

De C à B , l'analyse de code par les méthodes formelles

IPA Éric Jaeger, SGDN/DCSSI/SDS/LTI
MPRI2 (P6), Année 2004-2005

12 septembre 2005

Résumé

De nombreuses approches destinées à assurer la confiance dans les logiciels ont été proposées, qu'il s'agisse de méthodes de gestion de projets ou de nouveaux paradigmes de programmation ; ces approches ne sont cependant applicables qu'à des développements nouveaux.

La *Méthode B* est un exemple d'une telle approche, proposant un formalisme rigoureux qui permet d'écrire une spécification dans un langage défini par une sémantique précise, puis par des étapes successives de raffinements d'aboutir à un code que l'on peut prouver conforme à cette spécification. La *Méthode B* est implémentée par plusieurs outils, dont *Click'n'Prove/B4Free v2*.

La Direction centrale de la sécurité des systèmes d'information (DCSSI) est amenée à évaluer le niveau de confiance dans l'efficacité et la robustesse d'un produit de sécurité, en analysant les programmes qui entrent dans sa composition. Lorsque le développement de ces programmes a été conduit avec rigueur, l'analyste peut s'appuyer sur une abondante documentation et dispose des spécifications d'architecture ; dans le cas contraire, il est utile de reconstruire une telle documentation.

Il est donc intéressant d'analyser la possibilité de mener, dans le cadre de la *Méthode B*, un raffinement à rebours, qui partant d'un code permet de produire une spécification ; le recours à cette méthode permet de disposer d'un langage de spécification formel puissant, et surtout de permettre de mener une activité de preuve d'adéquation entre le code et la spécification obtenue.

Le présent rapport étudie la faisabilité et décrit le prototypage de l'outil *ABCBA* (*ABstraction of C in B Assistant*), qui permet de mener cette activité de raffinement à rebours. À partir d'un code *C*, une démarche est proposée, permettant d'aboutir à une spécification formelle, pour laquelle il est possible de mener une activité de preuve d'adéquation.

Ce rapport comporte également deux appendices :

- L'appendice 1 est un court rappel sur le *C*, principalement dédié à l'identification des lacunes de spécifications de ce langage (tel que défini par la norme ANSI). De telles lacunes, implicites ou explicites, se traduisent par des implémentations très variées, fortement dépendantes de l'architecture système considérée, et nuisent à la caractérisation du comportement des programmes et à leur portabilité.
- L'appendice 2 propose une synthèse de la théorie de la *Méthode B* ; cette synthèse se veut complète, et intègre de nombreux exemples, des analyses et des commentaires.

Table des matières

1	Introduction	6
1.1	Améliorer la confiance dans les logiciels	6
1.2	La prise en compte de l'existant	6
1.3	Évaluer le niveau de confiance	7
1.4	Objectif de l'étude	7
1.5	Avertissement	7
2	Rappels sur le B	8
2.1	Calcul des prédicats et théorie ensembliste	8
2.2	Substitutions généralisées	9
2.3	Structuration d'un développement en B	9
2.3.1	Machine abstraite	10
2.3.2	Raffinement	10
2.3.3	Implémentation	12
2.3.4	Traduction dans un langage impératif	12
2.3.5	Modularité des développements	12
2.3.6	Restrictions sur le langage des substitutions	13
2.4	Les preuves en B	14
2.5	Quelques remarques sur l'implémentation du B	14
3	Introduction à la démarche d'abstraction	15
3.1	Philosophie générale	15
3.2	Objet du processus d'abstraction	16
3.3	Limitations	16
3.4	Abstractions et spécification	17
3.5	Premières illustrations du processus d'abstraction	18
3.5.1	Division par deux	19
3.5.2	Multiplication par deux	19
3.5.3	Division par un entier	21

3.5.4	Appels de fonction et variables globales	22
3.6	Une première synthèse du processus d'abstraction	26
4	Modélisation arithmétique des opérations entières	27
4.1	Sensibilisation	27
4.2	Les types entiers du C	28
4.2.1	Représentation	28
4.2.2	Le cast	29
4.3	Modélisation arithmétique des cast	29
4.4	Modélisation arithmétique des opérations	30
4.5	Modules des entiers C en B	31
4.5.1	Le type <i>unsigned char</i>	31
4.5.2	Le type registre	32
4.5.3	Modularité	34
5	Modélisation des structures de contrôle	35
5.1	Structures <i>if</i>	35
5.2	Structures <i>switch</i>	37
5.3	Structures <i>while</i>	37
5.4	Structures <i>do</i>	39
5.5	Structures <i>for</i>	39
5.6	Instructions de rupture	39
5.7	Instruction <i>goto</i>	40
6	Modélisation des pointeurs et des structures	41
6.1	Fonction de valuation	41
6.2	Modélisation des constantes	42
6.3	Modélisation des structures	42
6.3.1	Fonctions de valuation multiples	42
6.3.2	Produit cartésien	43
6.3.3	Problème des initialisations partielles	44
6.4	Modélisation des unions	45
6.5	Modélisation des pointeurs	46
6.5.1	Modélisation de la mémoire	47
6.5.2	Modélisation des variables pointeurs	48
6.5.3	Opérateur de déréférencement	48
6.5.4	Arithmétique des pointeurs	49
6.5.5	Modélisation des pointeurs de pointeurs	50

6.5.6	Limitations de modélisation des pointeurs	53
6.6	Modélisation des tableaux	53
6.6.1	Tableaux à une dimension	53
6.6.2	Tableaux à plusieurs dimensions	53
6.7	Problématiques résiduelles et simplifications	54
6.7.1	Initialisation des variables	54
6.7.2	Spécification d'une opération modifiant une variable externe	55
6.7.3	Non-utilisation des opérations d'accès en lecture	56
6.7.4	Accès direct à la mémoire	57
6.7.5	Paramétrage des modèles par les types	57
6.7.6	Conclusion sur les modèles	57
7	Pre-cooking, traduction et analyse	58
8	Formalisation du pre-cooking	60
8.1	Renommage	60
8.2	Traitement des expressions	61
8.3	Réorganisation des structures <i>switch</i>	65
8.4	Modélisation des variables et des tableaux	65
9	Formalisation de la traduction	68
9.1	Problématique générale de la traduction	68
9.1.1	Génération de l'abstraction de référence	68
9.1.2	Génération d'une spécification initiale	69
9.1.3	Décomposition de la traduction	69
9.2	Etiquetage de l'AST	70
9.3	Fonctions de traduction	72
9.3.1	Déclarations de fonctions	72
9.3.2	Structures de contrôle	73
9.3.3	Instructions de rupture	75
9.3.4	Appels de fonctions et affectations	76
9.4	Organisation des modules	78
10	Synthèse sur l'outil <i>ABCBA</i>	79
10.1	Architecture	79
10.2	Choix de modélisation et limitations	79
10.3	Pre-cooking	80
10.4	Traduction	80

<i>TABLE DES MATIÈRES</i>	5
10.5 Edition	81
10.6 Preuve	81
11 Conclusion	82
A Modélisations des variables, pointeurs et structures	84
A.1 Entiers	85
A.2 Registres	86
A.3 Structures	87
A.4 Pointeurs sur type	88
A.5 Pointeurs sur pointeur	89

Chapitre 1

Introduction

1.1 Améliorer la confiance dans les logiciels

La question de la confiance dans les logiciels, qu'il s'agisse de sûreté ou de sécurité, reste plus que jamais d'actualité.

De nombreuses approches ont été proposées, depuis plusieurs décennies, pour améliorer le niveau d'assurance, qu'il s'agisse de démarches procédurales (insistant sur les méthodes de travail, le suivi du développement, une documentation rigoureuse et traçable), ou de nouvelles formes de programmation, parmi lesquelles on peut citer la programmation structurée, la programmation littérale, la programmation contractuelle, ou encore la programmation orientée objet...

L'une des approches les plus satisfaisantes, encore que délicate et coûteuse, est l'utilisation des méthodes formelles, i.e. d'un formalisme mathématique défini précisément, syntaxiquement et sémantiquement. Selon la méthode considérée, l'utilisation de ce formalisme peut se limiter à la rédaction de spécifications non ambiguës, ou être étendu à des activités de preuves voire à la programmation elle-même.

L'intérêt pratique d'une méthode formelle, cependant, est fortement dépendant de l'existence d'outils implémentant son formalisme, et éventuellement permettant d'automatiser certaines activités, telles que la production de preuves.

La *méthode B*, basée sur le calcul des prédicats et une théorie ensembliste simplifiée, veut offrir une solution complète, permettant d'écrire une spécification formelle, et par *raffinements* successifs de produire un programme prouvé conforme à cette spécification. Elle est implémentée dans différents outils automatisant les vérifications, et offrant une assistance dans les activités de preuves.

On peut admettre que le niveau de confiance en un logiciel développé dans le formalisme de la *méthode B* est optimal¹.

1.2 La prise en compte de l'existant

Il est intéressant de noter que les approches proposées pour améliorer le niveau de confiance dans les logiciels ne sont généralement applicables qu'à des développements nouveaux.

Il faut reconnaître cependant que, pour de nombreux systèmes, il ne peut être question de lancer des

¹Mais pas total, il dépend notamment de la confiance dans la correction et la complétion de la spécification formelle rédigée, dans les outils implémentant la méthode... ainsi que dans le compilateur, le système d'exploitation et le matériel.

développements complets ; l'existant croît constamment, et sa réutilisation ne peut être remise en cause. Les développements modernes travaillent à des niveaux d'abstraction toujours plus élevés, et utilisent fréquemment des services sous-jacents supposés corrects et conformes – pour ne citer que quelques exemples, on considèrera les services offerts par les systèmes d'exploitations, les moteurs de bases de données, les piles protocolaires, ou encore les environnements graphiques.

La question de la confiance dans des codes pré-existants, ou légués, reste donc posée.

1.3 Évaluer le niveau de confiance

La Direction centrale de la sécurité des systèmes d'information (DCSSI) a une mission de prestation de services, pour évaluer, connaître et faire connaître les vulnérabilités et les menaces, aider à les prévenir et contrer les attaques portées aux systèmes d'information.

Elle doit notamment être capable d'établir un niveau de confiance dans l'efficacité et la robustesse d'un produit de sécurité, en analysant les programmes qui entrent dans sa composition.

Lorsque le développement de ces programmes a été conduit avec rigueur, l'analyste peut s'appuyer sur une abondante documentation et dispose des spécifications d'architecture, des spécifications générales et des spécifications détaillées. Dans le cas contraire, il est utile de reconstruire une telle documentation.

1.4 Objectif de l'étude

Le présent document étudie la faisabilité d'un processus de reconstruction d'une documentation, à partir d'un programme écrit en *C* ; les mêmes principes sont a priori applicables à d'autres langages impératifs classiques.

Plus précisément, en se basant sur le formalisme de la *Méthode B*, il décrit comment par un processus de raffinement à rebours, nommé processus d'abstraction, il est possible d'extraire une spécification formelle, que l'on peut prouver conforme au code analysé.

Un outil d'assistance à ce processus d'abstraction, nommé *ABCBA* (pour *ABstraction of C in B Assistant*), est décrit, et quelques éléments de prototypage sont proposés.

1.5 Avertissement

Ce document se veut plus didactique que synthétique ; il décrit petit à petit le processus d'abstraction, en détaillant les arguments permettant de faire des choix de modélisation.

En conséquence, dans certains cas plusieurs modélisations pour les mêmes objets sont proposées successivement, en fonction de la prise en compte de besoins variés. La démarche retenue vise toutefois à construire une modélisation finale qui soit exhaustive, prenant en compte toutes les caractéristiques identifiées comme nécessaires.

Chapitre 2

Rappels sur le B

La *Méthode B* est présentée de manière plus détaillée en appendice 1. Ce chapitre a pour objet de mettre en exergue les points clés, en particulier ceux pouvant se traduire par des contraintes dans la réalisation de l'abstraction d'un code en une spécification.

2.1 Calcul des prédicats et théorie ensembliste

La théorie du B est basée sur le calcul des prédicats enrichi de constructions ensemblistes. La syntaxe associée est :

$Pred ::=$ $ Pred \wedge Pred$ $ Pred \Rightarrow Pred$ $ \neg Pred$ $ \forall Var \cdot Pred$ $ [Var := Expr]Pred$ $ Expr = Expr$ $ Expr \in Set$	$Expr ::=$ Var $[Var := Expr]Expr$ $Expr, Expr$ $choice(Set)$ Set
$Var ::=$ $Ident$ Var, Var	$Set ::=$ $Set \times Set$ $\mathbb{P}(Set)$ $\{Var \mid Pred\}$ BIG

On notera en particulier la définition de substitutions simples, de la forme $[Var := Expr]Pred$ ou $[Var := Expr]Expr$.

Pour chaque construction, une sémantique associée est définie sous la forme de règles d'inférence sur des prédicats. Cette sémantique est restrictive vis-à-vis de la syntaxe, au sens où l'ensemble des termes ayant du sens est un sous-ensemble strict des termes syntaxiquement bien construits.

Exemple 2.1 (Syntaxe et sémantique des ensembles définis en compréhension)

<i>Syntaxe</i>	<i>Sémantique</i>
$\{Var \mid Pred\}$	$E \in \{x \mid x \in S \wedge P\} \Leftrightarrow (E \in S \wedge [x := E]P) \text{ si } x \setminus S$

Cette sémantique est plus restrictive que la syntaxe, puisqu'elle ne donne du sens qu'à une forme précise des ensembles admis pas la syntaxe; notons aussi qu'il s'agit bien d'une sémantique de prédicats, puisqu'elle ne décrit pas ce qu'est un ensemble défini en compréhension, mais comment on peut prouver l'appartenance à un tel ensemble.

Une notion de typage est d'ailleurs introduite, permettant d'éliminer rapidement les termes n'ayant

pas de signification¹.

À partir de ces constructions élémentaires, la *Méthode B* fournit des définitions permettant des constructions de plus en plus complexes, telles que l'union, l'intersection, les relations, les fonctions, etc. Cette théorie est également suffisante pour définir les constructions inductives par des points fixes ; la *Méthode B* décrit notamment une construction de l'ensemble des sous-ensembles finis d'un ensemble, de \mathbb{N} , des séquences finies, des arbres finis, des relations ou des fonctions itérées.

2.2 Substitutions généralisées

La syntaxe des prédicats est étendue, en introduisant une nouvelle catégorie syntaxique, les substitutions généralisées :

$$\begin{array}{l}
 \textit{Pred} ::= \dots \\
 \quad | \quad [\textit{Subst}]\textit{Pred} \\
 \textit{Expr} ::= \dots \\
 \quad | \quad [\textit{Subst}]\textit{Expr} \\
 \textit{Subst} ::= \textit{Var} := \textit{Expr} \\
 \quad | \quad \textit{skip} \\
 \quad | \quad \textit{Pred}|\textit{Subst} \\
 \quad | \quad \textit{Subst} \parallel \textit{Subst} \\
 \quad | \quad \textit{Pred} \implies \textit{Subst} \\
 \quad | \quad @\textit{Var} \cdot \textit{Subst} \\
 \quad | \quad \textit{Subst}||\textit{Subst} \\
 \quad | \quad [\textit{Var} := \textit{Expr}]\textit{Subst} \\
 \quad | \quad \textit{Subst}; \textit{Subst} \\
 \quad | \quad \textit{WHILE } \textit{Pred} \textit{ DO } \textit{Subst} \textit{ INVARIANT } \textit{Pred} \textit{ VARIANT } \textit{Expr} \textit{ END}
 \end{array}$$

La sémantique des substitutions est définie sous la forme de transformateurs de prédicats². Les différentes constructions introduisent les notions de pré-condition, de garde, de choix non déterministe, de composition parallèle, de composition séquentielle, de boucle.

Exemple 2.2 (Sémantique de la substitution gardée)

<i>Sémantique</i>	
$([P \implies S]R)$	$\Leftrightarrow (P \Rightarrow [S]R)$

On dira que la substitution S réalise le prédicat P si le prédicat $[S]P$ est prouvable.

À partir de ces constructions élémentaires, la *Méthode B* fournit des définitions permettant des constructions rappelant celles utilisées en programmation impérative classique, telles que IF THEN ELSE, CASE, etc.

2.3 Structuration d'un développement en B

Un développement en B est le plus souvent constitué d'un ensemble de machines abstraites qui sont raffinées jusqu'à obtention d'une implémentation, traduisible en un programme dans un langage impératif tel que le C .

On appelle *composant* une machine abstraite, un raffinement ou une implémentation ; on appelle *module* une chaîne de composants liés deux à deux par raffinement.

¹Attention, c'est une présentation extrêmement simpliste de la notion de type-checking en B .

²Cette sémantique est subtile, et il est absolument fondamental de ne pas avoir une vision opérative des substitutions, qui conduirait à des raisonnements erronés.

2.3.1 Machine abstraite

Un développement en B commence par la définition d'une machine abstraite, qui correspond à la spécification du logiciel que l'on souhaite développer.

Une machine abstraite est composée de constantes et de variables, décrivant un état global, de propriétés sur ces constantes et variables (en particulier un invariant), et enfin d'opérations permettant de lire ou de modifier l'état global. On peut également paramétrer une machine abstraite.

La syntaxe simplifiée d'une machine abstraite est décrite ci-dessous :

$$\begin{aligned}
 \textit{Machine} & ::= \text{MACHINE } [\textit{Ident} \mid \textit{Ident}([\textit{Ident}]^+)] \\
 & \quad \text{CONSTRAINTS } \textit{Pred} \\
 & \quad \text{CONSTANTS } [\textit{Ident}]^+ \\
 & \quad \text{PROPERTIES } \textit{Pred} \\
 & \quad \text{VARIABLES } [\textit{Ident}]^+ \\
 & \quad \text{INVARIANT } \textit{Pred} \\
 & \quad \text{INITIALISATION } \textit{Subst} \\
 & \quad \text{OPERATIONS } [\textit{Oper}]^+ \\
 & \quad \text{END} \\
 \textit{Oper} & ::= \textit{Ident} \hat{=} \textit{Subst} \\
 & \quad \mid \textit{Ident}([\textit{Ident}]^+) \hat{=} \textit{Subst} \\
 & \quad \mid \textit{Ident} \leftarrow \textit{Ident} \hat{=} \textit{Subst} \\
 & \quad \mid \textit{Ident} \leftarrow \textit{Ident}([\textit{Ident}]^+) \hat{=} \textit{Subst}
 \end{aligned}$$

Comme on le voit, les opérations sont en fait des substitutions, et un programme faisant appel à une opération peut être décrit par un transformateur de prédicats.

À une telle machine sont associées des obligations de preuves, i.e. des prédicats qui doivent être démontrés afin d'assurer que la machine abstraite est valide :

- Établissement de l'invariant, la vérification de la validité de l'invariant vis à vis de l'initialisation des variables.
- Préservation de l'invariant, la vérification de la validité de l'invariant après l'appel d'une opération.

Notons que formellement, les obligations de preuves ont toujours la forme d'une implication. D'autres obligations de preuves sont différées en B : les preuves d'existence d'une instantiation valide pour les paramètres, les constantes et les variables (valide signifiant ici respectant les contraintes exprimées).

2.3.2 Raffinement

Une machine abstraite doit être raffinée ; le raffinement correspond à une représentation du logiciel dérivée de la spécification mais plus déterministe, plus concrète. Il est possible d'avoir plusieurs raffinements successifs (le raffinement n raffine le raffinement $n - 1$).

Le raffinement est tout d'abord défini pour les substitutions (i.e. pour les opérations). Si S_R est un raffinement de S_A , ce qui se note $S_A \sqsubseteq S_R$, alors tout prédicat réalisé par S_A est réalisé par S_R ³ :

$$[S_A]P \Rightarrow [S_R]P$$

Exemple 2.3 (Réduction du non-déterminisme) $(x := a \parallel x := b) \sqsubseteq (x := a)$

Exemple 2.4 (Affaiblissement de la pré-condition) $(x > 5 \mid x := x - 1) \sqsubseteq (x > 1 \mid x := x - 1)$

³Cette définition d'ordre supérieur est ramenée en B à une définition de premier ordre par une caractérisation sous forme de transformateurs d'ensembles.

\sqsubseteq est réflexive, anti-symétrique et transitive, et définit donc un ordre partiel.

Le raffinement est ensuite étendu aux machines. Si M_R est un raffinement de M_A , ce qui se note $M_A \sqsubseteq M_R$, alors M_R doit offrir les mêmes opérations que M_A (les mêmes signatures) ; une opération de M_A et de M_R peut cependant être décrite sous la forme de deux substitutions différentes S_A et S_R .

Formellement, en notant T_A le type de la variable x_A de la machine M_A et T_R le type de la variable x_R de la machine M_R , M_R raffine M_A si pour chaque opération on peut vérifier :

$$\textcircled{x}_A \cdot (x_A \in T_A \implies S_A) \sqsubseteq \textcircled{x}_R \cdot (x_R \in T_R \implies S_R)$$

Moralement, si M_R raffine M_A , alors toute substitution externe faisant appel aux opérations de M_A et de M_R donnera le même résultat ; les deux machines ne peuvent pas être distinguées (dans leur domaine d'emploi).

En particulier, il convient de noter que l'état de la machine peut être décrit de manière totalement différente dans un raffinement associé ; un invariant de collage permet alors d'exprimer le lien entre les deux représentations.

Les obligations de preuves associées au raffinement sont les suivantes :

- L'initialisation de M_R raffine l'initialisation de M_A .
- Les opérations de M_R raffinent les opérations de M_A .

Exemple 2.5 (Le maximier) *La puissance de cette définition du raffinement est parfaitement illustrée par la machine maximier qui offre deux opérations, l'ajout d'un entier naturel non nul à un ensemble (initialement vide) et l'extraction du maximum de cet ensemble :*

```
MACHINE maximier0
  VARIABLES S
  INVARIANT S ∈ ℱ(ℕ1)
  INITIALISATION S := ∅
  OPERATIONS enter(n) ≐ n ∈ ℕ1 | S := S ∪ {n}
              m ← maximum ≐ S ≠ ∅ | m := max(S)
END
```

Cette machine peut être raffinée comme suit :

```
REFINEMENT maximier1
  REFINES maximier0
  VARIABLES z
  INVARIANT z = max(S ∪ {0})
  INITIALISATION z := 0
  OPERATIONS enter(n) ≐ n ∈ ℕ | z := max({z, n})
              m ← maximum ≐ z ≠ 0 | m := z
END
```

Alors que la spécification mémorise un sous-ensemble fini de \mathbb{N}_1 , le raffinement ne mémorise que la valeur maximale introduite – il y a donc formellement une perte d'information. Cependant, vu de l'extérieur, il n'est pas possible de distinguer les deux machines, puisque les règles de visibilité ne permettent que des appels aux deux opérations proposées.

Il faut noter qu'un raffinement a une syntaxe proche de celle d'une machine abstraite ; en pratique cependant, un raffinement est utilisé pour produire, en conjonction avec la machine abstraite qu'il raffine, une nouvelle machine abstraite.

Cela signifie en particulier qu'il n'est pas possible d'écrire un raffinement sans avoir une machine abstraite.

2.3.3 Implémentation

Le dernier raffinement, dans la *Méthode B*, est nommé implémentation.

À ce stade, ne sont autorisées que les constructions du langage qui sont implémentables (dans un langage impératif classique), en particulier les substitutions non déterministes ou les substitutions parallèles sont proscrites, et les variables et constantes doivent être représentables.

Des machines particulières, appelées machines de bases, peuvent être importées dans une implémentation. Ces machines offrent des services associés par exemple à la manipulation de variables scalaires, de fichiers, etc.

C'est au plus tard au stade de l'implémentation que les obligations de preuves différées doivent être déchargées, telles que par exemple l'existence d'instances valides pour les objets manipulés.

L'implémentation permet aussi d'utiliser la récursion dans les opérations, avec la syntaxe suivante :

$$\begin{array}{l}
 \text{Oper} ::= \dots \\
 \quad | \text{ Ident} \hat{=} \text{ REC Expr THEN Subst END} \\
 \quad | \text{ Ident}([\text{Ident}]^+) \hat{=} \text{ REC Expr THEN Subst END} \\
 \quad | \text{ Ident} \leftarrow \text{ Ident} \hat{=} \text{ REC Expr THEN Subst END} \\
 \quad | \text{ Ident} \leftarrow \text{ Ident}([\text{Ident}]^+) \hat{=} \text{ REC Expr THEN Subst END}
 \end{array}$$

Notons qu'il ne s'agit pas d'une nouvelle forme de substitution, mais d'une nouvelle forme d'opération. L'expression qui suit le mot clé REC désigne un variant, et la substitution *Subst* est censée contenir un appel à l'opération elle-même.

Si une opération $O_I \hat{=} \text{ REC } V \text{ THEN } T \text{ END}$ raffine l'opération $O_A \hat{=} \text{ PRE } P \text{ THEN } S \text{ END}$, les obligations de preuves associées sont les suivantes :

- Le variant décroît à chaque appel (et son type est bien fondé).
- L'opération obtenue en remplaçant les appels (récursifs) à O_I par des appels (non récursifs) à l'opération abstraite O_A est un raffinement de l'opération abstraite O_A .

2.3.4 Traduction dans un langage impératif

La dernière étape d'un développement standard en *B* est la traduction de l'implémentation en un code dans un langage impératif. Cette fonctionnalité ne relève pas de la *Méthode B* en tant que telle, mais des outils utilisés ; les langages proposés sont pour l'instant C, Java, Ada⁴.

De manière simplifiée, la traduction consiste soit à traduire dans un langage donné les objets et opérations (qui n'utilisent plus à ce stade que des constructions standards et implémentables), soit à utiliser le code associé à une machine de base. Une machine de base est en effet une spécification à laquelle est directement associée un code (qui n'a pas été raffiné ou prouvé).

2.3.5 Modularité des développements

Différents mécanismes de composition de composants sont proposés par la *Méthode B* ; en pratique ces mécanismes se ramènent toujours à un mécanisme d'inclusion ou d'utilisation d'une machine abstraite par un composant. Ces mécanismes de composition se traduisent pas un graphe de dépendance, qui ne doit contenir aucun cycle.

La clause IMPORTS importe une machine abstraite *A* dans une implémentation *I* ; elle permet de créer une instance de *A* (en particulier les éventuels paramètres doivent être instanciés) afin d'utiliser ses données et ses opérations pour l'implémentation des données et des opérations de *I*.

⁴Le *B* est naturellement associé à la production de code dans un langage impératif.

La clause **SEES** rend visible une instance d'une machine abstraite A dans un composant C ; cette instance existe car A doit être importée (et éventuellement incluse) quelque part dans le projet. Elle permet à C de lire les ensembles, constantes, variables de A et d'appeler les opérations de A ne modifiant pas l'état (les variables). Formellement, la machine A doit être importée quelque part dans le projet ; de plus si A est vue par C et si R raffine de C , alors il est obligatoire que R voit également A .

La clause **INCLUDES** inclut une machine abstraite A dans un composant C ; elle permet de créer une instance de A (en particulier les éventuels paramètres doivent être instanciés), de lire ses ensembles, constantes, variables et d'appeler ses opérations (cependant, il est interdit dans une opération de C de faire appel en parallèle à deux opérations actives d'une même machine A , actives signifiant ici modifiant l'état). Si C inclut A et que R raffine C , alors R doit inclure A .

Remarque 2.1 (Prise en compte du INCLUDES) *En pratique, l'outil Click'n'Prove/B4Free v2 introduit des limitations beaucoup plus drastiques. L'absence d'appels parallèles d'opérations actives est vérifiée au moment du type-checking, mais lors de la génération des obligations de preuve ce sont les appels parallèles à deux opérations quelconques d'une même machine incluse qui sont apparemment interdits.*

La clause **INCLUDES** est transitive, si une machine M_2 inclut une machine M_1 , et qu'une machine M_3 inclut une machine M_2 , alors les ensembles, constantes et variables de M_1 peuvent être lus par M_3 ; par contre les opérations de M_1 ne peuvent être appelées dans M_3 , à moins qu'elles n'aient été promues dans M_2 . Notons aussi que dans ce cas de figure, de manière évidente M_3 ne peut inclure M_1 .

La clause **PROMOTES** transforment des opérations d'une machine abstraite A incluse ou importée en opérations du composant C commandant l'inclusion ou l'import⁵ ; la clause **EXTENDS** correspond à une clause **INCLUDES** ou **IMPORTS** suivie de la promotion de l'ensemble des opérations par une clause **PROMOTES**.

La clause **USES** permet à des machines abstraites A_1, A_2 de partager les données d'une instance de la machine utilisée A_u ; un composant C doit ensuite inclure les machines A_1, A_2 et A_u (l'instance considérée pour l'utilisation), et les machines abstraites A_1 et A_2 ne peuvent pas être raffinées. Les ensembles, constantes et variables de A_u peuvent être lues par A_1, A_2 , mais les opérations de A_u ne peuvent être appelées.

2.3.6 Restrictions sur le langage des substitutions

Les machines abstraites, raffinements et implémentations utilisent un sous-ensemble du langage des substitutions généralisées.

On notera en particulier les restrictions suivantes :

- Les machines abstraites ne peuvent utiliser les substitutions avec variables locales (**VAR**), les substitutions séquentielles (**;**), les opérations récursives (**REC**).
- Les raffinements ne peuvent utiliser les opérations récursives (**REC**).
- Les implémentations ne peuvent utiliser les substitutions avec pré-condition (**PRE**), les substitutions non-déterministes (**CHOICE, ANY, ::**), les substitutions parallèles (**||**), les substitutions avec définition locale (**LET**).

⁵Il s'agit d'une simplification, puisque les objets manipulés et l'invariant diffèrent.

2.4 Les preuves en B

Les preuves en B correspondent à des obligations de preuves générées automatiquement ; il est possible d'ajouter à ces obligations de preuves des assertions. Les assertions permettent notamment d'expliciter des étapes intermédiaires de preuves, des lemmes, qui pourront être exploités pour décharger les obligations de preuves.

Notons qu'on ne peut pas directement faire de preuves sur les opérations ; on peut indirectement cependant décrire des propriétés intéressantes sur les opérations, par exemple en utilisant des représentations par fonctions constantes, ou en explicitant dans la substitution correspondant à l'opération des pré et post-conditions surnuméraires.

2.5 Quelques remarques sur l'implémentation du B

La *Méthode B* est une méthode formelle décrite dans le *B-Book* ([Abr96]).

Différentes implémentations de cette méthode sont disponibles ; l'implémentation utilisée dans le cadre de ce stage est *B4Free v2* associé à *Click'n'Prove v2*⁶.

Il existe quelques différences entre la théorie et la pratique (l'implémentation) ; ces différences peuvent résulter de simplifications, d'optimisations, de complétions ou de corrections. On notera en particulier, pour ce qui concerne le couple *B4Free/Click'n'Prove v2* :

- Les entiers relatifs deviennent des objets natifs, l'ensemble **BIG** disparaît, au profit de \mathbb{N} , \mathbb{Z} , mais aussi de **NAT** et de **INT**, désignant respectivement les versions implémentables des naturels et des entiers.
- Le type-checking implémenté ne correspond pas exactement à celui du B .
- Le type-checking est effectué préalablement aux activités de preuve ; il est possible pendant la preuve de manipuler des termes qui ne type-checkent pas.
- Certains axiomes ont été simplifiés et remplacés par des schémas, modifiant la sémantique formelle.
- Les prédicats de typages associés dans le prouveur aux ensembles donnés sont différents de ceux proposés par la théorie.
- Les opérations récursives ne sont pas implémentées.
- Les substitutions **WHILE** ne sont autorisées que dans les implémentations.
- Les restrictions associées à l'inclusion de machines sont plus fortes.
- Les définitions conditionnelles sont implémentées dans le prouveur sous une forme simplifiée, dans laquelle les conditions ont disparu ; l'utilisation de ces définitions génère formellement de nouvelles obligations de preuves, sous la forme de delta-lemmes, qui sont prouvés en fin de développement.

⁶ *B4Free v1.1* et *Click'n'Prove v1.2* ont été utilisés pour les premiers mois du stage.

Chapitre 3

Introduction à la démarche d'abstraction

3.1 Philosophie générale

Le processus d'abstraction doit permettre la production, à partir d'un code C , d'une spécification formelle correspondante écrite dans le formalisme du B .

Clairement, un tel processus ne peut être automatisé; un programme est un ensemble complexe d'opérations auquel on peut associer un nombre illimité de spécifications, dont très peu sont réellement pertinentes, au sens où elles décrivent à haut niveau la sémantique enfouie pendant le processus de développement. L'objectif visé peut d'ailleurs être d'extraire non pas une spécification fonctionnelle, mais des propriétés permettant par exemple d'identifier les conditions de bon fonctionnement (absence de plantages, de débordements) – ou plus simplement la terminaison, ce qui prouve bien l'indécidabilité du problème plus général considéré¹.

Cependant, il reste possible d'assister un utilisateur dans un tel processus d'extraction de spécifications ou de propriétés :

- En automatisant certaines étapes de l'abstraction.
- En permettant de prouver formellement que la spécification produite correspond au code analysé, ou tout au moins en fournissant un niveau d'assurance satisfaisant.

Le langage C n'a pas été défini avec une sémantique formelle²; une preuve de correspondance entre un code C est une spécification apparaît donc impossible à ce stade.

Cependant, il est tout à fait envisageable d'associer à un code C une implémentation ou un raffinement en B . Il s'agit ici d'une étape de traduction, que l'on veut simple³ (puisque non prouvée), et autant que possible automatique.

Une implémentation ou un raffinement dispose d'une sémantique formelle; il est donc envisageable de rédiger, avec l'assistance d'un outil, des abstractions successives, jusqu'à aboutir à une spécification sous la forme d'une machine abstraite manipulant des concepts de haut niveau.

¹En fait, l'indécidabilité n'est pas directement liée au processus d'abstraction mais à l'activité de preuve nécessaire à assurer l'adéquation.

²De nombreux travaux ont proposé une telle formalisation, mais ils traitent en général un sous-ensemble du langage.

³On verra cependant que le C comporte de nombreux pièges.

3.2 Objet du processus d'abstraction

On considère que le processus d'abstraction vise toujours à la spécification ou à l'extraction de propriétés d'une fonction⁴ dont le code est écrit en langage C .

Rappelons qu'un programme en C correspond à la fonction `main` qui peut accepter des paramètres en entrée, et qui peut retourner une valeur (de type `int`) généralement interprétée comme un code d'erreur⁵.

Le code des fonctions analysées est supposé disponible; lorsque des fonctions sont utilisées sans connaissance du source, par exemple lors de l'utilisation de bibliothèques, les prototypes et les spécifications informelles (spécifications, manuel utilisateur, voire commentaires) sont supposés connus.

3.3 Limitations

Dans le cadre de cette étude, les limitations suivantes ont été acceptées d'emblée :

Modélisation 1 *Seuls les programmes séquentiels sont considérés.*

L'approche retenue dans le cadre de cette étude ne permet pas d'aborder la programmation par interruptions (ou événementielle) ou concurrente.

Modélisation 2 *Seuls les programmes traitant des expressions entières sont considérés.*

Il s'agit d'une simplification qu'il est possible de contourner ultérieurement sans remise en cause des résultats acquis. Rappelons qu'il n'y a pas de traitement des réels, ou des flottants, dans la théorie de la *Méthode B*, ni d'implémentation, qu'il s'agisse d'objets natifs ou définis dans une bibliothèque, dans les outils *Click'n'Prove/B4Free v2*.

Modélisation 3 *Seuls les programmes non récursifs sont considérés.*

Formellement, la théorie de la *Méthode B* aborde la question de l'induction et de la récursivité; en dehors des objets abstraits que l'on peut définir inductivement, tels que les ensembles et fonctions, il est notamment possible dans une implémentation (et seulement dans une implémentation) de définir une opération récursive.

En pratique cependant, *Click'n'Prove/B4Free v2* n'implémente pas cette fonctionnalité, et il ne serait pas possible de prouver la conformité d'une telle implémentation avec un raffinement proposant une représentation non récursive. La possibilité de traduire directement une fonction récursive sous une forme non récursive a été envisagée, mais abandonnée pour cette étude, car elle constitue une transformation jugée trop délicate pour être menée en dehors de tout processus de preuve.

Modélisation 4 *Seules les fonctions admettant un nombre fixe de paramètres sont considérées.*

Le C offre une possibilité de définir, en utilisant une ellipse, des fonctions admettant un nombre variable de paramètres. Bien qu'il soit possible de définir en B des opérations qui acceptent en paramètre des objets structurés (tableaux, ensembles, séquences) et donc de représenter ce type de fonctions C , cette étude ne considèrera cette possibilité.

⁴On parle ici de fonction au sens du langage C , i.e. formellement un objet non fonctionnel puisque pouvant dépendre de paramètres implicites.

⁵Dans le cas général, cette fonction ne nous intéressera donc pas en tant que telle, ne constituant qu'une enveloppe commode des traitements effectués – son analyse permettra principalement d'identifier les cas d'erreurs prévus par le programmeur.

Modélisation 5 *Les programmes analysés sont supposés compilables.*

On ne souhaite pas, dans le processus d'abstraction, avoir à vérifier que le code C est correct ; à titre d'exemple, un code C déclarant une constante et la modifiant pourra être abstrait, analysé et spécifié sans difficulté, l'abstraction ne prenant pas en compte le spécificateur `const`.

3.4 Abstractions et spécification

Dans le processus d'abstraction, il apparaît possible de traduire automatiquement un code C en une implémentation ou un raffinement en B ; ces deux représentations sont en effets très proches.

Les principaux critères de choix sont les suivants :

- Une implémentation B permet de définir des opérations récursives et ainsi d'abstraire un code C récursif (en pratique cependant les opérations récursives ne sont pas prises en compte dans l'outil *Click'n'Prove/B4Free v2.0*).
- Une implémentation B permet d'utiliser les substitutions `WHILE` (ceci correspond à une limitation de l'outil *Click'n'Prove/B4Free v2.0* et non de la *Méthode B*).
- Une implémentation B permet d'utiliser le mécanisme d'import, potentiellement plus intéressant que le mécanisme d'inclusion offert dans un raffinement B .
- Une implémentation B impose les preuves d'existences d'instances valides.
- Un raffinement B permet de manipuler les opérations arithmétiques formelles, alors qu'une implémentation ne permet de traiter que d'objets implémentables.
- Un raffinement B permet de définir des objets abstraits, tels que par exemple la fonction d'allocation de la mémoire permettant la modélisation des pointeurs.
- Un raffinement B permet d'utiliser des substitutions non déterministes, et en particulier de définir des miracles, des substitutions raffinant toute substitution.

Nous choisissons donc d'utiliser comme représentation de la traduction d'un code C un raffinement. En raison des limitations de l'outil *Click'n'Prove/B4Free v2.0* utilisé dans le cadre de l'étude, en présence de boucles dans le code C traduites par une substitution `WHILE`, une partie du processus de preuve devra faire l'objet d'un traitement manuel.

Le raffinement produit automatiquement par traduction à partir du code C est nommé *abstraction de référence*, et identifié par l'indice $a0$. Il constitue par la suite le seul lien avec le code C analysé et ne peut jamais être modifié.

Un tel raffinement, non inclus dans un module, n'a guère d'intérêt en B^6 ; il n'est en particulier pas possible de mener une activité de preuve.

Il est donc nécessaire de produire, toujours automatiquement et à partir du code, une machine abstraite complétant l'abstraction de référence.

Les caractéristiques propres des machines abstraites ne permettent pas d'envisager une traduction automatique vers un niveau d'abstraction élevé – on notera par exemple que le C est une séquence d'opérations élémentaires, alors que les machines abstraites du B ne permettent pas d'utiliser l'opérateur de séquentialisation des substitutions.

Pour autant, on souhaite que la machine abstraite produite soit telle que l'abstraction de référence en constitue un raffinement correct. L'idée est donc de définir cette machine abstraite par l'association des signatures des objets considérés.

Exemple 3.1 (Quelques exemples de machines génériques) *Les prototypes de fonctions suivants sont transcriposables directement sous la forme d'opérations pouvant être intégrées à une ma-*

⁶Un raffinement en B n'est pas défini de manière isolée mais permet, en association avec une machine abstraite, de produire une nouvelle machine abstraite.

*chine abstraite quelconque*⁷ :

```

void function1();      →  function1()=BEGIN skip END
void function2(int x); →  function2(x)=PRE x:INT THEN skip END
int function3();      →  out<--function3()=BEGIN out::INT END
int function4(int x); →  out<--function4(x)=PRE x:INT THEN out::INT END

```

Cette machine abstraite est nommée *spécification* et est identifiée par l'indice *s*. Elle peut bien sûr être modifiée par l'utilisateur, puisque sa complétion constitue le but ultime du processus d'abstraction.

On nommera *preuve d'adéquation* la preuve que l'abstraction de référence est un raffinement correct de la spécification. Une bonne stratégie de preuve consiste à décomposer le problème en procédant par des raffinements successifs ; l'utilisateur est donc autorisé à ajouter des raffinements, nommés *abstractions intermédiaires* et identifiés par l'indice *a* suivi d'un nombre (plus ce nombre est grand, plus l'abstraction est importante et proche de la spécification), lui permettant de mener progressivement l'abstraction, et de décomposer la preuve d'adéquation.

Un raffinement intermédiaire sera toujours introduit entre le dernier raffinement (celui dont l'indice est le plus élevé) et la spécification, par copie du dernier raffinement ou de la spécification.

Notons qu'il est possible de conserver indéfiniment la spécification générique produite par l'outil, et d'enrichir les seules abstractions. Cependant, les propriétés d'un raffinement en *B* ne sont exploitées que dans les raffinements ultérieurs ; seules les propriétés associées à la spécification peuvent être exportées vers d'autres spécifications ou d'autres abstractions.

Il faudra donc, dans le processus d'abstraction, s'assurer que les principales propriétés soient remontées jusqu'à la spécification ; une telle remontée pourra d'ailleurs fréquemment être automatisée.

Rappelons enfin que certaines obligations de preuves, en *B*, peuvent être différées à l'implémentation ; c'est par exemple le cas de la preuve d'existence d'instances valides pour les objets définis. En l'absence d'implémentation, des assertions devront être générées automatiquement pour imposer le déchargement immédiat des obligations de preuves différées.

La démarche d'abstraction proposée peut donc se résumer de la manière suivante :

- À partir du code *C*, un traducteur produit l'abstraction de référence et une première version de la spécification. L'abstraction de référence reflète le code, complété par des pré-conditions et des assertions correspondant aux obligations de preuves normalement différées à l'implémentation ; la spécification se limite à la description des signatures.
- L'utilisateur est invité à produire la première preuve d'adéquation entre l'abstraction de référence et la spécification générique.
- L'utilisateur peut ensuite créer une abstraction intermédiaire, modifier une abstraction intermédiaire ou modifier la spécification ; de telles modifications nécessitent de reconduire la preuve d'adéquation.
- Le processus se termine quand l'utilisateur s'estime satisfait des propriétés et spécifications qu'il a pu extraire et promouvoir jusqu'à la spécification, et qu'il a fourni la preuve d'adéquation correspondante.

3.5 Premières illustrations du processus d'abstraction

Le processus d'abstraction considéré dans la section précédente est ici illustré à travers quelques exemples.

Ces exemples, ainsi que leur traitement, sont volontairement très simples, ayant pour vocation de jeter un premier éclairage sur la démarche et d'identifier de grandes caractéristiques fonctionnelles

⁷Sous hypothèse de non-dépendance vis à vis des variables globales représentées dans la machine.

que l'on peut attendre de l'outil *ABCBA* et non de formaliser le processus d'abstraction.

Notons que le processus d'abstraction est ici mené directement sur les machines abstraites et les raffinements *B*, en acceptant les contraintes associées. Il apparaît cependant évident qu'une interface dédiée peut permettre de masquer les éléments non pertinents de ce formalisme. Ainsi, les noms des composants sont générés automatiquement et ne peuvent pas être modifiés par l'utilisateur ; de même les relations de raffinement entre composants, les inclusions, sont gérées par l'outil.

3.5.1 Division par deux

On considère la fonction *C* suivante :

```
int div2(int x){return x/2;}
```

Il est possible de traduire automatiquement cette fonction *C* sous la forme d'un raffinement, et de lui associer une machine générique. L'opération de traduction se veut aussi simple que possible, sans restructuration du code, puisqu'elle échappe à la preuve. En première approximation, on obtient donc le module suivant :

```
MACHINE div2_s
OPERATIONS out<--div2(x)=PRE x:INT THEN out::INT END
END

REFINEMENT div2_a0
REFINES div2_s
OPERATIONS out<--div2(x)=PRE x:INT THEN out:=x/2 END
END
```

La présence d'une machine abstraite (associée à un raffinement) permet d'entamer le processus de preuve, ici trivial.

Vu la simplicité du code considéré, il n'est pas utile d'insérer un nouveau niveau d'abstraction ; l'utilisateur peut directement modifier la spécification pour identifier la propriété qu'il juge pertinente, ici le fait qu'il s'agisse d'une division entière par défaut⁸ :

```
MACHINE div2_s
OPERATIONS
  out<--div2(x)=PRE x:INT THEN
    ANY y WHERE y:INT & (2*y=x or 2*y+1=x) THEN out:=y END
  END
END
```

3.5.2 Multiplication par deux

On considère la fonction *C* suivante :

```
unsigned char mul2(unsigned char x){return x*2;}
```

En première approximation, la génération automatique donnerait le module suivant :

```
MACHINE mul2_s
OPERATIONS out<--mul2(x)=PRE x:0..255 THEN out::0..255 END
END
```

⁸Pour mémoire, il s'agit d'un choix d'implémentation laissé libre par la définition du *C*.

```

REFINEMENT mul2_a0
REFINES mul2_s
OPERATIONS out<--mul2(x)=PRE x:0..255 THEN out:=x*2 END
END

```

En tentant de décharger les obligations de preuves associées, nous observons rapidement cependant que cette modélisation serait fautive, malgré la simplicité apparente du code considéré. Il y a en effet une possibilité de dépassement par la multiplication, ce qui signifie ici que l'opération `mul2_a0.mul2` ne raffine pas l'opération `mul2_s.mul2`.

Nous sommes ici confronté à la présence, en *C*, d'opérations implicites de conversions de type, ou `cast`; la modélisation des opérations étant formelle en *B*, il est nécessaire d'explicitier ces `casts`.

Une méthode simple consiste à fournir avec l'outil *ABCBA* des bibliothèques de machines abstraites associées aux types du *C*; la constitution de telles bibliothèques est décrite dans le chapitre 4⁹.

Nous utilisons dans notre exemple la spécification `uchar_s`, qui définit notamment un ensemble `UCHAR = 0..255` et une opération `uchar_cast` correspondant au `cast` implicite (dans sa version stricte) d'une valeur entière quelconque en un `unsigned char` du *C*.

Cette machine est associée à une abstraction de référence `uchar_a0`; l'idée ici est que l'utilisateur peut vouloir modifier `uchar_s`, et que cette abstraction de référence, non modifiable, permet de préserver la validité des preuves d'adéquation.

Notre génération automatique doit donc produire le module suivant :

```

MACHINE mul2_s
INCLUDES uchar_s
OPERATIONS
  out<--mul2(x)=PRE x:UCHAR THEN out::UCHAR END
END

REFINEMENT mul2_a0
REFINES mul2_s
INCLUDES uchar_s
OPERATIONS
  out<--mul2(x)=PRE x:UCHAR THEN
    out<--uchar_cast(x*2)
  END
END

```

Les obligations de preuves associée à ce module sont automatiquement déchargées.

L'utilisateur peut enfin modifier la machine abstraite pour mettre en évidence certaines propriétés. Dans le cas présent, on suppose qu'il sait que la fonction ne sera utilisée qu'avec un paramètre dont la valeur est dans `0..100`; il identifie donc la pré-condition correspondante¹⁰, et peut aussi

⁹En particulier, différentes philosophies fondamentalement différentes de modélisation du `cast` sont présentées, plus ou moins strictes.

¹⁰Un outil puissant, puisqu'il faudra à toute utilisation de cette opération prouver le respect de cette condition.

expliciter une post-condition rappelant le typage initial :

```

MACHINE mul2_s
INCLUDES uchar_s
OPERATIONS
  out<--mul2(x)=PRE x:0..100 THEN
    ANY result WHERE result:UCHAR & result=x*2 THEN
      out:=result
    END
  END
END

```

Notons que l'on conserve bien la relation de raffinement entre `mul2_s` telle que modifiée et `mul2_a0` – il y a en particulier affaiblissement de la pré-condition (i.e. $x \in 0..100 \Rightarrow x \in 0..255$). Notons aussi que la modification de la machine nécessite de reprendre l'ensemble des preuves déjà effectuées¹¹.

3.5.3 Division par un entier

On considère la fonction C suivante :

```
char div(char x,char y){return x/(3+y/2);}
```

La génération automatique doit produire le module suivant :

```

MACHINE div_s
INCLUDES schar_s
OPERATIONS
  out<--div(x,y)=PRE x:SCHAR & y:SCHAR THEN out::SCHAR END
END

REFINEMENT div_a0
REFINES div_s
INCLUDES schar_s
OPERATIONS
  out<---div(x,y)=PRE x:SCHAR & y:SCHAR THEN
    PRE not(3+y/2=0) THEN
      out<--schar_cast(x/(3+y/2))
    END
  END
END

```

Comme on le voit, on décide ici d'expliciter dans l'abstraction de référence la condition de validité associée à l'opération de division, i.e. que le diviseur soit non-nul, sous la forme d'une pré-condition locale.

En l'état, ce module n'est pas prouvable, puisqu'on ne peut décharger l'obligation de preuve associée à la pré-condition de validité de la division :

$$y \in -128.. + 127 \not\Rightarrow 3 + y/2 \neq 0$$

¹¹L'implémentation du B dans *Click'n'Prove/B4Free v2* présente cependant la remarquable propriété de pouvoir automatiquement réexploiter les preuves déjà effectuées, même après modification des fichiers associés.

Il est envisageable, ici, de remonter automatiquement cette pré-condition du niveau local au niveau global de l'opération, et de promouvoir la nouvelle pré-condition vers la spécification, comme suit :

```

MACHINE div_s
INCLUDES schar_s
OPERATIONS
  out<--div(x,y)=PRE x:SCHAR & y:SCHAR & 3+y/2/=0 THEN out::SCHAR END
END

REFINEMENT div_a0
REFINES div_s
INCLUDES schar_s
OPERATIONS
  out<---div(x,y)=PRE x:SCHAR & y:SCHAR & 3+y/2/=0 THEN
    out<--schar_cast(x/(3+y/2))
  END
END

```

Dans le cas général cependant, cette remontée n'est pas toujours simple, puisque la pré-condition peut être exprimée à l'aide de variables locales et nécessiterait pour être valide à plus haut niveau des opérations de transformation de prédicat non abordées dans le cadre de cette étude.

Il reste cependant intéressant, une fois effectué par l'utilisateur un travail de remontée du niveau local au niveau global de l'opération, de prévoir une fonctionnalité de promotion automatique vers les abstractions supérieures et la spécification.

On admet donc ici que l'utilisateur effectue à la main, et sous le contrôle de l'outil, la remontée et la promotion de la pré-condition, puis modifie la spécification pour décrire une pré-condition suffisante plus explicite, et formaliser l'opération sans faire appel au cast (le résultat de l'opération restant dans le domaine de valeur du type `signed char`) :

```

MACHINE div_s
INCLUDES schar_s
OPERATIONS
  out<--div(x,y)=PRE x:SCHAR & y:SCHAR & y>=-5 THEN out:=x/(3+y/2) END
END

```

3.5.4 Appels de fonction et variables globales

On considère le code *C* suivant :

```

int val;
void mod(int x) {val=x;}
int mul(int x) {return val*x;}
int smq(int x) {int m=mul(x); return sqrt(m+val);}

```

La fonction `sqrt` est supposée externe, connue par la seule déclaration de son prototype et par un commentaire complaisamment fourni par le développeur :

```

/* sqrt(x) returns the square root of x, rounded down */
int sqrt(int x)

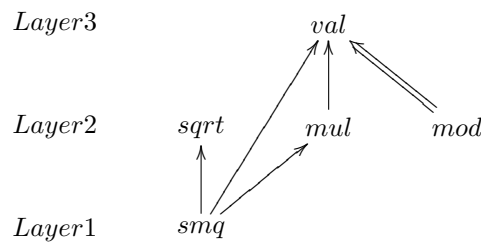
```

La variable globale `val` décrit un état interne du programme que l'on veut modéliser ; cet état interne peut être représenté en *B* par une variable formelle, qui doit être accessible en lecture et en écriture aux opérations `mod`, `mul` et `smq` (soit directement, soit par l'utilisation d'opération dédiées à la manipulation de cette variable).

Cette question réglée, nous sommes cependant confronté à d'autres difficultés pour la traduction de ce programme.

Nous pouvons en effet envisager de produire un unique module, avec une abstraction de référence qui définit les opérations `sqrt`, `mod`, `mul` et `smq`. Cette architecture n'est cependant pas valable, car les appels d'une opération par une autre opération du même module ne sont pas autorisés¹².

Au contraire, nous pouvons tenter de définir une architecture en plusieurs modules à partir du graphe de dépendance, à partir duquel nous définissons des couches logicielles¹³ :



La clause `INCLUDES` permet à un composant d'inclure une machine abstraite, i.e. de pouvoir lire ses ensembles, constantes et variables, et de pouvoir faire appel à ses opérations¹⁴. La clause `EXTENDS` correspond à une inclusion, suivie d'une promotion de l'ensemble des opérations de la machine abstraite incluse – qui sont alors considérées comme des opérations du composant incluant. Notons qu'une machine ne peut faire appel à ses propres opérations dans ses opérations, mais qu'elle peut faire appel aux opérations d'une machine incluse, même si celles-ci ont été promues.

Le premier objet à modéliser est la variable globale. Dans le cas général, la variable globale `val` pourra être modifiée par différentes fonctions, potentiellement liées par une dépendance (donc devant être définies dans des modules différents). Il est donc préférable d'attacher à cette variable une opération de lecture¹⁵ et une opération d'écriture, dans une spécification dédiée (à laquelle on attache une abstraction de référence afin de préserver la preuve d'adéquation en cas de modification de la spécification) :

```

MACHINE layer3_s
EXTENDS sint_s
VARIABLES val
INVARIANT val:SINT
INITIALISATION val::SINT
OPERATIONS
  Wval(x)=PRE x:SINT THEN val:=x END ;
  out<--Rval=BEGIN out:=val END
END
  
```

¹²L'origine de cette restriction n'est pas claire, et ne semble pas justifiée par la théorie dans laquelle un appel d'opération n'est qu'une simple substitution.

¹³Un algorithme correspondant sera décrit ultérieurement.

¹⁴Rappelons que les appels parallèles à des opérations actives d'une même machine incluse sont interdits par le *B*, et que les appels parallèles à des opérations d'une même machine incluse sont interdits par *Click'n'Prove/B4free v2*.

¹⁵Cette opération est en pratique optionnelle, puisqu'en travaillant par inclusion, cette variable sera accessible en lecture aux autres composants.

```

REFINEMENT layer3_a0
REFINES layer3_s
EXTENDS sint_s
VARIABLES val
INVARIANT val:SINT
INITIALISATION val::SINT
OPERATIONS
  Wval(x)=PRE x:SINT THEN val:=x END ;
  out<--Rval=BEGIN out:=val END
END

```

Nous pouvons ensuite développer le module associé aux fonctions `mod`, `mul` et `sqrt` :

```

MACHINE layer2_s
EXTENDS layer3_s
OPERATIONS
  mod_(x)=PRE x:SINT THEN val_put(x) END ;
  out<--mul(x)=PRE x:SINT THEN out::SINT END ;
  out<--sqrt(x)=PRE x:SINT THEN out::SINT END ;
END

REFINEMENT layer2_a0
REFINES layer2_s
EXTENDS layer3_s
OPERATIONS
  mod_(x)=PRE x:SINT THEN val_put(x) END ;
  out<--mul(x)=PRE x:SINT THEN
    VAR loc IN loc<--val_get ; out<--sint_cast(loc*x) END
    END ;
  out<--sqrt(x)=PRE x:SINT THEN out::{} END
END

```

Première remarque, nous avons été obligé de renommer `mod`, qui est un mot clé du B ne pouvant pas être utilisé comme nom d'opération.

Seconde remarque, nous ne pouvons pas dans la spécification modéliser l'opération `mod_` par un simple `skip`, bien qu'elle n'ait aucune valeur de sortie. En effet, par inclusion, la machine `mod_s` hérite de la variable `val`, qui est modifiée par le raffinement de `mod_` dans `layer2_a0`; moralement, cette opération a un paramètre de sortie caché.

Troisième remarque, en retenant cette approche par couches logicielles, nous avons décidé d'inclure la fonction `sqrt`. Jusqu'à présent, il n'était envisagé pour une telle fonction, dont nous n'avons que le prototype, que de générer une spécification, sans raffinement associé. En incluant l'opération correspondante dans un module plus général, nous sommes amenés à produire un raffinement. Un tel raffinement peut être, en première approche, une simple copie de la spécification, mais cela pose des problèmes de maintien de l'adéquation lors des modifications de la spécification (en particulier dans la mesure où l'abstraction de référence n'est pas modifiable).

L'idée est donc de conserver l'approche associant à une telle fonction une abstraction de référence. L'abstraction de référence proposée a cependant la particularité d'être générique, i.e. raffiner toute spécification.

Remarque 3.1 (Les miracles) *Une opération concrète C raffine une opération abstraite A si tout prédicat réalisé par A est également réalisé par C . Dès lors, toute opération C qui aurait pour particularité de rendre vrai tout prédicat est un raffinement générique; une telle opération est appelée un miracle.*

La substitution $x := \emptyset$ est un tel miracle, en effet, si P est un prédicat on a :

$$\begin{aligned}
[x := \emptyset]P &\Leftrightarrow [\text{@}y \cdot (y \in \emptyset \implies x := y)]P \\
&\Leftrightarrow \forall y \cdot ([y \in \emptyset \implies x := y]P) \\
&\Leftrightarrow \forall y \cdot (y \in \emptyset \implies [x := y]P) \\
&\Leftrightarrow \forall y \cdot (\perp \implies [x := y]P) \\
&\Leftrightarrow \forall y \cdot (\top) \\
&\Leftrightarrow \top
\end{aligned}$$

Il nous reste enfin à produire le module associé à la fonction `smq` :

```

MACHINE layer1_s
EXTENDS layer2_s
OPERATIONS
  out<--smq(x)=PRE x:SINT THEN out::SINT END
END

REFINEMENT layer1_a0
REFINES layer1_s
EXTENDS layer2_s
OPERATIONS
  out<--smq(x)=PRE x:SINT THEN
    VAR m,loc1,loc2 IN
      m<--mul(x) ;
      loc1<--val_get ;
      loc2<--sint_cast(m+loc1) ;
      out<--sqrt(loc2)
    END
  END
END

```

L'adéquation de l'ensemble des modules ainsi produit est prouvable automatiquement.

Enfin, l'utilisateur peut par exemple modifier les spécifications comme suit¹⁶ :

```

MACHINE layer2_s
EXTENDS layer3_s
OPERATIONS
  mod_(x)=PRE x:SINT THEN val_put(x) END ;
  out<--mul(x)=PRE x:SINT THEN out:=sint_cast(val*x) END ;
  out<--sqrt(x)=PRE x:SINT & x>=0 THEN
    ANY result WHERE result:0..46340 &
      result*result<=x &
      (result+1)*(result+1)>x
    THEN out:=result END
  END

MACHINE layer1_s
EXTENDS layer2_s
OPERATIONS
  out<--smq(x)=PRE x:SINT & val*(x+1)>=0 THEN
    out::0..46340
  END
END

```

¹⁶Notre utilisateur, versé dans les arcanes de l'arithmétique, est supposé avoir calculé que 46340 est la valeur maximale que peut renvoyer `sqrt`.

Ici encore l'adéquation est prouvable automatiquement, ce qui nous a permis d'explicitier une pré-condition, $x \in SINT \wedge val * (x + 1) \geq 0$, pour laquelle la valeur de retour est dans l'intervalle 0..46340.

3.6 Une première synthèse du processus d'abstraction

A partir de ces quelques exemples, on peut proposer quelques premiers principes relatifs au processus d'abstraction :

1. L'outil génère automatiquement à partir du code C un ensemble de modules B dans un processus de traduction :
 - (a) Le code C est traduit en un raffinement, nommé abstraction de référence, auquel est associée une machine abstraite, nommée spécification, ne comportant que les signatures des objets. L'abstraction de référence n'est pas toujours un raffinement correct de cette spécification initiale, certaines conditions de validité pouvant ne pas avoir été remontées.
 - (b) Les fonctions C dont seul le prototype est disponible peuvent être prises en compte, soit sous forme de spécification isolée, soit sous forme d'une spécification associée à un raffinement miraculeux.
 - (c) Le code C est analysé pour définir une hiérarchie de modules, hiérarchie traduite par des séquences d'inclusions ; les promotions d'opérations doivent être limitées au strict nécessaire.
 - (d) Le résultat du processus de traduction modélise explicitement l'ensemble des opérations implicites du C , notamment les cast et les opérations arithmétiques sur des entiers.
 - (e) L'outil assure la traçabilité de la génération automatique par la production de commentaires.
 - (f) L'outil génère des assertions liées aux obligations de preuve différées.
2. La modification de l'abstraction de référence est interdite, le processus d'abstraction se limite à l'ajout d'abstractions de niveau supérieur, à la modification de ces abstractions ou de la spécification.
3. Toute modification nécessite la reprise de la preuve d'adéquation.
4. L'objectif ultime est d'obtenir une spécification prouvée adéquate aussi fine que souhaitée, passant d'un paradigme concret, déterministe et séquentiel à un paradigme abstrait, non déterministe et parallèle.

On peut également identifier pour l'outil quelques fonctionnalités souhaitables en termes d'interfaces :

1. L'outil propose une visualisation adaptée du code C et du module associée, en éliminant les éléments purement techniques ou automatiques.
2. L'outil autorise la définition de post-conditions, voire d'invariants locaux.
3. L'outil autorise l'insertion d'une nouvelle abstraction a_{n+1} entre la dernière abstraction a_n et la spécification s , par copie de l'abstraction a_n ; les modifications nécessaires au maintien de la cohérence du module sont automatiques.
4. L'outil autorise la modification d'une abstraction ou d'une spécification ; les modifications des pré-conditions globales sont automatiquement propagées vers la spécification, les modifications des post-conditions globales peuvent être automatiquement propagées vers l'abstraction de référence.
5. L'outil propose une bibliothèque de spécifications pour les fonctions des principales bibliothèques du C .
6. L'outil propose une bibliothèque d'assertions pertinentes pour les preuves d'adéquation.

Chapitre 4

Modélisation arithmétique des opérations entières

Les types entiers du C sont de représentations binaires ; la sémantique des opérations associées à ces types relève de la définition du langage, de son implémentation, et du microprocesseur considéré.

L'objet de ce chapitre est d'aborder la question des spécifications et des abstractions de référence des opérations associées aux entiers¹, notamment pour les opérations de cast.

4.1 Sensibilisation

Les codes C suivants, compilés et exécutés sur l'architecture de référence, donnent parfois des résultats que l'on peut qualifier d'inattendus :

[E1]	<code>char x=0; x=x-1;</code>	\equiv	<code>x := -1</code>
[E2]	<code>unsigned char x=0; x=x-1;</code>	\equiv	<code>x := 255</code>
[E3]	<code>char x=127; x=x*2;</code>	\equiv	<code>x := -2</code>
[E4]	<code>unsigned char x=127; x=x*2;</code>	\equiv	<code>x := 254</code>
[E5]	<code>unsigned char x=128; x=x*2; x=x/2;</code>	\equiv	<code>x := 0</code>
[E6]	<code>unsigned char x=128; x=(x*2)/2;</code>	\equiv	<code>x := 128</code>
[E7]	<code>char x=-128; x=x*2; x=x/2;</code>	\equiv	<code>x := 0</code>
[E8]	<code>char x=-128; x=(x*2)/2;</code>	\equiv	<code>x := -128</code>
[E9]	<code>int x=1000000; x=x*x;</code>	\equiv	<code>x := -727379968</code>
[E10]	<code>unsigned int x=1000000; x=x*x;</code>	\equiv	<code>x := 3567587328</code>

¹En d'autres termes, comment modéliser " $a + b$ " en C en $B...$

4.2 Les types entiers du C

4.2.1 Représentation

Le C offre les types entiers natifs suivants² :

Type	Taille (bits)	Codage	Valeur minimale	Valeur maximale
char signed char	8	Complément à 2	-128	+127
unsigned char	8	Binaire	0	+255
short short int signed short signed short int int signed int long long int signed long signed long int	32	Complément à 2	-2^{31}	$+2^{31} - 1$
unsigned short unsigned short int unsigned int unsigned long unsigned long int	32	Binaire	0	$+2^{32} - 1$

Pour une valeur entière V on note $[V]_2$ la représentation binaire de taille minimale, $[V]_{2(n)}$ la représentation binaire sur n bits, $[V]_{2C}$ la représentation en complément à 2 de taille minimale, et $[V]_{2C(n)}$ la représentation en complément à 2 sur n bits (par hypothèse on admet dans ces notations n suffisamment grand).

Pour une suite de bits B on note $\langle B \rangle_2$ la valeur associée dans une interprétation binaire et $\langle B \rangle_{2C}$ la valeur associée dans une interprétation en complément à 2.

On note également $|B|$ la taille, $B \downarrow_n$ la tronquature aux n bits de poids faibles, $B \uparrow_{n(0)}$ l'extension à n bits (avec $n \geq |B|$) par ajout de 0, $B \uparrow_{n(S)}$ l'extension à n bits (avec $n \geq |B|$) par copie du bit de poids fort, et enfin \overline{B} le complémentaire.

Rappelons que la représentation en complément à 2 vérifie les propriétés suivantes :

$$\begin{aligned}
 [-V]_{2C(n)} &= (\overline{[V]_{2C(n)}} + 1) \downarrow_n \\
 V \in 0..(2^{n-1} - 1) &\Rightarrow [V]_{2C(n)} = [V]_{2(n)} \\
 V \in (-2^{n-1})..0 &\Rightarrow [V]_{2C(n)} = (\overline{[-V]_{2(n)}} + 1) \downarrow_n \\
 V \in (-2^{n-1})..(2^{n-1} - 1) \wedge m > n &\Rightarrow [V]_{2C(m)} = [V]_{2C(n)} \uparrow_{m(S)}
 \end{aligned}$$

²Comme indiqué en appendice 2, les caractéristiques des types données ici ne sont pas issues de la définition du C mais de l'analyse de l'implémentation du C sur l'architecture de référence.

4.2.2 Le cast

Le cast entre types entiers, en C , apparaît être une simple opération de projection d'une représentation binaire dans une autre :

- Si le cast est effectué vers une représentation de même taille il n'y a aucune modification.
- Si le cast est effectué vers une représentation de taille inférieure, il y a tronquature.
- Si le cast est effectuée depuis un type signé vers une représentation de taille supérieure, il y a extension du bit de signe (le bit de poids fort est recopié autant de fois que nécessaire).
- Si le cast est effectuée depuis un type non signé vers une représentation de taille supérieure, il y a extension par remplissage avec des bits à 0.

L'interprétation de la suite binaire ainsi obtenue dépend de la représentation associée à au type vers lequel opère le cast, i.e. binaire s'il est non signé et en complément à 2 si il est signé.

4.3 Modélisation arithmétique des cast

Les représentations binaires vérifient les propriétés arithmétiques suivantes :

$$\begin{array}{ll}
 \langle \overline{B} \rangle_2 = 2^{|B|} - 1 - \langle B \rangle_2 & \langle B \downarrow_n \rangle_2 = \langle B \rangle_2 \bmod 2^n \\
 \langle B \uparrow_{n(0)} \rangle_2 = \langle B \rangle_2 & \langle B \uparrow_{n(S)} \rangle_{2C} = \langle B \rangle_{2C} \\
 \langle B \rangle_2 < 2^{|B|-1} \Rightarrow \langle B \rangle_{2C} = \langle B \rangle_2 & \langle B \rangle_{2C} \geq 0 \Rightarrow \langle B \rangle_2 = \langle B \rangle_{2C} \\
 \langle B \rangle_2 \geq 2^{|B|-1} \Rightarrow \langle B \rangle_{2C} = \langle B \rangle_2 - 2^{|B|} & \langle B \rangle_{2C} < 0 \Rightarrow \langle B \rangle_2 = \langle B \rangle_{2C} + 2^{|B|}
 \end{array}$$

Remarque 4.1 (Fonction modulo) *Le modulo utilisé ici est celui de la Méthode B, a mod b n'est défini que si $a \geq 0$ et $b > 0$.*

On note $cast_S(m, B)$ (et $cast_U(m, B)$) l'interprétation du cast de la suite binaire B sur un type signé (et non signé) de m bits ; le cast est décrit par :

- Taille de destination supérieure ou égale, $m \geq n$:

$$cast_U(m, [V]_{2(n)}) = V$$

$$cast_S(m, [V]_{2C(n)}) = V$$

$$cast_U(m, [V]_{2C(n)}) = \begin{cases} V & \text{si } V \geq 0 \\ V + 2^m & \text{si } V < 0 \end{cases}$$

$$cast_S(m, [V]_{2(n)}) = \begin{cases} V & \text{si } V < 2^{m-1} \\ V - 2^m & \text{si } V \geq 2^{m-1} \end{cases}$$

- Taille de destination inférieur, $m < n$:

$$cast_U(m, [V]_{2(n)}) = V \bmod 2^m$$

$$cast_S(m, [V]_{2C(n)}) = \begin{cases} V \bmod 2^m & \text{si } V \geq 0 \wedge V \bmod 2^m < 2^{m-1} \\ V \bmod 2^m - 2^m & \text{si } V \geq 0 \wedge V \bmod 2^m \geq 2^{m-1} \\ (V + 2^n) \bmod 2^m & \text{si } V < 0 \wedge (V + 2^n) \bmod 2^m < 2^{m-1} \\ (V + 2^n) \bmod 2^m - 2^m & \text{si } V < 0 \wedge (V + 2^n) \bmod 2^m \geq 2^{m-1} \end{cases}$$

$$cast_U(m, [V]_{2C(n)}) = \begin{cases} V \bmod 2^m & \text{si } V \geq 0 \\ (V + 2^n) \bmod 2^m & \text{si } V < 0 \end{cases}$$

$$cast_S(m, [V]_{2(n)}) = \begin{cases} V \bmod 2^m & \text{si } V \bmod 2^m < 2^{m-1} \\ V \bmod 2^m - 2^m & \text{si } V \bmod 2^m \geq 2^{m-1} \end{cases}$$

On peut aussi proposer, de manière plus synthétique en travaillant sur les valeurs :

- Taille de destination supérieure ou égale, $m \geq n$:

$$\begin{aligned} \text{cast}_U(m, V) &= \begin{cases} V & \text{si } V \geq 0 \\ V + 2^m & \text{si } V < 0 \end{cases} \\ \text{cast}_S(m, V) &= \begin{cases} V & \text{si } V < 2^{m-1} \\ V - 2^m & \text{si } V \geq 2^{m-1} \end{cases} \end{aligned}$$

- Taille de destination inférieur, $m < n$:

$$\begin{aligned} \text{cast}_U(m, V) &= \begin{cases} V \bmod 2^m & \text{si } V \geq 0 \\ (V + 2^n) \bmod 2^m & \text{si } V < 0 \end{cases} \\ \text{cast}_S(m, V) &= \begin{cases} V \bmod 2^m & \text{si } V \geq 0 \wedge V \bmod 2^m < 2^{m-1} \\ V \bmod 2^m - 2^m & \text{si } V \geq 0 \wedge V \bmod 2^m \geq 2^{m-1} \\ (V + 2^n) \bmod 2^m & \text{si } V < 0 \wedge (V + 2^n) \bmod 2^m < 2^{m-1} \\ (V + 2^n) \bmod 2^m - 2^m & \text{si } V < 0 \wedge (V + 2^n) \bmod 2^m \geq 2^{m-1} \end{cases} \end{aligned}$$

Notons enfin que l'on peut éliminer toutes les occurrences de n dans les formules ci-dessus en réécrivant les expressions de la forme $(V + 2^n) \bmod 2^m$, avec $m < n$, comme suit :

$$\begin{aligned} (V + 2^n) \bmod 2^m &= (V + 2^{n-m} \cdot 2^m) \bmod 2^m \\ &= V + \lambda \cdot 2^m \text{ avec } \lambda = \min(\{l \mid l \in \mathbb{N} \wedge V + l \cdot 2^m \geq 0\}) \end{aligned}$$

4.4 Modélisation arithmétique des opérations

Il faut également, si on veut représenter fidèlement les opérations arithmétiques du C , modéliser les processus de calcul implémentés par le C ; une telle modélisation n'est pas immédiate, comme l'illustrent les exemple [E5], [E6] et [E9] de la section 5.1.

En particulier, la différence de résultat entre [E5] et [E6] est choquante ; une analyse du code assembleur généré permet de mettre en évidence que les opérations de cast ne sont effectuées qu'au moment des lectures et écriture en mémoire (lecture et écriture d'une variable), les calculs intermédiaires étant effectués sur un registre 32 bits³.

Le type des registres est normalement représenté par le type `int`, on peut donc considérer que les deux programmes ci-dessous sont équivalents :

```

unsigned char x=128;
x=(x*2)/2;
≡
unsigned char x=128;
int reg=x; // Lecture mémoire
int reg=reg*2; // Multiplication
int reg=reg/2; // Division
x=(unsigned char)reg; // Ecriture mémoire

```

Une fois établi le fait que les calculs intermédiaires sont virtuellement réalisés comme des `int`, d'autres questions se posent ; par exemple, que se passe-t-il quand un résultat intermédiaire provoque un débordement ?

La réponse à cette question ne peut être obtenue que par une analyse fine du fonctionnement du microprocesseur ; tout au plus peut-on noter, avec les exemples [E9] et [E10], qu'il semble que pour ce qui concerne la multiplication, le résultat obtenu correspond effectivement à un cast sur un `int` de la valeur réelle du résultat⁴.

³Il s'agit ici d'une généralisation ; pour l'exemple précis considéré, i.e. $x = x * 2 / 2$, le code assembleur généré est optimisé, les deux opérations ont tout simplement été supprimées.

⁴Sans sous-entendre en aucune façon que le microprocesseur travaille sur plus de 32 bits ; en pratique, l'algorithme

4.5 Modules des entiers C en B

Nous pouvons maintenant proposer une modélisation formelle en B des opérations sur les entiers ; nous ne traitons ici que deux exemples représentatifs, le type `unsigned char` et le type des registres.

4.5.1 Le type *unsigned char*

La spécification $uchar_s$ et l'abstraction de référence $uchar_{a0}$ définissent les opérations sur le type `unsigned char`.

Comme on l'a vu, aucune opération n'est réalisée dans ce type ; elles sont effectuées en `int`, le résultat subissant un cast ; la spécification propose donc cette seule opération de cast.

Trois spécifications différentes pour l'opération `cast` peuvent être considérées, présentées ici sous forme d'opérations distinctes :

```

MACHINE  $uchar_s$ 
CONSTANTS  $minuchar, maxuchar, UCHAR, sizeuchar$ 
PROPERTIES  $minuchar = 0 \wedge maxuchar = 255 \wedge$ 
            $UCHAR = minuchar..maxuchar \wedge$ 
            $sizeuchar = maxuchar - minuchar + 1$ 
OPERATIONS
   $out \leftarrow uchar\_cast\_s(x) \hat{=} PRE\ x \in UCHAR\ THEN\ out := x\ END ;$ 
   $out \leftarrow uchar\_cast\_m(x) \hat{=} PRE\ x \in \mathbb{Z}\ THEN$ 
    IF  $x \in UCHAR\ THEN\ out := x\ ELSE\ out :: UCHAR\ END$ 
  END ;
   $out \leftarrow uchar\_cast\_p(x) \hat{=} PRE\ x \in \mathbb{Z}\ THEN$ 
    IF  $x \geq 0\ THEN$ 
       $out := x \bmod sizeuchar$ 
    ELSE
       $out := x + \min(\{l \mid l \in \mathbb{N} \wedge l \bmod sizeuchar = 0 \wedge x + l \geq 0\})$ 
    END
  END
END

```

L'opération $uchar_cast_s$ est la version stricte ; le cast est réduit à une restriction de l'identité⁵, et en pratique impose la preuve qu'il n'y a pas de dépassement dès lors que l'on manipule des variables de type `unsigned char`.

L'opération $uchar_cast_m$ est la version moyenne ; le cast est décrit comme donnant un résultat indéterminé si on sort de la plage de valeur du type `unsigned char`, et en pratique autorise les cas de dépassements mais interdit de prouver des propriétés intéressantes dans de tels cas.

L'opération $uchar_cast_p$ est la version permissive ; le cast est décrit de manière très précise, arithmétique, les dépassements sont donc autorisés et compris, en ce sens que le résultat du cast peut être décrit par une expression arithmétique précise que l'on peut utiliser dans les preuves.

Une seule de ces opérations doit être conservée dans la machine, sous le nom $uchar_cast$.

de multiplication utilisé est sans doute par puissances croissantes, les retenues de poids fort sont perdues mais les bits de poids faibles sont corrects.

⁵Cette opération est néanmoins nécessaire, puisqu'elle est utilisée dans l'abstraction de référence générée à partir du code C , abstraction qui ne peut être modifiée par l'utilisateur.

Quelque soit la spécification retenue, on lui associe l'abstraction de référence suivante :

```

REFINEMENT uchara0
REFINES uchars
OPERATIONS
  out ← uchar_cast(x) ≐ PRE x ∈ ℤ THEN
    IF x ≥ 0 THEN
      out := x mod sizeuchar
    ELSE
      out := x + min({l | l ∈ ℕ ∧ l mod sizeuchar = 0 ∧ x + l >= 0})
    END
  END
END

```

Rappelons que la définition d'une abstraction de référence permet de laisser à l'utilisateur la possibilité de modifier la spécification associée comme il l'entend, tant que ce qu'il décrit est en adéquation avec ce qui est effectivement réalisé. En particulier, l'utilisateur pourra, sans remettre en cause l'adéquation, choisir dans la spécification la version stricte, moyenne ou permissive.

4.5.2 Le type registre

La spécification *register*_s et l'abstraction de référence *register*_{a0} définissent les opérations arithmétiques.

La spécification propose l'opération de cast – que l'on peut voir ici comme caractérisant le chargement d'un registre, mais également les opérations arithmétiques (ici limitées aux seules opérations de base) ; la version “moyenne” de la spécification est retenue⁶ :

```

MACHINE registers
CONSTANTS minreg, maxreg, REG, sizereg
PROPERTIES minreg = -2147483648 ∧ maxreg = 2147483647 ∧
  REG = minreg..maxreg ∧
  sizereg = maxreg - minreg + 1
OPERATIONS
  out ← reg_cast(x) ≐ PRE x ∈ ℤ THEN
    IF x ∈ REG THEN out := x ELSE out ∈ REG END
  END ;
  out ← reg_add(x, y) ≐ PRE x ∈ REG ∧ y ∈ REG THEN
    IF x + y ∈ REG THEN out := x + y ELSE out ∈ REG END
  END ;
  out ← reg_sub(x, y) ≐ PRE x ∈ REG ∧ y ∈ REG THEN
    IF x - y ∈ REG THEN out := x - y ELSE out ∈ REG END
  END ;
  out ← reg_mul(x, y) ≐ PRE x ∈ REG ∧ y ∈ REG THEN
    IF x * y ∈ REG THEN out := x * y ELSE out ∈ REG END
  END ;
  out ← reg_div(x, y) ≐ PRE x ∈ REG ∧ y ∈ REG ∧ y ≠ 0 THEN
    IF x/y ∈ REG THEN out := x/y ELSE out ∈ REG END
  END
END

```

⁶Dans la version “stricte” la pré-condition de l'addition est $x \in SINT \wedge y \in SINT \wedge x + y \in SINT$.

L'abstraction de référence associée détaille les différentes opérations⁷ :

```

REFINEMENT sinta0
REFINES sints
CONSTANTS smd
PROPERTIES smd =  $\lambda n \cdot (n \in \mathbb{Z} \mid n + \min(\{l \mid l \in \mathbb{N} \wedge l \bmod \text{sizereg} = 0 \wedge n + l \geq 0\}))$ 
OPERATIONS
  out  $\leftarrow$  reg_cast(x)  $\hat{=}$  PRE x  $\in$   $\mathbb{Z}$  THEN
    IF x  $\geq$  0 THEN
      IF x  $\bmod$  sizereg < maxreg THEN out := x  $\bmod$  sizereg
      ELSE out := (x  $\bmod$  sizereg) - sizereg END
    ELSE
      IF smd(x) < maxreg THEN out := smd(x)
      ELSE out := smd(x) - sizereg END
    END
  END ;
  out  $\leftarrow$  reg_add(x, y)  $\hat{=}$  PRE x  $\in$   $\mathbb{Z} \wedge y \in \mathbb{Z}$  THEN
    IF x + y  $\geq$  0 THEN
      IF (x + y)  $\bmod$  sizereg < maxreg THEN out := (x + y)  $\bmod$  sizereg
      ELSE out := ((x + y)  $\bmod$  sizereg) - sizereg END
    ELSE
      IF smd(x + y) < maxreg THEN out := smd(x + y)
      ELSE out := smd(x + y) - sizereg END
    END
  END ;
  out  $\leftarrow$  reg_sub(x, y)  $\hat{=}$  PRE x  $\in$   $\mathbb{Z} \wedge y \in \mathbb{Z}$  THEN
    IF x - y  $\geq$  0 THEN
      IF (x - y)  $\bmod$  sizereg < maxreg THEN out := (x - y)  $\bmod$  sizereg
      ELSE out := ((x - y)  $\bmod$  sizereg) - sizereg END
    ELSE
      IF smd(x - y) < maxreg THEN out := smd(x - y)
      ELSE out := smd(x - y) - sizereg END
    END
  END ;
  out  $\leftarrow$  reg_mul(x, y)  $\hat{=}$  PRE x  $\in$   $\mathbb{Z} \wedge y \in \mathbb{Z}$  THEN
    IF x * y  $\geq$  0 THEN
      IF (x * y)  $\bmod$  sizereg < maxreg THEN out := (x * y)  $\bmod$  sizereg
      ELSE out := ((x * y)  $\bmod$  sizereg) - sizereg END
    ELSE
      IF smd(x * y) < maxreg THEN out := smd(x * y)
      ELSE out := smd(x * y) - sizereg END
    END
  END ;
  out  $\leftarrow$  reg_div(x, y)  $\hat{=}$  PRE x  $\in$   $\mathbb{Z} \wedge y \in \mathbb{Z} \wedge y \neq 0$  THEN
    IF x/y  $\geq$  0 THEN
      IF (x/y)  $\bmod$  sizereg < maxreg THEN out := (x/y)  $\bmod$  sizereg
      ELSE out := ((x/y)  $\bmod$  sizereg) - sizereg END
    ELSE
      IF smd(x/y) < maxreg THEN out := smd(x/y)
      ELSE out := smd(x/y) - sizereg END
    END
  END ;
END

```

⁷Elle constitue donc une modélisation du langage, du compilateur et du microprocesseur.

4.5.3 Modularité

La spécification $uchar_s$ et son abstraction de référence associée $uchar_{a0}$ sont présentées de manière générique ; la modification de ce module pour qu'il puisse représenter un autre type entier se limite en effet à la modification des noms et de la clause `PROPERTIES`, afin de modifier les valeurs minimales et maximales.

En pratique, il est plus approprié de définir une spécification et une abstraction générique, qui peuvent ensuite être instanciées avec le type entier que l'on souhaite modéliser. Cette approche, plus modulaire, évite de multiplier les modèles et permet de factoriser les éventuelles preuves.

Une modélisation générique est présentée en annexe A.

Chapitre 5

Modélisation des structures de contrôle

Ce chapitre rappelle les différentes formes de structures de contrôle du C , et pour chacune d'entre elles propose une substitution B permettant de la modéliser dans une abstraction.

La procédure de traduction est présentée de manière informelle ; ce chapitre a pour vocation d'identifier les problématiques à traiter. La formalisation de la traduction est présentée dans les chapitre 8 et 9.

5.1 Structures *if*

La forme générale d'une structure `if` en C et la traduction en B proposée sont données ci-dessous :

$$\begin{array}{l} \text{if } (expr) \text{ bloc}^1 \\ \text{else } \text{bloc}^2 \end{array} \xrightarrow{B} \begin{array}{l} \text{IF } expr \text{ THEN } \text{bloc}^1 \\ \text{ELSE } \text{bloc}^2 \text{ END} \end{array}$$

Cette traduction est cependant simpliste, puisque différents problèmes se posent en pratique ; principalement, il s'agit du traitement des instructions `continue`, `break` et `return`, abordé dans la section 5.6 du présent chapitre, et des effets de bords.

On parle ici d'effets de bord lorsque l'évaluation d'une expression passée en paramètre modifie la valeur de une ou plusieurs variables. De tels effets de bords résultent d'une philosophie générale du langage C ne distinguant pas les différentes catégories syntaxiques ; en pratique ils sont très fréquents, étant utilisés pour assurer une plus grande compacité du code.

Exemple 5.1 (Effet de bord) *Une illustration de l'effet de bord est fournie par une erreur de programmation fréquente, résultant d'une distraction :*

```
if (a=b) printf("b!=0\n"); else printf("b==0\n");
```

En effet, dans la syntaxe du C , `==` désigne l'opérateur booléen et `=` l'affectation.

L'instruction `if` décrite ici a donc pour effet d'affecter à `a` la valeur de `b`. Le "résultat" booléen de cette affectation est vrai si et seulement si la valeur de `b` est différente de 0¹.

¹Conformément à la spécification du C ; notons que ce langage reste proche de l'assembleur, et que le `if` est traduit par une instruction de branchement conditionnel dont le comportement dépend d'un registre d'état enregistrant le fait que la dernière opération effectuée a donné un résultat nul.

Le B , au contraire, distingue les substitutions et les expressions; les effets de bords ne sont donc pas directement représentables.

Nous nous proposons donc de traiter du problème des effets de bord en procédant, préalablement à la traduction, à une étape de *pre-cooking* – une réécriture interne au C , que l'on admet neutre vis-à-vis du fonctionnement du code considéré. L'idée est donc de proposer une traduction du C au B pour des formes normales de code, ainsi qu'un processus de normalisation de code C .

Le pre-cooking doit nous permettre, dans un premier temps, d'éliminer les effets de bords apparaissant dans l'expression paramétrant une structure `if` d'un code C .

On considère qu'il y a effet de bord dans une expression si elle comporte :

- Un opérateur d'affectation (`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `|=`).
- un opérateur préfixe (`++`, `--`).
- un opérateur postfixe (`++`, `--`).
- un appel de fonction².

Le principe général du pre-cooking éliminant les effets de bord pour les structures `if` est le suivant, avec $expr_s$ une expression comportant un effet de bord :

$$\begin{array}{l} \text{if } (expr_s) \text{ bloc}^1 \\ \text{else } \text{bloc}^2 \end{array} \xrightarrow{P_c} \begin{array}{l} \{ \text{int tmp}=(\text{int})expr_s; \\ \text{if } (\text{tmp}) \text{ bloc}^1 \\ \text{else } \text{bloc}^2 \} \end{array}$$

Remarque 5.1 (Appels d'opération en B) L'appel d'une fonction C se traduit en B par un appel d'opération, dont la syntaxe est fondamentalement différente, comme le montre l'exemple suivant :

$$\begin{array}{l} x=\text{sqrt}(\text{abs}(y)); \end{array} \xrightarrow{B} \begin{array}{l} \text{VAR tmp IN} \\ \text{tmp} \leftarrow \text{abs}(y); x \leftarrow \text{sqrt}(tmp) \\ \text{END} \end{array}$$

Indépendamment des problèmes d'effet de bord, des considérations syntaxiques nous imposeraient donc d'éliminer, préalablement à la traduction, les appels de fonctions dans les expressions.

Remarque 5.2 (Utilisation du ELSIF) Le B offre des substitutions `IF` dans laquelle il est possible de déclarer des blocs `ELSIF`; on peut vouloir utiliser cette possibilité pour proposer une traduction plus compacte de la forme suivante :

$$\begin{array}{l} \text{if } (expr_s^1) \text{ bloc}^1 \\ \text{else if } (expr_s^2) \text{ bloc}^2 \\ \dots \\ \text{else if } (expr_s^n) \text{ bloc}^n \\ \text{else } \text{bloc}^d \end{array} \xrightarrow{B} \begin{array}{l} \text{IF } expr_s^1 \text{ THEN } \text{bloc}^1 \\ \text{ELSEIF } expr_s^2 \text{ THEN } \text{bloc}^2 \\ \dots \\ \text{ELSEIF } expr_s^n \text{ THEN } \text{bloc}^n \\ \text{ELSE } \text{bloc}^d \\ \text{END} \end{array}$$

Cette traduction n'a cependant pas été retenue, pour des raisons de simplicité, mais aussi parce que le pre-cooking mène à modifier très fréquemment la structuration du code C considéré et interdit cette traduction, comme dans l'exemple suivant :

$$\begin{array}{l} \text{if } (expr_s^1) \text{ then } \text{bloc}^1 \\ \text{else if } (expr_s^2) \text{ then } \text{bloc}^2 \\ \text{else } \text{bloc}^3 \end{array} \xrightarrow{P_c} \begin{array}{l} \{ \text{int tmp1}=expr_s^1; \\ \text{if } \text{tmp1 } \text{bloc}^1 \\ \text{else} \\ \{ \text{int tmp2}=expr_s^2; \\ \text{if } \text{tmp2 } \text{bloc}^2 \\ \text{else } \text{bloc}^3 \} \} \end{array}$$

²Dans le cas général, une fonction peut en effet modifier les paramètres passés par référence, ou modifier des variables globales utilisées par ailleurs dans l'expression.

Remarque 5.3 (Expressions conditionnelles) *Le C permet de définir des expressions conditionnelles selon la syntaxe $e?a:b$; de telles expressions ne sont pas considérées comme une structure de contrôle.*

5.2 Structures switch

En première approche, nous choisissons de limiter la traduction à une forme précise de structure `switch` du C, reflétant le mieux la substitution CASE proposée par le B :

<pre>switch (expr) { case c¹:bloc¹ break; case c²:bloc² break; ... case cⁿ:blocⁿ break; default:bloc^d break}</pre>	\xrightarrow{B}	<pre>CASE expr OF EITHER c¹ THEN bloc¹ OR c² THEN bloc² ... OR cⁿ THEN blocⁿ ELSE bloc^d END</pre>
---	-------------------	--

Pour mémoire, *expr* est une expression entière sans effet de bord, et c^1, c^2, \dots, c^n sont des constantes entières distinctes.

Remarque 5.4 (CASE et SELECT en B) *La substitution CASE impose que les alternatives soient disjointes, la substitution SELECT permet des alternatives non disjointes mais est dans ce cas non déterministe.*

On peut ensuite étendre la traduction de formes plus générales des structure `switch` par pre-cooking – i.e. comme défini précédemment une réécriture interne au C sans effet observable à l'exécution. Le pre-cooking doit nous permettre d'éliminer les effets de bord, à l'image de ce qui était déjà proposé pour la structure `if`, mais aussi le cas échéant de réorganiser la structure de contrôle `switch` pour la mettre dans la forme ci-dessus.

Il nous faut donc des règles permettant de normaliser l'utilisation des `break`³, de la forme suivante :

<pre>switch (expr) { ... label:bloc }</pre>	$\xrightarrow{P_c}$	<pre>switch (expr) { ... label:bloc break; }</pre>
<pre>label¹:bloc¹ label²:bloc² break;</pre>	$\xrightarrow{P_c}$	<pre>label¹:bloc¹ bloc² break; label²:bloc² break;</pre>
<pre>label¹:bloc¹ break; bloc²</pre>	$\xrightarrow{P_c}$	<pre>label¹:bloc¹ break;</pre>

5.3 Structures while

La traduction d'une structure `while` est la suivante :

<pre>while (expr) bloc</pre>	\xrightarrow{B}	<pre>WHILE expr DO bloc INVARIANT exprⁱ VARIANT expr^v END</pre>
------------------------------	-------------------	---

³Rappelons qu'en C La clause `default` n'est effectivement sélectionnée que si toute les alternatives proposées sont fausses, mais qu'elle peut être placée n'importe où dans le `switch`, y compris en première alternative.

Nous sommes confrontés pour cette traduction à un problème d'identification de l'invariant $expr^i$ et du variant $expr^v$ qui sont obligatoires en B . Rappelons que les substitutions du B sont définies non pas comme des opérateurs sur des données, mais bien comme des transformateurs de prédicats, et que la définition de la substitution **WHILE** est donnée par l'équivalence suivante :

$$\left(\begin{array}{l} I \\ \wedge \forall x \cdot (I \wedge P \Rightarrow [S]I) \\ \wedge \forall x \cdot (I \Rightarrow V \in \mathbb{N}) \\ \wedge \forall x \cdot (I \wedge P \Rightarrow [n := V][S](V < n)) \\ \wedge \forall x \cdot (I \wedge \neg P \Rightarrow R) \end{array} \right) \Leftrightarrow \left[\begin{array}{l} \text{WHILE } P \\ \text{DO } S \\ \text{INVARIANT } I \\ \text{VARIANT } V \\ \text{END} \end{array} \right] R$$

Cette définition montre que les expressions $expr^i$ et $expr^v$ ont un rôle fondamental non seulement pour prouver les propriétés de la substitution **WHILE** mais aussi pour exprimer la nature même de cette substitution, comme l'illustre l'exemple suivant :

Exemple 5.2 (Recherche du minimum) *La substitution suivante réalise le prédicat $m = \min(S)$, pour $S \in \mathbb{F}_1(\mathbb{N})$:*

```

      WHILE  $m \notin S$ 
      DO  $m := m + 1$ 
 $m := 0$ ;  INVARIANT  $m \in 0..\min(S)$ 
          VARIANT  $\min(S) - m$ 
      END

```

On peut par contre prouver que la même substitution où l'invariant est affaibli ne réalise pas, dans le cas général, $m = \min(S)$:

```

      WHILE  $m \notin S$ 
      DO  $m := m + 1$ 
 $m := 0$ ;  INVARIANT  $m \in \mathbb{N}$ 
          VARIANT  $\min(S) - m$ 
      END

```

Clairement, le choix d'un bon variant et d'un bon invariant dépend de la propriété que l'on cherche à démontrer. Leur génération automatique n'est pas considérée dans cette étude, les expressions $expr^i$ et $expr^v$ sont donc laissées libres ; elles devront être modifiées par l'utilisateur dans l'abstraction de référence⁴.

Notons enfin que, comme précédemment, des règles de pre-cooking doivent permettre d'éliminer les effets de bord.

⁴Jusqu'à présent cette abstraction était considérée comme non modifiable, car constituant le seul lien avec le code C analysé ; nous admettons dans la suite que des modifications sont possibles, mais uniquement sur des éléments additionnels au code, tels que le variant ou l'invariant ici.

5.4 Structures *do*

La traduction d'une structure *do* est la suivante⁵ :

$$\text{do } \text{bloc} \text{ while } (\text{expr}) \quad \xrightarrow{B} \quad \begin{array}{l} \text{VAR } \text{first} \text{ IN} \\ \text{first} := \text{TRUE}; \\ \text{WHILE } \text{first} = \text{TRUE} \vee \text{expr} \\ \text{DO } \text{first} := \text{FALSE}; \text{bloc} \\ \text{INVARIANT } \text{expr}^i \\ \text{VARIANT } \text{expr}^v \\ \text{END} \\ \text{END} \end{array}$$

Des règles de pre-cooking doivent permettre d'éliminer les effets de bord.

5.5 Structures *for*

La traduction d'une structure *for* est la suivante :

$$\text{for } (\text{bloc}^1; \text{expr}; \text{bloc}^2) \text{ bloc}^3 \quad \xrightarrow{B} \quad \begin{array}{l} \text{bloc}^1; \\ \text{WHILE } \text{expr} \\ \text{DO } \text{bloc}^3; \text{bloc}^2 \\ \text{INVARIANT } \text{expr}^i \\ \text{VARIANT } \text{expr}^v \\ \text{END} \end{array}$$

Des règles de pre-cooking doivent permettre d'éliminer les effets de bord.

5.6 Instructions de rupture

On désigne sous le vocable *instructions de rupture* les instructions *continue*, *break* et *return*. Ces instructions ont la particularité de modifier l'exécution des structures de contrôle, et en pratiquent provoquent des phénomènes d'interférences entre les structures, comme l'illustre l'exemple suivant :

Exemple 5.3 (Débordement du *break*)

```
switch (exprs)
{ ...
  case ci: if (expri) break; ...
  ... }
```

En effet, le *break* dans cet exemple ne s'applique pas à la structure *if* où il apparaît mais à la sur-structure *switch*; la traduction doit donc, bien qu'étant décrite par des règles inductives, prendre en compte ce type de phénomène.

Essentiellement, il s'agit de voir une instruction de rupture comme empêchant l'exécution des instructions qui la suivent dans la structure à laquelle elle s'applique. On peut simuler ce type de mécanisme par des substitutions conditionnelles.

Ainsi, les deux codes suivants sont observationnellement équivalents :

⁵En première approche, il semble possible par pre-cooking de réécrire une structure *do* en une structure *while*; une telle équivalence, cependant, ne vaut que si le bloc d'instructions constituant le corps de la boucle ne contient pas d'instructions de rupture.

Exemple 5.4 (Modélisation du break dans un while)

```

while (exprw)
{ bloc1
  if (expri) { bloc2 break; bloc3 }
  bloc4 }

```

 \equiv

```

int brk=0;
while (exprw && !brk)
{ bloc1
  if (expri) { bloc2 brk=1; }
  if (!brk) bloc4 }

```

Nous pouvons donc envisager de traiter les instructions de rupture par pre-cooking ; nous choisissons cependant de les traiter dans le processus de traduction (cf. le chapitre 9).

Remarque 5.5 (Commande *exit*) *exit* n'est pas un mot clé réservé du C, mais une fonction utilitaire définie dans la bibliothèque *stdlib.h*. Elle n'est pas traitée dans le cadre de cette étude.

5.7 Instruction *goto*

La modélisation du *goto* apparaît difficile, du moins dans le cas général, puisque différents cas de figure sont possibles :

- Associé à une condition, le *goto* peut correspondre à une exécution conditionnelle de code et doit être modélisé par un IF.
- Le *goto* peut correspondre à une exécution de rupture, et correspondre à une suite de *break* ;.
- Enfin, le *goto* permet de coder une boucle et doit être modélisé par un WHILE.

Pour cette étude, aucune modélisation n'est proposée pour le *goto*⁶

Modélisation 6 *Seuls les programmes sans goto sont considérés.*

⁶Notons que si le *goto* est à prendre en compte au moins de manière partielle, son utilisation comme une instruction de rupture est la plus importante, permettant de contourner ce qui est perçu comme une lacune de ce langage.

Chapitre 6

Modélisation des pointeurs et des structures

Les objets manipulés jusqu'à présent dans les codes C se sont limités à des variables entières ; ce chapitre propose d'étendre la modélisation aux structures et aux pointeurs.

6.1 Fonction de valuation

Les modélisations présentées dans le chapitre 3 associait à une variable (entière) d'un code C une variable de type correspondant en B .

Une autre démarche consiste à définir en B l'ensemble des identifiants et une fonction de valuation, ou plus exactement, en raison des contraintes de typage, un ensemble d'identifiants et une fonction de valuation par type.

Avec cette démarche, il n'est plus nécessaire de prévoir une opération d'accès en lecture (écriture) par variable apparaissant dans le code C ; une opération d'accès en lecture (écriture) par type, acceptant en paramètre un identifiant de variable, suffit.

D'autre part, dans notre modélisation nous pouvons déclarer la fonction de valuation comme une fonction partielle de l'espace des identifiants dans l'espace des valeurs, ce qui permet de représenter l'état d'une variable non initialisée, et implicitement d'en interdire la lecture.

Exemple 6.1 (Modélisation de variables entières)

<code>int x=0;</code>	\xrightarrow{B}	SETS $IDSINT = \{x, y\}$
<code>int y;</code>		VARIABLES $valsint$
		INVARIANT $valsint \in IDSINT \leftrightarrow SINT$
		INITIALISATION $valsint := \{x \mapsto 0\};$
		OPERATIONS
		$Wsint(i, v) \hat{=} \text{PRE } i \in IDSINT \wedge v \in SINT$
		$\text{THEN } valsint(i) := v \text{ END};$
		$out \leftarrow Rsint(i) \hat{=} \text{PRE } i \in \text{dom}(valsint)$
		$\text{THEN } out := valsint(i) \text{ END}$

Nous explicitons ici par une pré-condition le fait qu'une variable doit être initialisée avant d'être lue. Cette contrainte est cependant implicite, liée à l'utilisation de la fonction de valuation, et la pré-condition peut être remplacée par $i \in IDSINT$.

6.2 Modélisation des constantes

Une constante du C est normalement caractérisée par le fait qu'elle ne peut être écrite qu'une seule fois, à l'initialisation¹.

Formellement, l'abstraction d'une constante globale dans un code C peut être représentée par une constante en B . Une telle constante B est "initialisée" par une propriété, et ne peut qu'être lue (elle est d'ailleurs visible par inclusion, il n'est donc pas nécessaire de définir une opération d'accès).

Cette approche, cependant, pose quelques difficultés, en particulier dans le cas de l'initialisation ; en effet, les propriétés associées aux constantes en B ne peuvent dépendre que des autres constantes et des ensembles donnés.

On choisit donc de représenter les constantes comme des variables, selon la modélisation proposée dans le paragraphe précédent. La représentation en B sous forme de variable n'est pas gênante, puisque si le code C ne modifie pas la constante, la variable ne sera jamais modifiée dans l'abstraction de référence.

Remarque 6.1 (Utilisation de l'invariant) *Si l'on souhaite insister sur le caractère constant d'une variable, on peut dans cette modélisation définir un invariant de la forme :*

$$\text{valtype}(id) = \text{cste}$$

6.3 Modélisation des structures

Une structure, en C , est un n -uplet ; à chaque position correspond un identifiant de champ et un type. La *méthode B* permet de représenter une telle structure de différentes manières, notamment comme un produit cartésien de type, ou comme une fonction partielle². L'outil *B4Free/Click'n'Prove v2.0* propose en complément des constructions dédiées, non considérées ici.

6.3.1 Fonctions de valuation multiples

En première approche, nous pouvons modéliser une variable de type structure comme un unique identifiant associé à plusieurs fonctions de valuation.

Exemple 6.2 (Structures et fonctions de valuations)

<pre>struct cpl { int x; char y; }; struct cpl c1,c2;</pre>	\xrightarrow{B}	<pre>SETS IDCPL = {c1, c2} VARIABLES valcplx, valcply INVARIANT valcplx ∈ IDPOINT ↔ SINT ∧ valcply ∈ IDPOINT ↔ SCHAR INITIALISATION valcplx = ∅; valcply = ∅</pre>
--	-------------------	---

Dans ce cas, on ne peut cependant manipuler que des structures ne contenant pas d'autres structures. On peut envisager de déplier les structures complexes, par pre-cooking ou lors de la traduction, mais un tel dépliage peut poser problème lors de la manipulation de structures par des pointeurs – de plus, en menant le dépliage à son terme, on scinde une variable de type structure en un ensemble de variables de type simple.

¹Cf. la remarque 6.6 pour un exemple de violation de ce principe.

²Sous réserve que les différents champs soient tous du même type

6.3.2 Produit cartésien

La représentation d'une structure par un produit cartésien constitue une modélisation plus appropriée, permettant la composition de structures, mais au prix d'une certaine lourdeur des opérations sur les champs.

En effet, contrairement à ce que l'on peut envisager en première approche, il n'est pas opportun de décrire l'accès aux champs sous la forme d'opérations d'accès dédiées, car de telles opérations ne peuvent être définies pour un type inhabité, comme dans l'exemple suivant :

Exemple 6.3 (Type structure inhabité)

```
struct pt { int x;
           int y; };
struct rect { struct pt p1;
             struct pt p2;
             char col; };
struct rect r1,r2;
```

En effet, si on associe au type `pt` une constante PT en B , et des opérations d'accès aux différents champs par simple projection, ces opérations doivent être paramétrées par un identifiant de variable, alors que le type PT est ici inhabité i.e. que l'ensemble énuméré des identifiants est vide – une situation à éviter en B .

Remarque 6.2 (Types inhabités) *Pour un type structure inhabité (i.e. ici n'ayant aucune instance globale), il n'est pas possible de définir l'ensemble des identifiants, ni de décrire la fonction de valuation, ni d'écrire les opérations d'accès; les types structures définis en C mais non directement utilisés seront donc traduits uniquement par la définition d'ensembles constants obtenus par produit cartésien.*

La même remarque s'applique à l'ensemble des types; on peut définir l'ensemble des valeurs mais pas les fonctions de valuation ou les opérations d'accès si le type est inhabité.

Les opérations d'accès à un champ d'un type structure ne peuvent donc être définies que si le type est habité; pourtant, elles sont nécessaires même lorsque le type est inhabité. Il est donc nécessaire de décrire les accès aux champs non pas sous la forme d'opérations mais sous la forme de fonctions.

En raison du typage du produit cartésien (non associatif), la définition de telles fonctions peut-être lourde; cependant, leur génération automatique ne pose guère de difficultés. Elles suivent en effet le schéma suivant, pour un type structuré tp dont les champs f_1, \dots, f_n sont typés par t_1, \dots, t_n :

$$\begin{aligned}
 Rtpf1 &= prj1(T1 \times \dots \times T_{n-1}, T_n); \dots; prj1(T1, T2) \\
 \dots & \\
 Rtpfi &= prj1(T1 \times \dots \times T_{n-1}, T_n); \dots; prj1(T1 \times \dots \times T_i, T_{i+1}); prj2(T1 \times \dots \times T_{i-1}, T_i) \\
 \dots & \\
 Rtpfn &= prj2(T1 \times \dots \times T_{n-1}, T_n)
 \end{aligned}$$

- Opérations d'écriture :

$$\begin{aligned}
 Wtpf1 &= \lambda s, v \cdot (s, v \in TP \times T1 \mid v \mapsto Rtpf2(s) \mapsto \dots \mapsto Rtpf2(s)) \\
 &= (\dots (prj2(TP, T_1) \otimes (prj1(TP, T_1); Rtpf2)) \otimes \dots) \otimes (prj1(TP, T_1); Rtpfn) \\
 \dots & \\
 Wtpfi &= (\dots (\dots (prj1(TP, T_i); Rtpf1) \otimes (prj1(TP, T_i); Rtpf2)) \otimes \dots) \\
 &\quad \otimes prj2(TP, T_i) \otimes \dots) \otimes (prj1(TP, T_i); Rtpfn) \\
 \dots & \\
 Wtpfn &= (((prj1(TP, T_n); Rtpf1) \otimes (prj1(TP, T_n); Rtpf2)) \otimes \dots) \otimes prj2(TP, T_n)
 \end{aligned}$$

Remarque 6.3 Une modélisation par produit cartésien construite selon une démarche dichotomique permet une définition plus compacte des fonctions d'accès aux champs d'une structure.

Exemple 6.4 (Modélisation d'une structure de structure) A titre d'illustration, reprenons l'exemple précédent 6.3. La première étape consiste à traduire la structure `pt` ; ce type étant inhabité, la traduction se limite à la définition en `B` de l'ensemble des valeurs et des fonctions d'accès aux champs :

```

CONSTANTS PT, Rptx, Rpty, Wptx, Wpty
PROPERTIES
  PT = SINT × SINT  ∧
  Rptx = prj1(SINT, SINT)  ∧
  Rpty = prj2(SINT, SINT)  ∧
  Wptx = prj2(PT, SINT) ⊗ (prj1(PT, SINT); Rpty)  ∧
  Wpty = (prj1(PT, SINT); Rptx) ⊗ prj2(PT, SINT)

```

On peut ensuite traduire le type `rect` et ses instances globales :

```

SETS IDRECT = {r1, r2}
CONSTANTS RECT, Rrectp1, Rrectp2, Rrectcol, Wrectp1, Wrectp2, Wrectcol
PROPERTIES
  RECT = (PT × PT) × SCHAR  ∧
  Rrectp1 = prj1(PT × PT, SCHAR); prj1(PT, PT)  ∧
  Rrectp2 = prj1(PT × PT, SCHAR); prj2(PT, PT)  ∧
  Rrectcol = prj2(PT × PT, SCHAR)  ∧
  Wrectp1 = (prj2(RECT, PT) ⊗ (prj1(RECT, PT); Rrectp2))
             ⊗ (prj1(RECT, PT); Rrectcolor)  ∧
  Wrectp2 = ((prj1(RECT, PT); Rrectp1) ⊗ prj2(RECT, PT))
             ⊗ (prj1(RECT, PT); Rrectcolor)  ∧
  Wrectcol = ((prj1(RECT, SCHAR); Rrectp1) ⊗ (prj1(RECT, SCHAR); Rrectp2))
             ⊗ prj2(RECT, SCHAR)
VARIABLES valrect
INVARIANT valrect ∈ IDRECT ↔ RECT
INITIALISATION valrect := ∅
OPERATIONS
  Wrect(id, r) ≐ PRE id ∈ IDRECT ∧ r ∈ RECT THEN valrect(id) := r END ;
  out ← Rrect(id) ≐ PRE id ∈ dom(valrect) THEN out := valrect(id) END

```

Avec cette modélisation, il est possible d'abstraire l'ensemble des opérations qui peuvent être effectuées sur une structure.

Exemple 6.5 (Accès aux champs d'une structure) Si `s` est du type structure `str` :

```

x=s.f   $\xrightarrow{B}$   VAR tmp IN tmp ← Rstr(s); x := Rstrf(tmp) END
s.f=x   $\xrightarrow{B}$   VAR tmp IN tmp ← Rstr(s); Wstr(s, Wstrf(tmp, x)) END

```

6.3.3 Problème des initialisations partielles

L'une des caractéristique de la modélisation proposée dans le paragraphe 6.1 est d'utiliser des fonctions de valuations partielles permettant de représenter l'état des variables non initialisées.

En l'occurrence, cependant, cette modélisation peut être la source de difficultés dans le cas de variables structurées, comme dans l'exemple très simple suivant :

Exemple 6.6 (Initialisation partielle d'une structure)

```

struct s {int f1;int f2};
struct s test;
test.f1=1;

```

En effet, nous sommes ici confrontés à un phénomène d'initialisation partielle, qui ne peut être représenté dans le modèle actuel, pour lequel l'objet `test` soit n'a pas de valeur, soit a une valeur pour chacun de ses champs.

Si on veut conserver la modélisation proposée pour les structures, une des solutions possible est de ne plus définir la fonction de valuation comme une fonction partielle, mais bien comme une fonction totale. L'initialisation de cette fonction est alors réalisée par une substitution non déterministe, de la forme :

$$valtype : \in IDTYPE \rightarrow TYPE$$

Remarque 6.4 *Pour prendre en compte l'initialisation de certaines variables, on peut utiliser la forme plus générale suivante :*

$$valtype : \in \{f \mid f \in IDTYPE \rightarrow TYPE \wedge f(id_i) = v_i \wedge \dots \wedge f(id_j) = v_j\}$$

Cette modélisation est un peu moins stricte, puisqu'il devient possible de lire une variable non initialisée ; cependant la valeur retournée dans ce cas est non spécifiée, et il reste délicat de prouver une propriété intéressante³.

Par souci de cohérence, nous appliquons ce principe non seulement aux structures, mais aussi à l'ensemble des types ; les fonctions de valuation seront donc toujours dans la suite des fonctions totales.

6.4 Modélisation des unions

Les types union du C sont des objets dangereux, permettant notamment de faire du *type-punning*, comme dans l'exemple suivant :

```

union mlt {int i;double d;}
int fn_mlt() {mlt a;a.d=3.0;return a.i;}

```

Il est possible de proposer un modèle adapté à ce type de programmation – en modélisant finement les interprétations (les casts) entre les différents types et en retenant pour l'union la représentation la plus détaillée (pouvant coder toutes les autres). Nous choisissons cependant de ne pas prendre en compte ce type de programmation dans le cadre de cette étude.

Modélisation 7 *Seuls les programmes sans type-punning sont considérés.*

Dès lors, notre modèle décrit une instance de type union comme un objet dont le type peut changer en cours d'exécution, mais uniquement lors d'une écriture ; une lecture ne peut que retourner le type (et la valeur) de la dernière écriture.

En s'inspirant de ce qui est proposé dans le paragraphe 6.3.1, on propose donc une modélisation combinant un identifiant et un ensemble de fonctions de valuations, mais pas d'ensemble des valeurs. Pour le cas très spécifique des unions, nous décidons de définir ces fonctions de valuations comme

³Notons également que cette approche décrit assez bien ce qui se passe en C .

des fonctions partielles, un invariant assurant qu'une seule de ces fonctions, au plus, ait une valeur associée à un identifiant donné. Les opérations d'écriture devront toujours définir la nouvelle valeur associée à l'identifiant pour une des valuations, et supprimer les autres valuations.

La modélisation générale d'une union est donc la suivante (les types t_1, \dots, t_n , éventuellement inhabités, sont supposés modélisés par ailleurs) :

```

union u { t1 f1;
        ...
        ti fi           $\xrightarrow{B}$ 
        ...
        tn fn; }
union u x,y;

SETS IDU = {x,y}
VARIABLES valut1,...,valuti,...,valutn
INVARIANT
  valut1 ∈ IDU ↔ T1  ∧ ... ∧  valuti ∈ IDU ↔ Ti  ∧ ... ∧  valutn ∈ IDU ↔ Tn  ∧
  ... ∧  dom(valut1) ∩ dom(valuti) = ∅  ∧ ... ∧  dom(valut1) ∩ dom(valutn) = ∅  ∧
  ... ∧  dom(valuti) ∩ dom(valutn) = ∅  ∧ ...
INITIALISATION
  valut1 := ∅ || ... || valuti := ∅ || ... || valutn := ∅
OPERATIONS
  Wut1(j,v) ≐ PRE j ∈ IDU ∧ v ∈ T1 THEN
    valut1(j) := v || ... || valuti := {j} ≪ valuti || ... || valutn := {j} ≪ valutn
  END ;
  out ← Rut1(j) ≐ PRE j ∈ dom(valut1) THEN out := valut1(j) END ;
  ...
  Wuti(j,v) ≐ PRE j ∈ IDU ∧ v ∈ Ti THEN
    valut1 := {j} ≪ valut1 || ... || valuti(j) := v || ... || valutn := {j} ≪ valutn
  END ;
  out ← Ruti(j) ≐ PRE j ∈ dom(valuti) THEN out := valuti(j) END ;
  ...
  Wutn(j,v) ≐ PRE j ∈ IDU ∧ v ∈ Tn THEN
    valut1 := {j} ≪ valut1 || ... || valuti := {j} ≪ valuti || ... || valutn(j) := v
  END ;
  out ← Rutn(j) ≐ PRE j ∈ dom(valutn) THEN out := valutn(j) END

```

Remarque 6.5 (Quantification sur les fonctions) *Les fonctions, en B , sont des relations, donc des ensembles, donc des objets du premier ordre sur lesquels il est possible de quantifier.*

Cependant, les contraintes de typage interdisent de quantifier sur des fonctions dont les signatures sont différentes, puisqu'elles ne sont pas de même type (le type d'une fonction est l'ensemble de toutes les fonctions ayant la même signature, i.e. ayant le même type de domaine et le même type de co-domaine).

Il n'y a donc pas d'expression plus compacte de l'invariant d'exclusion mutuelle ci-dessus.

Modélisation 8 *Les types unions ne sont pas considérés.*

6.5 Modélisation des pointeurs

Les modélisations en B pour les variables et les structures, telles que proposées pour l'instant, permettent d'associer une valeur à un identifiant. L'utilisation de pointeurs en C de manipuler un

ensemble de valeur d'une cardinalité finie, mais arbitrairement grande; c'est par exemple le cas des algorithmes de listes, ou l'utilisation d'une unique variable pointeur permet de mémoriser un nombre quelconque de valeurs.

Une nouvelle modélisation B est donc nécessaire pour prendre en compte les pointeurs du C ; plus que de représenter l'adresse mémorisée par le pointeur, elle doit permettre de modéliser la mémoire utilisée par le code C considéré.

6.5.1 Modélisation de la mémoire

Le pointeur est une variable mémorisant non pas la valeur, mais une adresse mémoire. Formellement, les pointeurs sont tous du type *adresse*; il n'y a d'ailleurs pas de type de pointeur en C , mais uniquement des instances de pointeur sur un type. Le type pointé permet d'indiquer la représentation associée à la valeur pointée.

A l'instanciation d'un pointeur, l'adresse est non déterminée ou `NULL`⁴; quoi qu'il en soit un tel pointeur ne peut être utilisé sans une allocation mémoire effective.

La principale fonction d'allocation offerte par le C est la fonction `malloc`; elle admet en paramètre la taille mémoire à réserver, en octets et retourne un pointeur sur la zone allouée. Lui est associée la fonction de désallocation `free`, qui prend en paramètre un pointeur retourné par un appel à la fonction `malloc` – l'absence d'un paramètre de taille indique que celle-ci est mémorisée lors de l'allocation.

Nous devons donc être capable de représenter dans une abstraction l'état de la mémoire, i.e. à la fois son contenu et les allocations. Comme précédemment, les contraintes de typage imposent de définir une allocation par type; cela se traduit par de nouveaux choix de modélisation :

Modélisation 9 *Chaque type manipulé est supposé disposer d'un espace mémoire propre et infini.*

Modélisation 10 *Les manipulations de types (cast) par pointeurs ne sont pas considérées.*

Pour un type `tp` en C nous définissons donc les entités B suivantes modélisant la mémoire associée à ce type :

```

VARIABLES memtp, lsttp
INVARIANT memtp ∈ ℕ1 ↔ TP  ∧  lsttp ∈ ℕ1 ↔ ℕ1  ∧
  ∀ a · (a ∈ dom(lsttp) ⇒ a ≤ lsttp(a))  ∧
  ∀ a · (a ∈ dom(lsttp) ⇒ a..lsttp(a) ∩ dom(lsttp) = {a})  ∧
  ∀ a · (a ∈ dom(lsttp) ⇒ a..lsttp(a) ⊆ dom(memtp))  ∧
  ∀ a · (a ∈ dom(memtp) ⇒ ∃ b · (b ∈ dom(lsttp) ∧ a ∈ b..lsttp(b)))
INITIALISATION memtp := ∅ || lsttp := ∅
OPERATIONS
  out ← malloctp(s) ≐ PRE s ∈ ℕ1 THEN
    ANY a, v WHERE a ∈ ℕ1  ∧
      ∀ b · (b ∈ dom(lsttp) ⇒ a..a + s - 1 ∩ b..lsttp(b) = ∅)  ∧
      v ∈ a..(a + s - 1) → TP
    THEN
      out := a || lsttp(a) := a + s - 1 || memtp := memtp <+ v
    END
  END ;
  freectp(a) ≐ PRE a ∈ dom(lsttp) THEN
    memtp := a..lsttp(a) <- memtp || lsttp := {a} <- lsttp
  END

```

⁴Cette valeur particulière ne doit pas être interprétée comme l'adresse 0, mais comme l'absence d'adresse.

L'unité utilisée pour les adresses est l'instance de type TP , et non pas l'octet.

Si une adresse a correspond au début d'une allocation, $lsttp(a)$ est la dernière adresse de cette allocation ; si a n'est pas le début d'une allocation, $lsttp(a)$ n'est pas définie. La fonction $memtp$ retourne pour une adresse la valeur mémorisée à cette adresse.

L'invariant assure qu'il n'y a aucun recouvrement entre deux allocations. De plus, la spécification de l'opération $malloctp$ utilise une substitution non déterministe pour le choix de l'adresse (dans le respect de l'invariant) ; il n'est donc pas possible d'exprimer une propriété entre les adresses de deux allocations.

L'invariant impose également qu'une adresse mémoire est utilisée si et seulement si cette adresse mémoire appartient à une allocation. Notons qu'une zone mémoire allouée mais non initialisée contient bien une valeur, mais que cette valeur n'est pas déterminée (cf. l'opération $malloctp$) ; cette spécification correspond au choix de modélisation proposé dans le paragraphe 6.3.3, permettant d'éviter les problèmes posés par les initialisations partielles.

6.5.2 Modélisation des variables pointeurs

Une fois définie la modélisation de la mémoire, nous pouvons proposer une abstraction pour les pointeurs ; ce sont ici des variables de type \mathbb{N} (nous nommons cependant STP le type des pointeurs sur le type tp), pour lesquelles nous utilisons la représentation définie précédemment, enrichie d'opérations d'accès correspondant à la lecture et l'écriture de la valeur pointée. Notons également que, pour des questions de clarté, nous renomons $adrstp$ la fonction de valuation qui, ici, retourne une adresse :

```

SETS  $IDSTP = \{...\}$ 
VARIABLES  $adrstp$ 
INVARIANT  $adrstp \in IDSTP \rightarrow \mathbb{N}$ 
INITIALISATION  $adrstp := IDSTP \times \{0\}$ 
OPERATIONS
   $Wstp(j, v) \hat{=} \text{PRE } j \in IDSTP \wedge v \in \mathbb{N} \text{ THEN } adrstp(j) := v \text{ END ;}$ 
   $out \leftarrow Rstp(j) \hat{=} \text{PRE } j \in IDSTP \text{ THEN } out := adrstp(j) \text{ END ;}$ 
   $WSstp(j, v) \hat{=} \text{PRE } j \in IDSTP \wedge adrstp(j) \in \text{dom}(memtp) \wedge v \in TP \text{ THEN}$ 
     $memtp(adrstp(j)) := v$ 
   $\text{END ;}$ 
   $out \leftarrow RSstp(j) \hat{=} \text{PRE } j \in IDSTP \wedge adrstp(j) \in \text{dom}(memtp) \text{ THEN}$ 
     $out := memtp(adrstp(j))$ 
   $\text{END}$ 

```

Notons qu'il est tout à fait possible ici de définir la fonction de valuation $adrstp$ comme une fonction partielle (le problème de l'initialisation partielle ne se posant pas) ; cependant cette modélisation n'est pas beaucoup plus intéressante puisque la valeur prédéfinie $NULL$, correspondant à l'adresse 0, joue pratiquement le même rôle⁵. $adrstp$ est donc bien une fonction totale, initialisant tous les pointeurs à $NULL$.

6.5.3 Opérateur de déréférencement

L'opérateur de déréférencement du C , noté $\&$, a la particularité de transformer une variable classique en pointeur ; il permet donc de violer la propriété selon laquelle l'espace alloué à une variable n'est utilisé par aucun autre objet – phénomène connu sous le nom d'*aliasing*.

⁵On peut lire un pointeur non initialisé, mais pas la valeur pointée.

Remarque 6.6 (Aliasing et constantes) *L'aliasing permet, en C, de violer le spécificateur `const` attaché à une déclaration, comme dans l'exemple suivant :*

```
const char cst=1;
char * ptr=&cst;
*ptr=2;
```

Nous considérons que ce type de code est incorrect; il est cependant modélisé par le processus d'abstraction qui ne distingue pas les constantes des variables.

Formellement, l'opérateur de déréférencement impose que notre modélisation puisse associer à toute variable son adresse d'allocation. Il n'est cependant pas nécessaire de modifier ou d'enrichir notre modèle; la solution la plus simple est encore de transformer toutes les variables du code *C* analysé en pointeurs!

Ce résultat peut s'obtenir par pre-cooking, selon les schémas suivants où `tp` est un type et `id`, `id_1`, `id_2` des identifiants de variable ne commençant ni par `*` ni par `&` :

```
tp id;  $\xrightarrow{P_c}$  tp * const id=malloc(1*sizeof(tp));

id  $\xrightarrow{P_c}$  *id

&id  $\xrightarrow{P_c}$  id
```

Remarque 6.7 (Conservation des types) *Les règles de pre-cooking précédentes ne sont applicables qu'aux instances concrètes; elle ne doivent notamment pas être appliquées aux définitions de structure ou aux prototypes de fonctions.*

Il est intéressant de noter que le code *C* obtenu par pre-cooking est tel que pour un type `tp` quelconque, il n'y a jamais d'instance; les seuls objets manipulés sont toujours des pointeurs. Pour mémoire, les notations proposées dans le paragraphe 6.5.2 proposait de nommer *IDTP* l'ensemble des identifiants des variables de type `tp`, et *IDSTP* l'ensemble des pointeurs de type `tp`; cette distinction n'a plus lieu d'être, ce qui permet de simplifier le nommage des ensembles, fonctions et opérations manipulées en supprimant le *s* préfixant le nom du type.

Modélisation 11 *Les déréférencements de pointeurs ne sont pas considérés.*

6.5.4 Arithmétique des pointeurs

Le *C* définit les opérations d'arithmétique sur les pointeurs suivantes :

- Incrémentation et décrémentation d'un pointeur non NULL.
- Addition ou soustraction d'un pointeur non NULL et d'un entier, retournant un pointeur.
- Différence de deux pointeurs sur une même allocation, retournant un entier.
- Comparaison (`<`, `≤`, `≥`, `>`) de deux pointeurs sur une même allocation, retournant un booléen.
- Egalité (`=`, `≠`) de deux pointeurs, retournant un booléen.

Bien que les pointeurs soient caractérisés comme étant de type \mathbb{N} dans notre modélisation, il est intéressant de modéliser ces opérations, en particulier afin de mettre en évidence les pré-conditions.

Afin d'alléger cette modélisation, nous ne représentons qu'un minimum d'opérations, suffisant cependant pour modéliser toutes les opérations possibles; notamment les opérations d'incrément, de décrémentation, de soustraction d'un entier sont toutes ramenées à l'opération d'addition d'un entier⁶.

⁶Nous parlons ici d'entiers formels, et non d'entiers implémentables; la soustraction de la plus petite valeur pourra bien être représentée, malgré la dissymétrie de la représentation en complément à 2.

L'arithmétique des pointeurs est donc modélisée par les opérations suivantes, complémentaires à celle proposées dans le paragraphe 6.5.2 :

OPERATIONS

```

out ← sumtp(j, i) ≐ PRE j ∈ IDTP ∧ i ∈ ℤ ∧ adrtp(j) + i >= 0 THEN
    out := adrtp(j) + i
END ;
out ← diftp(j, k) ≐ PRE j ∈ IDTP ∧ k ∈ IDTP ∧
    ∃ b · (b ∈ dom(lsttp) ∧ adrtp[{j, k}] ⊆ b.lsttp(b)) THEN
    out := adrtp(j) - adrtp(k)
END ;
out ← grtp(j, k) ≐ PRE j ∈ IDTP ∧ k ∈ IDTP ∧
    ∃ b · (b ∈ dom(lsttp) ∧ adrtp[{j, k}] ⊆ b.lsttp(b)) THEN
    out := bool(adrtp(j) > adrtp(k))
END ;
out ← eqltp(j, k) ≐ PRE j ∈ IDTP ∧ k ∈ IDTP THEN
    out := bool(adrtp(j) = adrtp(k))
END ;
out ← eqntp(j) ≐ PRE j ∈ IDTP THEN
    out := bool(adrtp(j) = 0)
END

```

Remarque 6.8 (Pré-conditions sur l'arithmétique des pointeurs) *L'opération d'addition, telle que définie, peut modifier un pointeur en lui faisant quitter son allocation, voire le faire pointer sur une autre allocation. Cependant, un tel comportement peut être jugé légitime puisque pouvant correspondre dans un code C à un cas limite ; une pré-condition interdisant un tel débordement serait trop contraignante.*

De manière générale, l'opération d'addition a été spécifiée de manière à ce que, avec $i > 0$, $(\text{ptr}+i)-i$ soit toujours valide ; il s'agit d'admettre des codes dans lesquels il y a incrémentation d'un pointeur, vérification et, en cas de débordement, décrémentation pour revenir à une situation valide. C'est aussi la raison pour laquelle l'incrémentation d'un pointeur nul est autorisée (il est impossible de prouver que la décrémentation d'un pointeur valide ne donne pas 0).

Le débordement est donc toléré, mais le modèle assure toutefois que dans un tel cas une lecture ou une écriture est invalide. Plus exactement, le modèle est tel qu'il est impossible de prouver, dans le cas d'un débordement, qu'une lecture ou une écriture respecte l'invariant.

6.5.5 Modélisation des pointeurs de pointeurs

En C, il est possible de faire des déclarations de la forme `tp **id` ; avec une telle déclaration, `**id` est une valeur de type `tp`, `*id` un pointeur sur `tp` (auquel on doit associer une allocation de taille `tp`), et `id` un pointeur sur pointeur (auquel on doit associer une allocation de taille adresse).

Moralement, la déclaration `type **id` définit un pointeur sur le type (possiblement inhabité) `type *` ; la modélisation doit donc définir l'ensemble des objets associés au type `tp`, mais aussi des objets complémentaires correspondant aux pointeurs de pointeurs sur `tp`, pour lequel on introduit la notation *ptp*.

Formellement, il nous faut une nouvelle mémoire avec sa gestion des allocations, et en plus des opérations d'accès direct et indirect, nous devons prévoir les opérations d'accès doublement indirect ; notons que les opérations *Rtp* (accès à l'adresse mémorisée par un pointeur sur `tp`) et *RSptp* (accès à la valeur pointée par un pointeur de pointeur sur `tp`) retournent toutes les deux un pointeur sur `tp`, mais sont différentes - essentiellement, la seconde demande en paramètre un identifiant formel qui a toutes les chances de ne pas exister.

Les objets complémentaires modélisant les pointeurs de pointeurs sur \mathbf{tp} sont donc :

```

SETS  $IDPTP = \{ \dots \}$ 
VARIABLES  $memtp, lsttp, adrtp$ 
INVARIANT  $memtp \in \mathbb{N}_1 \leftrightarrow \mathbb{N} \wedge lsttp \in \mathbb{N}_1 \leftrightarrow \mathbb{N}_1 \wedge$ 
 $adrtp \in IDPTP \rightarrow \mathbb{N} \wedge$ 
 $\forall a \cdot (a \in \text{dom}(lsttp) \Rightarrow a \leq lsttp(a)) \wedge$ 
 $\forall a \cdot (a \in \text{dom}(lsttp) \Rightarrow a..lsttp(a) \cap \text{dom}(lsttp) = \{a\}) \wedge$ 
 $\forall a \cdot (a \in \text{dom}(lsttp) \Rightarrow a..lsttp(a) \subseteq \text{dom}(memtp)) \wedge$ 
 $\forall a \cdot (a \in \text{dom}(memtp) \Rightarrow \exists b \cdot (b \in \text{dom}(lsttp) \wedge a \in b..lsttp(b)))$ 
INITIALISATION  $memtp := \emptyset \parallel lsttp := \emptyset \parallel adrtp := IDPTP \times \{0\}$ 
OPERATIONS
 $out \leftarrow malloctp(s) \hat{=} \text{PRE } s \in \mathbb{N}_1 \text{ THEN}$ 
 $\quad \text{ANY } a, v \text{ WHERE } a \in \mathbb{N}_1 \wedge$ 
 $\quad \quad \forall b \cdot (b \in \text{dom}(lsttp) \Rightarrow a..(a + s - 1) \cap b..lsttp(b) = \emptyset) \wedge$ 
 $\quad \quad v \in a..(a + s - 1) \rightarrow \mathbb{N}$ 
 $\quad \text{THEN}$ 
 $\quad \quad out := a \parallel lsttp(a) := a + s - 1 \parallel memtp := memtp \triangleleft v$ 
 $\quad \text{END}$ 
 $\text{END ;}$ 
 $freept(a) \hat{=} \text{PRE } a \in \text{dom}(lsttp) \text{ THEN}$ 
 $\quad memtp := a..lsttp(a) \triangleleft memtp \parallel lsttp := \{a\} \triangleleft lsttp$ 
 $\quad \text{END ;}$ 
 $out \leftarrow sumtp(j, i) \hat{=} \text{PRE } j \in IDPTP \wedge i \in \mathbb{Z} \wedge adrtp(j) + i \geq 0 \text{ THEN}$ 
 $\quad out := adrtp(j) + i$ 
 $\quad \text{END ;}$ 
 $out \leftarrow diftp(j, k) \hat{=} \text{PRE } j \in IDPTP \wedge k \in IDPTP \wedge$ 
 $\quad \exists b \cdot (b \in \text{dom}(lsttp) \wedge adrtp[\{j, k\}] \subseteq b..lsttp(b)) \text{ THEN}$ 
 $\quad out := adrtp(j) - adrtp(k)$ 
 $\quad \text{END ;}$ 
 $out \leftarrow gtrtp(j, k) \hat{=} \text{PRE } j \in IDPTP \wedge k \in IDPTP \wedge$ 
 $\quad \exists b \cdot (b \in \text{dom}(lsttp) \wedge adrtp[\{j, k\}] \subseteq b..lsttp(b)) \text{ THEN}$ 
 $\quad out := \text{bool}(adrtp(j) > adrtp(k))$ 
 $\quad \text{END ;}$ 
 $out \leftarrow eqltp(j, k) \hat{=} \text{PRE } j \in IDPTP \wedge k \in IDPTP \text{ THEN}$ 
 $\quad out := \text{bool}(adrtp(j) = adrtp(k))$ 
 $\quad \text{END ;}$ 
 $out \leftarrow eqnpt(j) \hat{=} \text{PRE } j \in IDPTP \text{ THEN}$ 
 $\quad out := \text{bool}(adrtp(j) = 0)$ 
 $\quad \text{END ;}$ 
 $Wptp(j, v) \hat{=} \text{PRE } j \in IDPTP \wedge v \in \mathbb{N} \text{ THEN } adrtp(j) := v \text{ END ;}$ 
 $out \leftarrow Rptp(j) \hat{=} \text{PRE } j \in IDPTP \text{ THEN } out := adrtp(j) \text{ END ;}$ 
 $WSptp(j, v) \hat{=} \text{PRE } j \in IDPTP \wedge v \in \mathbb{N} \wedge adrtp(j) \in \text{dom}(memtp) \text{ THEN}$ 
 $\quad memtp(adrtp(j)) := v$ 
 $\quad \text{END ;}$ 
 $out \leftarrow RSptp(j) \hat{=} \text{PRE } j \in IDPTP \wedge adrtp(j) \in \text{dom}(memtp) \text{ THEN}$ 
 $\quad out := memtp(adrtp(j))$ 
 $\quad \text{END ;}$ 
 $WSSptp(j, v) \hat{=} \text{PRE } j \in IDPTP \wedge v \in TP \wedge$ 
 $\quad adrtp(j) \in \text{dom}(memtp) \wedge$ 
 $\quad memtp(adrtp(j)) \in \text{dom}(memtp) \text{ THEN}$ 
 $\quad memtp(memtp(adrtp(j))) := v$ 
 $\quad \text{END ;}$ 
 $out \leftarrow RSSptp(j) \hat{=} \text{PRE } j \in IDPTP \wedge$ 
 $\quad adrtp(j) \in \text{dom}(memtp) \wedge$ 
 $\quad memtp(adrtp(j)) \in \text{dom}(memtp) \text{ THEN}$ 
 $\quad out := memtp(memtp(adrtp(j)))$ 
 $\quad \text{END}$ 

```

On peut, de manière évidente, poursuivre le raisonnement en prenant en compte les pointeurs de pointeurs de pointeurs, et ainsi de suite ; cependant, à chaque niveau supplémentaire de référencement, on doit complexifier le modèle. Dans le cadre de cette étude, on se limitera donc aux pointeurs de pointeurs.

Modélisation 12 *Les référencements d'ordre supérieur à 2 ne sont pas considérés.*

En présence dans un code C de déclarations de la forme tp **pp et tp *pt , les schémas de traductions seront les suivants :

$$\begin{array}{lcl}
 \text{pp=malloc(n*sizeof(tp *))} & \xrightarrow{B} & \begin{array}{l} \text{VAR tmp IN} \\ \text{tmp} \leftarrow \text{malloctp}(n) \\ \text{Wptp}(pp, tmp) \\ \text{END} \end{array} \\
 \\
 \text{*pp=malloc(1*sizeof(tp))} & \xrightarrow{B} & \begin{array}{l} \text{VAR tmp IN} \\ \text{tmp} \leftarrow \text{malloctp}(n) \\ \text{WSptp}(pp, tmp) \\ \text{END} \end{array} \\
 \\
 \text{pp=v} & \xrightarrow{B} & \text{Wptp}(pp, v) \\
 \\
 \text{*pp=v} & \xrightarrow{B} & \text{WSptp}(pp, v) \\
 \\
 \text{**pp=v} & \xrightarrow{B} & \text{WSSptp}(pp, v) \\
 \\
 \text{*pp=pt} & \xrightarrow{B} & \begin{array}{l} \text{VAR tmp IN} \\ \text{tmp} \leftarrow \text{Rtp}(pt) \\ \text{WSptp}(pp, tmp) \\ \text{END} \end{array}
 \end{array}$$

Remarque 6.9 (Construction modulaire) *La démarche proposée ici est de fusionner le modèle des pointeurs avec le modèle des pointeurs de pointeurs ; on obtient donc une spécification unique qui déclare simultanément les variables memtp et memptp , qui peuvent être modifiées par les opérations du modèle, dont WSSptp .*

Une autre approche consiste à définir la spécification tp_s des pointeurs et de l'inclure dans la spécification ptp_s des pointeurs de pointeurs, les opérations de tp_s étant promues dans ptp_s .

La variable memtp est alors externe à ptp_s , et ne peut être modifiée ; on doit donc ajouter à tp_s une opération directmemwrite d'accès direct à la mémoire, par adresse et non par identifiant, utilisable par l'opération WSSptp . L'utilisation de directmemwrite doit être strictement limitée à ptp_s , en particulier cette opération ne doit pas être promue.

Remarque 6.10 (Modélisation alternative) *Pour simplifier les modèles et les notations, il est également possible, par pre-cooking, de transformer les variables et les pointeurs en pointeurs de pointeurs, pour ne manipuler qu'un unique type d'objet. On peut dans ce cas traiter les déréférencements de pointeurs.*

Remarque 6.11 (Modélisation alternative) *Une autre simplification de la modélisation pourrait être de créer, pour tout identifiant id de pointeur de pointeur, un identifiant Sid correspondant à *id , pour pouvoir réutiliser au mieux la modélisation des pointeurs ; en particulier, il ne serait alors plus nécessaire de définir les opérations WSptp , RSptp , WSSptp , RSSptp , correspondant respectivement à Wtp , Rtp , WStp et RStp .*

6.5.6 Limitations de modélisation des pointeurs

Modélisation 13 *Les pointeurs de fonctions ne sont pas considérés.*

Remarque 6.12 (Manipulation de fonctions en B) *Comme indiqué dans la remarque 6.5, les fonctions sont, en B , des objets du premier ordre ; on peut envisager de représenter de manière limitée les pointeurs de fonctions. Il convient cependant de noter que toute modélisation devra respecter les contraintes de typage (les fonctions sont typées par leur signature), et que les fonctions C sont abstraites sous forme d'opérations B dont la modélisation par une fonction formelle peut être délicate.*

Modélisation 14 *Les pointeurs génériques définis par `void *` ne sont pas considérés.*

6.6 Modélisation des tableaux

6.6.1 Tableaux à une dimension

Formellement, un tableau est un objet statique, dont la taille est fixée à l'allocation ; le traitement de la déclaration d'un tableau est différent du traitement d'une allocation mémoire par un `malloc`.

En pratique cependant, il n'est pas possible de distinguer à l'exécution un tableau d'un pointeur sur une allocation mémoire ; la documentation du langage C confond d'ailleurs fréquemment les deux structures⁷. On a les équivalences observationnelles suivantes :

```
type tab1D[n]  ≡ type * const tab1D=malloc(n*sizeof(type))
tab1D[0]      ≡ *tab1D
tab1D[i]      ≡ *(tab1D+i)
```

Dans le processus d'abstraction, on se propose donc de ne pas modéliser les tableaux unidimensionnels ; par pre-cooking du code C , on peut en effet transformer les tableaux en pointeurs, et réutiliser la modélisation présentée dans la section 6.5.

Le principe de ces règles de pre-cooking est le suivant :

```
tp id[n];       $\xrightarrow{P_c}$  tp *id=malloc(n*sizeof(tp));
id[i]          $\xrightarrow{P_c}$  *(id+i)
&id[i]         $\xrightarrow{P_c}$  (id+i)
```

Remarque 6.13 *La modélisation la plus immédiate, en B , d'un tableau, est une fonction partielle dont le paramètre est l'indice du tableau. Cette modélisation n'est pas retenue, afin de limiter le nombre de représentation et de pouvoir traiter les opérations de pointeurs sur les tableaux.*

6.6.2 Tableaux à plusieurs dimensions

Le cas des tableaux à plusieurs dimensions est un peu plus délicat ; en C , la déclaration d'un tableau de dimension n correspond moralement à la déclaration d'un tableau de tableaux de dimension $n-1$. En particulier, c'est un pointeur de tableaux⁸.

⁷La norme spécifie les différences entre les deux objets, mais ces différences relèvent plus de l'architecture système et concernent essentiellement les développeurs de compilateurs.

⁸Et non un pointeur de pointeurs, la grande différence se situant au niveau de l'arithmétique des pointeurs.

A titre d'exemple, pour les tableaux à deux dimensions on a les équivalences observationnelles suivantes :

$$\begin{aligned} \text{tab2D}[0][0] &\equiv **\text{tab2D} \\ \text{tab2D}[i][j] &\equiv *((\text{tab2D}+i)+j) \end{aligned}$$

La représentation en C des tableaux multi-dimensionnels est donc complexe.

Nous choisissons de modéliser en B les tableaux à n dimensions plus simplement, en les ramenant à des tableaux à une dimension; un tableau de la forme `tp id[n][m]` est considéré équivalent à un tableau de la forme `tp id[n*m]`.

Encore une fois, cette transformation est effectuée par pre-cooking, et ne nécessite aucune modification de la modélisation. Les règles de pre-cooking doivent nous permettre non seulement de réécrire les expressions tabulaires sous la forme d'expressions de pointeurs, mais également de simplifier la représentation des tableaux multi-dimensionnels en réécrivant les expressions de la forme `tab[i][j]` ou `*(*(tab+i)+j)`.

Le principe des règles pour les tableaux à deux dimensions est donné ci-dessous :

$$\begin{aligned} \text{tp id}[n][m]; &\xrightarrow{P_c} \text{tp } *id=\text{malloc}(n*m*\text{sizeof}(tp)); \\ id[i][j] &\xrightarrow{P_c} *(id+i+n*j) \\ \&id[i][j] &\xrightarrow{P_c} (id+i+n*j) \\ *(id+i)+j &\xrightarrow{P_c} (id+i+n*j) \end{aligned}$$

Remarque 6.14 (Contraintes sur les indices) *Cette modélisation fait disparaître les dimensions réelles d'un tableau multi-dimensionnel. On peut cependant envisager, pendant le pre-cooking, de décorer le code C à l'aide d'éléments supplémentaires, sous formes de commentaires, notamment de pré-conditions sur les indices, remontées dans les abstractions.*

Ainsi, une déclaration `int tab[10];` suivie d'une expression contenant `tab[i]` génère une pré-condition $i \in 0..9$.

On peut poursuivre, si nécessaire, et proposer des règles pour un nombre quelconque de dimensions.

Modélisation 15 *Seuls les tableaux uni-dimensionnels ou bi-dimensionnels sont considérés.*

6.7 Problématiques résiduelles et simplifications

6.7.1 Initialisation des variables

On considère le code C suivant :

$$\begin{aligned} \text{int } v=0; \text{int } w=1; &\xrightarrow{P_c} \begin{aligned} &\text{int } *v=\text{malloc}(1*\text{sizeof}(\text{int})); \\ &*v=0; \text{int } *w=\text{malloc}(1*\text{sizeof}(\text{int})); \\ &*w=1; \end{aligned} \end{aligned}$$

`*v` est un pointeur sur `int`, on exploite donc le modèle *sint_s* (dans sa version habitée); en particulier, on a $v \in IDSINT$.

L'initialisation de *memsint*, *lastsint* et *adrsint* peut être décrite par la substitution suivante :

$$\begin{aligned} \text{VAR } tmp \text{ IN } tmp \leftarrow \text{mallocsint}(1); \text{adrsint}(v) := tmp; \text{memsint}(\text{adrsint}(v)) := 0 \text{ END} ; \\ \text{VAR } tmp \text{ IN } tmp \leftarrow \text{mallocsint}(1); \text{adrsint}(w) := tmp; \text{memsint}(\text{adrsint}(w)) := 1 \text{ END} \end{aligned}$$

Cette initialisation modifie $adrsint$ et $memsint$, ce qui ne pose pas de problème, ces variables étant définies dans la machine $sint_s$. Cependant, cette substitution d'initialisation, si elle est acceptable dans un raffinement, ne peut être utilisée dans une spécification – les substitutions VAR et les substitutions séquentielles y sont interdites.

On peut alors utiliser une substitution ANY :

```

ANY av,aw WHERE av ∈ ℕ1 ∧ aw ∈ ℕ1 − {av} THEN
  adrsint := {v ↦ av, w ↦ aw} ||
  lstsint := {av ↦ av, aw ↦ aw} ||
  memsint := {av ↦ 0, aw ↦ 1}
END

```

On peut également insister sur le caractère constant de l'adresse d'une variable, comme suit :

```

CONSTANTS adrv,adrw
PROPERTIES adrv ∈ ℕ1 ∧
           adrw ∈ ℕ1 − {adv}
INVARIANT adrsint(v) = adrv ∧ lstlink(adrv) = adrv ∧
           adrsint(w) = adrw ∧ lstlink(adrw) = adrw
INITIALISATION adrsint := {v ↦ adrv, w ↦ adrw} ||
               lstsint := {adv, adrw} ◁ id(ℤ) ||
               memsint := {adv ↦ 0, adrw ↦ 1}

```

Remarque 6.15 (Constantes) *Pour mémoire, on peut modéliser le caractère constant d'une variable en ajoutant un invariant de la forme $memsint(v) = cstv$.*

Remarque 6.16 (Modélisation alternative) *Il est également possible, pour représenter des initialisations complexes, de rejeter l'initialisation en début de la fonction `main`, lors du pre-cooking ou de la traduction.*

6.7.2 Spécification d'une opération modifiant une variable externe

On considère une structure auto-référente permettant de gérer une liste, et une fonction ajoutant un élément en tête de liste :

Exemple 6.7 (Gestion d'une liste)

```

struct link {int value;struct link *next;};
struct link *head=NULL;
void insert_first(int val)

```

La modélisation de la structure `link` est fournie dans une machine $link_s$ qui décrit :

- Un ensemble $LINK$ défini par un produit cartésien et des fonctions d'accès aux champs $Rlinkvalue$, $Rlinknext$, $Wlinkvalue$, $Wlinknext$.
- Le type étant habité, un ensemble d'identifiants $IDLINK$, des variables $memlink$, $lstlink$, $adrlink$, et des opérations $malloclink$, $freelink$, $Wlink$, $Rlink$, $WSlink$, $RSlink$, etc.

La fonction C `insert_first` est abstraite dans un autre composant incluant la spécification $link_s$.

Cette fonction a la particularité de créer une nouvelle allocation, correspondant à une nouvelle structure `link` ajoutée en tête de liste ; elle modifie donc les variables $adrlink$, $memlink$ et $lstlink$.

Ces variables étant externes, elles ne peuvent être modifiées que par l'appel à une opération dédiée. Il est donc délicat de spécifier correctement cette fonction.

Pour cette raison, on propose de définir dans *link_s* des opérations complémentaires permettant de modifier, de manière indéterministe, l'état de la machine (i.e. les variables), et utilisables pour spécifier l'opération correspondant à l'abstraction de `insert_first`.

Ces opérations doivent cependant être décrites de manière à pouvoir respecter l'invariant ; l'idée est de proposer pour chaque opération standard, une version non déterministe, pouvant modéliser un ou plusieurs appels de la version standard, avec des paramètres quelconques :

OPERATIONS

```

imalloclink ≐ ANY m, l WHERE m ∈ ℕ1 ↔ LINK ∧ memlink ⊆ m ∧
    l ∈ ℕ1 ↔ ℕ1 ∧ lstlink ⊆ l ∧
    ∀ a · (a ∈ dom(l) ⇒ a ≤ l(a))  ∧
    ∀ a · (a ∈ dom(l) ⇒ a..l(a) ∩ dom(l) = {a})  ∧
    ∀ a · (a ∈ dom(l) ⇒ a..l(a) ⊆ dom(m))  ∧
    ∀ a · (a ∈ dom(m) ⇒ ∃ b · (b ∈ dom(l) ∧ a ∈ b..l(b)))
    THEN memlink := m || lstlink := l END ;
ifreelink ≐ ANY m, l WHERE m ⊆ memlink ∧
    l ⊆ lstlink ∧
    ∀ a · (a ∈ dom(l) ⇒ a ≤ l(a))  ∧
    ∀ a · (a ∈ dom(l) ⇒ a..l(a) ∩ dom(l) = {a})  ∧
    ∀ a · (a ∈ dom(l) ⇒ a..l(a) ⊆ dom(m))  ∧
    ∀ a · (a ∈ dom(m) ⇒ ∃ b · (b ∈ dom(l) ∧ a ∈ b..l(b)))
    THEN memlink := m || lstlink := l END ;
iWlink ≐ ANY V WHERE V ∈ IDLINK ↔ ℕ THEN adrlink := adrlink <← V END ;
iWSlink ≐ ANY M WHERE M ∈ dom(memlink) ↔ LINK
    THEN memlink := memlink <← M END

```

Ces opérations ne sont pas fondamentales ; elles ont pour seul objectif de permettre une spécification automatique et ainsi de démarrer les preuves d'adéquation au plus vite, sans qu'une étape préalable de mise au point d'une spécification ne soit nécessaire, ou tout au moins en permettant de simplifier cette mise au point.

6.7.3 Non-utilisation des opérations d'accès en lecture

La définition d'opérations d'accès en lecture s'avère, en pratique, optionnelle, les variables des machines incluses étant visibles en lecture.

Plus encore, leur utilisation mène à la complexification de la traduction, puisqu'un appel d'opération nécessite l'utilisation d'une variable temporaire, comme illustré ci-dessous :

```

    head->value=current->value;
     $\xrightarrow{P_c}$  *head.value=*current.value;
     $\xrightarrow{B}$  VAR tmp1 IN tmp1 ← RSlink(current);
    VAR tmp2 IN tmp2 ← RSlink(head);
    tmp2 := Wlinkvalue(tmp2, Rlinkvalue(tmp1));
    WSlink(value, tmp2)
    END ;
END

```

Si les opérations de lecture sont remplacées par des lectures directes des variables, avec les notations de l'exemple 6.7 on peut sans variables temporaires écrire :

```

    head->value=current->value;
     $\xrightarrow{P_c, B}$  WSlink(head, Wlinkvalue(memlink(adrlink(head)),
    Rlinkvalue(memlink(adrlink(current))))

```

6.7.4 Accès direct à la mémoire

On reste dans le contexte de l'exemple 6.7, proposant une gestion de liste. On suppose que l'on trouve dans le code de l'une des fonctions une instruction de la forme suivante :

```
head->next->next=current;
```

`head->next` correspond au champ `next` de la structure pointée par `head`; sa valeur est une adresse, un pointeur sur une structure. Dans ce type d'accès, nous ne disposons pas des opérations nécessaires pour effectuer des lectures ou écritures, tout simplement parce que nous n'avons aucun identifiant associé à la structure pointée.

Il est donc bien nécessaire de prévoir des opérations de lecture et d'écriture à une adresse de la mémoire, à l'image de l'opération *directmemwrite* définie dans le paragraphe 6.5.5, mais promues et utilisables par toute machine incluante :

OPERATIONS

```
Wmemlink(a, v) = PRE a ∈ dom(memlink) ∧ v ∈ LINK THEN
                  memlink(a) := v
                  END ;
out ← Rmemlink(a) = PRE a ∈ dom(memlink) THEN
                    out := memlink(a)
                    END ;
```

Encore une fois, l'opération *Rmemlink* n'est pas indispensable, mais permet d'explicitier la précondition.

Ces opérations, dans la suite, remplacent l'ensemble des opérations d'accès indirect par identifiant, i.e. *WSlink*, *RSlink*, *WSSLink*, *RSSLink*.

6.7.5 Paramétrage des modèles par les types

Les machines abstraites peuvent être paramétrées, notamment par des ensembles. On peut donc envisager de définir des modèles de types génériques, qui seront paramétrés à l'instanciation, i.e. lors de l'inclusion.

L'intérêt n'est pas tant de simplifier le travail de traduction que de factoriser les preuves ; on notera par exemple que la preuve que l'opération d'allocation respecte l'invariant n'est pas triviale et peut être fournie, une fois pour toute, dans un type générique.

Remarque 6.17 (Ensembles finis) *Les ensembles passés en paramètres d'une machine abstraite sont toujours supposés finis et non vides ; cela signifie ici que nous ne pouvons donc pas utiliser le modèle générique des pointeurs pour l'instancier en un modèle de pointeurs sur pointeurs, qui sont des pointeurs sur \mathbb{N} , ensemble infini.*

6.7.6 Conclusion sur les modèles

Les modèles proposés pour les différents types de données sont fournis en annexe A.

Chapitre 7

Pre-cooking, traduction et analyse

La démarche proposée dans les chapitres précédents pour mener le processus d'abstraction consiste tout d'abord à faire un pre-cooking sur le code C , puis à traduire le code C réécrit en un ensemble de modules B , et enfin d'analyser les modules B pour les abstraire et extraire une spécification dans un cadre formel.

Il reste à aborder la question de la frontière entre ces différentes étapes.

Comme l'illustrent les trois exemples suivants, cette frontière est arbitraire ; selon la philosophie générale retenue, une importance plus ou moins grande peut être donnée à chacune des étapes :

Exemple 7.1 (Pre-cooking maximaliste)

$$\begin{array}{l} \text{if } (expr) \{ bloc^1 \text{ break; } bloc^2 \} \\ \text{else } \{ bloc^3 \text{ break; } bloc^4 \} \\ \xrightarrow{P_c} \text{if } (expr) \{ bloc^1 \} \\ \text{else } \{ bloc^3 \} \\ \text{break;} \\ \xrightarrow{B} \text{IF } expr \text{ THEN } bloc^1 \text{ ELSE } bloc^2 \text{ END;} \\ \text{flag}_{break} := TRUE \end{array}$$

Exemple 7.2 (Pre-cooking minimaliste, traduction maximaliste)

$$\begin{array}{l} \text{if } (expr) \{ bloc^1 \text{ break; } bloc^2 \} \\ \text{else } \{ bloc^3 \text{ break; } bloc^4 \} \\ \xrightarrow{P_c} \text{if } (expr) \{ bloc^1 \text{ break; } bloc^2 \} \\ \text{else } \{ bloc^3 \text{ break; } bloc^4 \} \\ \xrightarrow{B} \text{IF } expr \text{ THEN } bloc^1 \text{ ELSE } bloc^2 \text{ END;} \\ \text{flag}_{break} := TRUE \end{array}$$

Exemple 7.3 (Pre-cooking et traduction minimalistes)

$$\begin{array}{l} \text{if } (expr) \{ bloc^1 \text{ break; } bloc^2 \} \\ \text{else } \{ bloc^3 \text{ break; } bloc^4 \} \\ \xrightarrow{P_c} \text{if } (expr) \{ bloc^1 \text{ break; } bloc^2 \} \\ \text{else } \{ bloc^3 \text{ break; } bloc^4 \} \\ \xrightarrow{B} \text{IF } expr \text{ THEN } bloc^1 ; \text{flag}_{break} := TRUE \text{ ELSE } bloc^2 ; \text{flag}_{break} := TRUE \text{ END} \end{array}$$

L'objectif de cette étude est de permettre l'analyse d'un code C dans l'environnement formel du B ; comme nous l'avons indiqué précédemment, toutes les étapes préalables à l'analyse interne au B sont des étapes de confiance – de fait, le pre-cooking et la traduction échappent à la preuve¹.

Idéalement, donc, ces deux étapes préalables à l'analyse doivent être aussi simples que possibles, afin de minimiser le risque d'erreur. De manière générale, la démarche retenue correspond donc à celle présentée par l'exemple 7.3.

Cette simplification du pre-cooking et de la traduction à un coût, puisqu'une partie de la charge de travail est reportée à l'analyse, qui doit être menée par l'humain; en repartant de l'exemple 7.3, par exemple, c'est à l'utilisateur de proposer une abstraction simplificatrice, telle que la suivante :

Exemple 7.4

$$\begin{array}{l} \text{IF } expr \text{ THEN } bloc^1 ; flag_{break} := TRUE \text{ ELSE } bloc^2 ; flag_{break} := TRUE \text{ END} \\ \xrightarrow{Abs} \text{IF } expr \text{ THEN } bloc^1 \text{ ELSE } bloc^2 \text{ END ; } flag_{break} := TRUE \end{array}$$

Par contre, une fois cette abstraction proposée, il est possible de prouver formellement qu'elle est équivalente à la version obtenue par traduction.

Remarque 7.1 (Post-cooking) *Comme l'expose cet exemple, on peut envisager, après la traduction, la génération d'une abstraction de référence ($a0$), mais aussi d'abstractions de niveau supérieur, obtenues par un processus de réécriture – désigné sous le terme de post-cooking par analogie avec le pre-cooking. L'objectif du post-cooking est de réécrire un module B en un autre module B , supposé équivalent; cette équivalence doit être formellement par une preuve d'adéquation.*

Le post-cooking n'est pas abordé dans le cadre de cette étude.

¹Tout au plus, on peut prévoir le cas échéant une procédure de test des binaires, qui à l'issue du pre-cooking vérifie autant que faire se peut que le code obtenu par réécriture est équivalent au code original.

Chapitre 8

Formalisation du pre-cooking

Le pre-cooking, introduit dans les chapitres 5 et 6, est défini comme une étape préalable à la traduction vers le B . Il consiste en une réécriture du code C analysé, visant à obtenir un nouveau code C dont le fonctionnement est observationnellement équivalent, mais dont la forme est normalisée ; cette normalisation permet d'éliminer les constructions du C qui ne sont pas traduisibles en B , ou qui ne sont pas directement représentables dans notre modélisation.

Il s'agit donc notamment :

- D'éliminer les effets de bords.
- De remplacer les variables et les tableaux par des pointeurs.
- De s'assurer que les structures `switch` analysées sont dans une forme normale, permettant une traduction directe vers le B en utilisant une substitution `CASE`.

En complément, on s'autorise également dans le pre-cooking à remplacer certains opérateurs, tels que les opérateurs d'incrémentement préfixés ou postfixés, pour réduire le nombre de règles de la traduction.

Enfin, le pre-cooking peut permettre de mettre en évidence certaines constructions, par exemple en identifiant et en nommant les types anonymes¹.

8.1 Renommage

Comme cela a été illustré dans le paragraphe 3.5.4, des problèmes de nommage peuvent se poser lors de la traduction d'un code C en B . Les mots-clés du B ne peuvent en effet pas être utilisés pour nommer une opération, une variable ou une constante.

Il est donc nécessaire de prévoir un renommage de certains objets du C modélisés en B . Pour des raisons pratiques, ce renommage est réalisé de manière systématique sur tous les objets, en première étape du pre-cooking ; de la sorte, il est notamment possible de distinguer les objets du code C original des objets supplémentaires générés par le pre-cooking, telles que les variables temporaires utilisées pour la décomposition des expressions complexes.

Le principe retenu pour le renommage est très simple, puisqu'on décide que tout identifiant de variable, de fonction ou de type du code C original doit être postfixé par `_G` (marquant les identifiants d'objets globaux), `_L` (marquant les identifiants d'objets locaux) ou `_F` (marquant les identifiants formels) ; aucun mot clé du B ne se termine par ces symboles, ce qui garantit l'absence de conflits.

¹Les règles associées à l'extraction et au nommage des types ne sont pas détaillées dans le cadre de cette étude.

Le schéma de réécriture, appliqué en une unique passe, est le suivant :

$$\boxed{id \xrightarrow[\text{global}]{P_c} id_G}$$

$$\boxed{id \xrightarrow[\text{local}]{P_c} id_L}$$

$$\boxed{id \xrightarrow[\text{formel}]{P_c} id_F}$$

Dans la pratique, il faut pour appliquer ces schémas pouvoir caractériser les identifiants selon le principe suivant :

- Le nom d'une fonction, d'une variable globale, d'un pointeur global ou d'un tableau global est un identifiant global.
- Le nom d'une d'une variable locale, d'un pointeur local ou d'un tableau local est un identifiant local.
- Le nom d'un type structure, d'un type union, d'un type énuméré, d'un type redéfini (par `typedef`), d'un champ dans un type structure ou union, ou encore d'un paramètre de fonction, est un identifiant formel.

Signalons qu'un label n'est pas considéré ici comme un identificateur, et que l'on ne renomme pas les mots-clés du C , dont par exemple `return`².

Exemple 8.1

<pre>int x=0; struct point {int x,int y}; struct point fn(char a) { struct point tmp; tmp.x=x; tmp.y=2*a; return tmp; }</pre>	$\xrightarrow{P_c}$	<pre>int x_G=0; struct point_F {int x_F,int y_F}; struct point_F fn_G(char a_F) { struct point_F tmp_L; tmp_L.x_F=x_G; tmp_L.y_F=2*a_F; return tmp_L; }</pre>
---	---------------------	---

8.2 Traitement des expressions

On peut considérer, a priori, que le pre-cooking relatif au traitement des expressions doit se limiter à la suppression des effets de bord, qui ne sont pas directement représentables en B ; dans la pratique, cependant, la nécessité d'étendre le pre-cooking à la simplification de toutes les expressions non élémentaires est indispensable.

En effet, de par notre modélisation, les appels d'opérations arithmétiques (ou logiques) ou les appels de fonctions du code C sont modélisés par des appels d'opérations en B . Une expression, même simple et sans effet de bord, est donc souvent traduite par une séquence d'appels imposant la déclaration de variables locales, comme l'illustre l'exemple suivant :

Exemple 8.2 (Traduction d'une expression)

<pre>x=sqrt(3*x);</pre>	$\xrightarrow{P_c, B}$	<pre>VAR tmp IN tmp ← mul(3, x_L); x_L ← sqrt_G(tmp) END</pre>
-------------------------	------------------------	--

²Le problème du renommage des fonctions de bibliothèque doit également être abordé; il est nécessaire pour éviter les conflits, mais un tel renommage n'est pas toujours répercutable dans des bibliothèques dont la source n'est pas fourni. On peut alors, par exemple, utiliser des macros de substitution `#define`.

Il apparaît donc pertinent de proposer la décomposition systématique des expressions au niveau du pre-cooking, en définissant un ensemble unique de règles qui permettent également d'éliminer les effets de bord et certains opérateurs indésirables.

Dans la suite, *val* désigne une variable ou une constante, et les expressions (non réduites à une variable ou une constante) sont désignées par *expr* ; l'indice *s* signifie que l'expression comporte un effet de bord explicite, i.e. un opérateur d'affectation (=, +=, -=, *=, /=, %=, <<=, >>=, &=, |=), un opérateur préfixe (++ , --), *b* permettant plus spécifiquement de marquer la présence d'un opérateur préfixe, *a* d'un opérateur postfixe et \perp l'absence de ces opérateurs³.

Une expression comporte obligatoirement un opérateur ou un appel de fonction, sans quoi elle se réduit à *val*.

La première règle permet de réécrire les appels de fonction purs :

$$\frac{P_c}{\rightarrow} \boxed{\begin{array}{l} fn_G(expr_{\sigma_1}^1, \dots, expr_{\sigma_{i-1}}^{i-1}, expr_{\sigma}^i, val^{i+1}, \dots, val^n); \\ \{ type \mathbf{tmp}_{cpt} = expr_{\sigma}^i; \\ fn_G(expr_{\sigma_1}^1, \dots, expr_{\sigma_{i-1}}^{i-1}, \dots, \mathbf{tmp}_{cpt}, val^{i+1}, \dots, val^n); \} \end{array}}$$

Dans cette règle comme dans les suivantes, \mathbf{tmp}_{cpt} désigne une variable fraîche dont l'identifiant n'est pas un mot-clé du *B* ; dans la pratique, le nom de variable frais est obtenu en concaténant à *tmp* la valeur du compteur *cpt*, qui est incrémenté à chaque génération de variable locale⁴.

Remarque 8.1 (Paramètres de retour ignorés) *Il est valide, en C, de faire un appel de pur (sans affectation) à une fonction retournant une valeur ; dans ce cas de figure, le pre-cooking doit également déclarer une variable temporaire pour récupérer le résultat, afin que l'appel puisse être modélisé correctement en B.*

Remarque 8.2 (Effets de bords multiples) *La règle proposée ci-dessus permet de réécrire l'appel d'une fonction dans lequel plusieurs effets de bords interfèrent.*

Le C ne spécifie pas l'ordre d'évaluation, et la multiplication des effets de bord rend alors impossible la spécification du résultat obtenu, comme dans l'exemple suivant dont le résultat dépend de l'architecture considérée⁵ :

```
int dif(int a,int b) {return a-b;}
int main()
{ int x=1;
  return dif(++x,++x); }
```

Notons qu'il n'y a réellement indétermination sur le résultat que si différentes expressions ont un effet de bord sur une même variable ; il faut cependant noter que cette condition n'est pas toujours facile à évaluer, puisque les fonctions peuvent implicitement modifier des variables globales.

Une modélisation formelle de cet indéterminisme de spécification est possible en B, sous la forme d'une substitution CHOICE gardée (et non une substitution ||), mais n'a pas été retenue.

La règle proposée ci-dessus simule une évaluation à rebours des paramètres (ce qui semble être le cas sur l'architecture de référence) ; il faut toutefois noter que ce type de règle peut constituer une erreur de modélisation si différents effets de bord ont une influence sur la même variable.

³Rappelons que cela ne signifie pas qu'il n'y a pas d'effet de bord lié à l'évaluation de l'expression ; un appel de fonction peut masquer la modification d'une variable globale.

⁴Notons que le nom des variables générées par pre-cooking n'est pas post-fixé par *L*.

⁵La norme n'impose pas au compilateur de spécifier son implémentation ; une version originale faisant dépendre l'ordre d'évaluation de l'heure système est donc valide.

Le schéma général des autres règles de pre-cooking éliminant les effets de bord et réécrivant les expressions complexes dans les structures de contrôle sont les suivants :

$\text{if } (expr_\sigma) \text{ bloc}^1 \quad \xrightarrow{P_c} \quad \left\{ \begin{array}{l} \text{int tmp}_{cpt}=(\text{int})expr_\sigma; \\ \text{if } (\text{tmp}_{cpt}) \text{ bloc}^1 \\ \text{else } \text{bloc}^2 \end{array} \right\}$
$\text{switch } (expr_\sigma) \text{ caselst} \quad \xrightarrow{P_c} \quad \left\{ \begin{array}{l} \text{int tmp}_{cpt}=(\text{int})expr_\sigma; \\ \text{switch } (\text{tmp}_{cpt}) \text{ caselst} \end{array} \right\}$
$\text{while } (expr_\sigma) \text{ bloc} \quad \xrightarrow{P_c} \quad \left\{ \begin{array}{l} \text{int tmp}_{cpt}=(\text{int})expr_\sigma; \\ \text{while } (\text{tmp}_{cpt}) \\ \left\{ \begin{array}{l} \text{bloc} \\ \text{tmp}_{cpt}=(\text{int})expr_\sigma; \end{array} \right\} \end{array} \right\}$
$\text{do } \text{bloc} \text{ while } (expr_\sigma) \quad \xrightarrow{P_c} \quad \left\{ \begin{array}{l} \text{int tmp}_{cpt}; \\ \text{do } \left\{ \begin{array}{l} \text{bloc} \\ \text{tmp}_{cpt}=(\text{int})expr_\sigma; \end{array} \right\} \\ \text{while } \text{tmp}_{cpt} \end{array} \right\}$
$\begin{array}{l} \text{for } (\text{bloc}^1; expr_\sigma; \text{bloc}^2) \text{ bloc}^3 \\ \\ \xrightarrow{P_c} \quad \left\{ \begin{array}{l} \text{int tmp}_{cpt}; \\ \text{for } (\text{bloc}^1, \text{tmp}_{cpt}=(\text{int})expr_\sigma; \\ \quad \text{tmp}; \\ \quad \text{bloc}^2, \text{tmp}_{cpt}=(\text{int})expr_\sigma) \text{ bloc}^3 \end{array} \right\} \end{array}$

Remarque 8.3 (Séquence et boucle for) Une structure `for` accepte en paramètre une séquence, i.e. des instructions séparées par des virgules; cette propriété est utilisée dans la règle ci-dessus.

Une séquence est évaluée de gauche à droite, et peut aussi être utilisée dans les appels de fonctions, ou de manière plus générale dans les expressions; à une telle séquence est associée un résultat qui a le type et la valeur de l'opérande de droite.

Exemple 8.3 Les deux code suivants sont observationnellement identiques :

$$\text{for } (i=0; i++<5; i++) \text{ bloc} \quad \xrightarrow{P_c} \quad \left\{ \begin{array}{l} \text{int tmp}; \\ \text{for } (i=0, \text{tmp}=i++; \text{tmp}<5; i++, \text{tmp}=i++) \text{ bloc} \end{array} \right\}$$

Les schémas de suppression des opérateurs préfixés et postfixés sont les suivants :

$[type] \text{ var}^1 = expr_b(++\text{var}^2); \quad \xrightarrow{P_c} \quad \begin{array}{l} \text{var}^2 = \text{var}^2 + 1; \\ [type] \text{ var}^1 = [++\text{var}^2 := \text{var}^2] expr_b; \end{array}$
$[type] \text{ var}^1 = expr_b(--\text{var}^2); \quad \xrightarrow{P_c} \quad \begin{array}{l} \text{var}^2 = \text{var}^2 - 1; \\ [type] \text{ var}^1 = [--\text{var}^2 := \text{var}^2] expr_b; \end{array}$
$[type] \text{ var}^1 = expr_a(\text{var}^2++); \quad \xrightarrow{P_c} \quad \begin{array}{l} [type] \text{ var}^1 = [\text{var}^2++ := \text{var}^2] expr_a; \\ \text{var}^2 = \text{var}^2 + 1; \end{array}$
$[type] \text{ var}^1 = expr_a(\text{var}^2--); \quad \xrightarrow{P_c} \quad \begin{array}{l} [type] \text{ var}^1 = [\text{var}^2-- := \text{var}^2] expr_a; \\ \text{var}^2 = \text{var}^2 - 1; \end{array}$
$\left\{ \begin{array}{l} \text{var}++; \\ ++\text{var}; \end{array} \right. \quad \xrightarrow{P_c} \quad \text{var} = \text{var} + 1;$
$\left\{ \begin{array}{l} \text{var}--; \\ --\text{var}; \end{array} \right. \quad \xrightarrow{P_c} \quad \text{var} = \text{var} + 1;$

Remarque 8.4 (Stratégie) Notons que l'on suppose ici que les opérateurs préfixés et postfixés sont toujours utilisés dans le cadre d'une affectation; ceci résulte de la stratégie implicite qui consiste à appliquer au préalable les règles de simplification des expressions – et permet de grandement simplifier la modélisation de ces opérateurs.

Remarque 8.5 (Sémantique de ++ et --) La sémantique des opérateurs préfixés et postfixés est peu claire; à titre d'exemple, on peut considérer l'opération $x = ((y++) - (y++))$; et se demander si après exécution x vaut 0, 1 ou -1 .

En fait, dans ce type de cas le résultat n'est pas spécifié par le langage, et pire encore pour une architecture donnée le résultat peut varier selon le programme, en fonction des optimisations réalisées par le compilateur.

De manière générale, ce type de code doit être considéré comme proscrit. On admet donc que dans les programmes traités par pre-cooking, toute variable avec une opérateur préfixe ou postfixe n'apparaît qu'une fois, et uniquement à droite – ce qui interdit également les affectations de la forme $x = x++$;⁶.

Remarque 8.6 (Séquence et boucle for) Les réécritures ci-dessus sont un peu différentes si l'opérateur préfixé ou postfixé apparaît dans une suite d'instructions séparées par des virgules, comme c'est le cas pour les paramètres des boucles `for`.

On considère également comme indésirables les opérateurs d'affectation, avec le schéma suivant où le @ désigne +, -, *, /, %, <<, >>, & ou | :

$$\boxed{var@=expr; \quad \xrightarrow{P_c} \quad var=var@expr;}$$

Remarque 8.7 (Opérateurs binaires) Des règles relatives aux opérateurs binaires sont ici explicitées, mais ces opérateurs ne sont pas modélisés dans le cadre de l'étude.

Il nous reste enfin à proposer des schémas de réécriture des expressions complexes dans les affectations. On suppose qu'à ce stade, les opérateurs d'incrémement et de décrémement préfixés et postfixés ont été éliminés; les expressions considérées ne comportent donc que des appels de fonction et des opérations.

Afin de proposer un schéma de réécriture unique, on confond dans ce qui suit les fonctions et les opérations, désignés par fn utilisé en notation préfixe :

$$\boxed{[type] var=fn(expr_{\perp}); \quad \xrightarrow{P_c} \quad \begin{array}{l} \{ type \text{ tmp}_{cpt}; \\ \text{tmp}_{cpt}=expr_{\perp}; \\ [type] var=fn(\text{tmp}_{cpt}); \end{array}}$$

Le type de la variable locale déclarée est le type de retour de la fonction ou, pour les opérations arithmétiques ou logiques, le type est `regtype`, correspondant au type des registres, défini en C par une déclaration ajoutée en début de code :

```
typedef int regtype;
```

⁶On pourra se référer à la définition des *sequencepoints* en C pour approfondir la question.

8.3 Réorganisation des structures *switch*

Comme indiqué dans la section 5.2, on peut modéliser par une substitution CASE du *B* une structure **switch** du *C* si cette dernière est de la forme suivante :

```
switch (expr)
{ case c1:bloc1 break;
  case c2:bloc2 break;
  ...
  case cn:blocn break;
  default:blocd break; }
```

Le *C* n'impose pas qu'un **break** apparaisse en fin de chaque alternative d'une structure **switch**; le principe de fonctionnement est qu'un fois une alternative sélectionnée, le bloc d'instructions associé et les suivants sont exécutés séquentiellement, jusqu'à rencontre d'une instruction de rupture.

Les schémas de règles suivants sont applicables strictement dans une structure **switch**, *label*, *label*¹ et *label*² y désignent indifféremment un **case** ou un **default** :

$label:instr^1 \text{ break; } instr^2 \xrightarrow{P_c} label:instr^1 \text{ break;}$
--

$\begin{array}{l} \text{switch (expr)} \\ \{ \dots \\ \quad label:bloc \} \end{array} \xrightarrow{P_c} \begin{array}{l} \text{switch (expr)} \\ \{ \dots \\ \quad label:bloc \text{ break; } \} \end{array}$

$\begin{array}{l} label^1:bloc^1 \\ label^2:bloc^2 \text{ break;} \end{array} \xrightarrow{P_c} \begin{array}{l} label^1:bloc^1 \text{ bloc}^2 \text{ break;} \\ label^2:bloc^2 \text{ break;} \end{array}$

$\begin{array}{l} \text{switch(expr)} \\ \{ \text{default}:instr^d \text{ break;} \\ \quad \text{case } c^i:instr^i \text{ break;} \\ \quad \dots \} \end{array} \xrightarrow{P_c} \begin{array}{l} \text{switch(expr)} \\ \{ \text{case } c^i:instr^i \text{ break;} \\ \quad \text{default}:instr^d \text{ break;} \\ \quad \dots \} \end{array}$

$\begin{array}{l} \text{case } c^i:instr_i \text{ break;} \\ \text{default}:instr^d \text{ break;} \\ \text{case } c^j:instr_j \text{ break;} \end{array} \xrightarrow{P_c} \begin{array}{l} \text{case } c^i:instr_i \text{ break;} \\ \text{case } c^j:instr_j \text{ break;} \\ \text{default}:instr^d \text{ break;} \end{array}$

On peut enfin prévoir, en l'absence de clause **default**, l'ajout de cette clause, se réduisant à une instruction **break**; il est cependant plus simple de prévoir un schéma de traduction supplémentaire pour les structures **switch** sans clause **default**⁷.

8.4 Modélisation des variables et des tableaux

La prise en compte de l'opérateur de déréférencement, dans le paragraphe 6.5.3, nous a conduit à une modélisation dans laquelle seuls les pointeurs sont représentés. Le pre-cooking doit donc nous permettre de remplacer les instances de variables, globales ou locales, par des pointeurs.

Les règles de pre-cooking permettent de réécrire les instructions faisant intervenir un nom de variable globale selon les schémas suivants :

$type \text{ var_G}; \xrightarrow{P_c} type \text{ *var_G=malloc(1*sizeof(type));}$

$type \text{ var_G=expr}; \xrightarrow{P_c} \begin{array}{l} type \text{ *var_G=malloc(1*sizeof(type));} \\ \text{*var_G=expr;} \end{array}$

⁷Tout comme on prévoit des schémas de traduction pour les structures *if* sans *else*, etc.

$\&var_G \xrightarrow{P_c} var_G$

$var_G \xrightarrow{P_c} *var_G$

Remarque 8.8 (Initialisation des variables globales) *Les schémas ci-dessus transforment l'initialisation de la variable en écriture dans une zone allouée; une telle instruction ne peut cependant pas apparaître dans la zone globale d'un code.*

En pratique, on propose de déplacer ces initialisations (ainsi, éventuellement, que les allocations permettant l'initialisations des pointeurs) en début de la fonction `main_G`.

Les schémas applicables pour le remplacement des variables locale sont proches de ceux proposés pour les variables globales; une petite adaptation est cependant nécessaire, car une variable locale est associée à une position mémoire de la pile, et disparaît à la fin du bloc dans lequel elle a été déclarée. Le schéma de réécriture complet est donc le suivant :

$\left\{ \begin{array}{l} bloc^1 \\ type\ var_; \\ bloc^2 \end{array} \right\} \xrightarrow{P_c} \left\{ \begin{array}{l} bloc^1 \\ type\ *var_ = malloc(1 * sizeof(type)); \\ bloc^2 \\ free(var_); \end{array} \right\}$

$\left\{ \begin{array}{l} bloc^1 \\ type\ var_ = expr; \\ bloc^2 \end{array} \right\} \xrightarrow{P_c} \left\{ \begin{array}{l} bloc^1 \\ type\ *var_ = malloc(1 * sizeof(type)); \\ *var_ = expr; \\ bloc^2 \\ free(var_); \end{array} \right\}$
--

$\&var_L \xrightarrow{P_c} var_L$

$var_L \xrightarrow{P_c} *var_L$

Notons que ces schémas ne s'appliquent pas aux variables temporaires créées par le pre-cooking ou aux paramètres formels d'une fonction.

Remarque 8.9 (Variables locales et pointeurs globaux) *Le code C suivant est compilé sans erreur :*

```
int *ptr;
void strange()
{ int i=0;
  ptr=&i;
  return; }
```

Une fois appelée la fonction `strange`, le pointeur `ptr` est chargé avec une adresse qui se trouve, en pratique, dans la pile; la lecture de ce pointeur après appel de la fonction `strange` est valide du point de vue du système, mais la valeur de retour est indéterminée, la pile pouvant être modifiée.

La modélisation proposée ne reflète pas exactement ce fonctionnement, puisque la variable locale est considérée comme désallouée en fin de bloc; il sera donc impossible de déterminer la valeur de retour d'une lecture de la valeur pointée, ni même de prouver que l'adresse du pointeur correspond bien à une zone allouée.

De la même façon, la modélisation retenue pour les tableaux est également basée sur les pointeurs; ici encore le pre-cooking doit remplacer les expressions tabulaires par des expressions utilisant l'arithmétique des pointeurs. Les schémas de règles sont les suivants :

$type\ tab1D_G[expr]; \xrightarrow{P_c} type\ *tab1D_G = malloc(expr * sizeof(type));$
--

$\frac{\{ \text{bloc}^1 \quad \text{type tab1D_L}[expr]; \quad \text{bloc}^2 \}}{\frac{P_c}{\rightarrow}} \quad \{ \text{bloc}^1 \quad \text{type *tab1D_L}=\text{malloc}(expr*\text{sizeof}(type)); \quad \text{bloc}^2 \quad \text{free}(tab1D_L); \}$
$\frac{P_c}{\rightarrow} \quad \frac{\text{type tab2D_G}[expr^1][expr^2];}{\text{type *tab2D_G}=\text{malloc}(expr^1*expr^2*\text{sizeof}(type));}$
$\frac{P_c}{\rightarrow} \quad \frac{\{ \text{bloc}^1 \quad \text{type tab2D_L}[expr^1][expr^2]; \quad \text{bloc}^2 \}}{\{ \text{bloc}^1 \quad \text{type *tab2D_L}=\text{malloc}(expr^1*expr^2*\text{sizeof}(type)); \quad \text{bloc}^2 \quad \text{free}(tab2D_L); \}}$
$\frac{P_c}{\rightarrow} \quad \frac{\&tab1D_G[expr]}{tab1D_G+expr}$
$\frac{P_c}{\rightarrow} \quad \frac{\&tab1D_L[expr]}{tab1D_L+expr}$
$\frac{P_c}{\rightarrow} \quad \frac{ptr_G[expr]}{ptr_G+expr}$
$\frac{P_c}{\rightarrow} \quad \frac{ptr_L[expr]}{ptr_L+expr}$
$\frac{P_c}{\rightarrow} \quad \frac{ptr_F[expr]}{ptr_F+expr}$
$\frac{P_c}{\rightarrow} \quad \frac{\&tab2D_G[expr^1][expr^2]}{tab2D_G+expr^1+\text{width}(tab2D_G)*expr^2}$
$\frac{P_c}{\rightarrow} \quad \frac{\&tab2D_L[expr^1][expr^2]}{tab2D_L+expr^1+\text{width}(tab2D_L)*expr^2}$
$\frac{P_c}{\rightarrow} \quad \frac{tab2D_G[expr^1][expr^2]}{*(tab2D_G+expr^1+\text{width}(tab2D_G)*expr^2)}$
$\frac{P_c}{\rightarrow} \quad \frac{tab2D_L[expr^1][expr^2]}{*(tab2D_L+expr^1+\text{width}(tab2D_L)*expr^2)}$
$\frac{P_c}{\rightarrow} \quad \frac{*(tab2D_G+expr^1)+expr^2}{tab2D_G+expr^1+\text{width}(tab2D_G)*expr^2}$
$\frac{P_c}{\rightarrow} \quad \frac{*(tab2D_L+expr^1)+expr^2}{tab2D_L+expr^1+\text{width}(tab2D_L)*expr^2}$

Notre réécriture est telle qu'elle connaît les identifiants des tableaux à 2 dimensions, ainsi que leur largeur (*width*)⁸.

En termes de stratégie, notons que la réécriture permettant la modélisation de tableau doit être préalable à la réécriture des expressions complexes – ou tout au moins que les règles correspondantes doivent être appliquées aux expressions dimensionnant des tableaux.

Enfin, pour minimiser le nombre de formes à traiter dans la traduction et expliciter au mieux la nature des opérations d'accès, on ajoute la règle suivante :

$$\frac{P_c}{\rightarrow} \quad \frac{expr \rightarrow field}{*(expr).field}$$

⁸Seuls les tableaux à 1 et 2 dimensions sont considérés dans le cadre de cette étude.

Chapitre 9

Formalisation de la traduction

Ce chapitre traite de la traduction d'un code C , préalablement traité par pre-cooking, en un ensemble de modules B le représentant.

Pour mémoire, rappelons qu'à un code C , la traduction associe une abstraction de référence (indiquée par $a0$) sous la forme d'un raffinement B , et une spécification (indiquée par s) sous la forme d'une machine abstraite B .

Le chapitre 5 propose des modélisation des structures de contrôle du C sous la forme de substitutions du B ; ces modélisations sont ici traduites par des règles, qui prennent également en compte les instructions de rupture **continue**, **break** et **return**. Toutefois, pour rester conforme à la philosophie générale proposée dans le chapitre 7, la traduction se veut aussi simple que possible, ce qui permet de limiter le risque d'erreurs.

La traduction permet également de modéliser les structures de données manipulées par le code C , conformément aux propositions du chapitre 6. Une partie du traitement permettant la modélisation est réalisé par le pre-cooking, par exemple le remplacement des variables par des pointeurs; il reste cependant à dériver pour chaque type manipulé un ou plusieurs modules B , et à les architecturer de manière cohérente, avant de traduire les expressions sur les objets en conformité avec les choix de modélisation.

9.1 Problématique générale de la traduction

9.1.1 Génération de l'abstraction de référence

Le chapitre 5 présente les substitutions B permettant de représenter les structures de contrôle du C ¹. Cependant, alors qu'une substitution B , dans sa forme générale, est une structure inductive dont l'interprétation suit également un schéma inductif, l'exécution d'une structure de contrôle en C peut être interrompue par une instruction **break**, **continue** ou **return**, constituant des instructions de rupture.

En substance, le problème des instructions de rupture est qu'elles sont parfois applicables à une sur-structure de la structure où elles apparaissent :

- **break** traverse les séquences et les **if** pour s'appliquer à la première structure **switch**, **while**, **do** ou **for** supérieure.
- **continue** traverse les séquences, les **if** et les **switch** pour s'appliquer à la première structure **while**, **do** ou **for** supérieure.

¹Notons en passant que l'équivalence n'est pas évidente; par exemple, le **if** du C est décrit de manière opérative, dans un contexte d'exécution séquentielle, alors que le **IF** du B est défini comme un transformateur de prédicats.

- `return` traverse les toutes les structures pour s'appliquer à la fonction.

On peut cependant simuler ce type de mécanisme par des structures conditionnelles. Ainsi, les deux codes suivants sont observationnellement équivalents :

Exemple 9.1 (Modélisation du `break` dans un `while`)

```

while (expr_1)
{ instr_1;
  if (expr_2) { instr_2
               break; }
  instr_4 }
≡
int flag_break=0;
while (expr_1 && !flag_break)
{ instr_1;
  if (expr_2) { instr_2
               flag_break=1; }
  if (!flag_break) instr_4 }

```

Comme indiqué, le `break` dans cet exemple ne s'applique pas à la structure `if` mais bien à la sur-structure `while`; une telle instruction de rupture ne peut pas être modélisée directement en B , puisqu'elle contredit l'approche inductive, et une approche similaire à celle proposée par le code de droite doit être considérée.

Il est possible d'utiliser le pre-cooking pour transformer un code C quelconque en code dans lequel les instructions de rupture ont disparues (à l'exception du `return` qui doit être factorisé), mais cette solution n'est pas conforme avec la philosophie générale retenue.

C'est donc pendant la traduction que les transformations strictement nécessaires du code C sont effectuées; à cette fin, le processus de traduction d'un code C est décomposé comme suit :

- Production de l'AST (*Abstract Syntactical Tree*), non détaillée ici.
- Etiquetage de l'AST, permettant d'identifier pour chaque structure les instructions de rupture qu'elle contient et qui sont susceptibles de modifier l'exécution des structures supérieures ou voisines.
- Production des modules B en exploitant les informations de l'AST étiqueté.

9.1.2 Génération d'une spécification initiale

Le problème de la spécification initiale est différent; il s'agit ici de proposer une machine abstraite qui soit en adéquation avec le raffinement que constitue l'abstraction de référence.

La production d'une telle spécification, de manière automatique, soulève de nombreux problèmes, dont le passage d'un paradigme séquentiel et déterministe (le code C) à un paradigme parallèle et non déterministe (la machine abstraite du B).

Essentiellement, on s'appliquera donc à modéliser les prototypes, en laissant à l'utilisateur le soin de construire une spécification plus détaillée conforme à ses attentes.

9.1.3 Décomposition de la traduction

On cherche à traduire un code C , le résultat de la traduction étant un ensemble d'entités B . On note C le code analysé et $code_B$ la fonction de traduction; $code_B(C)$ est donc le résultat de la traduction.

Afin de simplifier l'expression de la fonction $code_B$, il apparaît pertinent de proposer des sous-fonctions spécialisées :

- fun_s retourne la spécification d'une fonction du code C .
- fun_{a0} retourne l'abstraction de référence d'une fonction du code C .
- $bloc_{a0}$ retourne l'abstraction de référence d'un bloc d'instructions.

- val_{a0} retourne la traduction d'une valeur (variable ou constante) apparaissant à droite d'une affectation ou utilisée comme argument.
- aff_{a0} retourne la traduction d'une affectation.

La description de ces différentes fonctions est inductive, comme l'illustre l'exemple suivant :

Exemple 9.2 (Traduction d'une structure if)

$$bloc_{a0} \left(\begin{array}{l} \text{if } (a) \text{ bloc} \\ \text{else } x=3*x+x; \end{array} \right) \xrightarrow{B} \begin{array}{l} \text{IF } val_{a0}(a) \neq 0 \text{ THEN } bloc_{a0}(bloc) \\ \text{ELSE } bloc_{a0}(x:=3*x;) \end{array}$$

9.2 Etiquetage de l'AST

L'étiquetage de l'AST permet d'identifier les catégories syntaxiques pour les différentes structures de contrôle. Comme précédemment, les catégories sont précisées par des indices ; un c indique la présence d'au moins un **continue**, un b d'au moins un **break**, et un r d'au moins un **return**. Le symbole \perp est utilisé pour l'absence, les autres symboles (γ pour c , β pour b , ρ pour r et δ pour un indice quelconque) sont utilisés comme des méta-variables.

L'étiquetage de l'AST se fait des feuilles au noeud ; la catégorie de départ (les feuilles de l'AST) est **instr**, qui désigne une instruction éventuellement vide (i.e. ε) qui n'est ni une structure de contrôle, ni un **break**, ni un **continue**, ni un **return**. A partir des feuilles, il est possible de décrire les séquences (*seq*), les structures *if*, *switch*, *while*, *do*, *for* et les blocs (*bloc*).

On peut d'ores et déjà noter les particularité suivantes :

- *seq* ne nécessite qu'une seule position d'indice (valant c , b , r ou \perp), en raison d'un phénomène d'absorption².
- *switch* ne nécessite que deux positions d'indice (pour c et r) ; pour mémoire, comme indiqué dans le paragraphe 5.2, seules les structures *switch* dont tous les cas se terminent par **break** sont traduisibles et sont considérées pour la traduction³.
- *while*, *do*, *for* nécessitent trois positions d'indices ; cependant, les indices c et b marquent la présence d'instructions **continue** et **break** applicables à la structure elle-même, alors que l'indice r marque la présence d'au moins une instruction **return** applicable à l'ensemble des sur-structures jusqu'à la fonction.

Les règles d'étiquetage pour les séquences sont les suivantes :

$$\begin{array}{l} \boxed{\{instr; \xrightarrow{AST} [seq_{\perp}]}} \quad \boxed{instr; \} \xrightarrow{AST} seq_{\perp}} \\ \boxed{[seq_{\delta} instr; \xrightarrow{AST} [seq_{\delta}]}]} \quad \boxed{instr; seq_{\delta}] \xrightarrow{AST} seq_{\delta}} \\ \boxed{\{instr; \} \xrightarrow{AST} [seq_{\perp}]}} \quad \boxed{\{ \} \xrightarrow{AST} [seq_{\perp}]}} \\ \boxed{\{continue; \xrightarrow{AST} [seq_c]}} \quad \boxed{continue; \} \xrightarrow{AST} seq_c} \\ \boxed{[seq_{\perp} continue; \xrightarrow{AST} [seq_c]}} \quad \boxed{[seq_c continue; \xrightarrow{AST} [seq_c]}} \\ \boxed{[seq_b continue; \xrightarrow{AST} [seq_b]}} \quad \boxed{[seq_r continue; \xrightarrow{AST} [seq_r]}} \\ \boxed{continue; seq_{\delta}] \xrightarrow{AST} seq_c} \quad \boxed{\{continue; \} \xrightarrow{AST} [seq_c]}} \end{array}$$

²Les instructions qui suivent une instruction de rupture dans une séquence ne sont jamais exécutées.

³Le pre-cooking permet de garantir que le code C en entrée respecte cette contrainte.

$\boxed{\{\text{break}; \xrightarrow{AST} [seq_b]}}$	$\boxed{\text{break}; \xrightarrow{AST} seq_b}$
$\boxed{[seq_\perp \text{ break}; \xrightarrow{AST} [seq_b]}}$	$\boxed{[seq_c \text{ break}; \xrightarrow{AST} [seq_c]}}$
$\boxed{[seq_b \text{ break}; \xrightarrow{AST} [seq_b]}}$	$\boxed{[seq_r \text{ break}; \xrightarrow{AST} [seq_r]}}$
$\boxed{\text{break}; seq_\delta] \xrightarrow{AST} seq_b}$	$\boxed{\{\text{break}; \xrightarrow{AST} [seq_b]}}$
$\boxed{\{\text{return}; \xrightarrow{AST} [seq_r]}}$	$\boxed{\text{return}; \xrightarrow{AST} seq_r}$
$\boxed{[seq_\perp \text{ return}; \xrightarrow{AST} [seq_r]}}$	$\boxed{[seq_c \text{ return}; \xrightarrow{AST} [seq_c]}}$
$\boxed{[seq_b \text{ return}; \xrightarrow{AST} [seq_b]}}$	$\boxed{[seq_r \text{ return}; \xrightarrow{AST} [seq_r]}}$
$\boxed{\text{return}; seq_\delta] \xrightarrow{AST} seq_r}$	$\boxed{\{\text{return}; \xrightarrow{AST} [seq_r]}}$
$\boxed{\{\text{return}(val); \xrightarrow{AST} [seq_r]}}$	$\boxed{\text{return}(val); \xrightarrow{AST} seq_r}$
$\boxed{[seq_\perp \text{ return}(val); \xrightarrow{AST} [seq_r]}}$	$\boxed{[seq_c \text{ return}(val); \xrightarrow{AST} [seq_c]}}$
$\boxed{[seq_b \text{ return}(val); \xrightarrow{AST} [seq_b]}}$	$\boxed{[seq_r \text{ return}(val); \xrightarrow{AST} [seq_r]}}$
$\boxed{\text{return}(val); seq_\delta] \xrightarrow{AST} seq_r}$	$\boxed{\{\text{return}(val); \xrightarrow{AST} [seq_r]}}$
$\boxed{[seq_\perp seq_\delta] \xrightarrow{AST} [seq_\delta]}$	$\boxed{[seq_c seq_\delta] \xrightarrow{AST} [seq_c]}$
$\boxed{[seq_b seq_\delta] \xrightarrow{AST} [seq_b]}$	$\boxed{[seq_r seq_\delta] \xrightarrow{AST} [seq_r]}$

Remarque 9.1 (Phénomène d'absorption) *Sous l'hypothèse de code correct (compilable), les éventuelles instructions continue, break et return ne peuvent apparaître que dans une structure à laquelle elles sont applicables.*

L'apparition d'une telle instruction dans une séquence signifie donc qu'une sur-structure va être interrompue; en particulier, toute instruction suivant une instruction de rupture dans une séquence n'est jamais exécutée.

Cette particularité justifie le phénomène d'absorption décrit par la règle $[seq_c seq_\delta] \xrightarrow{AST} [seq_c]$ par exemple.

De manière moins exhaustive, les principes d'étiquetage pour les structures de contrôles sont les suivants⁴ :

$$\boxed{\text{if } (val) [seq_\delta] \text{ else } [seq_{\delta'}] \xrightarrow{AST} if_{\delta \cup \delta'}}$$

⁴Rappelons que le pre-cooking permet de garantir que ces structures ont en paramètre une variable ou une constante.

$\text{switch } (val) \{ \text{case } cst_1 : seq_{\delta_1} \xrightarrow{AST} switch_{(\delta_1 \cup \dots \cup \delta_n) \cap \{c,r\}} \dots \text{case } cst_n : seq_{\delta_n} \}$
$\text{while } (val) seq_{\delta} \xrightarrow{AST} switch_{\delta}$
$\text{do } seq_{\delta} \text{ while } (val); \xrightarrow{AST} do_{\delta}$
$\text{for } (seq_{\perp}^1; val; seq_{\perp}^2) seq_{\delta}^3 \xrightarrow{AST} for_{\delta}$

On définit enfin la catégorie *bloc*, qui est une extension de *seq* (en particulier, toute séquence est un bloc); contrairement à *seq*, *bloc* peut contenir des structures de contrôle, et nécessite 3 positions d'indice pour refléter les instructions de rupture qu'il peut contenir. Seules quelques règles sont données ici pour illustrer le principe général d'étiquetage :

$[seq_{\perp} switch_{cr} \xrightarrow{AST} [bloc_{c\perp r}]$
$[seq_{\perp} for_{cbr} \xrightarrow{AST} [bloc_{\perp\perp r}]$

Notons que la deuxième règle reflète le fait que les instructions **continue** et **break** apparaissant dans une structure *for* sont applicables à cette structure, et ne sont pas promues vers les structures supérieures.

La définition de la catégorie *bloc* nous permet en fait de compléter les règles d'étiquetages pour les structures de contrôle; pour chacune des règles précédentes, explicitées pour des structures ne contenant que des séquences, on propose une nouvelle règle identique pour les structures contenant des blocs. On a par exemple pour les structures **if** la règle suivante :

$\text{if } (val) [bloc_{\gamma_1\beta_1\rho_1}^1] \text{ else } [bloc_{\gamma_2\beta_2\rho_2}^2] \xrightarrow{AST} if_{(\gamma_1 \cup \gamma_2)(\beta_1 \cup \beta_2)(\rho_1 \cup \rho_2)}$
--

9.3 Fonctions de traduction

9.3.1 Déclarations de fonctions

La traduction d'une fonction *C* est décrite par les schémas de règles suivants :

$\begin{array}{l} fun_{a0}(type\ id_G(type^1\ id_F^1, \dots, type^n\ id_F^n) [bloc_{\perp\perp r}]) \\ \xrightarrow{B} \quad out \leftarrow id_G(id_P^1, \dots, id_P^n) \hat{=} \\ \quad \text{PRE } id_P^1 \in TYPE^1 \wedge \dots \wedge id_P^n \in TYPE^n \text{ THEN} \\ \quad \text{VAR } ret_C, id_F^1, \dots, id_F^n \text{ IN} \\ \quad \quad id_F^1, \dots, id_F^n := id_P^1, \dots, id_P^n; \\ \quad \quad ret_C := FALSE; \\ \quad \quad out := TYPE; \\ \quad \quad bloc_{a0}([bloc_{\perp\perp r}]) \\ \quad \text{END} \\ \text{END} \end{array}$
$\begin{array}{l} fun_s(type\ id_G(type^1\ id_F^1, \dots, type^n\ id_F^n) [bloc_{\perp\perp r}]) \\ \xrightarrow{B} \quad out \leftarrow id_G(id_P^1, \dots, id_P^n) \hat{=} \\ \quad \text{PRE } id_P^1 \in TYPE^1 \wedge \dots \wedge id_P^n \in TYPE^n \text{ THEN} \\ \quad \quad out := TYPE \\ \quad \text{END} \end{array}$

$\frac{B}{\text{fun}_{a0}(\text{void } id_G(\text{type}^1 id_F^1, \dots, \text{type}^n id_F^n) [bloc_{\perp\perp r}])}$ <pre> out ← id_G(id_P^1, ..., id_P^n) ≐ PRE id_P^1 ∈ TYPE^1 ∧ ... ∧ id_P^n ∈ TYPE^n THEN VAR ret_C, out, id_F^1, ..., id_F^n IN id_F^1, ..., id_F^n := id_P^1, ..., id_P^n ; ret_C := FALSE ; bloc_{a0}([bloc_{\perp\perp r}]) END END </pre>
--

$\frac{B}{\text{fun}_s(\text{void } id_G(\text{type}^1 id_F^1, \dots, \text{type}^n id_F^n) [bloc_{\perp\perp r}])}$ <pre> out ← id_G(id_P^1, ..., id_P^n) ≐ PRE id_P^1 ∈ TYPE^1 ∧ ... ∧ id_P^n ∈ TYPE^n THEN skip END </pre>
--

Les paramètres formels sont ici renommés, puis recopiés en début d'opération dans des variables locales dont le nom est celui des paramètres formels de la fonction originale ; ceci permet de modéliser le passage de paramètre par valeur, i.e. de copies, et autorise la modification de ces copies, alors que le B interdit la modification des paramètres formels d'une opération⁵.

Les abstractions de référence comportent systématiquement la déclaration d'une variable locale ret_C , permettant de modéliser l'instruction de rupture **return** ; cette instruction doit obligatoirement apparaître dans le bloc d'instructions associé, comme indiqué par les indices dans la règle de traduction.

De manière générale, le paramètre éventuel associé à l'instruction **return** est mémorisé dans out , qui est soit le paramètre formel de sortie, soit une variable locale qui n'est pas utilisée par ailleurs. L'initialisation de out à une valeur non déterminée pour les fonctions retournant une valeur permet de modéliser les fonctions où l'exécution d'une instruction **return** correctement paramétrée n'est pas toujours garantie.

Remarque 9.2 (Traduction de prototypes purs) *Notons que si le corps de la fonction n'est pas connu, on peut décider soit de se limiter à une spécification du prototype, soit de définir une abstraction de référence sous la forme d'un miracle (cf. la remarque 3.1).*

9.3.2 Structures de contrôle

La prise en compte des délimiteurs de blocs respecte les schémas suivants, selon que la fonction var_{a0} (retournant les variables locales au bloc)⁶ retourne vide ou non :

$\frac{B}{\text{bloc}_{a0}([bloc_{\gamma\beta\rho}])}$ <pre> bloc_{a0}(bloc_{\gamma\beta\rho}) </pre>	$\frac{B}{\text{bloc}_{a0}([bloc_{\gamma\beta\rho}])}$ <pre> VAR var_{a0}(bloc_{\gamma\beta\rho}) IN bloc_{a0}(bloc_{\gamma\beta\rho}) END </pre>
---	---

⁵La sémantique des appels d'opérations du B indique que si une modification d'un paramètre formel était possible, cette modification se répercuterait sur la variable utilisée pour l'appel ; l'appel d'une telle opération avec une variable de la machine rendrait caduque les preuves de préservation de l'invariant, l'appel avec une constante serait contradictoire.

⁶Les règles associées à la fonction var_{a0} ne sont pas détaillées dans le cadre de cette étude.

Le principe de la traduction pour les structures *if* est le suivant :

$$\boxed{\begin{array}{l} \text{bloc}_{a0}(if_{\gamma\beta\rho}(val, bloc^1, bloc^2) bloc^3) \\ \xrightarrow{B} \quad \text{IF } val_{a0}(val) \neq 0 \text{ THEN } bloc_{a0}(bloc^1) \text{ ELSE } bloc_{a0}(bloc^2) \text{ END ;} \\ \quad \text{IF } \neg(\gamma \vee \beta \vee \rho) \text{ THEN } bloc_{a0}(bloc^3) \text{ END} \end{array}}$$

Nous utilisons, dans ce schéma de règles comme dans les suivants, des raccourcis d'écriture. Ainsi, l'expression $\text{IF } \neg(\gamma \vee \beta \vee \rho)$ correspond à une traduction de la forme :

$$\text{IF } \neg(cont_C = TRUE \vee break_C = TRUE \vee ret_C = TRUE)$$

$cont_C$ et $break_C$ sont des variables locales (fraîches) dont la déclaration et l'initialisation à *FALSE* sont générées lors de la traduction de la structure à laquelle elles se rapportent⁷ ; ret_C est la variable locale dont la déclaration et l'initialisation à *FALSE* sont générées lors de la traduction de la fonction (cf. le paragraphe 9.3.1).

Nous admettons dans de telles expressions que les variables et les expressions associées ne sont générées que si nécessaire ; en particulier, lorsque l'indice correspondant vaut \perp , le test est simplifié. Des règles plus explicites peuvent être facilement dérivées, mais elles ne sont pas détaillées ici en raison de la combinatoire importante ; ainsi, la traduction des structures *if* est détaillée par seize règles (absence ou présence d'une clause *else*, et des trois instructions de rupture), dont deux exemples sont fournis ci-dessous à titre d'illustration :

$$\begin{array}{l} \text{bloc}_{a0}(if_{\perp\perp\perp}(val, bloc^1, bloc^2) bloc^3) \\ \xrightarrow{B} \quad \text{IF } val_{a0}(val) \neq 0 \text{ THEN } bloc_{a0}(bloc^1) \text{ ELSE } bloc_{a0}(bloc^2) \text{ END ;} \\ \quad \text{bloc}_{a0}(bloc^3) \\ \\ \text{bloc}_{a0}(if_{c\perp\perp}(val, bloc^1) bloc^3) \\ \xrightarrow{B} \quad \text{IF } val_{a0}(val) \neq 0 \text{ THEN } bloc_{a0}(bloc^1) \text{ END ;} \\ \quad \text{IF } cont_C = FALSE \text{ THEN } bloc_{a0}(bloc^3) \text{ END} \end{array}$$

Les schémas de règles pour les autres structures de contrôle sont les suivants :

$$\boxed{\begin{array}{l} \text{bloc}_{a0}(switch_{\gamma\rho}(val, cst^1 : bloc^1 / \dots / cst^n : bloc^n / dflt : bloc^d) bloc^z) \\ \xrightarrow{B} \quad \text{VAR } \gamma \text{ IN} \\ \quad \gamma := FALSE ; \\ \quad \text{CASE } val_{a0}(val) \\ \quad \quad \text{EITHER } cst^1 \text{ THEN } bloc_{a0}(bloc^1) \\ \quad \quad \text{OR } \dots \\ \quad \quad \text{OR } cst^n \text{ THEN } bloc_{a0}(bloc^n) \\ \quad \quad \text{ELSE } bloc_{a0}(bloc^d) \\ \quad \text{END} \\ \quad \text{END ;} \\ \quad \text{IF } \neg(\gamma \vee \rho) \text{ THEN } bloc_{a0}(bloc^z) \text{ END} \\ \quad \text{END} \end{array}}$$

⁷Cette structure existe bien, par hypothèse de correction du code *C*.

$$\frac{}{bloc_{a0}(while_{\gamma\beta\rho}(val, bloc^1) bloc^z)}$$

$$\xrightarrow{B} \text{VAR } \gamma, \beta \text{ IN}$$

$$\quad \gamma, \beta := FALSE, FALSE ;$$

$$\quad \text{WHILE } val_{a0}(val) \neq 0 \wedge \neg(\gamma \vee \beta \vee \rho)$$

$$\quad \text{DO } bloc_{a0}(bloc^1)$$

$$\quad \quad \text{INVARIANT } tbd$$

$$\quad \quad \text{VARIANT } tbd$$

$$\quad \text{END}$$

$$\quad \text{END ;}$$

$$\quad \text{IF } \neg\rho \text{ THEN } (bloc^z) \text{ END}$$

$$\frac{}{bloc_{a0}(do_{\gamma\beta\rho}(val, bloc^1) bloc^z)}$$

$$\xrightarrow{B} \text{VAR } \gamma, \beta, fst \text{ IN}$$

$$\quad \gamma, \beta, fst := FALSE, FALSE, TRUE ;$$

$$\quad \text{WHILE } fst \vee (val_{a0}(val) \neq 0 \wedge \neg(\gamma \vee \beta \vee \rho))$$

$$\quad \text{DO } fst := FALSE ; bloc_{a0}(bloc^1)$$

$$\quad \quad \text{INVARIANT } tbd$$

$$\quad \quad \text{VARIANT } tbd$$

$$\quad \text{END}$$

$$\quad \text{END ;}$$

$$\quad \text{IF } \neg\rho \text{ THEN } (bloc^z) \text{ END}$$

$$\frac{}{bloc_{a0}(for_{\gamma\beta\rho}(bloc^1_{\perp}, val, bloc^2_{\perp}, bloc^3_{\gamma\beta\rho}) bloc^z)}$$

$$\xrightarrow{B} \text{VAR } \gamma, \beta \text{ IN}$$

$$\quad \gamma, \beta := FALSE, FALSE ;$$

$$\quad bloc_{a0}(bloc^1_{\perp}) ;$$

$$\quad \text{WHILE } (val_{a0}(val) \neq 0 \wedge \neg(\gamma \vee \beta \vee \rho))$$

$$\quad \text{DO}$$

$$\quad \quad bloc_{a0}(bloc^3_{\gamma\beta\rho}) ;$$

$$\quad \quad \text{IF } \neg(\gamma \vee \beta \vee \rho) \text{ THEN } bloc_{a0}(bloc^2) \text{ END}$$

$$\quad \quad \text{INVARIANT } tbd$$

$$\quad \quad \text{VARIANT } tbd$$

$$\quad \text{END}$$

$$\quad \text{END ;}$$

$$\quad \text{IF } \neg\rho \text{ THEN } (bloc^z) \text{ END}$$

9.3.3 Instructions de rupture

La traduction des instructions de rupture se fait selon les schémas suivants :

$$bloc_{a0}(\text{continue} ; bloc_{\gamma\beta\rho}) \xrightarrow{B} \gamma := TRUE$$

$$bloc_{a0}(\text{break} ; bloc_{\gamma\beta\rho}) \xrightarrow{B} \beta := TRUE$$

$$bloc_{a0}(\text{return} ; bloc_{\gamma\beta\rho}) \xrightarrow{B} \rho := TRUE$$

$$bloc_{a0}(\text{return } val ; bloc_{\gamma\beta\rho}) \xrightarrow{B} out := val ; \rho := TRUE$$

9.3.4 Appels de fonctions et affectations

Le pre-cooking permet de garantir qu'une fonction ou un opérateur ne sont appelés qu'avec des paramètres qui sont des variables ou des constantes ; les schémas de traduction pour les appels de fonctions purs et les affectations sont les suivants :

$$\boxed{ \text{bloc}_{a0}(\text{fn_G}(\text{val}^1, \dots, \text{val}^n); \text{bloc}_{\alpha\beta\gamma}) \xrightarrow{B} \text{fn_G}(\text{val}_{a0}(\text{val}^1), \dots, \text{val}_{a0}(\text{val}^n)); \text{bloc}_{a0}(\text{bloc}_{\alpha\beta\gamma}) }$$

$$\boxed{ \text{bloc}_{a0}(\text{var}=\text{val}); \text{bloc}_{\alpha\beta\gamma} \xrightarrow{B} \text{aff}_{a0}(\text{var}, \text{val}_{a0}(\text{val})); \text{bloc}_{a0}(\text{bloc}_{\alpha\beta\gamma}) }$$

$$\boxed{ \text{bloc}_{a0}(\text{var}=\text{fn_G}(\text{val}^1, \dots, \text{val}^n); \text{bloc}_{\alpha\beta\gamma}) \xrightarrow{B} \text{VAR } \text{tmp} \text{ IN} \\ \text{tmp} \leftarrow \text{fn_G}(\text{val}_{a0}(\text{val}^1), \dots, \text{val}_{a0}(\text{val}^n)); \\ \text{aff}_{a0}(\text{var}, \text{tmp}); \\ \text{END}; \\ \text{bloc}_{a0}(\text{bloc}_{\alpha\beta\gamma}) }$$

$$\boxed{ \text{bloc}_{a0}(\text{var}=\text{ops}(\text{val}^1, \dots, \text{val}^n); \text{bloc}_{\alpha\beta\gamma}) \xrightarrow{B} \text{VAR } \text{tmp} \text{ IN} \\ \text{tmp} \leftarrow \text{ops}(\text{val}_{a0}(\text{val}^1), \dots, \text{val}_{a0}(\text{val}^n)); \\ \text{aff}_{a0}(\text{var}, \text{tmp}); \\ \text{END}; \\ \text{bloc}_{a0}(\text{bloc}_{\alpha\beta\gamma}) }$$

La fonction de traduction pour les valeurs, val_{a0} , exploite le renommage effectué par le pre-cooking. Les variables postfixées par le symbole $_G$ sont les variables globales, déclarées comme des identifiants applicables à une fonction de valuation ; si ce postfixe n'apparaît pas, les variables correspondent à des variables locales du code C original, à des variables locales générées par le pre-cooking, ou encore à des paramètres formel.

Les schémas associés à val_{a0} sont donc les suivants, avec acc une expression d'accès aux champs :

$$\boxed{ \text{val}_{a0}(\text{cst}) \xrightarrow{B} \text{cst} }$$

$$\boxed{ \text{val}_{a0}(\text{tmp}_i) \xrightarrow{B} \text{tmp}_i }$$

$$\boxed{ \text{val}_{a0}(\text{var_L}) \xrightarrow{B} \text{var_L} }$$

$$\boxed{ \text{val}_{a0}(\text{var_F}) \xrightarrow{B} \text{var_F} }$$

$$\boxed{ \text{val}_{a0}(\text{var_G}) \xrightarrow{B} \text{type.adr}(\text{type.var_G}) }$$

$$\boxed{ \text{val}_{a0}(*\text{acc}) \xrightarrow{B} \text{type.mem}(\text{val}_{a0}(\text{acc})) }$$

$$\boxed{ \text{val}_{a0}(\text{acc.field_F}) \xrightarrow{B} \text{type.rfield}(\text{val}_{a0}(\text{acc})) }$$

Exemple 9.3 (Lecture d'une structure de structure) *En réutilisant la structure définie dans l'exemple 6.3, on a :*

$$\begin{array}{l}
y=r.p1.x; \\
\frac{P_c}{\longrightarrow} \quad *y_L=*r_G.p1_F.x_F; \\
\frac{B}{\longrightarrow} \quad aff_{a0}(*y_L, val_{a0}(*r_G.p1_F.x_F)) \\
\frac{B}{\longrightarrow} \quad aff_{a0}(*y_L, pt.rx(val_{a0}(*r_G.p1_F))) \\
\frac{B}{\longrightarrow} \quad aff_{a0}(*y_L, pt.rx(rect.rp1(val_{a0}(*r_G)))) \\
\frac{B}{\longrightarrow} \quad aff_{a0}(*y_L, pt.rx(rect.rp1(mem(val_{a0}(r_G)))))) \\
\frac{B}{\longrightarrow} \quad aff_{a0}(*y_L, pt.rx(rect.rp1(mem(adr(r_G))))))
\end{array}$$

Exemple 9.4 (Lecture d'une structure auto-référentielle) *En réutilisant la structure de liste définie dans l'exemple 6.7, et en faisant abstraction du préfixe de type (valant toujours link ici) on a :*

$$\begin{array}{l}
x=head->next->value; \\
\frac{P_c}{\longrightarrow} \quad *x_L=>(*head_G.next_F).value_F; \\
\frac{B}{\longrightarrow} \quad aff_{a0}(*x_L, val_{a0}(*head_G.next_F).value_F)) \\
\frac{B}{\longrightarrow} \quad aff_{a0}(*x_L, rvalue(val_{a0}(*head_G.next_F))) \\
\frac{B}{\longrightarrow} \quad aff_{a0}(*x_L, rvalue(mem(val_{a0}(*head_G.next_F)))) \\
\frac{B}{\longrightarrow} \quad aff_{a0}(*x_L, rvalue(mem(rnext(val_{a0}(*head_G)))))) \\
\frac{B}{\longrightarrow} \quad aff_{a0}(*x_L, rvalue(mem(rnext(mem(val_{a0}(head_G)))))) \\
\frac{B}{\longrightarrow} \quad aff_{a0}(*x_L, rvalue(mem(rnext(mem(adr(head_G))))))
\end{array}$$

La fonction de traduction aff_{a0} exploite également le renommage :

$$\begin{array}{l}
\boxed{aff_{a0}(tmp_i, val) \xrightarrow{B} tmp_i := val_{a0}(val)} \\
\boxed{aff_{a0}(var_L, val) \xrightarrow{B} var_L := val_{a0}(val)} \\
\boxed{aff_{a0}(var_F, val) \xrightarrow{B} var_F := val_{a0}(val)} \\
\boxed{aff_{a0}(var_G, val) \xrightarrow{B} type.Wadr(type.adr(type.var_G), val_{a0}(val))} \\
\boxed{aff_{a0}(*acc, val) \xrightarrow{B} type.Wmem(val_{a0}(acc), val_{a0}(val))} \\
\boxed{\frac{B}{\longrightarrow} \quad \begin{array}{l} aff_{a0}(*acc[.field_F]^*.field^z_F, val) \\ aff_{a0}(*acc[.field_F]^*, wfield^z(val_{a0}(*acc[.field_F]^*), val) \end{array}}
\end{array}$$

Exemple 9.5 (Ecriture d'une structure de structure) *En réutilisant la structure définie dans l'exemple 6.3, on a :*

$$\begin{array}{l}
r.p1.x=y; \\
\frac{P_c}{\longrightarrow} \quad *r_G.p1_F.x_F=*y_L; \\
\frac{B}{\longrightarrow} \quad aff_{a0}(*r_G.p1_F.x_F, val_{a0}(*y_L)) \\
\frac{B}{\longrightarrow} \quad aff_{a0}(*r_G.p1_F, wx(val_{a0}(*r_G.p1_F), val_{a0}(*y_L))) \\
\frac{B}{\longrightarrow} \quad aff_{a0}(*r_G, wp1(val_{a0}(*r_G), wx(val_{a0}(*r_G.p1_F), val_{a0}(*y_L)))) \\
\frac{B}{\longrightarrow} \quad rect.Wmem(val_{a0}(r_G), wp1(val_{a0}(*r_G), wx(val_{a0}(*r_G.p1_F), val_{a0}(*y_L)))) \\
\frac{B}{\longrightarrow} \quad rect.Wmem(rect.adr(r_G), rect.wp1(rect.mem(rect.adr(r_G)), \\
\quad pt.wx(rect.rp1(mem(adr(r_G))), mem(y_L))))
\end{array}$$

Exemple 9.6 (Ecriture d'une structure auto-référentielle) *En réutilisant la structure de liste définie dans l'exemple 6.7, et en faisant abstraction du préfixe de type (valant toujours link ici) on a :*

$$\begin{aligned}
& \text{head} \rightarrow \text{next} \rightarrow \text{value} = \text{x}; \\
& \xrightarrow{P_c} *(*\text{head_G.next_F}).\text{value_F} = \text{x_L}; \\
& \xrightarrow{B} \text{aff}_{a0}(*(*\text{head_G.next_F}).\text{value_F}, \text{val}_{a0}(\text{x_L})) \\
& \xrightarrow{B} \text{aff}_{a0}(*(*\text{head_G.next_F}).\text{value_F}, \text{mem}(\text{x_L})) \\
& \xrightarrow{B} \text{Wmem}(\text{val}_{a0}(*(*\text{head_G.next_F})), \text{wfield}(\text{val}_{a0}(*(*\text{head_G.next_F})), \text{mem}(\text{x_L}))) \\
& \xrightarrow{B} \text{Wmem}(\text{mem}(\text{val}_{a0}(*\text{head_G.next_F})), \\
& \quad \text{wfield}(\text{mem}(\text{val}_{a0}(*\text{head_G.next_F})), \text{mem}(\text{x_L}))) \\
& \xrightarrow{B} \text{Wmem}(\text{mem}(\text{rnext}(\text{val}_{a0}(*\text{head_G}))), \\
& \quad \text{wfield}(\text{rnext}(\text{mem}(\text{val}_{a0}(*\text{head_G}))), \text{mem}(\text{x_L}))) \\
& \xrightarrow{B} \text{Wmem}(\text{mem}(\text{rnext}(\text{mem}(\text{val}_{a0}(\text{head_G})))), \\
& \quad \text{wfield}(\text{rnext}(\text{mem}(\text{mem}(\text{val}_{a0}(\text{head_G}))), \text{mem}(\text{x_L}))) \\
& \xrightarrow{B} \text{Wmem}(\text{mem}(\text{rnext}(\text{mem}(\text{adr}(\text{head_G})))), \\
& \quad \text{wfield}(\text{rnext}(\text{mem}(\text{mem}(\text{adr}(\text{head_G}))), \text{mem}(\text{x_L})))
\end{aligned}$$

9.4 Organisation des modules

Les fonctions de traduction présentées dans la section précédente permettent d'obtenir une collection d'objets B permettant la modélisation du code C analysé.

Il reste cependant nécessaire de pouvoir structurer ces objets en modules ; cette contrainte résulte de limitations posées par la *Méthode B*, telles que l'interdiction à une opération d'une machine abstraite de faire appel à une opération de la même machine⁸.

La traduction d'un code C doit donc comprendre la génération d'un graphe de dépendance, à partir duquel l'architecture des modules peut être dérivée. En substance, si un objet A fait appel à un objet B , alors ils ne peuvent être définis dans le même module ; nous pouvons donc linéariser le graphe de dépendance, ce qui conduit à générer le plus grand nombre possible de modules (un par objet), ou stratifier, pour générer le plus petit nombre possible de modules. L'algorithme suivant permet de dériver une stratification du graphe :

```

Insérer les objets en couche 0
Pour  $n$  allant de 0 au nombre d'objets, faire :
  Pour  $A$  parcourant les objets de la couche  $n$ , faire :
    Si  $A$  appelle  $B$  de la couche  $m \leq n$  alors déplacer  $B$  vers la couche  $n + 1$ 

```

⁸Notons que cette limitation, comme quelques autres, n'a pas de justification théorique et reflète plutôt une démarche intellectuelle préconisée par l'auteur.

Chapitre 10

Synthèse sur l’outil *ABCBA*

L’outil *ABCBA*, pour *AB*straction of *C* in *B* Assistant, doit permettre l’abstraction de code *C* en une spécification *B* que l’on peut prouver adéquate au code analysé.

Ce chapitre décrit les éléments de spécification de cet outil.

10.1 Architecture

Dans le cadre de cette étude, quatre grands blocs fonctionnels sont mis en évidence pour décrire les services qui doivent être offerts par *ABCBA* :

- Pre-cooking : la réécriture du code *C* initial en un code *C* observationnellement équivalent, mais dont la forme est normale vis-à-vis d’un système de réécriture.
- Traduction : la traduction du code *C* en forme normale en un ensemble de machines abstraites (appelées spécifications) et de raffinement (appelés abstractions de référence) modélisant son comportement.
- Post-cooking : la réécriture de l’abstraction de référence, constituée de modules *B*, en une abstraction intermédiaire supposée équivalente ; l’équivalence peut ensuite être formellement prouvée dans le cadre d’une preuve d’adéquation. Le post-cooking n’a pas été abordé dans cette étude.
- Edition : la création d’abstractions intermédiaires, et la modification des abstractions intermédiaires et des spécifications.
- Preuve : l’environnement de preuve permettant de démontrer l’adéquation entre les spécifications et les abstractions de référence qui leur sont associées ; les abstractions intermédiaires permettent de décomposer cette preuve.

10.2 Choix de modélisation et limitations

Dans le cadre de cette étude, différentes approches pour la modélisation ont été retenues, et certaines limitations ont été acceptées. La liste suivante détaille ces choix et leur associe un indicateur estimant leur influence sur l’étude telle que proposée ici ; **** signifie une remise en cause fondamentale de l’étude, * correspond à une modification aisée ou un petit complément :

- **** Seuls les programmes séquentiels sont considérés.
- *** Les pointeurs de fonctions ne sont pas considérés.
- *** Seuls les programmes sans `goto` sont considérés.
- *** Seuls les programmes non récursifs sont considérés¹.

¹A noter que cette limitation résulte d’une limitation de l’outil *Click’n’Prove/B4Free v2.0* n’implémentant pas les opérations récursives définies par la théorie du *B*.

- ** Les programmes analysés sont supposés compilables.
- ** Seuls les programmes sans type-punning sont considérés.
- ** Les manipulations de types (cast) par pointeurs ne sont pas considérées.
- ** Les pointeurs génériques définis par `void *` ne sont pas considérés.
- ** Seules les fonctions admettant un nombre fixe de paramètres sont considérées.
- ** Chaque type manipulé est supposé disposer d'un espace mémoire propre et infini.
- * Seuls les programmes traitant des expressions entières sont considérés.
- * Les déréférencements de pointeurs ne sont pas considérés.
- * Les référencements d'ordre supérieur à 2 ne sont pas considérés.
- * Les types unions ne sont pas considérés.
- * Seuls les tableaux uni-dimensionnels ou bi-dimensionnels sont considérés.

10.3 Pre-cooking

Le pre-cooking vise à la réécriture d'un code C en un code équivalent, mais dans lequel :

- Les identifiants sont renommés, afin d'éviter des conflits avec les mots-clés du B .
- Les expressions complexes sont décomposées.
- Les effets de bords sont éliminés.
- Les structures `switch` sont mises dans une forme standard.
- Les variables et tableaux sont remplacés par des pointeurs.

La spécification détaillée du pre-cooking est fournie dans le chapitre 8.

L'implémentation du pre-cooking peut être réalisée dans de nombreux environnements dédiés à l'analyse lexicale et/ou à la réécriture ; on notera entre autres les outils *ANTLR* (analyse lexicale, construction d'un AST, et réécriture d'arbre) ou encore *Elan* (réécriture), voire l'extraction du compilateur *gcc* des fonctions d'analyse et leur modification.

Par ailleurs, des outils spécifiques de réécriture de code C sont disponibles, parmi lesquels *CIL* (*C Intermediate Language*), dont les fonctionnalités sont proches de celles attendues².

10.4 Traduction

La traduction produit, à partir du code C réécrit par pre-cooking, un ensemble de spécifications (machines abstraites) et d'abstractions de référence (raffinements). Elle vise plus particulièrement à la modélisation des variables globales et des fonctions du code C , qui constituent les objets justifiant l'existence des modules.

Cette traduction repose sur de nombreux choix de modélisation ; en particulier, un modèle générique est proposé pour les opérations, les types entiers et les pointeurs. La traduction permet aussi de modéliser la structure de contrôle du C et les instructions de rupture, qui sont sans équivalent en B . Enfin, la traduction propose une architecture de modules B compatible des contraintes de visibilité imposées par la *Méthode B*.

Remarque 10.1 (Modèles et normes) *Notons que les modèles pourront dépendre de la norme considérée pour la définition du langage C (ANSI C, GNU C, etc.), du compilateur considéré voire des options de compilation, et du microprocesseur. Les modèles devront autant que possible expliciter les références considérées.*

On attend de la traduction une représentation fine du comportement du code C , sous les hypothèses

² *CIL* dans sa version actuelle a cependant la particularité de générer de nombreux *goto*.

de modélisation présentées précédemment, dans l'abstraction de référence. Les opérations implicites ou explicites du C sont modélisées.

La spécification obtenue par traduction peut se limiter à la définition des prototypes de fonctions.

La spécification détaillée du processus de traduction est fournie dans le chapitre 9.

Il est souhaitable qu'en complément, des bibliothèques de spécifications des principales fonctions du C soient fournies.

10.5 Edition

L'objectif de l'édition est de permettre à l'utilisateur de créer des abstractions intermédiaires, de les modifier, et de modifier les spécifications. La modification des spécifications permet de mettre en évidence et d'exporter les propriétés abstraites du code C considéré; c'est à l'utilisateur de définir les propriétés pertinentes qu'il souhaite mettre en évidence.

L'abstraction de référence, produite par la traduction, ne peut être modifiée³; elle constitue le seul lien avec le code C initial.

Il est souhaitable que des interfaces dédiées soient offertes, permettant l'édition des modules. Ces interfaces peuvent masquer les éléments non pertinents pour l'utilisateur, assurer un maintien automatique de cohérence lors des modifications (notamment l'ajout d'abstractions intermédiaires), et permettre de définir des compléments au langage, par exemple en offrant la possibilité de définir des invariants locaux ou des post-conditions.

10.6 Preuve

Les spécifications doivent être prouvées adéquates vis à vis des abstractions de référence; ceci correspond à une preuve de raffinement des spécifications par les abstractions de référence.

Les obligations de preuves sont générées automatiquement; elles correspondent aux obligations de preuves définies par la théorie du B , complétées par les obligations de preuves différées, relatives à l'existence des objets manipulés⁴.

L'environnement de preuve doit offrir un bon niveau d'automatisation. Des bibliothèques d'assertions peuvent être définies et incorporées aux modèles générique pour faciliter les preuves.

L'utilisation de l'environnement *Click'n'Prove/B4Free v2.0* est la solution retenue.

³A l'exception des variants et invariants dans les substitutions *WHILE*.

⁴Selon l'outil utilisé, les obligations de preuve pourront également avoir à inclure la preuve des delta-lemmes, i.e. de respect des conditions de validité des définitions.

Chapitre 11

Conclusion

Cette étude propose une démarche d'abstraction du code C en modules B ; elle décrit des modèles et des processus qui permettent de dériver, de manière automatique, une abstraction de référence représentant finement le code analysé ainsi qu'une spécification générique élémentaire.

L'étude, dans les limites définies par les choix de modélisation précédemment exposés, prouve la faisabilité du processus d'abstraction ; elle constitue également une nouvelle démarche de définition d'une sémantique formelle du langage C .

Notons cependant que si le pre-cooking et la traduction permettent effectivement de proposer un modèle formel d'un code C de manière automatique, l'identification de propriétés pertinentes dans une spécification et la preuve d'adéquation entre cette spécification et le code C restent indécidables dans le cas général. De même, il faut noter que la traduction ne propose évidemment pas de génération automatique de variant et d'invariant de boucle.

De manière plus pragmatique, cela signifie que la possibilité de traduire un code C en B ne constitue pas une garantie que des propriétés intéressantes pourront être prouvées. Il nous faut considérer la possibilité de simplifier les modèles génériques proposés ici, pour les adapter au niveau minimum de complexité nécessaire permettant de représenter le code analysé – voire de paramétrer le niveau modélisation objet par objet en fonction de la propriété que l'on cherche à prouver.

Dans le cadre de l'étude, certaines limitations et particularités de l'outil *Click'n'Prove/B4Free v2.0* se traduisent par des contraintes importantes ; on notera notamment :

- L'interdiction de faire deux appels parallèles à des opérations d'une même machine.
- L'interdiction d'utiliser les substitutions WHILE en dehors d'une implémentation – se traduisant par l'impossibilité de manipuler, dans le processus d'abstraction proposé, les boucles.
- Une interprétation du type-checking et des règles de preuves proposés par le B .
- L'absence d'implémentation des opérations récursives.

D'autres implémentations de la *Méthode B* existent, ne présentant pas forcément ces mêmes limitations, mais n'ont pu être évaluées.

De même, la *Méthode B* elle-même impose des restrictions qui ne semblent pas justifiées par la théorie du B , mais génèrent de nouvelles contraintes ; on notera en particulier :

- L'interdiction d'utiliser des substitutions séquentielles dans les machines abstraites.
- L'interdiction, pour une opération, de faire appel à une autre opération.
- L'interdiction de définir des opérations récursives ailleurs que dans une implémentation.
- La non explicitation de certaines obligations de preuves différées.

Des évolutions de la *Méthode B* et des outils l'implémentant apparaissent souhaitables préalablement à toute exploitation sérieuse de la démarche d'abstraction proposée dans ce document.

Bibliographie

- [KR90] B.W. Kernighan, D.M. Ritchie. *Le langage C (deuxième édition)*. Masson – Prentice Hall, avril 1990.
- [Abr96] J.-R. Abrial. *The B Book – Assigning Programs To Meanings*. Cambridge University Press, August 1996.
- [Lan96] K. Lano. *The B Language and Method – A Guide to Practical Formal Development*. Springer, 1996.
- [Abr99] J.-R. Abrial. *Event Driven System Construction*. April 1999.
- [Abr00] J.-R. Abrial. *B : 2000 et plus (v3.2)*. Janvier 2000.
- [Abr01] J.-R. Abrial. *Event Driven Sequential Program Construction*. Janvier 2001.
- [Mat01] *Event B Reference Manual*. MATISSE Project, June 2001.
- [CM02] D. Cansell, D. Méry. *Development of recursively defined functions using the B event-driven approach*. B day, January 2002.
- [AM02] J.-R. Abrial, L. Mussat. *On Using Conditionnal Definitions in Formal Theories*. (Bert Editor) LNCS 2272 Springer (ZB 2002).
- [Kai00] A.-J. Kaijanaho. *The formal method known as B and a sketch for its implementation*. Master's Thesis in Information Technology, University of Jyväskylä, December 2002.
- [BP04] D. Bert, M.-L. Potet. *Spécification en B – Support de cours*. Ecole des Jeunes Chercheurs en Programmation 2004, juin 2004.
- [AtB] *Documentation de l'Atelier B*. Clearsy.
- [CIL] G. Necula, S. McPeak, W. Weimer, B. Liblit, R. To, A. Bhargava. *C Intermediate Language Documentation* University of Berkeley, 2001.

Annexe A

Modélisations des variables, pointeurs et structures

Cette annexe contient les modélisations finales pour les catégories suivantes :

- Entiers,
- Registres,
- Structures,
- Pointeurs sur type,
- Pointeurs sur pointeur.

A.1 Entiers

```

MACHINE  $genint_s(minval, maxval)$ 
CONSTRAINTS  $minval \in \mathbb{Z} \wedge maxval \in \mathbb{Z} \wedge minval < maxval \wedge$ 
               $\exists n \cdot (n \in \mathbb{N}_1 \wedge 2 * n = maxval - minval + 1)$ 
CONSTANTS  $RANGE$ 
PROPERTIES  $RANGE = minval..maxval$ 
OPERATIONS
   $out \leftarrow cast(x) \hat{=} \text{PRE } x \in RANGE \text{ THEN } out := x \text{ END}$ 
END

```

```

REFINEMENT  $genint_{a0}(minval, maxval)$ 
REFINES  $genint_s$ 
CONSTANTS  $size, smd$ 
PROPERTIES  $size = maxval - minval + 1 \wedge$ 
             $smd = \lambda n \cdot (n \in \mathbb{N} \mid n \bmod size) \cup$ 
             $\lambda n \cdot (n \in \mathbb{Z}_1 \mid n + size * \min(\{m \mid m \in \mathbb{N} \wedge n + size * m \geq 0\}))$ 
OPERATIONS
   $out \leftarrow cast(x) \hat{=}$ 
  PRE  $x \in \mathbb{Z}$  THEN
    IF  $smd(x) \in RANGE$  THEN  $out := smd(x)$  ELSE  $out := smd(x) - size$  END
  END
END

```

Exemple A.1 (Type *SCHAR*)

```

MACHINE  $schar_s$ 
EXTENDS  $schar.genint_s(-128, 127)$ 
END

```

Remarque A.1 (Définition des entiers implémentables) Click'n'Prove/B4Free v2.0 définit l'ensemble $INT = -2147483647..2147483647$. Ces deux bornes sont les valeurs entières extrêmes manipulables par l'outil; l'instanciation $genint_s(-2147483648, 2147483647)$ est donc refusée en raison du dépassement.

A.2 Registres

```

MACHINE registers
CONSTANTS minreg, maxreg, REG
PROPERTIES minreg = -2147483648 ∧ maxreg = 2147483647 ∧ REG = minreg..maxreg
OPERATIONS
  out ← cast(x) ≐ PRE x ∈ REG THEN out := x END ;
  out ← sum(x, y) ≐ PRE x ∈ REG ∧ y ∈ REG ∧ x + y ∈ REG THEN out := x + y END ;
  out ← sub(x, y) ≐ PRE x ∈ REG ∧ y ∈ REG ∧ x - y ∈ REG THEN out := x - y END ;
  out ← mul(x, y) ≐ PRE x ∈ REG ∧ y ∈ REG ∧ x * y ∈ REG THEN out := x * y END ;
  out ← div(x, y) ≐ PRE x ∈ REG ∧ y ∈ REG ∧ y ≠ 0 ∧ x/y ∈ REG
    THEN out := x/y END ;
END

```

```

REFINEMENT registera0
REFINES registers
CONSTANTS size, smd
PROPERTIES size = maxreg - minreg + 1
           smd = λn · (n ∈ ℕ | n mod size) ∪
           λn · (n ∈ ℤ1 | n + size * min({m | m ∈ ℕ ∧ n + size * m ≥ 0}))
OPERATIONS
  out ← cast(x) ≐
    PRE x ∈ ℤ THEN
      IF smd(x) ∈ REG THEN out := smd(x) ELSE out := smd(x) - size END
    END
  out ← sum(x, y) ≐
    PRE x ∈ REG ∧ y ∈ REG THEN
      IF smd(x + y) ∈ REG THEN out := smd(x + y) ELSE out := smd(x + y) - size END
    END
  out ← sub(x, y) ≐
    PRE x ∈ REG ∧ y ∈ REG THEN
      IF smd(x - y) ∈ REG THEN out := smd(x - y) ELSE out := smd(x - y) - size END
    END
  out ← mul(x, y) ≐
    PRE x ∈ REG ∧ y ∈ REG THEN
      IF smd(x * y) ∈ REG THEN out := smd(x * y) ELSE out := smd(x * y) - size END
    END
  out ← div(x, y) ≐
    PRE x ∈ REG ∧ y ∈ REG ∧ y ≠ 0 THEN
      IF smd(x/y) ∈ REG THEN out := smd(x/y) ELSE out := smd(x/y) - size END
    END
END

```

Remarque A.2 (Définition des entiers implémentables) Comme indiqué dans la remarque A.1, la propriété $\text{minreg} = -2147483648$ est refusée par Click'n'Prove/B4Free v2.0.

A.3 Structures

```

MACHINE struct2s( $T_1, T_2$ )
CONSTANTS TYPE, r1, r2, w1, w2
PROPERTIES
  TYPE =  $T_1 \times T_2 \wedge$ 
  r1 = prj1( $T_1, T_2$ )  $\wedge$ 
  r2 = prj2( $T_1, T_2$ )  $\wedge$ 
  w1 = prj2(TYPE,  $T_1$ )  $\otimes$  (prj1(TYPE,  $T_1$ ); r2)  $\wedge$ 
  w2 = (prj1(TYPE,  $T_2$ ); r1)  $\otimes$  prj2(TYPE,  $T_2$ )
END

```

```

MACHINE struct3s( $T_1, T_2, T_3$ )
CONSTANTS TYPE, r1, r2, r3, w1, w2, w3
PROPERTIES
  TYPE =  $(T_1 \times T_2) \times T_3 \wedge$ 
  r1 = prj1( $T_1 \times T_2, T_3$ ); prj1( $T_1, T_2$ )  $\wedge$ 
  r1 = prj1( $T_1 \times T_2, T_3$ ); prj2( $T_1, T_2$ )  $\wedge$ 
  r3 = prj2( $T_1 \times T_2, T_3$ )  $\wedge$ 
  w1 = (prj2((TYPE,  $T_1$ )  $\otimes$  (prj1(TYPE,  $T_1$ ); r2))  $\otimes$  (prj1(TYPE,  $T_1$ ); r3))  $\wedge$ 
  w2 = ((prj1(TYPE,  $T_2$ ); r1)  $\otimes$  prj2(TYPE,  $T_2$ ))  $\otimes$  (prj1(TYPE,  $T_2$ ); r3)  $\wedge$ 
  w3 = ((prj1(TYPE,  $T_3$ ); r1)  $\otimes$  (prj1(TYPE,  $T_3$ ); r2))  $\otimes$  prj2(TYPE,  $T_3$ )  $\wedge$ 
END

```

```

MACHINE struct4s( $T_1, T_2, T_3, T_4$ )
CONSTANTS TYPE, r1, r2, r3, r4, w1, w2, w3, w4
PROPERTIES
  TYPE =  $(T_1 \times T_2) \times (T_3 \times T_4) \wedge$ 
  r1 = prj1( $T_1 \times T_2, T_3 \times T_4$ ); prj1( $T_1, T_2$ )  $\wedge$ 
  r2 = prj1( $T_1 \times T_2, T_3 \times T_4$ ); prj2( $T_1, T_2$ )  $\wedge$ 
  r3 = prj2( $T_1 \times T_2, T_3 \times T_4$ ); prj1( $T_3, T_4$ )  $\wedge$ 
  r4 = prj2( $T_1 \times T_2, T_3 \times T_4$ ); prj2( $T_3, T_4$ )  $\wedge$ 
  w1 = (prj2(TYPE,  $T_1$ )  $\otimes$  (prj1(TYPE,  $T_1$ ); r2))  $\otimes$ 
      ((prj1(TYPE,  $T_1$ ); r3)  $\otimes$  (prj1(TYPE,  $T_1$ ); r4))  $\wedge$ 
  w2 = ((prj1(TYPE,  $T_2$ ); r1)  $\otimes$  prj2(TYPE,  $T_2$ ))  $\otimes$ 
      ((prj1(TYPE,  $T_2$ ); r3)  $\otimes$  (prj1(TYPE,  $T_2$ ); r4))  $\wedge$ 
  w3 = ((prj1(TYPE,  $T_3$ ); r1)  $\otimes$  (prj1(TYPE,  $T_3$ ); r2))  $\otimes$ 
      (prj2(TYPE,  $T_3$ )  $\otimes$  (prj1(TYPE,  $T_3$ ); r4))  $\wedge$ 
  w3 = ((prj1(TYPE,  $T_4$ ); r1)  $\otimes$  (prj1(TYPE,  $T_4$ ); r2))  $\otimes$ 
      ((prj1(TYPE,  $T_4$ ); r3)  $\otimes$  prj2(TYPE,  $T_4$ ))  $\wedge$ 
END

```

A.4 Pointeurs sur type

```

MACHINE ptrtps(TYPE, IDP)
VARIABLES mem, lst, adr
INVARIANT mem ∈ ℕ1 ↔ TYPE ∧ lst ∈ ℕ1 ↔ ℕ1 ∧ adr ∈ IDP → ℕ ∧
  lst ⊆ geq ∧
  ∀ x · (x ∈ dom(lst) ⇒ x..lst(x) ∩ dom(lst) = {x}) ∧
  ∀ x · (x ∈ dom(lst) ⇒ x..lst(x) ⊆ dom(mem)) ∧
  ∀ x · (x ∈ dom(mem) ⇒ ∃ y · (y ∈ dom(lst) ∧ x ∈ y..lst(y)))
INITIALISATION
ANY m, l, a WHERE m ∈ ℕ1 ↔ TYPE ∧ l ⊆ ℕ1 < geq ∧ a ∈ IDP → ℕ ∧
  ∀ x · (x ∈ dom(l) ⇒ x..l(x) ∩ dom(l) = {x}) ∧
  ∀ x · (x ∈ dom(l) ⇒ x..l(x) ⊆ dom(m)) ∧
  ∀ x · (x ∈ dom(m) ⇒ ∃ y · (y ∈ dom(l) ∧ x ∈ y..l(y)))
THEN mem := m || lst := l || adr := a END
OPERATIONS
out ← difadr(i, j) ≐ PRE i ∈ IDP ∧ j ∈ IDP ∧ ∃ x · (x ∈ dom(lst) ∧ adr[{i, j}] ⊆ x..lst(x))
  THEN out := adr(i) - adr(j) END ;
out ← gtradr(i, j) ≐ PRE i ∈ IDP ∧ j ∈ IDP ∧ ∃ x · (x ∈ dom(lst) ∧ adr[{i, j}] ⊆ x..lst(x))
  THEN out := bool(adr(i) > adr(j)) END ;
out ← malloc(s) ≐ PRE s ∈ ℕ1 THEN
  ANY a, v WHERE a ∈ ℕ1 ∧
    ∀ x · (x ∈ dom(lst) ⇒ a..a + s - 1 ∩ x..lst(x) = ∅) ∧
    v ∈ a..(a + s - 1) → TYPE THEN
    out := a || lst(a) := a + s - 1 || mem := mem <+ v
  END
END ;
imalloc ≐ ANY m, l WHERE m ∈ ℕ1 ↔ TYPE ∧ mem ⊆ m ∧ l ⊆ geq ∧ lst ⊆ l ∧
  ∀ x · (x ∈ dom(l) ⇒ x..l(x) ∩ dom(l) = {x}) ∧
  ∀ x · (x ∈ dom(l) ⇒ x..l(x) ⊆ dom(m)) ∧
  ∀ x · (x ∈ dom(m) ⇒ ∃ y · (y ∈ dom(l) ∧ x ∈ y..l(y)))
  THEN mem := m || lst := l END ;
free(a) ≐ PRE a ∈ dom(lst) THEN mem := a..lst(a) < mem || lst := {a} < lst END ;
ifree ≐ ANY m, l WHERE m ⊆ mem ∧ l ⊆ lst ∧ ∀ x · (x ∈ dom(l) ⇒ x..l(x) ⊆ dom(m)) ∧
  ∀ x · (x ∈ dom(m) ⇒ ∃ y · (y ∈ dom(l) ∧ x ∈ y..l(y)))
  THEN mem := m || lst := l END ;
Wadr(i, v) ≐ PRE i ∈ IDP ∧ v ∈ ℕ THEN adr(i) := v END ;
iWadr ≐ BEGIN adr := IDP → ℕ END ;
out ← Radr(i) ≐ PRE i ∈ IDP THEN out := adr(i) END ;
Wmem(a, v) ≐ PRE a ∈ dom(mem) ∧ v ∈ TYPE THEN mem(a) := v END ;
iWmem ≐ BEGIN mem ∈ dom(mem) → TYPE END ;
out ← Rmem(a) ≐ PRE a ∈ dom(mem) THEN out := mem(a) END ;
scramble ≐ ANY m, l, a WHERE m ∈ ℕ1 ↔ TYPE ∧ l ⊆ ℕ1 < geq ∧ a ∈ IDP → ℕ ∧
  ∀ x · (x ∈ dom(l) ⇒ x..l(x) ∩ dom(l) = {x}) ∧
  ∀ x · (x ∈ dom(l) ⇒ x..l(x) ⊆ dom(m)) ∧
  ∀ x · (x ∈ dom(m) ⇒ ∃ y · (y ∈ dom(l) ∧ x ∈ y..l(y)))
  THEN mem := m || lst := l || adr := a END
END

```

A.5 Pointeurs sur pointeur

```

MACHINE ptrptrs(TYPE, IDP, IDPP)
EXTENDS ptrtps(TYPE, IDP)
VARIABLES pmem, plst, padr
INVARIANT pmem ∈ ℕ1 ↔ ℕ ∧ plst ∈ ℕ1 ↔ ℕ1 ∧ padr ∈ IDPP → ℕ ∧
  plst ⊆ geq ∧
  ∀ x · (x ∈ dom(plst) ⇒ x..plst(x) ∩ dom(plst) = {x}) ∧
  ∀ x · (x ∈ dom(plst) ⇒ x..plst(x) ⊆ dom(pmem)) ∧
  ∀ x · (x ∈ dom(pmem) ⇒ ∃ y · (y ∈ dom(plst) ∧ x ∈ y..plst(y)))
INITIALISATION
  ANY m, l, a WHERE m ∈ ℕ1 ↔ ℕ ∧ l ⊆ ℕ1 < geq ∧ a ∈ IDPP → ℕ ∧
    ∀ x · (x ∈ dom(l) ⇒ x..l(x) ∩ dom(l) = {x}) ∧
    ∀ x · (x ∈ dom(l) ⇒ x..l(x) ⊆ dom(m)) ∧
    ∀ x · (x ∈ dom(m) ⇒ ∃ y · (y ∈ dom(l) ∧ x ∈ y..l(y)))
  THEN pmem := m || plst := l || padr := a END
OPERATIONS
  out ← difpdr(i, j) ≐
    PRE i ∈ IDPP ∧ j ∈ IDPP ∧ ∃ x · (x ∈ dom(plst) ∧ padr[{i, j}] ⊆ x..plst(x))
    THEN out := padr(i) - padr(j) END ;
  out ← gtrpdr(i, j) ≐
    PRE i ∈ IDPP ∧ j ∈ IDPP ∧ ∃ x · (x ∈ dom(plst) ∧ padr[{i, j}] ⊆ x..plst(x))
    THEN out := bool(padr(i) > padr(j)) END ;
  out ← pmalloc(s) ≐ PRE s ∈ ℕ1 THEN
    ANY a, v WHERE a ∈ ℕ1 ∧
      ∀ x · (x ∈ dom(plst) ⇒ a..a + s - 1 ∩ x..plst(x) = ∅) ∧
      v ∈ a..(a + s - 1) → ℕ THEN
      out := a || plst(a) := a + s - 1 || pmem := pmem <+ v
    END
  END ;
  ipmalloc ≐ ANY m, l WHERE pmem ⊆ m ∧ plst ⊆ l ∧ l ⊆ geq ∧
    ∀ x · (x ∈ dom(l) ⇒ x..l(x) ∩ dom(l) = {x}) ∧
    ∀ x · (x ∈ dom(l) ⇒ x..l(x) ⊆ dom(m)) ∧
    ∀ x · (x ∈ dom(m) ⇒ ∃ y · (y ∈ dom(l) ∧ x ∈ y..l(y)))
    THEN pmem := m || plst := l END ;
  pfree(a) ≐
    PRE a ∈ dom(plst) THEN pmem := a..plst(a) < pmem || plst := {a} < plst END ;
  ipfree ≐ ANY m, l WHERE m ⊆ pmem ∧ l ⊆ plst ∧ ∀ a · (a ∈ dom(l) ⇒ a..l(a) ⊆ dom(m)) ∧
    ∀ a · (a ∈ dom(m) ⇒ ∃ b · (b ∈ dom(l) ∧ a ∈ b..l(b)))
    THEN pmem := m || plst := l END ;
  Wpdr(i, v) ≐ PRE i ∈ IDPP ∧ v ∈ ℕ THEN padr(i) := v END ;
  iWpdr ≐ BEGIN padr := IDPP → ℕ END ;
  out ← Rpdr(i) ≐ PRE i ∈ IDPP THEN out := padr(i) END ;
  Wpmem(a, v) ≐ PRE a ∈ dom(pmem) ∧ v ∈ ℕ THEN pmem(a) := v END ;
  iWpmem ≐ BEGIN pmem := dom(pmem) → ℕ END ;
  out ← Rpmem(a) ≐ PRE a ∈ dom(pmem) THEN out := pmem(a) END ;
  pscramble ≐ ANY m, l, a WHERE m ∈ ℕ1 ↔ ℕ ∧ l ⊆ ℕ1 < geq ∧ a ∈ IDPP → ℕ ∧
    ∀ x · (x ∈ dom(l) ⇒ x..l(x) ∩ dom(l) = {x}) ∧
    ∀ x · (x ∈ dom(l) ⇒ x..l(x) ⊆ dom(m)) ∧
    ∀ x · (x ∈ dom(m) ⇒ ∃ y · (y ∈ dom(l) ∧ x ∈ y..l(y)))
    THEN pmem := m || plst := l || padr := a END
END

```