Rapport de stage - TN09

Gestionnaire mémoire dédié à la vérification à l'exécution de programmes C

Automne 2020, du 2020-09-01 au 2021-02-12

Stagiaire :
Auguste Roma [1]

Maître de stage :
Signoles Julien [2]

Suiveur UTC :
Bonnet Stéphane [3]

1. <roma@maug.fr> cycle génie Informatique, Sorbonne Universités, université de technologie de Compiègne, France
2. <julien.signoles@cea.fr> CEA, LIST, Software Security and Reliability Laboratory, Palaiseau, France
3. <stephane.bonnet@utc.fr> Sorbonne Universités, université de technologie de Compiègne, CNRS, Heudiasyc UMR 7253, Compiègne, France

Remerciements

Solitairement, je n'aurais pas pu réaliser ce stage, écrire ce rapport, concevoir *meal* ni acquérir tant de connaissances et de compétences en génie informatique.

Je tiens à remercier mon maître de stage Julien SIGNOLES, qui m'a beaucoup appris. Il a aménagé un stage qui me correspond et chaque contact avec lui a été enrichissant pour moi. Basile DESLOGES et Dara LY ont, eux aussi, eu la patience d'écouter mes balbutiements, et m'ont apporté de précieuses critiques. Nos échanges réguliers ont contribué au développement d'*E-ACSL* autant qu'au mien. Je remercie mon suiveur Stéphane BONNET pour sa confiance, pour les échanges que nous avons eus à propos de mon orientation et pour la qualité de ses enseignements, qui ont rendu ce stage possible. Ces quatre personnes ont participé à la construction d'un ingénieur, œuvre de longue haleine. Je leur en suis reconnaissant.

Merci aux relecteurs de ce rapport, Julien SIGNOLES et Thibaud ANTIGNAC.

Merci à tous les contributeurs actuels et passés de Frama-C et d'*E-ACSL*, en particulier aux personnes ayant écrit la documentation et les tutoriels. La qualité du code et de la documentation m'a permis de monter en compétence rapidement. Programmer Frama-C, à mon petit rythme, fût un réel plaisir.

Ce stage aurait été très différent sans les qualité humaines de mes collègues. Je profite de cet espace personnel pour remercier très chaleureusement Aymeric, Dario, Tibo, Basile, les Julien, Dara, Gabriel, Yaëlle, Zak, Dongho, Grégoire, Augustin, André, Allan, Adrien, Guillaume, Lesly-Ann, Maxime, Florent, Myriam, Olivier, Yackolley et David, pour leur gentillesse, leur passion, leur spontanéité, leur humour, leur courage, leurs singularités, leur sensibilité, et les quelques particules subatomiques d'amitié que nous avons pu échanger au cours de nos interactions, malgré leur brièveté et leur rareté, malgré mon introvertion et malgré la crise que nous vivons. Et j'en oublie, parmi les quarante autres personnes que j'ai pu croiser au LSL et dans son voisinage. J'admire la capacité du LSL à accueillir ses « non-permanents », et je regrette de ne pas avoir donné autant que j'ai reçu. Je dois également saluer les goûts culinaires et musicaux de certains, qui rendent les espaces réels et virtuels du LSL plus agréables #MétalChouquettes.
Ils et elles ne liront pas ce rapport, mais je dois aussi écrire ici que ce stage n'aurait pas été possible sans Corentin, Rémy, les Louis, Rindra, Diego, les Mathilde, Philippe, Léo, Léa, Rachel, Maya, Mathieu, César, Stéphane, Maël, Andrés, Quentin, Ugo, Salomé, Anthony, Éléonore, Jad, Marion, Jo, Julia, Olivia et tant d'autres personnes ayant rendu cette période de ma vie plus belle et moins sombre.
Merci à mes parents et à ma sœur.
Toutes les personnes mentionnées dans ce paragraphe m'ont, d'une façon ou d'une autre, soutenu. Elles ont participé à la construction d'un être humain, œuvre de longue haleine. Je leur en suis extrêmement reconnaissant.

Sommaire

Résumé technique

Le langage ACSL permet d'annoter un programme C avec des propriétés logiques spécifiant son comportement attendu.

E-ACSL, greffon de la plateforme d'analyse Frama-C [4], compile le programme C et ses annotations en un nouveau programme C, dont la correction vis-à-vis des propriétés est vérifiée dynamiquement (c'est-à-dire pendant son exécution).

Au cours de ce stage, j'ai commencé une démarche d'optimisation du code généré par *E-ACSL*, notamment en ce qui concerne la mémoire. Pour ce faire, j'ai étudié le comportement d'*E-ACSL* pour déterminer des points d'optimisation possibles, écrit une bibliothèque C définissant un gestionnaire mémoire, puis j'ai adapté *E-ACSL* pour qu'il génère des appels à cette bibliothèque.

Ce document rend compte de la réalisation de ma mission. Il présente des recherches, techniques et procédés que j'ai utilisés ou développés, mais aussi l'organisation et la portée de mes travaux, et ainsi que ceux du LSL et du CEA LIST en général.

1. Structure et organisation du travail

Mon stage s'est déroulé au centre de Saclay du Commissariat à l'Énergie Atomique et aux Énergies Alternatives, ou CEA. Plus précisément, j'ai travaillé dans le Laboratoire de Sûreté et de Sécurité des Logiciels (LSL), situé à Palaiseau, qui fait partie du Département Ingénierie des Logiciels et Sûreté (DILS) de l'institut CEA List. Le List (Laboratoire pour l'Intégration des Systèmes et de la Technologie) est l'un des trois instituts de la Direction de la Recherche Technologique (DRT) du CEA ; cette direction constitue l'une des trois branches civiles du CEA. L'organisation du CEA sera développée dans la suite de cette section.

1.1 Structure de l'employeur

1.1.1 Organisation générale du CEA

Le Commissariat à l'Énergie Atomique et aux Énergies Alternatives, ou CEA, est une administration centrale dont la mission fût longtemps de développer les applications de l'énergie nucléaire dans les domaines scientifique, industriel, et de la défense nationale. Le spectre des domaines de recherches du CEA est aujourd'hui plus large et comprend notamment les nouvelles technologies de l'énergie, la physique, la chimie et la biologie, l'électronique et l'informatique. Les recherches peuvent être fondamentales ou appliquées. Cela répond à une demande historique de l'État de développer et maîtriser le nucléaire et ses externalités. Ainsi, le CEA est sous la tutelle conjointe des ministères de la Recherche et de l'Enseignement Supérieur, de l'Énergie, de l'Industrie et de la Défense. Il compte environ 16000 employés permanents, 1400 doctorants et post-doctorants, 1300 alternants, et 1000 autres CDD, soit environ 20000 collaborateurs.

Le CEA est implanté en 10 centres civils ou militaires localisés en France (voir figure 1.1). Mon stage a été effectué dans le plus important d'entre eux, celui de Saclay, dans lequel environ 7000 employés travaillent. Les centres civils (Saclay, Grenoble, Marcoule...) regroupent des instituts de recherche du CEA, et des entités extérieures au CEA, pour des raisons historiques [1], de mutualisation des moyens [2], ou d'incitation aux partenariats avec le secteur privé (startups, recherche privée...).

Chaque institut du CEA mène des recherches civiles dans un domaine donné, et est dirigé par l'une des trois branches opérationnelles civiles de la direction du CEA : direction des énergies, Direction de la Recherche Fondamentale, ou Direction de la Recherche Technologique.

Certaines entités du CEA ne suivent pas cette organisation en instituts et directions

1. par exemple : l'IRSN est devenu indépendant du CEA, mais est toujours dans ses locaux
2. par exemple : au plateau de Saclay, l'infrastructure réseau, des bibliothèques, et certains locaux sont mutualisés avec des entités publiques et privées tierces

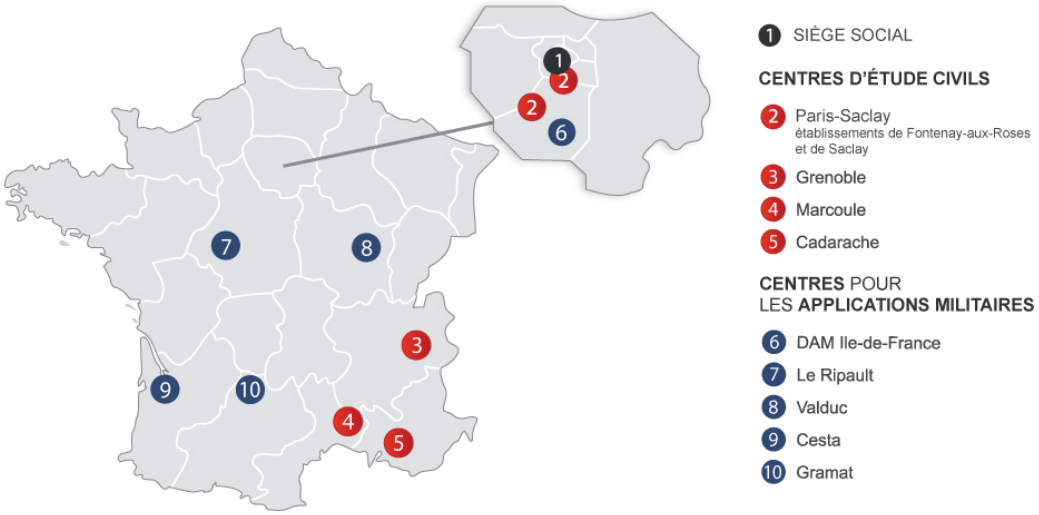


FIGURE 1.1 – Carte des centres de recherche du CEA

opérationnelles, par exemple l'Institut national des sciences et techniques nucléaires (INSTN), ou l'Agence ITER France (AIF). Le CEA mène aussi des travaux de génie civil : historiquement, la construction de centres de recherche nucléaires, et aujourd'hui leur démantèlement et leur assainissement, sont des tâches extrêmement spécifiques qui ne peuvent être externalisées.

Le CEA dispose également de directions fonctionnelles, qui gèrent les aspects centraux et non-opérationnels de son activité : informatique, ressources humaines, sécurité et sûreté, *etc.* Par ailleurs, les directions de centre gèrent ces aspects non-opérationnels à l'échelle locale. Cette autorité locale est complémentaire à celle des directions opérationnelles. Par exemple, les aspects liés à la crise sanitaire CoViD-19 ont été gérés par les directions de centre.

Pour résumer : les instituts de recherche correspondent à des groupes d'ingénieurs et de chercheurs d'un même domaine, et sont sous l'autorité d'une direction opérationnelle. Les centres de recherche sont des implantations géographiques ; chaque centre dispose d'une direction locale. Un centre accueille plusieurs instituts ; un institut peut mener ses travaux sur plusieurs centres.

Les centres militaires ont une organisation hiérarchique plus traditionnelle : la direction de centre est sous l'autorité de la DAM (Direction des Applications Militaires). Il n'y a pas de double gouvernance par instituts et centres.

1.1.2 Gouvernance du CEA

La direction du CEA est assurée par l'Administrateur Général, représentant de l'État, qui délègue son pouvoir aux autre membres de la direction générale. Le CEA reste sous la tutelle de l'État : certaines décisions ne peuvent être prises sans l'accord des ministres. La puissance publique, notamment le ministère en charge de l'Énergie et le gouvernement en général, mais aussi les collectivités territoriales, peuvent orienter les décisions stratégiques du CEA. Par le vote des lois de budget et de programmation de la recherche civile ou militaire, le pouvoir législatif aussi oriente la répartition des moyens sur les sujets de recherche. Réciproquement, le CEA a une mission d'information auprès des puissances publiques, et est notamment inscrit en tant que représentant d'intérêts au Parlement.

Par ailleurs, des instances d'évaluation scientifiques indépendantes peuvent également

participer à la prise de décision : ainsi, le Conseil scientifique du CEA, le *Visiting Committee* du CEA et le Haut Conseil de l'Évaluation de la Recherche et de l'Enseignement Supérieur (HCERES) évaluent l'activité du CEA. Afin de garantir leur indépendance, ces instances sont constitutionnellement placées sous l'autorité du Haut-Commissaire à l'énergie atomique qui en assure la synthèse et la restitution auprès des autorités de tutelles, du Conseil d'administration et des instances représentatives internes au CEA. Le haut-commissaire à l'Énergie atomique n'est pas membre du CEA, mais il est nommé en Conseil des ministres. Il a pour rôle de conseiller l'administrateur général du CEA ainsi que le président de la République et les membres du gouvernement français sur les questions relatives au nucléaire et au CEA. Soustrait à toute tutelle hiérarchique au sein du CEA, il fournit un regard extérieur, exclusivement sur les aspects scientifiques. Le premier haut-commissaire a été Frédéric Joliot-Curie.

Au niveau national et au niveau de chaque centre existent des instances de représentation du personnel et des organisations syndicales, prévues par la loi et la convention de travail du CEA. Elles participent à l'organisation du travail au CEA, et leur accord peut être nécessaire pour prendre certaines décisions.

1.1.3 Missions du CEA

Les missions du CEA sont opérées par ses quatre directions opérationnelles.

La Direction des Applications Militaires (DAM) est en charge, pour l'État, d'une mission de défense et de sûreté nationale. Concrètement, il s'agit des applications militaires de la physique nucléaire, notamment les armes nucléaires, les réacteurs nucléaires navals, et la lutte contre la prolifération nucléaire.

Une branche civile du CEA, la Direction de la Recherche Fondamentale (DRF) est dédiée à la recherche fondamentale, essentiellement en sciences de la matière et en sciences du vivant, mais aussi en physique, en mathématiques et en informatique. Des activités de recherche fondamentale existent également au sein des autres branches du CEA, bien que ces activités ne correspondent pas directement aux missions de ces branches.

La Direction de la Recherche Technologique (DRT) et la direction des énergies sont les deux autres branches civiles du CEA. Elles regroupent des recherches dans les domaines des énergies renouvelables et nucléaires, et des recherches plus technologiques, dans le domaine de l'énergie, mais aussi de l'information, la communication, de l'énergie et de la santé. La mission actuelle de la DRT est principalement le soutien à l'industrie.

26% des publications du CEA civil portent sur les énergies renouvelables et le nucléaire ; 32% des publications portent sur des technologies pour l'industrie ; 43% d'entre elles sont de la recherche fondamentale. [3]

1.1.4 Partenaires et moyens du CEA

L'ensemble des ministères de tutelle du CEA lui apportent un budget fixe, prévu par des lois et décrets.

Par ailleurs, des projets des collectivités locales, nationales, européennes ou internationales [4], peuvent lui apporter d'autres moyens. Les partenaires peuvent également être privés. Ces

3. 1% des publications correspondent à plus d'une mission.

4. par exemple, le projet ITER rassemble l'U.E. ainsi que le Royaume-Uni, l'Inde, le Japon, la Chine, la Russie, la Corée du Sud, les États-Unis et la Suisse.

moyens peuvent être d'envergures variées, tant en financement qu'en durée : le financement d'une thèse, et la consolidation de la cybersécurité en Europe, sont séparés de plusieurs ordres de grandeur. En 2019, le CEA est le troisième bénéficiaire du programme européen « horizon 2020 », et a un taux de succès de 24%. 67% des publications du CEA sont des co-publications internationales ; 75% d'entre elles sont européennes.

Au total, le CEA a un budget de 5 milliards d'euros, dont 2,7 dédiés aux activités de défense et d'assainissement. Pour les activités civiles de recherche fondamentale et de recherche en énergie, les recettes externes sont comprises entre 20% et 30%. En revanche, l'activité de recherche technologique (c'est-à-dire, de soutien à l'industrie) est financée à 73% par des tiers. Le CEA a plus de 700 partenaires industriels, dont 350 financent les recherches à hauteur de 50000 euros par an.

Il est courant que des travaux de recherche mobilisent des entités tierces, par des modalités variées. Ainsi, des projets de recherche publics peuvent mobiliser plusieurs laboratoires ; l'encadrement, le financement, ou la réalisation d'une thèse peuvent avoir lieu dans plusieurs entités publiques ou privées ; des partenariats bilatéraux peuvent signés avec le secteur privé ; et les instituts et centres peuvent signer des conventions avec des partenaires académiques (universités et autres centres de recherche), ce qui permet notamment la création d'unités mixtes de recherche.

Ainsi, le CEA possède, a accès, et met parfois à disposition de ses partenaires académiques, des (très grandes) infrastructures de recherche. Le CEA dispose ainsi de réseaux informatiques, de centres de calcul, et d'instrumentations spécifiques aux recherches menées : sciences informatiques, mathématiques, astrophysique, physique nucléaire, énergie, sciences de la matière, sciences de la terre, biologie. Certaines parties de ces infrastructures sont mutualisées à l'échelle nationale ou européenne ; d'autres sont réservées au CEA.

1.2 La DRT et le List

Le CEA List est l'institut de la DRT où j'ai fait mon stage.

La DRT (Direction de la Recherche Technologique du CEA, également connue sous le nom de CEA Tech) est présente surtout à Grenoble, et représente environ 5000 personnes. Sa mission actuelle est la recherche technologique, en soutien à l'industrie, c'est-à-dire accompagnée d'opérations de transferts de connaissances et de technologies. La DRT est spécialisée dans certains domaines, et est en effet composée de trois instituts : Leti (électronique et miniaturisation), Liten (énergie renouvelable, notamment solaire et batteries électriques), et List.

L'institut CEA List effectue de la recherche appliquée en informatique, et dans des domaines connexes, tels que la robotique. Il représente plus de 800 personnes, et est implanté sur quatre sites du plateau de Saclay.

Le List est lui-même divisé en plusieurs départements, dont le Département d'Ingénierie des Logiciels et Systèmes, qui représente environ 100 personnes, regroupées en laboratoires. Les laboratoires regroupent des ingénieurs et chercheurs travaillant sur les mêmes projets, mais n'appartenant pas nécessairement aux mêmes communautés scientifiques [5]. La structure des laboratoires n'est pas imperméable : il arrive que des projets mobilisent les compétences des chercheurs de plusieurs laboratoires.

Le LSL (Laboratoire de Sûreté et de Sécurité des Logiciels) calcule son propre budget. Au CEA, l'unité budgétaire de base est le département, mais en pratique, le DILS (Département

5. on peut trouver, par exemple, des chimistes et des informaticiens dans le même laboratoire

Ingénierie des Logiciels et Sûreté) demande à chacun de ses laboratoires d'être à l'équilibre financier (ce qui nécessite de calculer un budget propre). Les bugets des départements sont alimentés par le CEA et l'État via des dispositifs divers (lien de tutelle, institut Carnot, ANR, BPI...), mais aussi par des projets de recherche d'autres puissances publiques (UE, collectivités territoriales), et par des contrats bilatéraux avec de nombreux industriels. Les ingénieurs et chercheurs du LSL sont salariés du CEA, qui alloue leurs heures de travail à ces projets.

Par ailleurs, l'implantation géographique du CEA List en région parisienne permet au personnel d'intervenir dans les cours de nombreux établissements d'enseignement supérieur. Le CEA est d'ailleurs un « institut membre » de l'Université Paris-Saclay.

1.3 Travaux du LSL et Frama-C

Le LSL est le laboratoire où j'ai effectué mon stage. Frama-C, le logiciel auquel j'ai contribué au cours du stage, est développé et adossé à la recherche menée par le LSL.

Le Laboratoire de Sûreté et de Sécurité des Logiciels du CEA List, ou LSL, est spécialisé dans la mise en œuvre de méthodes formelles pour la sûreté et la sécurité des logiciels. C'est un laboratoire de taille relativement grande, composé d'environ 25 permanents et 20 non-permanents (doctorants, stagiaires, ingénieurs...), dont l'activité est le développement de logiciels d'analyse de code, dont les deux plus importants sont les plateformes libres [6] Frama-C et BINSEC [7]. Leur conception nécessite de la recherche, mais aussi de l'expertise et des efforts techniques, dans plusieurs domaines, notamment : méthodes formelles pour l'analyse de programmes, sécurité des logiciels et génie informatique. BINSEC analyse des codes binaires, et Frama-C analyse des codes sources. Mon stage s'inscrit dans le développement de Frama-C.

Le développement de Frama-C, BINSEC, et d'autres outils, répond aux besoins d'autres entités du CEA, d'autres acteurs du secteur du nucléaire comme l'IRSN ou EDF, mais aussi aux besoins d'autres acteurs académiques et industriels. L'analyse de programmes a en particulier des applications dans l'audit et la certification de systèmes critiques, mais elle concerne de nombreux domaines, puisqu'elle peut être appliquée à la sûreté et la sécurité de tous les logiciels.

Bien que le développement de Frama-C soit adossé à de la recherche, sa qualité est d'un niveau pré-industriel, puisque les versions officielles de Frama-C sont des produits distribués et pouvant être intégrés à des solutions opérationnelles chez des partenaires industriels. Le développement de Frama-C est astreint à un haut niveau de qualité, et doit être adapté aux besoins des partenaires, ce qui entraîne l'utilisation de techniques de génie logiciel, notamment la présence de tests et d'intégration continue, ou encore une architecture spécifique (en *noyau* et *greffons*) comme on le verra en section 2.4.

6. code source distribué en licence LGPL

2. Mission

2.1 Sujet du stage

2.1.1 Contexte technique

Frama-C est une plate-forme logicielle facilitant le développement d'outils d'analyses de programmes C [4]. Chaque programme C analysé par Frama-C peut être annoté par des propriétés logiques écrites dans un langage appelé ACSL. Frama-C offre alors différentes techniques de vérification pour garantir que le programme satisfait ces propriétés. Chaque technique de vérification est codée dans un greffon de Frama-C dédié.

Une de ces techniques a notamment pour but de traduire des propriétés ACSL en programmes C intégrés au programme analysé. Cette traduction, programmée dans Frama-C au sein d'un greffon appelé *E-ACSL*, permet d'obtenir un nouveau programme C dont la correction vis-à-vis des propriétés est vérifiée dynamiquement, pendant son exécution : par défaut, si une propriété est fausse à l'exécution, elle est reportée à l'utilisateur et l'exécution du programme est arrêtée. Cette technique est appelée la vérification à l'exécution, ou *runtime assertion checking*.

Le but du stage est d'améliorer les performances (en mémoire) d'*E-ACSL*.

2.1.2 Problématique et motivation du stage

Le code C généré par le greffon *E-ACSL* et intégré au sein du programme analysé utilise de la mémoire pour son usage propre, par exemple pour sauvegarder des résultats de calculs intermédiaires ou se souvenir de valeurs de certaines variables du programme avant qu'elles ne changent.

Moralement, *E-ACSL* peut être vu comme un compilateur, et l'instrumentation qu'il génère peut être vue comme un environnement d'exécution ou *runtime*, dans la mesure où l'ensemble des accès du programme à sa mémoire peuvent être surveillés. Cette instrumentation a pour but de calculer la valeur logique des propriétés entrées par l'utilisateur. À cette fin, elle peut avoir besoin de stocker des résultats intermédiaires, mais aussi de modéliser la mémoire du programme instrumenté, en enregistrant chaque accès à la mémoire, ou en retenant d'anciennes valeurs de variables. Les calculs et l'observation de la mémoire ont un coût, en espace et en temps, d'où la volonté d'optimiser le code généré.

L'amélioration des performances de Frama-C et de ses greffons rend l'utilisation d'outils de vérification formelle plus facile. En particulier, *E-ACSL* permet, dans certains cas, de déboguer plus rapidement des annotations ACSL.

Voici un exemple de construction non-optimale : l'ancienne version d'*E-ACSL* utilise

une variable de type `int`[1] pour stocker une valeur Booléenne (représentable sur un seul bit).

Des travaux récents menés au sein du LSL ont permis de montrer qu'il était possible de définir l'instrumentation du programme (c'est-à-dire la façon dont le code généré est intégré) d'une manière indépendante de la façon dont la mémoire dont elle a besoin est gérée [5]. C'est à ce dernier aspect que ce stage s'intéresse.

Ce stage a consisté en l'implémentation efficace d'une bibliothèque C définissant un gestionnaire mémoire prenant en compte les besoins et l'utilisation spécifiques d'*E-ACSL* afin d'optimiser le code qu'il génère.

2.1.3 Évolution du sujet

Le sujet était légèrement trop ambitieux, puisqu'une phase de test (banc d'essai) était prévue mais n'a pas pu être réalisée au cours du stage. Il était en effet prévu que plusieurs implémentations (partageant la même interface) soient réalisées et comparées. En revanche l'intégration dans la suite de tests d'*E-ACSL* a été commencée, ce qui a permis de détecter des erreurs dans le code produit au cours du stage, mais aussi dans le code préexistant.

Le cahier des charges comprenait une phase d'étude des optimisations possibles (qui a été faite de matière plus exhaustive que prévu), une phase de conception de l'interface de la bibliothèque, des implémenations de cette interface, et des bancs d'essais.

Au cours des deux premiers mois de stage, en parallèle de mon appropriation de Frama-C et d'*E-ACSL*, j'ai étudié le code généré par *E-ACSL* afin de proposer des optimisations. Cette étude nous a confortés dans certains choix techniques. Elle a également révélé un défaut qui n'était pas déjà connu.

Le sujet du stage ne prévoyait pas forcément une revue aussi exhaustive des points d'amélioration d'*E-ACSL*, ni l'implémentation dans *E-ACSL* du compilateur générant les appels à la bibliothèque C.

2.2 Planification

Au début du stage, nous avions déjà une vision de l'emploi du temps du stage.

— entre 1 et 2 mois : initiation à Frama-C et *E-ACSL*, appropriation des résultats théoriques auxquels est adossé le sujet du stage, travail de documentation autour des techniques à mettre en œuvre pour le gestionnaire mémoire
— environ 1 mois : premier audit d'*E-ACSL*, suggestion d'optimisations, validation de la pertinence et de la faisabilité des optimisations par l'équipe
— 1 mois : spécification et conception du nouveau schéma de compilation, en vue d'implémenter les optimisations retenues
— environ 2 mois : implémentation, vérification de la conformité du code produit au modèle théorique, et de la non-régression par rapport au code précédent
— quelques semaines : comparaison des différentes implémentations proposées entre elles, banc d'essai (environ 3 implémentations prévues)

Cet emploi du temps a été plutôt respecté. Cependant, il n'était pas prévu que je programme moi-même, en OCaml, la partie d'*E-ACSL* qui générerait les appels à la

1. la norme C99 garantit qu'une variable de type `int` est représentée sur au moins 2 octets. La plupart des compilateurs utilisent 4 octets, par exemple GCC sur les architectures IA-32 ou AMD64.

bibliothèque C. Cela a pris un temps supplémentaire. Contrairement à ce qui était prévu, une seule implémentation de la bibliothèque C a été réalisée, donc il n'y a pas eu de banc d'essai. Par ailleurs, les tests ne sont pas complètement écrits, par manque de temps, mais aussi parce que le nouveau compilateur ne couvre pas complètement ACSL, comme on le verra au chapitre suivant.

Cet emploi du temps n'était pas très détaillé au niveau de la conception du nouveau schéma de compilation, puis de son implémentation, puisqu'il n'était pas prévu que je les réalise seul. Ce fut la partie la plus technique, la plus prolifique, et la plus autonome.

2.3 Contributions

À mon arrivée, *E-ACSL* supportait un grand nombre des primitives d'ACSL [7]. Les efforts de recherche et de développement de l'équipe sont assez divers, et sont orientés par les besoins des différents partenaires industriels et académiques : ajout de fonctionnalités, corrections de bogues, optimisations, support et étude des fondements théoriques.

Comme on l'a vu plus haut, l'instrumentation étant coûteuse, son optimisation est un axe de travail important. À ce titre, une partie des travaux de Dara Ly [5], doctorant du LSL, ont permis de montrer la pertinence du sujet de mon stage, comme résumé dans l'annexe I.

Les travaux de Dara ont donc « ouvert la voie » : je n'ai réalisé aucune recherche sur le plan théorique. En revanche, sur le plan technique, même si mon équipe avait quelques intuitions et quelques attentes, le projet d'optimisation de la mémoire d'instrumentation n'avait pas commencé.

Mes contributions ont été :

- Une étude du code généré, qui a permis de suggérer certaines optimisations. Certaines d'entre elles n'étaient pas connues de l'équipe, et certaines d'entre elles ont pu être directement implémentées.
- L'écriture d'une interface de bibliothèque C appelée *meh*, fournissant des primitives d'accès et de calcul sur une pile d'objets [2] de taille arbitraire.
- La spécification d'un langage intermédiaire pour *E-ACSL* (assembleur) appelé *meal*, dont les primitives représentent des opérations élémentaires exécutées par *meh*.
- L'implémentation (unique) d'une bibliothèque *meh*.
- La spécification du schéma de compilation (traduction ACSL vers C en passant par *meal*).
- La réécriture (partielle) d'*E-ACSL* utilisant ces schémas de compilation.
- L'intégration (partielle) de mon code à la suite de tests d'*E-ACSL*.

J'ai présenté hebdomadairement mon travail, et ai bénéficié de conseils sur tous les plans, de la part de Julien Signoles, ingénieur-chercheur et maître de stage, Basile Desloges, ingénieur, et de Dara Ly, doctorant. Les deux idées les plus structurantes - la structure de pile utilisée dans *meh*, et l'utilisation d'un assembleur intermédiaire, *meal* - ne sont pas miennes, mais sont celles de mon encadrant. On peut m'attribuer la réalisation des contributions listées ci-desus.

Grâce à ces réalisations, j'ai montré que des techniques (structure de pile, assembleur intermédiaire) pouvaient être intégrées à *E-ACSL*. Pour le calcul de la valeur de vérité de certaines propriétés, nous sommes convaincus que ces techniques apportent un gain de

2. *objects* au sens de la norme C99

performance (en mémoire, et peut-être en temps dans certains cas), même si cela n'a pas encore été évalué expérimentalement. Dans cette mesure, mon travail est une preuve de concept.

Bien qu'une partie de mon code soit susceptible d'être intégrée à une version officielle d'*E-ACSL*, le développement n'est pas fini. Comme on le verra dans la suite de ce rapport, l'intégration avec l'ancien schéma de compilation n'est pas terminée. De plus, il faudrait supporter un plus grand sous-ensemble d'ACSL. Enfin, bien que mes contributions soient simples et lisibles, il reste encore des tests à faire avant de livrer mon code aux utilisateurs. De manière générale, l'équipe de Frama-C est astreinte à un haut niveau de qualité, dans la mesure où la plate-forme est libre [3] et utilisée pour vérifier des systèmes dits « critiques », dont la panne peut avoir des conséquences dramatiques pour les personnes, les biens ou l'environnement.

2.4 Outils, technologies, méthodes de travail pour Frama-C et *E-ACSL*

2.4.1 ACSL et *E-ACSL*

ACSL est un langage permettant d'annoter un code source C avec des propriétés logiques, qui devront ensuite être vérifiées ou prouvées par des outils d'analyse ou de compilation.

Par exemple, le code source suivant :

```
int x = 0;
//@ assert x != 1;
```

sera transformé par *E-ACSL* en ce code :

```
int x = 0;
int x = 0;
__e_acsl_assert(x != 1,"Assertion","main","x != 1","temp.c",3);
```

Si le test échoue, une erreur sera remontée à l'utilisateur.

De même, on peut écrire un contrat de fonction : si les pré-conditions du contrat sont respectées par l'appelant, alors on lui garantit que les post-conditions du contrat seront respectées par l'appelée.

```
/*@
requires x > 0;
ensures \result > 0;
*/
int inc(int x) {
  return x + 1;
}

int main() {
  return inc(-1);
}
```

3. code source distribué sous une licence permissive

Ce code est correct, mais sa vérification va échouer parce qu'on a spécifié que `inc` ne prenait que des valeurs positives en entrée. Un outil d'analyse statique pourrait détecter cette erreur. L'instrumentation d'*E-ACSL* le fera à l'exécution, en insérant entre l'appelant et l'appelée une fonction servant à vérifier la pré-condition et la post-condition :

```
int inc(int x)
{
    return x + 1;
}

int main(void)
{
    return __gen_e_acsl_inc(-1);
}

/*@ requires x > 0;
    ensures \result > 0; */
int __gen_e_acsl_inc(int x)
{
  int __retres;
  __e_acsl_assert(x > 0,"Precondition","inc","x > 0","temp.c",2);
  __retres = inc(x);
  __e_acsl_assert(__retres > 0,"Postcondition","inc","\\result > 0","temp
     .c",
                  3);
  return __retres;
}
```

2.4.2 Structures de données pour l'analyse (et la compilation)

On a vu que *E-ACSL* compilait du code C annoté en ACSL vers du code C.

Un compilateur analyse un code source, puis le synthétise en un code de niveau d'abstraction identique ou moindre. L'écriture d'un compilateur nécessite des structures de donnée particulières, et des outils appropriés pour les manipuler.

La structure de donnée centrale est *l'Arbre de Syntaxe Abstraite*, ou AST pour *Abstract Syntax Tree*. Il s'agit d'une représentation d'un programme, à partir de laquelle on peut générer du code, ou faire des analyses.

Un AST n'a pas nécessairement la forme que le programmeur a donné à son code : une phase préliminaire peut le transformer pour le simplifier ou rendre une analyse future plus facile (sans changer la sémantique du programme). Un exemple classique est l'omission des parenthèses de groupement puisque, dans un AST, le groupement des opérandes est explicité par la structure de l'arbre [9].

En plus d'un certain nombre de *greffons* dédiés à des analyses particulières, Frama-C fournit un *noyau*, qui génère un AST pour le langage C étendu par des annotations ACSL. L'AST est codé d'une façon standardisée, appelée CIL (*C Intermediate Language*) [6]. Alors qu'un AST est le plus souvent conçu pour optimiser une compilation, la forme de l'AST CIL a été conçue pour faciliter les analyses. Entre autres particularités, le noyau procède à des transformations qui normalisent le code, pour le rendre plus uniforme et plus simple à analyser. Par exemple, toutes les boucles sont codées avec la construction

`while`. Ainsi, les constructions de boucle plus riches telles que `for` sont converties en un algorithme équivalent et utilisant `while`. Comme la sémantique est préservée, l'AST reste compilable, ce qui rend possible l'existence d'*E-ACSL*.

Pour générer du code à partir de l'AST (ou l'analyser), on utilise des concepts issus de la programmation orientée objet (tels que les visiteurs), mais aussi des concepts issus de la programmation fonctionnelle (applications partielles, déstructuration, filtrage et reconnaissance de motifs...). Le langage OCaml fournit beaucoup de ces outils.

De plus, OCaml est un langage fortement typé. Cette propriété est souhaitable lorsqu'on manipule des structures complexes, en particulier pour vérifier et valider le code. En outre, le compilateur ocamlc procède à de nombreuses optimisations qui rendent le code généré assez performant, ce qui est souhaitable lorsqu'on analyse ou compile de grandes bases de code (industrielles).

Frama-C et ses greffons sont écrits en OCaml. Ainsi, Frama-C et OCaml fournissent au développeur d'un greffon :

— un environnement permettant aux greffons, modules et fonctions de s'échanger des données complexes en garantissant certaines propriétés de cohérence ;
— des structures de données, des primitives et des outils d'analyse élémentaires ;
— des greffons, vus comme des modules (au sens d'OCaml), fournissant des analyses.

Par sa forme et sa conception, la plate-forme Frama-C permet donc la cohabitation de plusieurs programmes très différents, et pas nécessairement couplés. Cela permet par exemple à des développeurs tiers d'écrire leurs propres greffons et leurs propres analyses. L'organisation du code en noyau et greffons, entraîne une organisation du travail analogue.

2.4.3 Organisation du travail

Ainsi, des équipes se constituent autour de sujets de recherche et/ou de développement précis. Souvent, chaque greffon est associé à une équipe. Certaines équipes peuvent travailler très indépendamment les unes des autres, parfois pour développer des analyses complémentaires.

Cela donne une base de code très grande [4], dont aucune personne ne peut avoir une vision d'ensemble. C'est pourquoi un groupe d'ingénieurs-chercheurs considérés comme experts et ayant des compétences complémentaires se réunit tous les quinze jours en « noyau », pour décider de l'organisation du travail autour du noyau de Frama-C, mais aussi des aspects dont plusieurs greffons dépendent.

Certains greffons sont couplés, mais la règle générale est le respect de l'interface avec le noyau. Cela rend chaque greffon indépendant. Ainsi, l'équipe de développement d'un greffon peut avoir des méthodes de travail propres. Cependant, certains éléments sont communs à toute l'équipe de développement de Frama-C :

— La forge logicielle (gitlab), qui centralise :
  — la gestion des versions (avec git),
  — le suivi des bogues, et
  — les relectures de code et les fusions de version.
  Cet aspect permet notamment de répartir et d'affecter rapidement certaines tâches comme la correction de bogues.
— Le cycle de distribution de Frama-C (une version majeure tous les 6 mois environ).
— Les outils d'analyse, de test et d'intégration continue du code de Frama-C :
  — la suite de tests, présentée ci-après,
  — le peluchage du code (*code linting*), et

4. $5,2 \times 10^3$ fichiers et $5,7 \times 10^5$ lignes de code

— les autres tests nécessaires avant livraison du produit (compatibilité avec l'environnement, présence d'une licence libre/propriétaire, numérotation des versions, cohérence des exemples de code dans les manuels...).

L'uniformisation de ces éléments n'est pas nécessaire au développement de chaque partie du code de Frama-C, mais elle permet d'améliorer la coopération entre tous ses développeurs, y compris d'éventuels contributeurs extérieurs. Elle facilite notamment l'ajout de code et la maintenance.

2.4.4 Test et validation

Tout ajout de code à la base de code principale (qui sera distribuée) doit être faite par une *demande de fusion* ou *merge request*. Cela entraîne systématiquement une relecture, et parfois des discussions techniques. Les versions du code n'étant pas validées par les différents tests automatiques ne peuvent être fusionnées à la version principale : la forge logicielle le refuse.

Certains tests, dits *de non-régression*, sont extrêmement stricts et nécessitent une lecture attentive de la part d'un humain. Ces tests prédisent la sortie du programme : s'ils sont contredits, cela implique souvent une modification du comportement du programme, qui doit être justifiée. L'humain étant faillible, certains changements de sortie peuvent néanmoins être validés par erreur, de temps en temps. Au cours de mon stage, lorsque j'ai commencé à regarder la suite de tests, j'ai d'ailleurs trouvé un bogue dormant par validation par un humain, c'est-à-dire que la sortie du programme révélait l'existence de ce bogue, mais elle avait été incorrectement validée, et ainsi le bogue a été « dissimulé ».

Ces tests sont exécutés avec un outil développé en interne, nommé ptests. Cet outil s'intègre dans l'environnement de développement et est paramétré par les développeurs de chaque greffon. Ainsi, pour de tester l'entièreté du code de Frama-C, il suffit de taper dans un terminal `make tests`. Au cours du développement d'un greffon nommé « plugin », on pourrait être amené à taper les commandes suivantes :

```
./bin/ptests.opt plugin
./bin/ptests.opt -update plugin
make tests
```

On commence par exécuter la suite de tests correspondant au greffon « plugin » afin d'afficher les tests dont la sortie diffère de l'oracle. Après avoir examiné ces différences, on valide les nouvelles sorties, c'est-à-dire qu'on remplace les oracles. Enfin, on exécute les tests de l'entièreté du code de Frama-C, afin de vérifier que nos modifications n'ont pas eu d'effets de bord non désirés [8].

3. Rechercher des optimisations du code généré par *E-ACSL* : une étude d'un compilateur en boîte noire

La première étape du stage, mon initiation à *E-ACSL*, a aussi été utile à mon équipe, puisqu'elle a consisté en une étude d'*E-ACSL*. Mon œil neuf, non biaisé, a permis de poser des questions pertinentes tout en apprenant vite. Il a notamment permis plusieurs optimisations possibles dont l'équipe n'avait pas déjà connaissance.

L'annexe II présente quelques-unes des améliorations que j'ai proposées à l'équipe. Certaines d'entre elles ont été retenues et implémentées immédiatement ; certaines ont nécessité un travail de fond qui a constitué le reste de mon stage ; et les autres n'étaient pas pertinentes, ou nécessitaient d'autres réalisations qui sortent du cadre du stage.

Par honnêteté intellectuelle, pour refléter ma progression et mon apprentissage, et pour aider le lecteur à comprendre certains choix techniques qui ont été faits dans la suite du stage, j'ai choisi de présenter des propositions qui n'étaient pas pertinentes ou qui n'ont pas été retenues.

Les deux constats les plus importants ont structuré la suite du stage et sont présentés ici.

3.1 Les opérateurs et types de C ne sont pas adaptés au calcul des prédicats

ACSL permet d'exprimer des propriétés logiques sous la forme de « prédicats » [1], dont les valeurs possibles sont « vrai » ou « faux » : ils sont isomorphes aux Booléens. Par conséquent, *E-ACSL* génère un code encodant beaucoup de valeurs Booléennes, et utilisants des opérateurs logiques (sur ces valeurs Booléennes) ; or les types et opérateurs de C sont inadaptés au stockage au calcul sur les Booléens.

Cela vient du fait que C a été conçu pour être extrêmement proche du fonctionnement physique des machines les plus répandues à l'époque de sa conception. Ces machines de la famille CISC [1], dont les processeurs x86 sont les héritiers, calculent sur des mots dont la taille est supérieure ou égale à un octet, c'est-à-dire qu'on ne peut pas adresser ou opérer sur des bits. De plus, les opérateurs de C correspondent (grossièrement) aux opérations fournies par les assembleurs les plus répandus, or ces opérations manipulent des données sur un ou plusieurs octets. C'est pour cela que, d'une façon ou d'une autre, on utilise des registres entiers pour calculer sur des Booléens.

1. *Complex Instruction Set Computer*, ou ordinateur à jeu d'instructions étendu

Il y a ainsi un compromis : si on veut utiliser les opérations fournies par le processeur (gain de temps), on doit utiliser ses types de donnée (perte de mémoire). Par exemple, si on veut utiliser l'opération ADD de l'assembleur x86, il faut présenter au processeur une donnée d'une taille supérieure ou égale à celle du plus petit registre : *al*, qui contient un octet, c'est-à-dire, en C, un `char`.

Cependant, depuis les années 1970, les machines sont devenues plus rapides. De plus, si les industriels sont prêts à instrumenter leur programme avec *E-ACSL*, un environnement d'exécution ajoutant un certain *overhead* (un « surcoût »), c'est que la vitesse d'exécution n'est pas toujours le facteur le plus important.

On peut choisir le compromis inverse : une perte de temps, pour un gain de mémoire. L'idée est d'utiliser non pas une, mais plusieurs opérations du processeur, pour manipuler des bits. En effet, certaines opérations (décalages, opérations bit-à-bit) permettent de programmer l'encodage de Booléens sur les registres du processeur, ainsi qu'un décodage, et des opérations sur les données ainsi encodées. On peut donc faire du calcul des prédicats. Au lieu d'utiliser les opérations logiques fournies nativement par le processeur, qui utilisent n registres comme opérandes, on utilise de petites fonctions C qui manipulent les n bits d'une variable de type et (donc) de taille connues.

La suite de ce rapport détaille la mise en oeuvre pratique de ce compromis, présente les résultats obtenus, et, pour quelques exemples bien choisis, montre une amélioration du code généré.

Par manque de temps, aucune mesure de performance (empirique) n'a été réalisée. Le postulat est qu'une multiplication par 3 ou 4 du nombre d'instructions au processeur est souhaitable si elle permet de diviser par 8 ou 16 la mémoire utilisée. Cela doit être nuancé par la complexification des accès à la mémoire. Cependant, la technique qui va être présentée permet justement de limiter les accès à la mémoire, dans une certaine mesure.

3.2 On peut générer du code non paresseux

ACSL utilise une logique ternaire, dans la mesure où les prédicats peuvent avoir trois valeurs « de vérité » : *true*, *false*, ou *undefined*. Ce troisième état est dû au lien entre ACSL et C : la valeur de certains prédicats est fonction d'expressions C, qui peuvent être *undefined*, ou dont l'évaluation peut être un *undefined behaviour* [3].

Nous avons fait la conjecture (par l'exemple, dans l'annexe II) qu'il est utile d'implémenter la *paresse* des opérateurs ACSL (tels que `||`, `&&`, `=>`...) garantie par la norme ACSL, seulement si la valeur de l'une des opérandes est susceptible d'être *undefined*. Plus précisément, si et seulement si :

— l'évaluation de l'une des opérandes peut amener le programme dans une branche où les autres opérandes ne sont pas évaluées, **ET**

— ces autres opérandes peuvent avoir, à l'exécution, la valeur *undefined*.

De plus, statiquement on peut savoir que certains prédicats ne peuvent pas avoir la valeur *undefined*. Il est par exemple évident qu'une constante ne peut pas être *undefined* ; de même, une expression constituée seulement de constantes logiques et d'opérateurs logiques ne peut être *undefined*.

Notre postulat est qu'il est possible de laisser *E-ACSL* décider s'il va effectivement implémenter la paresse. Dans la suite du stage, on développe un système de calcul qui ne fonctionne que pour des prédicats dont les opérandes ne peuvent pas être *undefined*. (Par manque de temps, on n'a pas encore recherché un algorithme qui détecte des prédicats

qui ne peuvent être *undefined*, et utilisé cette décision pour qu'*E-ACSL* n'utilise pas ce système de calcul lorsque la paresse est nécessaire).

4. Compiler différemment les annotations ACSL : structure de pile et langage intermédiaire

Comme on l'a vu, la version actuelle d'*E-ACSL* utilise les types, structures de contrôle et opérateurs de C pour faire du calcul des prédicats. Dans la nouvelle version d'*E-ACSL*, présentée dans ce rapport, les opérateurs sont programmés sous la forme de petites fonctions C qui manipulent des bits stockés dans une structure de données bien choisie.

Le choix de cette structure de données est crucial. Par définition, le choix de la structure de données détermine les algorithmes qui seront utilisés pour manipuler la mémoire, et donc les performances de l'instrumentation.

De plus, nous avons considéré qu'il n'était pas souhaitable d'introduire des niveaux d'abstraction entre la structure de données physique et le code appelant (généré par *E-ACSL*). Par conséquent, le choix de la structure de données détermine l'interface exposée au code appelant.

On peut aussi raisonner de façon opposée. On détermine la forme qu'on souhaite donner au code appelant, et on en déduit la forme idéale que la structure de données devrait avoir.

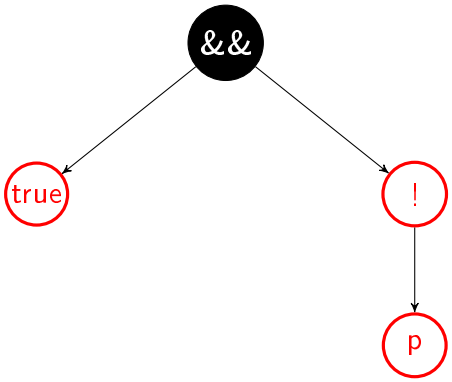
Comme illustré dans l'exemple de code 4.1, un prédicat ACSL est la racine d'un arbre constitué d'autres prédicats, et de termes [1]. La compilation d'un prédicat ACSL est donc similaire à la compilation d'une expression C : pour pouvoir compiler le prédicat père, il faut d'abord avoir compilé tous ses termes et prédicats enfants.

Code 4.1 – Un prédicat ACSL

```
!p && \true
```

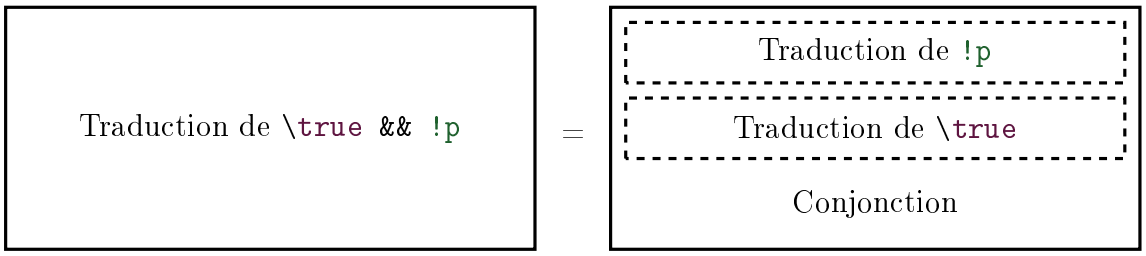
Ici, `!p && \true` est le prédicat racine, dont les deux fils sont le prédicat constant `\true` et le prédicat `!p`, qui a lui-même un enfant, le prédicat `p`. Cet arbre est représenté ci-dessous.

FIGURE 4.1 – Arbre abstrait correspondant au prédicat 4.1

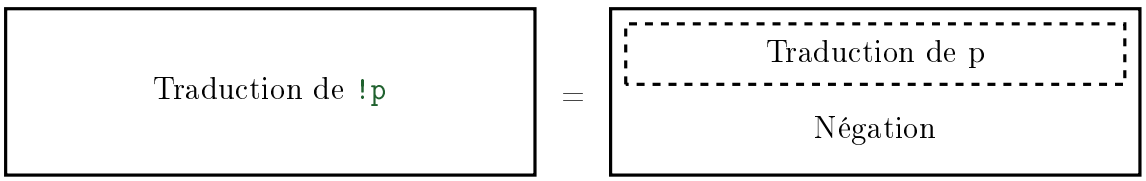


E-ACSL est un compilateur dont le langage de sortie est le C. On ne peut pas, ou on ne souhaite pas, utiliser certains opérateurs logiques natifs [1]. Le langage de destination est donc un sous-langage de C. On ne peut donc pas utiliser les expressions de C, et on ne peut donc pas exprimer d'arbre arithmétique. Il va donc falloir « aplatir » cet arbre, c'est-à-dire le transformer en un code composé d'une succession d'instructions élémentaires bien ordonnées, qui seront, au sens de C, de simples appels de fonction.

La compilation sera donc récursive : lorsqu'on commence à écrire le code traduisant un nœud, on suppose que la traduction de ses fils (c'est-à-dire de ses opérandes) a déjà été écrite. Cette hypothèse constitue un invariant de compilation.



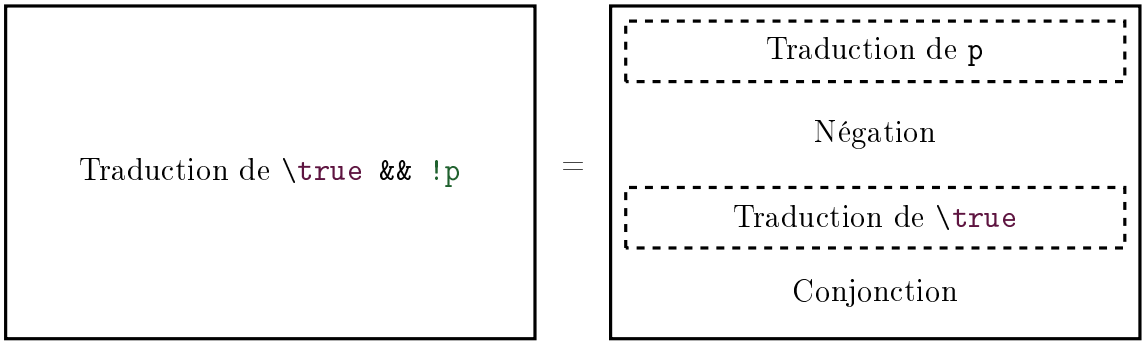
Ainsi, lorsqu'on compile le prédicat `\true && !p`, avant d'écrire le code qui va effectivement faire la conjonction, on doit d'abord traduire les prédicats `\true` et `!p`. On entre donc dans un niveau de récursion inférieur.



Une fois qu'on a traduit `!p`, on utilise ce résultat dans le niveau de récursion supérieur :

1. l'opérateur ACSL `&&` est paresseux : si l'évaluation de seconde opérande est un *undefined behaviour*, on peut éviter son évaluation en donnant une première opérande fausse. Par ailleurs, les opérateurs de C ont une règle de promotion des types entiers qui gâche de la mémoire (bien qu'elle soit utile pour assurer la correction de certaines expressions de façon intuitive).

la traduction de `\true && !p`.



Dans ce schéma de compilation, **la donnée de chaque opération est le résultat de l'opération précédente.** Le langage de destination est un langage séquentiel, dont les primitives sont des opérations élémentaires. Dans la suite de ce rapport, on qualifiera un tel langage d'*assembleur*.

De plus, nous souhaitons que notre bibliothèque gère elle-même la mémoire. Les procédures exposées par la bibliothèque C, et par conséquent les primitives de l'assembleur, ne doivent pas permettre de contrôler l'occupation de la mémoire physique de la machine. C'est une différence majeure avec les assembleurs des processeurs de micro-ordinateurs (x86 et ARM), qui permettent de déplacer des données depuis et vers toutes les mémoires.

La compilation d'expression arithmétiques vers des assembleurs est un problème ancien. Ainsi, il existe des façons classiques de gérer la mémoire afin de passer le résultat d'une opération à l'opération suivante.

Comme notre assembleur doit permettre à son utilisateur [2] :
— de faire entrer et sortir des données [3],
— d'opérer sur ces données, de façon atomique,
— d'utiliser implicitement le(s) résultat(s) d'opération(s) en tant que donnée(s) de l'opération suivante,
— mais pas de contrôler l'encodage ni la position en mémoire des données,
le choix d'une structure de pile est adapté.

En effet, la structure de pile permettra à l'utilisateur de faire entrer et sortir des données en les empilant et en les dépilant, mais il ne contrôlera par leur stockage physique. En revanche, on pourra lui garantir, pour chaque opération o_i, que les opérandes sont les $n_i \in \mathbb{N}^*$ premières données de la pile, et que le(s) résultats sont les $m_i \in \mathbb{N}$ premières données de la pile.

On peut remarquer que l'algorithme schématisé précédemment est incomplet. En effet, avant d'injecter la traduction de `!p` dans la traduction de `\true && !p`, il aurait fallu d'abord traduire `p`. De même, la traduction de `\true && !p` sera achevée seulement quand `\true` sera traduit.

La traduction de `\true` consistera en l'appel de la primitive qui empile la valeur de vérité « vrai ». Concrètement, le code C généré sera : `eacsl_meh_push_BOOL(true)`. En revanche, la traduction de `p` est non nécessairement triviale. S'il s'agit d'une expression C de type `bool`, on peut l'empiler (avec `eacsl_meh_push_BOOL(p)`). Mais si `p` est logique [4]

2. la personne ou le programme qui écrit un programme utilisant les primitives de notre assembleur
3. en entrée : les feuilles des prédicats ; en sortie : les valeurs de vérité des prédicats
4. un prédicat construit avec ACSL qui ne n'existe pas en tant qu'expression C dans le code source

(ou correspond à une expression C d'un type autre que `bool`), la compilation n'est pas terminée, et nécessite encore au moins une descente récursive.

Jusqu'ici, dans ce rapport, les primitives du langage assembleur (qui représentent des opérations à faire sur les données contenues dans la pile) n'étaient pas distinguées du code C généré (constitué d'appels aux fonctions exposées par la bibliothèque C qui implémentent ces opérations). Dans la version actuelle du projet, à chaque primitive du langage assembleur correspond une unique fonction de la bibliothèque C, et à chaque fonction exposée par la bibliothèque C correspond une unique primitive du langage assembleur [5]. Mais il est tout de même utile de distinguer le code assembleur du code C, pour plusieurs raisons :

— il est plus facile de réaliser certaines optimisations [6] sur l'assembleur que sur le code C

— l'édition directe d'un fichier C (qui a une syntaxe plus riche que le langage assembleur) est plus permissive (elle permet d'introduire plus de bogues) que l'édition d'une structure de données prévue exactement pour le langage assembleur qu'on définit. [7]

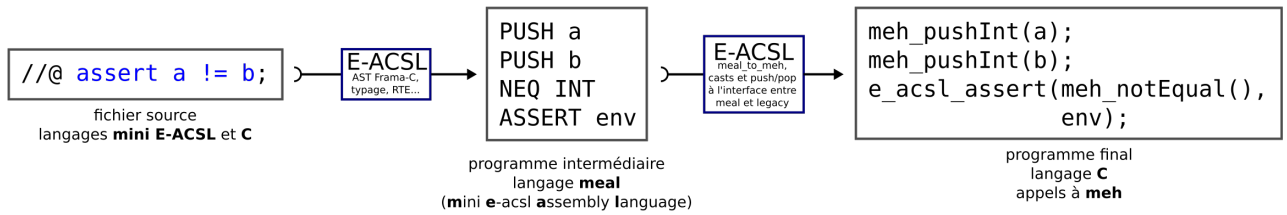
Par ailleurs, on pourrait vouloir introduire des primitives du langage assembleur qui génèrent un code C plus complexe qu'un simple appel de fonction (par exemple, les opérateurs paresseux ont besoin d'un branchement, voir l'annexe II).

Il est donc utile de séparer la génération du code assembleur de la génération du code C. Dans la suite du rapport, les termes suivants seront utilisés :

— *meal* est le langage assembleur dans lequel les termes et prédicats sont traduits en une suite d'opérations sur une pile (abstraite)

— *meh* est une bibliothèque C exposant des fonctions correspondant à des opérations sur une pile (cachée à l'utilisateur)

FIGURE 4.2 – Compilation par un langage intermédiaire



annoté

5. si on fait abstraction des types et éventuels arguments des opérations

6. par exemple détecter un empilement et un dépilement successifs de la même donnée

7. On aurait aussi pu éditer directement un fichier C, puis vérifier si sa syntaxe correspond à celle du langage assembleur

5. Spécification de l'assembleur *meal* et des structures en mémoire physique de *meh*

5.1 La spécification de *meal*

À partir du tableau de l'annexe II.2.1, on définit un sous-ensemble d'ACSL avec la sémantique d'*E-ACSL* qui ne contient que des termes et prédicats *simples*. On l'appelera *mini E-ACSL*.

À chaque opérateur de *mini E-ACSL* on associe un opérateur de *meal*. À l'exception des opérateurs d'empilement, de dépilement et d'assertion, tous les opérateurs de *meal* ne prennent aucun argument et ne retournent aucun résultat. En effet, il est implicite que ces opérateurs manipulent les données de la pile.

Cependant, ces opérateurs doivent aussi interpréter la pile : si on demande l'addition de deux `char`, *meh* va dépiler 16 bits, interpréter les 8 premiers comme un nombre, et les 8 suivants comme un autre nombre. On voit bien qu'il ne s'agit pas de la même opération que l'addition de deux `int`, par exemple. Au final, toutes les opérations arithmétiques et logiques doivent donc être typées. Le type constitue, en quelque sorte, un argument.

meal a été spécifié dans un premier temps à l'aide d'un document texte, qu'on peut trouver en annexe III.

Ensuite, *meal* a été modélisé en OCaml, par un fichier qu'on peut trouver en annexe III.10. Il peut y avoir quelques différences entre la spécification et sa modélisation. Les types permettent d'ailleurs d'interdire *à la compilation d'E-ACSL* certaines opérations (par exemple opérer un décalage de bits sur un nombre réel).

5.2 Structure physique de la pile

Une fois l'assembleur spécifié, on peut déterminer la façon dont la pile est représentée en mémoire, et modifiée par les opérateurs.

Tous les termes supportés actuellement par *meh* doivent correspondre à des types entiers de C (on ne supporte par les GMP, nombres de précision arbitraire). En outre, on a choisi de gérer les prédicats en les encodants sur des bits physiques (sans utiliser des types de C).

On a trois possibilités :

— créer une pile où cohabitent des entiers C (correspondant aux termes) et des structures de donnée spéciales (dédiées au stockage des bits correspondant aux

prédicats),
— créer deux piles, l'une pour les termes, l'autre pour les prédicats, ou
— créer une pile par type.

Dans tous les cas, *meh* devra travailler sur plusieurs espaces interprétés différemment. Cependant, mélanger des bits et des objets C de taille plus grande (8, 16, 32, 64 bits...) conduit :

— soit à une occupation de l'espace non-optimale (utiliser 8 bits pour stocker moins de 8 prédicats), à cause de stratégies d'alignement,
— soit à des algorithmes complexes d'interprétation de la mémoire non-alignée.

La solution de la « cohabitation » des bits avec les types entiers ne semble donc pas être la meilleure, vu les contraintes de la gestion de la mémoire en C.

Par ailleurs, on voit, dans la spécification de l'assembleur, que la majorité des opérateurs sont dans deux classes : ceux qui prennent des prédicats en entrée et retournent un prédicat en sortie ; et ceux qui prennent des termes en entrée et retournent des termes en sortie. Cela nous incite à créer deux piles : l'une pour les termes, l'autre pour les prédicats. Il semble que la deuxième solution soit la plus adaptée, puisque choisir entre plusieurs piles à chaque opération peut être coûteux.

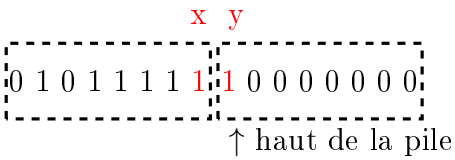
Ainsi, on choisit d'avoir une pile d'entiers C correspondant à des termes, et une pile de bits correspondant à des prédicats.

Ces mémoires sont allouées statiquement, par des tableaux de `char`, ce qui, d'après la norme C, fournit au programmeur C des segments de mémoire « continus », dans lesquels on peut stocker des données arbitraires (à condition qu'elles puissent être encodées comme une succession d'octets).

Dans le cas de la pile des termes, on utilise donc des *casts* de pointeurs pour interpréter les différents éléments stockés dans la pile. Le pointeur de pile est de type `char`, ce qui correspond au plus petit élément stockable ; pour lire un `short`, ce pointeur de pile sera ainsi *casted*.

Dans le cas de la pile des Booléens, on voit les `char` comme des « cellules de bits ». La variable d'état n'est pas un pointeur de pile, mais simplement une variable entière qui compte le nombre de bits stockés dans la pile. Cela permet, dans une certaine mesure, de faire abstraction de la segmentation de l'espace en cellules. En effet, les opérations à deux opérandes pourraient travailler sur la fin d'une cellule et le début de la suivante.

FIGURE 5.1 – Comment calculer `x && y` ?



Pour résoudre ce problème, on doit lire les cellules deux par deux. Pour éviter de faire deux accès mémoire (pour lire deux `char` !) on préfère changer de type, et interpréter la mémoire comme un tableau de double-cellules. Cette réinterprétation de l'espace entraîne un adressage différent (la double-cellule d'indice n est la concaténation de la cellule $2n$ et de la cellule $2n + 1$). Comme la variable d'état compte les bits (et non pas les cellules), ce problème de ré-adressage est résolu aisément par des opérations arithmétiques simples (modulo...).

6. *meh* : écriture des opérations en C

6.1 Opérations sur les termes

L'écriture des opérations sur des termes a été extrêmement simple. Les seuls termes supportés actuellement par *meh* étant ceux traduits par des types entiers de C, il suffit d'interpréter l'espace de la pile comme étant du type passé en argument, et d'utiliser un opérateur arithmétique fourni par C, pour effectuer une opération.

Toutefois, une difficulté n'a pas été surmontée au cours de ce stage : l'explosion combinatoire des types. En effet, l'addition d'un `int` avec un `char`, l'addition d'un `char` avec un `int`, et l'addition d'un `int` avec un `int` sont trois opérations différentes, puisque la mémoire est interprétée différemment. D'ailleurs, certaines opérations peuvent même prendre trois types en entrée (pour définir le type de sortie).

J'ai écrit ces opérations en C manuellement : je n'ai donc pas pu toutes les écrire au cours du stage.

Ainsi, tous les types ne sont pas supportés : actuellement, toutes les opérandes d'un opérateur doivent être du même type. Par conséquent, le code généré par ma version d'*E-ACSL* est non-optimal. *E-ACSL* dispose d'un système de type qui prend des décisions plus précises que les types que *meh* met à sa disposition.

Les pistes qu'on pourrait explorer sont :
- le développement de code C par macro du compilateur C (à la compilation du code C de *meh*)
- la génération de code C par *E-ACSL* (à la compilation du code C+ACSL instrumenté)
- la génération ponctuelle d'une version complète (mais difficillement maintenable) de *meh* par un programme écrit ad hoc.

Dans tous les cas, il faudra qu'*E-ACSL* génère au moins la taille des piles (qui sont allouées statiquement par des tableaux dont la taille est une constante de *meh*).

6.2 Opérations sur les prédicats

La manipulation de la pile des Booléens a été plus compliquée, car C ne fournit pas de type « bit », et par conséquent, pas de *cast* permettant de réinterpréter l'espace mémoire comme une suite de bits, ni d'opération sur des variables de type bit.

L'écriture des opérations sur les bits a été la seule de tout le stage à nécessiter (un peu) de recherche. En effet, j'ai choisi de trouver moi-même des algorithmes de calcul sur les Booléens. Cette phase de recherche aurait pu être évitée par une recherche documentaire, par exemple une lecture de [2].

6.3 Calcul du « ET » logique

Voici par exemple une technique qui permet de calculer un « ET » logique sur des bits contenus dans des types entiers de C, à l'aide d'opérateurs de C.

6.3.1 Premier algorithme

L'idée centrale est d'utiliser l'opérateurs de bits (*bitwise*) « OU » de C pour sélectionner par un masque les bits qui participeront à l'opération.

Voici deux exemples. Supposons qu'il existe un type entier de C de 6 bits, et que nous souhaitons calculer « ET » sur les bits 3 et 5 (le bit 0 est à l'extrême droite). Le masque de sélection des bits est sur le première ligne ; la donnée est sur la seconde ligne.

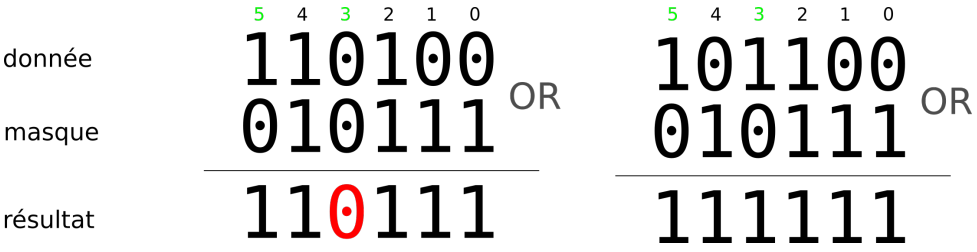


FIGURE 6.1 – « ET » logique sur un type entier, par masque de sélection des bits et opérateur de bits (*bitwise*) « OU »

Les bits 3 et 5 du masque sont à zéro, donc les bits 3 et 5 de la donnée participeront à l'opération. Les autres bits du masque sont à un, donc les autres bits de la donnée ne participeront pas à l'opération.

Sur l'exemple de droite, on a obtenu 111111 parce que tous les bits sélectionnés par le masque étaient à 1. Sur l'exemple de gauche, on n'a pas obtenu 111111 parce que le troisième bit de la donnée était à 0.

Il ne reste plus qu'à trouver une techniquement permettant de transformer 111111 en « vrai », et toutes les autres valeurs en « faux ». On peut par exemple additionner 1, et décaler à droite.

6.3.2 Algorithme utilisé dans *meh*

Cette technique est élégante, mais elle a été un peu modifiée avant d'être utilisée dans la version actuelle de *meh*. En effet,

- au lieu de générer dynamiquement des masques sélectionnant les deux bits participant à l'opération, on préfère décaler les bits de la donnée à droite
- puisqu'on a seulement deux bits de donnée, et qu'ils sont consécutifs, au lieu d'appliquer un masque « OU », il suffit d'additionner 1
- enfin, on décale le résultat à droite pour qu'il se trouve au sommet de la pile (et qu'il écrase les deux opérandes).

Cette seconde version de l'algorithme va être expliquée pas à pas ci-dessous. Les autres opérateurs logiques de *meh* ont été conçus de façon analogue.

Remarque : dans cette section, lorsqu'on parle de décalages de bits « à droite » ou « à gauche », on fait référence à l'écriture traditionnelle où le bit de poids fort est placé à gauche, et le bit de poids faible à droite. Un décalage de bits vers la gauche est donc une multiplication par deux, et un décalage de bits vers la droite est donc une division par deux. Mais les schémas présentés ci-dessous placent les bits de poids faible à gauche, pour que la pile puisse être lue de gauche à droite.

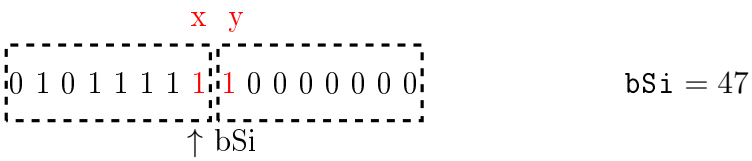
Extraire les deux bits opérandes

Voici une représentation de la pile : les deux rectangles en pointillé représente les deux cellules les plus hautes. Le rectangle de droite représente la cellule au sommet de la pile. `bSi` (pour *boolean Stack index*) est la variable comptant le nombre de bits stockés. En « additionnant » `bSi` à l'adresse de début de la pile, on obtient le prochain emplacement libre : c'est ce que représente la flèche.

On souhaite calculer un « ET » logique : cela implique que les deux bits de donnée sont au sommet de la pile. Ce sont donc les (`bSi`−2)-ième et (`bSi`−1)-ième bits de la pile. Ils sont notés x et y. Ils peuvent avoir été empilés par l'utilisateur ou être le résultat de calculs précédents.

FIGURE 6.2 – Calculer `x && y` avec *meh*

On commence par décrémenter le compteur de la pile des Booléens (*Boolean Stack Index*), afin d'adresser plus simplement nos deux bits de donnée : `bSi-=2;`

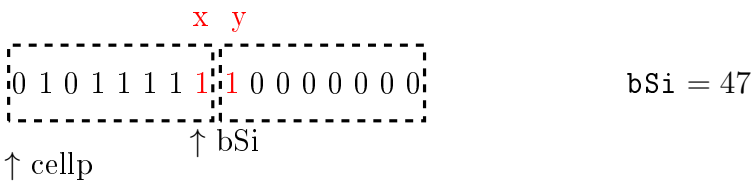


À présent, on calcule un pointeur vers la première cellule :

```
bS_doublecell* const cellp = (bS_doublecell*)(boolStack + (bSi /
   BS_CELL_BIT) );
```

`boolStack` pointe sur le début (le bas) de la pile, et `BS_CELL_BIT` est le nombre de bits contenus dans chaque cellule. Cette division (euclidienne) nous donne bien l'adresse de la cellule contenant le `bSi`-ième bit. Enfin, on convertit par un *cast* ce pointeur de cellule en un pointeur de « double-cellule ». Le type `bS_doublecell` permet de lire plus d'une cellule à la fois (au moins deux). [1]

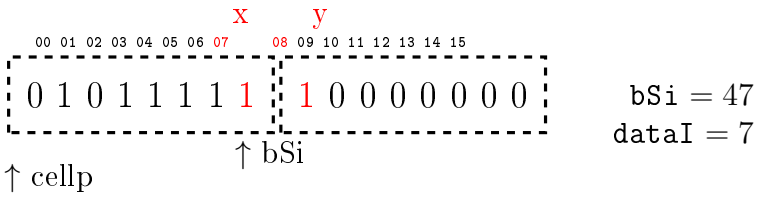
1. concrètement, pour GCC sur AMD64, on choisit `char` pour la cellule et `short` pour la double-cellule.



On calcule maintenant la position de nos deux bits de donnée à l'intérieur de la double-cellule :

```
const bS_bit_index dataI = bSi % BS_CELL_BIT;
```

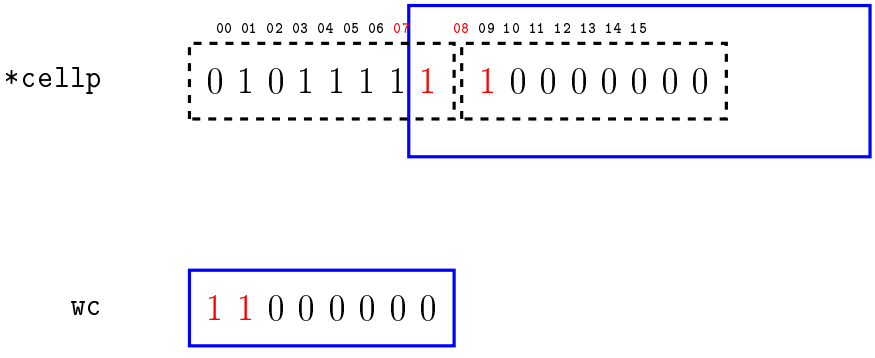
Le premier bit de donnée est le `dataI`-ième bit de la double-cellule ; le second bit de donnée est le (`dataI`+1)-ième bit de la cellule.



Avec un décalage de bits, on copie ces deux bits dans une variable :

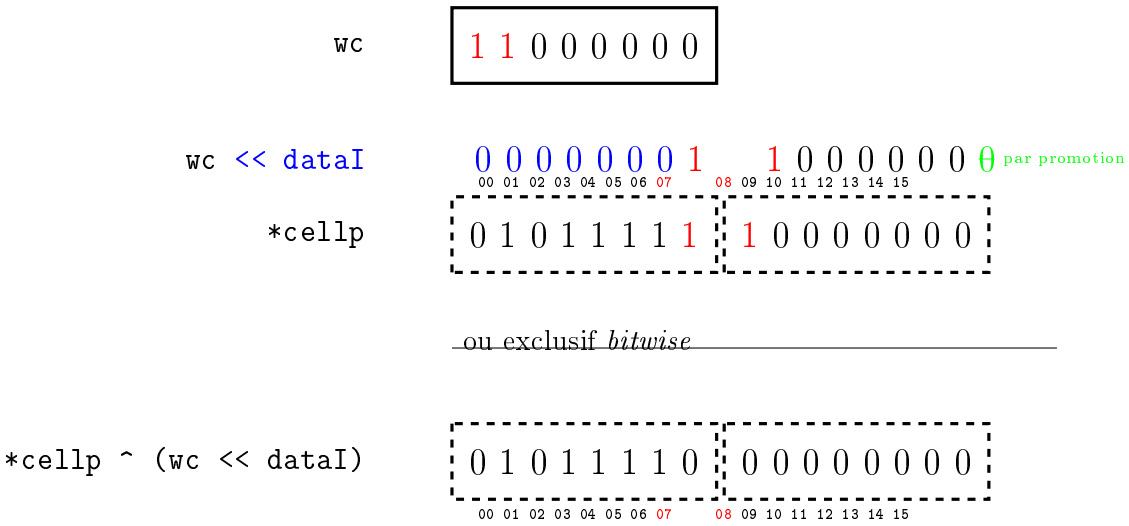
```
const char wc = *cellp >> dataI;
```

Décaler `dataI` fois à droite revient à supprimer les bits « à droite du `dataI`-ième bit ». Les bits « à gauche », c'est-à-dire au-dessus du sommet de la pile, sont tous nuls (par invariant de compilation). Si le décalage devrait créer des bits supplémentaires, il seraient également à zéro. Par conséquent, les deux premiers bits du `char wc` sont nos deux bits de donnée, et tous les autres bits de `wc` sont à zéro.



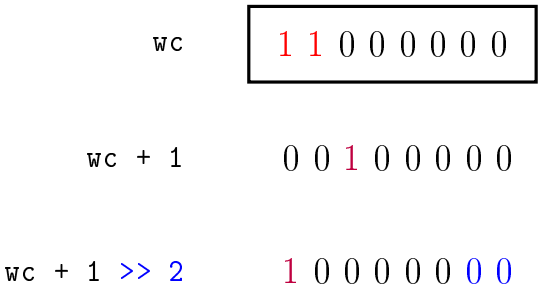
Effacer les opérandes de la pile

On décale `wc` à droite (c'est-à-dire qu'on lui ajoute des zéros), et on utilise l'opérateur de bits (*bitwise*) « OU » exclusif de C (`^`) pour remettre à zéro les bits de la pile qui contenaient la donnée : `*cellp = *cellp ^ (wc << dataI);`



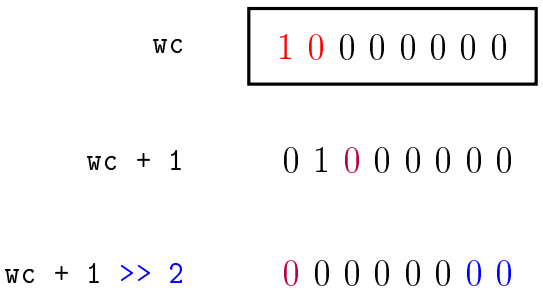
Calculer le « ET » logique

Comme on l'a dit plus haut, il suffit d'additionner 1 à `wc`. Si les deux bits sont à 1, alors on obtient `011 + 1 = 100`. Sinon, on obtient un nombre inférieur à `100`, qui s'écrit `0xx`. On voit que le résultat du « ET » logique est le troisième bit. Au final, l'algorithme consiste en une incrémentation, puis un double décalage vers la droite (c'est-à-dire une division par quatre) :



Remarquer que l'écriture des nombres est inversée (bits de poids faible à gauche, bits de poids fort à droite).

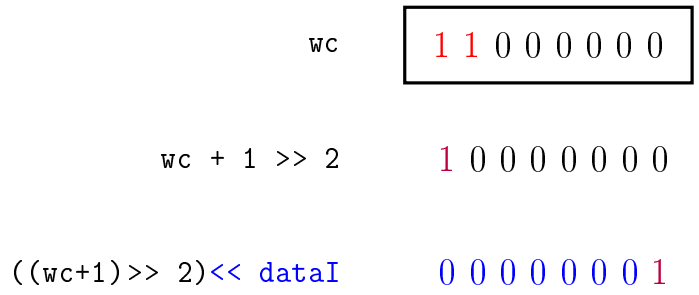
Afin de se convaincre, un autre exemple.



Empiler efficacement le résultat

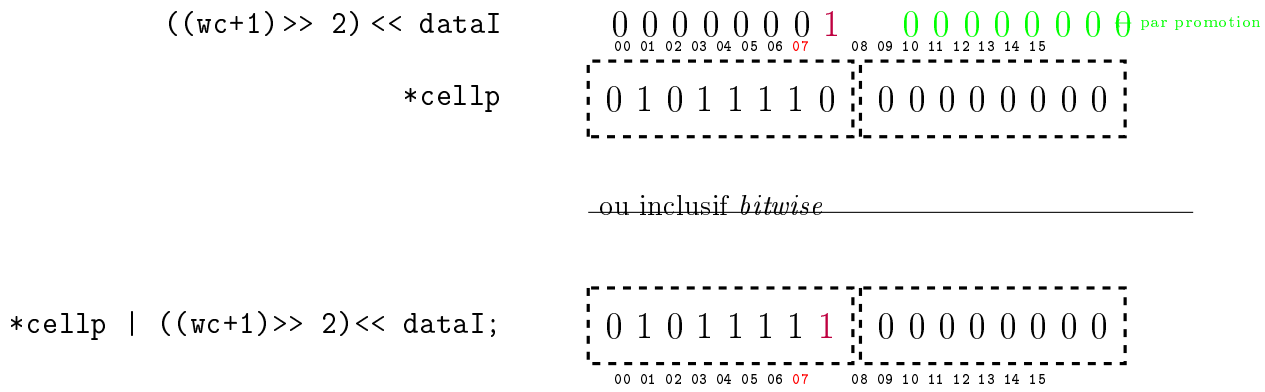
La façon dont on indexe les bits et les cellules, et l'invariant consistant à mettre à zéro les cases vides, n'ont pas été choisis par hasard : ils permettent d'empiler des bits avec seulement deux opérateurs de bits (*bitwise*) C : un décalage et un « OU ».

Notre résultat se trouve sur le bit 0, mais nous souhaitons l'empiler, c'est-à-dire l'écrire sur le `dataI`-ième bit de la double-cellule. Si on décale notre résultat `dataI` fois vers la gauche, il se trouve alors sur le `dataI`-ième bit. L'expression finale est alors `((wc+1)>> 2)<< dataI`.



Les propriétés des opérateurs de décalage font que tous les bits autres que le `dataI`-ième sont à zéro. Le `dataI`-ième bit de la double-cellule est lui aussi à zéro. Par conséquent, si on opère un « OU » bit à bit, on met à jour seulement le `dataI`-ième bit de la double-cellule :

```
*cellp |= ((wc+1) >> 2) << dataI;
```



Au final, le code de l'opérateur `AND` est le suivant :

Code 6.1 – Opérateur AND

```
inline void eacsl_meh_AND() {
  bSi-=2;
  //pointer to the cell
  bS_doublecell* const cellp = (bS_doublecell*)(boolStack + (bSi /
     BS_CELL_BIT) );
  //our 2 bits are data are at dataI and dataI+1
```

```
  const bS_bit_index dataI = bSi % BS_CELL_BIT;
  //move them to the beginning
  const char wc = *cellp >> dataI;
  //erase them from the stack
  *cellp = *cellp ^ (wc << dataI);
  //add 1 : the result is at bit 3
  //extract the result, then paste it to bit dataI (top of stack)
  *cellp |= ((wc+1) >> 2) << dataI;
  bSi++;
}
```

Les autres opérateurs logiques sont conçus de façon analogue ; la seule difficulté supplémentaire peut être la non-commutativité.

6.4 Opérations sur des termes et des prédicats

La manipulation conjointe de la pile des termes et de la piles des Booléens n'a pas causé de difficulté supplémentaire.

6.5 Tests unitaires des opérations

Après l'écriture de chaque opération, des tests ont été écrits dans un fichier C. Pour les opérateurs logiques, toutes les entrées possibles sont testées. Pour les opérateurs arithmétiques de données signées, tous les couples de signes sont testés. Enfin, l'intégrité des données mises sur les deux piles est testée : on empile et on dépile des données arbitraires, et on vérifie si elles sont constantes (malgré d'autres opérations au-dessus d'elles). Ces tests ne sont pas encore intégrés à la suite de tests d'*E-ACSL*, par manque de temps.

7. Écriture du compilateur

On considère qu'un prédicat est la racine d'un arbre dont les nœuds sont des opérateurs (ou d'autres primitives d'ACSL). À partir de chaque nœud, on peut former un sous-arbre, qui est un prédicat ou un terme.

Après plusieurs analyses fournies par diverses parties du noyau et d'*E-ACSL*, le code source (C annoté en ACSL) est modélisé sous la forme d'un tel AST[1]. C'est à partir de cet AST qu'on va compiler vers *meal*.

Dans la suite de ce rapport, par « opération meal », on désigne un couple formé d'un opérateur de *meal*, et d'un n-uplet de types compatibles avec cet opérateur (n dépend de l'opérateur).

Afin d'expliquer l'algorithme de compilation, on commence par ne considérer que les prédicats dont la descendance est consituée seulement de prédicats.

On écrit une fonction récursive dont les arguments sont un prédicat et une liste d'opérations meal, et dont la sortie est une liste d'opérations meal.

La fonction, pour traduire un prédicat à n nœuds fils, va faire n appels à elle-même, successivement, en augmentant la taille de la liste à chaque appel. (La fonction ne fait qu'ajouter des éléments à la liste). Ensuite, elle ajoute à la liste l'opérateur *meal* correspondant au nœud racine.

Par exemple, pour traduire `p && q` :

Code 7.1 – Récursion pour la traduction d'un prédicat à deux opérandes

```
liste := traduire(p, liste)
liste := traduire(q, liste)
liste := AND -> liste
```

L'ensemble des termes et prédicats de *mini E-ACSL* sont traduits selon ce principe. Mais l'algorithme est un peu plus complexe.

Il existe une fonction de traduction des termes, et une fonction de traduction des prédicats : ces deux fonctions sont récursives et mutuellement récursives.

De plus, *meal* et *meh* sont limitants et ne permettent pas de compiler l'ensemble d'ACSL (mais seulement *mini E-ACSL*). Si on veut construire une version utilisable d'*E-ACSL*, et supporter un ensemble assez grand d'ACSL (malgré les limitations de *meh*), il faut intégrer la compilation *meal* dans le compilateur pré-existant (qu'on appelera *legacy* dans la suite de ce rapport). Cela signifique que certains termes et prédicats seront gérés par la chaîne de compilation *meal*, et que d'autres seront gérés par la compilation *legacy*. Comme la compilation d'un prédicat requiert la compilation d'autres termes et prédicats, le code généré peut être un « mélange » de *meal* et de C *legacy* !

1. voir définition en section 2.4.2.

Il a donc fallu prévoir des « interfaces », qui fonctionnent par empilement d'expressions (`e_acsl_meh_push_BOOL(exp)`) et expressions de dépilement (`e_acsl_meh_pop_BOOL()`). En effet, en compilation *legacy*, la valeur d'un nœud est passée au nœud père par une expression (c'est-à-dire que le code d'un nœud contient des expressions dont l'évaluation permet d'obtenir la valeur de ses nœud fils) ; alors qu'en compilation *meal*, le passage des valeurs des nœuds enfants est « implicite » : elles sont dans la pile.

Par conséquent, si un nœud est en train d'être traduit en *meal* et la compilation de son fils ne peut être faite qu'en *legacy*, alors il faudra empiler l'expression retournée par la compilation *legacy*. De même, si un nœud est en train d'être traduit en *legacy* et la compilation de son fils est faite en *meal*, il faudra dépiler la valeur (une seule fois [2]) pour obtenir une expression qu'on pourra injecter dans le code du père.

TABLE 7.1 – Principe de l'interface entre *legacy* et *meal*

Nœud fils \ Nœud père	*meal*	*legacy*
meal	Passage au sommet de la pile	Dépilement
legacy	Empilement	Passage par expression

Concrètement, la fonction présentée plus haut se décline pour ces quatre cas. (`meal` est la variable contenant la liste d'opérations *meal* ; `env` est son équivalent en C (il modélise un bloc de code voisinage des expressions générées)).

Code 7.2 – Traduction meal/meal (pseudo-langage)

```
fonction ET(p, q, meal)
meal = traduire(p, meal)
meal = traduire(q, meal)
retourner (AND -> meal)
```

Code 7.3 – Traduction legacy/legacy (pseudo-langage)

```
fonction ET(p, q, env)
e1, env = traduire(p, env)
e2, env = traduire(q, env)
retourner ((e1 && e2), env)
```

Code 7.4 – Traduction legacy/meal (pseudo-langage)

```
fonction ET(p, q, meal, env)
e1, env = traduire(p, env)
e2, env = traduire(q, env)
meal = AND -> PUSH(e2) -> PUSH(e1) -> meal
retourner (meal, env)
```

Code 7.5 – Traduction meal/legacy (pseudo-langage)

```
fonction ET(p, q, meal, env)
meal = traduire(p, meal)
```

2. on peut éventuellement créer une variable locale si on en a besoin plusieurs fois

```
meal = traduire(q, meal)
e2, env = resultatFonction(e_acsl_meh_POP, env)
e1, env = resultatFonction(e_acsl_meh_POP, env)
retourner( (e1 && e2), meal, env )
```

Et pour implémenter ces fonctions plus facilement, on a uniformisé la signature de ces fonctions. C'est-à-dire qu'au final, la fonction de traduction d'un prédicat a pour arguments :

— un prédicat,
— une liste d'opérations *meal* et
— un *environnement*[3],

et elle retourne :

— la même liste d'opérations *meal*, éventuellement agrandie et
— le même *environnement*, éventuellement étendu.

On a donc pris le parti de faire de *meal* notre *lingua franca* : toutes les parties d'*E-ACSL* qui traduisent un terme ou un prédicat sous la forme d'une expression (et d'un environnement étendu) verront leur résultat empilé.

On remarquera qu'en l'état actuel de *meh*, l'une des quatre cases du tableau ne peut pas être implémentée pour les termes. En effet, comme *meh* ne peut pas stocker certains types de données (les nombres de précision arbitraire traités par GMP), il est impossible d'empiler certains termes, même si leurs termes et prédicats ancêtres sont traduisibles en *meal* et en *meh*. Dans la version actuelle du projet, le compilateur ne supporte donc que les termes de *mini E-ACSL*.

En revanche, l'interface entre la génération de prédicat *meal* et la génération de prédicats *legacy* fonctionne, puisque la valeur de vérité d'un prédicat est forcément « vrai » ou « faux » (ou undefined, mais ce cas sont gérés à l'intérieur du code *legacy*).

Le dernier aspect qui complexifie l'algorithme présenté plus haut est la présence de deux traitements après compilation des nœuds.

Le premier est la conversion (par *casts* ou appels à GMP). En effet, pour réaliser les calculs de certains nœuds, on a parfois besoin que la traduction des opérandes soit retournée dans un certain type. Pour le moment, les nœuds requérant une traduction de leur fils dans un type spécifique[4] sont des nœuds non supportés par meal, donc traduits en *legacy* : on peut donc dépiler et appliquer la conversion.

Le second est l'ajout de gardes contre les erreurs à l'exécution (telles que les divisions par zéro). Dans legacy, ces gardes sont ajoutées directement au niveau du code C qui est généré. L'ajout de ces gardes est parfois compliqué dans le code généré à travers *meal*, et je n'ai pas cherché à garantir leur bon fonctionnement au cours de mon stage.

Voici un extrait du code de compilation des prédicats tel que je l'ai écrit.

Code 7.6 – Compilation des prédicats en *meal*

```
(* Convert an ACSL predicate into a corresponding C expression (if any)
   in the
   given environment. Also extend this environment which includes the
      generating
   constructs. *)
(* the rte parameter is used only
```

3. Structure de données propre à *E-ACSL* modélisant un bloc de code au voisinage d'un emplacement d'un programme C
4. différent de celui qui aurait été choisi pour un nœud racine

```
   if the predicate is as expressed as a C expression
   embedded in a meal bool push *)
and predicate_content_to_meal ?name kf ?rte ml env p =
  (** build some expression along with an env **)
  let loc = p.pred_loc in
  match p.pred_content with
  (** nodes translatable to meal **)
  | Pfalse    -> Mpush(MBool(false)) :: ml, env
  | Ptrue     -> Mpush(MBool(true)) :: ml, env
  | Pand(p1, p2) ->
    let ml, env = predicate_to_meal kf ?rte ml env p1 in
    let ml, env = predicate_to_meal kf ?rte ml env p2 in
    MAND :: ml, env
  | _ ->
    (** nodes not translatable no meal **)
    let e, env =
      match p.pred_content with
      | Pnot p ->
        let e, env = predicate_to_exp kf env p in
        Cil.new_exp ~loc (UnOp(LNot, e, Cil.intType)), env
      | Pforall _ | Pexists _ -> Quantif.quantif_to_exp kf env p
    in let e, env = cast_and_rte_predicate_as_exp ?name kf ?rte env p e in
    Mpush(Mexpr(MTBool,e)) :: ml, env

and predicate_content_to_exp ?name kf ?rte env p =
  let ml, env = predicate_content_to_meal ?name kf ?rte [] env p in
  let loc = p.pred_loc in
  let env = meal_to_meh_calls kf env ml ~loc in
  (*  if rte then *)
  (*copied Pand code*)
  Env.with_rte_and_result env true
    ~f:(fun env ->
        Env.rtl_call_to_new_var ~loc env kf None Meh.meh_bool "
           meh_pop_BOOL" []
      )


and predicate_to_exp ?name kf ?rte env p =
  (*code duplicated in cast_and_rte_predicate_as_exp*)
  let rte = match rte with None -> Env.generate_rte env | Some b -> b in
  Env.with_rte_and_result env false
    ~f:(fun env ->
        let e, env = predicate_content_to_exp ?name kf ~rte env p in
        let env = if rte then translate_rte kf env e else env in
        let cast = Typing.get_cast_of_predicate p in
        add_cast
          ~loc:p.pred_loc
          ?name
          env
          kf
          cast
```

```
          C_number
          None
          e
      )

and cast_and_rte_predicate_as_exp ?name kf ?rte env p e =
  (*code copied from predicate_to_exp*)
  let rte = match rte with None -> Env.generate_rte env | Some b -> b in
  Env.with_rte_and_result env false
    ~f:(fun env ->
        let env = if rte then translate_rte kf env e else env in
        let cast = Typing.get_cast_of_predicate p in
        add_cast
          ~loc:p.pred_loc
          ?name
          env
          kf
          cast
          C_number
          None
          e
      )
```

Les prédicats `Pfalse`, `Ptrue` et `Pand(p1, p2)` sont supportés par le compilateur *meal*. La branche qui fait la compilation de `Pand` fait un appel à `predicate_to_meal` (récursion), pour obtenir une liste d'opérations *meal* (appelée `ml` ici) étendue.

Les prédicats `Pnot`, `Pforall` et `Pexists` sont compilés de façon *legacy*. On peut voir que la branche en charge de `Pnot` fait un appel à `predicate_to_exp`, pour obtenir une expression et un environnement (appelé `env` ici) étendu.

J'aurais pu ne pas modifier la fonction `predicate_to_exp`, mais pour forcer la génération de code *meal*, j'ai remplacé le corps de cette fonction par :

— un appel à `predicate_to_meal`, qui crée une liste d'opérations *meal* (variable `ml`), suivi par
— un appel à `meal_to_meh_calls`, le compilateur qui traduit des opérations *meal* en des appels à *meh*en C modélisés par une expansion de l'environnement `env`, et
— l'ajout dans `env` d'une expression à laquelle on assigne le résultat de l'appel à la fonction `e_acsl_meh_pop_BOOL()`.

Ainsi, la signature de la fonction `predicate_to_exp` n'a pas changé : on retourne toujours une expression et un environnement étendu par la traduction du prédicat donné, mais cette traduction est réalisée en *meal*, et complétée par un dépilement.

On notera la présence d'une fonction `cast_and_rte_predicate_as_exp` dont le rôle est d'ajouter les conversions et gardes d'erreurs à l'exécution au moment de l'empilement d'expressions dont la génération a été déclenchée par une compilation *meal*.

La compilation des termes a été conçue de façon analogue.

Conclusion

Au cours de ce stage au Laboratoire de Sûreté et de Sécurité des Logiciels (LSL) du CEA List, institut de recherche technologique en informatique en soutien à l'industrie, j'ai travaillé sur la plateforme Frama-C, un outil d'analyse des programmes écrits en C, et spécifiés (annotés) par des propriétés logiques écrites en ACSL. Plus précisément, j'ai contribué au développement de *E-ACSL*, greffon de Frama-C traduisant les propriétés logiques en instrumentations, c'est-à-dire transformant les programmes C en leur ajoutant du code chargé de vérifier les annotations lors de l'exécution.

J'ai étudié le comportement d'*E-ACSL*, suggéré des optimisations du code généré, conçu un schéma de compilation implémentant ces optimisations, adapté *E-ACSL* pour qu'il utilise ce schéma de compilation, et partiellement fusionné le code existant avec ma nouvelle version. Le problème le plus important au moment du début de ce stage (la « prolifération » de variables intermédiaires de type `int` utilisées pour le calcul des booléens) est résolu.

Pour les termes et prédicats simples d'ACSL, ma version du compilateur *E-ACSL* fonctionne. Après avoir été augmentée de quelques autres opérations C et tests et prédicats ACSL, et scrupuleusement testée, la compilation passant par un langage intermédiaire tel que *meal*, et utilisant des appels à une bibliothèque C telle que *meh*, pourrait être distribuée.

L'amélioration des performances d'un greffon de Frama-C comme *E-ACSL* rend l'utilisation d'outils de vérification formelle plus facile. En effet, *E-ACSL* permet, dans certains cas, de déboguer rapidement des annotations ACSL ; par ailleurs, on peut, sur des systèmes bien dimensionnés, envisager d'instrumenter les exécutables en production. L'amélioration des performances d'*E-ACSL* augmente le nombre des systèmes sur lesquelles il peut être exécuté.

L'amélioration des performances d'*E-ACSL* rend la vérification à l'exécution plus accessible à l'industrie informatique en général, et pas seulement à l'industrie des systèmes critiques. Certains secteurs soumis à des délais de livraison courts (web...) ne peuvent pas, pour le moment, implémenter des procédures de qualité impliquant de la preuve de code. On peut espérer que l'amélioration des performances des outils de vérification comme Frama-C les rendent utilisables par une plus grande part de l'industrie, et favorisent leur intégration dans des outils d'intégration continue, améliorant ainsi la sûreté et la sécurité des applications en général.

Afin d'étudier *E-ACSL*,de concevoir *meal*, *meh*, et les schémas de compilation, j'ai utilisé de nombreux concepts théoriques issus de ma formation d'ingénieur, et j'en ai appris de nouveaux. Complexité, structures de données et récursion ont été des notions centrales au cours de ce stage. Quelques mathématiques, un peu de logique, et un peu de théorie des langages ne m'ont pas été inutiles, notamment à la lecture de certains textes scientifiques, mais aussi lors de la conception de *meal* et de *meh*.

J'ai renforcé ma connaissance de C, de la compilation et du fonctionnement des machines. Je ne les ai pas encore lues en entier, mais la norme C99 de l'ISO [3] et la référence x86 d'Intel me sont devenues familières.

J'ai découvert le paradigme fonctionnel et l'intérêt des langages fortement typés. OCaml est un langage fonctionnel, fortement typé, et orienté objet : il combine beaucoup de paradigmes modernes de programmation. J'ai aussi amélioré mes pratiques de développement. Par exemple, j'ai amélioré mon style de programmation, grâce aux conseils de mon maître de stage et de mon équipe. J'ai aussi été confronté à des techniques de génie logiciel, notamment les tests et l'architecture (du code), qui favorisent sa qualité et sa maintenabilité.

Enfin, ce stage m'a également permis de me familiariser avec la vérification de code, ce qui complète ma formation à l'UTC, notamment l'UV LO22[5].

Ce stage m'a permis d'augmenter ma culture technologique, notamment en ce qui concerne la programmation bas-niveau et la compilation. Il m'a aussi permis de confirmer mon appétence pour les aspects plus théoriques, mathématiques et scientifiques de l'informatique, notamment ceux liés à la sémantique. C'est pourquoi je souhaite toujours m'inscrire en filière IAD (Intelligence artificielle et science des données).

5. qui met plus l'accent sur l'architecture des systèmes, et sur les méthodes de raffinement de code

Glossaire

AST ou *Abstract Syntax Tree* ou encore arbre de la syntaxe abstraite. Il s'agit d'une représentation intermédiaire d'un programme, utilisée par des analyseurs et compilateurs pour parcourir un programme de façon abstraite, c'est-à-dire qu'on modélise la sémantique d'un programme par un arbre. Par exemple, lorsqu'on modélise l'expression arithmétique C `x + 1`, l'opérateur `+` est un nœud dont les deux fils sont les feuilles `x` et `1`. La forme de l'AST n'est pas nécessairement semblable à celle du programme.

E-ACSL est un *greffon* distribué avec Frama-C. Il supporte un sous-ensemble du langage ACSL. Il traduit des propriétés logiques (exprimées en ACSL) en instrumentations chargées de vérifier les propriétés à l'exécution. Cette instrumentation est une transformation du programme C : *E-ACSL* est donc un compilateur.

meal ou *mini E-ACSL assembly language* est un langage intermédiaire utilisé par *E-ACSL*. Ses primitives sont des opérations sur des données contenues dans une structure de pile.

meh ou *mini E-ACSL hugger-mugger* est une bibliothèque C fournissant une pile et des opérateurs sur cette pile. *meh* peut stocker la valeur de certains termes, et la valeurs de prédicats d'*E-ACSL*.

mini E-ACSL est un sous-langage d'ACSL (et ayant la sémantique définie par *E-ACSL*). On peut le compiler en *meal*, et compiler le code *meal* ainsi obtenu en un code C faisant des des appels à *meh*.

termes et prédicats sont des constructions d'ACSL. En ACSL, la valeur d'un prédicat est « vrai » ou « faux » ; en *E-ACSL*, elle peut aussi être « *undefined* ». Les opérandes des prédicats peuvent être des termes ou d'autres prédicats. Lire la référence d'ACSL : [1].

paresse d'un opérateur, ou *laziness* : il s'agit d'une propriété de certains opérateurs qui conditionnent l'évaluation d'une opérande à la valeur d'une autre opérande. Cela peut éviter des calculs, mais aussi des erreurs. Par exemple, sans la paresse d'ACSL, l'expression `\true || (42 == 1/0)` provoquerait des erreurs ; l'évaluation de la même expression, en C, est un *undefined behaviour* car la paresse n'est pas toujours garantie.

Bibliographie

[1] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy et Virgile Prevosto. *ACSL : ANSI/ISO C Specification Language*. 1.16. CEA LIST, Software Security Laboratory et INRIA. 2020. URL : https://github.com/acsl-language/acsl.

[2] Henry S. Warren, Jr. *Hacker's Delight*. Second. Boston : Addison-Wesley Professional, 2012. ISBN : 9780133084993.

[3] *International Standard ISO/IEC 9899*. Standard. Ultimate revision of the C99 standard. ISO/IEC JTC 1, sept. 2007.

[4] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles et Boris Yakobowski. « Frama-C : A Software Analysis Perspective ». English. In : *Formal Aspects of Computing* (jan. 2015), p. 1-37. URL : http://julien.signoles.free.fr/publis/2015_fac.pdf.

[5] Dara Ly, Nikolai Kosmatov, Frédéric Loulergue et Julien Signoles. « Verified Runtime Assertion Checking for Memory Properties ». In : *International Conference on Tests and Proofs (TAP)*. Juin 2020. URL : http://julien.signoles.free.fr/publis/2020_tap.pdf.

[6] George C. Necula, Scott McPeak, Shree Prakash Rahul et Westley Weimer. « CIL : Intermediate Language and Tools for Analysis and Transformation of C Programs ». In : *International Conference on Compiler Construction (CC'02)*. 2002.

[7] Julien Signoles. *Executable ANSI/ISO C Specification Language*. Version 1.16 – Frama-C plug-in E-ACSL version 22.0. Les termes et prédicats pas encore implémentés sont coloriés en rouge. Les termes et prédicats dont la sémantique est modifiée (par rapport à [1]) sont coloriés en bleu. CEA LIST, Software Security Laboratory. 2018. URL : https://www.frama-c.com/download/e-acsl/e-acsl-implementation.pdf.

[8] Julien Signoles, Thibaud Antignac, Loïc Correnson, Matthieu Lemerre et Virgile Prevosto. *Frama-C Plug-in Development Guide*. Release 22.0 (Titanium). CEA LIST, Software Security Laboratory. 2020. Chap. 4.5. URL : https://www.frama-c.com/download/frama-c-plugin-development-guide.pdf.

[9] Wikipédia. *Arbre de la syntaxe abstraite — Wikipédia, l'encyclopédie libre*. [En ligne ; Page disponible le 2-octobre-2020]. 2020. URL : http://fr.wikipedia.org/w/index.php?title=Arbre_de_la_syntaxe_abstraite&oldid=175220169.

Annexes

I. Fondement théorique sous-jacent au stage

Dans ce rapport est cité à plusieurs reprises l'article de Dara Ly, Nikolai Kosmatov, Frédéric Loulergue et Julien Signoles, intitulé « Verified Runtime Assertion Checking for Memory Properties » [5]. Cet article explique plusieurs choses :

- il modélise formalise, pour des propriétés liées à la mémoirew :
  - la transformation effectuée par *E-ACSL*,
  - la mémoire du programme source,
  - la mémoire du programme transformé (instrumenté), qui contient :
    - une mémoire « équivalente » à la mémoire du programme source (d'origine)
    - la mémoire d'observation, qui est un modèle de la mémoire observée
    - la mémoire d'instrumentation, qui est utilisée par *E-ACSL* pour faire ses propres calculs de vérification.
- il définit certaines propriétés attendues du modèle (par exemple le fait qu'une mémoire d'observation représente « bien » celle du programme observé), et pose certaines contraintes (par exemple la « séparation » de certaines mémoires)
- grâce à ces définitions et contraintes, il montre que :
  - la transformation est correcte. Elle ne change pas le comportement du programme lorsque les propriétés surveillées sont correctes, arrête l'exécution sinon.
  - les programmes en charge de vérifier les annotations **n'ont pas besoin de modifier la mémoire d'observation** (ni la mémoire du programme, évidemment) : ils n'écrivent que dans la mémoire d'instrumentation.
  - on peut construire un système de traduction ACSL vers C complet

Comme la mémoire d'instrumentation est totalement indépendante des deux autres, et est la seule dans laquelle les programmes de vérification des annotations écrivent, on peut en proposer différentes implémentations, à interface constante.

II. Quelques propositions d'amélioration du code généré par *E-ACSL*

La première phase du stage consista en l'étude du code généré par *E-ACSL* et en la proposition d'améliorations.

Ces remarques et ces propositions d'amélioration ne furent pas toutes retenues. Certaines d'entre elles n'étaient en effet pas pertinentes, car je ne connaissais pas certains aspects d'ACSL, par exemple la paresse de certains opérateurs. D'autres n'étaient pas implémentables, c'est-à-dire : qu'elles ne pouvaient pas être mises en œuvre facilement, pour des raisons techniques. Enfin, certaines remarques ont été retenues, car l'équipe était convaincue qu'elles apporteraient un gain de performance. En outre, les mises en œuvre de mes optimisations ont parfois différé de mes propositions, pour plusieurs raisons (par exemple, pour des raisons de performance non locales aux exemples cités, ou pour des raisons de factorisation du code).

Cette annexe présente quelques-unes de mes propositions, puis un tableau résumant mon étude exhaustive du code généré par chaque terme et chaque prédicat d'ACSL, accompagné de quelques propositions d'amélioration.

II.1 Opérations logiques et variables Booléennes : l'exemple du « ou » inclusif

Code source

```
int main(void) {
  int x = 0;
  /*@ assert x == 0 || x == 1; */
  return 0;
}
```

Code généré

```
int main(void)
{
  int __retres;
  int x = 0;
  {
    {
      int or;
      if (x == 0) {
        or = 1;
```

```
      }
      else {
        or = x == 1;
      }
      __e_acsl_assert(or,"Assertion","main","x == 0 || x == 1",
                      "ou.1.c",4);
    }
    /*@ assert x == 0 || x == 1; */ ;
  }
  __retres = 0;
  return __retres;
}
```

À partir de cet exemple, j'ai suggérer deux modifications :
— utiliser l'opérateur C `||` comme traduction de l'opérateur ACSL `||` et
— utiliser le type standard `bool`.

L'opérateur ACSL `||` est *paresseux* (*lazy*) [1]. Ce comportement est attendu par le programmeur ACSL, qui peut l'utiliser pour choisir d'évaluer ou de ne pas évaluer sa seconde opérande. Cette technique est utile pour éviter d'évaluer des expressions conduisant à des *undefined behaviours* au sens de la norme C99 [3]. Le branchement (`if ... else`) est nécessaire, et par conséquent on a un état à gérer et on utilise une variable : dans cet exemple, c'est la variable `or`.

Il n'est donc pas possible de traduire l'opérateur `||` de ACSL par l'opérateur `||` de C : ma proposition n'est donc pas acceptable en l'état.

Le type `signed int` est représenté sur au moins 4 octets [3], voire 8 octets pour des compilateurs et architectures modernes (tels que gcc sur x86). Ma proposition était d'utiliser plutôt le type `bool` et les constantes `true` et `false` introduits par la norme C99. Cette proposition apporte un peu de sûreté, mais il y aura de toute façon (explicitement ou implicitement) des casts vers des `int`. En effet, les opérateurs arithmétiques et logiques de C ont une règle de *promotion*. Cela permet, dans certains cas, de garantir certaines propriétés de non-débordement, et cela permet au compilateur de réutiliser les opérations définies pour les autres types entiers. Au final, l'utilisation du type `bool` n'apporte aucun gain en espace.

Cependant, nous avons fait le postulat qu'il était possible de décider statiquement (c'est-à-dire au moment de la compilation par *E-ACSL*) et efficacement que la paresse n'était pas réellement utile dans certains cas. Dans ce rapport est présentée une autre solution, qui ne fait pas recours aux opérations sur les entiers de C (qui provoqueraient une *promotion* [3]).

II.2 Étude du code généré par chaque terme et chaque prédicat (sans combinaison)

L'étude de ces termes et prédicats est disjointe en sous-cas, en fonction des valeurs ou des types de leurs termes et prédicats fils.

Cette disjonction de cas permet de faire apparaître des motifs réccurrents, qui permettent de classer les termes et prédicats.

Pour les prédicats, on définit trois classes :

— Les prédicats **simples**. Ils peuvent être reconnus syntaxiquement et sont toujours traduits une expression C qui constitue une *condition* (au sens de [3]) valable (on parlera de « condition pure » dans la suite de ce document).
— Les prédicats **gris**. Selon la valeur de leurs termes et prédicats fils, ils peuvent parfois être traduits par une condition pure.
— Les prédicats **compliqués**. Le calcul de leur valeur nécessite toujours un algorithme compliqué (avec des structures de contrôle, du code séquentiel, des appels de fonction...) qui ne pourrait être exprimé par une simple expression C.

Pour les termes, on définit trois classes :

— Les termes **simples**. On peut les reconnaître par leur syntaxe. Ils sont toujours traduits par une expression C qui exprime un arbre d'opérations arithmétiques et logiques.
— Les termes **gris** terms. Selon la valeur de leurs termes et prédicats fils, ils peuvent parfois être traduits par un expression C exprimant un arbre d'opérations arithmétiques et logiques.
— Les termes **compliqués** terms. Ils sont toujours traduits par un algorithme compliqué (avec des structures de contrôle, du code séquentiel, des appels de fonction...) qui ne pourrait être exprimé par une simple expression C.

Parfois, des termes et prédicats gris sont traduits par du code compliqué (qui n'est pas une simple expression C). Décider si le terme est simple... est parfois compliqué. *E-ACSL* n'a pas vocation à faire des analyses poussées, ni à exécuter à l'avance le programme. C'est pourquoi toutes les propositions d'amélioration des termes et prédicats gris ne sont pas forcément implémentables ; il nécessitent parfois l'ajout d'heuristiques à *E-ACSL*.

Dans ce tableau, une taille d'allocation est donnée en *octets*, pour le compilateur gcc sur l'architecture AMD64. Lorsque le code généré fait appel à la bibliothèque de calcul de précision arbitraire GMP, seule l'allocation du terme considéré est rapportée dans ce tableau ; cependant, des termes parents ou enfants d'un terme converti en GMP sont susceptibles d'être eux-mêmes convertis en GMP, ce qui crée une augmentation de l'espace mémoire non rapportée dans ce tableau.

II.2.1 Forme du code généré par les termes et prédicats d'*E-ACSL*

Terme/prédicat	Sous-cas	Classe	Code généré actuellement	Alloc	Proposition d'amélioration	Alloc
terme simple		simple	arbre arith.	0		
predicat simple		simple	condition pure	0		
Const. $\in \mathbb{Z}$	tient dans constante C	simple	la constante C	4-16		0
Const. $\in \mathbb{Z}$	ne tient pas dans constante C	compliqué	GMP	16		
Constante à virgule	tient dans constante C	gris	GMP / constante	16		
Constante à virgule	ne tient pas dans constante C	compliqué	GMP	16		
Chaîne de car. const.		simple	la constante (multi-char)	0		0
`+ - * /`	rationels grands	compliqué	3 MPQs: 2 opérandes, 1 var. de rés.	48	réutil. var. d'1 opérd.	32
`>>` et `<<`	cas général	compliqué	3 MPZs: 2 opérandes, 1 var. de rés.	48	réutil. 1 opérd var.	
`a >> b` et `a << b`	b tient dans `bitnct` (alias `long`)	compliqué	`(MPZ)b < ULONG_MAX` ? `(MPZ)a << (ulong)(MPZ)b`	60	const. ulong, réutil. var. d'1 opérd.	36
`<, >, (<, >, !, =)=`	rationels grands	compliqué	3 GMP MPZs : 2 opérandes, 1 variable de résultat	48	réutil. var. d'1 opérd.	32
`<, >, (<, >, !, =)=`	a : tient, b : ne tient pas ds. const. C	simple	3 GMP MPZs : 2 opérandes, 1 variable de résultat	48		
`+a, -a`	C/ACSL int/`float`/real	simple	`+a, -a`	0		
`a == b, a != b`	tableaux (petits)	compliqué	boucle while. Var. d'itér: ulong. Résult.: int.	12	Var. ité. petite; assert. ds. corps boucle	2-8
`a == b, a != b`	strings (char[])	simple	**Comparaison de pointeurs!**	0-8		
`\let id = a; b`	a,b : *termes simples*	compliqué	1 var. de même type	s(a)		

Terme/prédicat	Sous-cas	Classe	Code généré actuellement	Alloc	Proposition d'amélioration	Alloc
`\let id = a; b`	a : `float`, tient dans const. C	simple	`double` const.			
`\let id = a; b`	a : non const./ne tient pas ds. type C	gris	1 int (res), 2 MPQs (assignee-ed)	36		
`\forall`, `\exists`	bornes petites	compliqué	2 ints (var. ité, rés. Booléen)	8	si fils d'une exp. : pas de bool. intermédiaire	4-8
`\forall`, `\exists`	bornes grandes	compliqué	3 ints, 4 MPZs	76	1 int, 3 MPZs	52
`\at`	1 var. simple	compliqué	var. copiée	s(var)		
`\at`	tabl., 1 index simple	compliqué	copié dans 1 var.	s(vrs)		
`\at`	tabl., index par quantif.	compliqué	alloc. approximative, copie exacte	Voir note [1]	alloc. exacte pour les index simples	
`\initialized`		compliqué	1 bool	4	si fils d'une exp. : pas de bool. intermédiaire	0-4
`\valid`		compliqué	1 bool + \initialized + mem primitive	8	si fils d'une exp. : pas de bool. intermédiaire	0-8
`\separated`		compliqué	1 bool + 1 \valid per pointer + mem prim	1+8p	pas de bool. inter. / réutil. bool `valid`	0-8
`\block_length`		compliqué	1 bool	4	si fils d'une exp. : pas de bool. intermédiaire	0-4
`behavior:`		compliqué	~ 1 bit par assumes + at et bools intermédiaires	8+a	Booléens inter. (ats redondants)	

1. `\at`'s compilation scheme (including its approximation of the size of the memory to allocate necessary to the copy of old parts of arrays) is detailed in *Runtime Assertion Checking with Frama-C/E-ACSL : the `\at` Construct*, a presentation given by Fonenantsoa Maurica and Julien Signoles at the *Méthodes de Test pour la*

Vérification et la Validation conference on the 7th of December 2018. Basically, the max of all branches is taken and is multiplied.

III. Spécification et modélisation du langage *meal*

(En anglais). The mini E-ACSL assembly language (*meal*) is a language for intermediate representation of a subset of *E-ACSL* terms and predicates called *mini E-ACSL*. *meal* provides elementary operations and access to a typed stack. *E-ACSL* is a compiler: it takes as input *mini E-ACSL* code, and outputs *meal* code. *meal* code can then be compiled to C code bound to the meh library. This assembly is done by *E-ACSL*, too.

The terms "assembly" and "assembler" have been chosen to highlight the fact that *meal* can be translated to C by an almost pure function applied to each line of code. However, it is not machine code (but it could easily be translated to machine code).

meal allows pushing and popping data to and from a stack, and performing operations on this data. Data is always typed, and all operations (including pushing and popping) are always typed. *meal* does not specify how the stack should be implemented. Thus, several implementations of the stack are possible, including a stack with variable element size, or several stacks (one per type).

III.1 *meal* types

For each *E-ACSL* and C type, there is a unique corresponding *meal* type.

III.1.1 term types

Signed and unsigned integer types (expressed as a number of bits):

```
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<number> ::= <digit>|<digit><number>
<sit> ::= S<number>
<uit> ::= U<number>
<it> ::= <sit> | <uit>
```

`MPZ` is a type for mathematical integers.

`MPQ` is a type for mathematical rationals.

`BOOL` is a type for boolean values.

Logic types:

```
<log> ::= MPZ | MPQ | BOOL
```

All in all:

```
<type> ::= <logictype> | <it>
<st> ::= <sit> | MPZ | MPQ
<cn> ::= <it> | BOOL | MPZ
<cnn> ::= <it> | MPZ
<num> ::= <it> | MPZ | MPQ
<fin> ::= <it> | BOOL
```

Some operators suppose that their operands are signed integers, and some operands suppose that they are unsigned integers. Be careful!

III.1.2 predicate type

In *meal*, algorithms for computing predicates have been generated; thus, any variable holding a predicate is a boolean. The `BOOL` type is used.

III.2 stack access primitives

```
PUSH type a
POP type
```

`a` can be a litteral or a C expression; in the latter case, it must be enclosed in brackets `( expr )`.

III.3 General rules for binary operators

III.3.1 Order of stack operands

For non-commutative binary operators, the second operand is at the top of the stack, and the first operand is behind.

III.3.2 Types

When operands can have several types, the type must be provided as an argument to the primitive.

For a binary operator, if the operands are not necessarily of the same type, the first argument to the primitive is the type of the first operand, and the second argument is the type of the second operand.

For a binary operator, if the result is not necessarily of the same type as of the argument(s), the last argument to the primitive is the type of the result.

III.4 primitives for predicate -> predicate operations

— **input**: The predicate (operand) must be loaded on the top of the stack, as a `BOOL`.
— **output**: A predicate at the top of the stack, as a `BOOL`.
— **invariants**: All the elements of the stack being under the operand are unchanged.
— other modifications of the environment: none

primitive	Description
NOT	Logical negation on a boolean.

III.5 primitives for term -> predicate operations

— **input**: The term (operand) must be at the top of the term stack.
— **output**: A boolean at the top of the boolean stack.
— **invariants**: All the elements behind the input and the output are unchanged
— other modifications of the environment: none

primitive	Description	Unsupported types
OPP `st`	Opposite	Unsigned types.

III.6 primitives for term -> term operations

— **input**: The term (operand) must be loaded on the top of the term stack.
— **output**: A term at the top of the stack.
— **invariants**: All the elements of the term stack being under the operand are unchanged.
— other modifications of the environment: none

primitive	Description	Unsupported types
NNL `type`	"Not nul" (logical negation on term)	
BCL `cn`	Bitwise complementation on integer type	MPQ

III.7 primitives for (term,term) -> term operations

`type` must be one of the tokens listed in the "term types" section.

— **input**: The two operands must be loaded on the top of the stack. They must be of the same type `type`, and the result must fit in this type. If the operator $\circ$ is not commutative (i.e. $op1 \circ op2 \neq op2 \circ op1$) then the second operand (op2) is the one at the top of the stack, and the first operand (op1) is the one below. For some operations, certain types are not supported or are not valid.
— **output**: The result is at the top of the stack, and is of type `type`.
— **invariants**: All the elements of the stack being under the two operands are unchanged.

— other modifications of the environment: the number of elements of the stack has been decremented by one.

primitive	Description	Unsupported types
ADD `num`	Add (op1 + op2)	BOOL
SUB `num`	Substract (op1 - op2)	BOOL
MUL `num`	Multiply (op1 × op2)	BOOL
DIV `num1 num2 numres`	Divide (res = op1/op2, op1: `num1`, op2: `num2`, res: `numres`)	BOOL
MOD `cnn1 cnn2 cnnres`	Modulo (res = op1%op2, op1: `cnn1`, op2: `cnn2`, res: `cnnres`)	MPQ, BOOL
BND `cn`	Bitwise and	MPQ
BOR `cn`	Bitwise or	MPQ
BXR `cn`	Bitwise xor	MPQ
LND `fin`	Logical and on terms (op1 ∧ op2)	MPZ, MPQ
LOR `fin`	Logical or on terms (op1 ∨ op2)	MPZ, MPQ
LXR `fin`	Logical xor on terms (op1 ⊻ op2)	MPZ, MPQ
SHR `cn1 cn2`	Bit shift right (op1: `cn1`, op2: `cn2`)	MPQ for both operands
SHL `cn1 cn2`	Bit shift left (op1: `cn1`, op2: `cn2`)	MPQ for both operands

III.8 primitives for (predicate, predicate) -> predicate operations

— **input**: The two operands must be predicate values loaded on the top of the stack.
— **output**: The result is a predicate value at the top of the stack.
— **invariants**: All of the elements of the stack being under the two operands are unchanged.
— other modifications of the environment: the number of elements of the stack has been decremented by one.

primitive	Description
AND	Logical and on predicates
OR	Logical or on predicates
XOR	Logical xor on predicates

III.9 primitives for (term, term) -> predicate operations

limitation: both terms must have the same type

`type` must be one of the values listed above in the "types" section.

— **input**: The two operands must be loaded on the top of the stack. They must be of the same type `type`. If the operator $\circ$ is not commutative (i.e. $op1 \circ op2 \neq op2 \circ op1$) then the second operand (op2) is the one at the top of the stack, and the first operand (op1) is the one below.
— **output**: The result is at the top of the stack, and is a predicate.
— **invariants**: All the elements of the stack being under the two operands are unchanged.
— other modifications of the environment: the number of elements of the stack has been decremented by one.

primitive	Description
EQ `type`	Terms are equal (op1 = op2)
NEQ `type`	Terms are unequal (op1 $\neq$ op2)
LTE `type`	op1 is less than or equal to op2 (op1 $\leq$ op2)
GTE `type`	op1 is greater than or equal to op2 (op1 $\geq$ op2)
LT `type`	op1 is less than op2
GT `type`	op1 is greater than op2

III.10 Modélisation de la spécification de *meal* en OCaml

La spécification n'a pas été entièrement respectée ; il manque notamment le support des types GMP, qui n'a pas été fait par manque de temps.

Grâce à cette modélisation, on sait, à la compilation d'E-ACSL, que le code *meal* qui sera généré respectera la spécification. C'est équivalent à une vérification de syntaxe.

```
(*
   ************************************************************************
   *)
(*                                                                      *)
(*  This file is part of the Frama-C's E-ACSL plug-in.                  *)
(*                                                                      *)
(*  Copyright (C) 2012-2020                                             *)
(*    CEA (Commissariat a l'energie atomique et aux energies            *)
(*         alternatives)                                                *)
(*                                                                      *)
(*  you can redistribute it and/or modify it under the terms of the GNU *)
(*  Lesser General Public License as published by the Free Software *)
(*  Foundation, version 2.1.                                            *)
(*                                                                      *)
(*  It is distributed in the hope that it will be useful,               *)
(*  but WITHOUT ANY WARRANTY; without even the implied warranty of *)
(*  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the   *)
(*  GNU Lesser General Public License for more details.                 *)
(*                                                                      *)
(*  See the GNU Lesser General Public License version 2.1               *)
(*  for more details (enclosed in the file licenses/LGPLv2.1).          *)
(*                                                                      *)
```

```
(*
   ****************************************************************************
   *)

(** The top-level representation of a source file in mini e-acsl assembly
    language (meal). *)

(** Currently, the only data types supported by meh are:
  * - attribute-less C integer types
  * - bools
  **)
(* C signed integer types supported by meh *)
type meh_ikind_S =
  | MISChar
  | MIInt
  | MIShort
  | MILong
  | MILongLong

(* C unsigned integer types supported by meh *)
type meh_ikind_U =
  | MIUChar
  | MIUInt
  | MIUShort
  | MIULong
  | MIULongLong

(* all C integer types supported by meh *)
type meh_ikind =
  | MikindS of meh_ikind_S (* meh C signed int *)
  | MikindU of meh_ikind_U (* meh C unsigned int *)

(* all data types of meal representing a number *)
type meal_number_type =
  | MTNi of meh_ikind (* meal C int *)
  | MTMpZ
  | MTMpQ

(* all data types of meal representing an integer *)
type meal_integer_type =
  | MTi of meh_ikind
  | MTMpZ

type meal_signed_number_type =
  | MTiS of meh_ikind_S
  | MTMpZ
  | MTMpQ

(* all data types of meal *)
type meal_type =
  | MTNumber of meal_number_type
```

```
  | MTBool

(* the actual data *)
type meal_data =
  | Mi of meh_ikind * Integer.t
  | MMpZ of Integer.t
  | MMpQ (*of ??? *)
  | MBool of Bool.t
  | Mexpr of meal_type * Cil_types.exp

and meal_operation =
  (* Operations on stack *)
  | Mpush of meal_data
  | Massert_without_msg (*POP a bool + call assert*)
  | Massert_with_msg of Cil_types.location * string * Smart_stmt.
     annotation_kind * Cil_types.kernel_function
  (* predicate -> predicate *)
  | MNOT
  (* term -> predicate *)
  | MNNL of meal_number_type
  (* term -> term *)
  | MBCL of meal_integer_type (*bitwise complement*)
  | MOPP of meal_signed_number_type (*opposite (change sign)*)
  (* (term, term) -> term *)
  | MADD of meal_number_type
  | MSUB of meal_number_type
  | MMUL of meal_number_type
  | MDIV of meal_number_type
  | MMOD of meal_integer_type
  | MBND of meal_integer_type
  | MBOR of meal_integer_type
  | MBXR of meal_integer_type
  | MSHR of meal_integer_type * meal_integer_type * meal_integer_type
  | MSHL of meal_integer_type * meal_integer_type * meal_integer_type
  (* (predicate, predicate) -> predicate *)
  | MAND
  | MOR
  | MXOR
  | MIMPLIES
  | MIFF
  (* (term, term) -> predicate *)
  | MEQ of meal_number_type * meal_number_type
  | MNEQ of meal_number_type * meal_number_type
  | MLTE of meal_number_type * meal_number_type
  | MGTE of meal_number_type * meal_number_type
  | MLT of meal_number_type * meal_number_type
  | MGT of meal_number_type * meal_number_type


(** what about      | MealPOP of mealType ??? **)
```

Table des matières