Pour que l'exception devienne la règle

Étude théorique et pratique d'un langage intermédiaire à base de continuations

Paul Patault
École normale supérieure Paris-Saclay

encadré par
Jean-Christophe Filliâtre
Andrei Paskevich

au
Laboratoire Méthodes Formelles

Le contexte général

Les méthodes formelles couvrent un large éventail de techniques visant à prouver diverses propriétés sur les programmes. Au delà des propriétés de sûreté d'exécution, la vérification déductive (ou preuve de programmes) permet d'établir la correction fonctionnelle complète. Cela signifie que, quelle que soit l'entrée, le programme se comporte comme prévu. Pour parvenir à prouver une telle propriété, nous devons pouvoir énoncer formellement le comportement attendu d'un programme : il s'agit de la *spécification*. La vérification déductive requiert l'extension du langage initial avec des primitives de spécification. Une possibilité consiste ensuite à développer un logiciel spécifiquement dédié à la preuve de programmes écrits dans ce langage. Alternativement, nous pouvons le traduire le langage augmenté vers un langage intermédiaire conçu de sorte à faciliter la preuve de programmes. Cette seconde solution est majoritairement préférée à la première, étant donné la conception spécifique du langage cible pour la vérification. Cela permet également de mutualiser l'implémentation du moteur de génération de conditions de vérification pour plusieurs langages sources.

Dans ce but, divers langages dédiés à la preuve de programmes ont déjà été proposés. En particulier, le logiciel Why3 est un outil de vérification déductive développé au Laboratoire Méthodes Formelles dans l'équipe Preuve de Programmes. Il propose un langage de programmation et de spécification, WhyML, et un générateur de conditions de vérification faisant appel à des démonstrateurs automatiques et interactifs. WhyML est utilisé comme cible dans différents outils de plus haut niveau pour la preuve de programmes Ada, C, Java ou Rust. Comprenant de multiples constructions syntaxiques, c'est également un langage élaboré pour être utilisé directement. Dans mon stage, je cherche à comprendre la bonne manière de traiter la génération des obligations de preuve dans un cadre générique, au travers d'un nouveau langage intermédiaire spécifiquement conçu pour cette tâche.

Le problème étudié

Le sujet de mon stage est l'étude de Coma, un langage de programmation avec contrats développé par Andrei Paskevich, enseignant-chercheur au Laboratoire Méthodes Formelles. Spécifiquement dédié à la vérification déductive, il est destiné à être un langage intermédiaire dans un système de preuve de programmes tel que Why3. Mon travail est d'étudier les qualités théoriques et pratiques de ce nouveau langage. Ainsi, dans une première partie, j'expérimente les capacités de Coma sous un angle « pratique », en tant que langage cible pour un langage de programmation impératif traditionnel. Ensuite, dans une seconde partie, j'étudie le système de types de Coma. Je propose une preuve des propriétés méta-théoriques de progrès et de préservation. Cette dernière est intéressante car le système de types de Coma garantit l'absence d'alias entre les références (variables mutables).

La contribution proposée

Les contributions proposées à l'issue de ce stage sont basées sur le rapport technique introduisant le langage Coma [16]. Elles peuvent être séparées en deux parties disjointes.

Dans un premier temps, une étude pratique du langage Coma en tant que langage intermédiaire est réalisée. Après l'introduction du langage impératif minimal While (section 2), je présente une sémantique formelle pour ce langage (sous-section 2.1). Ensuite, je propose une procédure de traduction de While vers Coma (sous-section 2.2). Le développement de la sémantique formelle pour While me permet de réaliser une preuve de correction de la traduction proposée (théorème 1), en utilisant un argument de préservation de sémantique (sous-section 2.3).

Dans un second temps, mon travail est centré sur l'étude de la méta-théorie du langage Coma (section 3). Les principaux résultats sont deux théorèmes. Il s'agit des propriétés de préservation du typage (théorème 3) et de progrès de calcul (théorème 2).

Les arguments en faveur de sa validité

Chaque contribution est accompagnée d'une preuve détaillée dans ce rapport. Ces preuves s'appuient sur des sémantiques formelles, dont une est définie dans ce rapport et l'autre provient du document décrivant Coma.

Le bilan et les perspectives

Le travail réalisé au cours de ce stage a permis de déceler différents problèmes dans la conception initiale du langage Coma. Les résultats méta-théoriques attendus ont imposé la modification de plusieurs définitions dans la sémantique et le système de types de Coma. Ayant maintenant plus de confiance en ce système, nous pouvons envisager d'avancer sur différents aspects.

Ce travail va être poursuivi au cours de ma thèse, sous la direction de Jean-Christophe Filliâtre et Andrei Paskevich. De nombreuses pistes sont envisagées pour l'étendre. De la même façon que dans ce rapport, nous pouvons les séparer en deux parties, théorique et pratique.

Pour la partie théorique, plusieurs questions restent ouvertes. La correction de la génération des conditions de vérification doit être prouvée. Il s'agit d'une propriété de préservation : si un programme est prouvé correct, alors il doit rester vérifiable après un pas de calcul. Cela garantit que le programme va satisfaire toutes les assertions au cours de son exécution, ce qui signifie qu'il est correct. Cette preuve requiert en amont la vérification du système de calcul des effets. C'est aussi une propriété de préservation : si les effets calculés sont corrects pour un programme Coma, alors ce même calcul sur le programme réduit doit être correct. La préservation de la sémantique après traduction vers un programme pur doit aussi être assurée. La transformation monadique, en éliminant l'état mutable, ne doit pas changer le comportement opérationnel du programme Coma. Par ailleurs, nous chercherons à démontrer la correction de la traduction du code fantôme dans les programmes Coma, par une démonstration de bi-simulation. Enfin, nous étudions en ce moment comment étendre le langage à l'ordre supérieur.

D'un autre côté, nous prévoyons d'implémenter la langage Coma ainsi que sa procédure de calcul de conditions de vérification. Cela permettrait de tester en pratique les capacités de ce langage sur des exemples autrement consistants, pour lesquels un calcul à la main n'est pas envisageable. De plus, l'implémentation de ce langage mène à son incorporation dans la plateforme de vérification déductive Why3. Cela offre de nombreuses possibilités pour réaliser des études de cas à une échelle plus importante.

Tous ces chantiers constituent un plan de travail pour la thèse que je vais mener sur ce sujet.

Il est de la règle de vouloir
la mort de l'exception.

Jean-Luc Godard
(Je vous salue, Sarajevo)

1 Contexte de travail

Dans cette section, nous introduisons le contexte dans lequel se place le travail réalisé au cours de ce stage. En particulier, nous présentons le langage Coma [16], développé au Laboratoire Méthodes Formelles par Andrei Paskevich, l'un de mes superviseurs.

1.1 Vérification déductive

La vérification déductive de programmes est la discipline de la preuve formelle mécanisée de la correspondance d'un programme à sa spécification. Le caractère déductif précise que cette méthode est différente d'autres techniques de vérification comme l'interprétation abstraite, le *model checking*, ou les tests. Plus spécifiquement, on s'intéresse ici à la vérification déductive de programmes impératifs.

Une logique pour les programmes. Afin de parvenir à la preuve d'un programme, nous avons besoin d'un langage pour exprimer formellement des propriétés à son sujet ; nous parlons de *spécification*. L'idée de décrire formellement le comportement d'un programme afin de prouver sa correction est déjà dans l'esprit de Turing en 1949 [19, 14], mais met un certain temps à s'imposer. Une première méthode, encore informelle, est proposée par Naur (1966) [15]. Celle-ci est affinée par Floyd (1967) [7], puis par Hoare (1969) [9] qui fondent ce que nous appelons aujourd'hui *la logique de Floyd-Hoare*. Vient avec cette logique la notation en triplet $\{P\}\ c\ \{Q\}$, correspondante à la phrase « si la propriété P est vraie, alors après l'exécution du programme c la propriété Q est vraie ». Un exemple d'axiome de cette logique est $\{P[x \mapsto v]\}\ x \leftarrow v\ \{P\}$, où x est une variable, v une valeur et $P[x \mapsto v]$ correspond à la formule P dans laquelle toute occurrence libre de x est remplacée par v. Cet axiome déclare que ce qui est vrai d'une valeur avant son affectation, est vrai après son affectation sous son nouveau nom. Quelques années plus tard, Dijkstra (1975) [4] propose un algorithme pour calculer une formule P à partir de c et Q, telle que $\vdash \{P\}\ c\ \{Q\}$. Intitulé *weakest precondiction* (WP), il calcule la plus faible pré-condition requise pour prouver la post-condition souhaitée du programme.

Explosion combinatoire. Le défaut principal de l'algorithme WP de Dijkstra est la croissance exponentielle de la taille de la formule calculée. Celle-ci peut par exemple apparaître avec une succession de branchements conditionnels (voir le programme correspondant en annexe A1). L'idée de factoriser les WP arrive alors plus récemment, avec un calcul proposé par Flanagan et Saxe (2001) [6, 11]. Dans l'outil de preuve de programmes Why3 [1, 2, 5], ce calcul est implanté en plus du calcul traditionnel.

1.2 Le langage Coma

Généalogie. Si l'on examine le calcul de plus faible pré-condition traditionnel, il apparaît un parallèle entre les règles régissant les sorties [1] standard et exceptionnelle. En effet, la symétrie entre les calculs pour les constructions `skip`/`raise` et `sequence`/`try_with` est flagrante (voir l'annexe A2 pour s'en convaincre). Dans un contexte de preuve de programmes, la distinction entre une sortie de bloc *à la* `return` par opposition à `raise`, est insignifiante. Nous pouvons donc mutualiser ces cas, faisant abstraction du mode de sortie. L'exception ne confirme plus la règle, l'exception *est* la règle.

1. Nous parlons ici de la sortie du programme au sens du flot de contrôle et non pas de la valeur de retour.

Le langage. Le langage Coma (abréviation pour *Continuation Machine*) est un langage adapté à la vérification déductive. Se présentant sous la forme *continuation-passing-style* (CPS) [18, 8], il convient comme représentation intermédiaire. Dans cette perspective de langage intermédiaire, la syntaxe de Coma vise une forme de minimalité ; celle-ci n'inclut que la définition et l'application de fonction, l'allocation de références, et quelques outils de spécification. Les composantes de base de Coma sont les *expressions*. Ce sont les constructions syntaxiques qui font des calculs et produisent des effets, par opposition aux *termes* qui correspondent aux données pures du langage. Une expression peut être encapsulée dans un *handler* (nom donné aux sous-routines du langage Coma) ou dans une fermeture anonyme. Le langage Coma est fortement typé. Une expression qui attend des arguments a la signature π (où π est la liste des types des arguments attendus) ; une expression pleinement appliquée est de type $\square$. Prenons la fonction d'insertion d'un élément dans une liste triée comme premier exemple pour illustrer la syntaxe de Coma. À des fins de simplicité, on suppose connaître les prédicats logiques `sorted` et `permut`. D'arité respective un et deux, le premier est satisfait avec une liste triée d'entiers, et le second si et seulement si les deux listes passées sont des permutations l'une de l'autre.

```
1 insert (x: int) (l: list int) (return (_: list int))
2 = { sorted l }
3   ↑ unList int l
4             ((h: int) (t: list int) →
5                 if (x < h)
6                    (→ break (cons x l))
7                    (→ insert x t ((r: list int) → break (cons h r))))
8             (→ break (cons x nil))
9   / break (r: list int) = { sorted r ∧ permut r (cons x l) } ↑ return r
```

La ligne 1 de ce code déclare un handler nommé `insert`. Ce handler est récursif et attend en entrée un entier `x`, une liste d'entiers `l` et une continuation `return` à laquelle on passe la liste résultante. L'implémentation de ce handler commence avec l'assertion « la liste `l` est triée ». Cette assertion est notée entre accolades et garde l'expression définie sur les lignes 3-8. Cette expression est placée sous une barrière (opérateur `↑`) qui n'affecte pas l'exécution mais uniquement le calcul des conditions de vérification. La barrière délimite la partie de l'expression qui peut n'être vérifiée qu'une seule fois pour toutes valeurs de paramètres. À l'inverse, l'assertion en ligne 2 et la définition locale du handler `break` en ligne 9 ne sont pas cachées sous la barrière et doivent donc être vérifiées à chaque appel de `insert`. Le symbole *slash* en ligne 9 sépare l'expression principale de la définition de `break` et peut être interprété comme le mot « où ». Nous pouvons remarquer que la vérification du corps du handler local `break` consiste à prouver que la continuation de `insert` est correcte. L'assertion `sorted r ∧ permut r (cons x l)` correspond donc à la post-condition de `insert`, tandis que la première sur la ligne 2 est la pré-condition de ce handler.

Revenons à la partie du code écrite sous la barrière. Nous utilisons la primitive `unList` de Coma, déstructurant la liste passée en paramètre et passant le résultat à l'une des deux continuations ; son comportement est proche de la construction `match-with` du langage OCaml. Soit la liste est vide, dans ce cas (ligne 8) la continuation est simplement de terminer l'insertion en appliquant la continuation de sortie à la liste contenant uniquement l'élément `x`. Soit la liste n'est pas vide, nous avons donc sur les lignes 4-7 la définition d'une continuation attendant `h` et `t`, la tête et la queue de la liste déstructurée. Deux sous-cas apparaissent : si `x < h`, alors il faut ajouter l'élément à insérer en tête de liste et continuer avec ce résultat (ligne 6) ; sinon, on insère récursivement `x` dans la sous-liste `t` et la continuation n'a plus qu'à remettre la tête `h` sur la liste résultante de cet appel récursif.

Mutabilité. Dans ce second exemple, nous présentons la mutabilité de Coma au travers d'une fonction de post-incrément de référence.

```
1 postIncr (&r: int) (return [r] (p: int))
2 = ((v: int) →
3       ↑ assign int &r (r+1) break
4       / break [r] = { r = v+1 } ↑ return v) r
```

Ce programme définit un handler nommé `postIncr` prenant en paramètres une référence `r` et une continuation `return`. Cette continuation a un pré-effet sur la référence `r` que l'on note `[r]`. Cela signifie qu'avant que cette continuation soit appelée la valeur de la référence `r` peut avoir changé. De plus, elle attend un paramètre entier `p` : la valeur de la référence avant son incrément. Le langage Coma n'ayant pas de `let` immutable, on simule cette construction avec un β-*redex*. La valeur initiale de `r` est donc stockée dans `v`. Ensuite, nous définissons localement le handler `break`, continuation finale passant `v` à `return`. Finalement, l'expression principale sous une barrière incrémente la référence `r` et passe le contrôle à la continuation `break`. Enfin, remarquons que, comme pour le handler `insert`, la barrière cache l'expression commençant par `assign` (ligne 3) et que l'assertion `r = v+1` est la post-condition de ce handler.

Système de types. Le système de types du langage Coma est relativement traditionnel. Il est proche du système Hindley-Milner avec applications de types explicites. Une différence majeure subsiste avec le système de types d'un langage ML habituel [3]. Il s'agit de l'assurance de l'absence d'alias entre les références. Cette propriété est encodée dans le système par la règle de typage de l'application à une référence, nommée T-AppR.

$$\frac{\Gamma, \Delta' \vdash e : (\&r : \tau)\,\pi \qquad \Delta' \text{ is } \Delta \text{ with all handler prototypes removed}}{\Gamma, \&r : \tau, \Delta \vdash e\ \&r : \pi}\ \text{(T-AppR)}$$

Dans cette règle de typage, la lettre π correspond au type de l'application de l'expression e à la référence $\&r$, tandis que l'expression de type $(\&r : \tau)\,\pi$ est le type de l'expression e. Cela signifie que cette expression, une fois appliquée à une référence de type τ (ici, r est un liant pour la suite du type π), est de type π.

Nous pouvons y remarquer deux choses : (i) la référence est retirée du contexte de typage ; (ii) le système retire également tous les handlers introduits après la référence. Ces deux conditions suffisent à exprimer l'absence d'alias. Lorsqu'une référence est passée en paramètre à un handler, celui-ci ne peut plus y accéder sous son ancien nom (par (i)) et tout autre handler pouvant mentionner cette référence dans sa définition (tout handler défini dans la portée lexicale de la référence) n'est plus accessible (par (ii)). Pour plus de détails, le lecteur peut se référer à la définition complète du système de types de Coma en annexe B2.

Calcul des conditions de vérification. La spécification d'un programme Coma est écrite dans le code. Nous utilisons la notation $\{\varphi\}\ e$ pour *garder* l'expression e sous l'assertion φ. La sémantique de cette construction est bloquante : l'exécution échoue si l'assertion φ ne tient pas. Les conditions de vérification (VC) sont les propriétés qu'une expression doit satisfaire pour assurer sa correction fonctionnelle. Les VC sont calculées avec l'opérateur $\complement^{\mathfrak{p}}_{\mathfrak{d}}$. Les deux booléens $\mathfrak{p}$ et $\mathfrak{d}$ sont des paramètres définissant le *mode* de calcul des VC. Il y quatre *modes* différents :

- $\complement^{\bot}_{\top}$ est le *mode appelé*, il détermine la correction d'une définition de handler ;
- $\complement^{\top}_{\bot}$ est le *mode appelant*, il détermine la correction d'un appel à un handler ;
- $\complement^{\top}_{\top}$ est le *mode total*, c'est la conjonction des deux modes précédents ;
- $\complement^{\bot}_{\bot}$ est le *mode nul*, produisant $\top$ sur toute expression pleinement appliquée.

La correction totale d'une expression e est donc assurée par la preuve de la formule obtenue par le calcul $\complement^{\top}_{\top}(e)$. Cet opérateur est défini par cas sur les expressions de Coma et le détail peut être trouvé en annexe B4.

Précisons également les actions des barrières dans ce calcul de VC. Celles-ci sont *fermantes* ou *ouvrantes*, respectivement notées avec les opérateurs ↑ et ↓. Elles n'ont aucun effet opérationnel, mais conditionnent le *mode* de calcul de VC de l'expression gardée. Ainsi, la barrière fermante ↑ (resp.

ouvrante ↓) a pour effet de « remonter » (resp. « descendre ») le paramètre $\mathfrak{d}$ (resp. $\mathfrak{p}$). Formellement, nous avons les définitions suivantes :

$$\mathsf{C}^{\mathfrak{p}}_{\mathfrak{d}}(\uparrow e) \triangleq \mathsf{C}^{\mathfrak{d}}_{\mathfrak{d}}(e) \qquad\qquad \mathsf{C}^{\mathfrak{p}}_{\mathfrak{d}}(\downarrow e) \triangleq \mathsf{C}^{\mathfrak{p}}_{\mathfrak{p}}(e)$$

L'introduction de barrières offre à l'utilisateur la possibilité de forcer le passage d'un mode à un autre. Ainsi, la barrière fermante force le passage du *mode appelant* au *mode nul* : le corps d'un handler n'est pas vérifié à chaque appel, il suffit de prouver la pré-condition ; et du *mode appelé* au *mode total* : on ne vérifie qu'une seule fois la définition pour toutes valeurs de paramètres. La barrière ouvrante est l'opération duale.

2 Le langage While

Le langage Coma étant défini comme une cible, traitons-le comme tel. Nous choisissons ici un langage déterministe, impératif, à état mutable, que nous appelons While. La base impérative du langage comporte différentes instructions : la déclaration locale de variable, l'assignation, l'instruction `skip` qui ne fait rien, la séquence qui enchaîne deux instructions et deux structures de contrôle simples, la conditionnelle `if-then-else` et la boucle `while`. Nous enrichissons le contrôle avec les instructions traditionnelles `break` et `continue`. De plus, nous ajoutons à cette base une instruction `assert` qui vérifie la validité d'une formule à l'exécution. De la même façon, nous offrons la possibilité d'attacher un invariant à une boucle `while`. Nous complétons également le langage d'une instruction particulière `halt` qui arrête le programme. Enfin, nous étendons While avec une construction spéciale de déstructuration de liste permettant de nommer sa tête et sa queue, seulement si elle n'est pas vide (sémantique bloquante).

Un programme While est une séquence d'instructions. Celle-ci peut contenir l'instruction `halt`, n'importe où dans le code, ce qui arrête l'exécution du programme. Elle peut se trouver aussi bien en dernière position dans le programme (dans ce cas, elle est inutile), qu'au milieu d'une boucle à la manière d'un `return`. La syntaxe abstraite de While est formellement définie sous la forme d'une grammaire BNF en figure 1[2].

Syntaxe de While. Nous avons choisi de spécifier le code à l'intérieur de lui-même. En effet, puisque les programmes While ne produisent pas de résultats et que les variables ont une portée limitée, aucun énoncé intéressant ne peut être formulé en dehors du code. La spécification d'un programme While est donc faite avec l'utilisation des assertions et des invariants de boucle. Ces formules sont partie intégrante du programme et peuvent faire référence aux variables définies dans leur contexte lexical. Enfin, notons que nous utilisons dans ce document les notations $\bot$ pour `false` et $\top$ pour `true` dans les formules logiques.

Exemple. Nous pouvons écrire un programme While calculant le coefficient binomial C_n^k (figure 2). Dans le code présenté, les variables n et k sont libres. Comme nous ne nous intéressons pas pour l'instant à la preuve de ce programme, nous omettons les invariants des deux boucles `while`.

2.1 Sémantique de While

Meyer (1986) [12] montre que la logique de Floyd-Hoare [7, 9], qui associe un axiome ou une règle d'inférence à chaque construction syntaxique d'un langage, définit une sémantique pour ce langage. Chaque triplet de Hoare établit ce que nous tenons pour vrai après l'exécution d'une construction en terme de ce qui était vrai avant. Remarquons qu'il est possible de présenter cette sémantique dans « l'autre sens » : sous la forme d'un calcul de plus faible pré-condition. Nous proposons une sémantique axiomatique pour le langage While sous cette seconde forme (voir figure 3). Le calcul de WP pour While est défini récursivement sur la forme syntaxique du programme. Il requiert,

2. Les formules logiques, dont le symbole non terminal est noté *formula*, sont définies en annexe A3.

```
   term ::= variable
          | true | false | ... | -1 | 0 | 1 | 2 | ...
          | term + term | term - term | term * term
          | term = term | term < term | term > term
          | cons term term | nil | ...

instruction ::= halt
          | skip
          | break | continue
          | assert formula
          | let variable = term
          | let variable , variable = term
          | variable := term
          | if term then instruction else instruction
          | while term invariant formula do instruction done
          | instruction ; instruction
```

FIGURE 1 – Définition du langage WHILE.

```
1  if n = 0 then assert { C(n,k) = 1 }; halt else skip;
2  let ni = 1;
3  let line_cur = cons 1 nil;
4  while true do
5      let ki = 0;
6      let h = 0;
7      let line_new = nil;
8      while line_cur <> nil do
9          let x, y = line_cur;
10         if ni = n and ki = k then
11             assert { C(n,k) = h + x };
12             halt
13         else skip;
14         line_new := cons (h + x) line_new;
15         h := x;
16         line_cur := y;
17         ki := ki + 1
18     done;
19     line_cur := cons 1 line_new;
20     ni := ni + 1
21 done
```

FIGURE 2 – Calcul de coefficient binomial en WHILE.

$$
\begin{array}{rcl}
\mathrm{WP}(\texttt{skip}, K, B, C) & \triangleq & K \\
\mathrm{WP}(\texttt{break}, K, B, C) & \triangleq & B \\
\mathrm{WP}(\texttt{continue}, K, B, C) & \triangleq & C \\
\mathrm{WP}(\texttt{halt}, K, B, C) & \triangleq & \top \\
\mathrm{WP}(\texttt{assert}\ \varphi, K, B, C) & \triangleq & \varphi \wedge (\varphi \to K) \\
\mathrm{WP}(x\ \texttt{:=}\ e, K, B, C) & \triangleq & K[x \mapsto e] \\
\mathrm{WP}(\texttt{let}\ x\ \texttt{=}\ e, K, B, C) & \triangleq & K[x \mapsto e] \\
\mathrm{WP}(\texttt{let}\ x, y\ \texttt{=}\ e, K, B, C) & \triangleq & e \neq \texttt{nil} \wedge \\
 & & \forall x : \tau_x \forall y : \texttt{list}\ \tau_x (e = \texttt{cons}\, x\ y \to K) \\
\mathrm{WP}(i_1\ \texttt{;}\ i_2, K, B, C) & \triangleq & \mathrm{WP}(i_1, \mathrm{WP}(i_2, K, B, C), B, C) \\
\mathrm{WP}(\texttt{if}\ c\ \texttt{then}\ i_1\ \texttt{else}\ i_2, K, B, C) & \triangleq & \texttt{if}\ c\ \texttt{then}\ \mathrm{WP}(i_1, K, B, C) \\
 & & \quad \texttt{else}\ \mathrm{WP}(i_2, K, B, C) \\
\mathrm{WP}(\texttt{while}\ c\ \texttt{invariant}\ \varphi\ \texttt{do}\ i\ \texttt{done}, K, B, C) & \triangleq & \varphi \wedge \forall \overline{q : \tau_q}\, (\varphi \to \\
 & & \quad \texttt{if}\ c\ \texttt{then}\ \mathrm{WP}(i, \varphi, K, \varphi)\ \texttt{else}\ K)
\end{array}
$$

FIGURE 3 – Sémantique axiomatique de WHILE.

en plus d'un programme, trois arguments : les post-conditions pour la sortie standard (notée K), en cas de `break` (notée B) ou de `continue` (notée C). L'initialisation de l'appel récursif est fait avec $\mathrm{WP}(\mathit{prog}, \top, \bot, \bot)$ où *prog* est le programme que l'on souhaite vérifier. Nous choisissons la formule $\top$ comme post-condition normale : le programme composé uniquement de l'instruction `halt` doit être prouvable. En revanche, les post-conditions exceptionnelles sont initialisées avec la formule $\bot$: ainsi tout programme atteignant un `break` ou un `continue` en dehors d'une boucle n'est pas prouvable. Les constructions impératives de base sont traitées de manière traditionnelle [10, 4]. Remarquons la présence de « $\forall \overline{q : \tau_q}$ » dans la règle de `while` : il s'agit des variables libres modifiées dans le corps de la boucle. Nous pouvons également noter que les règles de déclaration de variable et d'affectation sont identiques : en l'état, le langage WHILE est trop simple pour qu'une différence entre ces règles apparaisse. De plus, les règles relatives aux constructions `skip`, `break` et `continue` sont presque identiques. En effet, de l'égalité de traitement entre la sortie normale de bloc et les sorties exceptionnelles, ces trois instructions effectuent une opération de sortie de bloc similaire. La dernière règle particulière est celle du `let` déstructurant : il faut d'une part vérifier explicitement que la liste n'est pas vide et d'autre part lier les variables libres x et y dans la formule K. Enfin, nous admettons savoir typer les programme WHILE ; nous notons τ_x le type de la variable x.

2.2 Compilation vers COMA

La compilation du langage WHILE vers le langage COMA est relativement intuitive. Nous notons avec $[\![\ e \mid k, b, c\ ]\!]$ la compilation de l'expression e vers le langage COMA. Cette traduction est paramétrée par trois expressions COMA k, b et c, respectivement pour *continuation* normale, *break* et *continue*. La première est la continuation de l'expression que l'on traduit. Les deux suivantes sont des points mémorisés au passage de la boucle `while` pour les instructions `break` et `continue`. Ces paramètres sont initialisés avec `halt` pour k et `absurd` pour b et c. On utilise donc `halt` comme continuation finale, et `absurd` comme expression par défaut s'il y a un `break` ou `continue` en dehors d'une boucle `while` [3].

La compilation est détaillée en figure 4, mais certains points restent à éclaircir. Les termes de WHILE sont identiques à ceux de COMA ; ils n'ont pas besoin d'être traduits. L'instruction de branchement conditionnel `if-then-else` est traduite en utilisant le handler primitif `if` dans le langage COMA. Notons que nous factorisons la sortie avec le handler nommé `out` qui prend pour

3. Une fois une boucle traversée, l'expression `absurd` disparaît et est remplacée par les continuations des expressions `break` et `continue`.

$$
\begin{aligned}
[\![\ \texttt{skip} \mid k,\ b,\ c\ ]\!] &\triangleq k \\
[\![\ \texttt{break} \mid k,\ b,\ c\ ]\!] &\triangleq b \\
[\![\ \texttt{continue} \mid k,\ b,\ c\ ]\!] &\triangleq c \\
[\![\ \texttt{halt} \mid k,\ b,\ c\ ]\!] &\triangleq \texttt{halt} \\
[\![\ \texttt{assert}\ \varphi \mid k,\ b,\ c\ ]\!] &\triangleq \{\varphi\}\ k \\
[\![\ x\ \texttt{:=}\ e \mid k,\ b,\ c\ ]\!] &\triangleq \texttt{assign}\ \tau_x\ \&x\ e\ (\texttt{→}\ k) \\
[\![\ \texttt{let}\ x\ \texttt{=}\ e \mid k,\ b,\ c\ ]\!] &\triangleq k\ \texttt{/}\ \&x : \tau_x \texttt{=} e \\
[\![\ \texttt{let}\ x, y\ \texttt{=}\ e \mid k,\ b,\ c\ ]\!] &\triangleq \texttt{unList}\ \tau_x\ e \\
&\qquad ((h : \tau_x)\ (t : \texttt{list}\ \tau_x)\ \texttt{→} \\
&\qquad\quad k\ \texttt{/}\ \&x : \tau_x = h\ \texttt{/}\ \&y : \texttt{list}\ \tau_x = t) \\
&\qquad (\texttt{→}\ \texttt{absurd}) \\
[\![\ i_1\ \texttt{;}\ i_2 \mid k,\ b,\ c\ ]\!] &\triangleq [\![\ i_1 \mid [\![\ i_2 \mid k,\ b,\ c\ ]\!],\ b,\ c\ ]\!] \\
[\![\ \texttt{if}\ c\ \texttt{then}\ i_1\ \texttt{else}\ i_2 \mid k,\ b,\ c\ ]\!] &\triangleq \texttt{if}\ c\ (\texttt{→}\ [\![\ i_1 \mid \texttt{out},\ b,\ c\ ]\!]) \\
&\qquad\qquad (\texttt{→}\ [\![\ i_2 \mid \texttt{out},\ b,\ c\ ]\!]) \\
&\qquad \texttt{/ out}\ [\bar{q}] \texttt{=}\downarrow k \\
[\![\ \texttt{while}\ c\ \texttt{invariant}\ \varphi\ \texttt{do}\ i\ \texttt{done} \mid k,\ b,\ c\ ]\!] &\triangleq \texttt{loop} \\
&\qquad \texttt{/ loop}\ [\bar{q}] \\
&\qquad\quad \texttt{=}\{\varphi\}\uparrow \texttt{if}\ c\ (\texttt{→}\ [\![\ i \mid \texttt{loop},\ \texttt{out},\ \texttt{loop}\ ]\!]) \\
&\qquad\qquad\qquad\qquad (\texttt{→}\ \texttt{out}) \\
&\qquad \texttt{/ out}\ [\bar{q}] \texttt{=}\downarrow k
\end{aligned}
$$

FIGURE 4 – Compilation de WHILE en COMA.

définition l'expression k en y ajoutant une barrière ouvrante (opérateur ↓). L'introduction de la barrière permet de forcer le mode dans le calcul de conditions de vérification fait par COMA. Ici, cela permet de passer l'utilisation de `out` dans le corps du `if-then-else` du *mode appelant* au *mode total*. Nous devons également ajouter à ce handler ses pré-effets : ici, les variables modifiées dans les branches `then` et `else` du `if`. La boucle `while` est naturellement simulée avec un appel récursif terminal. Nous ajoutons à chaque entrée dans le handler `loop` une assertion suivie d'une barrière fermante. Cela transforme l'invariant de boucle φ en une pré-condition pour le handler `loop`. Pour les mêmes raisons que le branchement conditionnel, nous ajoutons une barrière ouvrante avant la continuation de sortie. Nous devons également ajouter les annotations de pré-effets des handlers `loop` et `out`. On ajoute une clause pour chaque variable modifiée dans le corps de la boucle `while`.

Remarquons que la traduction d'un programme WHILE « bien formé » (dans lequel les instructions `break` et `continue` se trouvent toujours sous un `while`), ne peut pas contenir de handler `absurd` qui ne soit pas introduit par un `let` déstructurant. En revanche, si le programme WHILE initial est « mal formé », avec un `break` ou un `continue` accessible en dehors d'une boucle `while`, alors le programme COMA résultant ne sera pas prouvable.

Exemple. Revenons à l'exemple du calcul de coefficient binomial C_n^k. Nous pouvons l'écrire sous la forme d'un handler dans le langage COMA. Nous supposons connue la fonction logique C_n^k utilisée dans la spécification. Ce qui nous intéresse ici étant la traduction des structures de contrôle, nous ne précisons pas ici comment prouver ce programme.

Comme dans les deux exemples précédents de programmes COMA, l'assertion (ligne 2) suivie de la barrière fermante (ligne 3) agit comme une pré-condition pour le handler `coefficient_binomial`. De la même façon, la définition locale du handler de sortie `break` introduit la post-condition. Se trouve ensuite sous la barrière principale un test d'égalité entre `k` et `n` avec les deux continuations possibles : appliquer `break` à `1` ou entrer dans la boucle. Deux handlers `loop` et `loop2` récursifs sont définis localement sous la barrière. Chacun simule une des boucles `while` du programme précédent. Le premier handler `loop` initialise des références mutables locales et le second permet d'itérer sur

```
coefficient_binomial (n: int) (k: int) (return (m: int))
= { 0 ≤ k ≤ n }
  ↑ if (k = n) (→ break 1) loop
    / loop [line_cur ni] =
        ↑ loop2
        / loop2 [line_cur line_new ki h] =
            unList (list int) line_cur
                   ((x: int) (y: list int)
                    → assign (list int) &line_new (cons (h + x) line_new)
                      (→ if (ni = n ∧ ki = k) (→ break (h + x))
                          (→ assign int &h x
                          (→ assign (list int) &line_cur y
                          (→ assign int &ki (ki+1) loop2)))))
                   (→ assign (list int) &line_cur (cons 1 line_new)
                      (→ assign int &ni (ni+1) loop))
        / &h: int = 0
        / &ki: int = 0
        / &line_new: list int = nil
    / &line_cur: list int = cons 1 nil
    / &ni: int = 1
  / break (m: int) = { C_n^k = m } ↑ return m
```

FIGURE 5 – Calcul de coefficient binomial en COMA.

la référence `line_cur`, la précédente ligne calculée, en utilisant la primitive `unList`. Si la ligne a encore un élément, on exécute une fois le corps de la boucle `while` et on continue. Sinon, la ligne est vide ; ainsi, comme dans le programme WHILE, on modifie les références `line_cur` et `n` puis on refait un tour de boucle principale.

Nous pouvons aussi calculer la compilation exacte du programme WHILE (figure 2). Le résultat produit (figure A4, en annexe) est très proche de celui écrit à la main dans lequel nous avons expansé quelques définitions de handlers.

2.3 Préservation de la sémantique

Dans cette section, nous proposons une preuve de la correction de la traduction de la compilation du langage WHILE vers le langage COMA par préservation de sémantique. La preuve est faite à partir des sémantiques axiomatiques des langages WHILE (définie en section 2.1) et COMA (définition en annexe B4). Le théorème postule donc l'équivalence des conditions de vérification générées à partir d'un programme WHILE et sa version compilée en COMA.

Théorème 1 (Préservation de la sémantique axiomatique par compilation de WHILE en COMA)**.** Pour tout programme WHILE P, les formules $\mathrm{WP}(P, \top, \bot, \bot)$ et $\complement_\top^\top(\llbracket\ P \mid \texttt{halt}, \texttt{absurd}, \texttt{absurd}\ \rrbracket)$ sont logiquement équivalentes.

Afin de passer l'induction, cet énoncé doit être généralisé en ses entrées initiales pour les calculs de conditions de vérification.

Lemme 2.1 (Théorème 1 généralisé)**.** Pour tout programme WHILE P, et toutes expressions COMA k, b et c, la formule $\mathrm{WP}(P, \complement_\top^\top(k), \complement_\top^\top(b), \complement_\top^\top(c))$ est équivalente à $\complement_\top^\top(\llbracket\ P \mid k,\ b,\ c\ \rrbracket)$.

Corollaire. Le théorème 1 est un cas particulier du lemme 2.1 dans lequel $k = \texttt{halt}$ et $b = c = \texttt{absurd}$. On a bien $\complement_\top^\top(\texttt{halt}) = \top$ et $\complement_\top^\top(\texttt{absurd})$ équivalente à $\bot$. Ainsi, on obtient précisément l'énoncé attendu.

Démonstration du Lemme 2.1. Par induction sur le programme P. Remarque : on note les égalités par définition avec = et les équivalences logiques avec ≡.

— `halt` :

$$\mathsf{C}^\top_\top(\llbracket\ \texttt{halt} \mid k,\ b,\ c\ \rrbracket) = \mathsf{C}^\top_\top(\texttt{halt}) \equiv \top = \mathrm{WP}(\texttt{halt}, \mathsf{C}^\top_\top(k), \mathsf{C}^\top_\top(b), \mathsf{C}^\top_\top(c))$$

— `skip` :

$$\mathsf{C}^\top_\top(\llbracket\ \texttt{skip} \mid k,\ b,\ c\ \rrbracket) = \mathsf{C}^\top_\top(k) = \mathrm{WP}(\texttt{skip}, \mathsf{C}^\top_\top(k), \mathsf{C}^\top_\top(b), \mathsf{C}^\top_\top(c))$$

— `break` :

$$\mathsf{C}^\top_\top(\llbracket\ \texttt{break} \mid k,\ b,\ c\ \rrbracket) = \mathsf{C}^\top_\top(b) = \mathrm{WP}(\texttt{break}, \mathsf{C}^\top_\top(k), \mathsf{C}^\top_\top(b), \mathsf{C}^\top_\top(c))$$

— `continue` :

$$\mathsf{C}^\top_\top(\llbracket\ \texttt{continue} \mid k,\ b,\ c\ \rrbracket) = \mathsf{C}^\top_\top(c) = \mathrm{WP}(\texttt{continue}, \mathsf{C}^\top_\top(k), \mathsf{C}^\top_\top(b), \mathsf{C}^\top_\top(c))$$

— assertion :

$$\begin{aligned}\mathsf{C}^\top_\top(\llbracket\ \texttt{assert}\ \varphi \mid k,\ b,\ c\ \rrbracket) &= \mathsf{C}^\top_\top(\{\varphi\}\, k)\\ &\equiv (\top \to \varphi) \wedge (\varphi \to \mathsf{C}^\top_\top(k))\\ &\equiv \varphi \wedge (\varphi \to \mathsf{C}^\top_\top(k))\\ &= \mathrm{WP}(\texttt{assert}\ \varphi, \mathsf{C}^\top_\top(k), \mathsf{C}^\top_\top(b), \mathsf{C}^\top_\top(c))\end{aligned}$$

— définition :

$$\begin{aligned}\mathsf{C}^\top_\top(\llbracket\ \texttt{let}\ x = e \mid k,\ b,\ c\ \rrbracket) &= \mathsf{C}^\top_\top(k\ /\ \&x : \tau_x = e)\\ &= \mathsf{C}^\top_\top(k)[x \mapsto e]\\ &= \mathrm{WP}(\texttt{let}\ x = e, \mathsf{C}^\top_\top(k), \mathsf{C}^\top_\top(b), \mathsf{C}^\top_\top(c))\end{aligned}$$

— définition déstructurante :

$$\begin{aligned}&\mathsf{C}^\top_\top(\llbracket\ \texttt{let}\ x, y = e \mid k,\ b,\ c\ \rrbracket)\\ &\quad= \mathsf{C}^\top_\top(\texttt{unList}\ \tau_x\ e\ ((h : \tau_x)\ (t : \texttt{list}\ \tau_x) \to k\ /\ \&x = h\ /\ \&y = t)\ (\to \texttt{absurd}))\\ &\quad= ((e = \texttt{nil}) \to \bot) \wedge \forall ht\ \big((\texttt{cons}\ h\ t = e) \to \mathsf{C}^\top_\top(k)[x \mapsto h, t \mapsto y]\big)\\ &\quad\equiv (e \neq \texttt{nil}) \wedge \forall xy\ \big((\texttt{cons}\ x\ y = e) \to \mathsf{C}^\top_\top(k)\big)\\ &\quad= \mathrm{WP}(\texttt{let}\ x, y = e, \mathsf{C}^\top_\top(k), \mathsf{C}^\top_\top(b), \mathsf{C}^\top_\top(c))\end{aligned}$$

— assignation :

$$\begin{aligned}\mathsf{C}^\top_\top(\llbracket\ x\ \texttt{:=}\ e \mid k,\ b,\ c\ \rrbracket) &= \mathsf{C}^\top_\top(\texttt{assign}\ \tau_x\ \&x\ e\ (\to k))\\ &= \texttt{return}\ e\ /\ \texttt{return}\ (x : \texttt{int}) \equiv \mathsf{C}^\top_\top(k)\\ &\equiv \mathsf{C}^\top_\top(k)[x \mapsto e] && \text{expansion de } \texttt{return},\\ &= \mathrm{WP}(x\ \texttt{:=}\ e, \mathsf{C}^\top_\top(k), \mathsf{C}^\top_\top(b), \mathsf{C}^\top_\top(c))\end{aligned}$$

— séquence :

$$\begin{aligned}&\mathsf{C}^\top_\top(\llbracket\ i_1\ ;\ i_2\ \mid k,\ b,\ c\ \rrbracket)\\ &\quad= \mathsf{C}^\top_\top(\llbracket\ i_1 \mid \llbracket\ i_2 \mid k,\ b,\ c\ \rrbracket,\ b,\ c\ \rrbracket)\\ &\quad\equiv \mathrm{WP}(i_1, \mathsf{C}^\top_\top(\llbracket\ i_2 \mid k,\ b,\ c\ \rrbracket), \mathsf{C}^\top_\top(b), \mathsf{C}^\top_\top(c)) && \text{par IH sur } i_1,\\ &\quad\equiv \mathrm{WP}(i_1, \mathrm{WP}(i_2, \mathsf{C}^\top_\top(k), \mathsf{C}^\top_\top(b), \mathsf{C}^\top_\top(c)), \mathsf{C}^\top_\top(b), \mathsf{C}^\top_\top(c)) && \text{par IH sur } i_2,\\ &\quad= \mathrm{WP}(i_1\ ;\ i_2, \mathsf{C}^\top_\top(k), \mathsf{C}^\top_\top(b), \mathsf{C}^\top_\top(c))\end{aligned}$$

— conditionnelle `if-then-else` :

$$
\begin{aligned}
&\complement_\top^\top(\llbracket\ \texttt{if}\ c\ \texttt{then}\ i_1\ \texttt{else}\ i_2 \mid k,\ b,\ c\ \rrbracket)\\
&\quad= \complement_\top^\top(\texttt{if}\ c\ (\to \llbracket\ i_1 \mid \texttt{out},\ b,\ c\ \rrbracket)\\
&\qquad\qquad (\to \llbracket\ i_2 \mid \texttt{out},\ b,\ c\ \rrbracket)\\
&\qquad\quad /\ \texttt{out}\ [\bar{q}] = \downarrow k)\\
&\quad= ((c \to \texttt{then}) \wedge (\neg c \to \texttt{else})\\
&\qquad /\ \texttt{then} \equiv \complement_\top^\top(\llbracket\ i_1 \mid \texttt{out},\ b,\ c\ \rrbracket)\\
&\qquad /\ \texttt{else} \equiv \complement_\top^\top(\llbracket\ i_2 \mid \texttt{out},\ b,\ c\ \rrbracket)) \wedge \complement_\top^\bot(\downarrow k)\\
&\qquad /\ \texttt{out}\ (\overline{q:\tau_q}) \equiv \complement_\bot^\top(\downarrow k)\\
&\quad= ((c \to \texttt{then}) \wedge (\neg c \to \texttt{else}) && \text{calcul des barrières,}\\
&\qquad /\ \texttt{then} \equiv \complement_\top^\top(\llbracket\ i_1 \mid \texttt{out},\ b,\ c\ \rrbracket)\\
&\qquad /\ \texttt{else} \equiv \complement_\top^\top(\llbracket\ i_2 \mid \texttt{out},\ b,\ c\ \rrbracket))\\
&\qquad /\ \texttt{out}\ (\overline{q:\tau_q}) \equiv \complement_\top^\top(k)\\
&\quad\equiv (c \to \complement_\top^\top(\llbracket\ i_1 \mid \texttt{out},\ b,\ c\ \rrbracket)) \wedge && \text{expansion de } \texttt{then} \text{ et } \texttt{else},\\
&\qquad (\neg c \to \complement_\top^\top(\llbracket\ i_2 \mid \texttt{out},\ b,\ c\ \rrbracket))\\
&\qquad /\ \texttt{out}\ (\overline{q:\tau_q}) \equiv \complement_\top^\top(k)\\
&\quad\equiv (c \to \mathrm{WP}(i_1, \complement_\top^\top(\texttt{out}), \complement_\top^\top(b), \complement_\top^\top(c))) \wedge && \text{par IH sur } i_1 \text{ et } i_2,\\
&\qquad (\neg c \to \mathrm{WP}(i_2, \complement_\top^\top(\texttt{out}), \complement_\top^\top(b), \complement_\top^\top(c)))\\
&\qquad /\ \texttt{out}\ (\overline{q:\tau_q}) \equiv \complement_\top^\top(k)\\
&\quad\equiv (c \to \mathrm{WP}(i_1, \complement_\top^\top(k), \complement_\top^\top(b), \complement_\top^\top(c))) \wedge && \text{expansion du prédicat } \texttt{out},\\
&\qquad (\neg c \to \mathrm{WP}(i_2, \complement_\top^\top(k), \complement_\top^\top(b), \complement_\top^\top(c)))\\
&\quad\equiv \texttt{if}\ c\ \texttt{then}\ \mathrm{WP}(i_1, \complement_\top^\top(k), \complement_\top^\top(b), \complement_\top^\top(c)) && \text{transformation en } \texttt{if},\\
&\qquad\quad \texttt{else}\ \mathrm{WP}(i_2, \complement_\top^\top(k), \complement_\top^\top(b), \complement_\top^\top(c))\\
&\quad= \mathrm{WP}(\texttt{if}\ c\ \texttt{then}\ i_1\ \texttt{else}\ i_2, \complement_\top^\top(k), \complement_\top^\top(b), \complement_\top^\top(c))
\end{aligned}
$$

— boucle `while` :

$$
\begin{aligned}
&\complement_\top^\top(\llbracket\ \texttt{while}\ c\ \texttt{invariant}\ \varphi\ \texttt{do}\ i\ \texttt{done} \mid k,\ b,\ c\ \rrbracket)\\
&\quad= \complement_\top^\top(\texttt{loop}\\
&\qquad /\ \texttt{loop}\ [\bar{q}] = \{\varphi\} \uparrow \texttt{if}\ c\ (\to \llbracket\ i \mid \texttt{loop},\ \texttt{out},\ \texttt{loop}\ \rrbracket)\ (\to \texttt{out})\\
&\qquad /\ \texttt{out}\ [\bar{q}] = \downarrow k)\\
&\quad= (\ \complement_\top^\top(\texttt{loop}) \wedge\\
&\qquad \forall\, \overline{q:\tau_q}\ (\varphi \to \complement_\top^\bot(\uparrow \texttt{if}\ c\ (\to \llbracket\ i \mid \texttt{loop},\ \texttt{out},\ \texttt{loop}\ \rrbracket)\ (\to \texttt{out})))\\
&\qquad /\ \texttt{loop}\ (\overline{q:\tau_q}) \equiv\\
&\qquad\quad \varphi \wedge \varphi \to \complement_\bot^\top(\uparrow \texttt{if}\ c\ (\to \llbracket\ i \mid \texttt{loop},\ \texttt{out},\ \texttt{loop}\ \rrbracket)\ (\to \texttt{out})))\\
&\qquad \wedge \complement_\top^\bot(\downarrow k)\\
&\qquad /\ \texttt{out}\ (\overline{q:\tau_q}) \equiv \complement_\bot^\top(\downarrow k)\\
&\quad\equiv \complement_\top^\top(\texttt{loop}) \wedge && \text{calcul des barrières,}\\
&\qquad \forall\, \overline{q:\tau_q}\ (\varphi \to \complement_\top^\top(\texttt{if}\ c\ (\to \llbracket\ i \mid \texttt{loop},\ \texttt{out},\ \texttt{loop}\ \rrbracket)\ (\to \texttt{out})))\\
&\qquad /\ \texttt{loop}\ (\overline{q:\tau_q}) \equiv \varphi\\
&\qquad /\ \texttt{out}\ (\overline{q:\tau_q}) \equiv \complement_\top^\top(k)\\
&\quad= \texttt{loop}\ \bar{q} \wedge && \text{calcul des VC,}\\
&\qquad \forall\, \overline{q:\tau_q}\ (\varphi \to ((c \to \texttt{then}) \wedge (\neg c \to \texttt{else})\\
&\qquad\qquad /\ \texttt{then} \equiv \complement_\top^\top(\llbracket\ i \mid \texttt{loop},\ \texttt{out},\ \texttt{loop}\ \rrbracket)\\
&\qquad\qquad /\ \texttt{else} \equiv \texttt{out}\ \bar{q}))\\
&\qquad /\ \texttt{loop}\ (\overline{q:\tau_q}) \equiv \varphi\\
&\qquad /\ \texttt{out}\ (\overline{q:\tau_q}) \equiv \complement_\top^\top(k)\\
&\quad\equiv \varphi \wedge\\
&\qquad \forall\, \overline{q:\tau_q}\ (\varphi \to ((c \to \texttt{then}) \wedge (\neg c \to \texttt{else}) && \text{par IH sur } i,\\
&\qquad\qquad /\ \texttt{then} \equiv \mathrm{WP}(i, \varphi, \complement_\top^\top(k), \varphi)\\
&\qquad\qquad /\ \texttt{else} \equiv \complement_\top^\top(k)))\\
&\quad\equiv \varphi \wedge \forall\, \overline{q:\tau_q}\ (\varphi \to (\texttt{if}\ c\ \texttt{then}\ \mathrm{WP}(i, \varphi, \complement_\top^\top(k), \varphi)\ \texttt{else}\ \complement_\top^\top(k)) && \text{factorisation en } \texttt{if},\\
&\quad= \mathrm{WP}(\texttt{while}\ c\ \texttt{invariant}\ \varphi\ \texttt{do}\ i\ \texttt{done}, \complement_\top^\top(k), \complement_\top^\top(b), \complement_\top^\top(c)) && \square
\end{aligned}
$$

3 Méta-théorie

Nous avons vu que Coma est un langage formel fortement typé. Nous soutenons qu'il est nécessaire de prouver la sûreté de son système de types. Dans cette section, nous suivons le schéma de preuve de sûreté de typage traditionnel [17], au travers des lemmes de progrès et de préservation.

3.1 Définitions et hypothèses

Soit Γ_{prim}, le contexte de typage initial contenant exactement l'ensemble des handlers primitifs associés à leur prototype. La liste des handlers primitifs est définie en section 2 du document présentant Coma [16]. Nous appelons *close* toute expression admettant une dérivation de typage sous le contexte Γ_{prim}.

La définition de la sémantique opérationnelle à petits pas de Coma est recopiée en annexe B3. La relation de réduction $\longrightarrow$ est définie sur les expressions normalisées, puis étendue aux expressions closes. On note $\longrightarrow^*$ la clôture réflexive et transitive de la relation $\longrightarrow$. Tout pas de calcul a donc la condition implicite que l'expression à réduire est en forme normale ; si elle ne l'est pas, sa normalisation compte comme pas de calcul. La première règle est la substitution d'un handler par sa définition. Nous trouvons ensuite les β-réductions pour les différentes applications : de type (E-AppT), de terme (E-AppV), de référence (E-AppR) et de handler (E-AppH). La règle d'application à une fermeture (E-AppC) introduit une définition locale de handler. La sémantique de l'assertion est bloquante : si l'évaluation de la formule est fausse, alors on ne peut pas faire le pas de calcul. Enfin, les barrières sont traversées et la fermeture « vide » disparaît.

Cette sémantique omet la définition de l'opérateur de normalisation sur termes. On note $[\![s]\!]_\Sigma$ la *forme canonique* du terme s sous le contexte Σ. Afin de mener des preuves de méta-théorie, nous avons besoin des hypothèses suivantes sur cet opérateur.

Hypothèse 1 (Forme canonique des booléens)**.** Tout terme booléen, dont les variables libres sont dans l'environnement de normalisation Σ, se normalise en `true` ou `false`.

Hypothèse 2 (Forme canonique des entiers)**.** Tout terme de type `int`, dont les variables libres sont dans l'environnement de normalisation Σ, se normalise en un entier de type `int`.

Hypothèse 3 (Forme canonique des listes)**.** Soit α une variable de type quelconque. Pour toute liste ℓ de type `list` α telle que $\text{FV}(\ell) \in \Sigma$, soit $[\![\ell]\!]_\Sigma$ = `nil`, soit $[\![\ell]\!]_\Sigma$ = `cons` t_1 t_2, avec t_1 et t_2 deux termes canoniques respectivement de type α et `list` α.

Le lemme suivant nous permet de faire une légère modification à la sémantique opérationnelle proposée. Cette simplification facilite une preuve à venir.

Lemme 3.1 (Application de référence)**.** Pour toute expression e de la forme $((\&p : \tau)\ \pi \to d)\ \&r\ \bar{a}\ //\ \Lambda$, si $\Gamma_{\text{prim}} \vdash e : \square$, alors $e \equiv_\alpha ((\&r : \tau)(\pi \to d)[p \mapsto r])\ \&r\ \bar{a}\ //\ \Lambda$.

Démonstration. C'est une conséquence de l'absence d'alias mémoire, propriété impliquée par la règle de typage T-AppR. Après l'application à $\&r$, la référence est retirée du contexte de typage. Elle ne peut donc pas être libre dans l'expression e. Ainsi, le renommage n'effectue pas de captures illicites. □

Corollaire 3.1 (Modification d'une règle de sémantique)**.** Si l'on travaille avec des expressions bien typées, la règle E-AppR peut se réécrire sans utiliser de substitution, de la manière suivante : $((\&r : \tau)\ \pi \to e)\ \&r\ \bar{a}\ //\ \Lambda \longrightarrow (\pi \to e)\ \bar{a}\ //\ \Lambda$.

3.2 Sûreté du typage

La sûreté du typage est connue comme le fameux slogan de Milner : « Well-typed expressions do not go wrong. » [13]. En français, nous comprenons la sûreté comme étant la garantie que, si un

programme admet un type, alors il ne provoque aucune erreur à l'exécution. Formellement, cette maxime peut se formuler par la combinaison de trois propriétés : le progrès, la préservation de typage (*subject reduction lemma*) et la préservation de la validité des conditions de vérification. Dans ce travail, nous présentons des preuves pour les résultats de progrès et de préservation du typage.

Théorème 2 (Progrès)**.** Une expression close, correcte et de type $\square$ est réductible ou égale à `halt` : si $\Gamma_{\text{prim}} \vdash e : \square$ et la formule $\complement_{\top}^{\top}(e)$ est valide, alors $e = \texttt{halt}$ ou il existe e' tel que $e \longrightarrow e'$.

Théorème 3 (Préservation)**.** La réduction préserve le typage : si $e \longrightarrow e'$ et $\Gamma_{\text{prim}} \vdash e : \square$, alors $\Gamma_{\text{prim}} \vdash e' : \square$.

3.3 Preuve du progrès

Le progrès (théorème 2) indique qu'une expression de type $\square$ dont la condition de vérification est assurée est réductible ou égale à `halt`. Le langage Coma étant assez proche de ML, nous pourrions vouloir nous y ramener afin de pouvoir faire la preuve de manière classique comme proposé par Wright et Felleisen (1992) [20]. Cependant, une différence majeure entre Coma et les langages *à la* ML [3] est la garantie de l'absence d'alias mémoire. Cette propriété est assurée par le système de types de Coma, ce qui complique la traduction directe et l'utilisation de la preuve traditionnelle. La preuve de ce théorème se fait par analyse de cas, en suivant une décomposition générique pour les expressions du langage Coma.

Lemme 3.2 (Décomposition)**.** Pour toute expression e telle que $\Gamma \vdash e : \square$, il existe une décomposition $e = e_0\ \bar{a}\ //\ \Lambda$ où e_0 n'est pas une application et si $\bar{a} = \square$, alors e_0 n'est pas une définition ou allocation.

Démonstration. Par induction sur l'expression e :

- cas barrière fermante, $e = \uparrow e'_0$: instantané, il suffit de prendre $\bar{a} = \square$, $\Lambda = \square$ et $e_0 = e$.
- cas barrière ouvrante, $e = \downarrow e'_0$: identique au cas précédent.
- cas assertion, $e = \{\varphi\}\ e'_0$: identique au cas précédent.
- cas allocation, $e = e'_0\ /\ \&r : \tau = s$: on sait que $\Gamma, \&r : \tau \vdash e'_0 : \square$. Par hypothèse d'induction, on obtient une décomposition $e'_0 = e'_1\ \bar{a}'\ //\ \Lambda'$. Ainsi, nous pouvons conclure avec la décomposition $e = e'_1\ \bar{a}'\ //\ \Lambda'\ /\ \&r : \tau = s$, où $\Lambda = \Lambda'\ /\ \&r : \tau = s$, $e_0 = e'_1$ et $\bar{a} = \bar{a}'$. Les contraintes sont bien respectées.
- cas définition, $e = e'_0\ /\ h\,[\bar{q}] = d$: identique au cas précédent.
- cas application, $e = e'_0\ \bar{a}'$ (où tous les arguments sont poussés dans $\bar{a}'$) : par disjonction de cas sur la taille de la liste d'arguments $\bar{a}'$.
  - cas vide $\bar{a}' = \square$: alors on sait que $\Gamma \vdash e'_0 : \square$. Par hypothèse d'induction, on obtient une décomposition $e'_0 = e'_1\ \bar{a}_1'\ //\ \Lambda'$. Ainsi, nous pouvons conclure avec cette même décomposition $e = e'_1\ \bar{a}_1'\ //\ \Lambda'$ où $\Lambda = \Lambda'$, $e_0 = e'_1$ et $\bar{a} = \bar{a}_1'$. Les contraintes sont bien respectées.
  - cas non vide $\bar{a}' \neq \square$: par typage, on sait que e'_0 n'a pas le type $\square$. L'expression e'_0 est soit un handler, soit une fermeture. Dans les deux cas, ce n'est ni une définition ni une allocation. Avec $e_0 = e'_0$, $\bar{a} = \bar{a}'$ et $\Lambda = \square$, les contraintes sont bien respectées. La décomposition $e = e'_0\ \bar{a}'$ convient.

□

Ajoutons le lemme suivant, nécessaire pour conclure un cas de la preuve du progrès.

Lemme 3.3. Pour toute expression normale de la forme $\{\varphi\}\ e\ //\ \Lambda$, si $\complement_{\top}^{\top}(\{\varphi\}\ e\ //\ \Lambda)$, alors φ est vraie.

Démonstration. Par induction sur la taille du contexte Λ :

- cas vide ($\Lambda = \square$) : trivial par dépliage de la définition du calcul de VC. La conjonction $\varphi \wedge \complement_{\top}^{\top}(e)$ implique bien φ.

— cas non vide, définition ($\Lambda = \Lambda' \mathbin{/} h\,[\bar{q}]\,\pi = d$) : par hypothèse d'induction, on sait que la VC de la sous-expression $\complement_{\top}^{\top}(\{\varphi\}\ e \mathbin{//} \Lambda')$ implique φ. De plus, le calcul des conditions de vérification d'une définition de handler ne construit qu'une conjonction au-dessus du cas récursif $\complement_{\top}^{\top}(\{\varphi\}\ e \mathbin{//} \Lambda')$. L'implication logique est donc préservée.
— cas non vide, allocation ($\Lambda = \Lambda' \mathbin{/} \&r : \tau = s$) : par hypothèse d'induction, on sait que la VC de la sous-expression $\complement_{\top}^{\top}(\{\varphi\}\ e \mathbin{//} \Lambda')$ implique φ. Il reste à montrer que l'extension du contexte avec une allocation n'affaiblit pas la formule. Remarquons que, comme l'expression initiale est en forme normale, la référence r n'apparaît pas dans la formule φ. Nous pouvons donc conclure par le calcul :

$$
\begin{aligned}
& \complement_{\top}^{\top}(\{\varphi\}\ e \mathbin{//} \Lambda' \mathbin{/} \&r : \tau = s) \\
= \ & \complement_{\top}^{\top}(\{\varphi\}\ e \mathbin{//} \Lambda')[r \mapsto s] && \text{par définition de } \complement_{\top}^{\top} \\
\equiv \ & \complement_{\top}^{\top}(\{\varphi\}\ e[r \mapsto s] \mathbin{//} \Lambda'[r \mapsto s]) && r \notin \mathrm{FV}(\varphi) \\
\Rightarrow \ & \varphi && \text{la substitution n'a plus d'effet sur } \varphi
\end{aligned}
$$

Nous avons ainsi l'implication souhaitée.

□

Revenons à la preuve du théorème 2. On traite d'abord le cas d'une expression qui n'est pas en forme normale. Celui-ci peut être évacué : la règle E-Norm s'applique et le progrès est vérifié. Le reste de la preuve est fait par cas sur les différentes formes syntaxiques possibles pour la sous-expression e_0, élément de la décomposition $e = e_0\ \bar{a} \mathbin{//} \Lambda$ proposée dans le lemme 3.2.

— barrière fermante, $e_0 = \uparrow e_1$: le typage nous garantit que $\bar{a} = \square$. La règle E-TagU peut donc s'appliquer et l'expression se réduit en $e_1 \mathbin{//} \Lambda$.
— barrière ouvrante, $e_0 = \downarrow e_1$: identique au cas précédent.
— assertion, $e_0 = \{\varphi\}\, e_1$: le typage nous garantit que $\bar{a} = \square$. Pour pouvoir appliquer la règle E-Assert, il faut prouver la formule φ. Or, par hypothèse nous savons que la condition de vérification $\complement_{\top}^{\top}(e)$ est prouvée. Nous devons alors prouver que la formule φ n'est pas affaiblie en traversant le contexte formé par la condition de vérification, ce qui est assuré par le lemme 3.3. La règle E-Assert peut donc s'appliquer et l'expression se réduit en $e_1 \mathbin{//} \Lambda$.
— application de handler primitif :
  — si $e_0 = \texttt{div}$, alors on sait par le typage que $\bar{a} = s\ t\ d$. De plus, en utilisant le même argument que celui du lemme 3.3 nous pouvons déduire de l'hypothèse de correction que $|t| > 0$. L'expression se réduit donc en $d\ n \mathbin{//} \Lambda$ où n est bien le résultat de la division de s par t.
  — si $e_0 = \texttt{if}$, alors par le typage on sait que $\bar{a} = b\ d_1\ d_2$. L'expression est en forme normale, donc par l'hypothèse 1 on sait que le booléen b est égal à `true` ou `false`. L'expression se réduit soit en $d_1 \mathbin{//} \Lambda$, soit en $d_2 \mathbin{//} \Lambda$.
  — si $e_0 = \texttt{unList}$, alors par le typage on sait que $\bar{a} = \tau\ \ell\ d_1\ d_2$. Comme pour le cas précédent, l'expression est en forme normale donc par l'hypothèse 3 on sait que la liste ℓ est égale à `nil`, ou est de la forme `cons` $t_1\ t_2$ pour t_1 et t_2 quelconques. L'expression se réduit soit en $d_2 \mathbin{//} \Lambda$, soit en $d_1\ t_1\ t_2 \mathbin{//} \Lambda$ en fonction de la forme de ℓ.
  — si $e_0 = \texttt{assign}$, alors on sait par le typage que $\bar{a} = \tau\ \&r\ s\ d$, que la référence r est de type τ et qu'elle est définie dans le contexte Λ. Le contexte Λ peut donc s'écrire $\Lambda_{\mathrm{L}} \mathbin{/} \&r : \tau = t \mathbin{/} \Lambda_{\mathrm{R}}$ (pour un terme t quelconque). Ainsi, l'expression progresse en $d \mathbin{//} \Lambda_{\mathrm{L}} \mathbin{/} \&r : \tau = s \mathbin{/} \Lambda_{\mathrm{R}}$.
  — si $e_0 = \texttt{halt}$: le typage nous garantit que $\bar{a} = \square$. De plus, comme e est en forme normale, le contexte Λ est nécessairement vide : $\Lambda = \square$. Ainsi, l'expression e ne se réduit pas, mais est exactement égale à `halt`.
  — `absurd` : ce cas ne satisfait pas l'hypothèse de correction de l'expression à réduire. En effet, la condition de vérification de l'expression n'est pas prouvable : $\complement_{\top}^{\top}(\texttt{absurd}) \triangleq \bot$.
— application de handler quelconque, $e_0 = h$: on sait par le typage que le handler h est nécessairement défini dans le contexte Λ. L'expression progresse et se réduit en $d\ \bar{a} \mathbin{//} \Lambda$ où $\bar{a}$ n'a pas changé et d est la définition du handler h dans le contexte Λ.

— fermeture, $e_0 = \pi \to e_1$: on raisonne par disjonction de cas sur la taille de $\bar{a}$.
  — si $\bar{a} = \square$, alors par typage $\pi = \square$ et donc la règle E-Void fait progresser l'expression ;
  — sinon $\bar{a} \neq \square$, alors par typage $\pi \neq \square$. En particulier, on sait que le premier argument effectif est du même type que le premier paramètre formel. Ainsi, une des règles E-AppT, E-AppV, E-AppR, E-AppH ou E-AppC s'applique.

□

3.4 Preuve de la préservation

La préservation du typage (théorème 3) indique que le réduit d'une expression de type $\square$ est également de type $\square$. Pour les besoins de la preuve, quelques lemmes intermédiaires sont requis.

Lemme 3.4. Si $\Gamma, \Gamma' \vdash e : \pi$ et $\mathrm{dom}(\Delta) \cap (\mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Gamma')) = \varnothing$, alors $\Gamma, \Delta, \Gamma' \vdash e : \pi$.

Démonstration. Par induction sur la dérivation de typage. □

Corollaire 3.2 (Affaiblissement)**.** Si $\Gamma, \Gamma' \vdash e : \pi$ et $\mathrm{dom}(\Delta) \cap \mathrm{FS}(d) = \varnothing$, alors $\Gamma, \Delta, \Gamma' \vdash e : \pi$.

Démonstration. C'est un cas particulier du lemme précédent, en effet $\mathrm{FS}(d) \subseteq (\mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Gamma'))$. □

Lemme 3.5 (Substitution de terme)**.** Si $\Gamma, x : \tau, \Delta \vdash e : \pi$ et $\vdash s : \tau$, alors $\Gamma, x : \tau, \Delta \vdash e[x \mapsto s] : \pi$.

Démonstration. Comme s est typé dans le contexte vide, la preuve de ce lemme est une simple induction sur la dérivation de typage de $\Gamma, x : \tau, \Delta \vdash e : \pi$. Il suffit de recoller la preuve de typage du terme s à chaque utilisation de la variable x. □

Lemme 3.6 (Substitution de handler)**.** D'une dérivation de $\Gamma, g\,[\bar{q}]\,\varrho, \Delta, f\,[\bar{p}]\,\varrho, \Lambda \vdash e : \pi$, on peut construire une dérivation pour le jugement $\Gamma, g\,[\bar{q}]\,\varrho, \Delta, f\,[\bar{p}]\,\varrho, \Lambda \vdash e[f \mapsto g] : \pi$.

Démonstration. Remarquons que le handler f apparaît à droite du handler g dans le contexte de typage. Ainsi, si le handler f est appliqué à une référence r dans e, le typage impose que : soit la référence apparaît dans Λ, soit elle est locale à e. Dans les deux cas, elle se trouve nécessairement à droite de g dans le contexte de typage. Le résultat peut donc être obtenu avec une induction sur la dérivation de $\Gamma, g\,[\bar{q}]\,\varrho, \Delta, f\,[\bar{p}]\,\varrho, \Lambda \vdash e : \pi$. En particulier, la règle T-AppR ne pose pas de problèmes. □

Lemme 3.7 (Contexte stationnaire)**.** Pour toute expression $e \,//\, \Lambda$ en forme normale, si $e \,//\, \Lambda \longrightarrow e' \,//\, \Lambda'$, alors le contexte Λ' ne peut différer de Λ seulement par la valeur d'une référence. En particulier, la taille, l'ordre et le types des handlers et des références définis sont préservés entre les contextes Λ et Λ'. Cette relation est notée $\sim$.

Démonstration. À l'exception de la règle définie pour le handler primitif `assign`, aucune règle de sémantique ne modifie le contexte. Dans le cas de `assign`, le contexte est identique sauf en la valeur d'une référence modifiée. Ainsi, les conditions sont préservées pour toute réduction. □

Lemme 3.8 (Préservation du typage par normalisation)**.** L'opération de normalisation préserve le typage : si $\Gamma \vdash e : \pi$, alors $\Gamma \vdash [\![e]\!]_\square : \pi$.

Démonstration. Par induction sur l'expression e, nous pouvons montrer que chaque règle de normalisation préserve le typage. □

Nous devons généraliser l'énoncé de notre théorème pour pouvoir le prouver par induction sur la dérivation de typage de l'expression qui se réduit.

Lemme 3.9 (Théorème 3 généralisé)**.** Pour toute expression normale e telle que $\Gamma_{\text{prim}} \vdash e : \square$ qui se réduit en e', pour toute décomposition $e = e_0\ \bar{a}\ //\ \Lambda_0$ où $\Gamma_0 \vdash e_0 : \pi$ et si e_0 est l'application d'un handler primitif ($e_0 = h_{\text{prim}}\ \bar{a_0}$), alors $\bar{a}$ est vide, et si e_0 est une fermeture ($\pi' \to d$), alors elle n'a pas de paramètres (π' est vide), il existe une expression e'_0 et un contexte Λ'_0 tels que $e' = e'_0\ \bar{a}\ //\ \Lambda'_0$ où $\Lambda'_0 \sim \Lambda_0$ et $\Gamma_0 \vdash e'_0 : \pi$. Énoncé formellement, cela donne

$$
\begin{aligned}
&\forall e, e'.\\
&\qquad e = [\![e]\!]_{\square} \implies\\
&\qquad \Gamma_{\text{prim}} \vdash e : \square \implies\\
&\qquad e \longrightarrow e' \implies\\
&\forall e_0, \Lambda_0, \Gamma_0, \bar{a}, \pi.\\
&\qquad e = e_0\ \bar{a}\ //\ \Lambda_0 \implies\\
&\qquad (e_0 = h_{\text{prim}}\ \bar{a_0} \implies \bar{a} = \square) \implies\\
&\qquad (e_0 = \pi' \to d \implies \pi' = \square) \implies\\
&\qquad \Gamma_0 \vdash e_0 : \pi \implies\\
&\exists e'_0, \Lambda'_0.\\
&\qquad e' = e'_0\ \bar{a}\ //\ \Lambda'_0 \wedge\\
&\qquad \Lambda'_0 \sim \Lambda_0 \wedge\\
&\qquad \Gamma_0 \vdash e'_0 : \pi
\end{aligned}
$$

Démonstration. Par induction sur la taille de la sous-expression e_0.

— cas de base :

— application de fermeture à un type, $e_0 = (\alpha\, \pi_1 \to e)\ \theta$: on cherche à typer $e'_0 = (\pi_1 \to e_1)[\alpha \mapsto \theta]$, le réduit direct de e_0 par application de la règle E-AppT. Par la structure de $\Gamma_0 \vdash e_0 : \pi_0$ on déduit $\pi_0 = \pi'_0[\alpha \mapsto \theta]$ pour un certain π'_0. Par application successive des règles T-AppT et T-ParT, nous obtenons une dérivation de $\Gamma_0 \vdash \pi_1 \to e_1 : \pi'_0$; ce qui nous permet de déduire que $\pi'_0 = \pi_1$, et donc $\pi_0 = \pi_1[\alpha \mapsto \theta]$ (égalité notée (I)). Ainsi, on peut appliquer la substitution à l'arbre de dérivation entier $\Gamma_0 \vdash \pi_1 \to e_1 : \pi_1$, pour obtenir une dérivation de $(\Gamma_0 \vdash \pi_1 \to e_1 : \pi_1)[\alpha \mapsto \theta]$. En poussant la substitution, on obtient une dérivation de $\Gamma_0[\alpha \mapsto \theta] \vdash (\pi_1 \to e_1)[\alpha \mapsto \theta] : \pi_1[\alpha \mapsto \theta]$. Comme α n'a pas d'occurrence libre dans Γ_0, on sait que $\Gamma_0[\alpha \mapsto \theta] = \Gamma_0$. Enfin, par définition de e'_0 et par l'égalité (I), nous connaissons une dérivation pour $\Gamma_0 \vdash e'_0 : \pi_0$.

— application de fermeture à un terme, $e_0 = ((x : \tau)\, \pi_1 \to e_1)\ s$: on cherche à typer $e'_0 = (\pi_1 \to e_1)[x \mapsto s]$ le réduit direct de e_0 par application de la règle E-AppV. La dérivation de typage $\Gamma_0 \vdash e_0 : \pi_0$ nous donne $\Gamma_0 \vdash s : \tau$ et $\Gamma_0 \vdash ((x : \tau)\, \pi_1 \to e_1) : (x : \tau)\, \pi_0$. Par application de la règle T-ParV, on connaît une dérivation (notée (I)) de $\Gamma_0, x : \tau \vdash \pi_1 \to e_1 : \pi_0$. De plus, comme l'expression e_0 est normale, alors $s = [\![s]\!]_{\square}$, ainsi s peut être typé dans le contexte vide (on note (II) la dérivation de $\vdash s : \tau$). Enfin, le lemme 3.5 de substitution de termes peut être appliqué à (I) et (II), ce qui nous permet de conclure $\Gamma_0 \vdash (\pi_1 \to e_1)[x \mapsto s] : \pi_0$.

— application de fermeture à une référence $e_0 = ((\&r : \tau)\, \pi_1 \to e_1)\ \&r$: on cherche à typer $e'_0 = \pi_1 \to e_1$ le réduit direct de e_0 (par application de la règle E-AppR). La dérivation de typage $\Gamma_0 \vdash e_0 : \pi_0$ nous donne $\Gamma_0^{\text{L}} \vdash ((\&r : \tau)\, \pi_1 \to e_1) : (\&r : \tau)\, \pi_0$ (où Γ_0^{L} est le contexte de typage restreint après l'application de $\&r$). On en déduit une dérivation (notée (I)) pour $\Gamma_0^{\text{L}}, \&r : \tau \vdash (\pi_1 \to e_1) : \pi_0$. Il reste à prouver $\Gamma_0 \vdash \pi_1 \to e_1 : \pi_0$, où $\Gamma_0 = \Gamma_0^{\text{L}}, \&r : \tau, \Gamma_0^{\text{R}}$. L'hypothèse (I) nous permet de déduire que $\text{dom}(\Gamma_0^{\text{R}}) \cap \text{FS}(\pi_1 \to e_1) = \varnothing$. Nous pouvons donc appliquer le résultat d'affaiblissement 3.2 pour conclure.

— application de fermeture à un handler $e_0 = ((g\, [\bar{q}]\, \varrho)\, \pi_1 \to e_1)\ f$: on cherche à typer $e'_0 = (\pi_1 \to e_1)[g \mapsto f]$ le réduit direct de e_0 (par application de la règle E-AppH). La dérivation de typage $\Gamma_0 \vdash e_0 : \pi_0$ nous donne $\Gamma_0 \vdash f : \varrho$, et d'autre part une pour $\Gamma_0 \vdash ((g\, [\bar{q}]\, \varrho)\, \pi_1 \to e_1) : (g\, [\bar{q}]\, \varrho)\, \pi_0$. Ainsi, $\Gamma_0, g\, [\bar{q}]\, \varrho \vdash \pi_1 \to e_1 : \pi_0$. Enfin, par le lemme 3.6 de substitution des lettres de handler, on conclut que $\Gamma_0 \vdash (\pi_1 \to e_1)[g \mapsto f] : \pi_0$.

— application de fermeture à une fermeture $e_0 = ((g\,[\bar{q}]\,\varrho)\,\pi_1 \to e_1)\ (\varrho \to d)$: on cherche à typer $e_0' = \pi_1 \to (e_1 \,/\, g\,[\bar{q}]\,\varrho = \downarrow d)$ le réduit direct de e_0 (par application de la règle E-AppC). Comme $\Gamma_0 \vdash e_0 : \pi_0$, on connaît une dérivation de $\Gamma_0, g\,[\bar{q}]\,\varrho, \pi_1 \vdash e_1 : \square$ que l'on note (I). De plus, on sait que la dérivation de $\Gamma_0 \vdash (\varrho \to d) : \varrho$ contient un sous arbre (noté (II)) $\Gamma_0, \varrho \vdash d : \square$. Nous pouvons donc construire la dérivation de typage souhaitée de la manière suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{(\text{II})^{\dagger}}{\Gamma_0,\ \pi_1,\ g\,[\bar{q}]\,\varrho,\ \varrho \vdash d : \square}}{\vdots}}{\Gamma_0, \pi_1,\ g\,[\bar{q}]\,\varrho \vdash \varrho \to \downarrow d : \varrho}\ \text{T-Par}\cdot \qquad \dfrac{(\text{I})^{*}}{\Gamma_0,\ \pi_1,\ g\,[\bar{q}]\,\varrho \vdash e_1 : \square}}{\Gamma_0,\ \pi_1 \vdash e_1 \,/\, g\,[\bar{q}]\,\varrho = \downarrow d : \square}\ \text{T-Defn}}{\vdots}}{\Gamma_0 \vdash e_0' = \pi_1 \to (e_1 \,/\, g\,[\bar{q}]\,\varrho = \downarrow d) : \pi}\ \text{T-Par}\cdot$$

— application de handler primitif :
  — cas $e_0 = \texttt{if true}\ e_1\ d_1$: dans ce cas particulier, on sait que $\pi = \square$ et $\bar{a} = \square$. Par hypothèse, on a une dérivation (I) pour $\Gamma_0 \vdash \texttt{if true}\ e_1\ d_1 : \square$. Par la règle sémantique correspondante, on obtient $e_0' = e_1$, donc $e' = e_1 \,//\, \Lambda_0'$. On prend $\Lambda_0' = \Lambda_0$ et $\Gamma_0 \vdash e_1 : \square$ est un sous-arbre strict de (I).
  — cas $e_0 = \texttt{if false}\ e_1\ d_1$: identique au cas précédent.
  — cas $e_0 = \texttt{unList}\ \tau\ \texttt{nil}\ e_1\ d_1$: identique au cas précédent.
  — cas $e_0 = \texttt{unList}\ \tau\ (\texttt{cons}\ t_1\ t_2)\ e_1\ d_1$: dans ce cas particulier, on sait que $\pi = \square$ et $\bar{a} = \square$. Par hypothèse, on a une dérivation (I) pour $\Gamma_0 \vdash \texttt{unList}\ \tau\ (\texttt{cons}\ t_1\ t_2)\ e_1\ d_1 : \square$. Par la règle sémantique correspondante, on obtient $e_0' = d_1\ t_1\ t_2$, donc $e' = d_1\ t_1\ t_2 \,//\, \Lambda_0'$. On prend $\Lambda_0' = \Lambda_0$ et le jugement $\Gamma_0 \vdash d_1\ t_1\ t_2 : \square$ est bien dérivable par (I). De plus, cette application est typée dans le même contexte Γ_0 car les termes t_1 et $\texttt{cons}\ t_1\ t_2$ ne sont pas des références, la règle T-AppR n'est donc pas appliquée.
  — cas $e_0 = \texttt{div}\ s\ t\ e_1$: dans ce cas particulier, on sait que $\pi = \square$ et $\bar{a} = \square$. Par hypothèse, on a une dérivation (I) pour $\Gamma_0 \vdash \texttt{div}\ s\ t\ e_1 : \square$. Par la règle sémantique correspondante, on obtient $e_0' = e_1\ n$, où n est le résultat de la division de s par t, donc $e' = e_1\ n \,//\, \Lambda_0'$. On prend $\Lambda_0' = \Lambda_0$, $n : \texttt{int}$ par définition et $\Gamma_0 \vdash e_1\ n : \square$ car $\Gamma_0 \vdash e_1 : \texttt{return []}\ \texttt{int}$ par (I). En effet, l'application est typée dans le même contexte Γ_0 car aucun des paramètres s, t, d et n n'est une référence.
  — cas $e_0 = \texttt{assign}\ \tau\ \&r\ s\ e_1$: comme dans les cas précédents, $\pi = \square$ et $\bar{a} = \square$. Par hypothèse, on a une dérivation pour $\Gamma_0 \vdash \texttt{assign}\ \tau\ \&r\ s\ e_1 : \square$, en particulier, on connaît une dérivation (notée (I)) de $\Gamma_0 \vdash e_1 : \square$. De plus, en appliquant la règle sémantique correspondante, on obtient $e_0 \longrightarrow e_1 = e_0'$. Aussi, $e' = e_1 \,//\, \Lambda_0'$ où Λ_0' est Λ_0 dans lequel la valeur associée à la référence $\&r$ a été remplacée par le terme s. Ainsi, la relation $\Lambda_0' \sim \Lambda_0$ est vérifié, et $\Gamma_0 \vdash e_1 : \square$ est exactement (I).
— assertion, $e_0 = \{\varphi\}\ e_1$: dans ce cas particulier, on sait que $\pi = \square$ et $\bar{a} = \square$. Par hypothèse, on a une dérivation (I) pour $\Gamma_0 \vdash \{\varphi\}\ e_1 : \square$. Par la règle sémantique E-Assert, on obtient $e_0' = e_1$, donc $e' = e_1 \,//\, \Lambda_0'$. On prend $\Lambda_0' = \Lambda_0$, et $\Gamma_0 \vdash e_1 : \square$ est un sous-arbre strict de (I).
— barrière fermante, $e_0 = \uparrow e_1$: dans ce cas particulier, on sait que $\pi = \square$ et $\bar{a} = \square$. Par hypothèse, on a une dérivation (I) pour $\Gamma_0 \vdash \uparrow e_1 : \square$. Par la règle sémantique E-TagU, on obtient $e_0' = e_1$, donc $e' = e_1 \,//\, \Lambda_0'$. On prend $\Lambda_0' = \Lambda_0$, et $\Gamma_0 \vdash e_1 : \square$ est un sous-arbre strict de (I).

†. La partie centrale du contexte de typage ($\pi_1, g\,[\bar{q}]\,\varrho$) ne lie rien dans l'expression d car elle provient de la fermeture appliquée dans e_0. Le résultat d'affaiblissement 3.2 permet de conclure.

∗. La syntaxe de Coma contraint la liste π_1 : elle ne peut pas contenir de références. Ainsi, l'ordre des handlers passés en paramètres à la fermeture n'a pas d'influence sur le typage.

— barrière ouvrante, $e_0 = \downarrow e_1$: identique au cas précédent.
— définition $e_0 = h$: ici, $e = h\ \bar{a}\ /\!/\ \Lambda_0$ et on sait par hypothèse que $\Gamma_0 \vdash h : \pi$ (noté (I)). Par la règle E-Defn, on obtient $e'_0 = d$, où d est la définition de h dans Λ_0. Ainsi, on a $e' = d\ \bar{a}\ /\!/\ \Lambda'_0$ où $\Lambda'_0 = \Lambda_0$. De plus, par (I) on déduit que $h\,[\bar{q}]\,\pi \in \Gamma_0$ et donc nécessairement $\Gamma_0 \vdash d : \pi$.

— cas avec IH :
— application à un type, $e_0 = e_1\ \theta$: par hypothèse on a une preuve de $\Gamma_0 \vdash e_0 : \pi_0[\alpha \mapsto \theta]$. De plus, on sait que $e = e_1\ \theta\ \bar{a}\ /\!/\ \Lambda_0$, or on peut factoriser l'expression e en $e = e_1\ \bar{a}_1\ /\!/\ \Lambda_0$ (où $\bar{a}_1 = \theta\ \bar{a}$). Par hypothèse d'induction sur $\Gamma_0 \vdash e_1 : \alpha\,\pi$, on obtient e'_1 le réduit de e_1 et une dérivation de $\Gamma_0 \vdash e'_1 : \alpha\,\pi$. Enfin, nous pouvons conclure avec la règle T-AppT qui nous permet de construire le jugement $\Gamma_0 \vdash e'_1\ \theta : \pi[\alpha \mapsto \theta]$.
— application à un terme, $e_0 = e_1\ s$: par hypothèse, on a une preuve de $\Gamma_0 \vdash e_0 : \pi$. De plus, on sait que $e = e_1\ s\ \bar{a}\ /\!/\ \Lambda_0$, or on peut factoriser l'expression e en $e = e_1\ \bar{a}_1\ /\!/\ \Lambda_0$ (où $\bar{a}_1 = s\ \bar{a}$). Par hypothèse d'induction sur $\Gamma_0 \vdash e_1 : (x : \tau)\,\pi$, on obtient e'_1 et la dérivation $\Gamma_0 \vdash e'_1 : (x : \tau)\,\pi$. Enfin, comme le contexte de typage reste Γ_0 (il n'y a pas d'application de référence), on peut construire une preuve de $\Gamma_0 \vdash e'_0 : \pi$.
— application à une référence, $e_0 = e_1\ \&r$: par hypothèse, on a une preuve de $\Gamma_0 \vdash e_0 : \pi$. De plus, on a $e = e_1\ \&r\ \bar{a}\ /\!/\ \Lambda_0$, or on peut factoriser l'expression e en $e = e_1\ \bar{a}_1\ /\!/\ \Lambda_0$ (où $\bar{a}_1 = \&r\ \bar{a}$). Par hypothèse d'induction sur $\overline{\Gamma_0} \vdash e_1 : (\&r : \tau)\,\pi$, on obtient e'_1 et la dérivation $\overline{\Gamma_0} \vdash e'_1 : (\&r : \tau)\,\pi$. Ce qui est précisément l'arbre de preuve à brancher sur notre dérivation de $\Gamma_0 \vdash e'_1\ \&r : \pi$ après application de la règle T-AppR.
— application à une fermeture, $e_0 = e_1\ (\varrho \to d)$: par hypothèse, on a une preuve de $\Gamma_0 \vdash e_0 : \pi$ (dont on note (I) le sous-arbre $\Gamma_0 \vdash (\varrho \to d) : \varrho$). De plus, on a $e = e_1\ (\varrho \to d)\ \bar{a}\ /\!/\ \Lambda_0$, or on peut factoriser l'expression e en $e = e_1\ \bar{a}_1\ /\!/\ \Lambda_0$ (où $\bar{a}_1 = (\varrho \to d)\ \bar{a}$). Par hypothèse d'induction sur $\Gamma_0 \vdash e_1 : (g\,[\bar{q}]\,\varrho)\,\pi$, on obtient e'_1 et la dérivation $\Gamma_0 \vdash e'_1 : (g\,[\bar{q}]\,\varrho)\,\pi$ (notée (II)). Enfin, comme le contexte de typage reste Γ_0 (il n'y a pas d'application de référence), on peut construire une preuve de $\Gamma_0 \vdash e'_0 : \pi$ à partir de (I) et (II) par T-AppH.
— application à un handler $e_0 = e_1\ f$: cas identique au précédent.
— allocation de référence, $e_0 = e_1\ /\ \&r : \tau = s$: par hypothèse on a une preuve de $\Gamma_0 \vdash e_0 : \square$ (on note (I) le sous arbre de typage de s). De plus, on a $e = e_1\ /\ \&r : \tau = s\ /\!/\ \Lambda_0$, et e peut se factoriser en $e = e_1\ /\!/\ \Lambda_0^+$ (où $\Lambda_0^+ = \&r : \tau = s\ /\!/\ \Lambda_0$). On peut donc appliquer notre hypothèse d'induction à $\Gamma_0, \&r : \tau \vdash e_1 : \square$. On obtient e'_1 et sa dérivation (II) $\Gamma_0, \&r : \tau \vdash e'_1 : \square$. Le but est donc prouvé en recollant les arbres :

$$\dfrac{\dfrac{}{\Gamma_0 \vdash s : \tau}\,(\mathrm{I}) \qquad \dfrac{}{\Gamma_0, \&r : \tau \vdash e'_1 : \square}\,(\mathrm{II})}{\Gamma_0 \vdash e'_1\ /\ \&r : \tau = s : \square}\ \text{T-Alloc}$$

— définition de handler, $e_0 = e_1\ /\ h\,[\bar{q}] = d$: par hypothèse, on a une preuve de $\Gamma_0 \vdash e_0 : \square$. En particulier, on sait que les variables des annotations de pré-effets appartiennent au contexte Γ_0 (on note ce fait (I)) et on connaît aussi π la signature du handler h ainsi que la preuve de dérivation relative (II). De plus, on a $e = e_1\ /\ h\,[\bar{q}] = d\ /\!/\ \Lambda_0$, et e peut se factoriser en $e = e_1\ /\!/\ \Lambda_0^+$ (où $\Lambda_0^+ = h\,[\bar{q}] = d\ /\!/\ \Lambda_0$). On peut donc appliquer notre hypothèse d'induction à $\Gamma_0, h\,[\bar{q}]\,\pi \vdash e_1 : \square$. On obtient e'_1 et une dérivation (notée (III)) du jugement $\Gamma_0, h\,[\bar{q}]\,\pi \vdash e'_1 : \square$. Le but est donc prouvé en recollant les arbres :

$$\dfrac{\dfrac{}{q : \&\tau \in \Gamma_0}\,(\mathrm{I}) \qquad \dfrac{}{\Gamma_0, h\,[\bar{q}]\,\pi \vdash d : \pi}\,(\mathrm{II}) \qquad \dfrac{}{\Gamma_0, h\,[\bar{q}]\,\pi \vdash e'_1 : \square}\,(\mathrm{III})}{\Gamma_0 \vdash e'_1\ /\ h\,[\bar{q}] = d : \square}\ \text{T-Defn}$$

□

Corollaire 3.3 (Préservation)**.** Pour tout e tel que $\Gamma_{\text{prim}} \vdash e : \square$, si $e \longrightarrow e'$, alors $\Gamma_{\text{prim}} \vdash e' : \square$.

Démonstration. Il s'agit d'un cas particulier du lemme 3.9 où $\bar{a} = \square$ et $\Lambda_0 = \square$. Ainsi $e = e_0$, $e' = e'_0$ et $\pi = \square$ ce qui revient exactement à notre énoncé. □

Conclusion

Parvenus à un point d'étape dans ce travail, récapitulons le chemin parcouru. Notre point de départ est le langage formel Coma, conçu comme langage intermédiaire pour la vérification déductive. Prônant une forme de minimalité, Coma se veut léger en nombre de constructions, tout en restant suffisamment expressif pour y traduire un langage de haut niveau.

Notre premier objectif était d'étudier en pratique les capacités de Coma en tant que représentation intermédiaire. Nous sommes alors partis de While, un langage impératif de haut niveau, pour lequel nous avons défini une sémantique formelle. Nous avons ensuite proposé une procédure de compilation, de While vers la cible Coma, que nous avons prouvé par un argument de préservation de sémantique.

Dans une seconde partie, nous avons examiné en détail le système de types de Coma. Nous avons proposé une partie de la preuve de sûreté de ce système, proche de Hindley-Milner, mais garantissant en plus l'absence d'alias entre les références (variables mutables). La preuve a été réalisée de manière traditionnelle, en suivant une méthode purement syntaxique, avec les résultats intermédiaires de progrès et de préservation. Grâce à cette propriété de sûreté du système de types, nous nous sommes assurés des fondations solides pour l'avenir du travail relatif à Coma.

Différents points restent à étudier. En effet, des résultats méta-théoriques attendus doivent être encore prouvés. En particulier, il reste à établir la préservation de la validité de la condition de vérification d'une expression qui se réduit. De la même façon, les procédures d'élimination du code fantôme et de l'état mutable de Coma (un programme est correct si et seulement si sa traduction pure l'est aussi) doivent être prouvées correctes. Aussi, la validité du calcul des effets doit être préservée par un pas de calcul. Enfin, nous travaillons en ce moment sur l'ouverture de Coma à l'ordre supérieur : actuellement, les continuations d'un handler ne peuvent pas recevoir de handler en paramètre. Nous avons pour objectif de résoudre chacune de ces tâches au début de la thèse que je vais commencer sur ce sujet.

Dans un second temps, l'implémentation du langage Coma est prévue. Il conviendra de programmer le typage, la sémantique et le calcul des conditions de vérification. Cela nous permettra de réaliser une étude à grande échelle de cas pratiques. Cette implémentation nous conduit à l'incorporation de Coma dans une plateforme de vérification déductive telle que Why3. Une première étape serait, par exemple, l'écriture du compilateur While vers Coma. De cette manière, nous parviendrons rapidement à la preuve formelle de programmes While. Ainsi, par extensions syntaxiques successives de While, nous pourrons atteindre un langage source aussi complexe que WhyML. Dans ces conditions, nous aurons la possibilité de comparer Coma à l'état de l'art, au langage WhyML en particulier. Ce sujet reste donc riche en pistes à explorer, et tout cela constitue un chantier de taille que nous envisageons de mener à terme pendant ma thèse.

Remerciements

Je tiens à remercier très chaleureusement mes encadrants de stage, Andrei et Jean-Christophe, pour le temps qu'ils m'ont consacré, pour l'intérêt qu'ils ont porté à ce travail et toutes les choses qu'ils ont su me transmettre. Ce stage a été pour moi très intéressant. Je suis enthousiaste de poursuivre ce travail en thèse dès la rentrée prochaine, au sein de la très accueillante et sympathique équipe Toccata.

Références

[1] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond et Andrei Paskevich : *The Why3 platform*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.64 édition, février 2011. `http://why3.lri.fr/`.

[2] François Bobot, Jean-Christophe Filliâtre, Claude Marché et Andrei Paskevich : Why3 : Shepherd your herd of provers. *In Boogie 2011 : First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.

[3] Dominique Clément, Thierry Despeyroux, Gilles Kahn et Joëlle Despeyroux : A simple applicative language : Mini-ML. *In Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 13–27, 1986.

[4] Edsger W. Dijkstra : Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, August 1975.

[5] Jean-Christophe Filliâtre et Andrei Paskevich : Why3 — where programs meet provers. *In* Matthias Felleisen et Philippa Gardner, éditeurs : *Proceedings of the 22nd European Symposium on Programming*, volume 7792 de *Lecture Notes in Computer Science*, pages 125–128. Springer, mars 2013.

[6] Cormac Flanagan et James B. Saxe : Avoiding exponential explosion : Generating compact verification conditions. *In Principles Of Programming Languages*, pages 193–205. ACM, 2001.

[7] Robert W. Floyd : Assigning meanings to programs. *In* J. T. Schwartz, éditeur : *Mathematical Aspects of Computer Science*, volume 19 de *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[8] Daniel P. Friedman, Christopher T. Haynes et Eugene Kohlbecker : *Programming with continuations*. Springer, 1984.

[9] C. A. R. Hoare : An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 and 583, octobre 1969.

[10] C. A. R. Hoare et Niklaus Wirth : An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2:335–355, 1973.

[11] K. Rustan M. Leino : Efficient weakest preconditions. Rapport technique MSR-TR-2004-34, Microsoft Research, 2004.

[12] Albert R. Meyer : Floyd-Hoare logic defines semantics. *Ann. Symp. on Logic In Computer Science*, 1986.

[13] Robin Milner : A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[14] F. Lockwood Morris et Cliff B. Jones : An early program proof by Alan Turing. *IEEE Annals of the History of Computing*, 6(02):139–143, 1984.

[15] Peter Naur : Proof of algorithms by general snapshots. *BIT Numerical Mathematics*, 6(4):310–316, 1966.

[16] Andrei Paskevich : Flexible verification conditions with continuations and barriers (version 2). Working paper or preprint `https://hal.archives-ouvertes.fr/hal-04115885`, 2023.

[17] Benjamin C. Pierce : *Types and programming languages*. MIT press, 2002.

[18] Guy L. Steele Jr : *Rabbit : A compiler for Scheme*. Massachusetts Institute of Technology, 1978.

[19] Alan M. Turing : Checking a large routine. *In Report of a Conference on High Speed Automatic Calculing Machines*, pages 67–69, Cambridge, 1949. Mathematical Laboratory.

[20] Andrew K. Wright et Matthias Felleisen : A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1992.

A Annexe 1

Dans cette annexe se trouvent des figures dont la lecture n'est pas nécessaire à la compréhension du texte. Elles apportent des précisions, mais auraient trop alourdi ce rapport.

```
let (x1, ..., xn) ← (1, ..., 1);
if x1 = 1 then x1 ← 0;
    ⋮
if xn = 1 then xn ← 0;
assert { x1 = ... = xn = 0 }
```

FIGURE A1 – Pseudo programme produisant une pré-condition de taille exponentielle.

$$
\begin{array}{rcl}
\mathrm{WP}(\texttt{skip}, Q, Q_E) & \triangleq & Q \\
\mathrm{WP}(\texttt{raise}\ E, Q, Q_E) & \triangleq & Q_E \\
\mathrm{WP}(e_1\ \texttt{;}\ e_2, Q, Q_E) & \triangleq & \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, Q_E), Q_E) \\
\mathrm{WP}(\texttt{try}\ e_1\ \texttt{with}\ E \to e_2, Q, Q_E) & \triangleq & \mathrm{WP}(e_1, Q, \mathrm{WP}(e_2, Q, Q_E))
\end{array}
$$

FIGURE A2 – Parallèle entre la sortie normale et exceptionnelle dans le calcul de WP traditionnel.

$$
\begin{array}{rcl}
\textit{binop} & ::= & \wedge \mid \vee \mid \rightarrow \mid \leftrightarrow \\
\textit{formula} & ::= & \textit{term} \\
& \mid & \textit{formula}\ \ \textit{binop}\ \ \textit{formula} \\
& \mid & \neg\ \textit{formula} \\
& \mid & \forall\ \textit{variable}\ \ \textit{formula} \\
& \mid & \exists\ \textit{variable}\ \ \textit{formula} \\
& \mid & \texttt{if}\ \textit{formula}\ \texttt{then}\ \textit{formula}\ \texttt{else}\ \textit{formula}
\end{array}
$$

FIGURE A3 – Définition des formules logiques.

```
if (n = 0)
   (→ { C_n^k = 1 } halt)
   (→ out0)
/ out0 =
  ↓ loop
  · / loop [line_cur ni] =
  ·   ↑ if true
  ·   ·    (→ loop2
  ·   ·       / loop2 [line_new h line_cur ki] =
  ·   ·         ↑ if (line_cur <> nil)
  ·   ·         ·    (→ unList (list int) line_cur
  ·   ·         ·         ((h: int) (t: list int)
  ·   ·         ·           → if (ni = n ∧ ki = k)
  ·   ·         ·                (→ { C_n^k = h + x } halt)
  ·   ·         ·                (→ out3)
  ·   ·         ·             / out3 =
  ·   ·         ·               ↓ (→ assign (list int) &line_new
  ·   ·         ·               ·                      (cons (h+x) line_new)
  ·   ·         ·               · (→ assign int &h x
  ·   ·         ·               · (→ assign (list int) &line_cur y
  ·   ·         ·               · (→ assign int &ki (ki+1) (→ loop2)))))
  ·   ·         ·             / &x: int = h
  ·   ·         ·             / &y: list int = t)
  ·   ·         ·         (→ absurd))
  ·   ·         ·    (→ out2)
  ·   ·       / out2 [line_cur line_new h ki] =
  ·   ·         ↓ assign (list int) &line_cur
  ·   ·         ·                   (cons 1 line_new)
  ·   ·         ·                   (→ assign int &ni (ni+1) loop)
  ·   ·       / &line_new: list int = nil
  ·   ·       / &h: int = 0
  ·   ·       / &ki: int = 0)
  ·   ·    (→ out1)
  · / out1 [line_cur ni] = ↓ halt
  · / &line_cur: list int = cons 1 nil
  · / &ni: int = 1
```

FIGURE A4 – Compilation exacte du programme WHILE calculant le coefficient binomial.

B Annexe 2

Dans cette annexe sont recopiées des définitions provenant du document de travail sur le langage Coma [16]. Il s'agit de la présentation des règles de typage, de la sémantique opérationnelle, ainsi que la définition du calcul de conditions de vérification pour une expression Coma.

Dans la figure suivante, $\mathrm{FS}(e)$ désigne les *free symbols* de l'expression e. Il s'agit de l'union de l'ensemble des variables libres et de celui des handlers libres.

$$
\begin{aligned}
\mathrm{FS}(h) &\triangleq \{h\} & \mathrm{FS}(\square \to e) &\triangleq \mathrm{FS}(e)\\
\mathrm{FS}(e\ \theta) &\triangleq \mathrm{FS}(e) \cup \mathrm{FS}(\theta) & \mathrm{FS}(\alpha\,\pi \to e) &\triangleq \mathrm{FS}(\pi \to e) \setminus \{\alpha\}\\
\mathrm{FS}(e\ s) &\triangleq \mathrm{FS}(e) \cup \mathrm{FS}(s) & \mathrm{FS}((x:\tau)\,\pi \to e) &\triangleq (\mathrm{FS}(\pi \to e) \setminus \{x\}) \cup \mathrm{FS}(\tau)\\
\mathrm{FS}(e\ \&r) &\triangleq \mathrm{FS}(e) \cup \{r\} & \mathrm{FS}((\&p:\tau)\,\pi \to e) &\triangleq (\mathrm{FS}(\pi \to e) \setminus \{p\}) \cup \mathrm{FS}(\tau)\\
\mathrm{FS}(e\ d) &\triangleq \mathrm{FS}(e) \cup \mathrm{FS}(d) & \mathrm{FS}((g\,[\bar{q}]\,\varrho)\,\pi \to e) &\triangleq (\mathrm{FS}(\pi \to e) \setminus \{g\}) \cup \mathrm{FS}(\varrho) \cup \{\bar{q}\}\\
\mathrm{FS}(\{\varphi\}\,e) &\triangleq \mathrm{FS}(e) \cup \mathrm{FS}(\varphi) & \mathrm{FS}(e\ /\ \&r:\tau = s) &\triangleq (\mathrm{FS}(e) \setminus \{r\}) \cup \mathrm{FS}(\tau) \cup \mathrm{FS}(s)\\
\mathrm{FS}(\uparrow e) &\triangleq \mathrm{FS}(e) & \mathrm{FS}(e\ /\ h\,[\bar{q}] = d) &\triangleq (\mathrm{FS}(e) \cup \mathrm{FS}(d) \cup \{\bar{q}\}) \setminus \{h\}\\
\mathrm{FS}(\downarrow e) &\triangleq \mathrm{FS}(e) & \mathrm{FS}(\varrho) &\triangleq \mathrm{FS}(\varrho \to \_) \setminus \{\_\}
\end{aligned}
$$

Figure B1 – « Free symbols », figure recopiée de la définition du langage Coma [16].

$$\frac{h\,[\bar{q}]\,\pi \in \Gamma}{\Gamma \vdash h : \pi}\ \text{(T-Ctxt)} \qquad \frac{\Gamma \vdash e : \square}{\Gamma \vdash \square \to e : \square}\ \text{(T-Void)}$$

$$\frac{\Gamma \vdash e : \alpha\,\pi}{\Gamma \vdash e\ \theta : \pi[\alpha \mapsto \theta]}\ \text{(T-AppT)} \qquad \frac{\Gamma \vdash \pi \to e : \pi \qquad \alpha \text{ is not free in } \Gamma}{\Gamma \vdash \alpha\,\pi \to e : \alpha\,\pi}\ \text{(T-ParT)}$$

$$\frac{\Gamma \vdash e : (x:\tau)\,\pi \qquad \Gamma \vdash s : \tau}{\Gamma \vdash e\ s : \pi}\ \text{(T-AppV)} \qquad \frac{\Gamma,\, x:\tau \vdash \pi \to e : \pi}{\Gamma \vdash (x:\tau)\,\pi \to e : (x:\tau)\,\pi}\ \text{(T-ParV)}$$

$$\frac{\Gamma, \Delta' \vdash e : (\&r:\tau)\,\pi \qquad \Delta' \text{ is } \Delta \text{ with all handler prototypes removed}}{\Gamma, \&r:\tau, \Delta \vdash e\ \&r : \pi}\ \text{(T-AppR)}$$

$$\frac{\Gamma \vdash e : (g\,[\bar{q}]\,\varrho)\,\pi \qquad \Gamma \vdash d : \varrho}{\Gamma \vdash e\ d : \pi}\ \text{(T-AppH)} \qquad \frac{\Gamma,\, \&p:\tau \vdash \pi \to e : \pi}{\Gamma \vdash (\&p:\tau)\,\pi \to e : (\&p:\tau)\,\pi}\ \text{(T-ParR)}$$

$$\frac{\overline{q:\&\tau} \in \Gamma \qquad \Gamma,\, g\,[\bar{q}]\,\varrho \vdash \pi \to e : \pi}{\Gamma \vdash (g\,[\bar{q}]\,\varrho)\,\pi \to e : (g\,[\bar{q}]\,\varrho)\,\pi}\ \text{(T-ParH)}$$

$$\frac{\overline{q:\&\tau} \in \Gamma \qquad \Gamma, h\,[\bar{q}]\,\pi \vdash d : \pi \qquad \Gamma, h\,[\bar{q}]\,\pi \vdash e : \square}{\Gamma \vdash e\ /\ h\,[\bar{q}] = d : \square}\ \text{(T-Defn)}$$

$$\frac{\Gamma \vdash s : \tau \qquad \Gamma, \&r:\tau \vdash e : \square}{\Gamma \vdash e\ /\ \&r:\tau = s : \square}\ \text{(T-Alloc)} \qquad \frac{\Gamma \vdash \varphi : \mathrm{Prop} \qquad \Gamma \vdash e : \square}{\Gamma \vdash \{\varphi\}\,e : \square}\ \text{(T-Assert)}$$

$$\frac{\Gamma \vdash e : \square}{\Gamma \vdash \uparrow e : \square}\ \text{(T-TagU)} \qquad \frac{\Gamma \vdash e : \square}{\Gamma \vdash \downarrow e : \square}\ \text{(T-TagD)}$$

Figure B2 – « Typing rules », figure recopiée de la définition du langage Coma [16].

Pour la sémantique, nous présentons d'abord la définition de l'opérateur de normalisation des expressions :

$$
\begin{aligned}
[\![h]\!]_\Sigma &\triangleq h & [\![e\ \theta]\!]_\Sigma &\triangleq [\![e]\!]_\Sigma\ \theta \\
[\![\pi \to e]\!]_\Sigma &\triangleq \pi \to e & [\![e\ s]\!]_\Sigma &\triangleq [\![e]\!]_\Sigma\ [\![s]\!]_\Sigma \\
[\![\{\varphi\}\ e]\!]_\Sigma &\triangleq \{[\![\varphi]\!]_\Sigma\}\ e & [\![e\ \&r]\!]_\Sigma &\triangleq [\![e]\!]_\Sigma\ \&r \\
[\![\uparrow e]\!]_\Sigma &\triangleq \uparrow e \qquad [\![\downarrow e]\!]_\Sigma \triangleq \downarrow e & [\![e\ d]\!]_\Sigma &\triangleq [\![e]\!]_\Sigma\ d
\end{aligned}
$$

$$
[\![e\ /\ \&r : \tau = s]\!]_\Sigma \triangleq \begin{cases} [\![e]\!]_{\Sigma, r \mapsto [\![s]\!]_\Sigma}\ /\ \&r : \tau = [\![s]\!]_\Sigma & \text{when } r \in \mathrm{FS}([\![e]\!]_{\Sigma, r \mapsto [\![s]\!]_\Sigma}), \\ [\![e]\!]_{\Sigma, r \mapsto [\![s]\!]_\Sigma} & \text{otherwise.} \end{cases}
$$

$$
[\![e\ /\ h\,[\bar{q}] = d]\!]_\Sigma \triangleq \begin{cases} [\![e]\!]_\Sigma\ /\ h\,[\bar{q}] = d & \text{when } h \in \mathrm{FS}([\![e]\!]_\Sigma), \\ [\![e]\!]_\Sigma & \text{otherwise.} \end{cases}
$$

Voici ensuite la liste des règles de sémantique :

$$\frac{e \neq [\![e]\!]_\square}{e \longrightarrow [\![e]\!]_\square} \quad \text{(E-Norm)}$$

$$\frac{h\,[\bar{q}] = d \in \Lambda}{h\ \bar{a}\ /\!/\ \Lambda \longrightarrow d\ \bar{a}\ /\!/\ \Lambda} \quad \text{(E-Defn)}$$

$$(\alpha\,\pi \to e)\ \theta\ \bar{a}\ /\!/\ \Lambda \longrightarrow (\pi \to e)[\alpha \mapsto \theta]\ \bar{a}\ /\!/\ \Lambda \quad \text{(E-AppT)}$$

$$((x : \tau)\,\pi \to e)\ s\ \bar{a}\ /\!/\ \Lambda \longrightarrow (\pi \to e)[x \mapsto s]\ \bar{a}\ /\!/\ \Lambda \quad \text{(E-AppV)}$$

$$((\&p : \tau)\,\pi \to e)\ \&r\ \bar{a}\ /\!/\ \Lambda \longrightarrow (\pi \to e)[p \mapsto r]\ \bar{a}\ /\!/\ \Lambda \quad \text{(E-AppR)}$$

$$((g\,[\bar{q}]\,\varrho)\,\pi \to e)\ f\ \bar{a}\ /\!/\ \Lambda \longrightarrow (\pi \to e)[g \mapsto f]\ \bar{a}\ /\!/\ \Lambda \quad \text{(E-AppH)}$$

$$((g\,[\bar{q}]\,\varrho)\,\pi \to e)\ (\varrho \to d)\ \bar{a}\ /\!/\ \Lambda \longrightarrow (\pi \to (e\ /\ g\,[\bar{q}]\,\varrho = \downarrow d))\ \bar{a}\ /\!/\ \Lambda \quad \text{(E-AppC)}$$

$$\square \to e\ /\!/\ \Lambda \longrightarrow e\ /\!/\ \Lambda \quad \text{(E-Void)}$$

$$\uparrow e\ /\!/\ \Lambda \longrightarrow e\ /\!/\ \Lambda \quad \text{(E-TagU)}$$

$$\downarrow e\ /\!/\ \Lambda \longrightarrow e\ /\!/\ \Lambda \quad \text{(E-TagD)}$$

$$\frac{\Vdash \varphi}{\{\varphi\}\ e\ /\!/\ \Lambda \longrightarrow e\ /\!/\ \Lambda} \quad \text{(E-Assert)}$$

$$\texttt{if true}\ d\ e\ /\!/\ \Lambda \longrightarrow d\ /\!/\ \Lambda \qquad \texttt{unList}\ \tau\ (\texttt{cons}\ t_1\ t_2)\ d\ e\ /\!/\ \Lambda \longrightarrow d\ t_1\ t_2\ /\!/\ \Lambda$$

$$\texttt{if false}\ d\ e\ /\!/\ \Lambda \longrightarrow e\ /\!/\ \Lambda \qquad \texttt{unList}\ \tau\ \texttt{nil}\ d\ e\ /\!/\ \Lambda \longrightarrow e\ /\!/\ \Lambda$$

$$\frac{s = n \cdot t + m \qquad 0 \leq m < |t|}{\texttt{div}\ s\ t\ e\ /\!/\ \Lambda \longrightarrow e\ n\ /\!/\ \Lambda} \qquad \texttt{assign}\ \tau\ \&r\ s\ e\ /\!/\ \Lambda, \&r : \tau = t, \mathrm{M} \longrightarrow e\ /\!/\ \Lambda, \&r : \tau = s, \mathrm{M}$$

Figure B3 – « Operational semantics », figure recopiée de la définition du langage Coma [16].

Enfin, vous retrouverez sur cette page la définition complète du calcul des conditions de vérification.

Trois opérations syntaxiques sont définies sur les formules :

$\Phi|_\pi^-$ is Φ where every VC subformula $h\ \bar{s}$, such that h is bound by π, is replaced with $\bot$.

$\Phi|_\pi^+$ is Φ where every VC subformula $h\ \bar{s}$, such that h is bound by π, is replaced with $\top$.

$\Phi|_\pi^\circ$ is Φ where we replace with $\top$ every VC subformula that has no free occurrences of handlers (acting as predicate symbols) defined inside Φ or bound by π.

Les handlers primitifs viennent avec un contrat de vérification :

$$
\begin{aligned}
&\texttt{if}: && \lambda(\texttt{c}:\texttt{bool})\ \texttt{then}\ \texttt{else}.\ (\texttt{c} \to \texttt{then}) \wedge (\neg\texttt{c} \to \texttt{else})\\
&\texttt{unList}: && \lambda\alpha\ (\texttt{l}:\texttt{list}\,\alpha)\ (\texttt{onCons}\ (\_:\alpha)\ (\_:\texttt{list}\,\alpha))\ \texttt{onNil}.\\
&&& \quad (\forall \texttt{h}:\alpha.\ \forall \texttt{t}:\texttt{list}\,\alpha.\ \texttt{l} = \texttt{cons}\ \texttt{h}\ \texttt{t} \to \texttt{onCons}\ \texttt{h}\ \texttt{t}) \wedge (\texttt{l} = \texttt{nil} \to \texttt{onNil})\\
&\texttt{div}: && \lambda(\texttt{n}:\texttt{int})\ (\texttt{m}:\texttt{int})\ (\texttt{return}\ (\_:\texttt{int}))\,.\\
&&& \quad \texttt{m} \neq 0 \wedge \forall \texttt{q}:\texttt{int}.\ \forall \texttt{r}:\texttt{int}.\ \texttt{n} = \texttt{m}\cdot\texttt{q} + \texttt{r} \wedge 0 \le \texttt{r} < |\texttt{m}| \to \texttt{return}\ \texttt{q}\\
&\texttt{assign}: && \lambda\alpha\ (\&\texttt{r}:\alpha)\ (\texttt{v}:\alpha)\ (\texttt{return}\ [\texttt{r}])\,.\,\texttt{return}\ \texttt{v}\\
&\texttt{absurd}: && \bot\\
&\texttt{halt}: && \top
\end{aligned}
$$

Définition de l'opérateur $\complement_\mathfrak{d}^\mathfrak{p}$:

$$
\mathcal{S}(h) \triangleq \begin{cases}
\lambda\pi.\Phi & \text{when } h \text{ is a primitive with contract } \lambda\pi.\Phi,\\
\lambda\pi.\, h\,\bar{q}\,\bar{z}_\pi & \text{when } h \text{ is a handler parameter } h\,[\bar{q}]\,\pi,\\
\lambda\pi.\bot & \text{when } h \text{ is defined recursively by } h\,[\bar{q}]\,\pi = d,\\
 & \quad \text{the current invocation of } h \text{ is inside } d,\\
 & \quad \text{and the result of } \mathcal{S}(h) \text{ is used in } \complement_\bot^\top(d),\\
\lambda\bar{\alpha}\pi.\, h\,\bar{\alpha}\,\bar{q}\,\bar{z}_\pi \wedge \complement_\bot^\top(d)|_\pi^\circ & \text{otherwise, when } h \text{ is defined by } h\,[\bar{q}]\,\bar{\alpha}\,\pi = d.
\end{cases}
$$

$$
\begin{aligned}
\complement_\mathfrak{d}^\mathfrak{p}(h) &\triangleq \lambda\pi.(\mathfrak{p} \to \Phi|_\pi^+) \wedge \Phi|_\pi^\circ && \text{when } \mathcal{S}(h) = \lambda\pi.\Phi\\
\complement_\mathfrak{d}^\mathfrak{p}(e\ \theta) &\triangleq (\lambda\pi.\Phi)[\alpha \mapsto \theta] && \text{when } \complement_\mathfrak{d}^\mathfrak{p}(e) = \lambda\alpha\pi.\Phi\\
\complement_\mathfrak{d}^\mathfrak{p}(e\ s) &\triangleq (\lambda\pi.\Phi)[x \mapsto s] && \text{when } \complement_\mathfrak{d}^\mathfrak{p}(e) = \lambda(x:\tau)\,\pi.\Phi\\
\complement_\mathfrak{d}^\mathfrak{p}(e\ \&r) &\triangleq (\lambda\pi.\Phi)[p \mapsto r] && \text{when } \complement_\mathfrak{d}^\mathfrak{p}(e) = \lambda(\&p:\tau)\,\pi.\Phi\\
\complement_\mathfrak{d}^\mathfrak{p}(e\ d) &\triangleq \lambda\pi.(\Phi\ /\ g\ (\overline{q:\tau_q})\ (\overline{z_\varrho:\tau_{z_\varrho}}) \equiv \Psi) && \text{when } \complement_\mathfrak{d}^\mathfrak{p}(e) = \lambda(g\,[\bar{q}]\,\varrho)\,\pi.\Phi\\
& && \quad \text{and } \complement_\mathfrak{d}^\mathfrak{p}(d) = \lambda\varrho.\Psi\\
\complement_\mathfrak{d}^\mathfrak{p}(e\ /\ \&r:\tau = s) &\triangleq \complement_\mathfrak{d}^\mathfrak{p}(e)[r \mapsto s]\\
\complement_\mathfrak{d}^\mathfrak{p}(e\ /\ h\,[\bar{q}]\,\bar{\alpha}\,\pi = d) &\triangleq (\complement_\mathfrak{d}^\mathfrak{p}(e) \wedge \forall\,\overline{q:\tau_q}.\,\forall\,\overline{z_\pi:\tau_{z_\pi}}.\,\complement_\mathfrak{p}^\bot(d)|_\pi^-)\ /\ h\,\bar{\alpha}\,(\overline{q:\tau_q})\,(\overline{z_\pi:\tau_{z_\pi}}) \equiv \complement_\bot^\top(d)|_\pi^+
\end{aligned}
$$

$$
\begin{aligned}
\complement_\mathfrak{d}^\mathfrak{p}(\pi \to e) &\triangleq \lambda\pi.\ \complement_\mathfrak{d}^\mathfrak{p}(e) \wedge \complement_{\neg\mathfrak{d}}^{\neg\mathfrak{p}}(e)|_\pi^\circ & \complement_\mathfrak{d}^\mathfrak{p}(\uparrow e) &\triangleq \complement_\mathfrak{d}^\mathfrak{d}(e)\\
\complement_\mathfrak{d}^\mathfrak{p}(\{\varphi\}\ e) &\triangleq (\mathfrak{p} \to \varphi) \wedge (\varphi \to \complement_\mathfrak{d}^\mathfrak{p}(e)) & \complement_\mathfrak{d}^\mathfrak{p}(\downarrow e) &\triangleq \complement_\mathfrak{p}^\mathfrak{p}(e)
\end{aligned}
$$

FIGURE B4 – « Verification condition generation », figures provenant du document introduisant le langage COMA [16].