de la mémoire, que l'on ne détaille pas pour le moment :

$$
\begin{aligned}
[\![\texttt{logical\_assert}(p);]\!]_\Sigma &\triangleq [\![p]\!]_\Pi \\
[\![s_1\ s_2]\!]_\Sigma &\triangleq [\![s_1]\!]_\Sigma\ [\![s_2]\!]_\Sigma \\
[\![\texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]_\Sigma &\triangleq \texttt{if}\ (e)\ \texttt{then}\ [\![s_1]\!]_\Sigma\ \texttt{else}\ [\![s_2]\!]_\Sigma \\
[\![\texttt{while}\ (e)\ s]\!]_\Sigma &\triangleq \texttt{while}\ (e)\ [\![s]\!]_\Sigma \\
[\![s]\!]_\Sigma &\triangleq \begin{cases} s\ s_{\text{track}} & \text{si } s \text{ modifie la mémoire} \\ s & \text{sinon.} \end{cases}
\end{aligned}
$$

```
35            while (lo <= hi)
36              let mid: int64 in
37                store_block(&mid, sizeof(int64));
38                mid = lo + (hi - lo) / 2;
39                initialize(&mid);
40                // logical_assert(\valid(t + mid));
41                let res_0: int8 in
42                  let res_1: int64* in
43                    res_1 = t + mid;
44                    res_0 = is_valid(res_1);
45                  end
46                  assert(res_0);
47                end
48                if (*(t + mid) == x) then
49                  idx = mid;
50                  initialize(&idx);
51                  lo = hi + 1;
52                  initialize(&lo);
53                else if (*(t + mid) < x) then
54                  lo = mid + 1;
55                  initialize(&lo);
56                else
57                  hi = mid - 1;
58                  initialize(&hi);
59                delete_block(&mid);
60              end
```

FIGURE 3.4 – Traduction de prédicat et suivi mémoire.

3.1.3 Traductions intermédiaires et variables de résultat

Revenons à présent à la traduction des prédicats. Les lignes 41 à 47 de la figure 3.4 illustrent deux aspects structurants de la traduction : la décomposition en traductions intermédiaires, et l'usage de variables locales pour enregistrer les résultats de ces calculs intermédiaires.

À chaque prédicat est associé d'une part un code de traduction, et d'autre part une variable dite de résultat. Ces variables sont, dans notre exemple, *res*_0 et *res*_1. L'objectif de la traduction est de faire en sorte que, pour chaque prédicat, à l'issue de l'exécution du code de traduction de ce prédicat, la variable de résultat associée ait une valeur reflétant celle du prédicat.

Ainsi, la variable *res*_0, qui correspond au prédicat global $\backslash\texttt{valid}(t + mid)$, est censée être mise à la valeur correspondant à ce prédicat par le code compris entre les lignes 42 à 45, afin que l'assertion de la ligne 47 aboutisse au même résultat que l'assertion source.

3.2 Mémoire d'observation

La fonction de la mémoire d'observation est de décrire le contenu de la mémoire d'exécution : elle doit permettre de déterminer quels sont les blocs valides, quelle est leur longueur, quels sont les accès valides, ou quel est l'état d'initialisation des données à un emplacement donné.

$$\begin{array}{rcl}
\mathsf{empty} & : & \mathbf{obs} \\
\mathsf{store_block} & : & \mathbf{obs} \times \mathbf{block} \times \mathbb{N} \to \mathbf{obs} \\
\mathsf{delete_block} & : & \mathbf{obs} \times \mathbf{block} \to \mathbf{obs} \\
\mathsf{is_valid} & : & \mathbf{obs} \times \mathbf{block} \to \mathbf{bool} \\
\mathsf{is_valid_access} & : & \mathbf{mtyp} \times \mathbf{obs} \times \mathbf{block} \times \mathbb{Z} \to \mathbf{bool} \\
\mathsf{is_initialized} & : & \mathbf{mtyp} \times \mathbf{obs} \times \mathbf{block} \times \mathbb{Z} \to \mathbf{bool} \\
\mathsf{initialize} & : & \mathbf{mtyp} \times \mathbf{obs} \times \mathbf{block} \times \mathbb{Z} \to \mathbf{obs} \\
\mathsf{length} & : & \mathbf{obs} \times \mathbf{block} \to \mathbb{N}
\end{array}$$

On définit pour la mémoire d'observation un critère analogue à la propriété d'accès valide (définition 1) de la mémoire d'observation.

Définition 5 (accès valide pour la mémoire d'observation)**.**

$$\begin{array}{rl}
\mathsf{is_valid_access}(\kappa, \overline{M}, b, \delta) = & \mathsf{is_valid}(\overline{M}, b) \\
& \&\& \ \delta \geq 0 \\
& \&\& \ \delta + \mathsf{sizeof}(\kappa) \leq \mathsf{length}(\overline{M}, b)
\end{array}$$

Deux axiomes définissent l'effet de l'enregistrement d'un bloc sur la validité des blocs. L'axiome 3.1 indique qu'enregistrer un bloc en mémoire d'observation via l'opération store_block a pour effet de le rendre valide dans l'état résultant. L'axiome 3.2 indique que l'enregistrement d'un bloc en mémoire d'observation n'altère pas la validité des autres blocs.

Axiome 3.1 (validblock après storeblock – même bloc)**.**

$$\mathsf{store\_block}(\overline{M_1}, b, n) = \overline{M_2} \implies \mathsf{is\_valid}(\overline{M_2}, b) = \top$$

Axiome 3.2 (validblock après storeblock – autre bloc)**.**

$$b \neq b' \wedge \mathsf{store\_block}(\overline{M_1}, b, n) = \overline{M_2} \implies \mathsf{is\_valid}(\overline{M_2}, b') = \mathsf{is\_valid}(\overline{M_1}, b')$$

De la même manière, deux axiomes définissent l'effet de la suppression d'un bloc sur la validité. L'axiome 3.3 indique que la suppression d'un bloc de la mémoire d'observation le rend invalide dans l'état résultant. L'axiome 3.4 indique que la suppression d'un bloc en mémoire d'observation n'altère pas la validité des autres blocs.

Axiome 3.3 (validblock après deleteblock – même bloc)**.**

$$\mathsf{delete\_block}(\overline{M_1}, b) = \overline{M_2} \implies \mathsf{is\_valid}(\overline{M_2}, b) = \bot$$

Axiome 3.4 (validblock après deleteblock – autre bloc)**.**

$$b \neq b' \wedge \mathsf{delete\_block}(\overline{M_1}, b) = \overline{M_2} \implies \mathsf{is\_valid}(\overline{M_2}, b') = \mathsf{is\_valid}(\overline{M_1}, b')$$

L'axiome 3.5 indique que l'initialisation d'un emplacement en mémoire d'observation n'altère pas la validité des blocs.

Axiome 3.5 (validblock après initialize)**.**

$$\mathsf{initialize}(\kappa, \overline{M_1}, b, \delta) = \overline{M_2} \implies \mathsf{is\_valid}(\overline{M_2}, b') = \mathsf{is\_valid}(\overline{M_1}, b')$$

Deux axiomes définissent l'effet d'un enregistrement de bloc sur les longueurs des blocs. L'axiome 3.6 indique que l'enregistrement d'un bloc en mémoire d'observation définit la lon-

gueur de ce bloc dans l'état résultant. L'axiome 3.7 indique que l'enregistrement d'un bloc n'altère pas la longueur des autres blocs.

Axiome 3.6 (length après storeblock – même bloc)**.**

$$\mathsf{store\_block}(\overline{M_1}, b, n) = \overline{M_2} \implies \mathsf{length}(\overline{M_2}, b) = n$$

Axiome 3.7 (length après storeblock – autre bloc)**.**

$$b \neq b' \wedge \mathsf{store\_block}(\overline{M_1}, b, n) = \overline{M_2} \implies \mathsf{length}(\overline{M_2}, b') = \mathsf{length}(\overline{M_1}, b')$$

Les axiomes 3.8 et 3.9 expriment la préservation des longueurs de bloc à travers les suppressions de blocs et initialisations d'emplacement. Toutefois, la longueur du bloc après suppression n'est pas spécifiée.

Axiome 3.8 (length après deleteblock)**.**

$$b \neq b' \wedge \mathsf{delete\_block}(\overline{M_1}, b) = \overline{M_2} \implies \mathsf{length}(\overline{M_2}, b') = \mathsf{length}(\overline{M_1}, b')$$

Axiome 3.9 (length après initialize)**.**

$$\mathsf{initialize}(\kappa, \overline{M_1}, b, \delta) = \overline{M_2} \implies \mathsf{length}(\overline{M_2}, b') = \mathsf{length}(\overline{M_1}, b')$$

L'axiome 3.10 indique que les données dans un bloc nouvellement enregistré ne sont pas initialisées.

Axiome 3.10 (is_initialized après storeblock – même bloc)**.**

$$\mathsf{store\_block}(\overline{M_1}, b, n) = \overline{M_2} \implies \mathsf{is\_initialized}(\kappa, \overline{M_2}, b, \delta) = \bot$$

L'enregistrement d'un bloc n'affecte pas le statut d'initialisation des données dans les autres blocs, comme indiqué par l'axiome 3.11. De la même manière, la suppression d'un bloc n'affecte pas le statut d'initialisation des données dans les autres blocs (axiome 3.12).

Axiome 3.11 (is_initialized après storeblock – autre bloc)**.**

$$\begin{array}{ll} & b \neq b' \ \wedge \ \mathsf{store\_block}(\overline{M_1}, b, n) = \overline{M_2} \\ \implies & \mathsf{is\_initialized}(\kappa, \overline{M_2}, b', \delta') = \mathsf{is\_initialized}(\kappa, \overline{M_1}, b', \delta') \end{array}$$

Axiome 3.12 (is_initialized après deleteblock – autre bloc)**.**

$$\begin{array}{ll} & b \neq b' \wedge \mathsf{delete\_block}(\overline{M_1}, b, n) = \overline{M_2} \\ \implies & \mathsf{is\_initialized}(\kappa, \overline{M_2}, b', \delta') = \mathsf{is\_initialized}(\kappa, \overline{M_1}, b', \delta') \end{array}$$

L'axiome 3.13 définit l'effet de l'opération d'initialisation, qui est de marquer comme initialisées les données à l'emplacement concerné. La spécification de l'opération est complétée par l'axiome 3.14, qui indique que le statut d'initialisation ne change pas en dehors de l'emplacement initialisé.

Axiome 3.13 (is_initialized après initialize – même bloc)**.**

$$\mathsf{initialize}(\kappa, \overline{M_1}, b, \delta) = \overline{M_2} \implies \mathsf{is\_initialized}(\kappa, \overline{M_2}, b, \delta) = \top$$

Axiome 3.14 (is_initialized après initialize – autre bloc)**.**

$$\begin{array}{rl} & (b \neq b' \vee \delta' \geq \delta + \mathsf{sizeof}(\kappa) \vee \delta \geq \delta' + \mathsf{sizeof}(\kappa')) \\ \wedge & \mathsf{initialize}(\kappa, \overline{M_1}, b, \delta) = \overline{M_2} \\ \Longrightarrow & \mathsf{is\_initialized}(\kappa', \overline{M_2}, b', \delta') = \mathsf{is\_initialized}(\kappa', \overline{M_1}, b', \delta') \end{array}$$

3.3 Langage cible

Utiliser la mémoire d'observation définie à la section précédente nécessite quelques aménagements au niveau du langage de programmation. Le plus souvent en programmation, une structure de données définie dans une bibliothèque est munie d'une interface de programmation constituée d'un ensemble de fonctions. Puisque notre langage d'étude n'inclut pas d'appels de fonctions, nous ne disposons pas de cette possibilité. L'interface de programmation de la mémoire d'observation sera donc directement intégrée au langage, définissant ainsi notre *langage cible*.

3.3.1 Syntaxe

En termes d'éléments syntaxiques, la traduction des prédicats fait disparaître la partie logique (prédicats et termes) du langage source, pour n'en garder que la partie programmatique. Ainsi les assertions sur des prédicats sont remplacées par une forme d'assertion équivalente à la macro `assert` du langage C, c'est-à-dire opérant sur des expressions du langage. Afin d'accéder aux méta-données de la mémoire d'observation, le code issu de la traduction utilise des instructions spécifiques, qui permettent :

— de tester la validité d'un pointeur ;
— de tester l'initialisation des données d'un pointeur ;
— d'obtenir l'adresse de base d'un pointeur ;
— d'obtenir le *décalage* d'un pointeur (par rapport à son adresse de base, comme défini en section 2.2) ;
— de consulter la longueur d'un bloc enregistré.

Le suivi des opérations mémoire nécessite, quant à lui, des instructions pour enregistrer, supprimer, ou modifier des données de la mémoire d'observation. Une fois ces adaptations opérées, on obtient pour le langage cible les instructions de la figure 3.5 ; on notera leur type **stmt**′.

On retrouve dans la partie supérieure les instructions communes au langage source et au langage cible, déjà présentes dans la figure 2.6. Dans la seconde partie se trouvent les instructions spécifiques au langage cible.

L'assertion du langage source `logical_assert`(p), qui porte sur un prédicat, est remplacée par une assertion `assert`(e) sur une expression. Celle-ci interrompt l'exécution du programme si l'expression e a une valeur nulle, et est sans effet dans le cas contraire. Viennent ensuite les instructions liées à la mémoire d'observation.

L'instruction `store_block`(p, e); se comporte comme une fonction prenant en arguments un pointeur p vers la tête d'un bloc (tel que résultant d'une allocation par exemple) et une expression e de type entier. Elle a pour effet d'enregistrer dans la mémoire d'observation l'information que le bloc associé à p est alloué et a la taille donnée par e (en octets).

Son complémentaire `delete_block`(p); prend également en argument un pointeur p vers la tête d'un bloc, et efface de la mémoire d'observation les informations liées à ce bloc.

$e =$ `is_valid`(p); consulte la validité du pointeur p dans la mémoire d'observation, et écrit dans la valeur gauche e un entier encodant celle-ci : 1 si le pointeur est valide, et 0 sinon.

Suivant le même fonctionnement, $e =$ `is_initialized`(p); écrit dans e le statut d'initialisation des données pointées par p, tel qu'enregistré dans la mémoire d'observation.

stmt′	$s ::=$	`skip;`	instruction nulle
	$\mid$	$e = e$;	affectation
	$\mid$	$e =$ `malloc`(e);	allocation
	$\mid$	`free`(e);	déallocation
	$\mid$	$s\ s$	séquence
	$\mid$	`if`(e) `then` s `else` s	instruction conditionnelle
	$\mid$	`while`$(e)\ s$	boucle
	$\mid$	`let` $x : \tau$ `in` s `end`	variable locale
	$\mid$	`assert`(e);	assertion sur une expression
	$\mid$	`store_block`(e, e);	enregistrement d'un bloc
	$\mid$	`delete_block`(e);	suppression d'un bloc
	$\mid$	$e =$ `is_valid`(e);	test de validité
	$\mid$	$e =$ `is_initialized`(e);	test d'initialisation
	$\mid$	`initialize`(e);	enregistrement d'initialisation
	$\mid$	$e =$ `base_address`(e);	adresse de base d'un pointeur
	$\mid$	$e =$ `offset`(e);	décalage d'un pointeur
	$\mid$	$e =$ `block_length`(e);	longueur du bloc d'un pointeur

FIGURE 3.5 – Instructions du langage cible.

L'instruction `initialize`(p); permet de modifier ce statut, en marquant comme initialisées les données pointées par p. On peut noter l'absence d'une opération complémentaire de « déinitialisation ». En effet, nous considérons comme initialisé tout emplacement mémoire ayant fait l'objet d'au moins une écriture : ce changement de statut n'est donc pas réversible.

Enfin, trois instructions donnent accès aux propriétés relatives aux blocs. De la même manière que les instructions de test de validité et d'initialisation, l'instruction $e =$ `base_address`(p); écrit dans e (évaluée comme valeur gauche) l'adresse de base du pointeur p. De même $e =$ `offset`(p); écrit dans e le décalage de p, et $e =$ `block_length`(p); écrit dans e la longueur du bloc pointé par p.

3.3.2 Sémantique des instructions

La sémantique du langage cible est définie par trois relations, dont deux déjà présentes dans le langage source :

— l'évaluation des expressions en valeurs, de la forme $E, M \vDash_{\mathrm{e}} e \Downarrow v$

— leur évaluation en valeurs gauches, de la forme $E, M \vDash_{\mathrm{lv}} e \Downarrow b, \delta$

— et l'évaluation des instructions, de la forme $E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} s \Downarrow M_2, \overline{M_2}$.

Comme dans le cas du langage source, les instructions mènent la mémoire du programme d'un état initial M_1 à un état final M_2 ; cependant nos programmes sont désormais aussi dotés d'un autre état : celui de la mémoire d'observation. Ainsi, un jugement d'évaluation de la forme $E, M_1, \overline{M_1} \vDash_{s'} s \Downarrow M_2, \overline{M_2}$ représente l'évaluation de s dans l'environnement E, la mémoire d'exécution M_1 et la mémoire d'observation $\overline{M_1}$, aboutissant à un état de la mémoire d'exécution M_2 et un état de la mémoire d'observation $\overline{M_2}$.

Le rôle de la mémoire d'observation apparaît le plus clairement dans les règles d'évaluation des instructions servant à la consulter ou la modifier. Ces règles sont présentées dans la figure 3.6.

S′-STOREBLOCK
$$\frac{E, M_1 \vDash_{\mathrm{e}} p \Downarrow \mathsf{Ptr}(b, 0) \qquad E, M_1 \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(n) \qquad \mathsf{store\_block}(\overline{M_1}, b, n) = \lfloor \overline{M_2} \rfloor}{E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} \texttt{store\_block}(p, e); \Downarrow M_1, \overline{M_2}}$$

S′-DELETEBLOCK
$$\frac{E, M_1 \vDash_{\mathrm{e}} p \Downarrow \mathsf{Ptr}(b, 0) \qquad \mathsf{delete\_block}(\overline{M_1}, b) = \lfloor \overline{M_2} \rfloor}{E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} \texttt{delete\_block}(p); \Downarrow M_1, \overline{M_2}}$$

S′-IS-VALID
$$\frac{\begin{array}{c} E, M_1 \vDash_{\mathrm{lv}} e_1 \Downarrow b_1, \delta_1 \qquad \mathsf{is\_valid\_access}(\mathsf{mtype}(\tau), \overline{M_1}, b_2, \delta_2) = \beta \\ E, M_1 \vDash_{\mathrm{e}} e_2 : \tau* \Downarrow \mathsf{Ptr}(b_2, \delta_2) \qquad \mathsf{store}(\mathrm{i}8, M_1, b_1, \delta_1, \mathsf{Int}(\beta)) = \lfloor M_2 \rfloor \end{array}}{E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} e_1 = \texttt{is\_valid}(e_2); \Downarrow M_2, \overline{M_1}}$$

S′-IS-INITIALIZED
$$\frac{\begin{array}{c} E, M_1 \vDash_{\mathrm{lv}} e_1 \Downarrow b_1, \delta_1 \qquad \mathsf{is\_initialized}(\mathsf{mtype}(\tau), \overline{M_1}, b_2, \delta_2) = \beta \\ E, M_1 \vDash_{\mathrm{e}} e_2 : \tau* \Downarrow \mathsf{Ptr}(b_2, \delta_2) \qquad \mathsf{store}(\mathrm{i}8, M_1, b_1, \delta_1, \mathsf{Int}(\beta)) = \lfloor M_2 \rfloor \end{array}}{E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} e_1 = \texttt{is\_initialized}(e_2); \Downarrow M_2, \overline{M_1}}$$

S′-INITIALIZE
$$\frac{\begin{array}{c} E, M_1 \vDash_{\mathrm{e}} e : \tau* \Downarrow \mathsf{Ptr}(b, \delta) \\ \mathsf{initialize}(\mathsf{mtype}(\tau), \overline{M_1}, b, \delta) = \lfloor \overline{M_2} \rfloor \end{array}}{E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} \texttt{initialize}(e); \Downarrow M_1, \overline{M_2}}$$

S′-BASEADDR
$$\frac{\begin{array}{c} E, M_1 \vDash_{\mathrm{lv}} e_1 : \tau* \Downarrow b_1, \delta_1 \qquad E, M_1 \vDash_{\mathrm{e}} e_2 \Downarrow \mathsf{Ptr}(b_2, \delta_2) \\ \mathsf{store}(\mathsf{mtype}(\tau*), M_1, b_1, \delta_1, \mathsf{Ptr}(b_2, 0)) = \lfloor M_2 \rfloor \end{array}}{E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} e_1 = \texttt{base\_address}(e_2); \Downarrow M_2, \overline{M_1}}$$

S′-BLOCK-LENGTH
$$\frac{\begin{array}{c} E, M_1 \vDash_{\mathrm{lv}} e_1 \Downarrow b_1, \delta_1 \\ E, M_1 \vDash_{\mathrm{e}} e_2 \Downarrow \mathsf{Ptr}(b_2, \delta_2) \qquad \mathsf{length}(\overline{M_1}, b_2) = \lfloor n \rfloor \qquad \mathsf{store}(\mathrm{i}64, M_1, b_1, \delta_1, \mathsf{Int}(n)) = \lfloor M_2 \rfloor \end{array}}{E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} e_1 = \texttt{block\_length}(e_2); \Downarrow M_2, \overline{M_1}}$$

S′-OFFSET
$$\frac{\begin{array}{c} E, M_1 \vDash_{\mathrm{lv}} e_1 \Downarrow b_1, \delta_1 \qquad E, M_1 \vDash_{\mathrm{e}} e_2 \Downarrow \mathsf{Ptr}(b_2, \delta_2) \\ \mathsf{store}(\mathrm{i}64, M_1, b_1, \delta_1, \mathsf{Int}(\delta_2)) = \lfloor M_2 \rfloor \end{array}}{E, M_1, \overline{M_1} \vDash_{\mathrm{s}'} e_1 = \texttt{offset}(e_2); \Downarrow M_2, \overline{M_1}}$$

FIGURE 3.6 – Évaluation des instructions du langage cible relatives à la mémoire.

L'instruction `store_block`(p, e); sert d'interface au niveau du langage cible à l'opération store_block de la mémoire d'observation (règle S'-STORE-BLOCK) : si p est évalué en un pointeur vers la tête d'un bloc b, et e en un entier n, l'information que le bloc b est alloué et de taille n est ajoutée à la mémoire d'observation $\overline{M_1}$, aboutissant à un nouvel état $\overline{M_2}$. La mémoire d'exécution, elle, est laissée inchangée.

De même, `delete_block`(p); effectue un appel à l'opération de mémoire d'observation sous-jacente en laissant intacte la mémoire d'exécution (règle S'-DELETE-BLOCK).

Dans le cas du test de validité (règle S'-IS-VALID), l'opération is_valid_access de la mémoire d'observation est utilisée pour acquérir le statut de validité du pointeur concerné, tandis que le résultat, encodé par un entier, est enregistré en mémoire d'exécution par une opération store.

De la même façon, le test d'initialisation (règle S'-IS-INITIALIZED) fait appel à l'opération is_initialized de la mémoire d'observation et à un store de la mémoire d'exécution.

L'instruction `initialize`(e); est, quant à elle, une traduction quasiment directe de la primitive initialize de la mémoire d'observation (règle S'-INITIALIZE).

Le calcul de certaines propriétés de bloc ne repose pas sur la mémoire d'observation, mais seulement sur la représentation des pointeurs en un bloc et un décalage. Ainsi, les instructions `base_address` et `offset` (règles S'-BASE-ADDR et S'-OFFSET) n'engendrent comme opération mémoire que l'écriture de leur résultat en mémoire d'exécution. Obtenir la longueur d'un bloc, via l'instruction `bloc_length` (règle S'-BLOCK-LENGTH) nécessite en revanche un appel à l'opération correspondante la mémoire d'observation.

Les règles d'évaluation des instructions restantes, présentées dans la figure 3.7 sont identiques à leurs équivalents du langage source, à la présence de la mémoire d'observation près. Seule diffère l'assertion (règle S'-ASSERT), qui repose maintenant sur l'évaluation d'une expression entière plutôt que sur celle d'un prédicat.

3.4 Définition formelle de la transformation

La transformation de programme est constituée de trois fonctions opérant respectivement sur les instructions, prédicats et termes du langage source, et renvoyant toutes une section de code en langage cible (de type **stmt'**).

3.4.1 Traduction des instructions et suivi de la mémoire

La traduction $[\![s]\!]_\Sigma$ d'une instruction source s (type **stmt**) est une instruction cible (type **stmt'**) obtenue à partir de s en insérant aux endroits appropriés des instructions de suivi des opérations mémoire, et en traduisant en code les assertions logiques contenues dans s. Sa définition inductive formelle est donnée dans la figure 3.8.

Le code généré pour une affectation est une séquence de deux instructions : la première est l'affectation source, et la seconde l'enregistrement en mémoire d'observation de l'initialisation de l'expression à gauche de l'affectation. À noter que la primitive d'initialisation n'est pas appliquée à l'expression telle quelle, mais à son adresse. De cette façon, l'« argument » de `initialize`$()$; est bien évalué en un pointeur, conformément à la sémantique de la primitive (règle S'-INITIALIZE de la figure 3.6).

L'instrumentation de l'allocation de mémoire (`malloc`) ajoute à l'instruction source deux instructions de suivi de la mémoire : d'une part l'enregistrement du bloc nouvellement alloué, d'autre part l'initialisation de l'emplacement mémoire où est écrit le pointeur vers le nouveau bloc. On retrouve ainsi dans l'instrumentation les deux opérations mémoire (allocation et écriture) intervenant dans la sémantique de l'instruction (section 2.3.5).

La déallocation, quant à elle, n'est accompagnée que de la suppression du bloc concerné de la mémoire d'observation.

$$\text{S'-ASSERT}\ \frac{E, M \vDash_{\text{e}} e \Downarrow \mathsf{Int}(n) \qquad n \neq 0}{E, M, \overline{M} \vDash_{\text{s}'} \texttt{assert}(e); \Downarrow M, \overline{M}}$$

$$\text{S'-SEQ}\ \frac{E, M_1, \overline{M_1} \vDash_{\text{s}'} s_1 \Downarrow M_2, \overline{M_2} \qquad E, M_2, \overline{M_2} \vDash_{\text{s}'} s_2 \Downarrow M_3, , \overline{M_3}}{E, M_1, \overline{M_1} \vDash_{\text{s}'} s_1\ s_2 \Downarrow M_3, \overline{M_3}}$$

$$\text{S'-IF-FALSE}\ \frac{E, M_1 \vDash_{\text{e}} e \Downarrow \mathsf{Int}(0) \qquad E, M_1, \overline{M_1} \vDash_{\text{s}'} s_2 \Downarrow M_2, \overline{M_2}}{E, M_1, \overline{M_1} \vDash_{\text{s}'} \texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow M_2, \overline{M_2}}$$

$$\text{S'-IF-TRUE}\ \frac{E, M_1 \vDash_{\text{e}} e \Downarrow \mathsf{Int}(n) \qquad n \neq 0 \qquad E, M_1, \overline{M_1} \vDash_{\text{s}'} s_1 \Downarrow M_2, \overline{M_2}}{E, M_1, \overline{M_1} \vDash_{\text{s}'} \texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow M_2, \overline{M_2}}$$

$$\text{S'-WHILE-FALSE}\ \frac{E, M \vDash_{\text{e}} e \Downarrow \mathsf{Int}(0)}{E, M, \overline{M} \vDash_{\text{s}'} \texttt{while}\ (e)\ s \Downarrow M, \overline{M}}$$

$$\text{S'-WHILE-TRUE}\ \frac{E, M_1 \vDash_{\text{e}} e \Downarrow \mathsf{Int}(n) \qquad n \neq 0 \qquad E, M_1, \overline{M_1} \vDash_{\text{s}'} s \Downarrow M_2, \overline{M_2} \qquad E, M_2, \overline{M_2} \vDash_{\text{s}'} \texttt{while}\ (e)\ s \Downarrow M_3, \overline{M_3}}{E, M_1, \overline{M_1} \vDash_{\text{s}'} \texttt{while}\ (e)\ s \Downarrow M_3, \overline{M_3}}$$

$$\text{S'-LET}\ \frac{E_2, M_2 = \mathsf{alloc\_var}(E_1, M_1, x, \tau) \qquad E_2, M_2, \overline{M_1} \vDash_{\text{s}'} s \Downarrow M_3, \overline{M_2} \qquad M_4 = \mathsf{dealloc\_var}(E_2, M_3, x)}{E_1, M_1, \overline{M_1} \vDash_{\text{s}'} \texttt{let}\ x : \tau\ \texttt{in}\ s\ \texttt{end} \Downarrow M_4, \overline{M_2}}$$

FIGURE 3.7 – Évaluation des instructions du langage cible similaires à celles du langage source.

Une dernière instruction nécessite d'être instrumentée pour le suivi de la mémoire. Il s'agit de l'introduction d'une variable locale, matérialisée dans notre langage par le mot-clé `let` et correspondant en C aux déclarations de variable dans une portée lexicale délimitée par des accolades. En effet, l'exécution de cette construction donne lieu à une allocation avant l'exécution de l'instruction qu'elle enclôt, et à la déallocation correspondante après. L'instrumentation reflète donc en mémoire d'observation ces deux opérations. Par ailleurs, l'instruction interne est récursivement instrumentée.

L'instrumentation des instructions de flot de contrôle (séquence, branchement conditionnel, boucle) consiste simplement en une application récursive.

Reste à traiter le cas des assertions sur des prédicats, qui constitue le cœur du problème. Étant donné que ce cas repose sur la traduction des prédicats, il est traité dans la section suivante.

$$
\begin{array}{rcl}
[\![\texttt{skip;}]\!]_\Sigma & \triangleq & \texttt{skip;} \\
[\![l = e;]\!]_\Sigma & \triangleq & l = e; \\
 & & \texttt{initialize}(\&l); \\
[\![p = \texttt{malloc}(e);]\!]_\Sigma & \triangleq & p = \texttt{malloc}(e); \\
 & & \texttt{store\_block}(p, e); \\
 & & \texttt{initialize}(\&p); \\
[\![\texttt{free}(p);]\!]_\Sigma & \triangleq & \texttt{free}(p); \\
 & & \texttt{delete\_block}(p); \\
[\![s_1\ s_2]\!]_\Sigma & \triangleq & [\![s_1]\!]_\Sigma\ [\![s_2]\!]_\Sigma \\
[\![\texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!]_\Sigma & \triangleq & \texttt{if}\ (e)\ \texttt{then}\ [\![s_1]\!]_\Sigma\ \texttt{else}\ [\![s_2]\!]_\Sigma \\
[\![\texttt{while}\ (e)\ s]\!]_\Sigma & \triangleq & \texttt{while}\ (e)\ [\![s]\!]_\Sigma \\
[\![\texttt{let}\ x : \tau\ \texttt{in}\ s\ \texttt{end}]\!]_\Sigma & \triangleq & \texttt{let}\ x : \tau\ \texttt{in} \\
 & & \quad \texttt{store\_block}(\&x, \mathsf{sizeof}(\tau)); \\
 & & \quad [\![s]\!]_\Sigma \\
 & & \quad \texttt{delete\_block}(\&x); \\
 & & \texttt{end} \\
[\![\texttt{logical\_assert}(p);]\!]_\Sigma & \triangleq & \texttt{let}\ \mathsf{res}(0)\texttt{in} \\
 & & \quad [\![p]\!]^0_\Pi; \\
 & & \quad \texttt{assert}(\mathsf{res}(0)); \\
 & & \texttt{end}
\end{array}
$$

FIGURE 3.8 – Traduction des instructions.

3.4.2 Traduction des prédicats

Traitement des variables Le but de la traduction des prédicats est de permettre, au niveau des instructions, la substitution des assertions du langage source, portant sur des prédicats, par celles du langage cible, qui portent, elles, sur des expressions.

Le principe général de cette traduction, que nous avions présenté dans les grandes lignes en section 3.1.3, consiste à associer à chaque prédicat variable dite de résultat dont la valeur au terme de l'exécution du code de traduction reflétera la valeur de vérité du prédicat. Ce principe correspond algorithmiquement à une traduction récursive, en suivant la structure syntaxique des prédicats : si p' est un sous-prédicat d'un prédicat p, le code de traduction de p (noté provisoirement $[\![p]\!]_\Pi$) inclura celui de p' (généré récursivement, et noté provisoirement $[\![p']\!]_\Pi$), qui calculera la valeur de la variable de résultat associée à p'. Celle-ci pourra donc être utilisée par $[\![p]\!]_\Pi$, qui devra en retour calculer (la valeur de) sa propre variable de résultat. Ce principe guide également la traduction des termes.

En application de ce principe, dans le code d'exemple de la figure 3.4 (page 36), dont un extrait est reproduit ci-dessous par commodité, le code de traduction du prédicat $\backslash\texttt{valid}(t + mid)$ (lignes 40 à 43) réalise, à la ligne 42, une écriture dans la variable de résultat *res*_0 associée à ce prédicat ; d'autre part, il se sert de la variable *res*_1 associée au terme $t + mid$, qui est calculée par le code de traduction de ce terme (ligne 41).

```
38              mid = lo + (hi - lo) / 2;
39              initialize(&mid);
40              // logical_assert(\valid(t + mid));
```

```
41                let res_0: int8 in
42                  let res_1: int64* in
43                    res_1 = t + mid;
44                    res_0 = is_valid(res_1);
45                  end
```

Les fonctions de traduction des prédicats et termes reposant sur ce principe d'introduction de variables intermédiaires, leur définition formelle requiert nécessairement un formalisme adapté pour rendre compte de la génération et de l'utilisation de ces variables. Le traitement des variables, et plus généralement des *noms*, est une problématique récurrente en langages de programmation, qui peut être approchée de différentes manières [Ayd+08]. Une solution relativement simple suffit dans notre cas, car les langages intervenant dans notre problème ne comprennent pas de constructions telles que les appels de fonctions, usuellement responsables des difficultés en la matière.

Nous avons donc retenu une approche simple consistant en une indexation des variables par des entiers naturels, autrement dit : une numérotation. Étant donné un programme source à traduire, on suppose l'existence d'un ensemble infini dénombrable $\mathcal{V}$ de variables, dont aucune ne figure dans le programme source. Cet ensemble est accompagné d'une fonction $\mathsf{res} : \mathbb{N} \to \mathcal{V}$ injective, associant donc à chaque entier naturel une variable unique, distincte à la fois de toute variable du programme source et de toute autre variable qui serait générée par un appel à la fonction res avec un argument différent.

En pratique, il est courant de réaliser une telle fonction en définissant $\mathsf{res}(n)$ comme la concaténation d'un préfixe bien choisi avec la représentation en chaîne de caractères de l'entier n. Ainsi, nos codes d'exemple correspondent à une telle mise en œuvre avec comme préfixe la chaîne de caractères « *res_* ».

Définition formelle Nous disposons à présent de tout l'outillage nécessaire à la définition de la traduction des prédicats. Étant donné un prédicat p à traduire, on notera $[\![p]\!]^n_\Pi$ le code de traduction de p calculant la variable de résultat $\mathsf{res}(n)$.

Commençons par traiter le dernier cas de la traduction des instructions, à savoir celui des assertions logiques (dernier cas de la figure 3.8), que nous avions laissé de côté dans la section précédente.

La traduction d'une assertion logique sur un prédicat p débute par l'introduction d'une variable de résultat pour p. Concernant le nom de cette variable, il suffit de prendre la première disponible, à savoir $\mathsf{res}(0)$. Le code de traduction de p, $[\![p]\!]^0_\Pi$, calcule ensuite la valeur de $\mathsf{res}(0)$, après quoi une assertion $\texttt{assert}(\mathsf{res}(0))$ termine ou non l'exécution, selon la valeur de $\mathsf{res}(0)$. Conformément à l'objectif global de la traduction, l'assertion logique `logical_assert` a été remplacée par une assertion simple `assert`. Notons que l'ensemble du code généré pour une assertion est contenu dans la portée lexicale locale créée par l'introduction de la variable de résultat.

Passons maintenant à la traduction des prédicats à proprement parler. La fonction de traduction est définie par induction sur la structure du prédicat à traduire. La figure 3.9 présente les cas d'induction correspondant au fragment logique du langage des prédicats : valeurs de vérité, négation, et conjonction.

Traduire les prédicats `\true` et `\false` se résume à écrire dans la variable de résultat leur encodage usuel par un entier (respectivement 1 et 0).

La négation est le cas le plus simple de traduction récursive. Une nouvelle variable de résultat, $\mathsf{res}(n+1)$, est d'abord déclarée, correspondant au prédicat nié, puis sa valeur calculée par traduction récursive, après quoi l'opérateur unaire de négation ! est appliqué. L'application récursive de la fonction de traduction est accompagnée d'une incrémentation du numéro de la

$$
\begin{array}{rcl}
[\![\backslash\texttt{true}]\!]^n_\Pi & \triangleq & \mathsf{res}(n) = 1; \\
[\![\backslash\texttt{false}]\!]^n_\Pi & \triangleq & \mathsf{res}(n) = 0; \\
[\![\neg p]\!]^n_\Pi & \triangleq & \texttt{let } \mathsf{res}(n+1) : \texttt{int8 in} \\
 & & \quad [\![p]\!]^{n+1}_\Pi \\
 & & \quad \mathsf{res}(n) = \,!\,\mathsf{res}(n+1); \\
 & & \texttt{end} \\
[\![p_1 \wedge p_2]\!]^n_\Pi & \triangleq & \texttt{let } \mathsf{res}(n+1) : \texttt{int8 in} \\
 & & \quad [\![p_1]\!]^{n+1}_\Pi \\
 & & \quad \texttt{if } (\mathsf{res}(n+1)) \\
 & & \quad \texttt{then let } \mathsf{res}(n+1) : \texttt{int8 in} \\
 & & \quad\quad [\![p_2]\!]^{n+1}_\Pi \\
 & & \quad\quad \mathsf{res}(n) = \mathsf{res}(n+1); \\
 & & \quad \texttt{end else} \\
 & & \quad\quad \mathsf{res}(n) = 0; \\
 & & \texttt{end}
\end{array}
$$

FIGURE 3.9 – Traduction des prédicats (constantes et connecteurs logiques).

variable de résultat « courante », de façon à ce que le code de traduction du prédicat nié écrive le résultat de son calcul dans $\mathsf{res}(n+1)$.

La traduction des conjonctions reflète le caractère paresseux de leur évaluation dans le langage source. Si cette propriété n'est pas strictement nécessaire à notre formalisation (étant donné qu'elle n'inclut pas la gestion des erreurs d'exécution), elle demeure néanmoins souhaitable pour la cohérence de la traduction, mais également utile pour établir ses propriétés sémantiques (cf. chapitre 4). Ainsi, dans la traduction d'une conjonction $p_1 \wedge p_2$, a lieu *dans un premier temps* l'évaluation de $[\![p_1]\!]^{n+1}_\Pi$ puis, *seulement si* le résultat $\mathsf{res}(n+1)$ est vrai, celle de $[\![p_2]\!]^{n+1}_\Pi$.

On peut remarquer que le code $[\![p_1 \wedge p_2]\!]^n_\Pi$ comprend deux déclarations de la variable $\mathsf{res}(n+1)$, l'une au tout début du code généré, et l'autre dans la branche `then` de la conditionnelle. Par ailleurs, les traductions récursives de p_1 et p_2 sont toutes deux faites avec le même indice de variable de résultat $n+1$. Ce double emploi de la variable $\mathsf{res}(n+1)$ ne met pas en danger la validité de la traduction : en effet, la première occurrence de la variable correspond à la validité de p_1, et n'est utilisée que pour déterminer la branche de la conditionnelle à exécuter. D'autre part, dans le cas où la condition est vraie et la branche `then` évaluée, la sémantique de notre langage indique que la nouvelle déclaration vient « masquer » l'ancienne.

Les trois formes de prédicats restantes, traitées dans la figure 3.10, correspondent aux propriétés mémoire (`\valid()` et `\initialized()`) d'une part, et aux comparaisons entre termes d'autre part. Ces prédicats portent sur des termes, et leur schéma de traduction fait par conséquent intervenir celui des termes, qui suit lui aussi le principe d'une variable de résultat associée à un code de calcul de la valeur de ce résultat.

Pour traduire un prédicat $\backslash\texttt{valid}(t)$ et enregistrer le résultat dans la variable $\mathsf{res}(n)$, une variable de résultat $\mathsf{res}(n+1)$ pour t est d'abord déclarée, avec le même type pointeur que celui de t. Cette déclaration est suivie du code $[\![t]\!]^{n+1}_\mathrm{T}$ calculant la valeur du terme, puis de la primitive `is_valid` pour consulter en mémoire d'observation la validité du pointeur t, et reporter celle-ci dans la variable de résultat du prédicat. La traduction d'un prédicat $\backslash\texttt{initialized}(t)$ est

identique, à la primitive de consultation de la mémoire d'observation près.

Les comparaisons de termes (égalité et inégalité) sont traduites par les opérateurs binaires correspondants.

$$
\begin{array}{rcl}
[\![\backslash\texttt{valid}(t:\tau)]\!]^n_{\Pi} & \triangleq & \texttt{let } \mathsf{res}(n+1):\tau \texttt{ in} \\
 & & \quad [\![t]\!]^{n+1}_{\mathrm{T}} \\
 & & \quad \mathsf{res}(n) = \texttt{is\_valid}(\mathsf{res}(n+1)); \\
 & & \texttt{end} \\
[\![\backslash\texttt{initialized}(t:\tau)]\!]^n_{\Pi} & \triangleq & \texttt{let } \mathsf{res}(n+1):\tau \texttt{ in} \\
 & & \quad [\![t]\!]^{n+1}_{\mathrm{T}} \\
 & & \quad \mathsf{res}(n) = \texttt{is\_initialized}(\mathsf{res}(n+1)); \\
 & & \texttt{end} \\
[\![(t_1:\tau_1) \bowtie (t_2:\tau_2)]\!]^n_{\Pi} & \triangleq & \texttt{let } \mathsf{res}(n+1):\tau_1 \texttt{ in} \\
 & & \texttt{let } \mathsf{res}(n+2):\tau_2 \texttt{ in} \\
 & & \quad [\![t_1]\!]^{n+1}_{\mathrm{T}} \\
 & & \quad [\![t_2]\!]^{n+2}_{\mathrm{T}} \\
 & & \quad \mathsf{res}(n) = \mathsf{res}(n+1)\ [\![\bowtie]\!]_{op}\ \mathsf{res}(n+2); \\
 & & \texttt{end} \\
 & & \texttt{end} \\
[\![=]\!]_{op} & & == \\
[\![\leq]\!]_{op} & & <=
\end{array}
$$

FIGURE 3.10 – Traduction des prédicats portant sur des termes.

3.4.3 Traduction des termes

La traduction des termes (figure 3.11) est réalisée selon le même principe que celle des prédicats, et suit donc un schéma similaire : déclaration d'une variable de résultat, calcul de la valeur à y écrire, et utilisation. La principale différence se situe au niveau du type de la variable de résultat : là où les variables de résultat correspondant aux prédicats ont toutes le même type entier (servant d'encodage des Booléens), le type d'une variable de résultat pour un terme doit correspondre à celui du terme traduit.

Pour une expression e, il suffit de l'évaluer et d'enregistrer son résultat dans la variable de résultat.

Dans le cas du déréférencement d'un pointeur t_1, une étape de calcul intermédiaire est nécessaire, car le pointeur en question est un terme : son calcul peut donc nécessiter des instructions. On introduit donc une variable de résultat $\mathsf{res}(n+1)$ pour t_1, puis on effectue son calcul à l'aide du code $[\![t]\!]^{n+1}_{\mathrm{T}}$. À l'issue de ce code, $\mathsf{res}(n+1)$ contient la valeur du pointeur, qu'il suffit alors de déréférencer. La valeur issue du déréférencement est finalement écrite dans la variable de résultat du terme courant.

Le traitement des opérateurs (unaires et binaires) s'apparente à celui des connecteurs logiques dans la traduction des prédicats : introduction d'une ou deux variable(s) de résultat, selon le nombre de sous-termes à traiter, calcul des valeurs des sous-termes, puis application de l'opération à traduire. La différence se situe au niveau du flot de contrôle, plus simple dans le cas des opérateurs binaires que dans celui des connecteurs logiques (binaires). En effet, s'agis-

$$
\begin{aligned}
[\![e]\!]^n_{\mathrm{T}} &\triangleq \mathsf{res}(n) = e;\\
[\![\bar{*}(t:\tau)]\!]^n_{\mathrm{T}} &\triangleq \texttt{let}\ \mathsf{res}(n+1) : \tau\ \texttt{in}\\
&\quad\ [\![t]\!]^{n+1}_{\mathrm{T}}\\
&\quad\ \mathsf{res}(n) = *\mathsf{res}(n+1);\\
&\quad\ \texttt{end}\\
[\![(t_1:\tau_1)\ \bar{\ddagger}\ (t_2:\tau_2)]\!]^n_{\mathrm{T}} &\triangleq \texttt{let}\ \mathsf{res}(n+1) : \tau_1\ \texttt{in}\\
&\quad\ \texttt{let}\ \mathsf{res}(n+2) : \tau_2\ \texttt{in}\\
&\quad\ [\![t_1]\!]^{n+1}_{\mathrm{T}}\\
&\quad\ [\![t_2]\!]^{n+2}_{\mathrm{T}}\\
&\quad\ \mathsf{res}(n) = \mathsf{res}(n+1)\ \ddagger\ \mathsf{res}(n+2);\\
&\quad\ \texttt{end}\\
&\quad\ \texttt{end}\\
[\![\ \bar{\dagger}\ (t:\tau)]\!]^n_{\mathrm{T}} &\triangleq \texttt{let}\ \mathsf{res}(n+1) : \tau\ \texttt{in}\\
&\quad\ [\![t_1]\!]^{n+1}_{\mathrm{T}}\\
&\quad\ \mathsf{res}(n) = \ \dagger\ \mathsf{res}(n+1);\\
&\quad\ \texttt{end}\\
[\![\backslash\texttt{base\_address}(t:\tau)]\!]^n_{\mathrm{T}} &\triangleq \texttt{let}\ \mathsf{res}(n+1) : \tau\ \texttt{in}\\
&\quad\ [\![t]\!]^{n+1}_{\mathrm{T}}\\
&\quad\ \mathsf{res}(n) = \texttt{base\_address}(\mathsf{res}(n+1));\\
&\quad\ \texttt{end}\\
[\![\backslash\texttt{offset}(t)]\!]^n_{\mathrm{T}} &\triangleq \texttt{let}\ \mathsf{res}(n+1) : \texttt{int64}\ \texttt{in}\\
&\quad\ [\![t]\!]^{n+1}_{\mathrm{T}}\\
&\quad\ \mathsf{res}(n) = \texttt{offset}(\mathsf{res}(n+1));\\
&\quad\ \texttt{end}\\
[\![\backslash\texttt{block\_length}(t)]\!]^n_{\mathrm{T}} &\triangleq \texttt{let}\ \mathsf{res}(n+1) : \texttt{int64}\ \texttt{in}\\
&\quad\ [\![t]\!]^{n+1}_{\mathrm{T}}\\
&\quad\ \mathsf{res}(n) = \texttt{block\_length}(\mathsf{res}(n+1));\\
&\quad\ \texttt{end}
\end{aligned}
$$

FIGURE 3.11 – Traduction des termes.

sant des termes, il n'y a pas à prendre en compte de sémantique paresseuse pour leur évaluation. Les deux calculs récursifs peuvent donc être effectués séquentiellement, sans faire appel à une conditionnelle.

3.5 État de l'art

L'idée d'assurer la sûreté mémoire de programmes C par instrumentation n'est pas nouvelle [Lue+06], mais le processus d'instrumentation lui-même a été relativement peu étudié.

Les problèmes de gestion de la mémoire en C ont été approchés par une variété d'outils, généralement connus sous la dénomination de débogueurs mémoire (*memory debuggers* en anglais). Ces outils sont généralement guidés par les besoins pratiques des programmeurs pour répondre, ce qui se traduit par un certain pragmatisme dans leur approche. Ils permettent ainsi typiquement être utilisés de manière entièrement automatique (sans devoir an-

noter manuellement le programme), et apportent une attention particulière aux performances. MemCheck [NS07], outil de la plateforme Valgrind [NS03], DrMemory [BZ11] et AddressSanitizer [Ser+12], peuvent être considérés comme représentatifs de cette catégorie.

Les approches formelles de la vérification d'assertion à l'exécution (outre les travaux autour d'E-ACSL [Sig21] présentés plus loin), concernent principalement le langage de spécification, JML (*Java Modeling Language*), qui a inspiré la conception d'ACSL. La thèse de CHEON [Che03] décrit ainsi la conception et la mise en œuvre d'un vérificateur d'assertions à l'exécution pour JML, précisant au passage certains aspects de la sémantique de JML. La formalisation a ensuite été poussée jusqu'au niveau de la sémantique mécanisée (dans l'assistant de preuve Coq) par LEHNER, au cours de sa thèse [Leh11].

Dans sa thèse sur l'analyse hybride (statique et dynamique) de flux d'information [Ass15], ASSAF emploie une mémoire annexe pour enregistrer des méta-données sur la mémoire principale, sur laquelle s'appuie un moniteur. L'idée peut être rapprochée de celle de mémoire d'observation introduite dans la section 3.3.2, bien que les données enregistrées dans un cas et dans l'autre soient de nature différente.

Plus récemment, et dans un contexte d'application plus proche, BENJAMIN, RIDOUX et SIGNOLES [BRS22] ont proposé une formalisation de l'arithmétique entière d'E-ACSL (typage et génération de code).

Chapitre 4

Propriétés sémantiques de la transformation

Ayant défini au chapitre précédent une transformation de programme pour la vérification à l'exécution de propriétés mémoires, nous nous tournons à présent vers l'étude des propriétés de cette transformation. La principale est une propriété de préservation sémantique entre programme source et programme cible, dont l'idée générale est présentée dans la première section. Cette propriété de haut niveau est exprimée en combinant trois propriétés de plus bas niveau, qui permettent de décrire des relations entre deux états d'exécution du programme (états de la mémoire, par exemple). La deuxième section présente ces trois relations tour à tour, accompagnées de propriétés utiles les concernant. Dans la troisième section, on se sert des relations précédemment introduites pour énoncer trois théorèmes portant respectivement sur la traduction des instructions, prédicats, et termes. Leurs preuves sont rassemblées dans la quatrième section. Une discussion sur l'état de l'art conclut le chapitre.

4.1 Aperçu, conventions et notations

Dans ce chapitre, on sera fréquemment amené à énoncer des propositions faisant intervenir à la fois des jugements d'évaluation du langage source et du langage cible. Afin de faciliter leur distinction, les objets formels (mémoire d'exécution, environnement, blocs, variables…) relevant du langage source seront surmontés d'un chapeau, tandis que ceux du langage cible conserveront leur notation d'origine. Ainsi, si dans un énoncé figurent un environnement noté $\widehat{E}$ et un autre noté E, on pourra sans crainte de confusion supposer que le premier est employé dans l'évaluation d'un programme source, et le second dans celle d'un programme cible.

La transformation étudiée vise en particulier à préserver la sémantique du programme, c'est-à-dire, informellement, à faire en sorte que « le programme traduit se comporte de la même façon que le programme source ». Pour être en mesure de formuler un théorème exprimant une telle propriété, il faut donc au préalable introduire ce que l'on appelle le comportement d'un programme et ce que signifie pour de tels comportements le fait d'être similaires.

Une propriété de préservation sémantique prendra souvent la forme d'un énoncé tel que : si l'évaluation du programme P dans l'état $\widehat{S_1}$ aboutit à l'état $\widehat{S_2}$, et $\widehat{S_1}$ est lié à S_1 par une certaine relation $\mathscr{R}$, alors l'évaluation de $[\![P]\!]$ dans l'état S_1 aboutit à un état S_2 satisfaisant avec $\widehat{S_2}$ la relation $\mathscr{R}$, $[\![\cdot]\!]$ étant la transformation étudiée. L'énoncé formel d'une telle propriété peut ainsi prendre la forme suivante :

$$\forall \widehat{S_1}, \widehat{S_2}, S_1, P, \left\{ \begin{array}{l} \widehat{S_1} \vDash P \Downarrow \widehat{S_2} \\ \widehat{S_1} \mathrel{\mathscr{R}} S_1 \end{array} \right. \implies \exists S_2, \left\{ \begin{array}{l} S_1 \vDash [\![P]\!] \Downarrow S_2 \\ \widehat{S_2} \mathrel{\mathscr{R}} S_2. \end{array} \right.$$

Cette idée peut également être représentée par un diagramme de commutation comme celui de la figure 4.1.

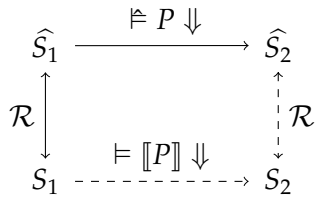

FIGURE 4.1 – Diagramme de commutation d'une propriété de préservation sémantique. Les flèches en trait plein représentent les relations en hypothèse de la propriété, tandis que celles en pointillés représentent les relations obtenues en conclusion.

Appliquer cette idée générale à la transformation étudiée nécessite une légère adaptation. En effet, l'évaluation d'une instruction source a lieu dans un *contexte* composé d'un environnement et d'une mémoire d'exécution, tandis que le résultat de l'évaluation est une mémoire d'exécution seule. De la même façon, l'évaluation des programmes du langage cible a lieu dans un contexte formé d'un environnement, d'une mémoire d'exécution et d'une mémoire d'observation ; le résultat de l'évaluation, cependant, ne comporte qu'une mémoire d'exécution et une mémoire d'observation, mais pas d'environnement.

Par conséquent, que ce soit dans le langage source ou le langage cible, le résultat d'une évaluation n'a pas le même type que son contexte, qui comporte un environnement en plus. Toutefois, si l'environnement n'apparaît pas formellement dans le résultat d'une évaluation, la sémantique de nos langages incite à considérer qu'il y est tout de même implicitement présent, bien qu'inchangé. Considérons, par exemple, l'évaluation des séquences d'instructions du langage source, définie par la règle S-SEQ.

$$\text{S-SEQ}\ \frac{\widehat{E}, \widehat{M_1} \vDash_{\text{s}} s_1 \Downarrow \widehat{M_2} \qquad \widehat{E}, \widehat{M_2} \vDash_{\text{s}} s_2 \Downarrow \widehat{M_3}}{\widehat{E}, \widehat{M_1} \vDash_{\text{s}} s_1\ s_2 \Downarrow \widehat{M_3}}$$

L'évaluation de la première instruction s_1, dans l'environnement $\widehat{E}$, aboutit à la mémoire (d'exécution) $\widehat{M_2}$, laquelle est ensuite utilisée dans le contexte d'évaluation de s_2 aux côtés du même environnement $\widehat{E}$ que pour s_1. L'environnement est donc préservé d'une évaluation à l'autre.

En fait, la seule règle d'évaluation faisant varier l'environnement est celle qui décrit l'introduction d'une variable locale, S-LET.

$$\text{S-LET}\ \frac{\widehat{E_2}, \widehat{M_2} = \mathsf{alloc\_var}(\widehat{E_1}, \widehat{M_1}, x, \kappa) \qquad \widehat{E_2}, \widehat{M_2} \vDash_{\text{s}} s \Downarrow \widehat{M_3} \qquad \widehat{M_4} = \mathsf{dealloc\_var}(\widehat{E_2}, \widehat{M_3}, x)}{\widehat{E_1}, \widehat{M_1} \vDash_{\text{s}} \texttt{let}\ x\ :\kappa\ \texttt{in}\ s\ \texttt{end} \Downarrow \widehat{M_4}}$$

L'évaluation du corps du `let`, à savoir celle de l'instruction s, a lieu dans l'environnement $\widehat{E_2}$ obtenu à partir de $\widehat{E_1}$ en y ajoutant la nouvelle variable x. Cependant, cet ajout n'intervient que pour évaluer s, et est sans effet en dehors. En particulier, toute instruction qui suivrait le `let` serait évaluée dans l'environnement initial $\widehat{E_1}$.

Par conséquent, pour définir la relation préservée par évaluation dans notre propriété de préservation sémantique, on peut naturellement considérer que l'environnement figurant dans le contexte d'évaluation des programmes est reproduit, inchangé, dans le résultat de l'évaluation. On pourra donc représenter le théorème étudié dans ce chapitre par le diagramme de la figure 4.2.

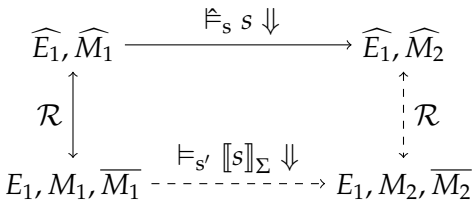

FIGURE 4.2 – Diagramme de commutation adapté aux langages d'étude.

Il reste à présent à caractériser l'élément principal du théorème : la relation $\mathcal{R}$, qui définit la correspondance attendue entre les contextes sources et cibles. Cette relation peut être conceptuellement décomposée en trois notions distinctes, présentées dans la section suivante.

4.2 Relations sur les états mémoire

On définit dans cette section quatre formes de relation sur des états mémoire, sur lesquelles sont fondés les invariants intervenant dans les théorèmes de préservation sémantique. La première, appelée *isomorphisme*, sert à exprimer la similitude entre deux mémoires d'exécution ayant le même contenu, à permutation près de leurs (identifiants de) blocs. La deuxième est un cas particulier de la première, dans lequel deux mémoires sont indistinguables vis-à-vis des opérations mémoires ; elle est appelée *équivalence*, et permet, en pratique, de raisonner comme on le ferait avec une égalité. La troisième relation définit les conditions sous lesquelles on considère qu'une mémoire d'observation rend adéquatement compte de l'état d'une mémoire d'exécution ; elle est donc appelée *représentation*. La quatrième et dernière exprime l'*extension* d'une mémoire d'exécution en une autre possédant « plus de contenu ».

Tout comme notre langage d'étude et son modèle mémoire, les relations définies dans cette section sont pour partie inspirées des travaux autour de CompCert. Les relations d'isomorphisme et d'extension sur les états mémoires, notamment, sont des spécialisations d'une notion générale de *plongement* (en anglais *embedding*) définie pour le modèle mémoire de CompCert [LB08].

4.2.1 Isomorphisme entre mémoires d'exécution

Cette relation assure le lien, dans le théorème de préservation sémantique, entre la mémoire d'exécution du programme source et celle du programme cible.

On dit que deux mémoires sont isomorphes si elles ont « le même contenu ». Plus précisément, il s'agit d'exprimer le fait qu'elles ne présentent pas de différence observable (via l'opération load), à permutation des (identifiants de) blocs près. En effet, dans le modèle mémoire tel qu'il est défini en section 2.2, les blocs sont des identifiants abstraits.

Définition 6 (permutation de blocs)**.**
Une permutation de blocs est une fonction σ : **block** $\to$ **block** bijective.

On note *id* la permutation identité, σ^{-1} la permutation inverse d'une permutation σ, et $(x \leftrightarrow y)$ la transposition de x et y, c'est-à-dire la permutation qui échange x et y en laissant fixes les autres blocs.

On désignera parfois par le seul terme *permutation* les permutations de blocs. À toute permutation de blocs, on peut faire correspondre une fonction sur les valeurs, qui reporte l'action de la permutation sur les blocs des pointeurs, et laisse inchangées les autres valeurs.

Définition 7 (permutation de blocs appliquée aux valeurs)**.**
Soit une permutation de blocs σ. On appelle permutation de valeurs engendrée par σ la fonction $\dot{\sigma}$: **value** $\to$ **value** définie par :

$$\begin{array}{rcl} \dot{\sigma}(\mathsf{Int}(n)) & \triangleq & \mathsf{Int}(n) \\ \dot{\sigma}(\mathsf{Undef}) & \triangleq & \mathsf{Undef} \\ \dot{\sigma}(\mathsf{Ptr}(b,\delta)) & \triangleq & \mathsf{Ptr}(\sigma(b),\delta). \end{array}$$

On peut maintenant définir la notion de mémoires isomorphes. Deux mémoires sont isomorphes lorsque l'application d'une permutation bien choisie préserve leurs blocs valides, les longueurs de ces blocs, et les valeurs qu'ils contiennent.

Définition 8 (mémoires isomorphes)**.**
On dit que deux mémoires M_1 et M_2 sont isomorphes, ce que l'on note $M_1 \sim M_2$, s'il existe une permutation de blocs σ telle que :

$$\begin{cases} \forall b,\ M_1 \vDash b \iff M_2 \vDash \sigma(b) \\ \forall b,\ M_1 \vDash b \implies \mathsf{length}(M_1, b) = \mathsf{length}(M_2, \sigma(b)) \\ \forall \kappa, b, \delta,\ M_1 \vDash \kappa\,@\,b, \delta \implies \mathsf{load}(\kappa, M_2, \sigma(b), \delta) = \lfloor \dot{\sigma}(v) \rfloor \\ \quad \text{avec } \lfloor v \rfloor = \mathsf{load}(\kappa, M_1, b, \delta) \end{cases}$$

Le modèle mémoire présenté en section 2.2.2 est une spécification partielle, qui donne en particulier peu d'informations sur le comportement de la mémoire lors d'opérations « illicites » telles qu'une lecture à un emplacement invalide. Par conséquent, notre définition d'isomorphisme doit, elle aussi, être limitée à la description des opérations « licites », faute de quoi elle serait trop forte pour être applicable en pratique. C'est pourquoi les deuxième et troisième propriétés de la définition, spécifiant respectivement les longueurs des blocs et leurs contenus, sont conditionnées par des hypothèses de « fonctionnement normal » de la mémoire.

La troisième propriété de la définition suppose que la lecture $\mathsf{load}(\kappa, M_1, b, \delta)$ réussit, et renvoie une certaine valeur v. Cette supposition est justifiée par l'axiome 2.18 et la condition d'accès valide $M_1 \vDash \kappa\,@\,b, \delta$.

Il sera fréquemment nécessaire de manipuler explicitement une substitution σ – non nécessairement unique – associée à deux mémoires isomorphes. On la notera alors en indice du symbole $\sim$, par exemple ainsi : $M \sim_\sigma M'$.

On définit aussi un isomorphisme entre environnements de manière à pouvoir ensuite étendre cette notion aux contextes d'évaluation, ouvrant ainsi la voie aux raisonnements sémantiques.

Définition 9 (environnements isomorphes)**.**
On dit que deux environnements E_1 et E_2 sont isomorphes, ce que l'on note $E_1 \sim E_2$, s'il existe une permutation de blocs σ telle que :

$$\forall x, E_2(x) = \begin{cases} \lfloor \sigma(b) \rfloor & \text{si } E_1(x) = \lfloor b \rfloor \\ \varepsilon & \text{si } E_1(x) = \varepsilon. \end{cases}$$

On étend la définition aux contextes d'évaluation, couples formés d'un environnement et d'une mémoire d'exécution. La définition impose, par cohérence, que la même permutation soit valide à la fois pour les environnements et les mémoires.

Définition 10.
Un contexte (E_1, M_1) est dit isomorphe à un second contexte (E_2, M_2) s'il existe une permutation σ telle que $E_1 \sim_\sigma E_2$ et $M_1 \sim_\sigma M_2$.

On note indifféremment les couples avec parenthèses, c'est-à-dire $(E_1, M_1) \sim (E_2, M_2)$, ou sans : $E_1, M_1 \sim E_2, M_2$. En outre, on fera apparaître la permutation σ en indice du symbole $\sim$ ou non, selon son utilité dans le contexte.

Étant donné que la relation d'isomorphisme traduit l'idée d'états mémoires ayant le même contenu, on peut s'attendre à ce qu'elle soit préservée lorsque ces états mémoires subissent des opérations « équivalentes », pour une certaine définition de ce qualificatif. La série de propriétés suivantes exprime cette compatibilité de la relation avec les opérations mémoire.

Propriété 4.1.
Deux mémoires isomorphes ont les mêmes accès valides à permutation près.

$$\forall M, M',\ M \sim_\sigma M' \implies \forall \kappa, b, \delta,\ \big(M \vDash \kappa @ b, \delta \iff M' \vDash \kappa @ \sigma(b), \delta\big)$$

Démonstration. Cette propriété découle directement de la définition 8 (en particulier des deux premières conditions dont elle est formée), en conjonction avec la définition 1. □

Propriété 4.2.
Les mémoires issues d'allocations de blocs d'une taille donnée dans deux mémoires isomorphes sont isomorphes.

$$\forall M_1, M_1', M_2, M_2', b_2, b_2', \sigma_1, n \quad \begin{cases} M_1 \sim_{\sigma_1} M_1' \\ \mathsf{alloc}(M_1, n) = (b_2, M_2) \\ \mathsf{alloc}(M_1', n) = (b_2', M_2') \end{cases} \implies M_2 \sim_{\sigma_2} M_2'$$

où $\sigma_2 \triangleq (\sigma_1(b_2) \leftrightarrow b_2') \circ \sigma_1 = \sigma_1 \circ (b_2 \leftrightarrow \sigma_1^{-1}(b_2'))$.

Démonstration. Commençons par établir une caractérisation pratique de σ_2, conséquence directe de sa définition comme composition de σ_1 et d'une transposition bien choisie :

$$\forall b, \sigma_2(b) = \begin{cases} b_2' & \text{si } b = b_2 \\ \sigma_1(b_2) & \text{si } b = \sigma_1^{-1}(b_2') \\ \sigma_1(b) & \text{sinon.} \end{cases}$$

On peut par ailleurs noter que $\sigma_1^{-1}(b_2')$ est nécessairement invalide dans M_1, puisque b_2' est invalide dans M_1' d'après l'axiome 2.1 et $M_1 \sim_{\sigma_1} M_1'$. Par conséquent, lorsque l'on considèrera un bloc valide dans M_1, on pourra d'office exclure le cas où ce bloc serait $\sigma_1^{-1}(b_2')$. De la même façon, $\sigma_1(b_2)$ est invalide dans M_1.

Soit b un bloc. Montrons que b est valide dans M_2 si et seulement si $\sigma_2(b)$ est valide dans M_2'. Si $b = b_2$, alors l'axiome 2.2 donne $M_2 \vDash b_2$ et $M_2' \vDash b_2'$ et l'équivalence est vérifiée. Si $b \neq b_2$, on a les trois équivalences suivantes :

$$M_2 \vDash b \iff M_1 \vDash b \iff M_1' \vDash \sigma_2(b) \iff M_2' \vDash \sigma_2(b).$$

On obtient la première (respectivement, la troisième) équivalence par application de l'axiome 2.2 avec l'hypothèse $b \neq b_2$ (respectivement $\sigma_2(b) \neq b_2'$, par bijectivité des permutations et caractérisation de σ_2) ; la deuxième équivalence est une conséquence de l'hypothèse $M_1 \sim_{\sigma_1} M_1'$.

Montrons maintenant que si b est valide dans M_2, alors $\mathsf{length}(M_2, b) = \mathsf{length}(M_2', \sigma_2(b))$. Si $b = b_2$, les blocs considérés ont la même longueur n en vertu de l'axiome 2.7. Si $b \neq b_2$, on a les égalités suivantes :

$$\mathsf{length}(M_2, b) = \mathsf{length}(M_1, b) = \mathsf{length}(M_1', \sigma_2(b)) = \mathsf{length}(M_2', \sigma_2(b)).$$

La première (respectivement, la troisième) égalité est justifiée par l'axiome 2.8 avec l'hypothèse $b \neq b_2$ (respectivement $\sigma_2(b) \neq b_2'$) ; la deuxième est une conséquence de l'hypothèse $M_1 \sim_{\sigma_1} M_1'$.

Il reste à montrer que les lectures valides renvoient les mêmes valeurs, à application de $\dot{\sigma_2}$ près. Soient donc b, κ et δ tels que $M_2 \vDash \kappa @ b, \delta$. Si $b = b_2$, les lectures dans b dans M_2 sont indéfinies (axiome 2.11), de mêmes que les lectures dans $\sigma_2(b) = b'_2$ dans M'_2. Si $b \neq b_2$ alors d'après l'axiome 2.12, $\mathsf{load}(\kappa, M_2, b, \delta) = \mathsf{load}(\kappa, M_1, b, \delta)$ et, de même,

$$\mathsf{load}(\kappa, M'_2, \sigma_2(b), \delta) = \mathsf{load}(\kappa, M'_1, \sigma_2(b), \delta).$$

Si $\lfloor v \rfloor = \mathsf{load}(\kappa, M_1, b, \delta)$, alors d'après l'hypothèse $M_1 \sim_{\sigma_1} M'_1$, on a $\mathsf{load}(\kappa, M'_1, \sigma_1(b), \delta) = \lfloor \dot{\sigma_1}(v) \rfloor$. Donc $\mathsf{load}(\kappa, M'_2, \sigma_2(b), \delta) = \lfloor \dot{\sigma_1}(v) \rfloor$. Il suffit donc de montrer que $\dot{\sigma_2}(v) = \dot{\sigma_1}(v)$.

Par définition de σ_2, c'est vrai tout le temps sauf si $v = \mathsf{Ptr}(b_2, \dots)$ ou $v = \mathsf{Ptr}(\sigma^{-1}(b'_2), \dots)$. Autrement dit, cela ne peut arriver que si M_1 contient un pointeur vers un bloc qui n'a jamais été alloué, ce qui est exclu par l'axiome 2.19. □

Propriété 4.3.
Soient M_1 et M'_1 deux mémoires isomorphes, σ une permutation associée à cet isomorphisme, et b un bloc valide dans M_1. Alors, la déallocation de b dans M_1 réussit, celle de $\sigma(b)$ dans M'_1 réussit également et les mémoires résultantes sont encore isomorphes.

$$\forall M_1, M'_1, b, \sigma,$$
$$\begin{cases} M_1 \sim_\sigma M'_1 \\ M_1 \vDash b \end{cases} \implies \begin{cases} \exists M_2,\ \mathsf{free}(M_1, b) = \lfloor M_2 \rfloor \\ \exists M'_2,\ \mathsf{free}(M'_1, \sigma(b)) = \lfloor M'_2 \rfloor \\ M_2 \sim_\sigma M'_2 \end{cases}$$

Démonstration. Par définition de $M_1 \sim_\sigma M'_1$, puisque b est valide dans M_1, $\sigma(b)$ est valide dans M'_1. Les existences de M_2 et de M'_2 sont donc assurées par l'axiome 2.5. Il reste à montrer que $M_2 \sim_\sigma M'_2$.

Soit b' un bloc. Montrons que $M_2 \vDash b' \iff M'_2 \vDash \sigma(b')$. Si $b' = b$, alors, d'après l' axiome 2.4, b' n'est pas valide dans M_2 et, par le même argument, $\sigma(b')$ n'est pas non plus valide dans M'_2. Si $b' \neq b$, alors, d'après l'axiome 2.3, b' a la même validité dans M_1 que dans M_2 : $M_1 \vDash b' \iff M_2 \vDash b'$. Par le même argument, $\sigma(b')$ a la même validité dans M'_1 que dans M'_2 : $M'_1 \vDash \sigma(b') \iff M'_2 \vDash \sigma(b')$. Or $M_1 \sim_\sigma M'_1$, donc $M_1 \vDash b' \iff M'_1 \vDash \sigma(b')$, ce qui permet d'établir l'équivalence recherchée : $M_2 \vDash b' \iff M'_2 \vDash \sigma(b')$.

Montrons maintenant que si b' est valide dans M_2, alors $\mathsf{length}(M_2, b') = \mathsf{length}(M'_2, \sigma(b'))$. Premièrement, puisque b' est valide dans M_2, en particulier $b' \neq b$ (axiome 2.4) ce qui permet d'appliquer l'axiome 2.9 pour obtenir $\mathsf{length}(M_2, b') = \mathsf{length}(M_1, b')$. Deuxièmement, d'après la conclusion du paragraphe précédent, on a en outre $M'_2 \vDash \sigma(b')$, et on peut donc appliquer le même raisonnement pour obtenir $\mathsf{length}(M'_2, \sigma(b')) = \mathsf{length}(M'_1, \sigma(b'))$. Troisièmement, puisque $M_1 \sim_\sigma M'_1$, $\mathsf{length}(M_1, b') = \mathsf{length}(M'_1, \sigma(b'))$. On conclut par combinaison des trois égalités démontrées.

Enfin, soient κ et δ tels que $M_2 \vDash \kappa @ b', \delta$. Comme précédemment, on déduit de $M_2 \vDash b'$ que $b' \neq b$. On a donc, d'après l'axiome 2.13, $\mathsf{load}(\kappa, M_2, b', \delta) = \mathsf{load}(\kappa, M_1, b', \delta)$, et de même $\mathsf{load}(\kappa, M'_2, \sigma(b'), \delta) = \mathsf{load}(\kappa, M'_1, \sigma(b'), \delta)$. Par ailleurs, d'après axiome 2.3 b' est valide dans M_1, et d'après axiome 2.9 $\mathsf{length}(M_2, b') = \mathsf{length}(M_1, b')$. On en déduit donc que $M_1 \vDash \kappa @ b', \delta$. De même, $M'_1 \vDash \kappa @ \sigma(b'), \delta$. On peut alors appliquer la troisième propriété de la définition de $\sim$ pour conclure. □

Propriété 4.4.
Soient M_1 et M'_1 deux mémoires isomorphes, σ une permutation associée à cet isomorphisme, et b un bloc. Si l'écriture de v dans M_1 réussit, celle de $\dot{\sigma}(v)$ dans M'_1 réussit également et les mémoires résultantes sont encore isomorphes. Si l'écriture de v dans M_1 échoue, celle de $\dot{\sigma}(v)$

dans M_1' échoue aussi.

$$\forall \kappa, M_1, M_1', b, \delta, v, \sigma, \text{ si } M_1 \sim_\sigma M_1', \text{ alors} \begin{cases} \text{si } \exists M_2, \ \mathsf{store}(\kappa, M_1, b, \delta, v) = \lfloor M_2 \rfloor, \text{ alors} \\ \quad \exists M_2', \mathsf{store}(\kappa, M_1', \sigma(b), \delta, \dot{\sigma}(v)) = \lfloor M_2' \rfloor \wedge M_2 \sim_\sigma M_2' \\ \text{et, si } \mathsf{store}(\kappa, M_1, b, \delta, v) = \varepsilon, \text{ alors } \mathsf{store}(\kappa, M_1', \sigma(b), \delta, \dot{\sigma}(v)) = \varepsilon \end{cases}$$

Démonstration. Nous allons tout d'abord établir que l'écriture dans M_1' réussit si et seulement si celle dans M_1 réussit également. En effet, d'après la propriété 4.1, les accès valides sont les mêmes à σ près, et d'après l'axiome 2.17, l'accès valide est une condition nécessaire et suffisante à la réussite d'une écriture.

On peut dès lors supposer l'existence des états mémoire M_2 et M_2' tels qu'énoncés, et démontrer qu'ils sont isomorphes.

L'opération d'écriture n'affecte pas la validité des blocs (axiome 2.6), donc, pour tout bloc b', l'équivalence $M_1 \vDash b' \iff M_1' \vDash \sigma(b')$ (vraie par hypothèse car $M_1 \sim_\sigma M_1'$) reste vraie pour les états résultant des opérations d'écriture, c'est-à-dire $M_2 \vDash b' \iff M_2' \vDash \sigma(b')$.

De même, les longueurs de blocs sont préservées par les opérations d'écriture (axiome 2.10), donc la propriété d'égalité des longueurs entre les blocs de M_1 et M_1' est préservée : $\forall b', \ \mathsf{length}(M_2, b') = \mathsf{length}(M_2', \sigma(b'))$.

De ces deux premiers résultats découle l'équivalence des accès valides : $\forall \kappa, b', \delta', \ M_2 \vDash \kappa' @ b', \delta' \iff M_2' \vDash \kappa' @ \sigma(b'), \delta'$.

Il reste donc à montrer la troisième partie de la définition d'isomorphisme. On se donne donc κ', b' et δ' tels que $M_2 \vDash \kappa' @ b', \delta'$, et v le résultat de la lecture en (b', δ') avec le type κ' : $\mathsf{load}(\kappa', M_2, b', \delta') = \lfloor v \rfloor$. Si $b' \neq b$, on a d'après l'axiome 2.15 $\mathsf{load}(\kappa', M_2, b', \delta') = \mathsf{load}(\kappa', M_1, b', \delta')$, c'est-à-dire $\mathsf{load}(\kappa', M_1, b', \delta') = \lfloor v \rfloor$. De cette égalité, on déduit par application de l'hypothèse $M_1 \sim_\sigma M_1'$ l'égalité suivante : $\mathsf{load}(\kappa', M_1', \sigma(b'), \delta') = \lfloor \dot{\sigma}(v) \rfloor$. Or $\sigma(b') = \sigma(b)$, ce qui permet d'appliquer à nouveau l'axiome 2.15 afin d'obtenir $\mathsf{load}(\kappa', M_1', \sigma(b'), \delta') = \mathsf{load}(\kappa', M_2', \sigma(b'), \delta')$. Finalement, la combinaison des égalités précédentes permet de conclure : $\mathsf{load}(\kappa', M_2', \sigma(b'), \delta') = \lfloor \dot{\sigma}(v) \rfloor$.

On traite de la même manière le cas de la lecture dans le bloc b (respectivement $\sigma(b)$), mais sur une plage d'indices disjointe de celle concernée par l'opération d'écriture, c'est-à-dire lorsque $\delta' + \mathsf{sizeof}(\kappa') \leq \delta$, ou $\delta + \mathsf{sizeof}(\kappa) \leq \delta'$.

Il ne reste donc plus que le cas d'une lecture dans le même bloc que l'écriture, et sur une plage d'indice partiellement commune. Or d'après l'axiome 2.16, cela revient nécessairement à supposer que $\kappa' = \kappa$, et donc en particulier $\delta' = \delta$. On applique alors l'axiome 2.14, et on obtient ainsi : $\mathsf{load}(\kappa', M_2, b', \delta') = \lfloor v \rfloor$. Par le même raisonnement, $\mathsf{load}(\kappa', M_2', \sigma(b'), \delta') = \lfloor \dot{\sigma}(v) \rfloor$, ce qui conclut la preuve. □

Une application sémantique directe de la notion d'isomorphisme de contexte concerne l'évaluation d'expression dans des contextes isomorphes : une même expression évaluée dans deux contextes isomorphes donne le même résultat, à permutation près.

Lemme 1.
Soient C et C' deux contextes d'évaluation isomorphes par une permutation σ, et e une expression. Si l'évaluation de e dans C donne une valeur v, alors son évaluation dans C' donne la valeur $\dot{\sigma}(v)$.

$$\forall C, C', e, v, \ \left(C \vDash_{\mathrm{e}} e \Downarrow v \ \wedge \ C \sim_\sigma C'\right) \implies C' \vDash_{\mathrm{e}} e \Downarrow \dot{\sigma}(v)$$

En outre, si l'évaluation de e en valeur gauche dans C donne un emplacement mémoire (b, δ), alors son évaluation en valeur gauche dans C' donne $(\sigma(b), \delta)$.

$$\forall C, C', e, v, \ \left(C \vDash_{\mathrm{lv}} e \Downarrow b, \delta \ \wedge \ C \sim_\sigma C'\right) \implies C' \vDash_{\mathrm{lv}} e \Downarrow \sigma(b), \delta$$

Démonstration. On se donne deux contextes $C = E, M$ et $C' = E', M'$ isomorphes via une permutation σ, et une expression e. En raison de la définition des relations d'évaluation des expressions en valeur et en valeur gauche (section 2.3), les deux énoncés du lemme sont à démontrer simultanément, par induction mutuelle sur l'évaluation de e.

Si e est une constante entière n, elle est évaluée à la même valeur $\mathsf{Int}(n)$ indépendamment du contexte. En particulier, on a bien $\mathsf{Int}(n) = \dot{\sigma}(\mathsf{Int}(n))$, puisque $\dot{\sigma}$ laisse inchangées les valeurs entières.

Si e est une variable x évaluée en valeur gauche, son évaluation a la forme :

$$\text{LV-VAR}\ \frac{E(x) = b}{E, M \vDash_{\text{lv}} x \Downarrow b, 0}$$

Par définition de $E \sim_\sigma E'$, $E'(x) = \sigma(E(x)) = \sigma(b)$. On peut donc écrire :

$$\text{LV-VAR}\ \frac{E'(x) = \sigma(b)}{E', M' \vDash_{\text{lv}} x \Downarrow \sigma(b), 0}$$

Dans le cas du déréférencement d'un pointeur, l'évaluation a la forme ci-dessous.

$$\text{LV-DEREF}\ \frac{E, M \vDash_{\text{e}} p \Downarrow \mathsf{Ptr}(b, \delta)}{E, M \vDash_{\text{lv}} *p \Downarrow b, \delta}$$

Ici aussi, on obtient le résultat attendu par application de directe de l'hypothèse d'induction : $E, M \vDash_{\text{e}} p \Downarrow \mathsf{Ptr}(b, \delta)$ donc $E', M' \vDash_{\text{e}} p \Downarrow \mathsf{Ptr}(\sigma(b), \delta)$, donc $E', M' \vDash_{\text{lv}} *p \Downarrow \sigma(b), \delta$.

Si e est une prise d'adresse, l'évaluation est de la forme :

$$\text{E-ADDR}\ \frac{E, M \vDash_{\text{lv}} e_1 \Downarrow b, \delta}{E, M \vDash_{\text{e}} \&e_1 \Downarrow \mathsf{Ptr}(b, \delta)}$$

En appliquant l'hypothèse d'induction à la prémisse $E, M \vDash_{\text{lv}} e_1 \Downarrow b, \delta$ on obtient l'évaluation $E', M' \vDash_{\text{lv}} e_1 \Downarrow \sigma(b), \delta$, ce qui permet de former le jugement d'évaluation suivant :

$$\text{E-ADDR}\ \frac{E', M' \vDash_{\text{lv}} e_1 \Downarrow \sigma(b), \delta}{E', M' \vDash_{\text{e}} \&e_1 \Downarrow \mathsf{Ptr}(\sigma(b), \delta)}$$

où $\mathsf{Ptr}(\sigma(b), \delta) = \dot{\sigma}(\mathsf{Ptr}(b, \delta))$ par définition de $\dot{\sigma}$.

Le cas de l'évaluation d'une valeur gauche fait appel à la définition d'isomorphisme entre mémoires.

$$\text{E-LVAL}\ \frac{E, M \vDash_{\text{lv}} e : \tau \Downarrow b, \delta \qquad \mathsf{load}(\mathsf{mtype}(\tau), M, b, \delta) = \lfloor v \rfloor \qquad v \neq \mathsf{Undef}}{E, M \vDash_{\text{e}} e \Downarrow v}$$

Ici, appliquer l'hypothèse d'induction à la première prémisse donne l'évaluation en valeur gauche suivante pour $e : E', M' \vDash_{\text{lv}} e : \tau \Downarrow \sigma(b), \delta$. Par définition de $M \sim_\sigma M'$, en conjonction avec la deuxième prémisse, on a de plus $\mathsf{load}(\mathsf{mtype}(\tau), M', \sigma(b), \delta) = \lfloor \dot{\sigma}(v) \rfloor$. Enfin, $v \neq \mathsf{Undef}$ donc $\dot{\sigma}(v) \neq \mathsf{Undef}$ (conséquence directe de la définition de $\dot{\sigma}$). On peut donc former le jugement suivant :

$$\text{E-LVAL}\ \frac{E', M' \vDash_{\text{lv}} e : \tau \Downarrow \sigma(b), \delta \qquad \mathsf{load}(\mathsf{mtype}(\tau), M, \sigma(b), \delta) = \lfloor \dot{\sigma}(v) \rfloor \qquad \dot{\sigma}(v) \neq \mathsf{Undef}}{E, M \vDash_{\text{e}} e \Downarrow \dot{\sigma}(v)}$$

Concernant les opérateurs, il s'agit dans la plupart des cas de calculs sur des entiers, et le résultat sera donc le même dans les deux évaluations. On ne traite donc ici que le cas de l'addition d'un pointeur et d'un entier.

$$\text{E-BINOP}\ \frac{E, M \vDash_{\text{e}} p : \tau* \Downarrow \mathsf{Ptr}(b, \delta) \qquad E, M \vDash_{\text{e}} e \Downarrow \mathsf{Int}(n) \qquad \mathsf{sem\_binop}(+, \mathsf{Ptr}(b, \delta), \mathsf{Int}(n)) = \lfloor \mathsf{Ptr}(b, \delta + n \cdot \mathsf{sizeof}(\tau)) \rfloor}{E, M \vDash_{\text{e}} p + e : \tau* \Downarrow \mathsf{Ptr}(b, \delta + n \cdot \mathsf{sizeof}(\tau))}$$

En appliquant l'hypothèse d'induction aux deux premières prémisses, on obtient d'une part $E', M' \vDash_{\text{e}} p : \tau* \Downarrow \mathsf{Ptr}(\sigma(b), \delta)$, et d'autre part $E', M' \vDash_{\text{e}} e \Downarrow \mathsf{Int}(n)$. On applique ensuite la définition de sem_binop pour obtenir :

$$\mathsf{sem\_binop}(+, \mathsf{Ptr}(\sigma(b), \delta), \mathsf{Int}(n)) = \lfloor \mathsf{Ptr}(\sigma(b), \delta + n \cdot \mathsf{sizeof}(\tau)) \rfloor,$$

et on peut ainsi établir le jugement d'évaluation souhaité.

$$\frac{E', M' \vDash_{\text{e}} p : \tau* \Downarrow \mathsf{Ptr}(\sigma(b), \delta) \qquad E', M' \vDash_{\text{e}} e \Downarrow \mathsf{Int}(n) \qquad \mathsf{sem\_binop}(+, \mathsf{Ptr}(\sigma(b), \delta), \mathsf{Int}(n)) = \lfloor \mathsf{Ptr}(\sigma(b), \delta + n \cdot \mathsf{sizeof}(\tau)) \rfloor}{E', M' \vDash_{\text{e}} p + e : \tau* \Downarrow \mathsf{Ptr}(\sigma(b), \delta + n \cdot \mathsf{sizeof}(\tau))}$$

□

4.2.2 Équivalence observationnelle entre mémoires d'exécution

L'équivalence observationnelle (ou simplement équivalence) entre deux états mémoire est un cas particulier d'isomorphisme, correspondant à une permutation de blocs associée égale à l'identité. Les deux mémoires ont alors le même comportement lors de toute lecture à accès valide.

Définition 11 (équivalence observationnelle)**.**
Deux mémoires M_1 et M_2 sont dites équivalentes, et notées $M_1 \equiv M_2$, lorsque $M_1 \sim_{\text{id}} M_2$, où id dénote la fonction identité sur l'ensemble des blocs.

L'équivalence est typiquement rencontrée lorsque deux opérations mémoires sont appliquées à un même état de départ dans des ordres différents.

Propriété 4.5 (équivalence par permutation d'une écriture et d'une allocation)**.**
Soient M_0 et M_0' deux mémoires équivalentes : $M_0 \equiv M_0'$. Soient en outre n, κ, b, b', δ, v, M_1, M_1', M_2 et M_2' tels que :

$$\begin{array}{rcl|rcl} b, M_1 & = & \mathsf{alloc}(M_0, n) & \lfloor M_1' \rfloor & = & \mathsf{store}(\kappa, M_0', b', \delta, v) \\ \lfloor M_2 \rfloor & = & \mathsf{store}(\kappa, M_1, b', \delta, v) & b, M_2' & = & \mathsf{alloc}(M_1', n) \end{array}$$

Alors M_2 et M_2' sont encore équivalentes : $M_2 \equiv M_2'$.

Démonstration. Il s'agit de montrer que $M_2 \sim_{\text{id}} M_2'$, sachant que $M_0 \sim_{\text{id}} M_0'$. On procède en trois temps, suivant la définition de la relation d'isomorphisme.

Montrons premièrement que M_2 et M_2' ont les mêmes blocs valides. Soit un bloc b'' valide dans M_2. Si $b'' = b$, alors $M_2 \vDash b''$, d'après les axiomes 2.2 et 2.6. Par ailleurs, $M_2' \vDash b''$ d'après l'axiome 2.2, la validité de b'' est donc la même dans les deux états M_2 et M_2'. Si $b'' = b'$, alors $M_2 \vDash b''$ d'après l'axiome 2.17. De même, $M_1' \vDash b''$, et donc d'après l'axiome 2.2, $M_2' \vDash b''$. Enfin, si $b'' \notin \{b, b'\}$, alors on se ramène à $M_0 \vDash b'' \iff M_0' \vDash b''$ via les axiomes 2.2 et 2.6.

Deuxièmement, montrons que les blocs valides dans M_2 et M_2' ont les mêmes longueurs. Pour un bloc $b'' \neq b$ valide dans M_0 et M_0', l'égalité est vraie par hypothèse ; les longueurs de blocs étant préservées par allocation et écriture (axiomes 2.8 et 2.10), elle est encore vraie pour M_2 et M_2'. Si $b'' = b$, l'axiome 2.7 permet d'établir $\mathsf{length}(M_1, b'') = \mathsf{length}(M_2', b'') = n$, et on conclut via l'axiome 2.10.

Troisièmement, montrons que toute lecture avec un accès valide aboutit au même résultat dans M_2 ou dans M_2'. Considérons b'', δ'' et κ'' tels que $M_2 \vDash \kappa''@b'', \delta''$. Si $b'' = b$, alors $\mathsf{load}(\kappa, M_1, b'', \delta'') = \lfloor \mathsf{Undef} \rfloor$ (axiome 2.11), donc $\mathsf{load}(\kappa, M_2, b'', \delta'') = \lfloor \mathsf{Undef} \rfloor$ (axiome 2.15). □

Propriété 4.6 (équivalence par permutation d'une écriture et d'une déallocation)**.**
Soient M_0 et M_0' deux mémoires équivalentes : $M_0 \equiv M_0'$. Soient en outre n, κ, b', δ, v, M_1, M_1', M_2 et M_2' tels que :

$$\begin{array}{rcl|rcl} \lfloor M_1 \rfloor & = & \mathsf{free}(M_0, b) & \lfloor M_1' \rfloor & = & \mathsf{store}(\kappa, M_1', b', \delta, v) \\ \lfloor M_2 \rfloor & = & \mathsf{store}(\kappa, M_1, b', \delta, v) & \lfloor M_2' \rfloor & = & \mathsf{free}(M_1', b) \end{array}$$

Alors M_2 et M_2' sont encore équivalentes : $M_2 \equiv M_2'$.

Propriété 4.7 (équivalence par allocation puis déallocation d'un même bloc)**.**
Soit M_0 une mémoire d'exécution, et b, M_1, n, M_2 tels que :

$$\begin{array}{rcl} b, M_1 & = & \mathsf{alloc}(M_0, n) \\ \lfloor M_2 \rfloor & = & \mathsf{free}(M_1, b) \end{array}$$

Alors $M_0 \equiv M_2$.

Cette propriété est un corollaire d'une propriété plus générale, exprimant la conservation de l'équivalence lorsque des écritures ont eu lieu dans le bloc alloué puis déalloué.

Propriété 4.8 (équivalence par allocation, écritures, déallocation d'un même bloc)**.**
Soit M_0 une mémoire d'exécution.

$$\begin{array}{rcl} b, M_1 & = & \mathsf{alloc}(M_0, n) \\ \lfloor M_2 \rfloor & = & \mathsf{store}(\kappa_1, M_1, b, \delta_1, v_1) \\ & \cdots & \\ \lfloor M_m \rfloor & = & \mathsf{store}(\kappa_{m-1}, M_{m-1}, b, \delta_{m-1}, v_{m-1}) \\ \lfloor M_{m+1} \rfloor & = & \mathsf{free}(M_m, b) \end{array}$$

Alors $M_0 \equiv M_{m+1}$.

Comme l'isomorphisme, l'équivalence est préservée par déallocation et écriture.

Propriété 4.9 (préservation de l'équivalence par déallocation)**.**
Si $M_0 \equiv M_0'$ et $\lfloor M_1 \rfloor = \mathsf{free}(M_0, b)$, alors $\exists M_1',\ \lfloor M_1' \rfloor = \mathsf{free}(M_0', b) \wedge M_1 \equiv M_1'$.

Propriété 4.10 (préservation de l'équivalence par écriture)**.**
Si $M_0 \equiv M_0'$ et $\lfloor M_1 \rfloor = \mathsf{store}(\kappa, M_0, b, \delta, v)$, alors $\exists M_1',\ \lfloor M_1' \rfloor = \mathsf{store}(\kappa, M_0', b, \delta, v) \wedge M_1 \equiv M_1'$.

4.2.3 Représentation d'une mémoire d'exécution par une mémoire d'observation

Le rôle de la mémoire d'observation est d'enregistrer des informations sur la mémoire d'exécution, mais encore faut-il que celles-ci soient exactes. La relation de représentation, définie ci-dessous, lie une mémoire d'observation à une mémoire d'exécution, et assure que les informations de la première reflètent fidèlement l'état de la seconde.

Définition 12 (représentation)**.**
On dit qu'une mémoire d'exécution M est représentée par une mémoire d'observation $\overline{M}$, ce qu'l'on note $M \triangleright \overline{M}$ si les trois conditions suivantes sont satisfaites :

1. tout bloc est valide dans M si et seulement si il est enregistré comme valide dans $\overline{M}$;

2. tout bloc valide a la même longueur dans M et dans $\overline{M}$;
3. pour tout accès valide dans M, la lecture à cet accès dans M renvoie une valeur initialisée (i.e. différente de Undef) si et seulement si les données correspondantes sont enregistrées comme initialisées dans $\overline{M}$.

Propriété 4.11 (représentation des accès valides)**.**
Si $M \triangleright \overline{M}$, alors tout accès mémoire est valide dans M si et seulement si il est déclaré valide dans $\overline{M}$.

$$\forall \kappa, b, \delta, M \vDash \kappa \,@\, b, \delta \iff \mathsf{is\_valid\_access}(\kappa, \overline{M}, b, \delta) = \top$$

Démonstration. Soit M une mémoire représentée par $\overline{M}$, et soient κ un type mémoire, b un bloc, et δ un décalage. D'après la première propriété de la définition, $M \vDash b \iff \mathsf{is\_valid}(\overline{M}, b)$. Par ailleurs, d'après la deuxième propriété, $\mathsf{length}(M, b) = \mathsf{length}(\overline{M}, b)$, donc $\delta + \mathsf{sizeof}(\kappa) \leq \mathsf{length}(M, b) \iff \delta + \mathsf{sizeof}(\kappa) \leq \mathsf{length}(\overline{M}, b)$.

La conclusion découle ensuite des deux définitions d'accès valide (en mémoire d'exécutions et d'observation). □

Propriété 4.12 (allocation et représentation)**.**
Soit une mémoire d'exécution M_1 représentée par une mémoire d'observation $\overline{M_1}$; soit en outre un entier naturel n, et (b, M_2) le résultat de l'allocation d'un bloc de taille n dans M_1. Alors le résultat de l'enregistrement dans $\overline{M_1}$ du bloc b avec la longueur n représente M_2.

$$\forall M_1, \overline{M_1}, M_2, b, n \quad \begin{cases} M_1 \triangleright \overline{M_1} \\ \mathsf{alloc}(M_1, n) = (b, M_2) \end{cases} \implies M_2 \triangleright \mathsf{store\_block}(\overline{M_1}, b, n)$$

Démonstration. Montrons l'une après l'autre les propriétés définissant $M_2 \triangleright \overline{M_2}$, où $\overline{M_2} \triangleq \mathsf{store\_block}(\overline{M_1}, b, n)$.

Soit b' un bloc. On distingue deux cas, selon que b' soit égal à b ou non. Si $b' = b$, alors, d'après l'axiome 2.2, b' est valide dans M_2, et d'après l'axiome 3.1, b' est également enregistré valide dans $\overline{M_2}$. Sinon, la validité de b' dans M_2 est équivalente à sa validité dans M_1 (toujours d'après l'axiome 2.2). Or par hypothèse, les blocs valides de M_1 sont exactement ceux enregistrés comme tels dans $\overline{M_1}$, donc $M_1 \vDash b' \iff \mathsf{is\_valid}(\overline{M_1}, b') = \top$. Enfin, d'après l'axiome 3.2, b' étant distinct de b, il est valide dans $\overline{M_1}$ si et seulement s'il l'est dans $\overline{M_2}$. Les blocs valides de M_2 sont donc exactement ceux enregistrés $\overline{M_2}$.

On suppose maintenant que b' est valide dans M_2. Montrons qu'il a la même longueur dans M_2 et dans $\overline{M_2}$. Si $b' = b$, alors $\mathsf{length}(M_2, b') = n = \mathsf{length}(\overline{M_2}, b)$. Si $b' \neq b$, alors la longueur de b' dans M_2 (respectivement dans $\overline{M_2}$) est égale à sa longueur dans M_1 (respectivement dans $\overline{M_1}$), d'après l'axiome 2.8.

On se donne à présent, outre b', un type mémoire κ et un décalage δ. Montrons que les données en (b', δ) dans M_2 avec le type κ sont initialisées si et seulement si elles sont enregistrées comme telles dans $\overline{M_2}$. Si $b' = b$, les données ne sont initialisées ni dans M_2 (axiome 2.11), ni dans $\overline{M_2}$ (axiome 3.10). Si $b' \neq b$, alors b' est valide dans M_1 (axiome 2.2), et par hypothèse $(\exists v \neq \mathsf{Undef}, \mathsf{load}(\kappa, M_1, b', \delta) = \lfloor v \rfloor) \iff \mathsf{is\_initialized}(\kappa, \overline{M_1}, b', \delta)$. Or $\mathsf{load}(\kappa, M_1, b', \delta) = \mathsf{load}(\kappa, M_2, b', \delta)$ d'après l'axiome 2.12, et d'après l'axiome 3.11 :

$$\mathsf{is\_initialized}(\kappa, \overline{M_1}, b', \delta) = \mathsf{is\_initialized}(\kappa, \overline{M_2}, b', \delta).$$

D'où la conclusion. □

Propriété 4.13.
Soit une mémoire d'exécution M_1 représentée par une mémoire d'observation $\overline{M_1}$; soit en outre

un bloc b. Si le résultat de la déallocation de b dans M_1 est une mémoire M_2, alors le résultat de la suppression de b dans $\overline{M_1}$ représente M_2.

$$\forall M_1, \overline{M_1}, M_2, b \quad \left\{ \begin{array}{l} M_1 \triangleright \overline{M_1} \\ \mathsf{free}(M_1, b) = \lfloor M_2 \rfloor \end{array} \right. \implies M_2 \triangleright \mathsf{delete\_block}(\overline{M_1}, b)$$

Démonstration. On note $\overline{M_2} \triangleq \mathsf{delete\_block}(\overline{M_1}, b)$. Soit b' un bloc, montrons que $M_2 \vDash b' \iff \mathsf{is\_valid}(\overline{M_2}, b') = \top$. Si $b' = b$, alors d'après l'axiome 2.4, $M_2 \nvDash b'$ et d'après l'axiome 3.3 $\mathsf{is\_valid}(\overline{M_2}, b') = \bot$. Si $b' \neq b$, alors d'après l'axiome 2.3, $M_2 \vDash b' \iff M_1 \vDash b'$, et d'après l'axiome 3.4, $\mathsf{is\_valid}(\overline{M_2}, b') = \mathsf{is\_valid}(\overline{M_1}, b')$. L'équivalence est donc démontrée.

On suppose à partir de maintenant que b' est valide dans M_2, ce qui implique en particulier $b' \neq b$ (axiome 3.3) et $\mathsf{is\_valid}(\overline{M_2}, b') = \top$ (conséquence de la proposition précédente). On obtient l'égalité $\mathsf{length}(M_2, b') = \mathsf{length}(\overline{M_2}, b')$ par application des axiomes 2.9 et 3.8.

Soient κ un type mémoire et δ un décalage tels que $M_2 \vDash \kappa @ b', \delta$. Montrons que $(\exists v \neq \mathsf{Undef},\ \mathsf{load}(\kappa, M_2, b', \delta) = \lfloor v \rfloor) \iff \mathsf{is\_initialized}(\kappa, \overline{M_2}, b', \delta) = \top$. Comme $b' \neq b$, par application de l'axiome 2.13, le résultat de la lecture dans M_2 est le même que dans M_1 : $\mathsf{load}(\kappa, M_2, b', \delta) = \mathsf{load}(\kappa, M_1, b', \delta)$. Par ailleurs, d'après l'axiome 3.12, le statut d'initialisation des données dans b' est inchangé : $\mathsf{is\_initialized}(\kappa, \overline{M_2}, b', \delta) = \mathsf{is\_initialized}(\kappa, \overline{M_1}, b', \delta)$. Or, par hypothèse $M_1 \triangleright \overline{M_1}$, par conséquent $(\exists v \neq \mathsf{Undef},\ \mathsf{load}(\kappa, M_1, b', \delta) = \lfloor v \rfloor) \iff \mathsf{is\_initialized}(\kappa, \overline{M_1}, b', \delta) = \top$. □

Propriété 4.14.
Soit une mémoire d'exécution M_1 représentée par une mémoire d'observation $\overline{M_1}$; soient en outre un type mémoire κ, un bloc b, un décalage δ, et une valeur $v \neq \mathsf{Undef}$. Si le résultat de l'écriture de v dans M_1 à l'emplacement (b, δ) est une mémoire M_2, alors le résultat de la l'initialisation de (b, δ) avec le type κ dans dans $\overline{M_1}$ représente M_2.

$$\forall M_1, \overline{M_1}, M_2, b \quad \left\{ \begin{array}{l} M_1 \triangleright \overline{M_1} \\ \mathsf{store}(\kappa, M_1, b, \delta, v) = \lfloor M_2 \rfloor \\ v \neq \mathsf{Undef} \end{array} \right. \implies M_2 \triangleright \mathsf{initialize}(\kappa, \overline{M_1}, b, \delta)$$

Démonstration. On note $\overline{M_2} \triangleq \mathsf{initialize}(\kappa, \overline{M_1}, b, \delta)$.

Par application de l'axiome 2.6, les blocs valides de M_2 sont les mêmes que ceux de M_1. De même, par application de l'axiome 3.5, les blocs valides de $\overline{M_2}$ sont les mêmes que ceux de $\overline{M_1}$. Par conséquent, puisque $M_1 \triangleright \overline{M_1}$, M_2 a les mêmes blocs valides que $\overline{M_2}$.

Par un raisonnement analogue, l'égalité des longueurs des blocs valides dans M_2 et dans $\overline{M_2}$ peut être déduite des axiomes 2.10 et 3.9.

Considérons maintenant un type mémoire κ' et un décalage δ' tels que $M_2 \vDash \kappa' @ b', \delta'$. Montrons que $(\exists v' \neq \mathsf{Undef},\ \mathsf{load}(\kappa', M_2, b', \delta') = \lfloor v' \rfloor) \iff \mathsf{is\_initialized}(\kappa', \overline{M_2}, b', \delta') = \top$. Si $b' \neq b$, les axiomes 2.15 et 3.14 permettent de réduire le but à l'hypothèse $M_1 \triangleright \overline{M_1}$. Si $b' = b$, on distingue plusieurs cas selon les positions relatives de δ et δ'. Si les plages d'écriture sont disjointes, c'est-à-dire si $\delta' \leq \delta + \mathsf{sizeof}(\kappa)$, ou $\delta' + \mathsf{sizeof}(\kappa') \leq \delta$, alors comme dans le cas $b' \neq b$, $\mathsf{load}(\kappa', M_2, b', \delta') = \mathsf{load}(\kappa', M_1, b', \delta')$ et $\mathsf{is\_initialized}(\kappa', \overline{M_2}, b', \delta') = \mathsf{is\_initialized}(\kappa', \overline{M_1}, b', \delta')$. On conclut comme précédemment par l'hypothèse $M_1 \triangleright \overline{M_1}$.

Si $b' = b$ et $\delta' < \delta + \mathsf{sizeof}(\kappa)$, ou $\delta' + \mathsf{sizeof}(\kappa') < \delta$, c'est-à-dire si les plages d'écriture ont une intersection non vide, alors l'axiome 2.16 force $\delta' = \delta$ et $\kappa' = \kappa$. On utilise alors les axiomes 2.14 et 3.13 pour conclure. □

L'équivalence est compatible (à gauche) avec la représentation, au sens où dans une relation de représentation, l'état mémoire d'exécution peut être remplacé par un état mémoire équivalent.

Propriété 4.15 (compatibilité entre équivalence et représentation)**.**
Si $M \equiv M'$ et $M \triangleright \overline{M}$, alors $M' \triangleright \overline{M}$.

Démonstration. Par définition, deux mémoires équivalentes ont :
— les mêmes blocs valides
— des blocs valides de même longueur dans les deux mémoires
— par conséquent, les mêmes accès valides
— les mêmes valeurs lors d'une lecture à un accès valide.
Ces éléments sont exactement ceux qui interviennent dans la définition de représentation d'une mémoire d'exécution M par une mémoire d'observation $\overline{M}$. S'ils sont préservés par équivalence avec une mémoire d'exécution M', la représentation l'est donc également. □

4.2.4 Extension

La relation d'extension intervient dans le théorème exprimant la validité de la traduction des prédicats. La traduction de prédicats $[\![\ ]\!]_\Pi$ définie au chapitre 3 repose sur l'introduction de variables intermédiaires pour calculer les prédicats. Ces variables sont allouées en mémoire, et définissent ainsi une zone de la mémoire du programme cible qui n'a pas d'équivalent dans le programme source.

On souhaite donc pouvoir faire la distinction entre la zone de la mémoire d'un programme cible utilisée pour l'exécution du programme instrumenté (comme dans le programme source), et celle qui résulte de la traduction (variables intermédiaires).

Définition 13 (extension d'une mémoire)**.**
On dit que M étend M', ce qu'on note $M \hookrightarrow M'$, si les conditions suivantes sont vérifiées :
— tout bloc valide dans M est également valide dans M', et a la même longueur dans les deux mémoires, autrement dit $\forall b, M \vDash b \implies M' \vDash b \wedge \mathsf{length}(M, b) = \mathsf{length}(M', b)$
— toute lecture réussie dans M réussit également dans M' et renvoie la même valeur, autrement dit $\forall \kappa, b, \delta,\ \mathsf{load}(\kappa, M, b, \delta) = \lfloor v \rfloor \implies \mathsf{load}(\kappa, M', b, \delta) = \lfloor v \rfloor$.

Propriété 4.16.
Soient M et M' deux mémoires. Si $M \hookrightarrow M'$, alors $\forall \kappa, b, \delta,\ M \vDash \kappa \mathbin{@} b, \delta \implies M' \vDash \kappa \mathbin{@} b, \delta$.

Démonstration. La propriété découle directement des définitions d'accès valide et d'extension. □

Propriété 4.17 (extension par allocation)**.**
Soient M, M', b et n tels que $b, M' = \mathsf{alloc}(M, n)$. Alors $M \hookrightarrow M'$.

Démonstration. La propriété découle directement de la définition d'extension et des axiomes du modèle mémoire concernant l'allocation. □

Étendre un contexte n'affecte pas l'évaluation des expressions, au sens où toute expression qui peut être évaluée dans un contexte donné peut également l'être dans une extension de ce contexte, avec le même résultat.

Lemme 2 (évaluation d'expression dans une mémoire étendue)**.**
Soient $C = (E, M)$ et $C' = (E', M')$ deux contextes, et e une expression. Si $C \hookrightarrow C'$, et si en outre les variables libres de e sont dans le domaine de E, alors d'une part $\forall v, C \vDash_{\mathrm{e}} e \Downarrow v \implies C' \vDash_{\mathrm{e}} e \Downarrow v$, et d'autre part $\forall b, \delta, C \vDash_{\mathrm{lv}} e \Downarrow b, \delta \implies C' \vDash_{\mathrm{lv}} e \Downarrow b, \delta$.

Démonstration. On procède par induction mutuelle sur les jugements d'évaluation $C \vDash_{\mathrm{e}} e \Downarrow v$ et $C \vDash_{\mathrm{lv}} e \Downarrow b, \delta$. Les cas E-LVAL et LV-VAR sont traités à l'aide de la définition 13, tandis que les autres cas sont directs. □

4.3 Théorèmes

Le premier théorème traite de la transformation des instructions. Il est construit autour de deux invariants : d'une part l'isomorphisme entre les mémoires d'exécution source et cible, et d'autre part la représentation de la mémoire d'exécution cible par la mémoire d'observation.

Étant donné que certains détails de la formalisation n'ont pas été spécifiés, concernant les opérateurs et leur sémantique, les théorèmes nécessitent l'hypothèse globale suivante :

Hypothèse 1 (sémantique des opérateurs)**.**
La sémantique des opérateurs binaires, unaires, et de comparaison vérifie les propriétés suivantes :

$$\begin{aligned}
\mathsf{sem_binop}(\ddagger, \dot{\sigma}(v_1), \dot{\sigma}(v_1)) &= \dot{\sigma}(\mathsf{sem_binop}(\ddagger, v_1, v_2)) \\
\mathsf{Int}(\mathsf{sem_cmp}(\bowtie, v_1, v_2)) &= \mathsf{sem_binop}(\llbracket \bowtie \rrbracket_{op}, v_1, v_2) \\
\mathsf{sem_unop}(!, \mathsf{Int}(n)) &= \begin{cases} 1 \text{ si } n = 0 \\ 0 \text{ sinon.} \end{cases}
\end{aligned}$$

Théorème 1 (préservation sémantique)**.**
Soit s un programme source évalué dans un contexte initial $(\widehat{E_i}, \widehat{M_i})$, et aboutissant à un état mémoire final $\widehat{M_f}$. Soit en outre $(E_i, M_i, \overline{M_i})$ un contexte cible initial tel que M_i soit isomorphe à $\widehat{M_i}$, et $\overline{M_i}$ représente M_i.

$$\begin{cases} \widehat{E_i}, \widehat{M_i} \vDash_{\mathrm{s}} s \Downarrow \widehat{M_f} \\ \widehat{M_i} \sim M_i \\ M_i \triangleright \overline{M_i} \end{cases}$$

Alors il existe des états mémoire M_f et $\overline{M_f}$ tels que l'évaluation de $\llbracket s \rrbracket_\Sigma$ dans le contexte cible initial ait pour résultat $(M_f, \overline{M_f})$, M_f soit isomorphe à $\widehat{M_f}$, et $\overline{M_f}$ représente M_f.

$$\begin{cases} E_i, M_i, \overline{M_i} \vDash_{\mathrm{s}'} \llbracket s \rrbracket_\Sigma \Downarrow M_f, \overline{M_f} \\ \widehat{M_f} \sim M_f \\ M_f \triangleright \overline{M_f} \end{cases}$$

La preuve de ce théorème, exposée dans la section suivante, est réalisée par induction sur la structure de la dérivation d'évaluation du programme source. Dans la plupart des cas, elle consiste à examiner une opération mémoire ayant lieu dans l'évaluation source, et à montrer ensuite que des opérations analogues sont réalisées dans la mémoire d'exécution du programme cible (préservant ainsi la relation d'isomorphisme avec la mémoire source), et dans la mémoire d'observation (préservant ainsi la propriété de représentation).

L'exception majeure à ce schéma est le cas des assertions. La traduction des prédicats (et donc des assertions) introduit dans le programme cible du code qui, contrairement à celui généré par l'instrumentation des autres instructions, n'est pas directement lié à celui du programme source (ou tout du moins pas aux instructions de nature impérative). Le traitement des assertions et des prédicats nécessite par conséquent un théorème à part entière.

De la même manière, le traitement des termes fait l'objet d'un théorème dédié, bien que très semblable à celui sur les prédicats, tant dans son énoncé que dans sa preuve.

Théorème 2 (traduction correcte des prédicats)**.**
Soit $\widehat{E}, \widehat{M}$ un contexte source, n un entier, β une valeur Booléenne, et p un prédicat dont l'évaluation dans $\widehat{E}, \widehat{M}$ aboutit au résultat β. Soit en outre E un environnement, M une mémoire d'exécution, et $\overline{M}$ une mémoire d'observation tels que :

— il existe une mémoire de programme M^p telle que $M^p \hookrightarrow M$;
— M^p soit isomorphe à $\widehat{M}$;
— $\overline{M}$ représente M^p.

Formellement :

$$\begin{cases} \widehat{E}, \widehat{M} \vDash_{\mathrm{p}} p \Downarrow \beta \\ M^p \hookrightarrow M \\ \widehat{M} \sim M^p \\ M^p \triangleright \overline{M} \end{cases}$$

Soit d'une part le contexte initial $(E_i, M_i) \triangleq \mathsf{alloc\_var}(E, M, \mathsf{res}(n), \mathtt{int8})$, et d'autre part $\lfloor M_i' \rfloor \triangleq \mathsf{store}(\mathrm{i}8, M_i, b, E_i(\mathsf{res}(n)), \mathsf{Int}(\beta))$. Alors il existe M_f telle que :

— l'évaluation du code de traduction $[\![p]\!]^n_{\Pi}$ dans le contexte $(E_i, M_i, \overline{M_i})$ aboutit à $(M_f, \overline{M})$;
— l'évaluation de la variable de résultat $\mathsf{res}(n)$ dans le contexte final (E_i, M_f) aboutit à l'encodage entier de la valeur de vérité de p ;
— la mémoire finale M_f est équivalente à M_i' (c'est-à-dire le résultat de l'écriture de la valeur de vérité de p dans la variable de résultat $\mathsf{res}(n)$, dans la mémoire initiale M_i).

Formellement :

$$\begin{cases} E_i, M_i, \overline{M} \vDash_{\mathrm{s}'} [\![p]\!]^n_{\Pi} \Downarrow M_f, \overline{M} \\ E_i, M_f \vDash_{\mathrm{e}} \mathsf{res}(n) \Downarrow \mathsf{Int}(\beta) \\ M_f \equiv M_i' \end{cases}$$

Théorème 3 (traduction correcte des termes)**.**
Soit $\widehat{E}, \widehat{M}$ un contexte source, n un entier, v une valeur, et t un terme de type τ dont l'évaluation dans $\widehat{E}, \widehat{M}$ aboutit à la valeur $\widehat{v}$. Soit en outre E un environnement, M une mémoire d'exécution, et $\overline{M}$ une mémoire d'observation tels que :

— il existe une mémoire de programme M^p telle que $M^p \hookrightarrow M$;
— M^p soit isomorphe à $\widehat{M}$;
— $\overline{M}$ représente M^p.

$$\begin{cases} \widehat{E}, \widehat{M} \vDash_{\mathrm{t}} t : \tau \Downarrow \widehat{v} \\ M^p \hookrightarrow M \\ \widehat{M} \sim_{\sigma} M^p \\ M^p \triangleright \overline{M} \end{cases}$$

Soit d'une part le contexte initial $(E_i, M_i) = \mathsf{alloc\_var}(E, M, \mathsf{res}(n), \tau)$, et d'autre part $\lfloor M_i' \rfloor \triangleq \mathsf{store}(\mathsf{mtype}(\tau), M_i, b, E_i(\mathsf{res}(n)), v)$, où $v \triangleq \dot{\sigma}(\widehat{v})$.

Alors il existe M_f telle que :

— l'évaluation du code de traduction $[\![t]\!]^n_{\mathrm{T}}$ dans le contexte $(E_i, M_i, \overline{M_i})$ aboutit à $(M_f, \overline{M})$;
— l'évaluation de la variable de résultat $\mathsf{res}(n)$ dans le contexte final (E_i, M_f) aboutit à la valeur v ;
— la mémoire finale M_f est équivalente à M_i' (c'est-à-dire le résultat de l'écriture de la valeur de vérité de p dans la variable de résultat $\mathsf{res}(n)$, dans la mémoire initiale M_i).

Formellement :

$$\begin{cases} E_i, M_i, \overline{M} \vDash_{\mathrm{s}'} [\![t]\!]^n_{\mathrm{T}} \Downarrow M_f, \overline{M} \\ E_i, M_f \vDash_{\mathrm{e}} \mathsf{res}(n) \Downarrow v \\ M_f \equiv M_i'. \end{cases}$$

4.4 Preuves

On présente dans section les preuves des trois théorèmes de préservation sémantique énoncés précédemment. Chacune de ces preuves consiste en un raisonnement par induction sur l'évaluation de l'objet syntaxique considéré. Les preuves sont exposées dans le même ordre que les théorèmes : on examine d'abord la traduction des instructions, puis celle des prédicats, puis celle des termes. Ce plan correspond à une « descente » dans la structure syntaxique des programmes : il est donc à rebours de l'ordre de dépendance logique entre les théorèmes. En effet, la preuve du théorème 1 (instructions) s'appuie sur le théorème 2 (prédicats), dont la preuve utilise elle-même le théorème 3.

4.4.1 Traduction des instructions

On procède par induction sur l'évaluation de s. Pour chacun des cas du raisonnement inductif, on construit un jugement d'évaluation pour le programme cible résultant de la transformation, et on prouve que l'état mémoire final de cette évaluation satisfait les propriétés énoncées dans le théorème.

Étant donné que cette preuve fait intervenir un grand nombre d'états mémoire, on adopte une convention de nommage systématique pour les désigner. Cette convention repose sur la notion intuitive de *pas d'exécution*, que l'on peut voir comme les étapes délimitées par l'exécution des instructions élémentaires.

Les codes sources seront présentés de manière à ce que chaque ligne corresponde à un pas d'exécution. Les états mémoire (d'exécution et d'observation) et environnements portant l'indice n désigneront l'état de l'évaluation du programme à la fin de la ligne n du code source. L'indice 0 désignera quant à lui l'état initial du programme, avant l'évaluation de la première instruction.

Ainsi, dans le code ci-dessous, E_0 correspond à l'environnement initial d'évaluation du programme, M_1 à l'état de la mémoire d'exécution résultant de l'évaluation de l'allocation de mémoire, et $\overline{M_2}$ de l'état de la mémoire d'observation après enregistrement du nouveau bloc via la primitive `store_block`.

```
1  l = malloc(e);
2  store_block(l,e);
3  initialize(&l);
```

Cas S-SKIP. Cas trivial. Il suffit de définir $C_f \triangleq C_i$.

Cas S-ASSIGN. Cas de base. On raisonne directement sur les états mémoire.

Évaluation source :

$$\frac{\widehat{E}, \widehat{M_i} \vDash_{\mathrm{e}} e_2 : \tau \Downarrow \widehat{v} \qquad \widehat{E}, \widehat{M_i} \vDash_{\mathrm{lv}} e_1 : \tau \Downarrow \widehat{b}, \widehat{\delta} \qquad \mathsf{store}(\mathsf{mtype}(\tau), \widehat{M_i}, \widehat{b}, \widehat{\delta}, \widehat{v}) = \lfloor \widehat{M_f} \rfloor}{\widehat{E}, \widehat{M_i} \vDash_{\mathrm{s}} e_1 = e_2; \Downarrow \widehat{M_f}}$$

Code instrumenté :

```
1  e_1 = e_2;
2  initialize(&e_1);
```

Il s'agit de construire un contexte d'évaluation cible $\mathscr{C}_f$ tel que :

$$\begin{cases} C_i, M_i, \overline{M_i} \vDash_{s'} e_1 = e_2;\ \texttt{initialize}(\&e_1); \Downarrow M_f, \overline{M_f} \\ \widehat{M_f} \sim M_f \\ M_f \triangleright \overline{M_f} \end{cases}$$

On définit les objets sémantiques appropriés pour pouvoir dériver une évaluation pour le code instrumenté.

$$\dfrac{\dfrac{E, M_i \vDash_{\mathrm{e}} e_2 : \tau \Downarrow v \qquad E, M_i \vDash_{\mathrm{lv}} e_1 : \tau \Downarrow b, \delta \qquad \mathsf{store}(\mathsf{mtype}(\tau), M_i, b, \delta, v) = \lfloor M_1 \rfloor}{\mathscr{C}_i \vDash_{\mathrm{s}'} e_1 = e_2; \Downarrow M_1, \overline{M_1}} \qquad \dfrac{\dots \qquad \overline{M_2} = \mathsf{initialize}(\tau, \overline{M_1}, b, \delta)}{\mathscr{C}_1 \vDash_{\mathrm{s}'} \texttt{initialize}(\&e_1); \Downarrow M_2, \overline{M_2}}}{\mathscr{C}_i \vDash_{\mathrm{s}'} e_1 = e_2; \texttt{initialize}(\&e_1); \Downarrow M_2, \overline{M_2}}$$

Soit σ une permutation de blocs telle que $\widehat{E_0}, \widehat{M_0} \sim_\sigma E_0, M_0$. On définit $b \triangleq \sigma(\widehat{b})$, $v \triangleq \dot{\sigma}(\widehat{v})$ et $\delta \triangleq \widehat{\delta}$, et on applique le lemme 1 pour obtenir les évaluations d'expressions de la dérivation précédente : $E, M_i \vDash_{\mathrm{e}} e_2 \Downarrow v$ et $E, M_i \vDash_{\mathrm{lv}} e_1 \Downarrow b, \delta$. On définit ensuite les états mémoire suivants :

$$\begin{array}{l|l} M_0 \triangleq M_i & \overline{M_0} \triangleq \overline{M_i} \\ \lfloor M_1 \rfloor \triangleq \mathsf{store}(\mathsf{mtype}(\tau), M_0, b, \delta, v) & \overline{M_1} \triangleq \overline{M_0} \\ M_3 \triangleq M_2 & \overline{M_2} \triangleq \mathsf{initialize}(\tau, \overline{M_1}, b, \delta) \end{array}$$

Montrons que la définition de M_1 est légale, c'est-à-dire que l'opération d'écriture réussit. Par hypothèse, l'écriture associée dans l'évaluation source réussit : $\mathsf{store}(\mathsf{mtype}(\tau), \widehat{M_0}, \widehat{b}, \widehat{\delta}, \widehat{v}) = \lfloor \widehat{M_1} \rfloor$. D'après la propriété 4.4, l'écriture cible réussit, et en outre $\widehat{M_1} \sim M_1$.

Par ailleurs, la mémoire d'observation finale représente la mémoire d'exécution finale ($M_f \triangleright \overline{M_f}$), par application de la propriété 4.14.

Cas S-MALLOC. Cas de base. On raisonne directement sur les états mémoire.

Évaluation source :

$$\dfrac{\widehat{E}, \widehat{M_i} \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(n) \qquad \mathsf{alloc}(\widehat{M_i}, 0, n) = (\widehat{b'}, \widehat{M_1}) \qquad \widehat{E}, \widehat{M_i} \vDash_{\mathrm{lv}} l : \tau* \Downarrow (\widehat{b}, \widehat{\delta}) \qquad \mathsf{store}(\mathsf{mtype}(\tau*), \widehat{M_1}, \widehat{b}, \widehat{\delta}, \mathsf{Ptr}(\widehat{b'}, 0)) = \lfloor \widehat{M_f} \rfloor}{\widehat{E}, \widehat{M_i} \vDash_{\mathrm{s}} l = \texttt{malloc}(e); \Downarrow \widehat{M_f}}$$

Traduction :

```
1  l = malloc(e);
2  store_block(l,e);
3  initialize(&l);
```

On cherche à construire une évaluation cible de la forme suivante, pour des valeurs adéquates des environnements et états mémoire.

$$\dfrac{\dfrac{\begin{array}{c} C_0 \vDash_{\mathrm{lv}} l : \tau* \Downarrow (b, \delta) \\ C_0 \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(n) \\ \mathsf{alloc}(M_0, 0, n) = (b', M_0') \end{array} \quad \mathsf{store}(\mathsf{mtype}(\tau*), M_0', b, \delta, \mathsf{Ptr}(b', 0)) = \lfloor M_1 \rfloor}{\mathscr{C}_0 \vDash_{\mathrm{s}} l = \texttt{malloc}(e); \Downarrow \mathscr{C}_1} \qquad \dfrac{\dots \qquad \dots}{\mathscr{C}_1 \vDash_{\mathrm{s}} \texttt{store\_block}(l, e); \Downarrow \mathscr{C}_2 \qquad \mathscr{C}_2 \vDash_{\mathrm{s}} \texttt{initialize}(\&l); \Downarrow \mathscr{C}_3}}{\mathscr{C}_0 \vDash_{\mathrm{s}} [\![l = \texttt{malloc}(e);]\!]_\Sigma \Downarrow M_3, \overline{M_3}}$$

Définition des mémoires :

$$\begin{array}{l|l} M_0 \triangleq M_i & \overline{M_0} \triangleq \overline{M_i} \\ (b', M_0') \triangleq \mathsf{alloc}(M_0, 0, n) & \overline{M_0'} \triangleq \overline{M_0} \\ \lfloor M_1 \rfloor \triangleq \mathsf{store}(\tau*, M_0', b, \delta, \mathsf{Ptr}(b', 0)) & \overline{M_1} \triangleq \overline{M_0'} \\ M_2 \triangleq M_1 & \overline{M_2} \triangleq \mathsf{store\_block}(\overline{M_1}, b', 0, n) \\ M_3 \triangleq M_2 & \overline{M_3} \triangleq \mathsf{initialize}(\tau*, \overline{M_2}, b, \delta) \end{array}$$

Par hypothèse $\widehat{M_0} \sim M_0$; l'isomorphisme est préservé par allocation (respectivement par écriture) d'après la propriété 4.2 (respectivement la propriété 4.4). De même, par hypothèse $M_0 \triangleright \overline{M_0}$, et la représentation est préservée par allocation (respectivement par écriture) d'après la propriété 4.12 (respectivement la propriété 4.14).

Cas S-FREE. Cas de base. On raisonne directement sur les états mémoire.

Évaluation source :

$$\frac{\widehat{E}, \widehat{M_0} \vDash_{\mathrm{e}} e \Downarrow \mathsf{Ptr}(\widehat{b}, 0) \qquad \mathsf{free}(\widehat{M_0}, \widehat{b}) = \lfloor \widehat{M_1} \rfloor}{\widehat{E}, \widehat{M_0} \vDash_{\mathrm{s}} \texttt{free}(e); \Downarrow \widehat{M_1}}$$

Traduction :

```
free(e);
delete_block(e);
```

Évaluation cible :

$$\frac{\dfrac{\mathscr{C}_0 \vDash_{\mathrm{e}} e \Downarrow \mathsf{Ptr}(b, 0) \quad \mathsf{free}(M_0, b) = \lfloor M_1 \rfloor}{\mathscr{C}_0 \vDash_{\mathrm{s}} \texttt{free}(e); \Downarrow M_1, \overline{M_1}} \qquad \dfrac{\mathscr{C}_1 \vDash_{\mathrm{e}} e \Downarrow \mathsf{Ptr}(b, 0) \quad \mathsf{delete\_block}(b, \overline{M_1}) = \overline{M_2}}{\mathscr{C}_1 \vDash_{\mathrm{s}} \texttt{delete\_block}(e); \Downarrow M_2, \overline{M_2}}}{\mathscr{C}_0 \vDash_{\mathrm{s}} \texttt{free}(e); \texttt{ delete\_block}(e); \Downarrow M_2, \overline{M_2}}$$

Définition des mémoires :

$$\begin{array}{l|l} M_0 \triangleq M_i & \overline{M_0} \triangleq \overline{M_i} \\ \lfloor M_1 \rfloor \triangleq \mathsf{free}(M_0, b,) & \overline{M_1} \triangleq \overline{M_0} \\ M_2 \triangleq M_1 & \overline{M_2} \triangleq \mathsf{delete\_block}(b, \overline{M_1}) \end{array}$$

Par hypothèse $\mathsf{free}(\widehat{M_0}, \widehat{b}) = \lfloor \widehat{M_1} \rfloor$, donc d'après l'axiome 2.4, $\widehat{M_0} \vDash \widehat{b}$. Or, $\widehat{M_0} \sim M_0$ par hypothèse, on a donc aussi $M_0 \vDash b$. L'axiome 2.4 permet alors de justifier la définition de M_1.

L'isomorphisme $\widehat{M_0} \sim M_0$ est préservé par déallocation de blocs correspondants (propriété 4.3) : $\widehat{M_1} \sim M_1$. De la même façon, la représentation $M_0 \triangleright \overline{M_0}$ est préservée par déallocations correspondantes (propriété 4.13) : $M_1 \triangleright \overline{M_1}$.

Cas S-LOGICAL-ASSERT. Ce cas repose sur les propriétés de la traduction des prédicats, exprimées par le théorème 2.

Évaluation source :

$$\frac{\widehat{C}_i \vDash_{\mathrm{p}} p \Downarrow \top}{\widehat{C}_i \vDash_{\mathrm{s}} \texttt{logical\_assert}(p); \Downarrow \widehat{M_f}}$$

Traduction :

```
let res(0) in
  [[p]]^0_Π
  assert(res(0));
end
```

On cherche à construire une évaluation cible de la forme suivante.

$$\frac{\dfrac{\dfrac{\dots}{\mathscr{C}_1 \vDash_{s'} [\![p]\!]^0_\Pi \Downarrow M_2, \overline{M_2}} \qquad \dfrac{\mathscr{C}_2 \vDash_{\mathrm{e}} \mathsf{res}(0) \Downarrow \mathsf{Int}(1) \qquad 1 \neq 0}{\mathscr{C}_2 \vDash_{s'} \texttt{assert}(\mathsf{res}(0)); \Downarrow M_3, \overline{M_3}}}{\mathscr{C}_1 \vDash_{s'} [\![p]\!]^0_\Pi \texttt{ assert}(\mathsf{res}(0)); \Downarrow M_3, \overline{M_3}} \quad C_1 = \mathsf{alloc\_var}(C_0, \tau, \mathsf{res}(0)) \qquad C_4 = \mathsf{dealloc\_var}(C_3, \mathsf{res}(0))}{\mathscr{C}_0 \vDash_{s'} \texttt{let } \mathsf{res}(0) : \tau \texttt{ in } [\![p]\!]^0_\Pi \texttt{ assert}(\mathsf{res}(0)); \texttt{end} \Downarrow M_4, \overline{M_4}}$$

Définitions des environnements et états mémoire :

$$\begin{array}{lcl|lcl|lcl} E_0 & \triangleq & E_i & M_0 & \triangleq & M_i & \overline{M_0} & \triangleq & \overline{M_i} \\ E_1 & \triangleq & \mathsf{add}(\mathsf{res}(0), b_1, E_i) & (b_1 M_1) & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\texttt{int8})) & \overline{M_1} & \triangleq & \overline{M_0} \\ E_2 & \triangleq & E_1 & M_2 & \triangleq & \ldots & \overline{M_2} & \triangleq & \overline{M_1} \\ E_3 & \triangleq & E_2 & M_3 & \triangleq & M_2 & \overline{M_3} & \triangleq & \overline{M_2} \\ E_4 & \triangleq & \mathsf{remove}(\mathsf{res}(0), E_3) & \lfloor M_4 \rfloor & \triangleq & \mathsf{free}(M_3, b_1) & \overline{M_4} & \triangleq & \overline{M_3} \end{array}$$

Montrons, dans un premier temps, que $E_i, M_i, \overline{M_i} \vDash_{\mathrm{s}'} \texttt{let...end} \Downarrow M_4, \overline{M_4}$, ce qui revient à montrer que les états mémoire et environnements ci-dessus sont bien définis. L'existence de M_2 résulte de l'application du théorème 2 à l'évaluation source $\widehat{E_i}, \widehat{M_i} \vDash_{\mathrm{p}} p \Downarrow \top$ accompagnée du contexte cible $\mathscr{C}_0 = (E_0, M_0, \overline{M_0}) = \mathscr{C}_i$. On définit $M^p \triangleq M_0$, état mémoire qui satisfait les relations suivantes :

— $M^p \hookrightarrow M_0$ par définition de $\hookrightarrow$;
— $\widehat{M_0} \sim M^p$ par hypothèse ;
— $M^p \triangleright \overline{M_0}$ par hypothèse.

D'après le théorème 2, il existe M_2 telle que :

$$\begin{cases} E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![p]\!]^0_\Pi \Downarrow M_2, \overline{M} \\ E_1, M_2 \vDash_{\mathrm{e}} \mathsf{res}(0) \Downarrow \mathsf{Int}(1) \\ M_2 \equiv M_1' \end{cases}$$

où M_1' est défini par $\lfloor M_1' \rfloor \triangleq \mathsf{store}(\mathrm{i}\,8, M_1, b_1, 0, \mathsf{Int}(1))$. L'état mémoire M_2 est donc bien défini.

D'autre part, $M_3 \triangleq M_2$ et $M_2 \equiv M_1'$, donc $M_3 \equiv M_1'$. Or $\lfloor M_1' \rfloor \triangleq \mathsf{store}(\mathrm{i}\,8, M_1, b_1, 0, \mathsf{Int}(1))$ et $M_1 \vDash b_1$ (d'après la définition de M_1 et l' axiome 2.2), par conséquent, d'après l' axiome 2.6, $M_1' \vDash b_1$. Il s'ensuit par définition de l'équivalence que $M_3 \vDash b_1$.

Montrons à présent que l'état mémoire obtenu M_4 satisfait les relations $\widehat{M_f} \sim M_4$ et $M_4 \triangleright \overline{M_4}$. On remarque tout d'abord que, d'après la règle S-LOGICAL-ASSERT, $\widehat{M_f} = \widehat{M_i}$, ce qui réduit l'isomorphisme à établir à $\widehat{M_i} \sim M_4$. Par ailleurs, $\overline{M_4} = \overline{M_i}$, donc la relation de représentation à démontrer peut être écrite $M_4 \triangleright \overline{M_i}$. Or, par hypothèse, $\widehat{M_i} \sim M_i$ et $M_i \triangleright \overline{M_i}$: il suffit donc de montrer que $M_0 \equiv M_4$ (par définition, $M_0 = M_i$). En effet, on aura d'une part $M_0 \sim M_4$ (l'équivalence étant un cas particulier d'isomorphisme) donc $\widehat{M_i} \sim M_0 \sim M_4$; et d'autre part $M_4 \triangleright \overline{M_i}$ par application de la propriété 4.15.

Puisque $M_2 \equiv M_1'$, on obtient, par la propriété 4.9, l'équivalence $M_4 \equiv M_4'$, où $\lfloor M_4' \rfloor \triangleq \mathsf{free}(M_1', b_1)$. Les états mémoire M_0, M_1, M_1' et M_4 sont alors liés par les relations suivantes :

$$\begin{cases} (b_1, M_1) = \mathsf{alloc}(M_0, \mathsf{sizeof}(\texttt{int8})) \\ \lfloor M_1' \rfloor = \mathsf{store}(\mathrm{i}\,8, M_1, b_1, 0, \mathsf{Int}(1)) \\ \lfloor M_4' \rfloor = \mathsf{free}(M_1', b_1) \end{cases}$$

M_4' est obtenue à partir de M_0 par allocation, écriture puis déallocation du bloc b_1. D'après la propriété 4.8, M_0 et M_4' sont donc équivalents : $M_0 \equiv M_4'$. Puisque $M_4' \equiv M_4$, on conclut par transitivité de l'équivalence.

Cas S-SEQ. Cas récursif.

Évaluation source :

$$\frac{\widehat{C_0} \vDash_{\mathrm{s}} s_1 \Downarrow \widehat{M_1} \qquad \widehat{C_1} \vDash_{\mathrm{s}} s_2 \Downarrow \widehat{M_2}}{\widehat{C_0} \vDash_{\mathrm{s}} s_1 s_2 \Downarrow \widehat{M_2}}$$

Traduction :

```
1 ⟦s1⟧Σ ⟦s2⟧Σ
```

Évaluation cible :

$$\frac{\mathscr{C}_0 \vDash_{s'} [\![s_1]\!]_\Sigma \Downarrow M_1, \overline{M_1} \qquad \mathscr{C}_1 \vDash_{s'} [\![s_2]\!]_\Sigma \Downarrow M_2, \overline{M_2}}{\mathscr{C}_0 \vDash_{s'} [\![s_1]\!]_\Sigma \ [\![s_2]\!]_\Sigma \Downarrow M_2, \overline{M_2}}$$

Par hypothèse d'induction sur l'évaluation de s_1, on a $\mathscr{C}_0 \vDash_{s'} [\![s_1]\!]_\Sigma \Downarrow M_1, \overline{M_1}$ avec $\widehat{M_1} \sim M_1$ et $M_1 \triangleright \overline{M_1}$. On peut donc appliquer l'hypothèse d'induction à l'évaluation de s_2, ce qui donne d'une part l'évaluation $\mathscr{C}_1 \vDash_{s'} [\![s_2]\!]_\Sigma \Downarrow M_2, \overline{M_2}$ pour construire la dérivation ci-dessus, et d'autre part les relations $\widehat{M_2} \sim M_2$ et $M_2 \triangleright \overline{M_2}$.

Cas S-IF-FALSE. Cas récursif.

Évaluation source :

$$\frac{\widehat{C_0} \vDash_e e \Downarrow \mathsf{Int}(0) \qquad \widehat{C_0} \vDash_s s_2 \Downarrow \widehat{M_1}}{\widehat{C_0} \vDash_s \texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow \widehat{M_1}}$$

Traduction :

```
1 if (e) then ⟦s1⟧Σ else ⟦s2⟧Σ
```

Évaluation cible :

$$\frac{C_0 \vDash_e e \Downarrow \mathsf{Int}(0) \qquad C_0 \vDash_{s'} [\![s_2]\!]_\Sigma \Downarrow M_1}{C_0 \vDash_{s'} \texttt{if}\ (e)\ \texttt{then}\ [\![s_1]\!]_\Sigma\ \texttt{else}\ [\![s_2]\!]_\Sigma \Downarrow M_1, \overline{M_1}}$$

D'une part, $\widehat{C_0}$ et C_0 sont isomorphes par hypothèse, et $\widehat{C_0} \vDash_e e \Downarrow \mathsf{Int}(0)$. Par application du lemme 1 (évaluation d'une même expression dans deux contextes isomorphes), la condition de branchement a la même valeur dans l'évaluation cible que dans l'évaluation source : $C_0 \vDash_e e \Downarrow \mathsf{Int}(0)$.

D'autre part, on peut appliquer l'hypothèse d'induction sur $\widehat{C_0} \vDash_s s_2 \Downarrow \widehat{M_1}$, ce qui établit les existences de M_1 et $\overline{M_1}$ telles que :

$$\begin{cases} C_0 \vDash_{s'} [\![s_2]\!]_\Sigma \Downarrow M_1, \overline{M_1} \\ \widehat{M_1} \sim M_1 \\ M_1 \triangleright \overline{M_1} \end{cases}$$

Cas S-IF-TRUE. Cas récursif, similaire au précédent.

Évaluation source :

$$\frac{\widehat{C_0} \vDash_e e \Downarrow \mathsf{Int}(n) \qquad n \neq 0 \qquad \widehat{C_0} \vDash_s s_1 \Downarrow \widehat{M_1}}{\widehat{C_0} \vDash_s \texttt{if}\ (e)\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \Downarrow \widehat{M_1}}$$

Traduction :

```
1 if (e) then ⟦s1⟧Σ else ⟦s2⟧Σ
```

Évaluation cible :

$$\frac{C_0 \vDash_e e \Downarrow \mathsf{Int}(n) \qquad n \neq 0 \qquad C_0 \vDash_{s'} [\![s_1]\!]_\Sigma \Downarrow M_1}{C_0 \vDash_{s'} \texttt{if}\ (e)\ \texttt{then}\ [\![s_1]\!]_\Sigma\ \texttt{else}\ [\![s_2]\!]_\Sigma \Downarrow M_1, \overline{M_1}}$$

D'une part, $\widehat{C_0}$ et C_0 sont isomorphes par hypothèse, et $\widehat{C_0} \vDash_{\text{e}} e \Downarrow \mathsf{Int}(n)$. Par application du lemme 1 (évaluation d'une même expression dans deux contextes isomorphes), la condition de branchement a la même valeur dans l'évaluation cible que dans l'évaluation source : $C_0 \vDash_{\text{e}} e \Downarrow \mathsf{Int}(n)$.

D'autre part, on peut appliquer l'hypothèse d'induction sur $\widehat{C_0} \vDash_{\text{s}} s_1 \Downarrow \widehat{M_1}$, ce qui établit les existences de M_1 et $\overline{M_1}$ telles que :

$$\begin{cases} C_0 \vDash_{\text{s}'} [\![s_1]\!]_\Sigma \Downarrow M_1, \overline{M_1} \\ \widehat{M_1} \sim M_1 \\ M_1 \triangleright \overline{M_1}. \end{cases}$$

Cas S-WHILE-FALSE. Cas de base.

Évaluation source :

$$\frac{\widehat{C_0} \vDash_{\text{e}} e \Downarrow \mathsf{Int}(0)}{\widehat{C_0} \vDash_{\text{s}} \texttt{while}\ (e)\ s \Downarrow \widehat{M_0}}$$

Traduction :

```
while (e) [[s]]_Σ
```

Évaluation cible :

$$\frac{C_0 \vDash_{\text{e}} e \Downarrow \mathsf{Int}(0)}{\mathscr{C}_0 \vDash_{\text{s}'} \texttt{while}\ (e)\ [\![s]\!]_\Sigma \Downarrow M_0, \overline{M_0}}$$

$\widehat{C_0}$ et C_0 sont isomorphes et $\widehat{C_0} \vDash_{\text{e}} e \Downarrow \mathsf{Int}(0)$, donc par application du lemme 1, $C_0 \vDash_{\text{e}} e \Downarrow \mathsf{Int}(0)$. Par ailleurs, $\widehat{M_0} \sim M_0$ et $M_0 \triangleright \overline{M_0}$ par hypothèse.

Cas S-WHILE-TRUE. Cas récursif.

Évaluation source :

$$\frac{\widehat{C_0} \vDash_{\text{e}} e \Downarrow \mathsf{Int}(n) \qquad n \neq 0 \qquad \widehat{C_0} \vDash_{\text{s}} s \Downarrow \widehat{M_1} \qquad \widehat{C_1} \vDash_{\text{s}} \texttt{while}\ (e)\ s \Downarrow \widehat{M_2}}{\widehat{C_0} \vDash_{\text{s}} \texttt{while}\ (e)\ s \Downarrow \widehat{M_2}}$$

Traduction :

```
while (e) [[s]]_Σ
```

Évaluation cible :

$$\frac{C_0 \vDash_{\text{e}} e \Downarrow \mathsf{Int}(n) \qquad n \neq 0 \qquad C_0 \vDash_{\text{s}'} [\![s]\!]_\Sigma \Downarrow M_1, \overline{M_1} \qquad C_1 \vDash_{\text{s}'} \texttt{while}\ (e)\ [\![s]\!]_\Sigma \Downarrow M_2, \overline{M_2}}{C_0 \vDash_{\text{s}'} \texttt{while}\ (e)\ [\![s]\!]_\Sigma \Downarrow M_2, \overline{M_2}}$$

$\widehat{C_0}$ et C_0 sont isomorphes et $\widehat{C_0} \vDash_{\text{e}} e \Downarrow \mathsf{Int}(n)$, donc par application du lemme 1, $C_0 \vDash_{\text{e}} e \Downarrow \mathsf{Int}(n)$. Par hypothèse d'induction sur $\widehat{C_0} \vDash_{\text{s}} s \Downarrow \widehat{M_1}$, il existe M_1 et $\overline{M_1}$ tels que :

$$\begin{cases} C_0 \vDash_{\text{s}'} [\![s]\!]_\Sigma \Downarrow M_1, \overline{M_1} \\ \widehat{M_1} \sim M_1 \\ M_1 \triangleright \overline{M_1} \end{cases}$$

On peut donc appliquer une seconde fois l'hypothèse d'induction, cette fois-ci à l'évaluation $\widehat{C_1} \vDash_{\text{s}} \texttt{while}\ (e)\ s \Downarrow \widehat{M_2}$. On en déduit les existences de M_2 et $\overline{M_2}$ telles que

$$\begin{cases} C_1 \vDash_{\text{s}'} [\![s]\!]_\Sigma \Downarrow M_2, \overline{M_2} \\ \widehat{M_2} \sim M_2 \\ M_2 \triangleright \overline{M_2}. \end{cases}$$

Cas S-LET. Ce cas récursif fait également intervenir un raisonnement sur les états mémoire.

Évaluation source :

$$\frac{\widehat{E_1}, \widehat{M_1} = \mathsf{alloc\_var}(\widehat{E_0}, \widehat{M_0}, x, \tau) \qquad \widehat{E_1}, \widehat{M_1} \vDash_{\mathrm{s}} s \Downarrow \widehat{M_2} \qquad \widehat{M_3} = \mathsf{dealloc\_var}(\widehat{E_1}, \widehat{M_2}, x)}{\widehat{E_0}, \widehat{M_0} \vDash_{\mathrm{s}} \texttt{let}\ x : \tau\ \texttt{in}\ s\ \texttt{end} \Downarrow \widehat{M_3}}$$

Traduction :

```
1  let x:τ in
2    store_block(&x,sizeof(τ));
3    [[s]]_Σ
4    delete_block(&x);
5  end
```

Évaluation cible :

$$\dfrac{\dfrac{\dfrac{\dots}{\mathscr{C}_1 \vDash_{s'} \texttt{store\_block}(\&x, \mathsf{sizeof}(\tau)); \Downarrow C_2} \qquad \dfrac{\dfrac{\dots}{\mathscr{C}_2 \vDash_{s'} [\![s]\!]_\Sigma \Downarrow C_3} \qquad \dfrac{\dots}{\mathscr{C}_3 \vDash_{s'} \texttt{delete\_block}(\&x); \Downarrow C_4}}{\mathscr{C}_2 \vDash_{s'} [\![s]\!]_\Sigma ; \texttt{delete\_block}(\dots); \Downarrow C_4}}{\mathscr{C}_1 \vDash_{s'} \texttt{store\_block}(\dots,); [\![s]\!]_\Sigma\ \texttt{delete\_block}(\dots); \Downarrow C_4} \qquad C_1 = \mathsf{alloc\_var}(C_0, x, \tau) \qquad C_5 = \mathsf{dealloc\_var}(C_4, x)}{\mathscr{C}_0 \vDash_{s'} \texttt{let}\ x : \tau\ \texttt{in}\ \texttt{store\_block}(\dots,); [\![s]\!]_\Sigma\ \texttt{delete\_block}(\dots); \texttt{end} \Downarrow C_5}$$

Définition des états mémoire :

$M_0 \triangleq M_i$	$\overline{M_0} \triangleq \overline{M_i}$
$b, M_1 \triangleq \mathsf{alloc}(M_0, \mathsf{sizeof}(\tau))$	$\overline{M_1} \triangleq \overline{M_0}$
$M_2 \triangleq M_1$	$\overline{M_2} \triangleq \mathsf{store\_block}(\overline{M_1}, b, \mathsf{sizeof}(\tau))$
$\dots$	$\dots$
$M_4 \triangleq M_3$	$\overline{M_4} \triangleq \mathsf{delete\_block}(\overline{M_3}, b)$
$\lfloor M_5 \rfloor \triangleq \mathsf{free}(M_4, b)$	$\overline{M_5} \triangleq \overline{M_4}$

$\widehat{M_1}$ et M_1 sont obtenus par allocations de même taille à partir d'états mémoire isomorphes ($\widehat{M_0} \sim M_0$ par hypothèse). D'après la propriété 4.2, ils sont donc encore isomorphes : $\widehat{M_1} \sim_\sigma M_1$ avec en outre $\widehat{b} = \sigma^{-1}(b)$.

De la même façon, $M_0 \triangleright \overline{M_0}$ par hypothèse et M_1 et $\overline{M_2}$ sont obtenus par allocation ou enregistrement d'un même bloc à partir de M_0 et $\overline{M_0}$: d'après la propriété 4.12, on a donc encore $M_1 \triangleright \overline{M_2}$.

On peut donc appliquer à l'évaluation source $\widehat{E_1}, \widehat{M_1} \vDash_{\mathrm{s}} s \Downarrow \widehat{M_2}$ l'hypothèse d'induction, ce qui permet de déduire les existences de M_3 et $\overline{M_3}$ telles que :

$$\begin{cases} C_2 \vDash_{s'} [\![s]\!]_\Sigma \Downarrow M_3, \overline{M_3} \\ \widehat{M_2} \sim_\sigma M_3 \\ M_3 \triangleright \overline{M_3}. \end{cases}$$

Montrons maintenant que la définition de M_5 est justifiée, c'est-à-dire, d'après l'axiome 2.5, que $M_4 \vDash b$. On note $\widehat{b}$ le résultat de l'allocation de la variable x dans le programme source : $\widehat{b} \triangleq E_1(x)$. La déallocation de $\widehat{b}$ en sortie du `let` source réussit, par hypothèse. D'après l' axiome 2.5, cela implique la validité de $\widehat{b}$ lors de la déallocation : $\widehat{M_2} \vDash \widehat{b}$. Or, on a montré que $\widehat{M_2} \sim_\sigma M_3$. Puisque $\widehat{b} = \sigma^{-1}(b)$, la validité de $\widehat{b}$ dans $\widehat{M_2}$ implique celle de b dans M_3, par définition d'isomorphisme.

Enfin, la relation $\widehat{M_3} \sim M_5$ est obtenue par application de la propriété 4.3 à la proposition $\widehat{M_2} \sim M_3$ démontrée précédemment. De même, $M_5 \triangleright \overline{M_5}$ découle de l'application de la propriété 4.13 à la proposition $M_3 \triangleright \overline{M_3}$. □

4.4.2 Traduction des prédicats

On procède par induction sur la structure de l'évaluation source. Dans chaque cas, le but est de construire, pour le code de traduction $[\![p]\!]^n_\Pi$, une évaluation (et les contextes intermédiaires associés) qui aboutit à une certaine mémoire d'exécution finale M_f. L'évaluation de $\mathsf{res}(n)$ dans le contexte final doit donner le résultat correspondant à la valeur de p dans l'évaluation source, et entre outre M_f doit préserver les invariants du théorème, en particulier les relations avec $\widehat{M}$ et $\overline{M}$.

Le travail de preuve au sein de chaque cas peut donc être décomposé de la manière suivante :

1. construction d'un squelette d'évaluation cible pour le code de traduction, c'est-à-dire définition de la structure de l'évaluation (règles d'évaluation employées), sans se préoccuper à ce stade des états mémoire et environnements impliqués ;
2. définition des états mémoire et environnements intervenant dans l'évaluation ;
3. preuve de validité des définitions, pour les états qui résultent d'opérations mémoire susceptibles d'échouer ;
4. preuve d'équivalence entre l'état mémoire final M_f et l'état fictif M'_i défini comme le résultat de l'écriture directe de la valeur de vérité du prédicat évalué dans l'état initial M_i (la valeur de vérité étant représentée sous forme d'entier via l'encodage usuel).

Comme dans la preuve du théorème 1, on associe à chaque ligne du code de traduction l'état mémoire et l'environnement du programme *une fois la ligne évaluée*. En accord avec cette convention, on définit le contexte d'évaluation initial $(E_0, M_0) = \mathsf{alloc\_var}(\mathsf{res}(n), E, M, \texttt{int8})$. On définit en outre $b_0 \triangleq E_i(\mathsf{res}(n))$, ce qui garantit, dans tous les cas d'induction, la proposition suivante : $M_0 \vDash \mathrm{i}\,8 @ b_0, 0$.

Il est important de noter que l'induction a lieu avec $\widehat{E}$, $\widehat{M}$, $\overline{M}$ et M^p fixés. Cette invariance correspond au fait que la séquence de code générée pour une assertion sur un prédicat n'est associée qu'à un unique contexte source (celui dans lequel est évalué le prédicat), et ne modifie pas la mémoire d'observation : elle ne fait que la consulter.

Cas P-TRUE. Évaluation source :

$$\text{P-TRUE}\ \frac{}{\widehat{E}, \widehat{M} \vDash_{\mathrm{p}} \backslash\texttt{true} \Downarrow \top}$$

Traduction :

```
1  res(n) = 1;
```

Évaluation cible :

$$\frac{E_i, M_i \vDash_{\mathrm{e}} 1 : \texttt{int8} \Downarrow \mathsf{Int}(1) \qquad \dfrac{E_i(\mathsf{res}(n)) = \lfloor b_0 \rfloor}{E_i, M_i \vDash_{\mathrm{lv}} \mathsf{res}(n) : \texttt{int8} \Downarrow b_0, 0} \qquad \mathsf{store}(\mathsf{mtype}(\texttt{int8}), M_i, b_0, 0, \mathsf{Int}(1)) = \lfloor M_1 \rfloor}{E_i, M_i, \overline{M} \vDash_{\mathrm{s}'} \mathsf{res}(n) = 1; \Downarrow M_1, \overline{M}}$$

Définition des états mémoire :

$$\begin{aligned} M_0 &\triangleq M_i \\ \lfloor M_1 \rfloor &\triangleq \mathsf{store}(\mathrm{i}\,8, M_0, b_0, 0, \mathsf{Int}(1)) \\ M_f &\triangleq M_1 \end{aligned}$$

Par définition de M_i, on a $M_i \vDash \mathtt{i8@}b_0, 0$. On en déduit par l'axiome 2.17 que l'écriture définissant M_1 réussit. D'autre part, l'évaluation suivante assure que le résultat de la traduction encode bien $\top$, la valeur de vérité du prédicat `\true`.

$$\frac{E_i, M_1 \vDash_{\text{lv}} \mathsf{res}(n) : \mathtt{int8} \Downarrow b_0, 0 \qquad \mathsf{load}(\mathsf{mtype}(\mathtt{int8}), M_1, b_0, 0) = \lfloor \mathsf{Int}(1) \rfloor \qquad \mathsf{Int}(1) \neq \mathsf{Undef}}{E_i, M_1 \vDash_{\text{e}} \mathsf{res}(n) \Downarrow \mathsf{Int}(1)}$$

Le résultat de l'opération de lecture est donné par l'axiome 2.14, appliqué à M_1. Enfin, M_f est obtenue à partir de M_i par allocation et écriture dans les blocs alloués. Par conséquent, si M^p est telle que $M^p \hookrightarrow M_i$, alors $M^p \hookrightarrow M_f$, par définition de $\hookrightarrow$.

Il reste à montrer que $M_f \equiv M_i'$, où $M_i' \triangleq \mathsf{store}(\mathtt{i8}, M_i, b_0, 0, \mathsf{Int}(1))$. Or, cette définition de M_i' est exactement celle de M_1, c'est-à-dire de M_f. L'équivalence est donc vraie, par réflexivité.

Cas P-FALSE. Ce cas est identique au précédent, *mutatis mutandis*.

Évaluation source :

$$\text{P-FALSE}\ \frac{}{\widehat{E}, \widehat{M} \vDash_{\text{p}} \backslash\mathtt{false} \Downarrow \bot}$$

Traduction :

```
1  res(n) = 0;
```

Évaluation cible :

$$\frac{E_i, M_i \vDash_{\text{e}} 1 : \mathtt{int8} \Downarrow \mathsf{Int}(0) \qquad \dfrac{E_i(\mathsf{res}(n)) = \lfloor b_0 \rfloor}{E_i, M_i \vDash_{\text{lv}} \mathsf{res}(n) : \mathtt{int8} \Downarrow b_0, 0} \qquad \mathsf{store}(\mathsf{mtype}(\mathtt{int8}), M_i, b_0, 0, \mathsf{Int}(0)) = \lfloor M_1 \rfloor}{E_i, M_i, \overline{M} \vDash_{\text{s}'} \mathsf{res}(n) = 1; \Downarrow M_1, \overline{M}}$$

Définition des états mémoire :

$$\begin{aligned} M_0 &\triangleq M_i \\ \lfloor M_1 \rfloor &\triangleq \mathsf{store}(\mathtt{i8}, M_0, b_0, 0, \mathsf{Int}(0)) \\ M_f &\triangleq M_1 \end{aligned}$$

Par définition de M_i, on a $M_i \vDash \mathtt{i8@}b, 0$. On en déduit par l'axiome 2.17 que l'écriture définissant M_1 réussit. D'autre part, la variable de résultat est bien évaluée en $\mathsf{Int}(0)$, encodage entier de $\bot$.

$$\frac{E_i, M_1 \vDash_{\text{lv}} \mathsf{res}(n) : \mathtt{int8} \Downarrow b_0, 0 \qquad \mathsf{load}(\mathsf{mtype}(\mathtt{int8}), M_1, b_0, 0) = \lfloor \mathsf{Int}(0) \rfloor \qquad \mathsf{Int}(0) \neq \mathsf{Undef}}{E_i, M_1 \vDash_{\text{e}} \mathsf{res}(n) \Downarrow \mathsf{Int}(0)}$$

Le résultat de l'opération de lecture est donné par l'axiome 2.14, appliqué à M_1.

L'état mémoire final M_f est obtenu à partir de M_i par allocation et écriture dans les blocs alloués. Par conséquent, si M^p est tel que $M^p \hookrightarrow M_i$, alors $M^p \hookrightarrow M_f$, par définition de $\hookrightarrow$.

Il reste à montrer que $M_f \equiv M_i'$, où $M_i' \triangleq \mathsf{store}(\mathtt{i8}, M_i, b_0, 0, \mathsf{Int}(0))$. Cette définition de M_i' est exactement celle de M_1, c'est-à-dire de M_f, donc l'équivalence est vraie.

Cas P-CONJ-LEFT. Évaluation source :

$$\text{P-CONJ-LEFT}\ \frac{\widehat{E}, \widehat{M} \vDash_{\text{p}} p_1 \Downarrow \bot}{\widehat{E}, \widehat{M} \vDash_{\text{p}} p_1 \wedge p_2 \Downarrow \bot}$$

Traduction :

```
let res(n+1) : int8 in
  [[p1]]_Π^{n+1}
  if (res(n+1))
  then let res(n+1) : int8 in
    [[p2]]_Π^{n+1}
    res(n) = res(n+1);
  end else
    res(n) = 0;
end
```

Évaluation cible :

$$\text{S'-ASSIGN}\ \frac{\begin{array}{c}E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \mathsf{Int}(0) \\ E_2, M_2 \vDash_{\mathrm{lv}} \mathsf{res}(n) \Downarrow b_0, 0 \qquad E_2, M_2 \vDash_{\mathrm{e}} 0 \Downarrow \mathsf{Int}(0) \\ \mathsf{store}(\mathrm{i}8, M_2, b_0, 0, \mathsf{Int}(0)) = \lfloor M_8 \rfloor\end{array}}{E_2, M_2, \overline{M} \vDash_{s'} \mathsf{res}(n) = 0; \Downarrow M_8, \overline{M}}$$

$$\text{S'-IF-FALSE}\ \frac{E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \mathsf{Int}(0)}{E_2, M_2, \overline{M} \vDash_{s'} \texttt{if}\ (\mathsf{res}(n+1)) \ldots \texttt{else}\ \mathsf{res}(n) = 0; \Downarrow M_8, \overline{M}}$$

$$\text{S'-SEQ}\ \frac{E_1, M_1, \overline{M} \vDash_{s'} [\![p_1]\!]_{\Pi}^{n+1} \Downarrow M_2, \overline{M}}{E_1, M_1, \overline{M} \vDash_{s'} [\![p_1]\!]_{\Pi}^{n+1}\ \texttt{if} \ldots \Downarrow M_8, \overline{M}}$$

$$\text{S'-LET}\ \frac{\mathsf{alloc\_var}(E_0, M_0, \mathsf{res}(n+1), \mathtt{int8}) = E_1, M_1 \qquad \mathsf{dealloc\_var}(E_8, M_8, \mathsf{res}(n+1)) = \lfloor M_9 \rfloor}{E_0, M_0, \overline{M} \vDash_{s'} \texttt{let} \ldots \texttt{end} \Downarrow M_9, \overline{M}}$$

Définition des environnements et états mémoire :

$$\begin{array}{rcl|rcl} b_1, M_1 & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\mathrm{i}8)) & E_1 & \triangleq & \mathsf{add}(\mathsf{res}(n+1), b_1, E_0) \\ M_2 & \triangleq & \ldots & E_2 & \triangleq & E_1 \\ \lfloor M_8 \rfloor & \triangleq & \mathsf{store}(\mathrm{i}8, M_2, b_0, 0, \mathtt{int}(0)) & E_8 & \triangleq & E_2 \\ \lfloor M_9 \rfloor & \triangleq & \mathsf{free}(M_8, b_1) & E_9 & \triangleq & \mathsf{remove}(\mathsf{res}(n+1), E_8) = E_0 \end{array}$$

M_2 est définie comme le résultat de l'application de l'hypothèse d'induction sur l'évaluation de $[\![p_1]\!]_{\Pi}^{n+1}$: la seule hypothèse à vérifier pour pouvoir appliquer inductivement le théorème est $M^p \hookrightarrow M_2$. Or, $M^p \hookrightarrow M$, et M_1 est le résultat d'une allocation dans M (associée à $\mathsf{res}(n)$). Donc, par définition de $\hookrightarrow$, $M^p \hookrightarrow M_1$. Puisque $\widehat{E}, \widehat{M} \vDash_{\mathrm{p}} p_1 \Downarrow \bot$, on obtient donc l'existence de M_2 telle que :

$$\begin{cases} E_1, M_1, \overline{M} \vDash_{s'} [\![p_1]\!]_{\Pi}^{n+1} \Downarrow M_2, \overline{M} \\ E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \mathsf{Int}(0) \\ M_2 \equiv M_1' \end{cases}$$

où $M_1' \triangleq \mathsf{store}(\mathrm{i}8, M_1, b_1, 0, \mathsf{Int}(0))$.

Montrons que M_8 est bien définie, c'est-à-dire que l'écriture la définissant réussit. Il suffit de montrer que $M_2 \vDash \mathrm{i}8 @ b_0, 0$, puis d'appliquer l'axiome 2.17. Or, d'après la propriété 2.1, $M_1 \vDash \mathrm{i}8 @ b_0, 0$, donc d'après la propriété 2.3 $M_1' \vDash \mathrm{i}8 @ b_0, 0$. Puisque $M_2 \equiv M_1'$, on a donc $M_2 \vDash \mathrm{i}8 @ b_0, 0$.

Pour justifier l'existence de M_9, il suffit de montrer que $M_8 \vDash b_1$, et d'appliquer ensuite l'axiome 2.5. $M_1 \vDash b_1$ d'après l'axiome 2.2, donc $M_1' \vDash b_1$ d'après l'axiome 2.6. Puisque $M_2 \equiv M_1'$, on a donc $M_2 \vDash b_1$ et, d'après l'axiome 2.6, $M_8 \vDash b_1$.

Il reste à montrer $M_9 \equiv M_0'$, où $M_0' \triangleq \mathsf{store}(\mathrm{i}8, M_0, E_0(\mathsf{res}(n)), 0, \mathsf{Int}(0))$. Considérons l'équivalence $M_2 \equiv M_1'$, obtenue précédemment par hypothèse d'induction. Dans l'évaluation cible, M_2 fait l'objet d'une écriture dans le bloc b_0, dont résulte M_8, puis le bloc $b_1 \neq b_0$ est déalloué dans M_8, ce qui donne M_9. L'application de la propriété 4.6 à cette séquence d'opérations assure l'existence d'états mémoire M_8' et M_9' tels que :

$$\begin{cases} M_8' = \mathsf{free}(M_1', b_1) \\ M_9' = \mathsf{store}(\mathrm{i}8, M_8', b_0, 0, \mathsf{Int}(0)) \\ M_9' \equiv M_2. \end{cases}$$

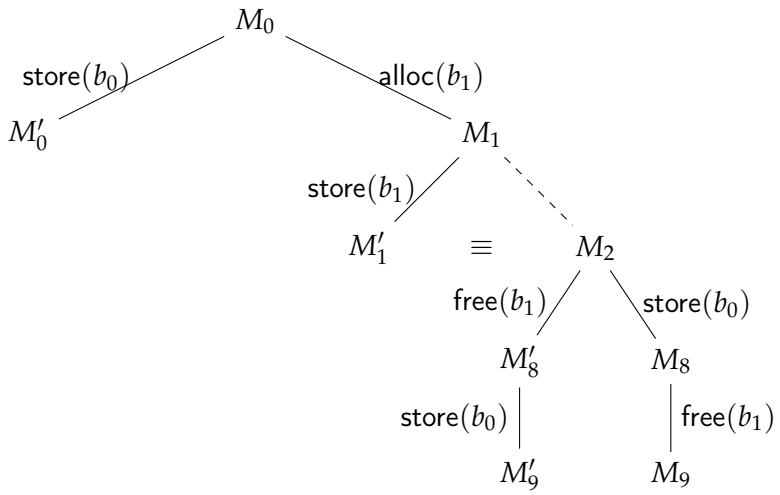

FIGURE 4.3 – Relations entre les états mémoire de l'évaluation cible dans le cas P-CONJ-LEFT. Les opérations sur la mémoire sont abrégées pour ne mentionner que le type d'opération (allocation, déallocation, écriture) et le bloc concerné.

En outre, M'_8 est obtenu à partir de M_0 par une succession de trois opérations : l'allocation d'un bloc b_1 (définition de M_1), une écriture dans ce même bloc (définition de M'_1), puis sa déallocation (définition de M'_8). On peut donc appliquer la propriété 4.8, pour obtenir l'équivalence $M'_8 \equiv M_0$. On a alors :

$$\begin{cases} M'_0 = \mathsf{store}(\mathrm{i}8, M_0, b_0, 0, \mathsf{Int}(0)) \\ M'_9 = \mathsf{store}(\mathrm{i}8, M'_8, b_0, 0, \mathsf{Int}(0)) \\ M'_8 \equiv M_0 \end{cases}$$

ce qui permet d'appliquer la propriété 4.10 pour conclure.

Cas P-CONJ-RIGHT. Évaluation source :

$$\text{P-CONJ-RIGHT}\ \frac{\widehat{E}, \widehat{M} \vDash_{\mathrm{p}} p_1 \Downarrow \top \qquad \widehat{E}, \widehat{M} \vDash_{\mathrm{p}} p_2 \Downarrow \beta}{\widehat{E}, \widehat{M} \vDash_{\mathrm{p}} p_1 \wedge p_2 \Downarrow \beta}$$

Traduction :

```
1  let res(n+1) : int8 in
2    [[p1]]^{n+1}_Π
3    if (res(n+1))
4    then let res(n+1) : int8 in
5      [[p2]]^{n+1}_Π
6      res(n) = res(n+1);
7    end else
8      res(n) = 0;
9  end
```

Évaluation cible :

$$
\textsc{S'-let}\ \dfrac{\mathsf{alloc\_var}(E_0, M_0, \mathsf{res}(n+1), \mathtt{int8}) = E_1, M_1 \qquad \textsc{S'-seq}\ \dfrac{E_1, M_1, \overline{M} \vDash_{s'} [\![p_1]\!]_{\Pi}^{n+1} \Downarrow M_2, \overline{M} \qquad \textsc{S'-if-true}\ \dfrac{E_2, M_2 \vDash_{e} \mathsf{res}(n+1) \Downarrow \mathsf{Int}(q) \qquad q \neq 0 \qquad \textsc{S'-let}\ \dfrac{\mathsf{alloc\_var}(E_3, M_3, \mathsf{res}(n+1), \mathtt{int8}) = E_4, M_4 \qquad \textsc{S'-seq}\ \dfrac{E_4, M_4, \overline{M} \vDash_{s'} [\![p_2]\!]_{\Pi}^{n+1} \Downarrow M_5, \overline{M} \qquad \textsc{S'-assign}\ \dfrac{\mathsf{Int}(r) \neq \mathsf{Undef} \quad E_5, M_5 \vDash_{lv} \mathsf{res}(n) \Downarrow b_0, 0 \quad E_5, M_5 \vDash_{e} \mathsf{res}(n+1) \Downarrow \mathsf{Int}(r) \quad \mathsf{store}(\mathrm{i}\,8, M_2, b_0, 0, \mathsf{Int}(r)) = \lfloor M_6 \rfloor}{E_5, M_5, \overline{M} \vDash_{s'} \mathsf{res}(n) = \mathsf{res}(n+1); \Downarrow M_6, \overline{M}}}{E_4, M_4, \overline{M} \vDash_{s'} [\![p_2]\!]_{\Pi}^{n+1}\ \mathsf{res}(n) = \mathsf{res}(n+1); \Downarrow M_6, \overline{M}} \qquad \mathsf{dealloc\_var}(E_6, M_6, \mathsf{res}(n+1)) = \lfloor M_7 \rfloor}{E_2, M_2, \overline{M} \vDash_{s'} \mathtt{let} \ldots \mathtt{end} \Downarrow M_7, \overline{M}}}{E_2, M_2, \overline{M} \vDash_{s'} \mathtt{if}\ (\mathsf{res}(n+1))\ \mathtt{then} \ldots \mathtt{else} \ldots \Downarrow M_8, \overline{M}}}{E_1, M_1, \overline{M} \vDash_{s'} [\![p_1]\!]_{\Pi}^{n+1}\ \mathtt{if} \ldots \Downarrow M_8, \overline{M}} \qquad \mathsf{dealloc\_var}(E_8, M_8, \mathsf{res}(n+1)) = \lfloor M_9 \rfloor}{E_0, M_0, \overline{M} \vDash_{s'} \mathtt{let} \ldots \mathtt{end} \Downarrow M_9, \overline{M}}
$$

Définition des environnements et états mémoire :

$$
\begin{array}{rcl|rcl}
b_1, M_1 & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\mathrm{i}\,8)) & E_1 & \triangleq & \mathsf{add}(\mathsf{res}(n+1), b_1, E_0) \\
M_2 & \triangleq & \ldots & E_2 & \triangleq & E_1 \\
M_3 & \triangleq & M_2 & E_3 & \triangleq & E_2 \\
b_4, M_4 & \triangleq & \mathsf{alloc}(M_3, \mathsf{sizeof}(\mathrm{i}\,8)) & E_4 & \triangleq & \mathsf{add}(\mathsf{res}(n+1), b_4, E_3) \\
M_5 & \triangleq & \ldots & E_5 & \triangleq & E_4 \\
\lfloor M_6 \rfloor & \triangleq & \mathsf{store}(\mathrm{i}\,8, M_5, b_0, 0, \mathsf{Int}(r)) & E_6 & \triangleq & E_5 \\
\lfloor M_7 \rfloor & \triangleq & \mathsf{free}(M_6, b_4) & E_7 & \triangleq & \mathsf{remove}(\mathsf{res}(n+1), E_6) = E_4 \\
M_8 & \triangleq & M_7 & E_8 & \triangleq & E_7 \\
\lfloor M_9 \rfloor & \triangleq & \mathsf{free}(M_8, b_1) & E_9 & \triangleq & \mathsf{remove}(\mathsf{res}(n+1), E_8) = E_0
\end{array}
$$

De même que dans le cas précédent, par application de l'hypothèse d'induction sur l'évaluation source $\widehat{E}, \widehat{M} \vDash_{p} p_1 \Downarrow \top$, on obtient l'existence de M_2, état mémoire vérifiant les propriétés suivantes :

$$
\begin{cases}
E_1, M_1, \overline{M} \vDash_{s'} [\![p_1]\!]_{\Pi}^{n+1} \Downarrow M_2, \overline{M} \\
E_1, M_2 \vDash_{e} \mathsf{res}(n+1) \Downarrow \mathsf{Int}(1) \\
M_2 \equiv M_1'.
\end{cases}
$$

En particulier, l'évaluation d'expression $E_2, M_2 \vDash_{e} \mathsf{res}(n+1) \Downarrow \mathsf{Int}(1)$, permet l'application de la règle S'-IF-TRUE.

Similairement à M_2, M_5 est défini par application de l'hypothèse d'induction sur l'évaluation source $\widehat{E}, \widehat{M} \vDash_{p} p_2 \Downarrow \beta$, avec les propriétés suivantes :

$$
\begin{cases}
E_4, M_4, \overline{M} \vDash_{s'} [\![p_2]\!]_{\Pi}^{n+1} \Downarrow M_5, \overline{M} \\
E_5, M_5 \vDash_{e} \mathsf{res}(n+1) \Downarrow \mathsf{Int}(\beta) \\
M_5 \equiv M_4'.
\end{cases}
$$

Montrons la validité de la définition de M_6, c'est-à-dire l'accès valide suivant : $M_5 \vDash \mathrm{i}\,8 @ b_0, 0$, où $b_0 \triangleq E(\mathsf{res}(n))$. On rappelle que, par définition de M_0, on a $M_0 \vDash \mathrm{i}\,8 @ b_0, 0$: il s'agit donc de montrer que la validité est préservée par les opérations mémoires subséquentes. D'après la propriété 2.1, l'accès est encore valide dans M_1. Par application de la propriété 2.3, il l'est aussi dans M_1', donc dans M_2, puisque les accès valides sont les mêmes dans deux mémoires équivalentes. Une nouvelle application de la propriété 2.1 permet d'obtenir $M_4 \vDash \mathrm{i}\,8 @ b_0, 0$. On en déduit $M_4' \vDash \mathrm{i}\,8 @ b_0, 0$ par une nouvelle application de la propriété 2.3. Par équivalence, $M_5 \vDash \mathrm{i}\,8 @ b_0, 0$.

Montrons à présent la validité de la définition de M_7, c'est-à-dire la proposition $M_6 \vDash b_4$. b_4 est valide dans M_4 par l'axiome 2.2. Sa validité est préservée dans M'_4 d'après l' axiome 2.6, et donc dans M_5 par équivalence. Une seconde application de l'axiome 2.6 permet de conclure.

Concernant la définition de M_9, il s'agit de montrer $M_8 \vDash b_1$. b_1 est valide dans M_1 en tant que nouveau bloc alloué (axiome 2.2). D'après l' axiome 2.6, il est encore valide dans $M_{1'}$, donc par équivalence, dans M_2. D'après axiome 2.2, le bloc est encore valide dans M_4, puis dans M'_4 d'après l'axiome 2.6, et dans M_5 par équivalence. À nouveau, l'axiome 2.6 assure que b_1 est valide dans M_6, ce qui reste vrai dans M_7 d'après l'axiome 2.3.

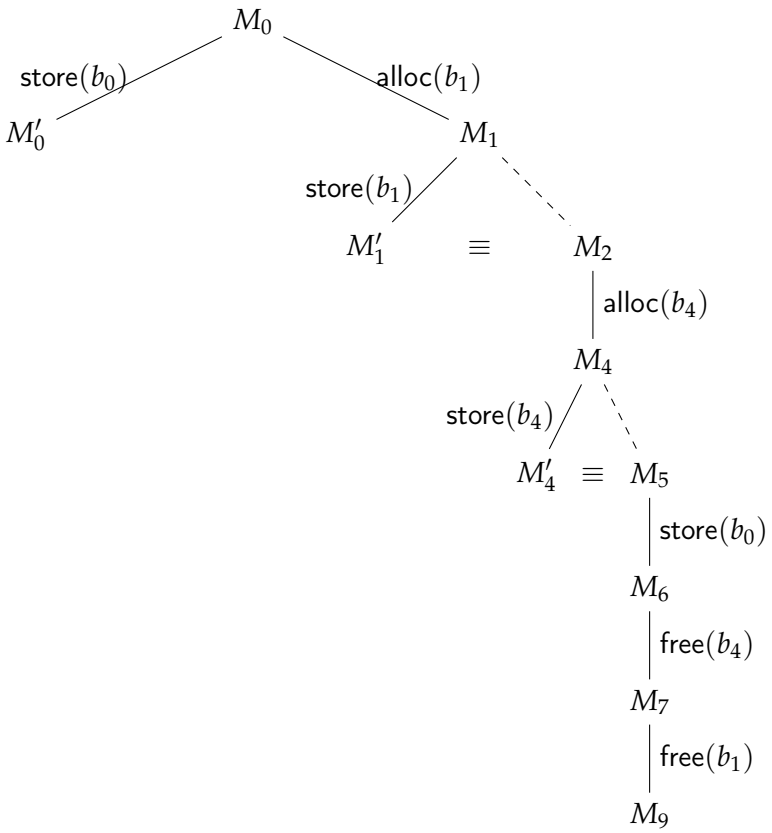

FIGURE 4.4 – Relations entre les états mémoire de l'évaluation cible dans le cas P-CONJ-RIGHT. Les opérations sur la mémoire sont abrégées pour ne mentionner que le type d'opération (allocation, déallocation, écriture) et le bloc concerné.

Il reste à montrer que $M_9 \equiv M'_0$. Pour ce faire, on utilise les différentes propriétés de la relation d'équivalence définie dans la section 4.2.2. La figure 4.4 représente les différentes opérations mémoires liant M_9 à M'_0.

À l'aide des propriétés 4.9 et 4.10, on définit les états fictifs M_{41}, M_{42} et M_{43}. On reporte ainsi sur l'état $M_{4'}$ les opérations définissant M_6, M_7 et M_9, et les états résultants leur sont respectivement équivalents.

$$\begin{array}{lcl|l} \lfloor M_{46} \rfloor & \triangleq & \mathsf{store}(\mathrm{i}8, M'_4, b_0, 0, \mathsf{Int}(r)) & M_{46} \equiv M_6 \\ \lfloor M_{47} \rfloor & \triangleq & \mathsf{free}(M_{46}, b_4) & M_{47} \equiv M_7 \\ \lfloor M_{49} \rfloor & \triangleq & \mathsf{free}(M_{47}, b_1) & M_{49} \equiv M_9 \end{array}$$

On réordonne maintenant l'écriture et les deux déallocations, en appliquant deux fois la propriété 4.6.

$$\begin{array}{rcl|l} \lfloor M'_{46} \rfloor & \triangleq & \mathsf{free}(M'_4, b_4) & \\ \lfloor M'_{47} \rfloor & \triangleq & \mathsf{free}(M'_{46}, b_1) & \\ \lfloor M'_{49} \rfloor & \triangleq & \mathsf{store}(\mathrm{i}8, M'_{47}, b_0, 0, \mathsf{Int}(r)) & M'_{49} \equiv M_{49} \end{array}$$

M'_{46} est obtenue à partir de M_2 par l'allocation du bloc b_4, une écriture dans ce bloc, puis sa déallocation. D'après la propriété 4.8, on a donc $M'_{46} \equiv M_2$, donc $M'_{46} \equiv M'_1$.

On définit les deux états fictifs suivants à partir de M'_1 :

$$\begin{array}{rcl|l} \lfloor M'_{17} \rfloor & \triangleq & \mathsf{free}(M'_1, b_1) & M'_{17} \equiv M'_{47} \\ \lfloor M'_{19} \rfloor & \triangleq & \mathsf{store}(\mathrm{i}8, M'_{17}, b_0, 0, \mathsf{Int}(r)) & M'_{19} \equiv M'_{49} \end{array}$$

M'_{17} est obtenu à partir de M_0 par allocation, écriture, puis déallocation du même bloc b_1. D'après la propriété 4.8, on a donc $M'_{17} \equiv M_0$. Par conséquent, d'après la propriété 4.10, $M'_{17} \equiv M'_0$. Cette dernière équivalence complète la chaîne suivante : $M'_0 \equiv M'_{17} \equiv M'_{47} \equiv M_{49} \equiv M_9$.

Cas P-NEG-FALSE. Évaluation source :

$$\text{P-NEG-FALSE}\ \frac{\widehat{E}, \widehat{M} \vDash_{\mathrm{p}} p \Downarrow \top}{\widehat{E}, \widehat{M} \vDash_{\mathrm{p}} \neg p \Downarrow \bot}$$

Traduction :

```
1  let res(n+1) : int8 in
2    [[p]]_Π^{n+1}
3    res(n) = ! res(n+1);
4  end
```

Évaluation cible :

$$\dfrac{\dfrac{E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![p]\!]_{\Pi}^{n+1} \Downarrow M_2, \overline{M} \qquad \dfrac{\dfrac{E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \mathsf{Int}(1) \quad \mathsf{sem\_unop}(!, \mathsf{Int}(1)) = \mathsf{Int}(0)}{E_2, M_2 \vDash_{\mathrm{e}} !\mathsf{res}(n+1) \Downarrow \mathsf{Int}(0)} \quad E_2, M_2 \vDash_{\mathrm{lv}} \mathsf{res}(n) \Downarrow b_0, 0 \quad \mathsf{Int}(0) \neq \mathsf{Undef} \quad \lfloor M_3 \rfloor = \mathsf{store}(\mathrm{i}8, M_2, b_0, 0, \mathsf{Int}(0))}{E_2, M_2, \overline{M} \vDash_{\mathrm{s}'} \mathsf{res}(n) = !\mathsf{res}(n+1); \Downarrow M_3, \overline{M}}}{E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![p]\!]_{\Pi}^{n+1}\ \mathsf{res}(n) = !\mathsf{res}(n+1); \Downarrow M_3, \overline{M}} \quad \mathsf{alloc\_var}(E_0, M_0, \mathsf{res}(n), \mathtt{int8}) = E_1, M_1 \quad \mathsf{dealloc\_var}(E_3, M_3, \mathsf{res}(n+1)) = \lfloor M_4 \rfloor}{E_0, M_0, \overline{M} \vDash_{\mathrm{s}'} \mathtt{let} \ldots \mathtt{end} \Downarrow M_4, \overline{M}}$$

Définition des environnements et états mémoire :

$$\begin{array}{rcl|rcl} b_1, M_1 & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\mathrm{i}8)) & E_1 & \triangleq & \mathsf{add}(\mathsf{res}(n+1), b_1, E_0) \\ M_2 & \triangleq & \ldots & E_2 & \triangleq & E_1 \\ \lfloor M_3 \rfloor & \triangleq & \mathsf{store}(\mathrm{i}8, M_2, b, 0, \mathsf{Int}(1)) & E_3 & \triangleq & E_2 \\ \lfloor M_4 \rfloor & \triangleq & \mathsf{free}(M_3, b_1) & E_4 & \triangleq & E_0 \end{array}$$

On applique l'hypothèse d'induction à l'évaluation source $\widehat{E}, \widehat{M} \vDash_{\mathrm{p}} p \Downarrow \top$, et on obtient les propositions suivantes :

$$\begin{cases} E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![p]\!]_{\Pi}^{n+1} \Downarrow M_2, \overline{M} \\ E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \mathsf{Int}(1) \\ M_2 \equiv M'_1 \end{cases}$$

avec $M_1' \triangleq \mathsf{store}(\mathrm{i}\,8, M_1, b_1, 0, \mathsf{Int}(1))$.

Afin de pouvoir définir M_3 par écriture, montrons que $M_2 \vDash \mathrm{i}\,8\,@\,b_0, 0$. Comme dans tous les cas de l'induction, on a $M_0 \vDash \mathrm{i}\,8\,@\,b_0, 0$ par définition de M_0. D'après les axiomes 2.2 et 2.8, on a encore $M_1 \vDash \mathrm{i}\,8\,@\,b_0, 0$. La validité est préservée par écriture d'après la propriété 2.3, par conséquent $M_1' \vDash \mathrm{i}\,8\,@\,b_0, 0$. On en déduit par équivalence $M_2 \vDash \mathrm{i}\,8\,@\,b_0, 0$.

Montrons maintenant que $M_3 \vDash b_1$, afin de justifier la définition de M_3. D'après l'axiome 2.2, $M_1 \vDash b_1$. La validité est préservée par écriture (axiome 2.6), donc $M_1' \vDash b_1$, ce qui implique par équivalence $M_2 \vDash b_1$. Une nouvelle application de l'axiome 2.6 permet de déduire $M_3 \vDash b_1$.

Il reste à montrer que $M_4 \equiv M_0'$. Pour ce faire, on introduit les états mémoire fictifs M_3' et M_4', obtenus à partir de M_2 en permutant les opérations d'écriture et de déallocation (voir la figure 4.5). D'après la propriété 4.6, $M_4' \equiv M_4$. On définit ensuite M_3'' et M_4'' en reportant déallocation et écriture sur M_1'. Les propriétés 4.9 et 4.10 donnent alors l'équivalence $M_4'' \equiv M_4'$. De plus, d'après la propriété 4.8, on a $M_0 \equiv M_3''$. Il ne reste alors qu'à appliquer la propriété 4.10 pour obtenir $M_0' \equiv M_4''$, et conclure par transitivité de l'équivalence.

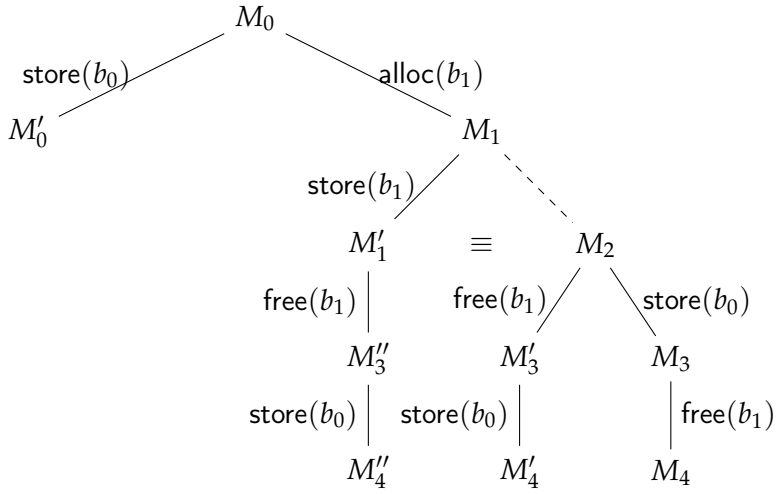

FIGURE 4.5 – Relations entre les états mémoire de l'évaluation cible dans le cas P-NEG-FALSE. Les opérations sur la mémoire sont abrégées pour ne mentionner que le type d'opération (allocation, déallocation, écriture) et le bloc concerné.

Cas P-NEG-TRUE. Cas complémentaire du précédent.

$$\frac{E, M \vDash_{\mathrm{p}} p \Downarrow \bot}{E, M \vDash_{\mathrm{p}} \neg p \Downarrow \top}\ \text{P-NEG-TRUE}$$

Traduction :

```
1  let res(n + 1) : int8 in
2    [[p]]^{n+1}_Π
3    res(n) = ! res(n + 1);
4  end
```

Évaluation cible :

$$\dfrac{\dfrac{E_1, M_1, \overline{M} \vDash_{s'} [\![p]\!]_{\Pi}^{n+1} \Downarrow M_2, \overline{M} \qquad \dfrac{\dfrac{E_2, M_2 \vDash_{e} \mathsf{res}(n+1) \Downarrow \mathsf{Int}(0)}{E_2, M_2 \vDash_{e} !\mathsf{res}(n+1) \Downarrow \mathsf{Int}(1)} \quad \mathsf{sem\_unop}(!, \mathsf{Int}(0)) = \mathsf{Int}(1)}{E_2, M_2, \overline{M} \vDash_{s'} \mathsf{res}(n) = !\mathsf{res}(n+1); \Downarrow M_3, \overline{M}}}{E_1, M_1, \overline{M} \vDash_{s'} [\![p]\!]_{\Pi}^{n+1} \ \mathsf{res}(n) = !\mathsf{res}(n+1); \Downarrow M_3, \overline{M}} \quad \dfrac{E_2, M_2 \vDash_{lv} \mathsf{res}(n) \Downarrow b_0, 0 \qquad \mathsf{Int}(0) \neq \mathsf{Undef} \quad \lfloor M_3 \rfloor = \mathsf{store}(\mathrm{i}8, M_2, b_0, 0, \mathsf{Int}(1))}{\mathsf{alloc\_var}(E_0, M_0, \mathsf{res}(n), \mathtt{int8}) = E_1, M_1 \qquad \mathsf{dealloc\_var}(E_3, M_3, \mathsf{res}(n+1)) = \lfloor M_4 \rfloor}}{E_0, M_0, \overline{M} \vDash_{s'} \mathtt{let} \ldots \mathtt{end} \Downarrow M_4, \overline{M}}$$

Définition des environnements et états mémoire :

$$\begin{array}{rcl|rcl} b_1, M_1 & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\mathrm{i}8)) & E_1 & \triangleq & \mathsf{add}(\mathsf{res}(n+1), b_1, E_0) \\ M_2 & \triangleq & \ldots & E_2 & \triangleq & E_1 \\ \lfloor M_3 \rfloor & \triangleq & \mathsf{store}(\mathrm{i}8, M_2, b_0, 0, \mathsf{Int}(1)) & E_3 & \triangleq & E_2 \\ \lfloor M_4 \rfloor & \triangleq & \mathsf{free}(M_3, b_1) & E_4 & \triangleq & E_0 \end{array}$$

On applique l'hypothèse d'induction à l'évaluation source $\widehat{E}, \widehat{M} \vDash_{p} p \Downarrow \top$, et on obtient les propositions suivantes :

$$\begin{cases} E_1, M_1, \overline{M} \vDash_{s'} [\![p]\!]_{\Pi}^{n+1} \Downarrow M_2, \overline{M} \\ E_2, M_2 \vDash_{e} \mathsf{res}(n+1) \Downarrow \mathsf{Int}(0) \\ M_2 \equiv M_1' \end{cases}$$

avec $M_1' \triangleq \mathsf{store}(\mathrm{i}8, M_1, b_1, 0, \mathsf{Int}(0))$.

L'écriture définissant M_3, ainsi que la déallocation définissant M_4, sont justifiées par le même raisonnement qu'au cas précédent. Similairement, l'équivalence $M_4 \equiv M_0'$ est démontrée comme au cas précédent.

Cas P-VALID.

$$\text{P-VALID} \ \dfrac{E, M \vDash_{t} t : \tau* \Downarrow \mathsf{Ptr}(\widehat{b}, \widehat{\delta}) \quad M \vDash \mathsf{mtype}(\tau*) \,@\, \widehat{b}, \widehat{\delta}}{E, M \vDash_{p} \backslash\mathtt{valid}(t) \Downarrow \top}$$

Traduction :

```
1  let res(n+1) : τ* in
2    [[t]]_T^{n+1}
3    res(n) = is_valid(res(n+1));
4  end
```

Évaluation cible :

$$\text{S'-SEQ} \ \dfrac{\text{S'-IS-VALID} \ \dfrac{E_1, M_1 \vDash_{e} \mathsf{res}(n+1) \Downarrow \mathsf{Ptr}(b, \delta) \qquad E_1, M_1 \vDash_{lv} \mathsf{res}(n) \Downarrow b_0, 0 \quad \mathsf{is\_valid\_access}(\mathsf{mtype}(\tau), \overline{M}, b, \delta) = \mathsf{true} \quad \mathsf{store}(\mathrm{i}8, M_2, b_0, 0, \mathsf{Int}(1)) = \lfloor M_3 \rfloor}{E_2, M_2, \overline{M} \vDash_{s'} \mathsf{res}(n) = \mathtt{is\_valid}(\mathsf{res}(n+1)); \Downarrow M_3, \overline{M}} \quad E_1, M_1, \overline{M} \vDash_{s'} [\![t]\!]_{T}^{n+1} \Downarrow M_2, \overline{M}}{E_1, M_1, \overline{M} \vDash_{s'} [\![t]\!]_{T}^{n+1} \ \mathsf{res}(n) = \mathtt{is\_valid}(\mathsf{res}(n+1)); \Downarrow M_3, \overline{M}}$$

$$\text{S'-LET} \ \dfrac{\mathsf{alloc\_var}(E_0, M_0, \mathsf{res}(n+1), \tau*) = E_1, M_1 \qquad \mathsf{dealloc\_var}(E_3, M_3, \mathsf{res}(n+1)) = \lfloor M_4 \rfloor}{E_0, M_0, \overline{M} \vDash_{s'} \mathtt{let} \ldots \mathtt{end} \Downarrow M_4, \overline{M}}$$

Définition des environnements et états mémoire :

$$\begin{array}{rcl|rcl} b_1, M_1 & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\mathrm{i}\,64)) & E_1 & \triangleq & \mathsf{add}(\mathsf{res}(n+1), b_1, E_0) \\ M_2 & \triangleq & \ldots & E_2 & \triangleq & E_1 \\ \lfloor M_3 \rfloor & \triangleq & \mathsf{store}(\mathrm{i}\,8, M_2, b_0, 0, \mathsf{Int}(1)) & E_3 & \triangleq & E_2 \\ \lfloor M_4 \rfloor & \triangleq & \mathsf{free}(M_3, b_1) & E_4 & \triangleq & \mathsf{remove}(\mathsf{res}(n+1), E_3) = E_0 \end{array}$$

M_2 est définie par application du théorème 3 à $\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} t \Downarrow \mathsf{Ptr}(\widehat{b}, \widehat{\delta})$, et vérifie donc les propositions suivantes :

$$\begin{cases} E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![t]\!]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M} \\ E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta})) \\ M_2 \equiv M_1' \end{cases}$$

où σ est la permutation associée à $\widehat{M} \sim M^p$, et $\lfloor M_1' \rfloor \triangleq \mathsf{store}(\mathsf{mtype}(\tau*), M_1, b_1, 0, \dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta})))$. On utilise ce résultat pour déterminer par identification b et δ de l'évaluation cible, en posant $\mathsf{Ptr}(b, \delta) = \dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta}))$, soit $b \triangleq \sigma(\widehat{b})$ et $\delta \triangleq \widehat{\delta}$.

M_3 est bien définie si $M_2 \vDash \mathrm{i}\,8 @ b_0, 0$. Par définition de M_0, l'accès est initialement valide : $M_0 \vDash \mathrm{i}\,8 @ b_0, 0$; d'après les axiomes 2.2 et 2.8, on a encore $M_1 \vDash \mathrm{i}\,8 @ b_0, 0$. D'après les axiomes 2.6 et 2.10, l'accès est encore valide dans M_1', donc dans M_2 par équivalence : $M_2 \vDash \mathrm{i}\,8 @ b_0, 0$.

M_4 est bien définie si $M_3 \vDash b_1$. Le bloc b_1 est valide dans M_1 en vertu de l'axiome 2.2 ; il le reste dans M_1' d'après l'axiome 2.6 et, par équivalence, dans M_2 : $M_2 \vDash b_1$. On en déduit en $M_3 \vDash b_1$ par un nouvelle application de l'axiome 2.6.

Par ailleurs, l'accès valide source $\widehat{M} \vDash \mathsf{mtype}(\tau*) @ \widehat{b}, \widehat{\delta}$, couplé à l'isomorphisme $\widehat{M} \sim M^p$, assure l'accès valide $M^p \vDash \mathsf{mtype}(\tau*) @ b, \delta$. Puisque $M^p \triangleright \overline{M}$, on a donc nécessairement $\mathsf{is\_valid\_access}(\mathsf{mtype}(\tau), \overline{M}, b, \delta') = \mathsf{true}$ (prémisse de la règle S'-IS-VALID).

Montrons maintenant que $M_4 \equiv M_0'$. On définit (voir la figure 4.6) les états fictifs M_3' et $M_{4'}$ obtenus en permutant la déallocation de b_1 (définition de M_4) et l'écriture dans b (définition de M_3) ; en outre, on définit M_3'' et M_4'' en reportant ces mêmes opérations à partir de M_1'. L'état M_4' vérifie alors $M_4' \equiv M_4$, d'après la propriété 4.6. De plus, M_4'' vérifie $M_4'' \equiv M_4'$, d'après les propriétés 4.9 et 4.10. On applique ensuite la propriété 4.8 à la séquence allocation-écriture-déallocation résultante pour obtenir $M_3' \equiv M_0$. On en déduit $M_4' \equiv M_0'$ par application de la propriété 4.10, ce qui établit l'équivalence recherchée.

Cas P-INVALID. Ce cas est le complémentaire du précédent.

$$\text{P-INVALID}\quad \frac{E, M \vDash_{\mathrm{t}} t : \tau* \Downarrow \mathsf{Ptr}(\widehat{b}, \widehat{\delta}) \qquad M \nvDash \mathsf{mtype}(\tau) @ \widehat{b}, \widehat{\delta}}{E, M \vDash_{\mathrm{p}} \backslash\mathtt{valid}(t) \Downarrow \bot}$$

Traduction :

```
1  let res(n+1) : τ* in
2    [[t]]_T^{n+1}
3    res(n) = is_valid(res(n+1));
4  end
```

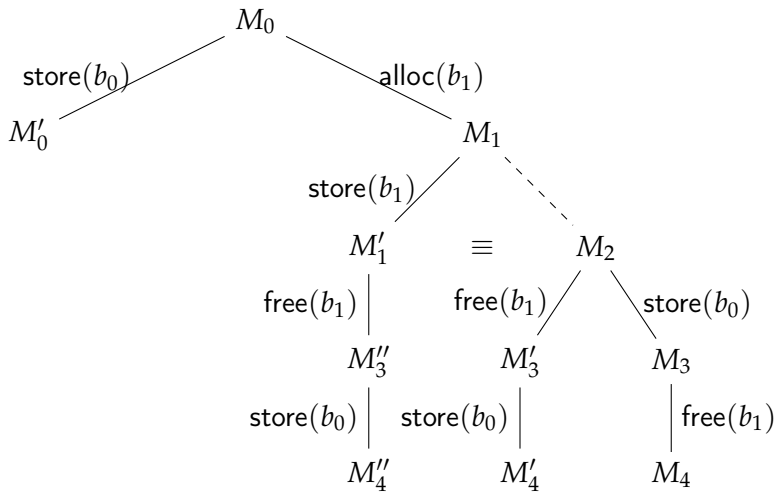

FIGURE 4.6 – Relations entre les états mémoire de l'évaluation cible dans le cas P-VALID. Les opérations sur la mémoire sont abrégées pour ne mentionner que le type d'opération (allocation, déallocation, écriture) et le bloc concerné.

Évaluation cible :

$$\text{S'-LET}\ \frac{\text{S'-SEQ}\ \dfrac{\text{S'-IS-VALID}\ \dfrac{\begin{array}{c}E_1, M_1 \vDash_{\text{e}} \mathsf{res}(n+1) \Downarrow \mathsf{Ptr}(b,\delta) \qquad E_1, M_1 \vDash_{\text{lv}} \mathsf{res}(n) \Downarrow b_0, 0 \\ \mathsf{is\_valid\_access}(\mathsf{mtype}(\tau), \overline{M}, b, \delta) = \mathsf{false} \\ \mathsf{store}(\mathrm{i}8, M_2, b_0, 0, \mathsf{Int}(0)) = \lfloor M_3 \rfloor\end{array}}{E_2, M_2, \overline{M} \vDash_{\text{s}'} \mathsf{res}(n) = \mathtt{is\_valid}(\mathsf{res}(n+1)); \Downarrow M_3, \overline{M}}}{E_1, M_1, \overline{M} \vDash_{\text{s}'} [\![t]\!]_{\text{T}}^{n+1} \ \mathsf{res}(n) = \mathtt{is\_valid}(\mathsf{res}(n+1)); \Downarrow M_3, \overline{M}}}{E_0, M_0, \overline{M} \vDash_{\text{s}'} \mathtt{let} \ldots \mathtt{end} \Downarrow M_4, \overline{M}}$$

Définition des environnements et états mémoire :

$$\begin{array}{rcl|rcl} b_1, M_1 & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\mathrm{i}64)) & E_1 & \triangleq & \mathsf{add}(\mathsf{res}(n+1), b_1, E_0) \\ M_2 & \triangleq & \ldots & E_2 & \triangleq & E_1 \\ \lfloor M_3 \rfloor & \triangleq & \mathsf{store}(\mathrm{i}8, M_2, b_0, 0, \mathsf{Int}(0)) & E_3 & \triangleq & E_2 \\ \lfloor M_4 \rfloor & \triangleq & \mathsf{free}(M_3, b_1) & E_4 & \triangleq & \mathsf{remove}(\mathsf{res}(n+1), E_3) = E_0 \end{array}$$

On applique le théorème 3 à $\widehat{E}, \widehat{M} \vDash_{\text{t}} t \Downarrow \mathsf{Ptr}(\widehat{b}, \widehat{\delta})$, et on obtient les propositions suivantes :

$$\begin{cases} E_1, M_1, \overline{M} \vDash_{\text{s}'} [\![t]\!]_{\text{T}}^{n+1} \Downarrow M_2, \overline{M} \\ E_2, M_2 \vDash_{\text{e}} \mathsf{res}(n+1) \Downarrow \dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta})) \\ M_2 \equiv M_1' \end{cases}$$

où σ est la permutation associée à $\widehat{M} \sim M^p$, et $\lfloor M_1' \rfloor = \mathsf{store}(\mathsf{mtype}(\tau*), M, b_1, 0, \dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta})))$. Comme dans le cas précédent, on identifie $\mathsf{Ptr}(b, \delta)$ et $\dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta}))$, ce qui définit $b \triangleq \sigma(\widehat{b})$ et $\delta \triangleq \widehat{\delta}$.

$\widehat{M} \not\vDash \mathsf{mtype}(\tau) @ \widehat{b}, \widehat{\delta}$ et $\widehat{M} \sim M^p$ donc $M^p \not\vDash \mathsf{mtype}(\tau) @ b, \delta$. De plus, $M^p \triangleright \overline{M}$, donc $\mathsf{is\_valid\_access}(\overline{M}, b) = \mathsf{false}$.

M_3 est bien définie si $M_2 \vDash \mathrm{i}\,8\,@\,b_0, 0$. Par définition de M_0, on a initialement $M_0 \vDash \mathrm{i}\,8\,@\,b_0, 0$. D'après la propriété 2.1, l'accès est encore valide après allocation : $M_1 \vDash \mathrm{i}\,8\,@\,b_0, 0$. D'après la propriété 2.3, on a de plus $M_1' \vDash \mathrm{i}\,8\,@\,b_0, 0$. Par suite de l'équivalence $M_2 \equiv M_1'$, on a donc $M_2 \vDash \mathrm{i}\,8\,@\,b_0, 0$.

M_4 est bien définie si $M_3 \vDash b_1$. Initialement, $M_1 \vDash b_1$ (axiome 2.2). D'après axiome 2.6, la validité est préservée après écriture : $M_1' \vDash b_1$, donc par équivalence $M_2 \vDash b_1$. Une nouvelle application de l' axiome 2.6 permet d'établir la validité dans M_4.

Il reste à montrer que $M_4 \equiv M_0'$, ce qui peut être établi par le même raisonnement que dans le cas précédent, *mutatis mutandis*.

Cas P-INITIALIZED.

$$\textsc{P-initialized}\ \frac{\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} t : \tau* \Downarrow \mathsf{Ptr}(\widehat{b}, \widehat{\delta}) \qquad \mathsf{load}(\mathsf{mtype}(\tau), \widehat{M}, \widehat{b}, \widehat{\delta}) = \lfloor \widehat{v} \rfloor \qquad \widehat{v} \neq \mathsf{Undef}}{\widehat{E}, \widehat{M} \vDash_{\mathrm{p}} \backslash\texttt{initialized}(t) \Downarrow \top}$$

Traduction :

```
1  let res(n+1) : τ in
2    [[t]]_T^{n+1}
3    res(n) = is_initialized(res(n+1));
4  end
```

Évaluation cible :

$$\textsc{S'-is-initialized}\ \frac{\begin{array}{c} E_1, M_1 \vDash_{\mathrm{lv}} \mathsf{res}(n) \Downarrow b_0, 0 \qquad E_1, M_1 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \mathsf{Ptr}(b, \delta) \\ \mathsf{is\_initialized}(\mathsf{mtype}(\tau), \overline{M}, b, \delta) = \mathsf{true} \\ \mathsf{store}(\mathrm{i}\,8, M_2, b_0, 0, \mathsf{Int}(1)) = \lfloor M_3 \rfloor \end{array}}{\begin{array}{c} E_2, M_2, \overline{M} \vDash_{\mathrm{s}'} \mathsf{res}(n) = \texttt{is\_initialized}(\mathsf{res}(n+1)); \Downarrow M_3, \overline{M} \\ E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![t]\!]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M} \end{array}}$$

$$\frac{E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![t]\!]_{\mathrm{T}}^{n+1}\ \mathsf{res}(n) = \texttt{is\_initialized}(\mathsf{res}(n+1)); \Downarrow M_3, \overline{M} \qquad \mathsf{alloc\_var}(E_0, M_0, \mathsf{res}(n+1), \texttt{int8}) = E_1, M_1 \qquad \mathsf{dealloc\_var}(E_3, M_3, \mathsf{res}(n+1)) = \lfloor M_4 \rfloor}{E_0, M_0, \overline{M} \vDash_{\mathrm{s}'} \texttt{let} \ldots \texttt{end} \Downarrow M_4, \overline{M}}$$

Définition des environnements et états mémoire :

$$\begin{array}{rcl|rcl} b_1, M_1 & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\mathrm{i}\,8)) & E_1 & \triangleq & \mathsf{add}(\mathsf{res}(n+1), b_1, E_0) \\ M_2 & \triangleq & \ldots & E_2 & \triangleq & E_1 \\ \lfloor M_3 \rfloor & \triangleq & \mathsf{store}(\mathrm{i}\,8, M_2, b_0, 0, \mathsf{Int}(1)) & E_3 & \triangleq & E_2 \\ \lfloor M_4 \rfloor & \triangleq & \mathsf{free}(M_3, b_1) & E_4 & \triangleq & \mathsf{remove}(b_1, E_3) = E_0 \end{array}$$

On applique le théorème 3 à $\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} t \Downarrow \mathsf{Ptr}(\widehat{b}, \widehat{\delta})$, et on obtient les propositions suivantes :

$$\begin{cases} E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![t]\!]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M} \\ E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta})) \\ M_2 \equiv M_1' \end{cases}$$

avec $\lfloor M_1' \rfloor = \mathsf{store}(\mathsf{mtype}(\tau*), M, b_1, 0, \dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta})))$. On identifie $\mathsf{Ptr}(b, \delta)$ et $\dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta}))$, ce qui définit $b \triangleq \dot{\sigma}(\widehat{b})$ et $\delta \triangleq \widehat{\delta}$.

Pour justifier la définition de M_3 par écriture, il suffit de montrer que $M_2 \vDash \mathrm{i}\,8\,@\,b_0, 0$ (axiome 2.17). D'après les axiomes 2.1 et 2.7, on a $M_0 \vDash \mathrm{i}\,8\,@\,b_0, 0$. La validité est préservée

lors d'une allocation (propriété 2.1), donc on a encore $M_1 \vDash \mathrm{i}8\,@\,b_0, 0$. D'après la propriété 2.3, on a encore $M_1' \vDash \mathrm{i}8\,@\,b_0, 0$. Par conséquent, par équivalence, $M_2 \vDash \mathrm{i}8\,@\,b_0, 0$.

La définition de M_3 est valide si $M_3 \vDash b_1$. Le bloc est initialement alloué dans M_0, donc d'après l'axiome 2.2 il est valide dans M_1. Les écritures n'affectent pas la valdité (axiome 2.6) donc b_1 est encore valid dans M_1, et donc dans M_2 par équivalence. Une nouvelle application de l'axiome 2.6, permet d'obtenir $M_3 \vDash b_1$.

Concernant le statut d'initialisation du terme évalué, on a dans l'évaluation source $\mathsf{load}(\mathsf{mtype}(\tau*), \widehat{M}, \widehat{b}, \widehat{\delta}) = \lfloor \widehat{v} \rfloor$ avec $\widehat{v} \neq \mathsf{Undef}$. $\dot{\sigma}$ conserve le caractère défini des valeurs, donc $\dot{\sigma}(\widehat{v}) \neq \mathsf{Undef}$. Si l'on définit $v = \dot{\sigma}(\widehat{v})$, on a donc, d'après l'isomorphisme $\widehat{M} \sim_\sigma M^p$, la lecture suivante dans M^p : $\mathsf{load}(\mathsf{mtype}(\tau*), M^p, b, \delta) = \lfloor v \rfloor$, avec $v \neq \mathsf{Undef}$. Puisque $M^p \triangleright \overline{M}$, on en déduit $\mathsf{is\_initialized}(\mathsf{mtype}(\tau), \overline{M}, b, \delta) = \mathsf{true}$.

Il reste à montrer que $M_4 \equiv M_0'$. Pour cela, on applique le même raisonnement que dans le cas P-VALID.

Cas P-UNINITIALIZED.

$$\text{P-UNINITIALIZED}\ \frac{\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} t : \tau \Downarrow \mathsf{Ptr}(\widehat{b}, \widehat{\delta}) \qquad \mathsf{load}(\mathsf{mtype}(\tau), \widehat{M}, \widehat{b}, \widehat{\delta}) = \lfloor \mathsf{Undef} \rfloor}{\widehat{E}, \widehat{M} \vDash_{\mathrm{p}} \backslash\texttt{initialized}(a) \Downarrow \bot}$$

Évaluation cible :

$$\text{S'-IS-UNINITIALIZED}\ \frac{\begin{array}{c} E_1, M_1 \vDash_{\mathrm{lv}} \mathsf{res}(n) \Downarrow b_0, 0 \qquad E_1, M_1 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \mathsf{Ptr}(b, \delta) \\ \mathsf{is\_initialized}(\mathsf{mtype}(\tau), \overline{M}, b, \delta) = \mathsf{false} \\ \mathsf{store}(\mathrm{i}8, M_2, b_0, 0, \mathsf{Int}(0)) = \lfloor M_3 \rfloor \\ E_2, M_2, \overline{M} \vDash_{\mathrm{s}'} \mathsf{res}(n) = \texttt{is\_initialized}(\mathsf{res}(n+1)); \Downarrow M_3, \overline{M} \\ E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![t]\!]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M} \end{array}}{\begin{array}{c} E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![t]\!]_{\mathrm{T}}^{n+1}\ \mathsf{res}(n) = \texttt{is\_initialized}(\mathsf{res}(n+1)); \Downarrow M_3, \overline{M} \\ \mathsf{alloc\_var}(E_0, M_0, \mathsf{res}(n+1), \texttt{int8}) = E_1, M_1 \qquad \mathsf{dealloc\_var}(E_3, M_3, \mathsf{res}(n+1)) = \lfloor M_4 \rfloor \\ E_0, M_0, \overline{M} \vDash_{\mathrm{s}'} \texttt{let} \ldots \texttt{end} \Downarrow M_4, \overline{M} \end{array}}$$

Définition des environnements et états mémoire :

$$\begin{array}{rcl|rcl} b_1, M_1 & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\mathrm{i}8)) & E_1 & \triangleq & \mathsf{add}(\mathsf{res}(n+1), b_1, E_0) \\ M_2 & \triangleq & \ldots & E_2 & \triangleq & E_1 \\ \lfloor M_3 \rfloor & \triangleq & \mathsf{store}(\mathrm{i}64, M_2, b_0, 0, \mathsf{Int}(1)) & E_3 & \triangleq & E_2 \\ \lfloor M_4 \rfloor & \triangleq & \mathsf{free}(M_3, b_1) & E_4 & \triangleq & \mathsf{remove}(b_1, E_3) = E_0 \end{array}$$

On applique le théorème 3 à $\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} t \Downarrow \mathsf{Ptr}(\widehat{b}, \widehat{\delta})$, et on obtient les propositions suivantes :

$$\begin{cases} E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![t]\!]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M} \\ E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta})) \\ M_2 \equiv M_1' \end{cases}$$

avec $\lfloor M_1' \rfloor = \mathsf{store}(\mathsf{mtype}(\tau*), M, b_1, 0, \dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta})))$. On identifie $\mathsf{Ptr}(b, \delta)$ et $\dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta}))$, ce qui définit $b \triangleq \dot{\sigma}(\widehat{b})$ et $\delta \triangleq \widehat{\delta}$.

Les opérations mémoires sont les mêmes que dans le cas P-INITIALIZED (à la valeur de l'écriture près), et sont donc justifiées de la même manière.

L'évaluation source résulte en une valeur indéfinie : $\mathsf{load}(\mathsf{mtype}(\tau*), \widehat{M}, \widehat{b}, \widehat{\delta}) = \lfloor \mathsf{Undef} \rfloor$. Par conséquent, en vertu de l'isomorphisme $\widehat{M} \sim_\sigma M^p$, la lecture correspondante dans M^p donne également une valeur indéfinie : $\mathsf{load}(\mathsf{mtype}(\tau*), M^p, b, \delta) = \lfloor \mathsf{Undef} \rfloor$. Puisque $M^p \triangleright \overline{M}$, on en déduit $\mathsf{is\_initialized}(\mathsf{mtype}(\tau), \overline{M}, b, \delta) = \mathsf{false}$.

Cas P-CMP. Évaluation source :

$$\text{P-CMP}\quad \frac{\widehat{E},\widehat{M} \Subset_{\mathrm{t}} t_1 \Downarrow \widehat{v_1} \qquad \widehat{E},\widehat{M} \Subset_{\mathrm{t}} t_2 \Downarrow \widehat{v_2} \qquad \mathsf{sem\_cmp}(\bowtie, \widehat{v_1}, \widehat{v_2}) = \beta}{\widehat{E},\widehat{M} \Subset_{\mathrm{p}} t_1 \ \bowtie \ t_2 \Downarrow \beta}$$

Traduction :

```
1  let res(n+1) in
2  let res(n+2) in
3    [[t]]_T^{n+1}
4    [[t]]_T^{n+2}
5    res(n) = res(n+1) [[ ⋈ ]]_op res(n+2);
6  end
7  end
```

Évaluation cible :

$$\dfrac{\dfrac{\dots}{E_2, M_2, \overline{M} \vDash_{s'} [\![t_1]\!]_{\mathrm{T}}^{n+1} \Downarrow M_3, \overline{M}} \quad \dfrac{\dfrac{\dots}{E_3, M_3, \overline{M} \vDash_{s'} [\![t_2]\!]_{\mathrm{T}}^{n+2} \Downarrow M_4, \overline{M}} \quad \dfrac{\begin{array}{c} E_4, M_4 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow v_1 \\ E_4, M_4 \vDash_{\mathrm{e}} \mathsf{res}(n+2) \Downarrow v_2 \\ \mathsf{sem\_binop}([\![\bowtie]\!]_{op}, v_1, v_2) = v \end{array}}{E_4, M_4 \vDash_{\mathrm{e}} \mathsf{res}(n+1) [\![\bowtie]\!]_{op} \mathsf{res}(n+2) \Downarrow v} \quad \dfrac{\begin{array}{c} E_4, M_4 \vDash_{\mathrm{lv}} \mathsf{res}(n) \Downarrow b_0, 0 \\ \mathsf{store}(\mathsf{mtype}(\tau), M_4, b_0, 0, v) = \lfloor M_5 \rfloor \end{array}}{E_4, M_4, \overline{M} \vDash_{s'} \mathsf{res}(n) = \dots; \Downarrow M_5, \overline{M}}}{E_3, M_3, \overline{M} \vDash_{s'} [\![t_2]\!]_{\mathrm{T}}^{n+2} \ \mathsf{res}(n) = \dots; \Downarrow M_5, \overline{M}}}{E_2, M_2, \overline{M} \vDash_{s'} [\![t_1]\!]_{\mathrm{T}}^{n+1} \ [\![t_2]\!]_{\mathrm{T}}^{n+2} \ \mathsf{res}(n) = \dots; \Downarrow M_5, \overline{M}} \quad \begin{array}{c} \mathsf{alloc\_var}(E_1, M_1, \mathsf{res}(n+2), \tau_2) = E_2, M_2 \qquad \mathsf{dealloc\_var}(E_5, M_5, \mathsf{res}(n+2)) = \lfloor M_6 \rfloor \end{array}}{E_1, M_1, \overline{M} \vDash_{s'} \texttt{let} \dots \texttt{end} \Downarrow M_6, \overline{M}} \quad \begin{array}{c} \mathsf{alloc\_var}(E_0, M_0, \mathsf{res}(n+1), \tau_1) = E_1, M_1 \qquad \mathsf{dealloc\_var}(E_6, M_6, \mathsf{res}(n+1)) = \lfloor M_7 \rfloor \end{array}}{E_0, M_0, \overline{M} \vDash_{s'} \texttt{let} \dots \texttt{end} \Downarrow M_7, \overline{M}}$$

Définition des environnements et états mémoire :

$$\begin{array}{rcl|rcl} b_1, M_1 & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\tau_1)) & E_1 & \triangleq & \mathsf{add}(\mathsf{res}(n+1), b_1, E_0) \\ b_2, M_2 & \triangleq & \mathsf{alloc}(M_1, \mathsf{sizeof}(\tau_2)) & E_2 & \triangleq & \mathsf{add}(\mathsf{res}(n+2), b_2, E_1) \\ M_3 & \triangleq & \dots & E_3 & \triangleq & E_2 \\ M_4 & \triangleq & \dots & E_4 & \triangleq & E_3 \\ \lfloor M_5 \rfloor & \triangleq & \mathsf{store}(\mathsf{mtype}(\tau), M_4, b_0, 0, v) & E_5 & \triangleq & E_4 \\ \lfloor M_6 \rfloor & \triangleq & \mathsf{free}(M_5, b_2) & E_6 & \triangleq & \mathsf{remove}(E_5, \mathsf{res}(n+2)) \\ \lfloor M_7 \rfloor & \triangleq & \mathsf{free}(M_7, b_1) & E_7 & \triangleq & \mathsf{remove}(E_6, \mathsf{res}(n+1)) \end{array}$$

Pour chacun des deux termes t_1 et t_2, on applique le théorème 3, ce qui résulte en les propositions suivantes :

$$\begin{cases} E_2, M_2, \overline{M} \vDash_{s'} [\![t_1]\!]_{\mathrm{T}}^{n+1} \Downarrow M_3, \overline{M} & \\ E_3, M_3 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow v_1 & \text{où} \quad v_1 \triangleq \dot{\sigma}(\widehat{v_1}) \\ M_3 \equiv M_2' & \text{où} \quad \lfloor M_2' \rfloor \triangleq \mathsf{store}(\mathsf{mtype}(\tau_1), M_2, b_1, 0, v_1) \\ & \\ E_3, M_3, \overline{M} \vDash_{s'} [\![t_2]\!]_{\mathrm{T}}^{n+2} \Downarrow M_4, \overline{M} & \\ E_4, M_4 \vDash_{\mathrm{e}} \mathsf{res}(n+2) \Downarrow v_2 & \text{où} \quad v_2 \triangleq \dot{\sigma}(\widehat{v_2}) \\ M_4 \equiv M_3' & \text{où} \quad \lfloor M_3' \rfloor \triangleq \mathsf{store}(\mathsf{mtype}(\tau_2), M_3, b_2, 0, v_2) \end{cases}$$

La définition de M_5 par écriture est valide si $M_4 \vDash \mathsf{mtype}(\tau) @ b_0, 0$. L'accès est valide dans M_0 par hypothèse ($M_0 \vDash \mathsf{mtype}(\tau) @ b_0, 0$), et M_1 est le résultat d'une allocation, donc

d'aprés la propriété 2.1, l'accès est encore valide dans M_1, puis dans M_2 par le même argument : $M_2 \vDash \mathsf{mtype}(\tau)\,@\,b_0, 0$. On en déduit par la propriété 2.3 la validité de l'accès dans M'_2, ce qui donne la validité dans M_3 par équivalence. On réitère ce raisonnement pour obtenir $M_4 \vDash \mathsf{mtype}(\tau)\,@\,b_0, 0$.

Pour justifier la définition de M_6 par déallocation, on établit $M_5 \vDash b_2$ afin de pouvoir appliquer l'axiome 2.5. $M_2 \vDash b_2$ d'après l'axiome 2.2, donc $M'_2 \vDash b_2$ (d'après l'axiome 2.6), d'où $M_3 \vDash b_2$ par équivalence. De même $M'_3 \vDash b_2$, donc $M_4 \vDash b_2$. On obtient $M_5 \vDash b_2$ par une nouvelle application de l'axiome 2.6.

Par un raisonnement analogue, on montre que $M_6 \vDash b_1$, ce qui justifie la définition de l'état mémoire M_7.

Il reste donc à montrer que $M_7 \equiv M'_0$, M'_0 étant défini par $\lfloor M_{0'} \rfloor = \mathsf{store}(\mathsf{i}\,8, M_0, b_0, 0, \mathsf{Int}(\beta))$. Commençons par établir $v = \mathsf{Int}(\beta)$. Les membres de l'égalité sont définis par :

$$v = \mathsf{sem\_binop}(\llbracket \bowtie \rrbracket_{op}, v_1, v_2)$$
$$\beta = \mathsf{sem\_cmp}(\bowtie, \widehat{v_1}, \widehat{v_2}).$$

L'égalité est établie en substituant les définitions $v_1 \triangleq \dot{\sigma}(\widehat{v_1})$ et $v_2 \triangleq \dot{\sigma}(\widehat{v_2})$, puis en appliquant l'hypothèse 1. Dès lors, on peut utiliser l'identité $\lfloor M_{0'} \rfloor = \mathsf{store}(\mathsf{i}\,8, M_0, b_0, 0, v)$ pour raisonner par équivalence et montrer que $M_7 \equiv M'_0$ dans les paragraphes suivants.

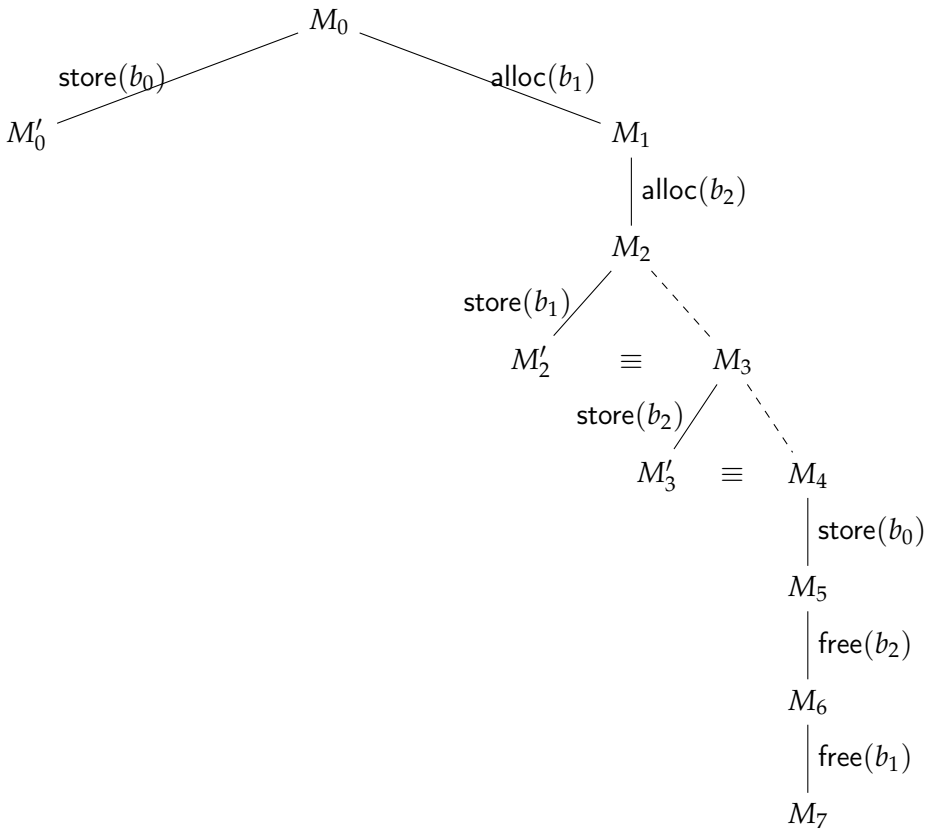

FIGURE 4.7 – Relations entre les états mémoire de l'évaluation cible dans le cas P-CMP. Les opérations sur la mémoire sont abrégées pour ne mentionner que le type d'opération (allocation, déallocation, écriture) et le bloc concerné.

La figure figure 4.7 illustre les relations entre les différents états mémoires impliqués. Si le raisonnement ne diffère pas conceptuellement de celui mis en œuvre dans les cas précédents, il nécessite dans le cas présent un nombre assez important d'étapes intermédiaires et d'états fictifs. La méthode employée reste cependant la même ; il s'agit toujours d'alterner entre trois étapes de raisonnement :

— soit *permuter* des opérations mémoire, par application des propriétés 4.5 et 4.6
— soit *reporter* des opérations à partir d'un état sur un état équivalent, grâce aux propriétés 4.9 et 4.10 ;
— soit enfin *simplifier* une suite de la forme allocation, écriture, déallocation (pour un même bloc), à l'aide de la propriété 4.8.

On commence par définir l'état fictif M_7' en permutant les opérations mémoires à partir de M_4, de manière à effectuer d'abord les deux déallocations puis l'écriture dans b. On obtient ainsi un état fictif équivalent à M_7. On reporte ensuite les trois opérations sur l'état M_3', ce qui définit un nouvel état fictif équivalent au précédent, donc à M_7. On reporte alors sur M_2' l'écriture dans b_2 (définition de M_3') et les opérations subséquentes ; l'état final de cette nouvelle séquence est encore équivalent à M_7. On permute ensuite l'écriture dans b_1 (définition de M_2') et l'allocation de b_2 (définition de M_2). Cette dernière permutation fait apparaître une séquence simplifiable concernant b_2. La simplification fait à son tour apparaître une nouvelle séquence simplifiable, concernant cette fois-ci b_1. Après simplification, la séquence considérée ne comporte plus que l'écriture dans b à partir de M_0, identique à celle définissant M_0'. L'état résultant est donc, comme tous les états finals définis dans le raisonnement, équivalent à M_7. Il est par ailleurs équivalent à M_0', d'où la conclusion.

□

4.4.3 Traduction des termes

De manière analogue à la preuve précédente, le théorème est démontré par induction sur la structure de l'évaluation source. On retrouve dans chaque cas un raisonnement de la même forme que dans la preuve précédente :

— identification de la structure de l'évaluation cible ;
— définition des environnements et états mémoires intervenant dans cette évaluation ;
— démonstration des conditions de validité pour les opérations d'écriture ou déallocation ;
— démonstration de l'équivalence entre l'état final de l'évaluation cible et l'état fictif défini par l'écriture du résultat dans la variable de résultat.

On adopte la même convention de notation que dans la preuve précédente pour ce qui est des états mémoire (et environnements). On définit de plus pour chaque cas le contexte $(E_0, M_0) = \mathsf{alloc\_var}(E, M, \mathsf{res}(n), \tau)$, où τ est le type du terme source évalué, et $b_0 \triangleq E_0(\mathsf{res}(n))$. On a alors $M_0 \vDash \mathsf{mtype}(\tau) \,@\, b_0, 0$.

Cas T-BASEADDR. Évaluation source :

$$\text{T-BASEADDR}\ \frac{\widehat{E}, \widehat{M} \vDash_{\mathsf{t}} t : \tau* \Downarrow \mathsf{Ptr}(\widehat{b}, \widehat{\delta})}{\widehat{E}, \widehat{M} \vDash_{\mathsf{t}} \backslash\texttt{base\_address}(t) \Downarrow \mathsf{Ptr}(\widehat{b}, 0)}$$

Traduction :

```
1  let res(n+1) : τ* in
2    [[t]]_T^{n+1}
3    res(n) = base_address(res(n+1));
4  end
```

Évaluation cible :

$$\frac{\begin{array}{c} E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \mathsf{Ptr}(b,\delta) \\ E_2, M_2 \vDash_{\mathrm{lv}} \mathsf{res}(n) \Downarrow b_0, 0 \qquad \mathsf{store}(\mathrm{i}64, M_2, b_0, 0, \mathsf{Ptr}(b,0)) = \lfloor M_3 \rfloor \end{array}}{E_2, M_2, \overline{M} \vDash_{\mathrm{s}'} \mathsf{res}(n) = \texttt{base\_address}(\mathsf{res}(n+1)); \Downarrow M_3, \overline{M}}$$

$$\frac{E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![t]\!]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M}}{E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![t]\!]_{\mathrm{T}}^{n+1}\ \mathsf{res}(n) = \texttt{base\_address}(\mathsf{res}(n+1)); \Downarrow M_3, \overline{M}}$$

$$\frac{\mathsf{alloc\_var}(E_0, M_0, \mathsf{res}(n+1), \tau*) = E_1, M_1 \qquad \mathsf{dealloc\_var}(E_3, M_3, \mathsf{res}(n+1)) = \lfloor M_4 \rfloor}{E_0, M_0, \overline{M} \vDash_{\mathrm{s}'} \texttt{let} \ldots \texttt{end} \Downarrow M_4, \overline{M}}$$

Définition des environnements et états mémoire :

$$\begin{array}{rcl|rcl} b_1, M_1 & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\tau*)) & E_1 & \triangleq & \mathsf{add}(\mathsf{res}(n+1), b_1, E_0) \\ M_2 & \triangleq & \ldots & E_2 & \triangleq & E_1 \\ \lfloor M_3 \rfloor & \triangleq & \mathsf{store}(\mathsf{mtype}(\tau*), M_2, b_0, 0, \mathsf{Ptr}(b,\delta)) & E_3 & \triangleq & E_2 \\ \lfloor M_4 \rfloor & \triangleq & \mathsf{free}(M_3, b_1) & E_4 & \triangleq & \mathsf{remove}(E_3, \mathsf{res}(n+1)) \end{array}$$

M_2 est défini par hypothèse d'induction sur l'évaluation de t, et vérifie les propriétés suivantes :

$$\begin{cases} E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![t]\!]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M} & \\ E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow v & \text{où} \quad v \triangleq \dot{\sigma}(\mathsf{Ptr}(b,\delta)) \\ M_3 \equiv M_2' & \text{où} \quad \lfloor M_2' \rfloor \triangleq \mathsf{store}(\mathsf{mtype}(\tau*), M_2, b_1, 0, v) \end{cases}$$

Montrons que M_3 est bien définie par écriture. Il suffit d'après l'axiome 2.17 de montrer que $M_2 \vDash \mathsf{mtype}(\tau*) \,@\, b_0, 0$. L'accès est valide dans M_0 par hypothèse ; il est encore valide dans M_1 d'après la propriété 2.1, puis dans M_1' d'après la propriété 2.3, donc dans M_2 également par équivalence.

La déallocation de b_1 est quant à elle bien définie si $M_3 \vDash b_1$. Or $M_1 \vDash b_1$ d'après l'axiome 2.2. Par conséquent $M_1' \vDash b_1$, d'après l'axiome 2.6, donc $M_2 \vDash b_1$ par équivalence. On applique à nouveau l'axiome 2.6 pour établir $M_3 \vDash b_1$.

Montrons maintenant que $M_4 \equiv M_0'$. On introduit les états fictifs M_4', défini en permutant écriture dans b_0 et déallocation de b_1, et et M_3'' et M_4'', définis en reportant ces mêmes opérations sur M_1' (voir la figure 4.8). D'après la propriété 4.6, on a $M_4' \equiv M_4$, et d'après les propriétés 4.9 et 4.10, $M_4'' \equiv M_4'$. En outre, d'après la propriété 4.8, $M_0 \equiv M_3''$. On en déduit $M_0' \equiv M_4''$ par application de la propriété 4.10.

Cas T-OFS. Évaluation source :

$$\text{T-OFS}\ \frac{\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} t \Downarrow \mathsf{Ptr}(\widehat{b}, \widehat{\delta})}{\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} \backslash\texttt{offset}(t) \Downarrow \mathsf{Int}(\widehat{\delta})}$$

Traduction :

```
1  let res(n+1) : τ* in
2    [[t]]_T^{n+1}
3    res(n) = offset(res(n+1));
4  end
```

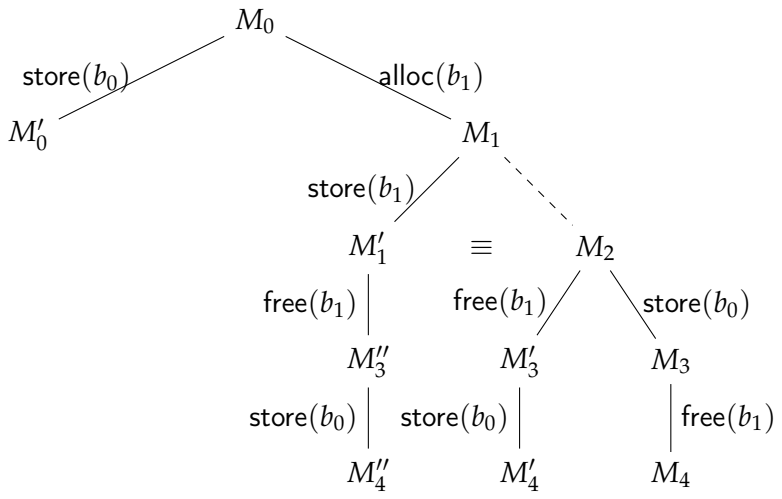

FIGURE 4.8 – Relations entre les états mémoire de l'évaluation cible dans le cas T-BASEADDR. Les opérations sur la mémoire sont abrégées pour ne mentionner que le type d'opération (allocation, déallocation, écriture) et le bloc concerné.

Évaluation cible :

$$\frac{\dfrac{\dfrac{E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \mathsf{Ptr}(b,\delta) \quad E_2, M_2 \vDash_{\mathrm{lv}} \mathsf{res}(n) \Downarrow b_0, 0 \quad \mathsf{store}(\mathrm{i}\,64, M_2, b_0, \mathsf{Int}(\delta)) = \lfloor M_3 \rfloor}{E_2, M_2, \overline{M} \vDash_{s'} \mathsf{res}(n) = \mathtt{offset}(\mathsf{res}(n+1)); \Downarrow M_3, \overline{M}} \quad E_1, M_1, \overline{M} \vDash_{s'} [\![t]\!]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M}}{E_1, M_1, \overline{M} \vDash_{s'} [\![t]\!]_{\mathrm{T}}^{n+1}\ \mathsf{res}(n) = \mathtt{offset}(\mathsf{res}(n+1)); \Downarrow M_3, \overline{M}} \quad \mathsf{alloc\_var}(E_0, M_0, \mathsf{res}(n+1), \tau*) = E_1, M_1 \quad \mathsf{dealloc\_var}(E_3, M_3, \mathsf{res}(n+1)) = \lfloor M_4 \rfloor}{E_0, M_0, \overline{M} \vDash_{s'} \mathtt{let} \ldots \mathtt{end} \Downarrow M_4, \overline{M}}$$

Définition des environnements et états mémoire :

$$\begin{array}{rcl|rcl} b_1, M_1 & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\tau*)) & E_1 & \triangleq & \mathsf{add}(\mathsf{res}(n+1), b_1, E_0) \\ M_2 & \triangleq & \ldots & E_2 & \triangleq & E_1 \\ \lfloor M_3 \rfloor & \triangleq & \mathsf{store}(\mathrm{i}\,64, M_2, b_0, 0, \mathsf{Int}(\delta)) & E_3 & \triangleq & E_2 \\ \lfloor M_4 \rfloor & \triangleq & \mathsf{free}(M_3, b_1) & E_4 & \triangleq & \mathsf{remove}(E_3, \mathsf{res}(n+1)) \end{array}$$

M_2 est défini par hypothèse d'induction sur l'évaluation de t, et vérifie les propriétés suivantes :

$$\begin{cases} E_1, M_1, \overline{M} \vDash_{s'} [\![t]\!]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M} & \\ E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow v & \text{où} \quad v \triangleq \dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta})) = \mathsf{Ptr}(\sigma(\widehat{b}), \widehat{\delta}) \\ M_3 \equiv M'_2 & \text{où} \quad \lfloor M'_2 \rfloor \triangleq \mathsf{store}(\mathsf{mtype}(\tau*), M_2, b_1, 0, v) \end{cases}$$

La deuxième propriété permet d'identifier $(b, \delta) = (\sigma(\widehat{b}), \widehat{\delta})$.

Les conditions d'accès valide $M_2 \vDash \mathrm{i}\,64\,@\,b_0, 0$ et $M_3 \vDash b_1$, qui garantissant la bonne définition de M_3 et de M_4, sont établies de par le même raisonnement qu'au cas précédent.

De même, la démonstration de l'équivalence $M_4 \equiv M'_0$ est la même qu'au cas précédent, la structure des opérations mémoire étant identique (seuls changent la valeur et le type de l'opération d'écriture).

Cas T-BLOCKLEN. Évaluation source :

$$\text{T-BLOCKLEN}\ \frac{\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} t \Downarrow \mathsf{Ptr}(\widehat{b}, \widehat{\delta}) \qquad \mathsf{length}(M, \widehat{b}) = \widehat{k}}{\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} \backslash\texttt{block\_length}(t) \Downarrow \mathsf{Int}(\widehat{k})}$$

Traduction :

```
let res(n+1) : τ* in
  [[t]]_T^{n+1}
  res(n) = block_length(res(n+1));
end
```

Évaluation cible :

$$\frac{\begin{array}{c} E_1, M_1 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \mathsf{Ptr}(b, \delta) \\ \mathsf{length}(\overline{M}, b) = \lfloor k \rfloor \qquad E_1, M_1 \vDash_{\mathrm{lv}} \mathsf{res}(n) \Downarrow b_0, 0 \qquad \mathsf{store}(\mathrm{i}\,64, M_2, b_0, 0, \mathsf{Int}(k)) = \lfloor M_3 \rfloor \\ \hline E_2, M_2, \overline{M} \vDash_{\mathrm{s}'} \mathsf{res}(n) = \texttt{block\_length}(\mathsf{res}(n+1)); \Downarrow M_3, \overline{M} \end{array} \qquad E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![t]\!]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M}}{\begin{array}{c} E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![t]\!]_{\mathrm{T}}^{n+1}\ \mathsf{res}(n) = \texttt{block\_length}(\mathsf{res}(n+1)); \Downarrow M_3, \overline{M} \\ \mathsf{alloc\_var}(E_0, M_0, \mathsf{res}(n+1), \tau*) = E_1, M_1 \qquad \mathsf{dealloc\_var}(E_3, M_3, \mathsf{res}(n+1)) = \lfloor M_4 \rfloor \\ \hline E_0, M_0, \overline{M} \vDash_{\mathrm{s}'} \texttt{let} \ldots \texttt{end} \Downarrow M_4, \overline{M} \end{array}}$$

Définition des environnements et états mémoire :

$$\begin{array}{rcl|rcl} b_1, M_1 & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\tau*)) & E_1 & \triangleq & \mathsf{add}(\mathsf{res}(n+1), b_1, E_0) \\ M_2 & \triangleq & \ldots & E_2 & \triangleq & E_1 \\ \lfloor M_3 \rfloor & \triangleq & \mathsf{store}(\mathrm{i}\,64, M_2, b_0, 0, \mathsf{Int}(k)) & E_3 & \triangleq & E_2 \\ \lfloor M_4 \rfloor & \triangleq & \mathsf{free}(M_3, b_1) & E_4 & \triangleq & \mathsf{remove}(E_3, \mathsf{res}(n+1)) \end{array}$$

M_2 est défini par hypothèse d'induction sur l'évaluation de t, et vérifie les propriétés suivantes :

$$\begin{cases} E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [\![t]\!]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M} & \\ E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow v & \text{où} \quad v \triangleq \dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta})) = \mathsf{Ptr}(\sigma(\widehat{b}), \widehat{\delta}) \\ M_3 \equiv M_2' & \text{où} \quad \lfloor M_2' \rfloor \triangleq \mathsf{store}(\mathsf{mtype}(\tau*), M_2, b_1, 0, v) \end{cases}$$

Les conditions d'accès valide $M_2 \vDash \mathrm{i}\,64\,@\,b_0, 0$ et $M_3 \vDash b_1$, qui garantissant la bonne définition de M_3 et de M_4, sont établies de par le même raisonnement que dans le cas T-BASEADDR.

De même, la démonstration de l'équivalence $M_4 \equiv M_0'$ a la même structure que dans le cas T-BASEADDR, les opérations mémoire étant similaires. Ici, l'utilisation de la propriété 4.10 repose sur l'égalité $k = \widehat{k}$, garantie par les hypothèses $M^p \triangleright \overline{M}$ et $M^p \hookrightarrow M$.

Cas T-EXPR. Évaluation source :

$$\text{T-EXPR}\ \frac{\widehat{E}, \widehat{M} \vDash_{\mathrm{e}} e \Downarrow \widehat{v}}{\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} e \Downarrow \widehat{v}}$$

Traduction :

```
res(n) = e;
```

Évaluation cible :

$$\frac{E_0, M_0 \vDash_{\mathrm{e}} e \Downarrow v \qquad E_0, M_0 \vDash_{\mathrm{lv}} \mathsf{res}(n) \Downarrow b_0, 0 \qquad \mathsf{store}(\mathsf{mtype}(\tau), M_0, b_0, 0, v) = \lfloor M_1 \rfloor}{E_0, M_0, \overline{M} \vDash_{\mathrm{s}'} \mathsf{res}(n) = e; \Downarrow M_1, \overline{M}}$$

L'écriture réussit car $M_0 \vDash \mathsf{mtype}(\tau)\,@\,b_0, 0$ par définition de M_0.

On définit $v \triangleq \dot{\sigma}(\widehat{v})$. Considérons la prémisse de l'évaluation source : $\widehat{E}, \widehat{M} \vDash_{\mathrm{e}} e \Downarrow \widehat{v}$. Par hypothèse, on a $(\widehat{E}, \widehat{M}) \sim (E^p, M^p)$, donc, d'après le lemme 1, $E^p, \widehat{M^p} \vDash_{\mathrm{e}} e \Downarrow \dot{\sigma}(\widehat{v})$. Par ailleurs $E^p, M^p \hookrightarrow E_0, M_0$, donc d'après le lemme 2, on a $E_0, M_0 \vDash_{\mathrm{e}} e \Downarrow \dot{\sigma}(\widehat{v})$, soit $E_0, M_0 \vDash_{\mathrm{e}} e \Downarrow v$.

Il reste à montrer que $M_1 \equiv M_0'$, où $\lfloor M_0' \rfloor \triangleq \mathsf{store}(\mathsf{mtype}(\tau), M_0, b_0, 0, \dot{\sigma}(\widehat{v}))$. On conclut en observant que la définition de M_0' coïncide avec celle de M_1.

Cas T-DEREF. Évaluation source :

$$\text{T-DEREF}\ \frac{\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} t : \tau* \Downarrow \mathsf{Ptr}(\widehat{b}, \widehat{\delta}) \qquad \mathsf{load}(\mathsf{mtype}(\tau), \widehat{M}, \widehat{b}, \widehat{\delta}) = \lfloor \widehat{v} \rfloor \qquad \widehat{v} \neq \mathsf{Undef}}{\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} *t \Downarrow \widehat{v}}$$

Traduction :

```
1  let res(n+1) : τ* in
2    [[t]]_T^{n+1}
3    res(n) = *res(n+1);
4  end
```

Évaluation cible :

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \mathsf{Ptr}(b, \delta)}{E_2, M_2 \vDash_{\mathrm{lv}} *\mathsf{res}(n+1) \Downarrow b, \delta} \quad \mathsf{load}(\mathsf{mtype}(\tau), M_1, b, \delta) = \lfloor \dot{\sigma}(\widehat{v}) \rfloor \qquad \dot{\sigma}(\widehat{v}) \neq \mathsf{Undef}}{E_2, M_2 \vDash_{\mathrm{e}} *\mathsf{res}(n+1) \Downarrow \dot{\sigma}(\widehat{v})} \quad E_2, M_2 \vDash_{\mathrm{lv}} \mathsf{res}(n) \Downarrow b_0, 0 \qquad \mathsf{store}(\mathsf{mtype}(\tau), M_2, b_0, 0, \dot{\sigma}(\widehat{v})) = \lfloor M_3 \rfloor}{E_2, M_2, \overline{M} \vDash_{\mathrm{s}'} \mathsf{res}(n) = *\mathsf{res}(n+1); \Downarrow M_3, \overline{M}} \quad E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [[t]]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M}}{E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [[t]]_{\mathrm{T}}^{n+1}\ \mathsf{res}(n) = *\mathsf{res}(n+1); \Downarrow M_3, \overline{M}} \quad \mathsf{alloc\_var}(E_0, M_0, \mathsf{res}(n+1), \tau) = E_1, M_1 \qquad \mathsf{dealloc\_var}(E_3, M_3, \mathsf{res}(n+1)) = \lfloor M_4 \rfloor}{E_0, M_0, \overline{M} \vDash_{\mathrm{s}'} \texttt{let} \ldots \texttt{end} \Downarrow M_4, \overline{M}}$$

Définition des environnements et états mémoire :

$$\begin{array}{rcl|rcl} b_1, M_1 & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\mathsf{mtype}(\tau*))) & E_1 & \triangleq & \mathsf{add}(\mathsf{res}(n+1), b_1, E_0) \\ M_2 & \triangleq & \ldots & E_2 & \triangleq & E_1 \\ \lfloor M_3 \rfloor & \triangleq & \mathsf{store}(\mathsf{mtype}(\tau), M_2, b_0, 0, \dot{\sigma}(\widehat{v})) & E_3 & \triangleq & E_2 \\ \lfloor M_4 \rfloor & \triangleq & \mathsf{free}(M_3, b_1) & E_4 & \triangleq & E_0 \end{array}$$

M_2 est défini par hypothèse d'induction sur l'évaluation de t, et vérifie les propriétés suivantes :

$$\begin{cases} E_1, M_1, \overline{M} \vDash_{\mathrm{s}'} [[t]]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M} \\ E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta})) \\ M_2 \equiv M_1' \quad \text{où} \quad \lfloor M_1' \rfloor \triangleq \mathsf{store}(\mathsf{mtype}(\tau*), M_1, b_1, 0, \dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta}))) \end{cases}$$

Par identification entre $\mathsf{Ptr}(b, \delta)$ et $\dot{\sigma}(\mathsf{Ptr}(\widehat{b}, \widehat{\delta}))$, on définit $b \triangleq \sigma(\widehat{b})$ et $\delta \triangleq \widehat{\delta}$.

Il suffit, pour justifier la définition de M_3 par écriture, de montrer que $M_2 \vDash \mathsf{mtype}(\tau) \,@\, b_0, 0$. Par définition de M_0, on a $M_0 \vDash \mathsf{mtype}(\tau) \,@\, b_0, 0$. La validité de l'accès est préservée par allocation, d'après la propriété 2.1, donc on a encore $M_1 \vDash \mathsf{mtype}(\tau) \,@\, b_0, 0$. L'accès est toujours valide dans M_1' d'après la propriété 2.3, donc $M_1' \vDash \mathsf{mtype}(\tau) \,@\, b_0, 0$, ce qui permet de déduire $M_2 \vDash \mathsf{mtype}(\tau) \,@\, b_0, 0$ par équivalence.

Montrons maintenant que l'opération de lecture $\mathsf{load}(\mathsf{mtype}(\tau), M_1, b, \delta)$ figurant dans l'évaluation cible renvoie effectivement la valeur $\dot{\sigma}(\widehat{v})$.

D'après l'évaluation source, $\mathsf{load}(\mathsf{mtype}(\tau), \widehat{M}, \widehat{b}, \widehat{\delta}) = \lfloor \widehat{v} \rfloor$. L'hypothèse d'isomorphisme entre $\widehat{M}$ et M^p donne :

$$\mathsf{load}(\mathsf{mtype}(\tau), M^p, \sigma(\widehat{b}), \widehat{\delta}) = \lfloor \dot{\sigma}(\widehat{v}) \rfloor, \tag{4.1}$$

soit, en substituant les définitions de b et δ :

$$\mathsf{load}(\mathsf{mtype}(\tau), M^p, b, \delta) = \lfloor \dot{\sigma}(\widehat{v}) \rfloor. \tag{4.2}$$

Or $E^p, M^p \hookrightarrow E, M$ par hypothèse, donc $E^p, M^p \hookrightarrow E_0, M_0$ d'après la propriété 4.17. Par définition d'extension, on a donc :

$$\mathsf{load}(\mathsf{mtype}(\tau), M^p, b, \delta) = \mathsf{load}(\mathsf{mtype}(\tau), M_0, b, \delta). \tag{4.3}$$

De plus, d'après l'axiome 2.12, le résultat de cette écriture ne change pas après allocation :

$$\mathsf{load}(\mathsf{mtype}(\tau), M_0, b, \delta) = \mathsf{load}(\mathsf{mtype}(\tau), M_1, b, \delta). \tag{4.4}$$

En combinant les équations (4.1) à (4.4), on obtient finalement $\mathsf{load}(\mathsf{mtype}(\tau), M_1, b, \delta) = \lfloor \dot{\sigma}(\widehat{v}) \rfloor$.

Il reste à montrer l'équivalence entre $M_4 \equiv M_0'$, où $\lfloor M_0' \rfloor \triangleq \mathsf{store}(\mathsf{mtype}(\tau), M_0, b_0, 0, \dot{\sigma}(\widehat{v}))$. Pour cela, on procède de la même façon que dan le cas T-BASEADDR.

Cas T-UNOP.

$$\frac{\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} t \Downarrow \widehat{v_1} \qquad \mathsf{sem\_unop}(\,\bar{\dagger}\,, \widehat{v_1}) = \lfloor \widehat{v_2} \rfloor}{\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} \,\bar{\dagger}\, t \Downarrow \widehat{v_2}}\ \text{T-UNOP}$$

Traduction :

```
1  let res(n+1) : τ in
2    [[t]]_T^{n+1}
3    res(n) = †res(n+1);
4  end
```

Évaluation cible :

$$\frac{\dfrac{E_1, M_1, \overline{M} \vDash_{s'} [\![t]\!]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M} \qquad \dfrac{\dfrac{E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \dot{\sigma}(\widehat{v_1}) \quad \mathsf{sem\_unop}(\,\dagger\,, \dot{\sigma}(\widehat{v_1})) = \dot{\sigma}(\widehat{v_2})}{E_2, M_2 \vDash_{\mathrm{e}} \,\dagger\, \mathsf{res}(n+1) \Downarrow \dot{\sigma}(\widehat{v_2})} \quad E_2, M_2 \vDash_{\mathrm{lv}} \mathsf{res}(n) \Downarrow b_0, 0 \quad \mathsf{store}(\mathsf{mtype}(\tau), M_2, b_0, 0, \dot{\sigma}(\widehat{v_2})) = \lfloor M_3 \rfloor}{E_2, M_2, \overline{M} \vDash_{s'} \mathsf{res}(n) = \,\dagger\, \mathsf{res}(n+1); \Downarrow M_3, \overline{M}}}{E_1, M_1, \overline{M} \vDash_{s'} [\![t]\!]_{\mathrm{T}}^{n+1} \;\mathsf{res}(n) = \,\dagger\, \mathsf{res}(n+1); \Downarrow M_3, \overline{M}} \quad \mathsf{alloc\_var}(E_0, M_0, \mathsf{res}(n+1), \tau) = E_1, M_1 \qquad \mathsf{dealloc\_var}(E_3, M_3, \mathsf{res}(n+1)) = \lfloor M_4 \rfloor}{E_0, M_0, \overline{M} \vDash_{s'} \texttt{let...end} \Downarrow M_4, \overline{M}}$$

Définition des environnements et états mémoire :

$$\begin{array}{rcl|rcl} b_1, M_1 & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\tau)) & E_1 & \triangleq & \mathsf{add}(\mathsf{res}(n+1), b_1, E_0) \\ M_2 & \triangleq & \ldots & E_2 & \triangleq & E_1 \\ \lfloor M_3 \rfloor & \triangleq & \mathsf{store}(\mathsf{mtype}(\tau), M_2, b_0, 0, \dot{\sigma}(\widehat{v_2})) & E_3 & \triangleq & E_2 \\ \lfloor M_4 \rfloor & \triangleq & \mathsf{free}(M_3, b_1) & E_4 & \triangleq & \mathsf{remove}(E_3, \mathsf{res}(n+1)) \end{array}$$

M_2 est défini par hypothèse d'induction sur l'évaluation de t, et vérifie les propriétés suivantes :

$$\begin{cases} E_1, M_1, \overline{M} \vDash_{s'} [\![t]\!]_{\mathrm{T}}^{n+1} \Downarrow M_2, \overline{M} \\ E_2, M_2 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow \dot{\sigma}(\widehat{v_1}) \\ M_2 \equiv M_1' \quad \text{où} \quad \lfloor M_1' \rfloor \triangleq \mathsf{store}(\mathsf{mtype}(\tau), M_2, b_1, 0, \dot{\sigma}(\widehat{v_1})). \end{cases}$$

La justification de la définition de M_3 par écriture est démontrée de la même façon que dans le cas précédent. La démonstration de l'équivalence $M_4 \equiv M_0'$ repose principalement sur l'hypothèse 1 (sémantique des opérateurs) pour établir l'identité $\mathsf{sem\_unop}(\dagger, \dot{\sigma}(\widehat{v_1})) = \dot{\sigma}(\widehat{v_2})$, utilisée dans l'évaluation cible, à partir de la prémisse de l'évaluation source $\mathsf{sem\_unop}(\bar{\dagger}, \widehat{v_1}) = \widehat{v_2}$. Le reste du raisonnement est similaire au traitement de l'équivalence dans le cas T-BASEADDR, par exemple.

Cas T-BINOP. Évaluation source :

$$\text{T-BINOP}\ \frac{\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} t_1 \Downarrow \widehat{v_1} \qquad \widehat{E}, \widehat{M} \vDash_{\mathrm{t}} t_2 \Downarrow \widehat{v_2} \qquad \mathsf{sem\_binop}(\bar{\ddagger}, \widehat{v_1}, \widehat{v_2}) = \lfloor \widehat{v_3} \rfloor}{\widehat{E}, \widehat{M} \vDash_{\mathrm{t}} t_1 \ddagger t_2 \Downarrow \widehat{v_3}}$$

Traduction :

```
1  let res(n+1) in
2  let res(n+2) in
3    [[t]]_T^{n+1}
4    [[t]]_T^{n+2}
5    res(n) = res(n+1) ‡ res(n+2);
6  end
7  end
```

Évaluation cible :

$$\dfrac{\dfrac{\dfrac{\ldots}{E_2, M_2, \overline{M} \vDash_{s'} [\![t_1]\!]_{\mathrm{T}}^{n+1} \Downarrow M_3, \overline{M}} \quad \dfrac{\dfrac{\ldots}{E_3, M_3, \overline{M} \vDash_{s'} [\![t_2]\!]_{\mathrm{T}}^{n+2} \Downarrow M_4, \overline{M}} \quad \dfrac{\dfrac{E_4, M_4 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \Downarrow v_1 \quad E_4, M_4 \vDash_{\mathrm{e}} \mathsf{res}(n+2) \Downarrow v_2 \quad \mathsf{sem\_binop}(\ddagger, v_1, v_2) = v}{E_4, M_4 \vDash_{\mathrm{e}} \mathsf{res}(n+1) \ddagger \mathsf{res}(n+2) \Downarrow v} \quad E_4, M_4 \vDash_{\mathrm{lv}} \mathsf{res}(n) \Downarrow b_0, 0 \quad \mathsf{store}(\mathsf{mtype}(\tau), M_4, b_0, 0, v) = \lfloor M_5 \rfloor}{E_4, M_4, \overline{M} \vDash_{s'} \mathsf{res}(n) = \ldots; \Downarrow M_5, \overline{M}}}{E_3, M_3, \overline{M} \vDash_{s'} [\![t_2]\!]_{\mathrm{T}}^{n+2}\ \mathsf{res}(n) = \ldots; \Downarrow M_5, \overline{M}}}{E_2, M_2, \overline{M} \vDash_{s'} [\![t_1]\!]_{\mathrm{T}}^{n+1}\ [\![t_2]\!]_{\mathrm{T}}^{n+2}\ \mathsf{res}(n) = \ldots; \Downarrow M_5, \overline{M}} \quad \mathsf{alloc\_var}(E_1, M_1, \mathsf{res}(n+2), \tau_2) = E_2, M_2 \quad \mathsf{dealloc\_var}(E_5, M_5, \mathsf{res}(n+2)) = \lfloor M_6 \rfloor}{E_1, M_1, \overline{M} \vDash_{s'} \mathtt{let} \ldots \mathtt{end} \Downarrow M_6, \overline{M}} \quad \mathsf{alloc\_var}(E_0, M_0, \mathsf{res}(n+1), \tau_1) = E_1, M_1 \quad \mathsf{dealloc\_var}(E_6, M_6, \mathsf{res}(n+1)) = \lfloor M_7 \rfloor}{E_0, M_0, \overline{M} \vDash_{s'} \mathtt{let} \ldots \mathtt{end} \Downarrow M_7, \overline{M}}$$

Définition des environnements et états mémoire :

$$
\begin{array}{rcl|rcl}
b_1, M_1 & \triangleq & \mathsf{alloc}(M_0, \mathsf{sizeof}(\tau_1)) & E_1 & \triangleq & \mathsf{add}(\mathsf{res}(n+1), b_1, E_0) \\
b_2, M_2 & \triangleq & \mathsf{alloc}(M_1, \mathsf{sizeof}(\tau_2)) & E_2 & \triangleq & \mathsf{add}(\mathsf{res}(n+2), b_2, E_1) \\
M_3 & \triangleq & \ldots & E_3 & \triangleq & E_2 \\
M_4 & \triangleq & \ldots & E_4 & \triangleq & E_3 \\
\lfloor M_5 \rfloor & \triangleq & \mathsf{store}(\mathsf{mtype}(\tau), M_4, b_0, 0, v) & E_5 & \triangleq & E_4 \\
\lfloor M_6 \rfloor & \triangleq & \mathsf{free}(M_5, b_2) & E_6 & \triangleq & \mathsf{remove}(E_5, \mathsf{res}(n+2)) \\
\lfloor M_7 \rfloor & \triangleq & \mathsf{free}(M_7, b_1) & E_7 & \triangleq & \mathsf{remove}(E_6, \mathsf{res}(n+1))
\end{array}
$$

M_3 (respectivement M_4) est défini par hypothèse d'induction appliquée à l'évaluation du terme t_1 (respectivement t_2). Le cas est ensuite traité de la même manière que le cas P-CMP du théorème 2.

□

Chapitre 5

Optimisation par analyse statique

Ce chapitre est consacré à l'étude d'une optimisation de l'instrumentation par analyse de flot de données [Kil73], technique mise en œuvre dans l'outil E-ACSL afin de réduire le surcoût en performance induit par l'instrumentation. L'analyse vise à déterminer un sous-ensemble d'emplacements mémoire minimal dont l'instrumentation permet au moniteur d'évaluer correctement les assertions du programme.

Nous définissons une telle analyse, et prouvons qu'elle est sûre, au sens où limiter l'instrumentation aux seuls emplacements désignés par l'analyse ne compromet pas la validité des verdicts du moniteur.

5.1 Présentation du problème

Le problème étudié ici est motivé par un souci de performance. La transformation de programme définie dans le chapitre 3 introduit des instructions additionnelles, qui ont un coût en temps d'exécution, et fait usage de la mémoire d'observation, ce qui a un coût en espace mémoire. Ce double surcoût, qui peut être important en pratique [VSK17], peut cependant être amoindri.

Considérons par exemple la fonction `search` du programme C présenté dans la figure 1.1. Son code source comporte une unique assertion, qui porte sur le prédicat de validité de pointeur $\backslash\texttt{valid}(\texttt{t} + \texttt{mid})$, et devient, après instrumentation intégrale par E-ACSL, le code de la figure 5.1. L'évaluation du prédicat par le moniteur ne nécessite *a priori* pas de disposer d'informations sur les variables `lo` et `hi` (ni sur l'état en mémoire de `mid`, seule la valeur de cette variable étant requise). Il serait donc possible de s'abstenir de générer le code d'instrumentation de ces variables, et d'améliorer ainsi les performances du programme instrumenté.

On peut envisager de généraliser cet exemple en introduisant l'idée d'une instrumentation sélective, où seuls les accès mémoires nécessaires à l'évaluation sont instrumentés. La figure 5.2 schématise l'architecture fonctionnelle d'une telle optimisation : une analyse statique est introduite en amont de la génération de moniteur, et détermine les accès mémoire qui doivent être instrumentés. L'ensemble des accès mémoire d'un programme n'étant pas calculable statiquement — du moins pas de manière exacte —, on a recours à une technique d'approximation : l'analyse de flot de données [NNH99]. Celle-ci permet, au moyen d'un calcul de point fixe, de calculer un sur-ensemble de l'ensemble recherché.

Se pose alors la question de la sûreté de l'optimisation réalisée : nous avons en effet déterminé au chapitre 4 que la sémantique des programmes source est préservée par une instrumentation intégrale, mais qu'en est-il dans le cas d'une instrumentation partielle ?

Suivant la description fonctionnelle de la figure 5.2, une approche naturelle pour étudier ce problème pourrait être la suivante :

```
1   int search(int *t, int length, int x)
2   {
3     int __retres;
4   • __e_acsl_store_block((void *)(& __retres),4UL);
5   • __e_acsl_store_block((void *)(& x),4UL);
6   • __e_acsl_store_block((void *)(& length),4UL);
7     __e_acsl_store_block((void *)(& t),8UL);
8     int lo = 0;
9   • __e_acsl_store_block((void *)(& lo),4UL);
10  • __e_acsl_full_init((void *)(& lo));
11    int hi = length - 1;
12  • __e_acsl_store_block((void *)(& hi),4UL);
13  • __e_acsl_full_init((void *)(& hi));
14    while (lo <= hi) {
15      int mid = lo + (hi - lo) / 2;
16  •   __e_acsl_store_block((void *)(& mid),4UL);
17  •   __e_acsl_full_init((void *)(& mid));
18      /*@ assert \valid(t + mid); */
19      {
20        int __gen_e_acsl_valid;
21        __gen_e_acsl_valid = __e_acsl_valid((void *)(t + mid),sizeof(int),
22                                            (void *)t,(void *)(& t));
23        __e_acsl_assert(__gen_e_acsl_valid,(char *)"Assertion",
24                        (char *)"search",(char *)"\\valid(t + mid)",5);
25      }
26      if (*(t + mid) == x) {
27  •     __e_acsl_full_init((void *)(& __retres));
28        __retres = mid;
29  •     __e_acsl_delete_block((void *)(& mid));
30        goto return_label;
31      }
32      else
33        if (*(t + mid) < x) {
34  •       __e_acsl_full_init((void *)(& lo));
35          lo = mid + 1;
36        }
37        else {
38  •       __e_acsl_full_init((void *)(& hi));
39          hi = mid - 1;
40        }
41  •   __e_acsl_delete_block((void *)(& mid));
42    }
43  • __e_acsl_full_init((void *)(& __retres));
44    __retres = -1;
45    return_label:
46    {
47  •   __e_acsl_delete_block((void *)(& x));
48  •   __e_acsl_delete_block((void *)(& length));
49      __e_acsl_delete_block((void *)(& t));
50  •   __e_acsl_delete_block((void *)(& hi));
51  •   __e_acsl_delete_block((void *)(& lo));
52  •   __e_acsl_delete_block((void *)(& __retres));
53      return __retres;
54    }
55  }
```

FIGURE 5.1 – Instrumentation intégrale de la fonction `search` de la figure 1.1 avec E-ACSL. Le code peut être optimisé en omettant les lignes marquées du symbole •. Les lignes faisant intervenir la variable `__retres` sont générées pour les besoins d'une fonctionnalité de Frama-C sur laquelle E-ACSL repose, et que nous ne traitons pas ici.

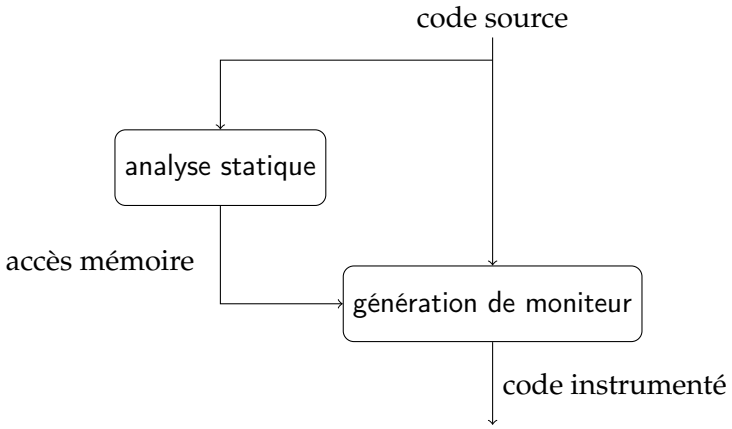

FIGURE 5.2 – Utilisation de l'analyse pour le paramétrage de la transformation. Seuls les accès mémoire calculés par l'analyse sont instrumentés durant la génération de moniteur.

1. ajouter à la transformation de programme définie au chapitre 3 un paramètre conditionnant la génération des instructions d'instrumentation ;
2. définir sur le langage source du chapitre 2 une analyse statique ayant pour rôle de paramétrer la transformation de programme modifiée ;
3. montrer l'équivalence sémantique, pour un programme source donné, entre le programme optimisé (c'est-à-dire issu de la transformation paramétrée par les résultats de l'analyse) et le programme non-optimisé (issu de la transformation non modifiée).

Si cette approche directe offre l'avantage d'expliciter l'articulation entre analyse statique et génération de moniteur, elle est toutefois relativement complexe à mettre en œuvre. Nous avons donc fait le choix de traiter le problème différemment.

Partant du principe que le rôle de l'analyse est de déterminer les accès mémoire qui devront être suivis en mémoire d'observation, nous intégrons cette dernière dans la sémantique du langage source [1]. Cette intégration, présentée dans la section 5.2, est réalisée de manière à simuler les instructions de suivi de la mémoire de la transformation définie en section 3.4.1. Notre langage d'étude pour ce chapitre peut ainsi être considéré comme une variante du langage défini au chapitre 2, dans laquelle les opérations sur la mémoire d'exécution sont doublées par leur équivalent dans la mémoire d'observation.

Nous définissons ensuite, dans la section 5.3, une analyse sur les programmes de ce langage, puis montrons dans la section 5.4 que l'évaluation des programmes est indifférente à toute altération de la mémoire d'observation en dehors de la région calculée par l'analyse.

5.2 Langage d'étude

Dans la description qui suit, nous appellerons *langage source* le langage défini au chapitre 2, tandis que le langage d'étude du chapitre sera appelé *langage d'analyse*.

Le langage d'analyse est une variante du langage source du chapitre 2, qui en diffère principalement par le style de sa sémantique, ainsi que par l'intégration dans sa sémantique du suivi des opérations mémoire, via la mémoire d'observation. Ces deux différences majeures étant d'ordre sémantique, la syntaxe, présentée dans la figure 5.3, est assez similaire à celle du langage source.

1. on parle couramment dans ce cas de *sémantique instrumentée* [Ben04], mais il s'agit d'une notion distincte de celle d'instrumentation de programme, qui est notre principal objet d'étude.

Valeurs gauches	$l ::=$	x	variable
	$\mid$	$*a$	déréférencement d'un pointeur
Zones mémoire	$a ::=$	l	valeur gauche
	$\mid$	$a +\!+\, e$	décalage de pointeur
	$\mid$	$\&l$	adresse
Expressions	$e ::=$	n	constante entière
	$\mid$	a	zone mémoire
	$\mid$	$e \ddagger e$	expressions arithmétiques
	$\mid$	$e == e \mid e <= e \mid !\, e$	expressions booléennes à valeur entière
Instructions	$s ::=$	$[\texttt{skip;}]^\ell$	sans effet
	$\mid$	$[l = e;]^\ell$	affectation
	$\mid$	$[l = \texttt{malloc}(e);]^\ell$	allocation
	$\mid$	$[\texttt{free}(a);]^\ell$	déallocation
	$\mid$	$[\texttt{logical\_assert}(p);]^\ell$	assertion
	$\mid$	$\texttt{if}\ ([e]^\ell)\ \texttt{then}\ s\ \texttt{else}\ s$	conditionnelle
	$\mid$	$\texttt{while}\ ([e]^\ell)\ s$	boucle
	$\mid$	$s\ s$	séquence
Prédicats	$p ::=$	$t = t \mid t \leq t$	comparateurs
	$\mid$	$p \wedge p \mid \neg p$	connecteurs logiques
	$\mid$	$\backslash\texttt{valid}(t)$	validité d'un pointeur t
	$\mid$	$\backslash\texttt{initialized}(t)$	statut d'initialisation de la valeur pointée par t
Termes	$t ::=$	e	expression
	$\mid$	$\backslash\texttt{base\_address}(t)$	adresse de base du bloc mémoire de t
	$\mid$	$\backslash\texttt{block\_length}(t)$	taille du bloc mémoire de t
	$\mid$	$\backslash\texttt{offset}(t)$	décalage de t dans son bloc mémoire
	$\mid$	$\ldots$	combinaison d'expressions et de termes ayant trait à la mémoire

FIGURE 5.3 – Syntaxe du langage d'étude.

La différence la plus notable réside dans la définition de deux catégories syntaxiques à part entière pour les données résidant en mémoire : les zones mémoire et les valeurs gauches. Il s'agit essentiellement d'un artefact technique pour distinguer syntaxiquement les expressions qui peuvent être associées à un emplacement mémoire, soit en tant que valeur gauche (voir section 2.3), soit en tant que pointeur (par prise d'adresse d'une valeur gauche, ou décalage à partir d'un pointeur existant). De la même façon, le décalage d'un pointeur (noté $+\!+$ pour le distinguer de l'addition) est considéré comme un opérateur distinct, plutôt que comme un cas d'opérateur binaire au même titre que les opérateurs arithmétiques.

Les instructions sont les mêmes que celles du langage source, à l'exception de l'introduction d'une variable locale (instruction `let`), que nous ne traitons pas ici : les variables de nos programmes sont globales. L'affectation et l'allocation diffèrent par ailleurs légèrement de leur présentation du chapitre 2, en ce que leur membre gauche n'est plus une expression mais une valeur gauche.

Enfin, pour les besoins de l'analyse, certains éléments syntaxiques sont *étiquetées*. Les étiquettes sont des identifiants abstraits qui permettent d'exprimer des relations entre éléments d'un programme en s'affranchissant de sa structure syntaxique. Dans notre cas, elles seront utilisées dans la section suivante pour définir le *graphe de flot de contrôle* du programme. On notera $[s]^\ell$ une instruction s étiquetée par ℓ, et on supposera les instructions étiquetées de manière unique dans un programme donné. Les instructions élémentaires sont étiquetées, ainsi que les conditions des conditionnelles et des boucles.

Le langage d'analyse possède une sémantique opérationnelle hybride : si les expressions, termes et prédicats sont évalués dans le style de la sémantique opérationnelle à grands pas, comme dans le langage source, les instructions en revanche sont munies d'une sémantique à petits pas.

À la différence des jugements d'évaluation du langage source, qui possèdent un contexte formé d'un environnement et d'un état mémoire, ceux du langage d'analyse ont pour seul contexte un état mémoire. Il s'agit là d'une conséquence de la suppression des variables locales : ici, les variables sont définies dans un environnement global E, que l'on considère implicite dans le reste de ce chapitre. Cet environnement associe à chaque variable présente dans le programme un bloc mémoire d'une taille correspondant au type de la variable en question. Quant aux états mémoire, ils sont définis de la même façon que dans les chapitres précédents, qu'il s'agisse de la mémoire d'exécution (section 2.2) ou de la mémoire d'observation (section 3.2).

Concernant les valeurs gauches, expressions, termes et prédicats, on retrouve donc des formes de jugements d'évaluation semblables à celles du langage source :

— $M \vDash_{\text{lv}} lv \Downarrow (b, \delta)$, évaluation d'une valeur gauche lv en un emplacement (b, δ) dans la mémoire M ;
— $M \vDash_{\text{e}} e \Downarrow v$, évaluation d'une expression e en une valeur v dans la mémoire M ;
— $M \vDash_{\text{t}} t \Downarrow v$, évaluation d'un terme t en une valeur v dans la mémoire M ;
— $M \vDash_{\text{p}} p \Downarrow \beta$, évaluation d'un prédicat p en une valeur de vérité β dans la mémoire M.

$$\text{LV-VAR}\ \frac{E(x) = b}{M \vDash_{\text{lv}} x \Downarrow (b, 0)} \qquad \text{LV-DEREF}\ \frac{M \vDash_{\text{e}} a \Downarrow \mathsf{Ptr}(b, \delta)}{M \vDash_{\text{lv}} *a \Downarrow (b, \delta)}$$

$$\text{E-LVAL}\ \frac{v \neq \texttt{Undef} \qquad M \vDash_{\text{lv}} l : \tau \Downarrow (b, \delta) \qquad \mathsf{load}(\mathsf{mtype}(\tau), M, b, \delta) = \lfloor v \rfloor}{M \vDash_{\text{e}} l \Downarrow v} \qquad \text{E-ADDR}\ \frac{M \vDash_{\text{lv}} l \Downarrow (b, \delta)}{M \vDash_{\text{e}} \&l \Downarrow \mathsf{Ptr}(b, \delta)}$$

$$\text{E-SHIFT}\ \frac{M \vDash_{\text{e}} a : \tau \Downarrow \mathsf{Ptr}(b, \delta) \qquad M \vDash_{\text{e}} e \Downarrow \mathsf{Int}(n)}{M \vDash_{\text{e}} a +\!+ e \Downarrow \mathsf{Ptr}(b, \delta + n \cdot \mathsf{sizeof}(\tau))}$$

$$\text{E-INT}\ \frac{}{M \vDash_{\text{e}} n \Downarrow \mathsf{Int}(n)} \qquad \text{E-OP}\ \frac{M \vDash_{\text{e}} e_1 \Downarrow \mathsf{Int}(n_1) \qquad M \vDash_{\text{e}} e_2 \Downarrow \mathsf{Int}(n_2)}{M \vDash_{\text{e}} e_1 \ddagger e_2 \Downarrow \mathsf{Int}(n_1 \ \ddagger \ n_2)}$$

$$\text{E-NEG-FALSE}\ \frac{M \vDash_{\text{e}} e \Downarrow \mathsf{Int}(n) \qquad n \neq 0}{M \vDash_{\text{e}} !e \Downarrow \mathsf{Int}(0)} \qquad \text{E-NEG-TRUE}\ \frac{M \vDash_{\text{e}} e \Downarrow \mathsf{Int}(0)}{M \vDash_{\text{e}} !e \Downarrow \mathsf{Int}(1)}$$

FIGURE 5.4 – Évaluation des expressions et valeurs gauches.

L'évaluation des expressions et valeurs gauches (figure 5.4) présente une forme quelque peu

différente de celle qu'elle prend dans le langage source (figure 2.12), du fait des changements dans la syntaxe des expressions, mais lui est tout à fait semblable en principe.

La sémantique des termes et des prédicats est, quant à elle, rigoureusement identique à son équivalent dans le langage source (figures 2.13 à 2.15), à la forme des contextes d'évaluation près (les contextes d'évaluation du langage source incluent un environnement, alors qu'il est omis dans le langage d'analyse).

Concernant les instructions, la sémantique se présente sous la forme de deux types de jugements d'évaluation, correspondant à deux types de pas d'évaluation :

— $\langle s, M_1, \overline{M_1}\rangle \to \langle M_2, \overline{M_2}\rangle$ représente l'évaluation de l'instruction s dans les états mémoire (respectivement d'exécution et d'observation) M_1 et $\overline{M_1}$, qui se conclut dans les états M_2 et $\overline{M_2}$;

— $\langle s_1, M_1, \overline{M_1}\rangle \to \langle s_2, M_2, \overline{M_2}\rangle$ représente l'évaluation de l'instruction s_1 dans les états mémoire M_1 et $\overline{M_1}$, qui se poursuit avec l'évaluation de s_2 dans M_2 et $\overline{M_2}$.

Les instructions élémentaires (figure 5.5) sont toutes définies par un jugement d'évaluation du premier type, qui décrit donc uniquement les effets en mémoire de ces instructions. En cela, ils sont assez semblables de leurs équivalents dans le langage source, quand bien même ils sont exprimés avec un formalisme différent.

Leur particularité tient plutôt à la duplication des opérations mémoire. Par exemple, l'évaluation d'une affectation, définie par la règle S-ASSIGN, a non seulement pour conséquence l'écriture de la valeur affectée en mémoire d'exécution, mais aussi l'initialisation en mémoire d'observation de l'emplacement de l'écriture. Ce, afin de reproduire l'effet de l'instrumentation.

Similairement, l'instruction `malloc` (règle S-MALLOC) effectue l'allocation d'un bloc (exécution), et l'enregistrement de bloc correspondant (observation). Ces deux opérations sont suivies de l'écriture du pointeur (exécution), et de l'initialisation qui lui correspond (observation).

Suivant le même principe, la déallocation d'un bloc (règle S-FREE) est doublée de sa suppression dans la mémoire d'observation. Enfin, tout comme dans le langage source, une assertion dont le prédicat est évalué à $\top$ est sans effet (les mémoires d'exécution et d'observation sont inchangées), et le cas où ce même prédicat est évalué à $\bot$ est indéfini.

$$\text{S-ASSIGN}\ \frac{\begin{array}{ll} M_1 \vDash_{\mathrm{e}} e : \tau \Downarrow v & \mathsf{store}(\mathsf{mtype}(\tau), M_1, b, \delta, v) = \lfloor M_2 \rfloor \\ M_1 \vDash_{\mathrm{lv}} l \Downarrow (b, \delta) & \mathsf{initialize}(\mathsf{mtype}(\tau), \overline{M_1}, b, \delta) = \overline{M_2} \end{array}}{\langle l = e;,\ M_1,\ \overline{M_1}\rangle \to \langle M_2,\ \overline{M_2}\rangle}$$

$$\text{S-MALLOC}\ \frac{\begin{array}{ll} M_1 \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(n) & \mathsf{store\_block}(\overline{M_1}, b', n) = \overline{M_2} \\ M_1 \vDash_{\mathrm{lv}} l : \tau* \Downarrow (b, \delta) & \mathsf{initialize}(\mathsf{mtype}(\tau*), \overline{M_2}, b, \delta) = \overline{M_3} \\ \mathsf{alloc}(M_1, n) = (b', M_2) & \mathsf{store}(\mathsf{mtype}(\tau*), M_2, b, \delta, \mathsf{Ptr}(b', 0)) = \lfloor M_3 \rfloor \end{array}}{\langle l = \texttt{malloc}(e);,\ M_1, \overline{M_1}\rangle \to \langle M_3,\ \overline{M_3}\rangle}$$

$$\text{S-FREE}\ \frac{M_1 \vDash_{\mathrm{e}} a \Downarrow (b, 0) \qquad \mathsf{free}(M_1, b) = \lfloor M_2 \rfloor \qquad \mathsf{delete\_block}(\overline{M_1}, b) = \overline{M_2}}{\langle \texttt{free}(a);,\ M_1,\ \overline{M_1}\rangle \to \langle M_2,\ \overline{M_2}\rangle}$$

$$\text{S-ASSERT}\ \frac{M \vDash_{\mathrm{p}} p \Downarrow \top}{\langle \texttt{logical\_assert}(p);,\ M,\ \overline{M}\rangle \to \langle M,\ \overline{M}\rangle} \qquad \text{S-SKIP}\ \frac{}{\langle \texttt{skip},\ M,\ \overline{M}\rangle \to \langle M,\ \overline{M}\rangle}$$

FIGURE 5.5 – Évaluation des instructions élémentaires. Les étiquettes sont omises.

La deuxième forme de jugement d'évaluation des instructions décrit le flot de contrôle du programme. On la retrouve donc naturellement dans la définition des boucles, conditionnelles, et séquences (figure 5.6). Par exemple, la règle S-IF-TRUE indique, lorsque la condition d'une conditionnelle est évaluée à un entier non nul, que la suite de l'évaluation consiste à évaluer la première branche de la conditionnelle. Dans le cas où la condition est évaluée à zéro, c'est la seconde branche qui est ensuite évaluée (règle S-IF-FALSE).

Dans le cas d'une séquence de deux instructions, on distingue deux cas selon l'évaluation de la première instruction. Soit celle-ci se conclut (règle S-SEQ-END), et l'évaluation continue alors avec la seconde instruction, dans l'état mémoire auquel a abouti la première ; soit l'évaluation de la première instruction se poursuit avec une nouvelle instruction et un nouvel état mémoire, auquel cas la séquence se poursuit de même (règle S-SEQ-CONT).

L'évaluation d'une boucle se conclut, sans modification de la mémoire, si sa condition est évaluée à zéro (règle S-WHILE-END). Sinon, on évalue le corps de la boucle, suivi de la boucle elle-même (règle S-WHILE-CONT).

$$\text{S-SEQ-CONT}\quad \frac{\langle s_1, M_1, \overline{M_1}\rangle \to \langle s_1', M_2, \overline{M_2}\rangle}{\langle s_1\ s_2, M_1, \overline{M_1}\rangle \to \langle s_1'\ s_2, M_2, \overline{M_2}\rangle} \qquad \text{S-SEQ-END}\quad \frac{\langle s_1, M_1, \overline{M_1}\rangle \to \langle M_2, \overline{M_2}\rangle}{\langle s_1\ s_2, M_1, \overline{M_1}\rangle \to \langle s_2, M_2, \overline{M_2}\rangle}$$

$$\text{S-IF-TRUE}\quad \frac{M \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(n) \qquad n \neq 0}{\langle \texttt{if}\ (e)\ \texttt{then}\ s_t\ \texttt{else}\ s_f, M, \overline{M}\rangle \to \langle s_t, M, \overline{M}\rangle} \qquad \text{S-IF-FALSE}\quad \frac{M \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(0)}{\langle \texttt{if}\ (e)\ \texttt{then}\ s_t\ \texttt{else}\ s_f, M, \overline{M}\rangle \to \langle s_f, M, \overline{M}\rangle}$$

$$\text{S-WHILE-CONT}\quad \frac{M \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(n) \qquad n \neq 0}{\langle \texttt{while}\ (e)\ s, M, \overline{M}\rangle \to \langle s\ \texttt{while}\ (e)\ s, M, \overline{M}\rangle} \qquad \text{S-WHILE-END}\quad \frac{M \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(0)}{\langle \texttt{while}\ (e)\ s, M, \overline{M}\rangle \to \langle M, \overline{M}\rangle}$$

FIGURE 5.6 – Évaluation des instructions composées. Les étiquettes sont omises.

5.3 Définition de l'analyse

Passons maintenant à la conception de l'analyse de flot de données. Celle-ci a pour fonction de déterminer, pour chaque instruction modifiant la mémoire d'un programme, si l'emplacement concerné est susceptible d'intervenir dans l'évaluation d'une assertion, ultérieurement dans l'exécution. De manière plus abstraite, il s'agit donc de savoir si une certaine ressource est susceptible d'être utilisée en aval du point d'examen.

Or, il existe une famille d'analyses de flot de données dont le but est précisément de répondre à ce type d'interrogation : les analyses de vivacité (*liveness analysis*). Elles sont couramment employées dans les compilateurs [ASU86], pour l'allocation de registre notamment. Une analyse de vivacité se présente typiquement sous la forme d'une fonction associant à chaque instruction du programme l'ensemble des variables dites « vivantes » en ce point, c'est-à-dire potentiellement utilisées plus loin dans le programme.

Un obstacle nous empêche cependant de recourir directement à une analyse de vivacité classique, comme par exemple celle présentée par NIELSON, NIELSON et HANKIN [NNH99] : la présence de pointeurs dans notre langage, et en particulier la possibilité d'*alias* entre expres-

sions, c'est-à-dire la possibilité que des expressions (ou plus précisément, des valeurs gauches) différentes référencent le même emplacement mémoire.

Les analyses de flot de données sont, en règle générale, significativement plus complexes en présence de pointeurs, si bien que de nombreuses analyses ont été développées pour traiter spécifiquement ces derniers [Hin01]. Par conséquent, dans la définition de notre analyse, nous supposerons disposer des résultats d'une analyse de pointeurs, préalablement effectuée sur le programme analysé. Cette hypothèse, que nous faisons ici dans un cadre théorique, correspond en pratique au fonctionnement de l'outil E-ACSL, dont l'analyse de flot de données est précédée par une analyse d'alias.

5.3.1 Graphe de flot de contrôle

La première étape de la définition de l'analyse consiste à définir le graphe de flot de contrôle (ou CFG, pour *Control Flow Graph*) du programme. Comme son nom l'indique, il s'agit d'un graphe qui décrit le flot de contrôle du programme, c'est-à-dire l'ordre d'exécution des différentes instructions qui composent celui-ci.

La figure 5.7 donne une représentation visuelle du CFG d'un programme simple. Le programme possède une unique instruction initiale (ici, étiquetée 1), et une instruction finale (ici, étiquetée 2), ou plus. L'exécution du programme démarre par l'instruction initiale, se déroule en suivant les arêtes (orientées) du graphe, et se termine (éventuellement) par une des instructions finales.

Étant donné un programme étiqueté s, le CFG de ce programme est formellement représenté par trois objets : l'étiquette de l'instruction initiale $\mathcal{I}(s)$, l'ensemble $\mathcal{F}(s)$ des étiquettes d'instructions finales, et l'ensemble $\Phi(s)$ des arêtes du CFG, qui sont des couples (ordonnés) d'étiquettes, ce qui définit donc également les nœuds du CFG. Étant donné que les étiquettes identifient des instructions, on parlera parfois d'instruction par abus de langage pour désigner des objets qui sont techniquement des étiquettes.

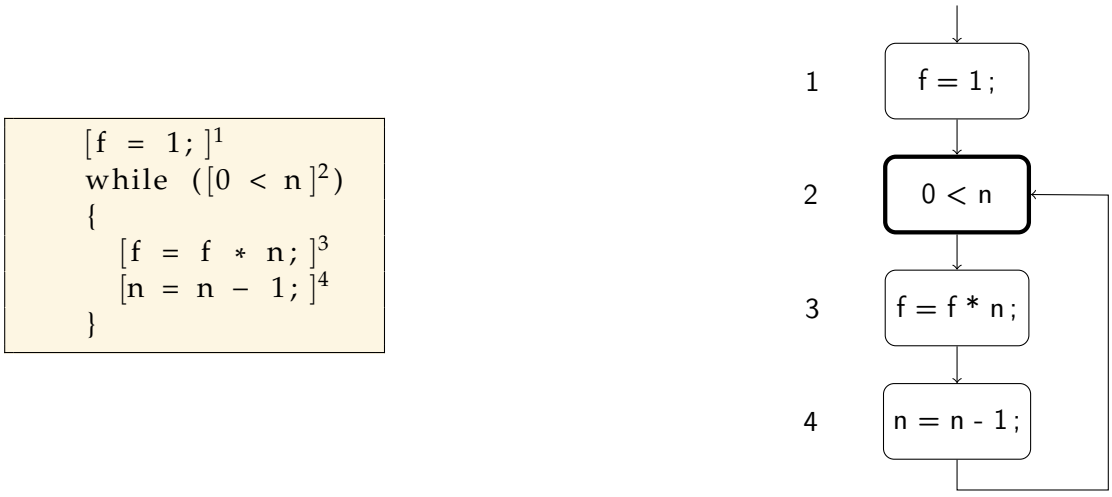

FIGURE 5.7 – Un programme simple et son CFG. La flèche incidente sur l'état 1 désigne l'état initial, tandis que la bordure épaisse de l'état 2 indique qu'il s'agit d'un état final.

Les fonctions $\mathcal{I}$, $\mathcal{F}$ et Φ sont définies inductivement suivant la syntaxe des instructions, respectivement dans les figures 5.8 à 5.10.

Pour toutes les instructions portant directement une étiquette, c'est-à-dire toutes sauf les séquences, l'étiquette en question est l'étiquette initiale. Dans une séquence d'instructions, l'étiquette initiale de la séquence est celle de la première instruction.

$$
\begin{array}{lcl}
\mathcal{I}([\texttt{skip;}]^\ell) & \triangleq & \ell \\
\mathcal{I}([lv = e;]^\ell) & \triangleq & \ell \\
\mathcal{I}([lv = \texttt{malloc}(e);]^\ell) & \triangleq & \ell \\
\mathcal{I}([\texttt{free}(e);]^\ell) & \triangleq & \ell \\
\mathcal{I}([\texttt{logical\_assert}(p);]^\ell) & \triangleq & \ell \\
\mathcal{I}(\texttt{if } ([e]^\ell) \texttt{ then } s_1 \texttt{ else } s_2) & \triangleq & \ell \\
\mathcal{I}(\texttt{while } ([e]^\ell)\ s) & \triangleq & \ell \\
\mathcal{I}(s_1\ s_2) & \triangleq & \mathcal{I}(s_1).
\end{array}
$$

FIGURE 5.8 – Instruction initiale d'un programme.

$$
\begin{array}{lcl}
\mathcal{F}([\texttt{skip;}]^\ell) & \triangleq & \{\ell\} \\
\mathcal{F}([lv = e;]^\ell) & \triangleq & \{\ell\} \\
\mathcal{F}([lv = \texttt{malloc}(e);]^\ell) & \triangleq & \{\ell\} \\
\mathcal{F}([\texttt{free}(e);]^\ell) & \triangleq & \{\ell\} \\
\mathcal{F}([\texttt{logical\_assert}(p);]^\ell) & \triangleq & \{\ell\} \\
\mathcal{F}(\texttt{if } ([e]^\ell) \texttt{ then } s_1 \texttt{ else } s_2) & \triangleq & \mathcal{F}(s_1) \cup \mathcal{F}(s_2) \\
\mathcal{F}(\texttt{while } ([e]^\ell)\ s) & \triangleq & \{\ell\} \\
\mathcal{F}(s_1\ s_2) & \triangleq & \mathcal{F}(s_2)
\end{array}
$$

FIGURE 5.9 – Instructions finales d'un programme.

Pour les instructions élémentaires, l'ensemble des instructions finales est le singleton qui contient leur étiquette. Pour une séquence, les instructions finales sont celles de la seconde composante de la séquence. Pour une conditionnelle, l'évaluation peut se terminer dans l'un ou l'autre branche, par conséquent l'ensemble des instructions finales est l'union des ensembles d'instructions finales des deux branches. Enfin, une boucle est, comme une instruction élémentaire, sa propre instruction finale.

$$
\begin{array}{lcl}
\Phi([\texttt{skip;}]^\ell) & \triangleq & \varnothing \\
\Phi([l = e;]^\ell) & \triangleq & \varnothing \\
\Phi([l = \texttt{malloc}(e);]^\ell) & \triangleq & \varnothing \\
\Phi([\texttt{free}(e);]^\ell) & \triangleq & \varnothing \\
\Phi([\texttt{logical\_assert}(p);]^\ell) & \triangleq & \varnothing \\
\Phi(s_1\ s_2) & \triangleq & \Phi(s_1) \cup \Phi(s_2) \cup \{\ (\ell, \mathcal{I}(s_2))\ |\ \ell \in \mathcal{F}(s_1)\ \} \\
\Phi(\texttt{if } ([e]^\ell) \texttt{ then } s_1 \texttt{ else } s_2) & \triangleq & \Phi(s_1) \cup \Phi(s_2) \cup \{\ (\ell, \mathcal{I}(s_1)),\ (\ell, \mathcal{I}(s_2))\ \} \\
\Phi(\texttt{while } ([e]^\ell)\ s) & \triangleq & \Phi(s) \cup \{\ (\ell', \ell)\ |\ \ell' \in \mathcal{F}(s)\ \} \cup \{\ (\ell,\ \mathcal{I}(s))\ \}
\end{array}
$$

FIGURE 5.10 – Flot d'un programme.

La fonction de flot Φ définit le graphe à proprement parler, puisque $\Phi(s)$ est l'ensemble des arêtes du CFG de s, chaque arête étant représentée par un couple formé d'une étiquette de départ et une étiquette d'arrivée. Les instructions élémentaires ne définissent pas d'arête, en raison précisément de leur caractère élémentaire. Les cas intéressants sont donc ceux des

boucles, conditionnelles, et séquences.

Pour une séquence, le graphe est formé par union des graphes des deux composantes de la séquence, et connexion de chaque instruction finale de la première composante à l'instruction initiale de la seconde.

Le graphe d'une conditionnelle est formé par union des graphes des deux branches, puis connexion de l'étiquette de la conditionnelle à l'étiquette initiale de chaque branche.

Enfin, dans le cas d'une boucle, chaque instruction finale du corps de la boucle est connectée à l'étiquette la boucle elle-même, tandis que celle-ci est connectée à l'étiquette initiale du corps de la boucle.

5.3.2 Traitement des zones mémoire : atteignabilité

Nous présentons dans cette section la notion d'ensemble de blocs *atteignable*, qui, bien qu'elle ne soit pas formellement requise pour la définition de l'analyse, est centrale dans sa conception. Il s'agit en effet, avec l'usage d'une analyse d'alias, du principal moyen mis en œuvre pour traiter le problème des pointeurs dans notre analyse.

On l'a dit, les zones mémoires sont des objets relativement complexes à traiter. On opère donc une simplification pour n'avoir à traiter que des variables, en introduisant la notion de variable de base d'une zone mémoire, définie ci-dessous, qui extrait la base (syntaxique) d'une zone mémoire. Concrètement, il s'agit de la première variable qui apparaît dans l'expression d'une zone mémoire lorsqu'elle est lue de gauche à droite :

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{base}(x) & \triangleq & x & \mathsf{base}(*a) & = & \mathsf{base}(a) \\ \mathsf{base}(\&x) & \triangleq & x & \mathsf{base}(\&(*a)) & = & \mathsf{base}(a) \\ & & & \mathsf{base}(a \mathbin{+\!+} e) & \triangleq & \mathsf{base}(a). \end{array}$$

La notion de variable de base repose sur l'idée suivante : à l'exécution, une zone mémoire complexe telle que, par exemple, $*(\&\mathtt{x} \mathbin{+\!+} *(\mathtt{t} \mathbin{+\!+} 3))$, désignera toujours un emplacement mémoire atteignable depuis le bloc mémoire associé à la première variable (ici x), par déréférencements et décalages successifs.

Cette observation définit la granularité de notre analyse : celle-ci considère des *ensembles de blocs atteignables depuis une variable*, par déréférencements et décalage. Cette notion, formalisée ci-après, permet de formuler les résultats de l'analyse en termes d'ensembles de variables, plutôt que de zones mémoire.

Définition 14 (Référencement)**.**
On dit qu'un bloc b référence un bloc b' dans M, et on note $b \mapsto_M b'$, si :

$$\exists \kappa, \delta, \delta', \mathsf{load}(\kappa, M, b, \delta) = \lfloor \mathsf{Ptr}(b', \delta') \rfloor.$$

Pour un état mémoire M donné, on note $\mapsto_M^\star$ la clôture réflexive transitive de la relation de référencement $\mapsto_M$.

Définition 15 (Ensemble atteignable)**.**
Soit b le bloc associé à la variable x dans l'environnement global E : $E(x) = b$. L'ensemble atteignable depuis x, noté $\mathcal{R}_M(\&x)$, est l'ensemble des b' tels que $b \mapsto_M^\star b'$.

Remarque 1 (écriture d'un entier)**.**
Étant donné un état mémoire M_1 et un ensemble atteignable $\mathcal{R}_{M_1}(\&x)$, si M_2 est le résultat de l'écriture d'un entier dans M_1, alors $\mathcal{R}_{M_2}(\&x) = \mathcal{R}_{M_1}(\&x)$.

Démonstration. Par typage, l'écriture d'un entier ne peut pas écraser un pointeur présent en mémoire. Par conséquent, le fait qu'un bloc contienne un pointeur vers un autre bloc est invariant

par écriture d'un entier. Les relations de référencement sont préservées, donc les ensembles atteignables également. □

Propriété admise 5.1 (écriture hors d'un ensemble atteignable)**.**
Étant donné un état mémoire M_1 et un ensemble atteignable $\mathcal{R}_{M_1}(\&x)$, si M_2 est le résultat d'une écriture dans M_1 dans un bloc en dehors de $\mathcal{R}_{M_1}(\&x)$, alors $\mathcal{R}_{M_2}(\&x) = \mathcal{R}_{M_1}(\&x)$.

Propriété admise 5.2 (extension par écriture d'un ensemble atteignable)**.**
Si M_1 est un état mémoire, et M_2 le résultat de l'écriture d'un pointeur dans M_1 :

$$\mathsf{store}(\mathsf{mtype}(\tau*), M_1, b, \delta, \mathsf{Ptr}(b_v, \delta_v)) = \lfloor M_2 \rfloor,$$

alors on a l'inclusion suivante pour tout ensemble atteignable :

$$\mathcal{R}_{M_2}(a) \subseteq \mathcal{R}_{M_1}(a) \cup \{\ b' \mid b_v \mapsto^\star_{M_1} b'\ \}.$$

Lemme 3 (atteignabilité depuis une adresse de base)**.**
Tout bloc atteignable depuis une zone mémoire donnée l'est également depuis sa base syntaxique :

$$\forall a, M,\ \mathcal{R}_M(a) \subseteq \mathcal{R}_M(\&\mathsf{base}(a)).$$

Démonstration. Par induction sur la zone mémoire a. □

5.3.3 Système de contraintes

Nous disposons à présent de tous les éléments pour poser la définition de notre analyse. Celle-ci est définie comme un couple de fonctions $(\mathsf{live_{in}}, \mathsf{live_{out}})$ associant à chaque étiquette du CFG un ensemble de variables, qui sont alors dites *vivantes* : la vivacité d'une variable en un point signifie que tous les blocs atteignables depuis cette variable doivent être instrumentés en ce point. Par ailleurs, l'analyse définit pour chaque étiquette un résultat *précédent* l'exécution de l'instruction associée (donné par la fonction $\mathsf{live_{in}}$), et un résultat *suivant* son exécution (donné par la fonction $\mathsf{live_{out}}$).

Les fonctions $\mathsf{live_{in}}$ et $\mathsf{live_{out}}$ sont définies comme les solutions d'un système de contraintes (ici, un système d'inclusions ensemblistes), présenté dans son ensemble dans la figure 5.11. Les contraintes sur $\mathsf{live_{in}}$ et $\mathsf{live_{out}}$ expriment l'idée générale de l'analyse de vivacité : la vivacité des variables est générée au niveau des instructions qui en font usage, par la fonction gen, comme indiqué par l'inclusion $\mathsf{live_{in}}(\ell) \supseteq \mathsf{live_{out}}(\ell) \cup \mathsf{gen}(\ell)$, et propagée en arrière dans le CFG suivant la fonction de flot, via l'inclusion $\mathsf{live_{out}} \supseteq \bigcup \{\mathsf{live_{in}}(\ell') \mid (\ell, \ell') \in \Phi(s)\}$ (le cas $\mathsf{live_{out}} \supseteq \emptyset$ si $\ell \in \mathcal{F}(s)$ correspond au fait qu'aucune variable n'est vivante à la fin du programme). Le caractère sur-approximant de l'analyse se traduit par l'usage de l'union ensembliste dans cette contrainte : en présence de plusieurs branches dans le programme, il faut instrumenter les variables issues de toutes les branches.

La génération des variables vivantes est définie par la fonction gen. Dans notre cas, celle-ci intervient sur deux types d'instructions : d'une part les assertions (puisque c'est là qu'il est fait usage de l'instrumentation), où elle collecte les variables apparaissant dans le prédicat évalué, au moyen des fonction π et θ, et d'autre part les affectations de pointeur.

Dans ce dernier cas, la variable de base du membre droit de l'affectation est ajoutée aux variables vivantes, sous une certaine condition. Ce cas correspond en fait à celui où le membre gauche lv fait partie des variables vivantes, auquel cas le membre droit e doit lui aussi être instrumenté (puisque lv prend sa valeur). La condition pourrait donc être $\mathsf{base}(lv) \in \mathsf{live_{out}}(\ell)$, mais il nous faut également tenir compte de la possibilité d'un alias entre lv et une variable vivante, quand bien même lv n'est pas lui-même marqué comme vivant.

$$
\begin{array}{rcl}
\mathsf{live}_{\mathsf{out}}(\ell) & \supseteq & \begin{cases} \emptyset & \text{si } \ell \in \mathcal{F}(s) \\ \bigcup \{\mathsf{live}_{\mathsf{in}}(\ell') \mid (\ell,\ell') \in \Phi(s)\} & \text{sinon} \end{cases} \\
\mathsf{live}_{\mathsf{in}}(\ell) & \supseteq & \mathsf{live}_{\mathsf{out}}(\ell) \cup \mathsf{gen}(\ell)
\end{array}
$$

$$
\mathsf{gen}\left([lv = e;]^\ell\right) = \begin{cases} \{\mathsf{base}(e)\} & \text{si } lv \text{ est un pointeur, et} \\ & \exists\, x \in \mathsf{live}_{\mathsf{out}}(\ell),\ \&x \mapsto^\star_{\mathcal{A}} lv \\ \emptyset & \text{sinon} \end{cases}
$$

$$
\begin{array}{rclrclrcl}
\mathsf{gen}\left([\mathtt{skip};]^\ell\right) & = & \emptyset & \mathsf{gen}\left([e]^\ell\right) & = & \emptyset & \mathsf{gen}\left([p]^\ell\right) & = & \pi(p) \\
\mathsf{gen}\left([lv = \mathtt{malloc}(e);]^\ell\right) & = & \emptyset & \mathsf{gen}\left([\mathtt{free}(l);]^\ell\right) & = & \emptyset & & &
\end{array}
$$

$$
\begin{array}{rclrcl}
\pi(\backslash\mathtt{valid}(a)) & = & \{\mathsf{base}(a)\} & \pi(p_1 \wedge p_2) & = & \pi(p_1) \cup \pi(p_2) \\
\pi(\backslash\mathtt{initialized}(a)) & = & \{\mathsf{base}(a)\} & \pi(t_1 \bowtie t_2) & = & \theta(t_1) \cup \theta(t_2) \\
\pi(\neg p) & = & \pi(p) & & & \\
\theta(e) & = & \emptyset & \theta(\backslash\mathtt{base\_address}(a)) & = & \{\mathsf{base}(a)\} \\
\theta(\backslash\mathtt{offset}(a)) & = & \{\mathsf{base}(a)\} & \theta(\backslash\mathtt{block\_length}(a)) & = & \{\mathsf{base}(a)\}
\end{array}
$$

FIGURE 5.11 – Système de contraintes définissant l'analyse de flot de données.

C'est là qu'intervient l'analyse d'alias préalable à notre analyse : nous supposons que ses résultats sont donnés sous la forme d'une relation $\mapsto_{\mathcal{A}}$ sur les zones mémoires du programme, la relation $x \mapsto_{\mathcal{A}} y$ signifiant que x peut pointer vers y, et $\mapsto^\star_{\mathcal{A}}$ désignant la clôture transitive de $\mapsto_{\mathcal{A}}$. La possibilité que lv soit en alias avec une variable vivante est donc formulée ainsi : $\exists\, x \in \mathsf{live}_{\mathsf{out}}(\ell),\ \&x \mapsto^\star_{\mathcal{A}} lv$.

Notons que notre analyse ne comporte pas, contrairement à une analyse de vivacité classique, d'effacement d'une variable vivante, par une fonction complémentaire de la fonction gen, usuellement noté kill.

Enfin, nous ne nous intéressons pas ici à la mise en œuvre de cette analyse, qui est réalisée en pratique par le calcul d'un point fixe d'une certaine fonction de transfert, correspondant au système de contraintes défini par la figure 5.11. On peut cependant souligner le fait que, dans la pratique, les larges approximations effectuées (concernant les variables de base, notamment) résultent d'un parti pris délibéré en faveur de la vitesse de l'analyse, plutôt que de sa précision.

5.4 Sûreté de l'analyse

La raison pour laquelle on souhaite spécifiquement *sur-approximer* l'ensemble dont il est question est la *sûreté* de l'outil. Dans le contexte d'un outil de vérification d'assertions à l'exécution comme E-ACSL, on dira que l'outil est sûr s'il évalue correctement toutes les assertions à vérifier dans le programme. En particulier, il ne peut considérer comme satisfaite une assertion qui, en réalité, ne l'est pas. Or, réduire l'instrumentation des accès mémoire revient à fournir au moniteur une information partielle sur l'état mémoire du programme. Pour que celui-ci puisse, malgré ce, établir le bon verdict, il est donc crucial qu'il dispose *a minima* des informations requises pour l'évaluation des assertions présentes dans le programme.

5.4.1 Théorèmes de sûreté

Nous exprimons donc la sûreté de l'analyse en comparant deux exécutions de programme, l'une correspondant intuitivement au programme non optimisé, donc intégralement instrumenté, et l'autre au programme optimisé, donc partiellement instrumenté. L'usage des résul-

tats d'analyse dans l'instrumentation partielle est traduite par l'hypothèse que les deux programmes ont la même mémoire d'observation, *s'agissant de la région calculée par l'analyse*. Cela se traduit formellement par l'usage d'une relation $\equiv_{\mathcal{B}}$, définie ci-après, qui est paramétrée par un ensemble de blocs $\mathcal{B}$ et qui indique que les deux mémoires d'observation en relation coïncident sur $\mathcal{B}$. Cet ensemble est constitué de l'ensemble des blocs atteignables depuis les (blocs des) variables marquées comme vivantes au point considéré. Le diagramme suivant illustre ce principe, dans le cas d'un pas d'exécution qui se conclut :

$$\begin{array}{rcl} \langle [s]^{\ell},\ M_1,\ \overline{M_1}\rangle & \to & \langle M_2,\ \overline{M_2}\rangle \\ \equiv_{\mathcal{B}_1} & & \equiv_{\mathcal{B}_2} \\ \langle [s]^{\ell},\ M_1,\ \overline{M_1'}\rangle & \to & \langle M_2,\ \overline{M_2'}\rangle \end{array}$$

Les ensembles $\mathcal{B}_1$ et $\mathcal{B}_2$ désignent respectivement les blocs atteignables (au sens de la section 5.3.2) depuis les variables marquées vivantes par l'analyse, avant (fonction $\mathsf{live}_{\mathsf{in}}$) et après (fonction $\mathsf{live}_{\mathsf{out}}$) l'instruction s.

Théorème 4.
Soit $\langle s, M_1, \overline{M_1}\rangle \to \langle M_2, \overline{M_2}\rangle$ un pas d'exécution, et soit $\overline{M_1'}$ une mémoire d'observation telle que $\overline{M_1} \equiv_{\mathcal{B}_1} \overline{M_1'}$, avec $\mathcal{B}_1 \triangleq \bigcup_{x \in \mathsf{live}_{\mathsf{in}}(\mathcal{I}(s))} \mathcal{R}_{M_1}(\&x)$.

Alors il existe une mémoire d'observation $\overline{M_2'}$ telle que $\langle s, M_1, \overline{M_1'}\rangle \to \langle M_2, \overline{M_2'}\rangle$ et $\overline{M_2} \equiv_{\mathcal{B}_2} \overline{M_2'}$, avec $\mathcal{B}_2 = \bigcup_{x \in \mathsf{live}_{\mathsf{out}}(\mathcal{I}(s))} \mathcal{R}_{M_2}(\&x)$.

Théorème 5.
Soit $\langle s_1, M_1, \overline{M_1}\rangle \to \langle s_2, M_2, \overline{M_2}\rangle$ un pas d'exécution, et soit $\overline{M_1'}$ une mémoire d'observation telle que $\overline{M_1} \equiv_{\mathcal{B}_1} \overline{M_1'}$, avec $\mathcal{B}_1 \triangleq \bigcup_{x \in \mathsf{live}_{\mathsf{in}}(\mathcal{I}(s_1))} \mathcal{R}_{M_1}(\&x)$.

Alors il existe une mémoire d'observation $\overline{M_2'}$ telle que $\langle s_1, M_1, \overline{M_1'}\rangle \to \langle s_2, M_2, \overline{M_2'}\rangle$ et $\overline{M_2} \equiv_{\mathcal{B}_2} \overline{M_2'}$, avec $\mathcal{B}_2 = \bigcup_{x \in \mathsf{live}_{\mathsf{in}}(\mathcal{I}(s_2))} \mathcal{R}_{M_2}(\&x)$.

Ces théorèmes sont tous deux conditionnés par l'hypothèse de sûreté de l'analyse d'alias : en effet, puisque celle-ci détermine en partie notre propre analyse, la sûreté de la seconde dépend de celle de la première. Il nous faut donc faire l'hypothèse que les résultats issus de l'analyse d'alias sont valides.

Hypothèse 2 (sûreté de l'analyse d'alias)**.**
Soient s un programme, a et a' deux zones mémoire de ce programme, M un mémoire d'exécution, et (b, δ) et (b', δ') deux emplacements mémoire tels que d'une part $M \vDash_{\mathrm{e}} a \Downarrow \mathsf{Ptr}(b, \delta)$ et d'autre part $M \vDash_{\mathrm{e}} a' \Downarrow \mathsf{Ptr}(b, \delta')$. Alors, $b \mapsto^{\star}_{M} b' \implies a \mapsto^{\star}_{\mathcal{A}} a'$.

Avant d'exposer les preuves de ces deux théorèmes, on pose dans la section suivante la définition formelle de la relation d'équivalence sur un ensemble de blocs, et on énonce quelques-unes de ses propriétés, qui serviront pour la démonstration des théorèmes.

5.4.2 Mémoires d'observation équivalentes

La définition d'équivalence ci-dessous, qui concerne la mémoire d'observation et est paramétrée par un ensemble de blocs, est à distinguer de celle définie dans le chapitre 4, qui concerne la mémoire d'exécution.

Définition 16 (équivalence)**.**
Deux mémoires d'observation $\overline{M}$ et $\overline{M'}$ sont dites équivalentes sur $\mathcal{B}$, ce que l'on note $\overline{M} \equiv_{\mathcal{B}} \overline{M'}$,

si :

$$\forall b \in \mathcal{B}, \begin{cases} & \mathsf{is\_valid}(\overline{M}, b) & \iff & \mathsf{is\_valid}(\overline{M'}, b) \\ & \mathsf{length}(\overline{M}, b) & = & \mathsf{length}(\overline{M'}, b) \\ \forall \kappa, \forall \delta \in \mathbb{Z} & \mathsf{is\_initialized}(\kappa, \overline{M}, b, \delta) & = & \mathsf{is\_initialized}(\kappa, \overline{M'}, b, \delta). \end{cases}$$

Propriété admise 5.3 (équivalence sur un sous-ensemble)**.**
Si $\overline{M} \equiv_X \overline{M'}$ et $Y \subseteq X$, alors $\overline{M} \equiv_Y \overline{M'}$.

Propriété admise 5.4 (équivalence sur une union de blocs)**.**
Si $\overline{M} \equiv_X \overline{M'}$ et $\overline{M} \equiv_Y \overline{M'}$, alors $\overline{M} \equiv_{X \cup Y} \overline{M'}$.

Propriété admise 5.5 (équivalence après initialisation)**.**
Si $\overline{M_2} \triangleq \mathsf{initialize}(\kappa, \overline{M_1}, b, \delta)$, alors $\overline{M_1} \equiv_{\{b\}^\complement} \overline{M_2}$.

Propriété admise 5.6 (préservation d'équivalence par enregistrement d'un bloc)**.**
Soient $\overline{M_1}$ et $\overline{M'_1}$ deux mémoires d'observation équivalentes sur un ensemble de blocs X : $\overline{M_1} \equiv_X \overline{M'_1}$. Soit $\overline{M_2}$ (respectivement $\overline{M'_2}$) le résultat de l'enregistrement d'un bloc b de longueur n dans $\overline{M_1}$ (respectivement $\overline{M'_1}$) :

$$\begin{aligned} \overline{M_2} &\triangleq \mathsf{alloc}(\overline{M_1}, b, n) \\ \overline{M'_2} &\triangleq \mathsf{alloc}(\overline{M'_1}, b, n). \end{aligned}$$

Alors $\overline{M_2} \equiv_X \overline{M'_2}$.

Propriété admise 5.7 (préservation d'équivalence par initialisation)**.**
Soient $\overline{M_1}$ et $\overline{M'_1}$ deux mémoires d'observation équivalentes sur un ensemble de blocs X : $M_1 \equiv_X M'_1$. Soit $\overline{M_2}$ (respectivement $\overline{M'_2}$) le résultat de l'initialisation d'un emplacement (b, δ) dans $\overline{M_1}$ (respectivement $\overline{M_1}'$) :

$$\begin{aligned} \overline{M_2} &\triangleq \mathsf{initialize}(\kappa, \overline{M_1}, b, \delta) \\ \overline{M'_2} &\triangleq \mathsf{initialize}(\kappa, \overline{M'_1}, b, \delta). \end{aligned}$$

Alors $\overline{M_2} \equiv_X \overline{M'_2}$.

Propriété admise 5.8 (préservation d'équivalence par suppression d'un bloc)**.**
Soient $\overline{M_1}$ et $\overline{M'_1}$ deux mémoires d'observation équivalentes sur un ensemble de blocs X : $M_1 \equiv_X M'_1$. Soit $\overline{M_2}$ (respectivement $\overline{M'_2}$) le résultat de la suppression d'un bloc b dans $\overline{M_1}$ (respectivement $\overline{M_1}$) :

$$\begin{aligned} \overline{M_2} &\triangleq \mathsf{delete\_block}(\overline{M_1}, b) \\ \overline{M'_2} &\triangleq \mathsf{delete\_block}(\overline{M'_1}, b). \end{aligned}$$

Alors $\overline{M_2} \equiv_X \overline{M'_2}$.

5.4.3 Preuve du théorème 4

Démonstration. Par disjonction de cas sur l'évaluation d'instruction.

Cas S-ASSIGN

$$\frac{M_1 \vDash_{\mathrm{e}} e : \tau \Downarrow v \quad \mathsf{store}(\mathsf{mtype}(\tau), M_1, b, \delta, v) = \lfloor M_2 \rfloor \qquad M_1 \vDash_{\mathrm{lv}} l \Downarrow (b, \delta) \quad \mathsf{initialize}(\mathsf{mtype}(\tau), \overline{M_1}, b, \delta) = \overline{M_2}}{\langle [l = e;]^\ell,\ M_1,\ \overline{M_1} \rangle \to \langle M_2,\ \overline{M_2} \rangle}$$

Les régions définies par l'analyse sont :

$$\mathcal{B}_1 \triangleq \bigcup_{x \in \mathsf{live_{in}}(\ell)} \mathcal{R}_{M_1}(\&x)$$
$$\mathcal{B}_2 \triangleq \bigcup_{x \in \mathsf{live_{out}}(\ell)} \mathcal{R}_{M_2}(\&x)$$

On définit $\overline{M_2'} \triangleq \mathsf{initialize}(\mathsf{mtype}(\tau), \overline{M_1'}, b, \delta)$, et on cherche à montrer que $\overline{M_2} \equiv_{\mathcal{B}_2} \overline{M_2'}$. Par hypothèse $\overline{M_1} \equiv_{\mathcal{B}_1} \overline{M_1'}$, et par application de la propriété 5.5, $\overline{M_2} \equiv_{\{b\}^{\complement}} \overline{M_1}$ et $\overline{M_2'} \equiv_{\{b\}^{\complement}} \overline{M_1'}$. On peut écrire ces trois équivalences ainsi :

$$\overline{M_2} \equiv_{\{b\}^{\complement}} \overline{M_1} \equiv_{\mathcal{B}_1} \overline{M_1'} \equiv_{\{b\}^{\complement}} \overline{M_2'}. \tag{5.1}$$

On distingue ensuite deux cas, selon l'appartenance de b à $\mathcal{B}_2$.

Dans le cas où $b \notin \mathcal{B}_2$, on commence par montrer que $\mathcal{B}_2 \subseteq \mathcal{B}_1$. Par définition de l'analyse de flot de données, $\forall \ell, \mathsf{live_{in}}(\ell) \supseteq \mathsf{live_{out}}(\ell)$, donc il suffit de montrer que $\forall x \in \mathsf{live_{out}}(\ell), \mathcal{R}_{M_1}(\&x) = \mathcal{R}_{M_2}(\&x)$. Or, b n'appartient à aucun des $\mathcal{R}_{M_2}(\&x)$, on peut donc appliquer la propriété 5.1 pour montrer les égalités souhaitées. Il suffit de désormais de réécrire l'hypothèse $b \notin \mathcal{B}_2$ en une inclusion $\mathcal{B}_2 \subseteq \{b\}^{\complement}$, de sorte que $\mathcal{B}_2$ est un sous-ensemble de tous les ensembles de la chaîne d'équivalence équation (5.1). On en déduit $\overline{M_2} \equiv_{\mathcal{B}_2} \overline{M_2'}$ grâce à la propriété 5.3.

Dans le cas où $b \in \mathcal{B}_2$, on distingue deux cas selon le type de e (entier ou pointeur). Dans les deux cas, on montre séparément $\overline{M_2} \equiv_{\{b\}} \overline{M_2'}$ et $\overline{M_2} \equiv_{\mathcal{B}_2 \setminus \{b\}} \overline{M_2'}$, puis on applique la propriété 5.4.

Si e est de type entier, on montre que $\mathcal{B}_2 \subseteq \mathcal{B}_1$ par le même raisonnement que dans le cas précédent, les égalités entre ensembles atteignables résultant cette fois de la remarque 1. On peut alors appliquer la propriété 5.3 aux inclusions $\mathcal{B}_2 \setminus \{b\} \subseteq \{b\}^{\complement}$ et $\mathcal{B}_2 \setminus \{b\} \subseteq \mathcal{B}_1$ pour conclure $\overline{M_2} \equiv_{\mathcal{B}_2 \setminus \{b\}} \overline{M_2'}$. Il reste alors à montrer que $\overline{M_2} \equiv_{\{b\}} \overline{M_2'}$. $b \in \mathcal{B}_2 \subseteq \mathcal{B}_1$, donc on peut appliquer la propriété 5.3 à l'hypothèse $\overline{M_1} \equiv_{\mathcal{B}_1} \overline{M_1'}$: on obtient ainsi $\overline{M_1} \equiv_{\{b\}} \overline{M_1'}$. On conclut en appliquant la propriété 5.7.

Si e est de type pointeur, on montre dans un premier temps que $\mathcal{B}_2 \subseteq \mathcal{B}_1$, ce qui permet ensuite de conclure de la même manière que dans le cas où e est de type entier. On commence par déterminer un sur-ensemble de $\mathcal{B}_2$ en appliquant la propriété 5.2 :

$$\mathcal{B}_2 \subseteq \bigcup_{x \in \mathsf{live_{out}}(\ell)} \mathcal{R}_{M_1}(\&x) \cup \mathcal{R}_{M_1}(e).$$

Il s'agit ensuite de montrer que chacun des ensembles du membre droit de l'inclusion est un inclus dans $\mathcal{B}_1$.

On a $\mathsf{live_{out}}(\ell) \subseteq \mathsf{live_{in}}(\ell)$ par définition de l'analyse, et $\mathcal{B}_1 \triangleq \bigcup_{x \in \mathsf{live_{in}}(\ell)} \mathcal{R}_{M_1}(\&x)$, donc :

$$\bigcup_{x \in \mathsf{live_{out}}(\ell)} \mathcal{R}_{M_1}(\&x) \subseteq \bigcup_{x \in \mathsf{live_{in}}(\ell)} \mathcal{R}_{M_1}(\&x) = \mathcal{B}_1.$$

Il reste à montrer que $\mathcal{R}_{M_1}(e) \subseteq \mathcal{B}_1$. D'après le lemme 3, $\mathcal{R}_{M_1}(e) \subseteq \mathcal{R}_{M_1}(\&\mathsf{base}(e))$. Il suffit donc de montrer que $\mathsf{base}(e) \in \mathsf{live_{in}}(\ell)$, sachant que, par définition de l'analyse, $\mathsf{live_{in}}(\ell) \supseteq \mathsf{gen}([l = e;]^\ell)$. Dans le cas où e est un pointeur, la définition de l'analyse spécifie que $\mathsf{gen}([l = e;]^\ell) = \{\mathsf{base}(e)\}$ à la condition suivante :

$$\exists x \in \mathsf{live_{out}}(\ell),\ \&x \mapsto^\star_{\mathcal{A}} l.$$

Or $b \in \mathcal{B}_2 \triangleq \bigcup_{x \in \mathsf{live_{out}}(\ell)} \mathcal{R}_{M_2}(\&x)$, donc, par définition, il existe un $x \in \mathsf{live_{out}}(\ell)$ tel que $b \in \mathcal{R}_{M_2}(\&x)$. Soit b_x le bloc statiquement alloué pour la variable $x : b_x \triangleq E(x)$. On a alors, par définition d'atteignabilité : $b_x \mapsto^\star_{M_2} b$. On conclut ce cas en appliquant l'hypothèse de sûreté de l'analyse d'alias $\mathcal{A}$ (hypothèse 2).

Cas S-MALLOC

$$\frac{\begin{array}{ll} M_1 \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(n) & \mathsf{store\_block}(\overline{M_1}, b, n) = \overline{M_2} \\ M_1 \vDash_{\mathrm{lv}} l : \tau* \Downarrow (b_l, \delta_l) & \mathsf{initialize}(\mathsf{mtype}(\tau*), \overline{M_2}, b_l, \delta_l) = \overline{M_3} \\ \mathsf{alloc}(M_1, n) = (b', M_2) & \mathsf{store}(\mathsf{mtype}(\tau*), M_2, b_l, \delta_l, \mathsf{Ptr}(b, 0)) = \lfloor M_3 \rfloor \end{array}}{\langle [l = \texttt{malloc}(e);]^\ell,\ M_1, \overline{M_1} \rangle \to \langle M_3,\ \overline{M_3} \rangle}$$

La sémantique de l'instruction d'allocation fait apparaître deux étapes : d'abord l'allocation à proprement parler, puis l'écriture du pointeur vers le bloc alloué. On transpose ces étapes dans la démonstration en posant les définitions suivantes :

$$\begin{array}{rcl} \overline{M_2}' & \triangleq & \mathsf{store\_block}(\overline{M_1'}, b, n) \\ \overline{M_3}' & \triangleq & \mathsf{initialize}(\mathsf{mtype}(\tau*), \overline{M_2'}, b_l, \delta_l) \\ \mathcal{B}_2 & \triangleq & \bigcup_{x \in \mathsf{live_{in}}(\ell)} \mathcal{R}_{M_2}(\&x) \end{array}$$

On procède ensuite en deux étapes : on montre d'abord l'équivalence $\overline{M_2} \equiv_{\mathcal{B}_2} \overline{M_2'}$, puis on utilise celle-ci pour établir $\overline{M_3} \equiv_{\mathcal{B}_3} \overline{M_3'}$.

Pour montrer que $\overline{M_2} \equiv_{\mathcal{B}_2} \overline{M_2'}$, il suffit de remarquer que l'allocation d'un bloc ne modifie pas les ensembles atteignables, puisque le nouveau bloc n'est référencé par aucun pointeur existant. Par conséquent, $\forall x \in \mathsf{live_{in}}(\ell),\ \mathcal{R}_{M_1}(\&x) = \mathcal{R}_{M_2}(\&x)$, et donc $\mathcal{B}_2 = \mathcal{B}_1$. Puisque, par hypothèse, $\overline{M_1} \equiv_{\mathcal{B}_1} \overline{M_1'}$, on obtient par application de la propriété 5.6 l'équivalence $\overline{M_2} \equiv_{\mathcal{B}_2} \overline{M_2'}$.

Montrons maintenant que $\overline{M_3} \equiv_{\mathcal{B}_3} \overline{M_3'}$. Le raisonnement est similaire à celui employé dans le cas d'une affectation de pointeur, car les opérations mémoire en jeu sont les mêmes : écriture d'un pointeur en mémoire d'exécution, et initialisation d'un emplacement en mémoire d'observation. Le bloc b est nouvellement enregistré dans $\overline{M_2}$, tout comme dans $\overline{M_2'}$; on a donc $\overline{M_2} \equiv_{\{b\}} \overline{M_2'}$. Puisque $\overline{M_2} \equiv_{\mathcal{B}_2} \overline{M_2'}$, on en déduit d'après la propriété 5.4 l'équivalence $\overline{M_2} \equiv_{\mathcal{B}_2 \cup \{b\}} \overline{M_2'}$. On distingue maintenant deux cas, selon l'appartenance de b_l à $\mathcal{B}_2$.

Si $b_l \notin \mathcal{B}_2$, alors d'après la propriété 5.1, pour tout $x \in \mathsf{live_{in}}(\ell)$, l'ensemble atteignable $\mathcal{R}_{M_2}(\&x)$ demeure inchangé après l'écriture dans b_l, donc $\mathcal{B}_3 = \mathcal{B}_2$. Il suffit maintenant de déduire $\overline{M_3} \equiv_{\mathcal{B}_2} \overline{M_3'}$ de la propriété 5.7, sachant qu'on a établi au paragraphe précédent $\overline{M_2} \equiv_{\mathcal{B}_2} \overline{M_2'}$.

Considérons enfin le cas $b_l \in \mathcal{B}_2$. D'après la propriété 5.2, lors de l'écriture du pointeur $\mathsf{Ptr}(b, 0)$, chacun des ensembles atteignables composant $\mathcal{B}_2$ est étendu au plus par l'ensemble $\{\ b' \mid b \mapsto^\star_{M_2} b'\ \}$. Or, b est un bloc nouvellement alloué, par conséquent il ne contient aucune valeur, donc aucun pointeur, donc l'ensemble ci-dessus est réduit à $\{b\}$. On a donc : $\forall x \in \mathsf{live_{in}}(\ell),\ \mathcal{R}_{M_3}(\&x) \subseteq \mathcal{R}_{M_2}(\&x) \cup \{b\}$, d'où $\mathcal{B}_{M_3} \subseteq \mathcal{B}_{M_2} \cup \{b\}$. On conclut en appliquant la propriété 5.7.

Cas S-FREE

$$\frac{M_1 \vDash_{\mathrm{e}} a \Downarrow (b, 0) \qquad \mathsf{free}(M_1, b) = \lfloor M_2 \rfloor \qquad \mathsf{delete\_block}(\overline{M_1}, b) = \overline{M_2}}{\langle [\texttt{free}(a);]^\ell,\ M_1,\ \overline{M_1} \rangle \to \langle M_2,\ \overline{M_2} \rangle}$$

$\overline{M_2}$ est le résultat de la suppression d'un bloc dans $\overline{M_1}$, tout ensemble atteignable dans $\overline{M_1}$ ne peut que décroître dans $\overline{M_2}$. Plus formellement, $\forall x \in \mathsf{live_{in}}(\ell),\ \mathcal{R}_{M_2}(\&x) \subseteq \mathcal{R}_{M_1}(\&x)$. Par

ailleurs, par définition de l'analyse, $\mathsf{live}_{\mathsf{in}}(\ell) \supseteq \mathsf{live}_{\mathsf{out}}(\ell)$, donc $\mathcal{B}_1 \supseteq \mathcal{B}_2$. Par application de la propriété 5.3 à l'hypothèse $\overline{M_1} \equiv_{\mathcal{B}_1} \overline{M_1'}$, on en déduit $\overline{M_1} \equiv_{\mathcal{B}_2} \overline{M_1'}$. On conclut par application de la propriété 5.8.

Cas S-SKIP

$$\text{S-SKIP}\quad \frac{}{\langle [\texttt{skip}]^{\ell},\ M,\ \overline{M}\rangle \to \langle M,\ \overline{M}\rangle}$$

Par définition de l'analyse, $\mathsf{live}_{\mathsf{in}}(\ell) \supseteq \mathsf{live}_{\mathsf{out}}(\ell)$, donc $\mathcal{B}_1 \supseteq \mathcal{B}_2$, donc, puisque $M \equiv_{\mathcal{B}_1} M'$ par hypothèse, on a $M \equiv_{\mathcal{B}_2} M'$ d'après la propriété 5.3.

Cas S-WHILE-END

$$\text{S-WHILE-END}\quad \frac{M \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(0)}{\langle \texttt{while}\ ([e]^{\ell})\ s, M, \overline{M}\rangle \to \langle M, \overline{M}\rangle}$$

Par le même raisonnement que dans le cas précédent, $\mathcal{B}_1 \supseteq \mathcal{B}_2$ donc $M \equiv_{\mathcal{B}_2} M'$.

Cas S-ASSERT

$$\text{S-ASSERT}\quad \frac{M \vDash_{\mathrm{p}} p \Downarrow \top}{\langle [\texttt{logical\_assert}(p);]^{\ell},\ M,\ \overline{M}\rangle \to \langle M,\ \overline{M}\rangle}$$

Par le même raisonnement que dans le cas S-SKIP, $\mathcal{B}_1 \supseteq \mathcal{B}_2$ donc $M \equiv_{\mathcal{B}_2} M'$.

□

5.4.4 Preuve du théorème 5

Démonstration. Soit $\langle s_1, M_1, \overline{M_1}\rangle \to \langle s_2, M_2, \overline{M_2}\rangle$ un pas d'exécution, et soit $\overline{M_1'}$ une mémoire d'observation telle que $\overline{M_1} \equiv_{\mathcal{B}_1} \overline{M_1'}$, avec $\mathcal{B}_1 = \bigcup_{x \in \mathsf{live}_{\mathsf{in}}(\mathcal{I}(s_1))} \mathcal{R}_{M_1}(\&x)$.

On veut montrer l'existence de $\overline{M_2'}$ telle que $\langle s_1, M_1, \overline{M_1'}\rangle \to \langle s_2, M_2, \overline{M_2'}\rangle$ et $\overline{M_2} \equiv_{\mathcal{B}_2} \overline{M_2'}$, avec $\mathcal{B}_2 = \bigcup_{x \in \mathsf{live}_{\mathsf{in}}(\mathcal{I}(s_2))} \mathcal{R}_{M_2}(\&x)$.

D'après la sémantique des instructions du langage d'analyse, les cas où l'exécution se poursuit correspondent à l'évaluation des instructions de flot de contrôle du programme : séquence, conditionnelle, ou boucle. Or, ces constructions ne modifient pas la mémoire du programme. Dans tous les cas traités ci-après, il suffira donc pour obtenir l'inclusion sur les ensembles de blocs $\mathcal{B}_1 \supseteq \mathcal{B}_2$ d'établir celle sur les variables vivantes $\mathsf{live}_{\mathsf{in}}(\mathcal{I}(s_1)) \supseteq \mathsf{live}_{\mathsf{in}}(\mathcal{I}(s_2))$.

Cas S-SEQ-END

$$\text{S-SEQ-END}\quad \frac{\langle s_1, M_1, \overline{M_1}\rangle \to \langle M_2, \overline{M_2}\rangle}{\langle s_1\ s_2, M_1, \overline{M_1}\rangle \to \langle s_2, M_2, \overline{M_2}\rangle}$$

Par application du théorème 4 à la prémisse de l'évaluation, on a $\overline{M_2} \equiv_{\mathcal{B}} \overline{M_2'}$, où $\mathcal{B} \triangleq \bigcup_{x \in \mathsf{live}_{\mathsf{out}}(\mathcal{I}(s_1))} \mathcal{R}_{M_2}(\&x)$. Or, par définition de l'analyse, $\mathsf{live}_{\mathsf{out}}(\mathcal{I}(s_1)) \supseteq \mathsf{live}_{\mathsf{in}}(\mathcal{I}(s_2))$, donc $\mathcal{B} \supseteq \mathcal{B}_2$. On conclut par application de la propriété 5.3.

Cas S-SEQ-CONT

$$\text{S-SEQ-CONT}\quad \frac{\langle s_1, M_1, \overline{M_1}\rangle \to \langle s_1', M_2, \overline{M_2}\rangle}{\langle s_1\ s_2, M_1, \overline{M_1}\rangle \to \langle s_1'\ s_2, M_2, \overline{M_2}\rangle}$$

On cherche $\overline{M_2'}$ tel que $\overline{M_2} \equiv_{\mathcal{B}_2} \overline{M_2'}$, où $\mathcal{B}_2 \triangleq \bigcup_{x \in \mathsf{live_{in}}(\mathcal{I}(s_1'\ s_2))} \mathcal{R}_{M_2}(\&x)$. Or, par hypothèse d'induction, il existe $\overline{M_2'}$ tel que $\overline{M_2} \equiv_{\mathcal{B}} \overline{M_2'}$, où $\mathcal{B} \triangleq \bigcup_{x \in \mathsf{live_{in}}(\mathcal{I}(s_1'))} \mathcal{R}_{M_2}(\&x)$. De plus, par définition de $\mathcal{I}$, $\mathcal{I}(s_1'\ s_2) = \mathcal{I}(s_1')$. L'équivalence recherchée est ainsi établie.

Cas S-IF-TRUE

$$\text{S-IF-TRUE}\quad \frac{M \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(n) \qquad n \neq 0}{\langle \texttt{if}\ ([e]^{\ell})\ \texttt{then}\ s_t\ \texttt{else}\ s_f, M, \overline{M}\rangle \to \langle s_t, M, \overline{M}\rangle}$$

On a, par hypothèse, $\overline{M'}$ tel que $\overline{M} \equiv_{\mathcal{B}_1} \overline{M'}$, où $\mathcal{B}_1 \triangleq \bigcup_{x \in \mathsf{live_{in}}(\ell)} \mathcal{R}_M(\&x)$. On cherche à montrer que $\overline{M} \equiv_{\mathcal{B}_2} \overline{M'}$, où $\mathcal{B}_2 \triangleq \bigcup_{x \in \mathsf{live_{in}}(\mathcal{I}(s_t))} \mathcal{R}_M(\&x)$.

Or, par définition du CFG, $(\ell, \mathcal{I}(s_t)) \in \Phi(\texttt{if}\ ([e]^{\ell})\ \texttt{then}\ s_t\ \texttt{else}\ s_f)$, donc $\mathsf{live_{in}}(\ell) \supseteq \mathsf{live_{in}}(\mathcal{I}(s_t))$. On en déduit que $\mathcal{B}_1 \supseteq \mathcal{B}_2$, ce qui permet de conclure par application de la propriété 5.3.

Cas S-IF-FALSE

$$\text{S-IF-FALSE}\quad \frac{M \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(0)}{\langle \texttt{if}\ ([e]^{\ell})\ \texttt{then}\ s_t\ \texttt{else}\ s_f, M, \overline{M}\rangle \to \langle s_f, M, \overline{M}\rangle}$$

Ce cas est démontré de la même manière que le précédent, en remplaçant s_t par s_f.

Cas S-WHILE-CONT

$$\text{S-WHILE-CONT}\quad \frac{M \vDash_{\mathrm{e}} e \Downarrow \mathsf{Int}(n) \qquad n \neq 0}{\langle \texttt{while}\ ([e]^{\ell})\ s, M, \overline{M}\rangle \to \langle s\ \texttt{while}\ ([e]^{\ell})\ s, M, \overline{M}\rangle}$$

On a, par hypothèse, $\overline{M'}$ tel que $\overline{M} \equiv_{\mathcal{B}_1} \overline{M'}$, où $\mathcal{B}_1 \triangleq \bigcup_{x \in \mathsf{live_{in}}(\ell)} \mathcal{R}_M(\&x)$. On cherche à montrer que $\overline{M} \equiv_{\mathcal{B}_2} \overline{M'}$, où $\mathcal{B}_2 \triangleq \bigcup_{x \in \mathsf{live_{in}}(\mathcal{I}(s\ \texttt{while}\ (e)\ s))} \mathcal{R}_M(\&x)$. Or, par définition, $\mathcal{I}(s\ \texttt{while}\ (e)\ s) = \mathcal{I}(s)$, donc on peut réécrire $\mathcal{B}_2 = \bigcup_{x \in \mathsf{live_{in}}(\mathcal{I}(s))} \mathcal{R}_M(\&x)$. De plus, par définition de la fonction de flot, $(\ell, \mathcal{I}(s)) \in \Phi(\texttt{while}\ ([e]^{\ell})\ s)$, donc $\mathsf{live_{in}}(\ell) \supseteq \mathsf{live_{in}}(\mathcal{I}(s))$. On en déduit $\mathcal{B}_1 \supseteq \mathcal{B}_2$, d'où la conclusion. □

5.5 Travaux connexes

La paramétrisation d'une transformation de programme par une analyse statique est un motif récurrent dans le domaine des langages de programmation, que l'on rencontre par exemple de manière quasi-systématique dans les compilateurs.

Dans le cas des instrumentations visant à assurer la sûreté mémoire, cette approche semble moins fréquente. Le débogueur mémoire MemSafe [SB13] utilise une analyse statique couplée à une instrumentation, opérant sur une représentation intermédiaire du compilateur LLVM [LA04].

De même, CCured [Nec+05] assure une partie de la sûreté mémoire par le biais d'une analyse statique, qui est complémentée par des vérifications à l'exécution pour les parties du programme où l'analyse ne parvient pas à établir la sûreté mémoire. L'analyse elle-même, qui prend la forme d'un système de types, est prouvée sûre.

CAWDOR [YJ12] est un outil de protection contre les vers (une classe de programme malveillants) par vérification dynamique, qui utilise une analyse statique à des fins d'optimisation, en aval de l'instrumentation (contrairement aux travaux cités précédemment, et à l'approche que nous avons présentée).

Les travaux que nous avons présenté dans ce chapitre s'appuient en partie sur une précédente étude [JKS16] de l'analyse pré-instrumentation d'E-ACSL. L'analyse qui y est définie est relativement proche de la nôtre, quoique plus complexe ; quant à sa sûreté, elle est conjecturée, et exprimée de manière semi-formelle.

Le souci de sûreté des analyses statiques a motivé un certain nombre de recherches visant à les formaliser à l'aide d'assistants de preuve. Ainsi, le compilateur CompCert inclut une analyse de flot de données [BGL06 ; Ler09], utilisée pour pratiquer la propagation de constantes et l'élimination de sous-expressions communes sur une représentation intermédiaire du compilateur.

Un outil d'analyse statique plus général est Verasco [Jou+15], un interpréteur abstrait pour le langage C, s'appuyant sur la sémantique de CompCert, mis en œuvre et prouvé correct dans l'assistant de preuve Coq.

Chapitre 6

Conclusion

Nous avons exposé dans ce mémoire une étude formelle d'E-ACSL, un générateur de moniteurs pour la vérification dynamique de programmes C, envisagé ici sous l'angle de la vérification de propriétés mémoire au moyen d'assertions.

6.1 Contributions

Tout d'abord, nous avons introduit un langage d'étude source, axé sur la gestion de la mémoire, et incluant des assertions sur des propriétés mémoire. Nous avons notamment proposé une sémantique pour les constructions du langage E-ACSL portant sur l'état de la mémoire, sémantique que nous avons exprimée vis-à-vis du modèle mémoire de notre langage, adapté de CompCert.

Nous avons ensuite défini l'instrumentation comme une transformation de programme, du langage source vers un langage cible, dont la principale caractéristique est d'intégrer une structure de données dédiée au suivi des propriétés mémoire : la mémoire d'observation. Cette dernière est caractérisée axiomatiquement de manière analogue à la mémoire dite d'exécution, et joue un rôle central dans la caractérisation de la transformation.

La transformation ainsi définie préserve la sémantique des programmes, ce que nous avons formalisé, au moyen d'un ensemble de relations sur les états mémoire, puis démontré.

Nous avons enfin proposé une analyse de flot de données pour l'optimisation de l'instrumentation, analyse dont nous avons établi la sûreté, ce qui signifie qu'opérer qu'une instrumentation partielle sur la base des résultats de l'analyse ne compromet pas la validité des verdicts délivrés par le moniteur.

6.2 Perspectives

Les travaux que nous avons présentés pourraient être améliorés selon un certain nombre d'axes différents.

Mémoire d'instrumentation À ce jour, la piste la plus prometteuse concerne, assez étonnamment, un objet qui n'apparaît nulle part en tant que tel dans nos travaux. Il se manifeste en fait en creux, dans la notion d'extension de mémoire, utilisée dans la preuve de préservation sémantique de l'instrumentation. L'extension est causée par l'introduction de variables intermédiaires utilisées pour l'évaluation des prédicats, lesquelles ne possèdent pas d'équivalent dans la mémoire du programme source. Il s'agit donc, en principe, d'une région de la mémoire distincte de celle où se déroule l'exécution du programme (hors instrumentation).

Cette description n'est pas sans rappeler le concept de mémoire d'observation, qui, une fois rendu explicite, s'est révélé primordial pour raisonner sur les propriétés sémantiques de la transformation. Aussi, il est permis d'espérer que dégager une notion de « mémoire d'instrumentation » à part entière apportera des bénéfices du même ordre, notamment dans le cas de schéma de traduction moins naïfs que ceux que nous avons considérés.

Extensions L'axe le plus évident, peut-être, au regard du contexte d'application d'E-ACSL et de Frama-C, serait le support d'une partie plus substantielle du langage C. Les types structures et unions, notamment, sont susceptibles de soulever des questions intéressantes en termes de propriétés mémoire ; ce, davantage encore en présence de coercions de types.

Une autre piste consiste à étendre non pas le langage de programmation, mais le langage de spécification. E-ACSL ne manque pas de constructions intéressantes du point de vue de l'instrumentation : on peut mentionner par exemple le prédicat `\valid_read`, qui introduit la notion de permission. Ce type d'extension affecterait principalement la mémoire d'observation, mais ne changerait *a priori* pas fondamentalement le schéma de traduction.

En revanche, l'ajout de constructions comme le `\at` d'E-ACSL, sorte de `goto` de la spécification, qui permet de faire référence à la valeur d'une expression en un autre point du programme que celui où figure l'annotation, appellerait selon toute vraisemblance un schéma de traduction assez différent.

Approfondissements Plutôt que d'étendre le périmètre du langage, il est aussi possible d'en affiner la sémantique. En effet, la sémantique que nous avons proposée pour le langage source (notre langage de référence) ne permet pas de rendre compte d'un certain nombre de comportements : erreurs d'exécution ou non-terminaison, par exemple.

Unification D'autre part, nous avons proposé deux sémantiques distinctes, pour deux langages très similaires (langages source et d'analyse) : il serait plus satisfaisant d'unifier les deux langages, et d'étudier, le cas échéant, l'équivalence entre les deux sémantiques proposées.

Mécanisation Nous avons pu constater, comme bien d'autres auparavant sans doute, que la manipulation d'objets formels un tant soit peu complexes avec pour seuls outils un crayon, du papier, et autant de vigilance que possible, constitue un exercice périlleux. Les multiples erreurs de raisonnement que nous avons rencontrées tout au long de cette thèse, parfois insignifiantes mais pas toujours, sont autant d'arguments en faveur d'une formalisation de ces travaux dans un assistant de preuve.

Exemple d'instrumentation complet

```
let t: int64* in
  store_block(&t, sizeof(int64*));
  let length: int64 in
    store_block(&length, sizeof(int64));
    let x: int64 in
      store_block(&x, sizeof(int64));
      length = 17;
      initialize(&length);
      t = malloc (length * sizeof(int64));
      store_block(t, length * sizeof(int64));
      initialize(&t);

      x = 0;
      initialize(&x);
      while (x < 17)
        *(t + x) = 2 * x + 1;
        initialize(&*(t+x));
        x = x + 1;
        initialize(&x);

      x = 5;
      initialize(&x);
      let idx: int64 in
        store_block(&idx, sizeof(int64));
        let lo: int64 in
          store_block(&lo, sizeof(int64));
          let hi: int64 in
            store_block(&hi, sizeof(int64));
            idx = -1;
            initialize(&idx);
            lo = 0;
            initialize(&lo);
            hi = length - 1 ;
            initialize(&hi);
            while (lo <= hi)
              let mid: int64 in
                store_block(&mid, sizeof(int64));
                mid = lo + (hi - lo) / 2;
                initialize(&mid);
                // logical_assert(\valid(t + mid));
                let res_0: int8 in
                  let res_1: int64* in
                    res_1 = t + mid;
                    res_0 = is_valid(res_1);
                  end
                  assert(res_0);
                end
                if (*(t + mid) == x) then
                  idx = mid;
```

```
50                  initialize(&idx);
51                  lo = hi + 1;
52                  initialize(&lo);
53                else if (*(t + mid) < x) then
54                  lo = mid + 1;
55                  initialize(&lo);
56                else
57                  hi = mid - 1;
58                  initialize(&hi);
59                delete_block(&mid);
60              end
61            delete_block(&hi);
62          end
63          delete_block(&lo);
64        end
65        delete_block(&idx);
66      end
67      delete_block(&x);
68    end
69    delete_block(&length);
70  end
71  delete_block(&t);
72 end
```

Bibliographie

[Ass15] Mounir ASSAF. « From qualitative to quantitative program analysis : permissive enforcement of secure information flow. (Approches qualitatives et quantitatives d'analyse de programmes : mise en oeuvre permissive de flux d'information sécurisés) ». Thèse de doct. University of Rennes 1, France, 2015.

[ASU86] Alfred V. AHO, Ravi SETHI et Jeffrey D. ULLMAN. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[Ayd+08] Brian E. AYDEMIR, Arthur CHARGUÉRAUD, Benjamin C. PIERCE, Randy POLLACK et Stephanie WEIRICH. « Engineering formal metatheory ». In : *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. ACM, 2008, p. 3-15. DOI : 10.1145/1328438.1328443.

[Ben04] Nick BENTON. « Simple relational correctness proofs for static analyses and program transformations ». In : *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. ACM, 2004, p. 14-25. DOI : 10.1145/964001.964003.

[BGL06] Yves BERTOT, Benjamin GRÉGOIRE et Xavier LEROY. « A Structured Approach to Proving Compiler Optimizations Based on Dataflow Analysis ». In : *Types for Proofs and Programs (TYPES)*. Springer, 2006.

[BL09] Sandrine BLAZY et Xavier LEROY. « Mechanized Semantics for the Clight Subset of the C Language ». In : *J. Autom. Reasoning* 43.3 (2009), p. 263-288. DOI : 10.1007/s10817-009-9148-3.

[BRS22] Thibaut BENJAMIN, Félix RIDOUX et Julien SIGNOLES. « Formalisation d'un vérificateur efficace d'assertions arithmétiques à l'exécution ». In : *33èmes Journées Francophones des Langages Applicatifs*. Saint-Médard-d'Excideuil, France, juin 2022, p. 24-41.

[BZ11] Derek BRUENING et Qin ZHAO. « Practical memory checking with Dr. Memory ». In : *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*. IEEE Computer Society, 2011, p. 213-223. DOI : 10.1109/CGO.2011.5764689.

[CC77] Patrick COUSOT et Radhia COUSOT. « Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints ». In : *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. ACM, 1977, p. 238-252. DOI : 10.1145/512950.512973.

[Che03] Yoonsik CHEON. « A runtime assertion checker for the Java Modeling Language ». Thèse de doct. Iowa State University, 2003.

[CR06] Lori A. CLARKE et David S. ROSENBLUM. « A historical perspective on runtime assertion checking in software development ». In : *ACM SIGSOFT Softw. Eng. Notes* 31.3 (2006), p. 25-37. DOI : 10.1145/1127878.1127900.

[CS12] Loïc CORRENSON et Julien SIGNOLES. « Combining Analyses for C Program Verification ». In : *Formal Methods for Industrial Critical Systems - 17th International Workshop, FMICS 2012, Paris, France, August 27-28, 2012. Proceedings*. 2012, p. 108-130. DOI : 10.1007/978-3-642-32469-7_8.

[Dij75] Edsger W. DIJKSTRA. « Guarded Commands, Nondeterminacy and Formal Derivation of Programs ». In : *Commun. ACM* 18.8 (1975), p. 453-457. DOI : 10.1145/360933.360975.

[DKS13] Mickaël DELAHAYE, Nikolai KOSMATOV et Julien SIGNOLES. « Common specification language for static and dynamic analysis of C programs ». In : *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*. ACM, 2013, p. 1230-1235. DOI : 10.1145/2480362.2480593.

[Flo67] R. W. FLOYD. « Assigning meanings to programs ». In : *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*. American Mathematical Society. Providence, 1967, 19-32.

[Her13] Paolo HERMS. « Certification of a Tool Chain for Deductive Program Verification ». Theses. Université Paris Sud - Paris XI, jan. 2013.

[Hin01] Michael HIND. « Pointer analysis : haven't we solved this problem yet ? » In : *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*. ACM, 2001, p. 54-61. DOI : 10.1145/379605.379665.

[Hoa69] C. A. R. HOARE. « An Axiomatic Basis for Computer Programming ». In : *Commun. ACM* 12.10 (1969), p. 576-580. DOI : 10.1145/363235.363259.

[JKS16] Arvid JAKOBSSON, Nikolai KOSMATOV et Julien SIGNOLES. « Fast as a shadow, expressive as a tree : Optimized memory monitoring for C ». In : *Sci. Comput. Program.* 132 (2016), p. 226-246. DOI : 10.1016/j.scico.2016.09.003.

[Jou+15] Jacques-Henri JOURDAN, Vincent LAPORTE, Sandrine BLAZY, Xavier LEROY et David PICHARDIE. « A Formally-Verified C Static Analyzer ». In : *SIGPLAN Not.* 50.1 (2015), p. 247-259. DOI : 10.1145/2775051.2676966.

[Kah87] Gilles KAHN. « Natural Semantics ». In : *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*. T. 247. Lecture Notes in Computer Science. Springer, 1987, p. 22-39. DOI : 10.1007/BFb0039592.

[Kil73] Gary A. KILDALL. « A Unified Approach to Global Program Optimization ». In : *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*. ACM Press, 1973, p. 194-206. DOI : 10.1145/512927.512945.

[Kir+15] Florent KIRCHNER, Nikolai KOSMATOV, Virgile PREVOSTO, Julien SIGNOLES et Boris YAKOBOWSKI. « Frama-C : A software analysis perspective ». In : *Formal Asp. Comput.* 27.3 (2015), p. 573-609. DOI : 10.1007/s00165-014-0326-7.

[Kre15] Robbert KREBBERS. « The C standard formalized in Coq ». Thèse de doct. Radboud University Nijmegen, 2015.

[LA04] Chris LATTNER et Vikram S. ADVE. « LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation ». In : *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 2004, p. 75-88. DOI : 10.1109/CGO.2004.1281665.

[LB08] Xavier LEROY et Sandrine BLAZY. « Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations ». In : *J. Autom. Reason.* 41.1 (2008), p. 1-31. DOI : 10.1007/s10817-008-9099-0.

[LBR06] Gary T. LEAVENS, Albert L. BAKER et Clyde RUBY. « Preliminary design of JML : a behavioral interface specification language for java ». In : *ACM SIGSOFT Software Engineering Notes* 31.3 (2006), p. 1-38. DOI : 10.1145/1127878.1127884.

[Leh11] Hermann LEHNER. « A Formal Definition of JML in Coq and its Application to Runtime Assertion Checking ». Thèse de doct. ETH Zurich, 2011.

[Ler09] Xavier LEROY. « A Formally Verified Compiler Back-end ». In : *J. Autom. Reason.* 43.4 (2009), p. 363-446. DOI : 10.1007/s10817-009-9155-4.

[LT93] Nancy G. LEVESON et Clark Savage TURNER. « Investigation of the Therac-25 Accidents ». In : *Computer* 26.7 (1993), p. 18-41. DOI : 10.1109/MC.1993.274940.

[Lue+06] Glenn R. LUECKE, James COYLE, Jim HOEKSTRA, Marina KRAEVA, Ying LI, Olga TABORSKAIA et Yanmei WANG. « A survey of systems for detecting serial run-time errors ». In : *Concurr. Comput. Pract. Exp.* 18.15 (2006), p. 1885-1907. DOI : 10.1002/cpe.1036.

[Ly+19] Dara LY, Nikolai KOSMATOV, Julien SIGNOLES et Frédéric LOULERGUE. « Soundness of a Dataflow Analysis for Memory Monitoring ». In : *Ada Lett.* 38.2 (déc. 2019), p. 97-108. DOI : 10.1145/3375408.3375416.

[Ly+20] Dara LY, Nikolai KOSMATOV, Frédéric LOULERGUE et Julien SIGNOLES. « Verified Runtime Assertion Checking for Memory Properties ». In : *Tests and Proofs - 14th International Conference, TAP@STAF 2020, Bergen, Norway, June 22-23, 2020, Proceedings [postponed]*. T. 12165. Lecture Notes in Computer Science. Springer, 2020, p. 100-121. DOI : 10.1007/978-3-030-50995-8_6.

[Mey08] Bertrand MEYER. « Seven Principles of Software Testing ». In : *Computer* 41.8 (2008), p. 99-101. DOI : 10.1109/MC.2008.306.

[Nec+05] George C. NECULA, Jeremy CONDIT, Matthew HARREN, Scott MCPEAK et Westley WEIMER. « CCured : Type-safe Retrofitting of Legacy Software ». In : *Programming Languages and Systems (TOPLAS)* 27 (2005).

[NN07] Hanne Riis NIELSON et Flemming NIELSON. *Semantics with Applications : An Appetizer*. Undergraduate Topics in Computer Science. Springer, 2007. DOI : 10.1007/978-1-84628-692-6.

[NNH99] Flemming NIELSON, Hanne Riis NIELSON et Chris HANKIN. *Principles of program analysis*. Springer, 1999. DOI : 10.1007/978-3-662-03811-6.

[Nor99] Michael NORRISH. « C formalised in HOL ». Thèse de doct. University of Cambridge, 1999.

[NS03] Nicholas NETHERCOTE et Julian SEWARD. « Valgrind : A Program Supervision Framework ». In : *Electron. Notes Theor. Comput. Sci.* 89.2 (2003), p. 44-66. DOI : 10.1016/S1571-0661(04)81042-9.

[NS07] Nicholas NETHERCOTE et Julian SEWARD. « How to shadow every byte of memory used by a program ». In : *Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE 2007, San Diego, California, USA, June 13-15, 2007*. 2007, p. 65-74. DOI : 10.1145/1254810.1254820.

[Ric53] Henry G. RICE. « Classes of Recursively Enumerable Sets and their Decision problems ». In : *Transactions of the American Mathematical Society* 83 (1953). DOI : 10.1090/S0002-9947-1953-0053041-6.

[SB13] Matthew S. SIMPSON et Rajeev BARUA. « MemSafe : ensuring the spatial and temporal memory safety of C at runtime ». In : *Software : Practice and Experience* 43 (2013).

[Ser+12] Konstantin SEREBRYANY, Derek BRUENING, Alexander POTAPENKO et Dmitriy VYUKOV. « AddressSanitizer : A Fast Address Sanity Checker ». In : *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*. 2012, p. 309-318.

[Sig18] Julien SIGNOLES. *From Static Analysis to Runtime Verification with Frama-C and E-ACSL*. Habilitation Thesis. Juill. 2018.

[Sig21] Julien SIGNOLES. « The E-ACSL perspective on runtime assertion checking ». In : *VORTEX 2021 : Proceedings of the 5th ACM International Workshop on Verification and mOnitoring at Runtime EXecution, Virtual Event, Denmark, 12 July 2021*. ACM, 2021, p. 8-12. DOI : 10.1145/3464974.3468451.

[ST12] Donald SANNELLA et Andrzej TARLECKI. *Foundations of Algebraic Specification and Formal Software Development*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2012. DOI : 10.1007/978-3-642-17336-3.

[Sze+13] Laszlo SZEKERES, Mathias PAYER, Tao WEI et Dawn SONG. « SoK : Eternal War in Memory ». In : *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 2013, p. 48-62. DOI : 10.1109/SP.2013.13.

[VSK17] Kostyantyn VOROBYOV, Julien SIGNOLES et Nikolai KOSMATOV. « Shadow state encoding for efficient monitoring of block-level properties ». In : *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management, ISMM 2017, Barcelona, Spain, June 18, 2017*. 2017, p. 47-58. DOI : 10.1145/3092255.3092269.

[YJ12] Jun YUAN et Rob JOHNSON. « CAWDOR : Compiler Assisted Worm Defense ». In : *Source Code Analysis and Manipulation (SCAM)*. IEEE, 2012.

Formalisation d'un vérificateur dynamique de propriétés mémoire pour programmes C

Résumé : La vérification d'assertions à l'exécution est une technique permettant de contrôler, lors de leur exécution, la conformité d'un programme vis-à-vis d'une spécification donnée sous forme d'annotations formelles : les assertions. Un procédé appelé instrumentation transforme les assertions en code exécutable, de manière à mettre en œuvre un moniteur en ligne pour le programme à vérifier. Le long de l'exécution du programme instrumenté, le moniteur contrôle la conformité du programme vis-à-vis des assertions, et, en cas de non-respect d'une assertion, met fin à l'exécution. Autrement, il laisse le comportement fonctionnel du programme inchangé.
La complexité de mise en œuvre de l'instrumentation dépend largement des propriétés exprimables dans le langage d'annotation. Dans le cas de programmes en langage C, l'outil E-ACSL (greffon de Frama-C, une plateforme open-source d'analyse de code C), permet la vérification de propriétés relatives à l'état mémoire du programme, mais requiert pour cela une instrumentation complexe.
La présente thèse est consacrée à la formalisation de cette instrumentation : il s'agit d'en donner une définition précise et d'étudier ses propriétés sémantiques. Nous proposons une modélisation du problème comme une traduction de programmes depuis un langage source, muni d'assertions logiques, vers un langage cible. Ce dernier est dépourvu d'assertions logique, mais intègre une structure de données, appelée mémoire d'observation, dédiée au suivi des propriétés mémoire. Nous donnons une caractérisation axiomatique de la mémoire d'observation, et utilisons celle-ci pour définir la sémantique du langage cible de la tranformation, dont nous montrons qu'elle est correcte vis-à-vis de la sémantique des langages concernés.
Additionellement, nous étudions l'optimisation de l'instrumentation par analyse de flot de données, technique mise en œuvre dans l'outil E-ACSL afin de réduire le surcoût en performance induit par l'instrumentation. L'analyse vise à déterminer un sous-ensemble d'emplacements mémoire minimal dont l'instrumentation permette au moniteur d'évaluer correctement les assertions du programme. Nous définissons une telle analyse, et prouvons qu'elle est sûre, au sens où limiter l'instrumentation aux seuls emplacements désignés par l'analyse ne compromet pas la validité des verdicts du moniteur.

Mots clés : propriétés mémoire, vérification dynamique, preuve formelle

Formalization of a Runtime Assertion Checker for Memory Properties of C Programs

Abstract : Runtime Assertion Checking is a program verification technique allowing to monitor programs during their execution, in order to check their validity with regards to some specification expressed as assertions, that is, formal annotations inserted in the source code. Assertions are translated into executable code through a process called instrumentation, thus implementing an inline monitor for the program. During execution, the monitor checks whether the program complies with its assertions, and aborts the execution in case of violation of an assertion. If no such violation occurs, the program's functional behavior is unchanged.
Instrumenting a program is a process whose implementation complexity is strongly related to the expressiveness of the annotation language. For C programs, the E-ACSL plugin of Frama-C (an opensource analysis platform for C programs), can express properties related to the memory state of the program, at the cost of a complex instrumentation.
In this thesis, we present a formalization of the instrumentation process, that is, a description accurate enough to enable formal reasoning about the semantic properties of the instrumentation. The problem is modeled as a translation, from a source language with logical assertions, to a target language. The latter has no logical assertions, instead featuring a data-structure we call observation memory, which is designed to keep track of memory properties. We present an axiomatic characterization of observation memory, and use it to define the translation's target language semantics. The translation is then proved correct with regards to the respective semanctics of source and target languages.
In addition, we study an optimization of the instrumentation through data-flow analysis. This optimization is implemented in E-ACSL in order to mitigate the performance costs of the instrumentation. The analysis aims at identifying a minimal subset of memory locations, whose instrumentation suffices for the monitor to evaluate the program's assertions soundly. We define such an analysis and prove its soundness, meaning that restricting the instrumentation to memory locations computed by the analysis does not threaten the validity of the monitor's verdicts.

Keywords : memory properties, runtime verification, formal proof

LABORATOIRE D'INFORMATIQUE FONDAMENTALE D'ORLEANS