Spécification et vérification de propriétés Typestates avec Frama-C

Sébastien Patte

To cite this version:

Sébastien Patte. Spécification et vérification de propriétés Typestates avec Frama-C. 2024. cea-04553229v2

HAL Id: cea-04553229
https://cea.hal.science/cea-04553229v2

Submitted on 2 May 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Spécification et Vérification de propriétés Typestates avec Frama-C

Sébastien Patte
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Résumé

Les logiciels pour systèmes critiques nécessitent des garanties fortes de sûreté et de sécurité, un bogue pouvant avoir de lourdes conséquences. Nous avons donc besoin de méthodes de vérification puissantes, comme les méthodes formelles. Frama-C est une plateforme open-source d'analyse formelle de code C, développée au CEA-List. Elle est accompagnée par ACSL, un langage de spécification formelle basé sur les contrats de fonctions.

Récemment, un greffon Frama-C a été réalisé pour vérifier des propriétés Typestates, qui restreignent les opérations possibles sur une structure de données, en fonction de son état courant. Ce greffon instrumente le programme original avec du code fantôme et des contrats ACSL, afin d'utiliser les analyseurs standard de Frama-C.

L'objectif principal de la thèse est de proposer une formalisation de cette instrumentation et de prouver sa correction. Pour cela, nous nous appuyons sur le méta-langage Skel, l'outil Necrocoq, et l'assistant de preuve Coq.

Mots-clés : Méthodes Formelles, Frama-C, Typestates, ACSL, Sémantiques Squelettiques, Skel, Coq

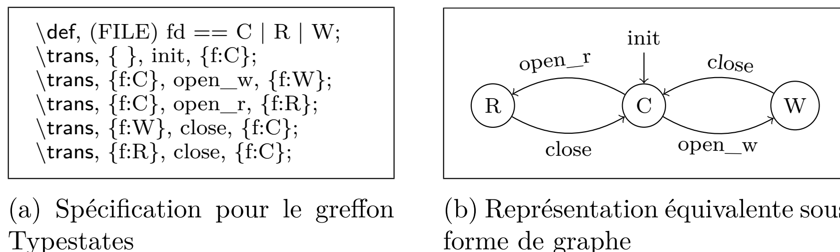
1 Typestates avec Frama-C

Frama-C[2] est une plateforme open-source d'analyse formelle de code C, développée au CEA-List. Le langage de spécification ACSL [1], utilisé par les greffons d'analyse majeurs de Frama-C, permet l'expression de contrats de fonctions, ou d'assertions évaluées à certains points de programme.

Récemment, un greffon a été conçu et développé pour spécifier et vérifier des propriétés Typestates [6]. Les Typestates servent à restreindre l'ensemble des opérations possibles sur une structure de données, en fonction de son état courant. Dans le contexte du greffon Typestates, les opérations considérées sont les fonctions d'une API. Ces fonctions sont l'unique moyen pour manipuler la structure de données depuis le code client. Le rôle du greffon est alors de vérifier que ces appels d'API respectent la spécification Typestates.

On définit une spécification Typestates par le type des objets à traquer, un ensemble de Typestates et des transitions entre ces Typestates. Chaque transition est associée à une fonction de l'API, elle indique que si chaque

objet à traquer en argument a le Typestate précisé en précondition, alors il aura le Typestate de la postcondition en sortie de la fonction. Dans le cas contraire, cet appel n'est pas autorisé.



(a) Spécification pour le greffon Typestates

(b) Représentation équivalente sous forme de graphe

FIGURE 1 – Exemple de spécification Typestate sur le type `FILE`

Par exemple, un fichier pourrait être dans 3 états différents, comme décrit dans la Figure 1 : **closed** (`C`), **read** (`R`), ou **write** (`W`). Après un appel à `init`, le fichier est dans l'état `C`, on peut alors lui appliquer la fonction `open_r` (ou `open_w`). Après cet appel on atteint l'état `R` (respectivement `W`), puis on peut appeler `close`, pour finalement retourner dans l'état `C`.

Le greffon existant instrumente le programme d'origine avec des variables et du code fantôme. Une variable fantôme, qu'on appellera un traqueur, est générée pour chaque objet à vérifier. Elle représente le Typestate de son objet en chaque point du programme. Pour chaque fonction de l'API on génère (1) un paramètre traqueur pour chaque objet à traquer dans les paramètres initiaux, (2) un contrat ACSL, (3) une liste d'instructions fantôme juste avant le point de retour de la fonction. Le contrat ACSL servira à vérifier, lors de l'analyse avec un des plugins de Frama-C, que la spécification est respectée à chaque appel. Les instructions fantômes modifient les traqueurs selon la transition déclenchée lors de l'appel, vérifiant ainsi le contrat ACSL. L'annexe A présente une instrumentation complète de l'exemple donné en Figure 1.

2 Formalisation en Skel

L'objectif principal de cette thèse est de proposer une formalisation de l'instrumentation, et de prouver sa correction. C'est-à-dire que si un programme instrumenté est valide, alors le programme d'origine respecte sa spécification Typestates. On pourra aussi montrer que l'instrumentation conserve le comportement du programme d'origine : l'instrumentation d'un programme valide s'évalue nécessairement vers la même valeur.

Dans un premier temps, on considère un mini-langage impératif inspiré du langage C, auquel on ajoute la spécification des propriétés Typestates. Ce langage comprend les appels de fonctions, les affectations et des opérations (adresse, déréférencement, opérations arithmétiques). On suppose aussi que tous les noms d'un programme d'entrée sont uniques. L'instrumentation génère un nom frais pour chaque traqueur, à partir du nom de la variable traquée. Dans le langage de sortie, on ajoute les contrats de fonctions, les

assertions logiques et les conditionnelles (nécessaires aux tests de la valeur des traqueurs dans les blocs de code fantôme). La syntaxe de ces langages en notation BNF est disponible en annexe B. L'annexe C présente une sélection des règles de sémantique, sous la forme de règles d'inférence.

Nous utilisons le méta-langage Skel [3] pour définir nos langages, leurs règles de sémantique, la fonction d'instrumentation et les fonctions utilitaires. Skel est un langage qui permet de décrire la sémantique de langages de programmation, puis d'utiliser les divers outils de la librairie Necro [5]. En particulier, utilisons l'outil Necrocoq [4] permettant de générer une formalisation Coq de nos sémantiques.

On pourra noter que les règles de sémantique et la fonction d'instrumentation se basent sur certaines fonctions utilitaires. Par exemple, dans les règles de sémantique, l'état mémoire représenté par des listes associatives, nécessite certaines fonctions pour manipuler ces listes (`find_assoc`, `rev`, `concat` ...).

3 Preuve en Coq

Necrocoq réalise un plongement profond des expressions Skel en Coq : les différentes constructions du langage Skel et leurs interprétations sont définies en Coq. Necrocoq fournit différentes interprétations du langage Skel, prouvées équivalentes, dont deux sémantiques à grand pas (inductive et itérative). On peut alors évaluer les fonctions qu'on a écrites en Skel, à partir de ces interprétations.

La version inductive se base sur l'induction de Coq, elle est plus facile à utiliser pour les cas simples. Cette version donne des principes d'induction sur les constructions du langage Skel, mais pas directement sur les fonctions qu'on a écrites en Skel. En effet, comme l'interprétation Skel est en général non-déterministe, et que l'évaluation d'une fonction écrite en Skel se fait en plusieurs étapes de réduction, on ne peut pas générer automatiquement de principes d'induction pour ces fonctions.

A contrario, la version itérative utilise une induction forte sur l'arbre de dérivation, ce qui permet par exemple de prouver des principes d'induction sur des fonctions Skel en fonction de leurs arguments. C'est donc en utilisant cette version itérative que j'ai prouvé des principes d'induction sur une partie des fonctions utilitaires et des règles de sémantique. Ces principes d'induction ont ensuite servi de base pour prouver des propriétés sur l'état mémoire. Par exemple, après l'écriture d'une valeur à un emplacement mémoire, la lecture de cet emplacement donne cette valeur, mais les valeurs aux autres emplacements restent inchangées. On en déduit alors facilement que juste après l'affectation d'une expression à une variable, l'évaluation de cette variable donne la même valeur que l'évaluation de l'expression.

Pour la suite de la preuve de correction, nous allons évaluer en même temps une instruction et son instrumentation et prouver une relation entre les états mémoire de ces évaluations. Dans un premier temps, on veut montrer

que l'état instrumenté (1) contient toutes les variables de l'état d'origine avec la même valeur associée, (2) a un traqueur pour chaque variable de Typestate dans l'état d'origine. Ces propriétés doivent être préservées par l'évaluation d'une instruction et de son instrumentation.

Dans le cas où cette instruction est un appel de fonction, sa réduction conduira, après un certain nombre d'étapes, vers l'évaluation du corps de la fonction appelée. Il nous faut donc d'abord prouver ces propriétés sur une liste d'instructions. Le principe d'induction naïf consistant à dire que si on prouve la préservation d'une propriété sur la première instruction de la liste, alors cette propriété est conservée par la suite de liste, n'est pas suffisant pour cette preuve. En effet, si l'instruction en tête de liste est un appel de fonction on a toujours le même problème : il nous faut d'abord la preuve sur une liste d'instruction. Nous utiliserons alors un principe d'induction profond, qui fait une distinction de cas en fonction de la première instruction. Si cette instruction est un appel, le prédicat inductif est supposé vrai sur l'évaluation du bloc de la fonction appelée, en plus de l'évaluation de la suite de liste, il reste alors à montrer que l'évaluation de la liste d'instructions complète préserve la propriété. Ce principe d'induction profond est déjà prouvé et devrait servir de base pour prouver la préservation des propriétés 1 et 2.

4 Conclusion

Dans cet article nous avons exposé comment le langage Skel nous permet de formaliser l'instrumentation du greffon Typestates de Frama-C, puis la manière dont Necrocoq nous permet de générer une formalisation Coq. Nous avons également présenté les propriétés déjà prouvées en Coq, ainsi que les efforts de preuve en cours.

Les prochains objectifs incluent la finalisation de la preuve de correction, puis l'expansion du langage considéré avec de nouvelles constructions telles que les conditionnelles, les boucles, ou les instructions `break` et `continue`.

Références

[1] Patrick Baudin et al. *ACSL : ANSI/ISO C Specification Language. Version 1.20.* Nov. 2024. url : `https://frama-c.com/download/acsl.pdf`.

[2] Patrick Baudin et al. « The dogged pursuit of bug-free C programs : the Frama-C software analysis platform ». Dans : *Commun. ACM* 64.8 (juill. 2021), p. 56-68. doi : `10.1145/3470569`.

[3] Martin Bodin et al. « Skeletal Semantics and their Interpretations ». Dans : *Proc. of the ACM on Programming Languages* 44 (2019), p. 1-31. doi : `10.1145/3290357`.

[4] Louis Noizet et Alan Schmitt. « Formalisation de Sémantiques Squelettiques ». Dans : *JFLA*. Jan. 2020, p. 1-14. url : https://inria.hal.science/hal-02512485.

[5] Louis Noizet et Alan Schmitt. *Stating and Handling Semantics with Skel and Necro*. Research Report RR-9449. Inria Rennes - Bretagne Atlantique, jan. 2022, p. 1-23. url : https://inria.hal.science/hal-03543701.

[6] Robert E. Strom et Shaula Yemini. « Typestate : A Programming Language Concept for Enhancing Software Reliability ». Dans : *IEEE Trans. Soft. Eng.* 12.1 (1986), p. 157-171. doi : 10.1109/TSE.1986.6312929.

A Exemple d'instrumentation

```
int init(int *f){
  *f = 0;
  return 0;
}
int close(int *f){
  *f = 0;
  return 0;
}
int open_r(int *f){
  *f = 1;
  return 0;
}
int open_w(int *f){
  *f = 2;
  return 0;
}
```

```
int main(){
  int f;
  init(&f);
  open_r(&f);
  close(&f);
  return 0;
}
//@ typestate \def, (int) fd == C | R | W;
//@ typestate \trans, {}, init, {f:C};
//@ typestate \trans, {f:C}, open_w, {f:W};
//@ typestate \trans, {f:C}, open_r, {f:R};
//@ typestate \trans, {f:W}, close, {f:C};
//@ typestate \trans, {f:R}, close, {f:C};
```

FIGURE 2 – Programme d'origine

On présente ici un exemple complet d'instrumentation, basé sur la spécification présentée dans la section 1. Dans cet exemple, on représente le type `FILE` d'un descripteur de fichier par le type `int`, afin de simplifier l'implémentation des fonctions de l'API. Dans le code d'origine, présenté dans la Figure 2, on initialise un nouveau fichier, qu'on ouvre en lecture avec `open_r`, puis referme avec `close`.

L'instrumentation de ce code produit celui qui est présenté dans la Figure 3. On remarque d'abord la définition du type enum ts qui est le type des traqueurs, représentant les 3 Typestates `C`, `R` et `W`. Chaque fonction de l'API obtient un nouveau paramètre ts_f, un contrat ACSL et une liste d'instructions fantôme. Ce nouveau paramètre est le traqueur du paramètre d'origine f, il représente son Typestate. Les contrats ACSL de ces fonctions permettent ensuite de vérifier, lors de l'analyse avec un des greffons de Frama-C, que la spécification Typestates est respectée à chaque appel. La liste d'instructions fantôme va effectivement mettre à jour le traqueur, en fonction de la transition déclenchée lors de l'appel.

Par exemple, le contrat de la fonction open_r demande à ce que le traqueur pointe vers la valeur C en précondition, puis la valeur R en postcondition. L'instruction fantôme générée teste que le traqueur pointe vers C, puis lui donne la valeur R lorsque cette condition est vérifiée. Cette instruction fantôme suffit alors à vérifier le contrat ACSL. En effet, en supposant la précondition globale $*ts_f == R$, le traqueur prend toujours la valeur R, vérifiant ainsi la postcondition.

Du côté du code client (ici la fonction main), un traqueur local ts_f est généré à partir de la variable d'origine f. À chaque appel d'une fonction d'API, le fichier étant passé par adresse &f, le traqueur est alors aussi passé par adresse en tant qu'argument fantôme supplémentaire &ts_f. La précondition \true de init est évidemment vérifiée, l'appel à init va donc mettre à jour la valeur de ts_f à C. La précondition $*ts_f == C$ de open_r est alors vérifiée, la valeur de ts_f sera donc R après l'appel à open_r. Alors, la précondition $*ts_f == R$ du deuxième comportement de close est vérifiée, le traqueur a donc la valeur C après l'appel à close. Finalement, l'assertion en fin de fonction main est bien vérifiée, vu que le traqueur a comme valeur le Typestate initial C, qui correspond à la valeur 0.

```
enum fd {
  C = 0,
  R = 1,
  W = 2
};

/*@
  requires \true;
  behavior ts_0:
    assumes \true;
    ensures *ts_c == C;
  complete behaviors;
  disjoint behaviors;
*/
int init(char *f)
  /*@ ghost (enum fd \ghost *ts_f)*/
{
  /*@ ghost *ts_f = C; */
  *f = 0;
  return 0;
}

/*@
  requires *ts_f == W ∨ *ts_f == R;
  behavior ts_0:
    assumes *ts_f == W;
    ensures *ts_f == C;
  behavior ts_1:
    assumes *ts_f == R;
    ensures *ts_f == C;
  complete behaviors;
  disjoint behaviors;
*/
int close(int *f)
  /*@ ghost (enum fd \ghost *ts_f)*/
{
  *f = 0;
  /*@ ghost
  if (*ts_f == W) {
    *ts_f = C;
  } else if (*ts_f == R) {
      *ts_f = C;
  };
  */
  return 0;
}

/*@
  requires *ts_f == C;
  behavior ts_0:
    assumes *ts_f == C;
    ensures *ts_f == R;
  complete behaviors;
  disjoint behaviors;
*/
int open_r(int *f)
  /*@ ghost (enum fd \ghost *ts_f)*/
{
  *f = 1;
  //@ ghost if (*ts_f == C) *ts_f = R;
  return 0;
}

/*@
  requires *ts_f == C;
  behavior ts_0:
    assumes *ts_f == C;
    ensures *ts_f == W;
  complete behaviors;
  disjoint behaviors;
*/
int open_w(int *f)
  /*@ ghost (enum fd \ghost *ts_f)*/
{
  *f = 2;
  //@ ghost if (*ts_f == C) *ts_f = W;
  return 0;
}

int main(){
  /*@ ghost enum fd ts_f; */
  int f;
  init(&f)/*@ ghost (&ts_f) */;
  op_r(&f) /*@ ghost (&ts_f) */;
  close(&f) /*@ ghost (&ts_f) */;
  /*@ assert typestates: ts_f == 0; */
}
```

FIGURE 3 – Programme instrumenté

B Langages

$$\texttt{cop} = \{<, >, ==, !=\}$$
$$\texttt{bop} = \{\&\&, ||\}$$
$$\texttt{aop} = \{+, -, *\}$$
$$\texttt{op} = \texttt{cop} \cup \texttt{bop} \cup \texttt{aop}$$

$$lv_1, lv_2 \in \texttt{lval} ::= \;|\; \texttt{ident} \;|\; *\,\texttt{lval}$$

$$s_1, s_2 \cdots \in \texttt{stmt} ::= \;|\; \textbf{skip} \;|\; \texttt{lval} := \texttt{expr} \;|\; \texttt{lval} := \texttt{ident}(\texttt{expr}_1, \ldots, \texttt{expr}_n) \;|\; \texttt{ident}(\texttt{expr}_1, \ldots, \texttt{expr}_n)$$

$$\vec{s_1}, \vec{s_2} \cdots \in \texttt{stmts} ::= (\texttt{stmt}\,;)^*$$

$$e_1, e_2 \cdots \in \texttt{expr} ::= \;|\; \texttt{lit} \;|\; \texttt{lval} \;|\; \&\,\texttt{ident} \;|\; \texttt{expr op expr}$$

$$\tau_1, \tau_2 \cdots \in \texttt{typ} ::= \;|\; \texttt{int} \;|\; \texttt{typ}\ *$$

$$d_1, d_2 \cdots \in \texttt{decl} ::= \texttt{typ ident}$$

$$\vec{d_1}, \vec{d_2} \cdots \in \texttt{decls} ::= (\texttt{decl}\,;)^*$$

$$b_1, b_2 \cdots \in \texttt{block} ::= \{\ \texttt{decls}\ \texttt{stmts}\ \textbf{return}\ \texttt{expr};\ \}$$

FIGURE 4 – Sous-ensemble commun du langage C

La Figure 4 décrit la syntaxe d'un sous-ensemble du langage C, qui est commun à nos deux langages : le langage des programmes d'origine et celui des programmes instrumentés. Dans ce langage on considère comme instructions les affectations, les appels de fonctions, ainsi que des instructions `skip` qui n'ont aucun effet. Dans la partie gauche des affectations, il peut y avoir un nom de variable avec un nombre arbitraire de symboles de déréférencement `*`. On nomme cette construction `lval` (left-value), la valeur de gauche.

Les expressions de ce langage peuvent être des constantes littérales entières, des `lval`, l'adresse & d'un nom de variable, ou une opération binaire entre deux expressions. Ces opérations sont réparties en trois catégories : opérateurs arithmétiques (`aop`), opérateurs logiques (`bop`) et comparaisons (`cop`).

Un type de ce langage est soit le type de base `int`, soit un pointeur vers un type. Pour l'instant `int` est le seul type de base, mais l'objectif est d'ensuite généraliser à plusieurs types de base avec les constantes littérales qui leur correspondent.

Les déclarations de variables sont alors composée d'un type et d'un nom de variables, elles sont séparées entre elles par des points-virgules.

Un bloc de code de ce langage est composé d'une liste de déclarations de variables locales, une liste d'instruction, ainsi qu'une unique instruction **return** e qui retourne une expression comme résultat.

$$
\begin{aligned}
\texttt{ts_def} &::= \text{//@ typestates \textbackslash def, } (\texttt{typ})\ \texttt{ident} == \texttt{states};\\
\texttt{states} &::= (\texttt{ident}\ |\ )^{*}\texttt{ident}\\
\texttt{trans} &::= \text{//@ typestates \textbackslash trans, } \{\texttt{binds}\}, \texttt{ident}, \{\texttt{binds}\};\\
\texttt{bind} &::= \texttt{ident : ident}\\
\texttt{binds} &::= \texttt{bind}_1, \dots, \texttt{bind}_n
\end{aligned}
$$

FIGURE 5 – Spécification Typestates

$$
\begin{aligned}
f, g, h, \dots \in \texttt{fundef} &::= \texttt{typ ident } (\texttt{decls})\ \texttt{block}\\
\texttt{main} &::= \textbf{int main}()\ \texttt{block}\\
\texttt{file} &::= \texttt{fundef}^{*}\\
&\ \texttt{main}\\
&\ (\texttt{ts_def ts_tr}^{*})
\end{aligned}
$$

FIGURE 6 – Langage d'entrée

B.1 Langage d'entrée

Le langage d'entrée, défini dans la Figure 6, est composé du sous-ensemble commun du langage C (Figure 4), de définitions de fonctions, d'une fonction `main`, ainsi que de la spécification Typestates.

Dans cette spécification Typestates, présentée dans la Figure 5, on peut définir un Typestate par son type, son nom et une liste d'états. En plus de cette définition, on donne une liste de transitions, chacune associée à une fonction de l'API. Dans chaque transition, on a une précondition qui indique le Typestate attendu en entrée pour certains arguments, ainsi qu'une postcondition qui donne le Typestate qu'auront les objets traqués en sortie de fonction. Notons que les paramètres spécifiés peuvent être différents entre la précondition et la postcondition. Cependant, les transitions non déterministes ne sont pas supportées. En particulier, tous les objets traqués doivent être spécifiés en postcondition, sinon on ne sait pas quelle est la valeur de leur Typestate après l'appel. De plus, les préconditions sur une fonction doivent être distinctes : un appel de fonction ne doit pas pouvoir déclencher plusieurs transitions.

B.2 Langage de sortie

Dans le langage de sortie, on ajoute une partie des constructions des prédicats ACSL : les constantes booléennes (\true et \false), les comparaisons arithmétiques entre deux expressions, et des opérations binaires entre ces prédicats ($\wedge$, $\vee$, et $\implies$).

On ajoute aussi les contrats ACSL aux définitions de fonctions. Chaque contrat est composé d'une précondition globale qui doit être vraie au moment de l'appel de fonction, ainsi que d'une liste de

$$\mathsf{pop} = \{\wedge, \vee, \Rightarrow\}$$

$$\begin{array}{lll} \texttt{typ} ::= \dots & \texttt{expr} ::= \dots & \texttt{pred} ::= \backslash\mathsf{true} \mid \backslash\mathsf{false} \\ \quad \mid \textbf{enum}\ \texttt{ident} & \quad \mid \texttt{ident} \quad \text{(enum tag)} & \quad \mid \texttt{expr}\ \texttt{cop}\ \texttt{expr} \\ & & \quad \mid \mathsf{pred}\ \mathsf{pop}\ \mathsf{pred} \end{array}$$

$$\begin{array}{ll} \texttt{behavior} ::= & \texttt{contract} ::= \\ \quad \textbf{behavior}\ x : & \quad /*@\ \textbf{requires}\ \texttt{pred}; \\ \qquad \textbf{assumes}\ \texttt{pred}; & \qquad \texttt{behavior}^{*}\ */ \\ \qquad \textbf{ensures}\ \texttt{pred}; & \end{array}$$

$$\texttt{enumdef} ::= \textbf{enum}\ \texttt{ident}\{\texttt{ident}, \dots, \texttt{ident}\};$$

$$\begin{array}{l} \texttt{stmt} ::= \dots \\ \qquad \mid //@\ \textbf{assert}\ \texttt{pred} \\ \qquad \mid \textbf{if}\ \texttt{expr}\ \textbf{then}\ \texttt{stmt}_1\ \textbf{else}\ \texttt{stmt}_2 \end{array}$$

$$f_1, f_2 \dots \in \texttt{fundef} ::= \texttt{contract}\ \texttt{typ}\ \texttt{ident}\ (\texttt{decls})\ \texttt{block}$$

$$\begin{array}{rl} \texttt{file} ::= & \texttt{enumdef} \\ & \texttt{fundef}^{*} \\ & \texttt{main} \end{array}$$

FIGURE 7 – Langage de sortie

comportements de la fonction. Chaque comportement est composé d'une précondition, ainsi que d'une postcondition qui sera vraie après l'appel si la précondition était vraie avant l'appel. On considère que ces comportements sont disjoints et complets, c'est-à-dire que lors d'un appel on a exactement une précondition qui doit être vérifiée. On ajoute aussi les assertions logiques, vérifiant un prédicat ACSL, parmi les instructions du langage.

Le langage de sortie comporte aussi les types énumérés qui serviront à définir le type des traqueurs, avec comme nouvelles constantes littérales les valeurs énumérées qui serviront à représenter les différents états du Typestate.

Enfin, on ajoute les conditionnelles, qui serviront à tester et mettre à jour la valeur des traqueurs, dans les listes d'instructions fantômes qui seront ajoutées dans les fonctions d'API lors de l'instrumentation.

C Sémantiques

C.1 Valeurs

Dans le cadre de nos deux sémantiques, les valeurs sont des entiers, des booléens ou des pointeurs vers un emplacement mémoire (possiblement l'emplacement `null`). L'évaluation d'un programme, qu'il soit du langage d'entrée ou de sortie, nous donne alors une valeur parmi ces trois catégories, à moins que l'évaluation ne bloque.

C.2 Spécification Typestate

On note la spécification Typestate $ts = (\tau_{ts}, x_{ts}, \overline{y}_{ts}, tr_{ts})$ où τ_{ts} est le type des variables à traquer, x_{ts} le nom du Typestate, $\overline{y}_{ts}$ la liste des états, et tr_{ts} la liste des transitions. La liste des états sera parfois notée sous la forme explicite $(y_{ts_i})_{i=1}^{n}$, en particulier on pourra faire référence au Typestate initial y_{ts_1}. On suppose que ce Typestate initial existe toujours, de toute façon une spécification Typestates sans Typestate n'a pas vraiment d'intérêt.

Afin de simplifier les notations, on considère ces valeurs comme globales dans toutes les règles de sémantiques, bien que dans la formalisation, elles soient en réalité passées de règle en règle.

C.3 État mémoire

L'état mémoire σ est représenté en Skel par trois listes associatives. On le notera $\sigma = (\sigma_l, \sigma_v, \sigma_{ts})$, où $\sigma_l : \texttt{ident} \to (\texttt{loc}, \texttt{typ})$ associe les noms de variables à leur emplacement mémoire et leurs types, $\sigma_v : \texttt{loc} \to \texttt{val}$ associe chaque emplacement mémoire à une valeur, et $\sigma_{ts} : \texttt{loc} \to \overline{y}_{ts}$ associe une valeur de Typestate aux emplacements mémoire des objets à traquer. Ce dernier environnement σ_{ts} n'est utilisé que dans la sémantique du langage d'entrée. En effet, dans le langage de sortie, les Typestate des objets sont représentés par leurs traqueurs.

$$\frac{l = (\texttt{Loc}\ \ x) \quad \sigma_l' = \sigma_l \uplus [x \mapsto (\tau, l)]}{\texttt{alloc}\ \ \tau\ x\ (\sigma_l, \sigma_v, \sigma_{ts}) = (\sigma_l', \sigma_v, \sigma_{ts}), l'}$$

$$\begin{aligned}
\texttt{write}\ \ l\ (\sigma_l, \sigma_v, \sigma_{ts})\ v &= (\sigma_l, \sigma_v \uplus [l \mapsto v], \sigma_{ts}) \\
\texttt{getloc}\ \ x\ (\sigma_l, \sigma_v, \sigma_{ts}) &= \sigma_l(x) \\
\texttt{read}\ \ l\ (\sigma_l, \sigma_v, \sigma_{ts}) &= \sigma_v(l) \\
\texttt{set_ts}\ \ l\ y\ (\sigma_l, \sigma_v, \sigma_{ts}) &= (\sigma_l, \sigma_v, \sigma_{ts} \uplus [l \mapsto y])
\end{aligned}$$

$$\frac{\texttt{alloc}\ \ \tau\ x\ \sigma_1 = (\sigma_2, l) \quad \texttt{write}\ \ l\ \sigma_2\ v = \sigma_3}{\texttt{bind_var}\ \ \tau\ x\ v\ \sigma_1 = \sigma_3}$$

FIGURE 8 – Fonctions de manipulation de l'état mémoire

On définit quelques fonctions primitives pour manipuler l'état mémoire, présentées dans la Figure 8, qu'on utilise ensuite dans les règles de sémantique.

La fonction alloc génère un nouvel emplacement mémoire l à partir du nom de variable, puis ajoute l'association $[x \mapsto (\tau, l)]$ dans σ_l. En fait, pour le moment le type des emplacements mémoire est composé d'un seul constructeur, qui prend en argument un nom de variable.

La fonction write va simplement ajouter une association emplacement/valeur $[l \mapsto v]$ dans σ_v. Quant aux fonctions getloc et read, elles regardent respectivement dans σ_l et σ_v pour renvoyer leur résultat. La fonction set_ts ajoute une association emplacement/nom $[l \mapsto x]$ dans σ_{ts}, alors que get_ts va lire le Typestate associé à un emplacement mémoire dans σ_{ts}. Enfin, la fonction bind_var fait appel à alloc pour obtenir un nouvel emplacement mémoire, puis écrit une valeur à cet emplacement.

$$\frac{\sigma'_l = \{[x \mapsto (l,\tau)] \in \sigma_l \mid (\tau\ x) \notin \overline{d}\}}{\texttt{clear}\ \overline{d}\ \sigma_l = \sigma'_l} \qquad \frac{\sigma'_{ts} = \{[(\texttt{Loc}\ x) \mapsto y] \in \sigma_{ts} \mid \forall \tau, (\tau\ x) \notin \overline{d}\}}{\texttt{clear}\ \overline{d}\ \sigma_{ts} = \sigma'_{ts}}$$

$$\frac{\sigma'_v = \{[(\texttt{Loc}\ x) \mapsto (\texttt{clear}\ \overline{d}\ v)] \mid (\forall \tau, (\tau\ x) \notin \overline{d}) \wedge [(\texttt{Loc}\ x) \mapsto v] \in \sigma_v\}}{\texttt{clear}\ \overline{d}\ \sigma_v = \sigma'_v}$$

$$\frac{\forall l, v \neq (\texttt{Ptr l})}{\texttt{clear}\ \overline{d}\ v = v} \qquad \frac{v = \texttt{Ptr}\ (\texttt{Loc}\ y) \quad \forall \tau, (\tau\ y) \notin \overline{d}}{\texttt{clear}\ \overline{d}\ v = v} \qquad \frac{v = \texttt{Ptr}\ (\texttt{Loc}\ y) \quad \exists \tau, (\tau\ y) \in \overline{d}}{\texttt{clear}\ \overline{d}\ (\texttt{Ptr}\ (\texttt{Loc}\ y)) = \texttt{Ptr null}}$$

$$\frac{\texttt{clear}\ \overline{d}\ \sigma_l = \sigma'_l \quad \texttt{clear}\ \overline{d}\ \sigma_v = \sigma'_v \quad \texttt{clear}\ \overline{d}\ \sigma_{ts} = \sigma'_{ts}}{\texttt{clear}\ \overline{d}\ (\sigma_l, \sigma_v, \sigma_{ts}) = (\sigma'_l, \sigma'_v, \sigma_{ts})}$$

FIGURE 9 – Fonction de nettoyage de l'état mémoire

À la fin de l'évaluation d'un appel de fonction (Figures 15 et 23), il faut nettoyer les variables locales de la fonction appelée. En effet, si après l'appel on se retrouve par exemple avec un pointeur qui fait référence à une de ces variables locales, alors son évaluation doit échouer. On utilise pour cela la fonction clear, définie dans la Figure 9, qui supprime dans σ toutes les occurrences des noms de $\overline{d}$. Plus précisément, la fonction va supprimer dans σ_l toutes les associations $[x \mapsto (l, \tau)]$ pour lesquelles x est un nom de $\overline{d}$. Dans σ_v et σ_{ts}, on supprime les associations ayant la clé $[(\texttt{Loc}\ x)]$ telle que x est un nom de $\overline{d}$. De plus, pour toutes les associations $[l \mapsto \texttt{Ptr y}]$ dans σ_v telles que y est un nom de $\overline{d}$, on remplace v par le pointeur null.

C.4 Accès aux définitions de fonctions

$$\frac{}{\texttt{getFundef}\ \ x_f\ (f :: \overline{g}) = (f, \overline{g})} \qquad \frac{x_f \neq x_g}{\texttt{getFundef}\ \ x_f\ (g :: \overline{h}) = \texttt{getFundef}\ \ x_f\ \overline{h}}$$

FIGURE 10 – Définition de getFundef

La Figure 10 définit une fonction primitive getFundef, qui trouve récursivement la définition d'une fonction dans une liste, à partir de son nom. En second résultat, la fonction renvoie aussi une nouvelle liste contenant toutes les fonctions définies après celle qui est cherchée. Cette fonction est utilisée lors de l'évaluation d'un appel de fonction (voir Figures 15 et 23), afin de récupérer le nom de la fonction appelée. Le bloc de la fonction appelée est alors évalué avec les définitions renvoyées par getFundef. Il

n'est donc pas possible d'évaluer des fonctions récursives ou mutuellement récursives, mais au moins on est certain que la pile d'appel sera bornée.

C.5 Évaluation des `lval`

$$\frac{\texttt{getloc}\ x\ \sigma = (l, \tau)}{\sigma \vdash_{lv} x \rightsquigarrow l} \qquad \frac{\sigma \vdash_{rv} lv \rightsquigarrow \texttt{Ptr}\ l}{\sigma \vdash_{lv} *lv \rightsquigarrow l}$$

FIGURE 11 – Évaluation d'une `lval` vers son emplacement mémoire

Le jugement $\vdash_{lv}$, présenté dans la Figure 11, évalue une `lval` vers son emplacement mémoire l. Dans le cas d'une variable, on cherche simplement l'emplacement dans l'état mémoire avec `getloc`. Dans le cas d'un déréférencement $*lv$, l'évaluation de lv avec $\vdash_{rv}$ doit donner un pointeur `Ptr` l, on renvoie alors l'emplacement mémoire pointé l.

$$\frac{\sigma \vdash_{lv} lv \rightsquigarrow l \quad \texttt{read}\ l\ \sigma = v}{\sigma \vdash_{rv} lv \rightsquigarrow v}$$

FIGURE 12 – Évaluation d'une `lval` vers sa valeur

Dans la Figure 12, on définit le jugement $\vdash_{rv}$, qui évalue une `lval` vers sa valeur associée dans l'état mémoire. Pour cela, on obtient l'emplacement de la `lval` avec $\vdash_{lv}$, puis on renvoie la valeur associée à cet emplacement mémoire grâce à la fonction `write`.

C.6 Langage d'entrée

C.6.1 Évaluation des expressions

$$\frac{}{\sigma \vdash_e i \rightsquigarrow i} \qquad \frac{\sigma \vdash_{rv} lv \rightsquigarrow v}{\sigma \vdash_e lv \rightsquigarrow v}$$

$$\frac{\vdash_e e_1 \rightsquigarrow i_1 \quad \vdash_e e_2 \rightsquigarrow i_2}{\vdash_e e_1\ \texttt{aop}\ e_2 \rightsquigarrow i_1\ \texttt{aop}\ i_2} \qquad \frac{\vdash_e e_1 \rightsquigarrow i_1 \quad \vdash_e e_2 \rightsquigarrow i_2}{\vdash_e e_1\ \texttt{cop}\ e_2 \rightsquigarrow i_1\ \texttt{cop}\ i_2} \qquad \frac{\vdash_e e_1 \rightsquigarrow b_1 \quad \vdash_e e_2 \rightsquigarrow b_2}{\vdash_e e_1\ \texttt{bop}\ e_2 \rightsquigarrow b_1\ \texttt{bop}\ b_2}$$

FIGURE 13 – Évaluation d'une expression

L'évaluation d'une expression est définie dans la Figure 13. Une constante littérale est simplement évaluée vers sa valeur entière, alors que l'évaluation d'une `lval` est déléguée à la règle $\vdash_{rv}$.

Ensuite, on a les opérations binaires, qu'on peut décomposer en deux grands cas différents : opérations arithmétiques et comparaison booléennes. Dans le cas d'une opération arithmétique ou d'une comparaison on évalue les deux expressions vers un entier, puis on applique l'opérateur sur ces entiers. Alors que pour une opération logique, les deux expressions s'évaluent vers une valeur booléenne avant qu'on applique l'opérateur.

$$\frac{}{\sigma, \overline{g} \vdash_s \texttt{skip} \leadsto \sigma}(\texttt{Skip})$$

$$\frac{\sigma_1 \vdash_{lv} lv \leadsto l \quad \sigma_1 \vdash_e e \leadsto v \quad \texttt{write}\ l\ \sigma_1\ v = \sigma_2}{\sigma_1, \overline{g} \vdash_s lv := e \leadsto \sigma_2}(\texttt{Assign})$$

$$\frac{\sigma, \overline{g} \vdash_{call} x_f(\overline{e}) \leadsto (\sigma', _)}{\sigma, \overline{g} \vdash_s x_f(\overline{e}) \leadsto \sigma'}(\texttt{Call})$$

$$\frac{\begin{array}{c}\sigma_1 \vdash_{lv} lv \leadsto l \\ \sigma_1, \overline{g} \vdash_{call} x_f(\overline{e}) \leadsto (\sigma_2, v) \\ \texttt{write}\ l\ \sigma_2\ v = \sigma_3\end{array}}{\sigma_1, \overline{g} \vdash_s lv := x_f(\overline{e}) \leadsto \sigma_3}(\texttt{AssignCall})$$

FIGURE 14 – Évaluation d'une instruction

C.6.2 Évaluation des instructions

La Figure 14 présente l'évaluation d'une instruction, avec le jugement associé $\vdash_s$ qui produit l'état mémoire suivant l'instruction à partir de l'état mémoire qui la précède. Dans le cas d'une affectation, on obtient l'emplacement mémoire l de la `lval` de gauche avec $\vdash_{lv}$, puis la valeur v de l'expression de droite avec $\vdash_e$, et on écrit la valeur v à l'emplacement l de l'état mémoire avec `write`. L'évaluation d'un appel de fonction est déchargé au jugement $\vdash_{call}$, qui produit l'état mémoire suivant l'appel et la valeur renvoyée par cet appel.

$$\frac{\forall i \in [1..n] : \texttt{bind_var}\ \tau_i\ x_i\ v_i\ \sigma_{i-1} = \sigma_i}{\sigma_0 \vdash_{args} ((\tau_i, x_i), v_i)_{i=1}^n \leadsto \sigma_n}$$

$$\frac{\begin{array}{c}\forall i \in [1..n] : \sigma_1 \vdash e_i \leadsto v_i \\ \texttt{getFundef}\ x_f\ \overline{g} = (f, \overline{h}) \qquad f = ((d_i)_{i=1}^n, b) \\ \sigma_1 \vdash_{args} (d_i, v_i)_{i=1}^n \leadsto \sigma_2 \quad \sigma_2, \overline{h} \vdash_{\texttt{block}} b \leadsto (\sigma_3, v) \\ \sigma_2, \sigma_3 \vdash_{trans} x_f \leadsto \sigma_4 \qquad \texttt{clear}\ (d_i)_{i=1}^n \sigma_4 = \sigma_5\end{array}}{\sigma_1, P_1 \vdash_{call} x_f((e_i)_{i=1}^n) \leadsto (\sigma_5, v)}$$

FIGURE 15 – Évaluation d'un appel

Ce jugement $\vdash_{call}$ est décrit dans la Figure 15. On ajoute dans l'état mémoire l'association entre les noms des paramètres formels et la valeur des arguments correspondants grâce au jugement $\vdash_{args}$. Notons qu'il n'est pas nécessaire d'ajouter de nouvelles associations dans l'environnement des Typestates σ_{ts} pour ces paramètres formels. En effet, les paramètres qu'on veut traquer sont des pointeurs, l'adresse pointée est donc déjà associée à un Typestate dans σ_{ts}. On utilise ensuite la fonction `getFundef` pour chercher la définition de la fonction appelée f, ainsi que la liste des fonctions qui sont définies après f. C'est à partir du nouvel état mémoire σ_2 et cette nouvelle liste de fonctions, qu'on évalue le bloc de la fonction avec $\vdash_{block}$, ce qui nous donne un nouvel état mémoire σ_3 et la valeur de retour v. Avant de finalement renvoyer l'état mémoire, il nous reste d'abord à appliquer les éventuelles

transitions Typestate avec $\vdash_{trans}$ (Figure 17), puis effacer les variables locales de l'état mémoire avec $\vdash_{clear}$.

$$\text{(Untracked)}\ \frac{\tau_{ts} \neq \tau \quad \texttt{alloc}\ \tau\ x\ \sigma_1 = (\sigma_2, l) \quad \texttt{write}\ l\ \sigma_2\ 0 = \sigma_3}{\sigma_1 \vdash_{initd} (\tau, x) \rightsquigarrow \sigma_3} \qquad \frac{\tau_{ts} = \tau \quad \texttt{alloc}\ \tau\ x\ \sigma_1 = (\sigma_2, l) \quad \texttt{set_ts}\ l\ y_1\ \sigma_2 = \sigma_3}{\sigma_1 \vdash_{initd} (\tau, x) \rightsquigarrow \sigma_3}\ \text{(Tracked)}$$

$$\frac{\forall i \in [1..n] : \sigma_{i-1} \vdash_{initd} d_i \rightsquigarrow \sigma_i}{\sigma_0 \vdash_{initds} (d_i)_{i=1}^n \rightsquigarrow \sigma_n} \qquad \frac{\forall i \in [1..n] : \sigma_{i-1} \vdash_{s} s_i \rightsquigarrow \sigma_i}{\sigma_0 \vdash_{stmts} (s_i)_{i=1}^n \rightsquigarrow \sigma_n}$$

$$\frac{\sigma_1 \vdash_{initds} \overline{d} \rightsquigarrow \sigma_2 \quad \sigma_2 \vdash_{stmts} \overline{s} \rightsquigarrow \sigma_3 \quad \sigma_3 \vdash_e e \rightsquigarrow v \quad \texttt{clear}\ \overline{d}\ \sigma_3 = \sigma_4}{\sigma_1 \vdash_{block} (\overline{d}, \overline{s}, e) \rightsquigarrow (\sigma_4, v)}$$

Figure 16 – Évaluation d'un bloc

Lors de l'évaluation d'un bloc, décrit dans Figure 16, on initialise d'abord chaque variable locale dans l'état mémoire avec des appels répétés à $\vdash_{initd}$, ce qui nous donne un nouvel état σ_2. À partir de cet état σ_2, on évalue la liste d'instructions $\overline{s}$, ce qui nous amène à l'état σ_3. Puis on évalue l'expression de retour à partir de cet état σ_3, ce qui nous donne la valeur de retour à renvoyer. Finalement, on efface les occurrences des variables de $\overline{d}$ dans σ_3, avant de renvoyer l'état mémoire.

Le jugement $\vdash_{initd}$, qui initialise les variables locales, fait une distinction entre les variables traquées qui ont le type du Typestate τ_{ts} et les variables non traquées.

Pour les variables non traquées, on alloue un nouvel emplacement mémoire avec `alloc`, puis on écrit la valeur 0 à cet emplacement mémoire. Pour les variables traquées on fait la même initialisation que pour les autres, mais en plus on lui associe le Typestate initial τ_{ts_1} dans l'état mémoire avec `set_ts`.

$$\frac{\forall i \in [1..n] :\ \sigma \vdash_{rv} x_i \rightsquigarrow (\texttt{Ptr}\ \ l_i) \quad \texttt{get_ts}\ l_i\ \sigma = y_i}{\sigma \vdash_{pre} (x_i, y_i)_{i=1}^n}$$

$$\frac{\forall i \in [1..n] :\ \sigma_{i-1} \vdash_{rv} x_i \rightsquigarrow (\texttt{Ptr}\ \ l_i) \quad \texttt{set_ts}\ l_i\ y_i\ \sigma_{i-1} = \sigma_i}{\sigma_0 \vdash_{post} (x_i, y_i)_{i=1}^n \rightsquigarrow \sigma_n}$$

$$\frac{\nexists (pre, x_f, post) \in tr}{\sigma \vdash_{trans} x_f \rightsquigarrow \sigma}\text{(notrans)}$$

$$\frac{\exists!\ (pre, x_f, post) \in tr : \sigma_1 \vdash pre \quad \sigma_2 \vdash post \rightsquigarrow \sigma_3}{\sigma_1, \sigma_2 \vdash_{trans} x_f \rightsquigarrow \sigma_3}\text{(trans)}$$

Figure 17 – Transition de Typestate

Le jugement $\vdash_{trans}$, présenté dans la Figure 17, va mettre à jour le Typestate des variables traquées dans l'état mémoire, en fonction de la transition déclenchée lors de l'appel d'une fonction. Si aucune transition ne correspond à la fonction appelée, alors on est dans le cas d'une fonction du code client, l'état mémoire n'est donc pas modifié. Dans le cas contraire, la fonction est une fonction d'API, on doit donc avoir une unique transition qui est déclenchée lors de l'appel. On utilise alors le jugement $\vdash_{pre}$

pour vérifier dans l'état mémoire que les valeurs des variables traquées correspondent à la précondition. Ensuite, le jugement $\vdash_{post}$ va mettre à jour le Typestate de ces variables dans l'état mémoire, pour les faire correspondre à la postcondition. Cette postcondition est évaluée dans l'état d'après l'évaluation du bloc de la fonction d'API appelée (voir Figure 15), on suppose alors que l'adresse pointée par les paramètres traqués n'a pas changé au cours de l'évaluation de ce bloc.

C.7 Langage de sortie

C.7.1 Évaluation des énumérations

$$\frac{}{\vdash \textbf{enum}\ x\ \{(y_i)_{i=1}^{n}\} \rightsquigarrow E \cup [y_i \mapsto i]_{i=1}^{n}}$$

Figure 18 – Évaluation d'une énumération

On considère un environnement global E : `ident` $\mapsto$ `nat`, qui associe chaque valeur énumérée à sa valeur entière. La Figure 18 présente la règle d'évaluation d'une énumération, qui associe dans E chaque valeur énumérée à son indice.

C.7.2 Évaluation des expressions et prédicats

$$\frac{}{\sigma \vdash_e i \rightsquigarrow i} \qquad \frac{\sigma \vdash_{rv} lv \rightsquigarrow v}{\sigma \vdash_e lv \rightsquigarrow v}$$

$$\frac{\vdash e_1 \rightsquigarrow i_1 \quad \vdash e_2 \rightsquigarrow i_2}{\vdash e_1\ \texttt{aop}\ e_2 \rightsquigarrow i_1\ \texttt{aop}\ i_2} \qquad \frac{\vdash e_1 \rightsquigarrow i_1 \quad \vdash e_2 \rightsquigarrow i_2}{\vdash e_1\ \texttt{cop}\ e_2 \rightsquigarrow i_1\ \texttt{cop}\ i_2} \qquad \frac{\vdash e_1 \rightsquigarrow b_1 \quad \vdash e_2 \rightsquigarrow b_2}{\vdash e_1\ \texttt{bop}\ e_2 \rightsquigarrow b_1\ \texttt{bop}\ b_2}$$

$$\frac{y \in dom(E)}{\sigma \vdash_e y \rightsquigarrow E(y)}$$

Figure 19 – Évaluation d'une expression

L'évaluation d'une expression est présentée dans la Figure 19, avec en gris les règles qui n'ont pas changé par rapport au langage d'entrée. La seule nouvelle règle est celle sur les valeurs énumérées, qui va simplement chercher la valeur correspondante dans E, l'environnement global des énumérations.

$$\frac{}{\sigma \vdash b \rightsquigarrow b} \qquad \frac{\vdash e_1 \rightsquigarrow i_1 \quad \vdash e_2 \rightsquigarrow i_2}{\vdash e_1\ \texttt{cop}\ e_2 \rightsquigarrow i_1\ \texttt{cop}\ i_2} \qquad \frac{\vdash_p p_1 \rightsquigarrow b_1 \quad \vdash_p p_2 \rightsquigarrow b_2}{\vdash p_1\ \texttt{pop}\ p_2 \rightsquigarrow b_1\ \texttt{pop}\ b_2}$$

Figure 20 – Évaluation d'un prédicat ACSL

L'évaluation d'un prédicat ACSL vers une valeur booléenne est décrit dans la Figure 20. On remarquera qu'il y a globalement trois manières d'évaluer ces prédicats. Soit le prédicat est déjà une valeur booléenne, soit on a une comparaison arithmétique, ou bien une comparaison booléenne entre deux prédicats. Pour la comparaison entre deux expressions, on évalue d'abord ces expressions vers un entier avant de faire la comparaison. De même pour la comparaison des prédicats, on les évalue vers une valeur booléenne avant de les comparer.

C.7.3 Évaluation des instructions

On présente l'évaluation des instructions dans la Figure 21. Parmi les nouvelles règles, on a d'abord les assertions, pour lesquelles on vérifie simplement que leur prédicat s'évalue vers `true` avec $\vdash p$.

$$\frac{}{\sigma, \overline{g} \vdash_s \texttt{skip} \rightsquigarrow \sigma}(\texttt{Skip})$$

$$\frac{\sigma_1 \vdash_{lv} lv \rightsquigarrow l \quad \sigma_1 \vdash_e e \rightsquigarrow v \quad \texttt{write}\ l\ \sigma_1\ v = \sigma_2}{\sigma_1, \overline{g} \vdash_s lv := e \rightsquigarrow \sigma_2}(\texttt{Assign})$$

$$\frac{\sigma, \overline{g} \vdash_{call} x_f(\overline{e}) \rightsquigarrow (\sigma', _)}{\sigma, \overline{g} \vdash_s x_f(\overline{e}) \rightsquigarrow \sigma'}(\texttt{Call})$$

$$\frac{\begin{array}{c}\sigma_1 \vdash_{lv} lv \rightsquigarrow l \\ \sigma_1, \overline{g} \vdash_{call} x_f(\overline{e}) \rightsquigarrow (\sigma_2, v) \\ \texttt{write}\ l\ \sigma_2\ v = \sigma_3\end{array}}{\sigma_1, \overline{g} \vdash_s lv := x_f(\overline{e}) \rightsquigarrow \sigma_3}(\texttt{AssignCall})$$

$$\frac{\sigma \vdash_p pred \rightsquigarrow \texttt{true}}{\sigma, \overline{g} \vdash_s \textbf{assert}\ pred \rightsquigarrow \sigma}(\texttt{Assert})$$

$$\frac{\sigma \vdash_e e \rightsquigarrow \texttt{true} \quad \sigma, \overline{g} \vdash_s s_1 \rightsquigarrow \sigma'}{\sigma, \overline{g} \vdash_s \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \rightsquigarrow \sigma'}(\texttt{If-true})$$

$$\frac{\sigma \vdash_e e \rightsquigarrow \texttt{false} \quad \sigma, \overline{g} \vdash_s s_2 \rightsquigarrow \sigma'}{\sigma, \overline{g} \vdash_s \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2 \rightsquigarrow \sigma'}(\texttt{If-false})$$

FIGURE 21 – Évaluation d'une instruction

Ensuite, on définit les règles d'évaluation de la conditionnelle, pour lesquelles on évalue l'expression de test vers une valeur booléenne, afin de décider si on évalue la branche **then** ou bien la branche **else**.

$$\frac{\begin{array}{c}\texttt{alloc}\ \tau\ x\ \sigma_1 = (\sigma_2, l) \\ \texttt{write}\ l\ \sigma_2\ 0 = \sigma_3\end{array}}{\sigma_1 \vdash_{initd} (\tau, x) \rightsquigarrow \sigma_3}$$

$$\frac{\forall i \in [1..n] : \sigma_{i-1} \vdash_{initd} d_i \rightsquigarrow \sigma_i}{\sigma_0 \vdash_{initds} (d_i)_{i=1}^n \rightsquigarrow \sigma_n} \qquad \frac{\forall i \in [1..n] : \sigma_{i-1} \vdash_{stmts} s_i \rightsquigarrow \sigma_i}{\sigma_0 \vdash_{stmts} (s_i)_{i=1}^n \rightsquigarrow \sigma_n}$$

$$\frac{\sigma_1 \vdash_{initds} \overline{d} \rightsquigarrow \sigma_2 \quad \sigma_2 \vdash_s s \rightsquigarrow \sigma_3 \quad \sigma_3 \vdash_e e \rightsquigarrow v \quad \texttt{clear}\ \overline{d}\ \sigma_3 = \sigma_4}{\sigma_1 \vdash_{block} (\overline{d}, s, e) \rightsquigarrow (\sigma_4, v)}$$

FIGURE 22 – Évaluation d'un bloc

L'évaluation d'un bloc, décrite dans la Figure 22 est très similaire à celle qu'on fait dans le langage d'entrée (Figure 16). La seule différence est que lorsqu'on initialise les variables locales, on se contente de les associer à la valeur 0 dans l'état mémoire. En effet, les transitions sont maintenant gérées avec les traqueurs et les contrats ACSL, on n'a donc plus besoin de stocker directement la valeur du Typestate des variables dans l'état mémoire.

L'évaluation d'un appel de fonction, décrit dans la Figure 23, ressemble à celle qu'on fait dans le

$$\frac{\forall i \in [1..n] : \texttt{bind_var}\ \tau_i\ x_i\ v_i\ \sigma_{i-1} = \sigma_i}{\sigma_0 \vdash_{args} ((\tau_i, x_i), v_i)_{i=1}^n \rightsquigarrow \sigma_n}$$

$$\frac{\begin{array}{c}\forall i \in [1..n] : \sigma_1 \vdash e_i \rightsquigarrow v_i \\ \texttt{getFundef}\ x_f\ \overline{g} = (f, \overline{h}) \quad f = (x_f, (d_i)_{i=1}^n, b, c) \\ \sigma_1 \vdash_{args} (d_i, v_i)_{i=1}^n \rightsquigarrow \sigma_2 \quad \sigma_2, \overline{g} \vdash_{block} b \rightsquigarrow (\sigma_3, v) \\ \sigma_1, \sigma_3 \vdash_{contract} c \qquad \texttt{clear}\ (d_i)_{i=1}^n \sigma_3 = \sigma_4\end{array}}{\sigma_1, \overline{g} \vdash_{call} f((e_i)_{i=1}^n) \rightsquigarrow (\sigma_4, v)}$$

Figure 23 – Évaluation d'un appel de fonction

langage d'entrée (Figure 15). Seulement, au lieu de gérer les transitions avec $\vdash_{trans}$, on vérifie avec $\vdash_{contract}$ que les états d'avant l'appel (σ_1) et d'après l'évaluation du bloc (σ_3) vérifient le contrat ACSL.

$$\frac{\begin{array}{c}\sigma_{pre} \vdash_{pred} p \rightsquigarrow \texttt{true} \\ \exists!\ (p_1, p_2) \in \overline{bh} :\ \sigma_{pre} \vdash_{pred} p_1 \rightsquigarrow \texttt{true} \\ \sigma_{post} \vdash_{pred} p_2 \rightsquigarrow \texttt{true}\end{array}}{\sigma_{pre}, \sigma_{post} \vdash_{contract} (p, \overline{bh})}$$

$$\frac{\sigma_{pre} \vdash_{pred} p \rightsquigarrow \texttt{true}}{\sigma_{pre}, \sigma_{post} \vdash_{contract} (p, [\,])}$$

Figure 24 – Évaluation d'un contrat de fonction ACSL

Ce jugement $\vdash_{contract}$ est présenté dans la Figure 24. Dans tous les cas, la précondition globale doit être vérifiée dans l'état avant l'appel σ_{pre}. Si on a plusieurs comportements dans le contrat, alors on doit avoir un seul de ces comportements dont la précondition est vraie dans σ_{pre}, sa postcondition doit alors être vraie dans l'état σ_{post}.

D Instrumentation

$$[\![x_{ts}, \overline{y}_{ts}]\!]^{enum} = \textbf{enum } x_{ts}\ \{\overline{y}_{ts}\}$$

FIGURE 25 – Génération du type des traqueurs

La fonction $[\![\cdot]\!]^{enum}$, décrite dans Figure 25, construit la définition du type énuméré correspondant aux traqueurs, à partir de la spécification Typestates. Plus précisément, ce type énuméré a comme nom celui du Typestate x_{ts}, et comme valeurs énumérées les noms des différents états du Typestate $\overline{y}_{ts}$.

Au cours de l'instrumentation, on va remplir et utiliser un environnement des traqueurs fantômes. Cet environnement γ : `ident` → `decl` associe chaque nom de variable du programme d'origine à la déclaration de son traqueur.

$$\begin{aligned} [\![x]\!]^{\text{arg}}_{\gamma} &= \texttt{snd}\ \gamma(x) \\ [\![*lv]\!]^{\text{arg}}_{\gamma} &= *[\![lv]\!]^{\text{arg}}_{\gamma} \\ [\![\&x]\!]^{\text{arg}}_{\gamma} &= \&[\![x]\!]^{\text{arg}}_{\gamma} \end{aligned} \qquad [\![(lv_i)_{i=1}^{n}]\!]^{args}_{\gamma} = ([\![lv_i]\!]^{arg}_{\gamma})_{i=1}^{n}$$

FIGURE 26 – Instrumentation d'une expression contenant une variable à traquer

Dans la Figure 26, on présente les fonctions qui servent à remplacer les variables d'origines par leur traqueur. Plus précisément, la fonction $[\![\cdot]\!]^{arg}_{\gamma}$ remplace les noms de variables d'origine dans une expression par le nom des traqueurs associés, à partir de l'environnement γ. Cette fonction est définie sur les cas d'une variable, d'une prise d'adresse et d'un déréférencement, elle échoue dans les autres cas (constante, opérateur binaire ...).

Pour transformer ainsi une liste d'expression on utilise la fonction $[\![\cdot]\!]^{args}_{\gamma}$. Chaque expression transformée avec succès par un appel à $[\![\cdot]\!]^{arg}_{\gamma}$ est alors gardée dans la liste finale, tandis que les autres sont ignorées.

$$[\![lv := e]\!]^{trackers}_{\gamma} = [\![lv]\!]^{arg}_{\gamma} := [\![e]\!]^{arg}_{\gamma}$$

FIGURE 27 – Instrumentation d'une affectation

La fonction $[\![\cdot]\!]^{trackers}_{\gamma}$, définie dans la Figure 27, produit une affectation de traqueur à partir d'une affectation sur une variable d'origine. On notera que cette fonction ne renvoie un résultat que lorsque chaque côté de l'affectation est traduit avec succès par $[\![\cdot]\!]^{arg}_{\gamma}$. En particulier, l'expression à droite ne peut pas être une constante, mais doit contenir une variable traquée. En effet, les variables traquées ne peuvent pas être initialisées directement depuis le code client, mais via des appels aux fonctions de l'API.

$$
\begin{aligned}
[\![(x_i, y_i)_{i=1}^n]\!]_\gamma^{comp_pre} &= \texttt{\textbackslash true}\ (\texttt{\&\&}\ *\ [\![x_i]\!]_\gamma^{arg} \texttt{==}\ y_i)_{i=1}^n \\
[\![(x_i, y_i)_{i=1}^n]\!]_\gamma^{assigns_post} &= (*[\![x_i]\!]_\gamma^{arg} := y_i)_{i=1}^n \\
\\
[\![[\,]]\!]_\gamma^{ts_stmt} &= \textbf{skip} \\
[\![(\{(pre, x_f, post)\} :: trans)]\!]_\gamma^{ts_stmt} &= \textbf{if}\ [\![pre]\!]_\gamma^{comp_pre} \\
&\quad\ \textbf{then}\ [\![post]\!]_\gamma^{assigns_post} \\
&\quad\ \textbf{else}\ [\![trans]\!]_\gamma^{ts_stmt} \\
[\![(\{(pre, x_f, post)\} :: trans)]\!]_\gamma^{ts_stmt} &= [\![post]\!]_\gamma^{assigns_post} \qquad \text{(si } pre \text{ est vide)} \\
\\
[\![\overline{d}, \overline{s}, e]\!]_{trans,\gamma}^{api_block} &= (\overline{d}, (\overline{s} +\!\!+ [\![trans]\!]_\gamma^{ts_stmt}), e)
\end{aligned}
$$

FIGURE 28 – Instrumentation du bloc d'une fonction d'API

D.1 Instrumentation de code API

Dans la Figure 28, on présente la fonction $[\![\cdot]\!]_\gamma^{api_block}$, qui instrumente le bloc d'une fonction d'API, en y insérant une liste d'instructions fantôme juste avant le point de retour. Cette liste est un embranchement de conditionnelles, générée à partir de la liste des transitions sur x_f, qui donne la valeur attendue en postcondition pour chaque traqueur en fonction de la transition dont la précondition est vraie avant ces instructions fantôme.

$$
\begin{aligned}
[\![(x_i : y_i)_{i=1}^n]\!]_\gamma^{binds_pred} &= \texttt{\textbackslash true} \bigwedge_{i=1}^{n} (*[\![x_i]\!]_\gamma^{arg} \texttt{==}\ y_i) \\
[\![((pre, x_f, post) :: trans)]\!]_\gamma^{behav} &= ([\![pre]\!]_\gamma^{binds_pred}, [\![post]\!]_\gamma^{binds_pred}) \\
&\quad :: [\![trans]\!]_\gamma^{behav} \\
[\![(p_i, _)_{i=1}^n]\!]^{requires} &= \texttt{\textbackslash false} \bigvee_{i=1}^{n} p_i \\
[\![trans]\!]^{contract} &= [\![[\![trans]\!]_\gamma^{behav}]\!]^{requires}, [\![trans]\!]_\gamma^{behav}
\end{aligned}
$$

FIGURE 29 – Génération du contrat ACSL pour une fonction d'API

La Figure 29 présente la génération d'un contrat ACSL pour une fonction de l'API. On a d'abord la fonction $[\![\cdot]\!]^{binds_pred}$, qui transforme une précondition ou une postcondition vers un prédicat ACSL. Ce contrat vérifie alors que le traqueur de chaque paramètre mentionné pointe vers le Typestate attendu.

Pour générer les (pré/post)-conditions des différents comportements de la fonction, la fonction $[\![\cdot]\!]^{behav}$ utilise $[\![\cdot]\!]^{bind_pred}$ sur les (pré/post)-conditions de chaque transition associée à la fonction.

On va aussi générer une précondition globale avec $[\![\cdot]\!]^{requires}$, en concaténant les préconditions de

tous les comportements de la fonction, précédemment générés grâce à $[\![\cdot]\!]^{behav}$.

D.2 Instrumentation de code client

$$
\begin{aligned}
[\![\mathbf{skip}]\!]_\gamma^{cli_st} &= \mathbf{skip} \\
[\![lv := e]\!]_\gamma^{cli_st} &= \begin{cases} lv := e;\ [\![lv := e]\!]_\gamma^{trackers} & \\ lv := e & \text{(si } [\![lv := e]\!]_\gamma^{trackers} \text{ échoue)} \end{cases} \\
[\![x_f(\overline{e})]\!]_\gamma^{cli_st} &= x_f(\overline{e}, [\![\overline{e}]\!]_\gamma^{args}) \\
[\![lv := x_f(\overline{e})]\!]_\gamma^{cli_st} &= lv := x_f(\overline{e}, [\![\overline{e}]\!]_\gamma^{args}) \\
[\![\overline{s}]\!]_\gamma^{cli_stmts} &= ([\![s_i]\!]_\gamma^{cli_st})_{i=1}^n
\end{aligned}
$$

FIGURE 30 – Instrumentation des instructions du code-client

La fonction $[\![\cdot]\!]_\gamma^{cli_st}$, définie dans la Figure 30, instrumente une instruction venant du code-client. Pour l'instrumentation d'une affectation, on insère l'affectation de traqueur correspondante, quand elle peut être générée avec $[\![\cdot]\!]_\gamma^{trackers}$. Dans les appels de fonctions, la liste des arguments traqueurs, générée avec $[\![\cdot]\!]_\gamma^{args}$, est ajoutée à la fin des arguments d'origine.

$$
\begin{aligned}
[\![\tau_{ts}]\!]^{ty} &= \mathbf{enum}\ x_{ts} \\
[\![*\tau]\!]^{ty} &= *[\![\tau]\!] \\
[\![(\tau, x)]\!]^{decl} &= ([\![\tau]\!]^{ty}, \mathtt{ts}(x)) \\
[\![(\tau_1 x_1)_{i=1}^n]\!]_\gamma^{locals} &= \begin{array}{c} (\tau_i x_i)_{i=1}^n +\!\!+ ([\![\tau_i x_i]\!]^{decl})_{i=1}^n, \\ \gamma \cup [x_i \mapsto [\![\tau_i x_i]\!]^{decl}]_{i=1}^n \end{array} \\
[\![\gamma]\!]^{assert} &= \forall [x \mapsto (\mathbf{enum}\ x_{ts}, y)] \in \gamma,\ \mathbf{assert}\ y = y_{ts_1}
\end{aligned}
$$

$$
\frac{[\![\overline{d}]\!]_{\gamma_1}^{locals} = (\overline{d'}, \gamma_2)}{[\![\overline{d}, \overline{s}, e]\!]_{\gamma_1}^{cli_block} = (\overline{d'}, ([\![\overline{s}]\!]_{\gamma_2}^{cli_stmts}; [\![\gamma_2 \backslash \gamma_1]\!]^{assert}), e)}
$$

FIGURE 31 – Instrumentation d'un bloc du code-client

Dans la Figure 31, on présente l'instrumentation d'un bloc de code client. La fonction $[\![\cdot]\!]^{ty}$ transforme un type τ vers sa version traqueur, à partir du moment où ce type τ pointe vers le type spécifié pour les objets du Typestate τ_{ts}. C'est-à-dire que le type τ_{ts} qui apparaît dans τ est remplacé par le type des traqueurs **enum** x_{ts}.

La fonction $[\![\cdot]\!]^{decl}$ transforme une déclaration vers sa version traqueur : le type τ est traduit avec $[\![ty]\!]$ et le nom du traqueur est généré à partir du nom d'origine de la variable. Pour transformer une liste de déclarations, on utilise la fonction $[\![\cdot]\!]^{locals}$, qui y ajoute les déclarations traduites vers leur version traqueur. Cette fonction renvoie aussi en second résultat une nouvelle version de la table des

traqueurs γ, où on a ajouté les associations entre les noms d'origine et les noms traduits avec succès grâce à $[\![\cdot]\!]^{decl}$.

L'instrumentation d'un bloc du code-client se fait alors avec la fonction $[\![\cdot]\!]^{cli_block}$. Les déclarations de variables locales sont étendues avec leur version traqueur grâce à $[\![\cdot]\!]^{locals}$, on obtient par la même occasion la table des traqueurs γ qui contient les associations entre les noms des variables d'origine et les déclarations de traqueurs générées. Puis, on instrumente les instructions du bloc avec $[\![\cdot]\!]_{\gamma}^{cli_stmts}$, auxquelles on ajoute à la suite les assertions générées avec $[\![\cdot]\!]_{\gamma}^{assert}$. Ces assertions vérifient que les traqueurs de base, qui ont exactement le type τ_{ts}, représentent le Typestate initial y_{ts_1} en fin de bloc.

D.3 Instrumentation des fonctions

$$\frac{\gamma = [x_i \mapsto (\textbf{enum } x_{ts}\texttt{*}, \texttt{ts}(x_i)) \mid \tau_i = \tau_{ts}\texttt{*}]_{i=1}^{n} \qquad \overline{d}_{tr} = (\gamma(x_i))_{i=1}^{n}}{[\![(\tau_1 x_1)_{i=1}^{n}]\!]^{params} = ((\tau_i x_i)_{i=1}^{n} +\!\!+ \overline{d}_{tr}), \gamma}$$

$$[\![tr]\!]_{x_f}^{trans} = [(pre, x_g, post) \mid (pre, x_g, post) \in tr \wedge x_f = x_g]$$

$$\frac{[\![\overline{d}]\!]^{params} = (\overline{d}', \gamma) \qquad [\![tr_{ts}]\!]_{x_f}^{trans} = [\,]}{[\![(x_f, \overline{d}, b)]\!]^{fun} = (x_f, \overline{d}', (\top, [\,]), [\![b]\!]_{\gamma}^{cli_block})}$$

$$\frac{[\![\overline{d}]\!]^{params} = (\overline{d}', \gamma) \qquad [\![tr_{ts}]\!]_{x_f}^{trans} = tr \neq [\,]}{[\![(x_f, \overline{d}, b)]\!]^{fun} = (x_f, \overline{d}', [\![tr]\!]_{\gamma}^{contract}, [\![b]\!]_{tr,\gamma}^{api_block})}$$

FIGURE 32 – Instrumentation d'une définition de fonction

On présente l'instrumentation d'une définition de fonction dans la Figure 32. On a d'abord la fonction $[\![\cdot]\!]_{\gamma}^{params}$, qui prend une liste de déclarations et y ajoute la liste de déclaration de traqueurs correspondante. Cette fonction donne aussi en second résultat la table des traqueurs γ mise à jour avec les associations entre les noms d'origine et les déclarations de traqueurs qu'on vient de générer. Contrairement aux déclarations locales, on ne génère une déclaration de traqueur que lorsque le type de la déclaration d'origine est exactement $\tau_{ts}\texttt{*}$, c'est-à-dire uniquement les pointeurs d'un seul niveau vers le type des Typestates et non plus d'un niveau arbitraire.

Ensuite, la fonction $[\![\cdot]\!]_{x_f}^{trans}$ nous permet de filtrer la liste des transitions afin de ne garder que les transitions sur un certain nom de fonction x_f.

Finalement, la fonction $[\![\cdot]\!]^{fun}$ va instrumenter la définition d'une fonction. Dans tous les cas on va étendre la liste des paramètres formels avec les traqueurs correspondants grâce à $[\![\cdot]\!]^{params}$.

S'il n'y a aucune transition associée au nom de la fonction, alors la fonction fait partie du code-client. On instrumente donc son bloc avec $[\![\cdot]\!]^{cli_block}$, et le contrat sera uniquement composé d'une précondition qui est toujours vraie.

Au contraire, s'il y a des transitions associées au nom de la fonction, alors cette fonction fait partie de l'API. On génère donc le contrat ACSL avec $[\![\cdot]\!]^{contract}$, et on instrumente le bloc avec $[\![\cdot]\!]^{api_block}$.