Université Paris Cité

École Doctorale 386 — Sciences Mathématiques de Paris Centre

Inria

Formal Verification of Heap Space Bounds under Garbage Collection

A Space Tale from the Stars

Alexandre MOINE

Thèse de doctorat d'INFORMATIQUE

dirigée par Arthur CHARGUÉRAUD et par François POTTIER

présentée et soutenue publiquement le 20 septembre 2024 devant un jury composé de :

Robbert KREBBERS	Associate Professor, Radboud University	(rapporteur)
Magnus MYREEN	Professor, Chalmers University of Technology	(rapporteur)
Delia KESNER	Professeure des universités, Université Paris Cité	(examinatrice)
Azalea RAAD	Associate Professor, Imperial College London	(examinatrice)
Yannick ZAKOWSKI	Chargé de recherche, Inria	(examinateur)
Arthur CHARGUÉRAUD	Directeur de recherche, Inria	(directeur)
François POTTIER	Directeur de recherche, Inria	(directeur)

Abstract

This thesis addresses the problem of reasoning about the heap space usage of concurrent programs in the presence of tracing garbage collection. While it is clear where space is consumed—that is, at allocation points—it is far more difficult to understand where space is recovered. Indeed, in the presence of tracing garbage collection, deallocation points are implicit: there is no syntactic "free" operation. Instead, from time to time, the garbage collector will deallocate *unreachable* memory blocks—that is, blocks that are not transitively reachable from the locations that are *roots* of the program that remains to execute. Hence, a central question of this thesis is: how to reason about unreachability?

To answer this question, we propose IrisFit, a Separation Logic built on top of the Iris framework. In order to track the reachability of memory blocks, IrisFit equips each allocated location with a *pointed-by-heap* assertion and a *pointed-by-thread* assertion, recording which reachable memory blocks point to this location and which threads hold this location as a root, respectively. Moreover, IrisFit makes use of *space credits*, a purely logical device that accounts for the available free space. Space credits are consumed by allocations and produced when the user proves a memory block unreachable.

This thesis points out a fundamental difficulty in the analysis of the worst-case heap space complexity of concurrent programs in the presence of tracing garbage collection. Indeed, if garbage collection phases and steps of the program's threads can be arbitrarily interleaved, then there exist undesirable scenarios in which a root held by a sleeping thread prevents a possibly large amount of memory from being freed. To remedy this problem, we propose two language features, namely *protected sections*, where garbage collection is disabled, and *polling points*, instructions that block the current thread if garbage collection has been requested by another thread. Protected sections can be exploited by the programmer to eliminate undesirable scenarios and thereby obtain better worst-case heap space complexity. Polling points can be inserted by the compiler in order to guarantee that no thread will indefinitely wait for the garbage collector to run.

We extend IrisFit to reason on protected sections. We prove that IrisFit offers both a safety guarantee (programs cannot crash and cannot exceed an initial heap space bound) and a liveness guarantee after automatic insertion of polling points (every memory allocation request is satisfied after a bounded number of steps by other threads and the garbage collector).

We illustrate the use of IrisFit via a number of case studies. In particular, we verify standard lock-free data structures such as Treiber's stack and Michael and Scott's queue, adequately decorated with protected sections. Moreover, we prove correct an encoding of closures—concrete heap-allocated data structures implementing the abstract concept of a first-class function.

The proof techniques we develop in this thesis are intended to apply to programs written in widespread languages such as OCaml and Java. IrisFit and all the presented results are mechanized in the Coq proof assistant.

Keywords Heap Space, Tracing Garbage Collection, Concurrency, Formal Verification, Separation Logic.

Résumé

Cette thèse aborde le problème du raisonnement formel sur l'utilisation de l'espace de tas (*heap space*) des programmes concurrents en présence d'un glaneur de cellules (*garbage collector*). Alors qu'il est facile d'identifier où l'espace est consommé, c'est-à-dire les points d'allocation, la présence d'un glaneur de cellules rend délicat de déterminer où l'espace est libéré, les points de désallocation étant implicites. En effet, le glaneur de cellules peut s'exécuter à n'importe quel moment pour désallouer des blocs mémoires *inaccessibles*, c'est-à-dire les blocs qui ne sont pas transitivement accessibles depuis les adresses racines (*roots*) du code restant à exécuter. Une question centrale à cette thèse est donc: comment raisonner sur l'inaccessibilité ?

Pour répondre à cette question, nous proposons IrisFit, une nouvelle logique de séparation construite au-dessus du cadre logique Iris. IrisFit garde trace de l'accessibilité des blocs mémoire en équipant chaque adresse d'une assertion *pointed-by-heap* et d'une assertion *pointed-by-thread*, enregistrant respectivement quel bloc mémoire accessible pointe vers cette adresse et quel fil (*thread*) considère cette adresse comme une racine. De plus, IrisFit exploite la notion de *crédits-espace*, un outil purement logique qui garde trace de la quantité d'espace libre disponible. Des crédits-espace sont consommés par les allocations et peuvent être récupérés en prouvant qu'un bloc mémoire est inaccessible.

Au cours de cette thèse, nous avons identifié une difficulté fondamentale dans l'analyse de la complexité en espace de tas au pire cas des programmes concurrents en présence d'un glaneur de cellules. En effet, si l'exécution du glaneur de cellules et la réduction des fils peuvent s'entrelacer de manière arbitraire, alors il existe des scénarios indésirables dans lesquels une racine retenue par un fil endormi empêche la désallocation d'une quantité de mémoire possiblement grande. Pour remédier à ce problème, nous proposons deux nouvelles constructions: les *sections protégées*, dans lesquelles le glaneur de cellules n'a pas le droit de s'exécuter, et les *points de polling*, des instructions qui bloquent le fil courant si un autre fil requiert l'exécution du glaneur de cellules. Les sections protégées peuvent être exploitées par le programmeur pour éliminer les scénarios indésirables décrits ci-dessus, permettant ainsi d'obtenir une meilleure complexité en espace de tas au pire cas. Les points de polling, quant à eux, peuvent être insérés par le compilateur pour garantir qu'aucun fil n'attend indéfiniment que le glaneur de cellules s'exécute.

Nous étendons IrisFit pour permettre de raisonner sur les sections protégées. Nous prouvons qu'IrisFit offre une garantie de sûreté (les programmes vérifiés ne peuvent pas produire une erreur et n'excèdent pas une borne donnée en espace de tas) et une garantie de vivacité (il existe une stratégie d'insertion des points de polling dans les programmes vérifiés garantissant que chaque requête d'allocation sera satisfaite après un nombre de pas borné).

Nous illustrons l'utilisation d'IrisFit avec plusieurs études de cas. En particulier, nous vérifions des structures de données concurrentes et non-bloquantes standards comme la pile de Treiber et la file de Michael et Scott. Nous montrons qu'en augmentant le code de ces structures avec des sections protégées, nous obtenons des implémentations qui garantissent la complexité en espace de tas intuitivement attendue. De plus, nous prouvons correcte une implémentation des *fermetures*, des structures de données concrètes allouées sur le tas implémentant le concept abstrait de fonction de première classe.

Les techniques de raisonnement que nous développons dans cette thèse sont destinées à s'appliquer à des langages répandus comme OCaml et Java. IrisFit ainsi que tous les résultats présentés sont formalisés dans l'assistant de preuve Coq.

Mots clés Espace de tas, Glaneur de cellules, Concurrence, Vérification formelle, Logique de séparation.

Résumé substantiel

Cette thèse est rédigée en anglais. Je fournis ici un résumé substantiel en français. Ce résumé correspond à la traduction des sections 1.3 et 1.4.

Vérification de bornes en consommation de ressources

Vérification de programme L'objectif le plus courant de la vérification d'un programme est d'établir sa *sûreté* et sa *correction fonctionnelle*, c'est-à-dire que l'exécution du programme ne produit pas d'erreur et calcule un résultat correct. Dans le domaine de la vérification déductive [Filliâtre, 2011], un programme est vérifié avec l'aide d'une *logique de programme*, c'est-à-dire un ensemble de règles de déduction dont la correction a été démontrée une fois pour toute. La logique de séparation Reynolds [2002] et la logique de séparation concurrente [Brookes et O'Hearn, 2016; O'Hearn, 2019; Jung et al., 2018b] sont des exemples de logiques de programme qui permettent un raisonnement compositionnel (c'est-à-dire, qui permettent de raisonner sur les différents composants du programme de manière isolée) en présence de fonctionnalités avancées comme l'allocation dynamique de mémoire, l'état mutable et la concurrence avec mémoire partagée.

Vérification de bornes en ressource Au-delà de la sûreté et de la correction fonctionnelle, il peut être désirable d'établir des bornes sur la consommation de ressources des programmes, c'est-à-dire montrer que l'exécution d'un programme ne consomme pas plus qu'une certaine quantité de ressource prédéfinie. En effet, un programme qui requiert une quantité trop grande de *temps* peut être problématique. Un programme qui requiert une quantité trop grande d'*espace de pile* produit un débordement de pile (*stack overflow*).

En supposant que l'on soit capable de dire où la ressource concernée est consommée et produite, et en quelle quantité, établir une borne sur la consommation de cette ressource peut être réduit à raisonner à propos de sureté. Pour ce faire, on peut construire une variante du programme instrumentée avec un *compteur de ressources*, c'est-à-dire une variable globale dont la valeur (un entier positif ou nul) indique quelle quantité de ressource est disponible.

Dans ce programme instrumenté, on place des instructions qui produisent une erreur si la valeur du compteur est négative. Si l'on est capable de montrer que ce programme instrumenté est sûr, c'est-à-dire que son exécution ne produit jamais d'erreur, alors on a effectivement établi une borne sur la consommation de ressource du programme original.

Le principe d'un compteur de ressource a été utilisé de nombreuses fois, avec des cadres variés pour établir la sûreté. Par exemple, Crary et Weirich [2000] exploitent un système de types dépendants ; Aspinall et al. [2007] exploitent une logique de programme dans le style de VDM ; Carbonneaux et al. [2015] exploitent une logique de Hoare ; He et al. [2009] exploitent une logique de séparation. La manière dont on résonne à propos de la valeur du compteur dépend du cadre choisi. Dans l'approche la plus directe, la valeur du compteur est explicitement décrite par les préconditions et postconditions de chaque fonction. C'est le cas, par exemple, de l'approche de He et al. [2009], où deux compteurs distincts sont utilisés pour mesurer l'espace de pile et l'espace de tas.

Dans des approches plus élaborées rendues possibles par la logique de séparation, le compteur n'est pas vu comme une valeur entière, mais comme un sac de *crédits*, qui peuvent être *possédés* de manière individuelle. Cela enlève le besoin de référer la valeur absolue du compteur. À la place, la spécification d'une fonction peut indiquer que cette fonction requiert un certain nombre de crédits et en produit un certain autre nombre.

Vérification de bornes en espace de tas, sans glaneur de cellules Un langage de programmation sans glaneur de cellules offre une instruction de désallocation explicite. Il est donc facile de dire où l'espace de tas est consommé et produit : une instruction d'allocation consomme la quantité d'espace qu'elle reçoit en argument ; une instruction de désallocation récupère l'espace occupé sur le tas par le bloc qui est sur le point d'être désalloué.

Dans un tel cadre, la logique de séparation traditionnelle étendue avec des *crédits-espace* peut être utilisée pour vérifier des bornes en espace de tas. À notre connaissance, une telle variante de la logique de séparation n'existe pas dans la littérature. Néanmoins, le travail d'Hofmann sur le langage de programmation typé LPFL [2000] peut-être vu comme un précurseur de cette idée. LPFL a une instruction d'allocation et une instruction de désallocation explicite, qui respectivement consomme et produit des valeurs d'un type linéaire, noté $\diamond$, dont les habitants se comportent comme des crédits-espace.

Vérification de bornes en espace de tas, en présence d'un glaneur de cellules En présence d'un glaneur de cellules, comment raisonner à propos de l'espace de tas ? Dans ce cadre, le langage de programmation n'a pas d'instruction de désallocation : il n'y a donc pas de point de programme évident où l'espace peut être récupéré ! Un glaneur de cellules (GC) peut être invoqué à des points arbitraires dans le temps et peut désallouer n'importe quel sous-ensemble de blocs *inaccessibles*. Un bloc inaccessible est un bloc qui n'est pas accessible depuis les *racines* en suivant un *chemin* dans le tas. Par conséquent, raisonner à propos de l'espace de tas en présence d'un GC requiert de raisonner d'une manière ou d'une autre à propos des racines et de l'inaccessibilité.

Madiot et Pottier [2022] font un premier pas vers la résolution de ce problème. Ils étendent la logique de séparation avec plusieurs concepts. Pour garder trace de l'espace libre, ils utilisent des crédits-espace. Ils considèrent la désallocation de la mémoire comme une opération logique : la personne qui vérifie le programme décide à quel moment cette opération doit être utilisée et quels blocs de mémoire doivent être supprimés. Cette décision est sujette à une obligation de preuve : un bloc mémoire ne peut être logiquement désalloué que s'il est prouvé inaccessible. Malheureusement, le concept d'inaccessibilité n'est pas local : ce dernier ne peut pas être facilement exprimé en termes d'assertions de la logique de séparation. Madiot et Pottier reformulent donc cette obligation de preuve comme suit : un bloc de mémoire peut être logiquement désalloué s'il n'a pas de prédécesseurs et s'il n'est lui-même pas une racine. Pour garder trace des prédécesseurs de chaque bloc mémoire, ils utilisent des assertions *pointed-by*. Pour savoir quels blocs sont des racines, ils concentrent leur attention sur un langage de bas niveau, où la pile est explicitement représentée dans le tas comme une collection de "cellules de pile". Ainsi, un bloc est une racine si et seulement s'il s'agit d'une cellule de pile.

Dans un travail publié au cours de cette thèse [Moine et al., 2023], nous passons les idées de Madiot et Pottier à l'échelle d'un langage de haut niveau, où la pile est implicite. Nous introduisons des assertions *Stackable* pour garder implicitement trace des "racines invisibles", c'est-à-dire des adresses qui sont des racines parce qu'elles dans le cadre de la pile (*stack frame*) d'un appelant indirect.

Aucun de ces articles ne se concentre sur la concurrence. Madiot et Pottier [2022] supportent théoriquement la concurrence, mais seulement pour un langage de bas niveau dans lequel la pile est explicite. De surcroît, Madiot et Pottier ne fournissent aucun exemple dans lequel la concurrence apparait. Moine et al. [2023] ne supportent pas la concurrence. De par sa conception, leur assertion *Stackable* garde trace d'une seule pile. L'étendre à la prise en charge de plusieurs piles, comme il en existe dans un cadre concurrent, n'est à priori pas simple.

Contributions et vue d'ensemble

Nous présentons maintenant une vue globale des contributions de cette thèse et proposons une vue d'ensemble de chaque chapitre.

Contribution n°1 : IrisFit, une logique de séparation concurrente pour l'espace de tas La contribution principale de cette thèse est IrisFit, la première logique de programme qui permet d'établir la sûreté, la correction fonctionnelle, ainsi que des *bornes au pire cas en espace de tas* pour des programmes concurrents en présence d'un glaneur de cellules. De plus, IrisFit permet du *raisonnement compositionnel*, c'est-à-dire que IrisFit permet de vérifier chaque composante du programme en isolation.

IrisFit propose des crédits-espace pour garder trace de l'espace libre. Nous utilisons et améliorons l'assertion *pointed-by-heap* de Madiot et Pottier [2022] pour garder trace des pointeurs du tas vers le tas et introduisons l'assertion *pointed-by-thread* pour garder des traces des pointeurs de la pile vers le tas. Toutes les deux, ces assertions permettent de prouver qu'une adresse est inaccessible. Lorsque l'utilisateur est capable de le faire, nous proposons une règle de *désallocation fantôme*, à la manière de Madiot et Pottier [2022]. Intuitivement, cette règle consomme les assertions pointed-by-heap et pointed-by-thread démontrant l'inaccessibilité, et produit des crédits-espace.

En comparaison avec la logique de séparation traditionnelle, et afin de garder trace de l'accessibilité des adresses mémoire, les règle de raisonnement ont des obligations de preuve supplémentaires. Néanmoins, nous sommes capables d'offrir à l'utilisateur des règles de raisonnement simplifiées *via* deux mécanismes que nous introduisons. Premièrement, grâce à un *mode* dédié, promettant de ne pas désallouer de blocs mémoire lors au cours d'une portée syntaxique, l'utilisateur obtient une logique de séparation quasi standard. Deuxièmement, nous offrons à l'utilisateur la possibilité de se rappeler qu'une obligation de preuve a déjà été satisfaite à l'aide d'un *souvenir*, ce qui permet de satisfaire automatiquement cette obligation si elle réapparait plus tard dans la preuve.

Contribution n°2 : sections protégées et points de polling Si l'on ne fait pas attention, en présence d'un glaneur de cellules, les programmes concurrents peuvent présenter une complexité en espace du tas sous-optimale dans le pire des cas, c'est-à-dire une complexité pire que ce que l'on pourrait imaginer naïvement. Nous proposons des constructions de langage de programmation qui permettent au programmeur d'éliminer les scénarios problématiques que nous avons découvert. Les nouvelles fonctionnalités que nous proposons comprennent des *sections protégées*, des sections du code où le glaneur de cellules est désactivé, et des *points de polling*, des instructions qui bloquent le thread en cours si l'exécution du glaneur de cellules a été demandé par un ou plusieurs autres threads. Le programmeur peut exploiter la présence de sections protégées pour obtenir de meilleures bornes en espace de tas au pire des cas. Nos sections protégées et nos points de pollings s'inspirent de mécanismes présents dans des implémentations de langages réels, tels que les "safe points" d'Ocaml 5. Cependant, nous pensons que les deux constructions que nous proposons se comportent mieux et introduisent une distinction importante entre une construction qui est insérée par le programmeur et qui est nécessaire pour garantir une bonne complexité de l'espace du tas dans le pire des cas (à savoir, les sections protégées), et une construction qui peut être automatiquement insérée par le compilateur et qui est nécessaire pour garantir la vivacité (à savoir, les points de polling).

Nous dotons IrisFit de règles de raisonnement pour les sections protégées, permettant une forme de désallocation logique "à l'avance". Ces règles permettent de désallouer logiquement une adresse qui est accessible à partir des sections protégées actuelles, mais avec l'obligation de prouver que cette adresse est inaccessible au moment où les sections protégées se terminent.

Contribution n°3 : correction de la logique IrisFit Nous démontrons la *correction* (*soundness*) d'IrisFit. Plus spécifiquement, nous établissons non seulement un théorème de *sûreté* (*safety*), garantissant qu'un programme vérifié ne peut pas produire d'erreur ni dépasser une borne initiale en espace de tas, mais aussi un théorème de *vivacité* (*liveness*), garantissant qu'il existe une stratégie d'insertion des points de polling dans les programmes vérifiés telle qu'aucun fil n'attende indéfiniment que le glaneur de cellules s'exécute.

Contribution n°4 : impact des fermetures en présence d'un glaneur de cellules Dans la plupart des langages de programmation fonctionnels, comme OCaml, les fonctions avec des variables libres sont compilées en *fermetures* (*closures*) [Landin, 1964; Appel, 1992].

Une fermeture est un bloc alloué sur le tas qui pointe vers le code de la fonction ainsi que la valeur de ses variables libres. Ainsi, les fermetures occupent de l'espace de tas et participent à l'accessibilité de leur environnement (c'est-à-dire, les variables libres de la fonction qu'elle représente). Les fermetures apparaissent partout dans les programmes fonctionnels : comprendre leur consommation d'espace est important. Pour ce faire, nous programmons la passe de compilation connue sous le nom de *closure conversion*, qui transforme les notations de haut niveau pour les fonctions en code de bas niveau allouant et manipulant les blocs de fermeture. Ensuite, nous prouvons des règles de raisonnement pour la création et l'appel de fermeture. Nous proposons d'abord une interface logique de bas niveau, révélant le code concret de la fonction implémentée. À partir de cette interface de bas niveau, nous dérivons une interface de haut niveau qui cache le code de la fonction et ne révèle que sa spécification.

Contribution n°5 : études de cas Pour illustrer IrisFit, nous proposons une variété d'études de cas. Pour chacune d'entre elles, nous établissons non seulement la correction fonctionelle, mais aussi des bornes en espace de tas pour chacune des fonctions impliquées.

Premièrement, IrisFit permet de raisonner sur des programmes séquentiels (c'est-à-dire, des programmes qui n'utilisent pas de primitives de la concurrence). Nous vérifions une implémentation des listes chaînées immutables ainsi que plusieurs fonctions sur cette structure de données. En particulier, nous vérifions une implémentation de la concaténation des listes en style par passage de continuation (*continuation-passing style*, CPS), illustrant les fermetures. Pour démontrer la compositionnalité de notre approche, nous vérifions plusieurs implémentations des piles et expliquons comment les composer. Enfin, nous démontrons la capacité de IrisFit de raisonner sur les structures de données circulaires au travers de la vérification d'une liste circulaire simplement chaînée.

Deuxièmement, IrisFit permet de raisonnement sur des programmes concurrents subtils. Nous proposons un encodage de la primitive *fetch-and-add* (FAA) avec des sections protégées. Nous vérifions un compteur concurrent monotone, implémenté avec une paire de fermetures partageant une référence mutable. Nous proposons une bibliothèque pour une forme de concurrence structurée appelée *async/finish*. Nous présentons ensuite deux études de cas de structures de données non-bloquantes. Nous vérifions la correction et établissons une borne en espace de tas pour la pile de Treiber [1986] et la file de Michael et Scott [1996]. Afin d'en établir les bornes en espace de tas intuitives, nous montrons comment ces structures doivent être corrigées avec des sections protégées.

Structure de ce document Dans le chapitre 2, nous introduisons les idées principales de LambdaFit, le langage formel que nous étudions, et d'IrisFit, la logique de programme associée. Dans le chapitre 3, nous expliquons avec l'exemple de la pile de Treiber pourquoi les sections protégées sont nécessaires pour garantir des bornes intuitives en espace de tas. Après ces intuitions, nous présentons dans le chapitre 4 la syntaxe et la sémantique formelle de LambdaFit. Nous plongeons ensuite dans IrisFit. Dans le chapitre 5, nous en présentons les assertions et dans le chapitre 6 nous en présentons les règles de raisonnement. Nous justifions ensuite la correction d'IrisFit. Dans le chapitre 7, nous exposons nos théorèmes de sûreté et de vivacité, et expliquons qu'ils découlent d'un théorème de correction principal. Nous esquissons la preuve de ce dernier dans le chapitre 8. Dans le chapitre 9, nous expliquons notre encodage des fermetures et les règles de raisonnement associées. Dans le chapitre 10, nous présentons la technique des triplets *avec souvenir*, simplifiant certaines obligations de preuves. Nous présentons ensuite nos études de cas séquentielles dans le chapitre 11 et nos études de cas concurrentes dans le chapitre 12. Dans le chapitre 13, nous couvrons les travaux connexes. Dans le chapitre 14, nous concluons et présentons des pistes de travaux futurs.

Remerciements

Je tiens tout d'abord à remercier mes deux directeurs de thèse, Arthur Charguéraud et François Pottier: votre duo a incarné tout ce qu'un doctorant peut espérer. Merci pour votre disponibilité et pour votre bienveillance à toute épreuve. Merci à toi, Arthur, pour ton implication et ton soutien sans faille qui ont fait fi de la distance. Merci d'avoir non seulement guidé mes pas dans mes premières vraies preuves formelles, mais aussi de m'avoir appris à en extraire le suc pour en parler dans un article. Merci à toi, François, pour ton enthousiasme scientifique qui m'a motivé jour après jour. Merci de m'avoir transmis une (petite) partie de ta connaissance de la science des autres, ainsi que de m'avoir appris les principes d'une rigueur salvatrice.

I want to thank the members of my jury: your participation is an honor. First and foremost, thanks to Robbert Krebbers and Magnus Myreen who accepted to review this thesis. Thank you for your detailed feedback and enthusiasm. Thanks to Azalea Raad, et merci à Yannick Zakowski, for participating in this jury. Un merci particulier à Delia Kesner d'avoir accepté de participer à ce jury: j'ai suivi ses cours en master et son enseignement a contribué à mon envie de faire de la recherche.

J'ai eu la chance de faire ma thèse dans une équipe extraordinaire, Cambium, que je tiens à remercier chaleureusement. Merci tout d'abord aux membres permanents que je n'ai pas encore cité: Damien Doligez, Yannick Forster, Xavier Leroy, Jean-Marie Madiot, Luc Maranget et Didier Rémy. Les discussions autour du café vont me manquer, des détails de conception des chars soviétiques aux critiques des derniers papiers acceptés à POPL. Merci à Florian Angeletti pour sa disponibilité, que ce soit pour parler des détails du compilateur OCaml ou du concert de la veille. Merci à Hélène Milome pour son aide administrative précieuse. Merci aussi aux ex-Cambium/Gallium. Merci à Armaël Guéneau et Jacques-Henri Jourdan pour les discussions quand nous nous sommes croisés. En plus de ces discussions, merci à Gabriel Scherer, avec qui j'ai eu le plaisir de collaborer.

Je tiens aussi évidemment à remercier les membres non-permanents de l'équipe, mes compagnons de galères. Merci à Clément Allain pour les échanges irisiens et la collaboration par preuve interposée. Merci à Clément Blaudeau pour le soutien avant les deadlines et pour sa science qui module jusqu'à la géopolitique. Merci à Basile Clément pour la collaboration et les parties de jeu de rôle occasionnelles. Merci à Nathanaëlle Courant pour les discussions enjouées. Merci à Chiara Daini d'avoir apporté un autre violon et un peu de véritable informatique chez Cambium. Merci à Paulo Emílio de Vilhena d'avoir accompagné mes premiers pas de thésard et pour les détours par l'Opéra. Merci à Rémy Seassau pour les moments de décompression, plus que nécessaire, après le travail. Thanks to Irene Yoon, whose arrival in the late office C332 brought laughter and pictures of cats.

During the second part of my PhD, I had the chance to collaborate remotely with two amazing people, Sam Westrick and Stephanie Balzer. Thank you, Sam, for trusting an intuition discussed on a small table at Ljubljana's castle and for the cool collaboration that followed. Thank you, Stephanie, for the awesome paper co-writing experience, for your advice and for your liveliness.

Enfin, je tiens à remercier ceux sans qui je n'aurais jamais fait de recherche. Merci aux équipes enseignantes de l'Université ~~Paris-Diderot de Paris~~ Paris-Cité pour tout ce qu'elles m'ont appris. Merci en particulier à Victor Lanvin d'avoir été un excellent chargé de TD et à Yann Régis-Gianas de m'avoir fait écrire mon premier article scientifique. I also want to thank Andrey Mokhov for supervising me during a Google Summer of Code internship: you started my academic life.

Heureusement, la vie d'un thésard ne se résume pas au monde universitaire. Je tiens à remercier tous mes ami·e·s, qui n'ont cessé de me soutenir et avec qui j'ai partagé des moments inoubliables au cours de ces trois ans.

Merci aux copains du début. Merci à Arnaud, expert en construction aérospatiale. Merci à Sacha, à qui certaines des musiques de ce manuscrit ne seront pas étrangères.

Merci aux ami·e·s de la math-info: vous n'avez fait que renforcer mon envie de faire l'informatique. Merci à Maxime pour ces discussions dont on ne veut pas qu'elles s'arrêtent, que ce soit sur Zoom ou au bord de l'Adriatique. Merci à Marie pour son goût partagé des vidéos avec des animaux mignons et pour les discussions sur le reste de la vie. Merci à Étienne pour le soutien et l'émulation OCamlesque mutuelle.

Merci aux ami·e·s d'après la licence, ceux-de-chez-l'IRIF-d'en-face, qui ont fait le même choix discutable de faire une thèse. Merci à Victor pour l'entraide pendant ce M2 sous COVID et pour les discussions passionnantes sur les choses importantes de la vie: le λ-calcul, l'amour, et le mélange des deux. Merci à Klara pour les expos parisiennes. Merci à Vincent, Thiago et Clément pour les soirées mémorables.

Merci aux ami·e·s qui me rappellent heureusement qu'il existe d'autres choses que la recherche et l'informatique. Merci au "club de Laives", Alaric, Antoine, Aymeric, Charlie, Laura, Luc, Néel, Pierre, Stéphane, ainsi que plus récemment Circé et Merlin, pour la découverte de tous ces cocktails "merveilleux" et les parties de jeux de société. Merci aux trois du dé à 5 faces. Merci à Clément pour sa présence et ses scénarios shaaniques. Merci à Émiland, dottore in ingegneria dell'informazione, pour les aventures qui ont impliqué successivement une forge, un four à bois et des bourgeons de sapins. Merci à Antonin, doctorant en chimie physique, pour la complicité que l'on partage depuis plus de quinze ans: que L. le L. et Pierre Loti m'en soient témoins, ça n'est que le début.

Merci à tous les autres, celles et ceux que je ne mentionne pas par manque de place: je ne vous oublie évidemment pas.

Enfin, merci à ma famille: les mots de la langue française sonnent bien creux pour vous exprimer toute ma gratitude. Merci à mes grands-parents de m'avoir transmis et de continuer de me transmettre bien plus qu'ils ne le pensent. Merci à mes oncles et tantes pour leur soutien inconditionnel. Merci en particulier à Emmanuel pour les discussions depuis le bout du monde, à Sylvaine pour les rires, et à Jean-Claude et Stéphanie pour leur accueil et tout ce qu'ils m'ont appris. Merci à mes cousins et cousines. Merci à Nicolas, docteur en STAPS (dont le manuscrit contient environ 4 fois plus de pages que le présent document!), pour ses encouragements de mes premières parties de jeux-vidéos à cette thèse. Merci à Julia pour sa présence constante, pour ses fêtes toujours mémorables, et de m'avoir appris à déceler l'invisible du vol des oiseaux. Merci aux chats, Aaron, Moumou et Keta, pour les ronrons au cœur de l'orage.

Merci à mes parents, pour absolument tout. Merci pour votre soutien pendant cette thèse et pendant mes études. Surtout, et c'est le plus important, merci pour l'enfance que vous m'avez offerte. Merci à toi, maman, docteure en sociologie, de m'avoir montré que faire une thèse relevait du domaine du possible. Merci à toi, papa, de m'avoir appris qu'en travaillant, j'y arriverais sans aucun doute. Enfin, merci à toi, Cécile, ma sœur jumelle, avec qui j'affronte et je savoure ce monde depuis le début.

Merci à tous, du fond du cœur.
Thank you all, from the bottom of my heart.

Contents

1 Introduction **15**
- 1.1 General Concepts . . . 16
- 1.2 Proving Programs Correct . . . 19
- 1.3 Verification of Resource Bounds . . . 23
- 1.4 Contributions and Overview . . . 24
- 1.5 Research Output: Publications and Mechanization . . . 26

2 Key Ideas **29**
- 2.1 Roots and Garbage Collection . . . 29
- 2.2 Maximum Heap Size and Blocking Memory Allocation . . . 30
- 2.3 Protected Sections . . . 31
- 2.4 Polling Points . . . 31
- 2.5 A Concurrent Separation Logic for Heap Space . . . 32
- 2.6 Closures . . . 33

3 Why Treiber's Stack Needs Protected Sections **35**
- 3.1 Naive Implementation of Treiber's Stack . . . 35
- 3.2 Space Usage of Treiber's Stack without Protected Sections . . . 36
- 3.3 Space Usage of Treiber's Stack with Protected Sections . . . 37

4 Syntax and Semantics of LambdaFit **39**
- 4.1 Syntax . . . 39
- 4.2 Memory Blocks, Stores, and Heap Size . . . 40
- 4.3 Thread Pools and Configurations . . . 40
- 4.4 The Head Reduction Relation . . . 42
- 4.5 The Step Relation . . . 42
- 4.6 The Garbage Collection Relation . . . 42
- 4.7 The Action Relation . . . 43
- 4.8 Enabled Actions . . . 43
- 4.9 The Main Reduction Relation . . . 44

5 Program Logic: Assertions **45**
- 5.1 Triples . . . 45
- 5.2 Ghost Updates . . . 46
- 5.3 Points-to Assertions . . . 46
- 5.4 Sizeof Assertions . . . 46
- 5.5 Space Credits . . . 47
- 5.6 Pointed-By-Heap Assertions . . . 48
- 5.7 Pointed-By-Thread Assertions . . . 49
- 5.8 Inside and Outside Assertions . . . 50
- 5.9 Deallocation Witnesses . . . 51
- 5.10 Liveness-Based Cancellable Invariants . . . 52

6 Program Logic: Reasoning Rules **53**
- 6.1 Logical Deallocation . . . 53
- 6.2 Reasoning Rules for Terms . . . 55
- 6.3 Reasoning about Protected Sections . . . 56
- 6.4 Reasoning under Evaluation Contexts . . . 57
- 6.5 Locally Trading Trimming for a Simpler and More Powerful Bind Rule . . . 58
- 6.6 Logical Deallocation of Cycles . . . 59

7 Safety, Liveness and Core Soundness Theorems **61**
7.1 Safety . . . 61
7.2 Liveness . . . 63
7.3 The Oblivious Semantics and the Core Soundness . . . 65
7.4 Deriving Safety from Core Soundness . . . 66
7.5 Deriving Liveness from Safety . . . 67

8 Proof of the Core Soundness Theorem **71**
8.1 Definition of the Weakest Precondition Modality . . . 71
8.2 Auxiliary Definitions . . . 72
8.3 Resource Algebras . . . 74
8.4 State Interpretation and Definition of Assertions . . . 76
8.5 Proving the Core Soundness Theorem . . . 79

9 Closures **83**
9.1 Environments . . . 83
9.2 Closure Implementation . . . 84
9.3 Low-Level Closure API . . . 84
9.4 Low-Level Closure API: Implementation Details . . . 85
9.5 High-Level Closure API . . . 86
9.6 High-Level Closure API: Implementation Details . . . 87

10 Triples with Souvenir **89**
10.1 Those Who Cannot Remember the Past Are Condemned to Repeat It . . . 89
10.2 Internals of Souvenirs . . . 90

11 Sequential Case Studies **91**
11.1 Containers: A Generic Approach . . . 91
11.2 Linked Lists and Linked List Reversal . . . 92
11.3 Continuation-Passing Style . . . 94
11.4 Sequential Stacks . . . 95
11.5 A Circular Singly-Linked List . . . 97

12 Concurrent Case Studies **101**
12.1 Atomic Triples . . . 101
12.2 Fetch-and-Add . . . 102
12.3 A Concurrent Counter Object . . . 104
12.4 An Async/Finish Library . . . 106
12.5 Treiber's Stack . . . 108
12.6 Michael and Scott's Queue . . . 112

13 Related Work **117**
13.1 Polling Points . . . 117
13.2 Protected Sections . . . 118
13.3 Reasoning about Space without a GC . . . 118
13.4 Reasoning about Space with a GC . . . 119
13.5 Space-Related Results for Compilers . . . 120
13.6 Safe Memory Reclamation Schemes . . . 120
13.7 Disentanglement . . . 121

14 Conclusion **123**
14.1 Mechanization . . . 123
14.2 Perspectives . . . 124

List of Figures

1 An unsafe-for-space implementation of Treiber's stack . . . 35
2 Initial and problematic states of the scenario. . . . 36
3 A safe-for-space implementation of Treiber's stack . . . 37

4 LambdaFit: syntax . . . 40
5 The head reduction relation . . . 41
6 The step relation . . . 41
7 The garbage collection relation . . . 41
8 The action relation . . . 43
9 Enabled actions and auxiliary predicates . . . 44
10 The main reduction relation . . . 44

11 Structural reasoning rules . . . 46
12 Reasoning rules of the "*sizeof*" assertion . . . 47
13 Reasoning rules for space credits . . . 47
14 Reasoning rules for the pointed-by-heap assertion . . . 48
15 Reasoning rules for the pointed-by-thread assertion . . . 49
16 Reasoning rules for "*inside*" and "*outside*" assertions . . . 50
17 Reasoning rules for deallocation witnesses . . . 51

18 Syntax-directed reasoning rules, without Bind and rules for protected sections 54
19 Reasoning rules: protected-section-specific rules . . . 56
20 Reasoning rules: the Bind rule . . . 57
21 Reasoning rules: additional mode-specific rules . . . 58
22 Reasoning rules: logical deallocation . . . 59

23 Predicates used to state the soundness theorem . . . 62
24 Predicates used to state the liveness theorem . . . 64
25 The oblivious reduction relation and associated predicates . . . 65
26 Predicates for the liveness condition . . . 68

27 Definition of the weakest precondition (WP) modality . . . 72
28 Definition of the state interpretation predicate . . . 77
29 Definition of assertions . . . 77
30 Lemmas for the proof of the Core Soundness Theorem . . . 80

31 Macros for closure construction and invocation . . . 84
32 Low-level API for closures . . . 84
33 Definition of the predicate *Closure* . . . 86
34 High-level API for closures . . . 86
35 Definition of the predicate *Spec* . . . 87

36 Key reasoning rules for triples with souvenir . . . 89
37 Definition of triples with souvenir . . . 90

38 Code and specification of linked list reversal . . . 93
39 Internals of linked lists . . . 94
40 Code and specification of linked list concatenation in continuation-passing style 95
41 Specification of possibly-bounded sequential stacks . . . 96
42 Code and specification of circular singly-linked lists . . . 97

43 Internals of circular singly-linked lists 98

44 Code and specification of fetch-and-add 103
45 Code and specification of a concurrent monotonic counter 104
46 Internals of the concurrent counter 105
47 Code and specification of an async/finish library 107
48 Specification of Treiber's stack 109
49 Internals of Treiber's stack 111
50 Code and specification of Michael and Scott's queue 113

51 Pattern matching may extend the lifetime of variables 126
52 Impact of dead code elimination (DCE) on weak pointers 127
53 Impact of common sub-expression elimination (CSE) on weak pointers 127

Chapter 1

Introduction

Prokofiev, S. (1935).
Violin Concerto No. 2, Andante Assai.

Computer programs are written by us, fallible humans. As such, programs can have bugs. In a world where programs impact an ever-growing part of our lives, bugs are a plague. Indeed, famous examples shed light on the impacts of bugs at various times and places. For example, each due to a different bug, the Therac-25 radiation therapy machine killed several patients [Leveson and Turner, 1993], the Ariane 5 rocket crashed during its first flight [Le Lann, 1997], and the whole northeast of America suffered from a power outage [U.S.-Canada Power System Outage Task Force, 2004]. Bugs can also introduce security vulnerabilities that can have huge impacts. For example, Durumeric et al. [2014] estimate that the Heartbleed vulnerability, due to a bug, allowed attackers to remotely read the protected memory of more than a quarter of the one million most visited websites.

These observations yield an ingenuous yet central question:

How to ensure that a program has no bugs?

Let us analyze this question. First, we need to understand what is a bug. A bug can be defined as an *unexpected behavior* of a program. This definition implies that there exists a set of *expected behaviors* for the considered program: this set is called its *specification*. Second, we need to understand what it means to ensure the absence of bugs. A widely used approach for unveiling bugs is to *test* the desired program, looking for undesired behaviors. Yet, although testing is pervasively exploited, it has a fundamental drawback. Indeed, testing only detects certain undesired behaviors, but in general cannot ensure that there are *no* undesired behaviors. The reason is that the set of possible behaviors is generally extremely large, and testing all behaviors one by one would thus take an unreasonable amount of time. Another approach—that we take in this thesis—is to *prove*, through mathematical reasoning, that all behaviors of a program are desired. Such proof is conducted by investigating the source code, without executing it.

A more precise formulation of our first question is hence:

How to prove that a program satisfies its specification?

Such a question is not new and was arguably first formulated by Turing in his seminal article "Checking a large routine" [Turing, 1949; Morris and Jones, 1984]. However, verifying programs has proved to be very difficult in practice. Despite a considerable amount of attention being dedicated toward this goal since Turing's days, work remains to be done. In particular, at the time of writing, there remain large classes of specifications for which no satisfactory proof system exists. The present thesis proposes a proof system for such a class. In detail, this thesis proposes a program logic that allows for establishing heap space bounds—that is, an upper bound on the amount of memory that a program may require to execute—for programs that are written in a high-level, concurrent programming language with tracing garbage collection. Hence, the proposed techniques apply to program written in languages such as OCaml or Java.

In summary, this thesis focuses on answering the motivating question:

How to prove heap space bounds for concurrent programs under tracing garbage collection?

This motivating question involves numerous keywords. We start by defining them more precisely (§1.1). Then, we give a bird's-eye overview of the state of the art of program verification, focusing on the general area of program logics (§1.2). After setting up this context, we comment on pre-existing work related to establishing heap space bounds and justify our approach (§1.3). We then explain the answer we propose to our question and present our contributions (§1.4). We finally comment on the research output of our work: articles published and under review, as well as the mechanization in which we formalize all of our results (§1.5).

1.1 General Concepts

Specifications of Programs Specifying a program—that is, describing a set of desired behaviors for this program—is not an easy task. Usually, the specification concerns *functional correctness*: this property ensures that the program does not crash and computes the correct result. For example, we expect the calculator app on a smartphone to compute the correct mathematical result of the input expression. Indeed, if we ask a calculator "what is the result of $37 + 5$?", we expect an answer, and we expect this answer to be 42.

Yet, specifications go beyond functional correctness. For example, one might be interested in *security* properties: the program should not leak secrets. One might also be interested in *cryptographic* properties: an encrypted message should reveal as little information as possible about the original message. In this thesis, we are interested in the *resource consumption* of a program. A computer consumes two kinds of fundamental resources to run a program: *time* and *space*. The computer's processor needs time to execute basic computation steps. Moreover, this processor interacts with the computer's memory to read the input, and to store intermediate computations and the final result. Other resources are also of interest, like the energy consumption of various tasks, or the entropy usage—that is, the number of random bits used—of probabilistic programs.

The amount of resources needed by a computer to run a program and return its result is important information. For example, let us take time consumption. Recall the example of a calculator app: if we ask a calculator "what is the result of $37 + 5$?", we expect the correct answer to come *quickly*. Failing to deliver an answer in the expected time may have worse consequences than having to calculate in our head. For example, web servers are targets of *denial of service* attack (DoS). This type of attack floods the server with requests, in the hope that the server takes so much time to answer all these requests that it becomes unusable for other users. In some sense, space is even more critical than time. Indeed, a program that runs out of space usually has little choice but to abort altogether. Moreover, the time invested for this aborted computation is forever lost. Hence, running out of space is an undesirable scenario, and programmers should write their programs in order to avoid it.

This thesis focuses on proving *memory bounds*, that is, the assurance that a program will not run out of memory provided that a certain amount—the bound—is initially available. Let us first explain what we mean here by "memory".

A Reminder on Memory There are different types of memory in a computer that are organized into a *memory hierarchy*. This hierarchy is organized from bottom to top, in order of increasing efficiency but decreasing capacity. At the very bottom appears *external memory*, which include hard disk drives (HDD), solid-state drives (SSD), or USB keys. This type of memory is slow to operate, but usually not expansive and available in mass. At the very top of this hierarchy appears processor registers and caches. These memories are extremely fast but usually scarce. In between external memory and processor registers and caches is an intermediate, efficient yet relatively scarce *random access memory* (RAM), also called *main memory* or simply *memory*. In particular, the RAM is the memory the program interacts with. This thesis focuses on proving bounds for the RAM (memory hereafter) usage

of programs. Let us review how a program, and through it the programmer, interacts with the memory. A large part of the interaction between the program and the memory is automated. Indeed, programmers write their programs in *high-level languages* which are then compiled (or translated) to basic instructions that the CPU understands. In a vast majority of the implementations of high-level languages, a distinction is made between three parts of the memory, which are handled differently: the *read-only memory*, the *stack* and the *heap*. Of these three types of memory, only the heap may be directly managed by the programmer. The read-only memory and the stack are automatically managed.

The read-only memory usually contains the code of the program itself, as well as *static variables*, that is, data that is initialized before the program starts executing and that will never change. For example, the two characters chains `"You win"` and `"Try again"` of a video game may very well be static variables. Analyzing the size of the read-only memory is not hard, as it is physically observable in the size of the compiled program.

The stack as well as the heap, are types of memory that can be read or written. In order to understand the stack, recall that programs usually consist of a number of *functions*, with a designated main function that is executed when the program begins. Each function can have *local variables*, that is, short-lived memory that the programmer uses to store intermediate computations. These local variables are allocated when a function starts executing, and forever destroyed when the function returns (that is, terminates). A central aspect is that a function can call other functions. This scheme leads to a *call stack*, or simply *stack*. Each function call has a designated *stack frame*, each storing the local variables of the called function. Calling a new function automatically allocates a new stack frame for the callee, which is destroyed when the callee terminates. Allocating memory on the stack is desirable because the programmer does not have to worry about it. However, the stack has a usually rather small size. Exceeding this pre-defined size raises a "stack overflow" which triggers the end of the program. Nowadays, the stack is usually large. Thus, the stack overflow error is rarely encountered by accident, and often rather reflects a diverging computation, making an unbounded number of function calls. Because stack frames have a static size, proving that a program has no stack overflow conceptually amounts to analyze the nesting of function calls [Carbonneaux et al., 2014; Ferdinand et al., 2006].

The discipline of the stack is rigid, so very often, the programmer needs *dynamic memory allocation*, which is more flexible. Such an allocation is made in the so-called *heap*, which can be understood as the "far west" of memory, as there are very few rules for the heap. The program manually requests memory by making an *allocation*. In return, the program obtains a *memory location*, pointing to a *memory block* that can be read and written, passed around between functions, and crucially that outlives a function call.

Memory Management How heap-allocated memory is freed? There are a range of approaches for deallocating heap memory, called *memory management techniques*. At the extremities of this range, there are two techniques: *manual memory management* and *automatic memory management*.

In languages with manual memory management, like C and C++, the user manually deallocates heap memory by calling a primitive generally named `free`. Such a setting gives rise to various types of bugs: negligent programmers can provoke "double free" bugs—when the same location is deallocated twice—and "use after free" bugs—when a memory location is read or written after deallocation. Another more subtle and dreadful type of bug is *memory leaks*. Indeed, forgetting to free unused memory does not cause any immediate error. Yet, over lengthy executions, the amount of available memory keeps decreasing until it is exhausted.

In languages with automatic memory management, like Python, Java, C♯, F♯, Scala, Haskell and OCaml, there is no `free` operation. Indeed, heap objects that are guaranteed to not be used anymore are *automatically* reclaimed from time to time. This technique eliminates *a-priori* "double free" and "use after free" errors. However, memory leaks are not eliminated.

Indeed, a program could be written in such a way that it is impossible to automatically guarantee that a part of the memory is not used beyond the considered program point, preventing the automatic collection of this part of memory. Yet, memory leaks are far less common with automatic memory management than with manual memory management.

There are two approaches to implementing automatic memory management. The first approach is *static*: the compiler over-approximates for each program point the set of locations that will not be used anymore, and automatically inserts a `free` operation. This approach is exploited by region-based memory management [Tofte and Talpin, 1997; Tofte et al., 2004] or more recently, to some extent, by Rust and its `Box` type. However, statically approximating by which time a location will not be used can be difficult for the compiler, and static approaches can make approximations that are too coarse, leading to an excessive consumption of memory. The second and more widely-used approach is *dynamic*: the compiler inserts code that will determine at runtime if a location can be safely deallocated or not. This automatic collection is called *garbage collection*.

A first implementation of garbage collection is *reference counting* [Collins, 1960]. The idea is to maintain, for each location, the number of references (from the stack or from the heap) to this object. When this number reaches 0, the object becomes unreachable and can be freed. Reference counting is appealing: the overall idea is simple, and space is conceptually freed as soon as possible—that is, when the reference count drops to 0. Yet, reference counting has drawbacks: it can be slow, as reference counting incurs additional reads and writes to account for the reference count, and it cannot handle cycles as-is. Indeed, even if the programmer drops the last reference to a cycle of memory locations, each location constituting this cycle continues to bear a non-zero reference count.

A second implementation of garbage collection is *tracing garbage collection* [Jones et al., 2012], introduced for the Lisp language [McCarthy, 1960], and now used for languages like Python, Java, C♯, F♯, Scala, Haskell and OCaml. Tracing garbage collection is implemented via a *garbage collector* (GC). The GC is itself a program that is part of a more global *runtime system*, which consists of code added by the compiler for providing various services to the program. The idea is that, from time to time, and in particular if there is a need for free space, the runtime system pauses the program, launches the GC to deallocate *unreachable* memory locations—that is, locations for which the program has "lost their address". How can the GC compute such a set of unreachable locations? First, the GC defines a set of *roots*. At a first approximation, roots are the set of locations that occur in the stack—that is, the local variables. Then, the GC conceptually "walks through the heap" and computes the set of all locations that can be reached from the roots, following heap paths. Locations unreachable by this process cannot be accessed by the program anymore and can be safely deallocated. Indeed, the sole way for the program to access a location is either by having it at hand in a local variable or by loading it from a heap block, which must itself be reachable. Tracing garbage collection can handle arbitrary cycles and can be implemented efficiently.

This thesis focuses on *heap space bounds of programs under tracing garbage collection*. Establishing such bounds is not trivial: while it is still clear where space is needed (that is, when the program allocates memory), it is far less clear where space can be reclaimed by the GC. Indeed, whether the GC can free a memory location or not depends on the reachability of this location.

Concurrency Modern computers come with *multi-core* processors. In such a processor, several processing units, or *cores*, execute instructions *at the same time*, asynchronously. The programmer can hope, in certain circumstances, to divide the execution time by a factor P by using P cores. Crucially, cores can *share* memory, and exchange information while they execute. Yet, programming in such a *shared-memory concurrent* world, where cores may interfere with each other through the shared memory, is an extremely subtle task [Herlihy and Shavit, 2012].

In particular, the manual management of memory in a concurrent setting is famously difficult [Michael, 2004]. Indeed, one core could free a location that is in use by another core! In fact, it is so difficult to manually manage memory in a concurrent setting that programmers rely on *semi-automatic* memory management—that is, they explicitly call functions that will free memory "when it is safe to do so". These semi-automatic systems are called *safe memory reclamation* (SMR) schemes. SMR schemes include hazard pointers [Michael, 2004; Michael et al., 2023] and read-copy-update (RCU) [McKenney, 2004; McKenney et al., 2023].

In such a concurrent world, garbage collection comes to save the day: management of memory is invisible to the user, anyway! Indeed, the language implementor takes the burden of writing a concurrent GC, a notoriously difficult task [Jones et al., 2012]. Moreover, analyzing the reachability of objects becomes an even harder task: a formal analysis needs to take every core into account.

1.2 Proving Programs Correct

Since Turing's days, two approaches for proving programs correct have been followed: mostly-automatic proofs, and mostly-manual proofs.

Mostly Automatic Proof of Programs Many works have proposed automatic and semi-automatic techniques to check that a program does not crash, or satisfy more precise specifications. To list a few, these techniques include type systems [Pierce, 2002], static analysis with abstract interpretation [Rival and Yi, 2020], and model-checking [Clarke et al., 2018]. These techniques have proven to scale well to large programs and have been widely adopted. For example, one of the main selling points of the Rust language is its elegant type system. Moreover, various static analyses and model checking are used at large in the industry. Yet, because all of these techniques emphasize automation, they generally *lack expressivity*. Indeed, these techniques focus on common and simple programming patterns whose verification can be automated, rather than more complex patterns that require human help.

The present thesis rather focuses on a *mostly-manual approach*. In such an approach, a human reasons on the program by hand, using a set of established reasoning rules, which usually form a *program logic*. The use of program logics for proving programs correct requires expertise and is more time-consuming than automated approaches; yet, it allows for verifying more subtle specifications of more complex programs.

A Brief History of Program Logics Based on the seminal ideas of Floyd [1967], Hoare [1969] proposes one of the first program logics. In Hoare logic, each instruction is annotated by a *precondition*—the state before the command is executed—and a *postcondition*—the state after the command is executed. Such a specification takes the form of a *Hoare triple* $\{P\}\ t\ \{Q\}$ where P is the precondition, t is the program being verified, and Q is the postcondition. Hoare's axiomatic approach splits the work to be done. First, the designer of a program logic proves correct a set of *axioms*, let us say *reasoning rules*, which are generic and constitute the program logic itself. Later, the user of the program logic can instantiate the reasoning rules for the proof of the particular program they are trying to prove correct. However, early Hoare logics were designed for reasoning about the global state of programs and did not scale well to large or complex programs, where the global state becomes unmanageable in its entirety.

Separation Logic Separation Logic [O'Hearn, 2019] was a breakthrough in the area of proof of programs. Separation Logic is a Hoare logic that was pioneered at the beginning of the 21st century, with seminal articles by O'Hearn et al. [2001] and Reynolds [2002]. The main idea of Separation Logic is amazingly simple, and in fact so simple that we may wonder why it took more than 30 years to come up with. Indeed, the motto of Separation Logic is *locality*: when reasoning about a computation, one only needs to take into account the parts

of the memory this computation reads and writes to. Phrased differently, one can prove that a program is correct by taking into account only a small portion of the memory, forgetting about the global (and potentially huge) state.

At a high level, Separation Logic assertions describe the content of the heap. The base connective of Separation Logic assertions is the points-to assertion, written $\ell \mapsto v$, asserting that the abstract memory location ℓ currently stores the value v. Then, the key idea of Separation Logic is to introduce the *separating conjunction* of two assertions P and Q, written $P * Q$. The assertion $P * Q$ asserts that P and Q hold *on disjoint parts of the heap*. For example, the assertion $\ell_1 \mapsto v_1 * \ell_2 \mapsto v_2$ asserts not only that the location ℓ_1 stores the value v_1 and the location ℓ_2 stores the value v_2, but also that ℓ_1 and ℓ_2 are distinct (since they describe separated parts of the heap). This separating conjunction allows us to make precise the locality promise of Separation Logic, via the FRAME rule:

$$\frac{\{P\}\ t\ \{Q\}}{\{P\ *\ R\}\ t\ \{Q\ *\ R\}}\ \textsc{Frame}$$

In this deduction-style presentation, the part above the horizontal bar is the premise, and the part below is the conclusion: the horizontal bar has to be understood as an implication. In short, the FRAME rule asserts that, if the execution of program t updates a heap represented by the assertion P into a heap represented by assertion Q, then it also does so with any larger heap, extended by a portion of heap represented by the assertion R. This part of the heap is left untouched by the execution of t. Therefore, the same assertion R can be introduced both in the precondition and in the postcondition.

Concurrent Separation Logic Sequential Separation Logic was already a breakthrough, but its emphasis on locality turned out to be extraordinarily well suited for concurrent programs. O'Hearn [2007] and Brookes [2007] introduce *Concurrent Separation Logic* (CSL), for which they received the 2016 Gödel prize [EATCS, 2016]. The citation summarizes:

> For the last thirty years experts have regarded pointer manipulation as an unsolved challenge for program verification and shared-memory concurrency as an even greater challenge. Now, thanks to CSL, both of these problems have been elegantly and efficiently solved; and they have the same solution.

Let us first explain why reasoning about concurrent programs is difficult. A *thread* is a sequence of instructions that the programmer requests to be executed *sequentially*, on the same core. Concurrent programs can *fork* threads to be potentially executed concurrently, on different cores. We reason here on *unstructured concurrency*: there is no primitive to wait for the termination of a thread. This approach contrasts with the less-expressive *structured concurrency*, such as fork/join or async/finish, in which there is a primitive to wait for a thread or a group of threads to terminate. The primitive to wait for the termination of threads can be encoded in unstructured concurrency: we hence focus on the latter.

Then comes the question of how to model the interaction between threads through the shared memory. Interaction only happens through the heap, which is shared among threads: each thread is equipped with its own stack that is not shared. In the traditional model of Lamport [1979], called *sequential consistency*, each thread takes turns executing *atomic* operations, for example, a read or a write in the memory. The sequentially consistent model is outdated for modern computers, which rely on *relaxed memory models*. Yet, it is still a good approximation of the reality, which we follow in this thesis.

A concurrent computation in the sequentially consistent model is usually represented by an *interleaving*, a particular sequential order of threads' atomic operations. Thus, one of the main difficulties of the verification of concurrent programs is the need to cater for *every possible interleaving*. The number of possible interleaving explodes quickly, and it becomes unbearable (and not modular anyway) to consider each interleaving one by one. That is where

CSL comes to save the day. The first idea is to *separate* the memory that each thread interacts with, leading to the following PAR rule, allowing to reason on $(t_1 \,||\, t_2)$, the parallel execution of two threads t_1 and t_2.

$$\frac{\{P_1\}\; t_1\; \{Q_1\} \qquad \{P_2\}\; t_2\; \{Q_2\}}{\{P_1 \;*\; P_2\}\; (t_1 \,||\, t_2)\; \{Q_1 \;*\; Q_2\}}\ \textsc{Par}$$

The premise contains two assertions, separated by a space: one has to prove the conjunction of these two premises in order to deduce the conclusion of the rule. The PAR rule allows for reasoning on t_1 and t_2 *in isolation*, as if the other thread was not running. Indeed, the reasoning rule asserts that, if it t_1 has precondition P_1 and postcondition Q_1, and t_2 has precondition P_2 and postcondition Q_2, then *any interleaving* of t_1 and t_2 has precondition $P_1 * P_2$ and postcondition $Q_1 * Q_2$. In summary, as the two threads read and modify separated parts of memory, they do not interfere with each other.

Threads may need to communicate via shared memory. To cater for this need, CSL originally offers *critical sections* to which can be tied *invariants*. A critical section is linked to a lock, and corresponds to a sequence of instructions that are guaranteed to *not* be interleaved with other critical sections: the sequence of instructions will be completely executed before another such sequence is executed. An invariant is a property that the program preserves between every critical section. The user of CSL can hence assume that an invariant holds at the beginning of a critical section and must prove that this invariant continues to hold by the time the critical section ends. Critical sections, implemented with locks, semaphores or other mechanisms, correspond to *coarse-grained* concurrency. However, at the logical level, one can assume a critical section around each atomic instruction, leading to *fine-grained* concurrency [Parkinson et al., 2007].

In short, CSL's approach circumscribes the difficulties: the user does not need to reason about interference from other threads with the memory it *owns*, but needs to indeed cater for interference on *shared* memory by making use of an invariant.

Iris CSL gave rise to numerous variants [Gotsman et al., 2007; Vafeiadis and Parkinson, 2007; Dinsdale-Young et al., 2013; Svendsen and Birkedal, 2014; da Rocha Pinto et al., 2014]. Observing and anticipating the growing number of variants of CSL, Parkinson [2010] advocated the need for a core logic, capturing the fundamental property of Separation Logic. Iris [Jung et al., 2015, 2018b] proposes a such solution. Iris is a modern and powerful Concurrent Separation Logic which crucially offers *user-defined higher-order ghost state*. While reasoning on the program, Iris allows for representing a ghost (or *fictional*) heap, that does not exist at runtime, but facilitates the reasoning. In this ghost heap, each ghost location is equipped with a *camera* (or *resource algebra*), which describes the operations available on this ghost location. The Iris approach stands out because (1) the user can define cameras and equip each ghost location with a camera that fits best the intended reasoning and (2) the ghost state is higher-order, meaning that a ghost location can store an Iris assertion. Similarly to the points-to assertion for physical locations, ghost locations are owned and can appear within Separation Logic assertions. On top of the ghost state, Iris constructs and offers a generic notion of invariant. Iris invariants allow sharing both physical and ghost locations among threads, which enables reasoning about interference in fine-grained concurrent programs.

Iris was quickly adopted by a large fraction of the community, as demonstrated by a myriad of works making use of Iris as their underlying logic [Kaiser et al., 2017; Jung et al., 2018a; Frumin et al., 2018; Mével et al., 2020; de Vilhena and Pottier, 2021], including the present thesis. As of May 2024, the website of the Iris project (`https://iris-project.org/`) lists more than 100 articles making use of Iris.

Machine-Checked Verification Until now, we have described various techniques, including Separation Logic, to prove that a program satisfies its specification. But there are two

questions. First, how to ensure that these techniques are *sound*, in the sense that they indeed prove that a verified specification holds? Second, how to ensure that the user of these techniques correctly applies them? For example, how to check that the rules of Separation Logic are correctly applied during the verification of a program?

A pen-and-paper proof is not satisfactory. Indeed, a proof is similar to a program in the sense that it is written by humans, and may also itself contain bugs. To mimic our initial question: how to ensure that a proof contains no bugs? *Proof assistants* come to help. A proof assistant checks, line by line, that the proof only applies correct reasoning rules of an *underlying proof system*. Then, if one trusts the proof assistant as well as its underlying proof system, one can trust the checked proof.

One can ask: what is the point? Indeed, we just changed "trusting the proof" into "trusting the proof assistant". The point is that we, as scientists, can trust the *same* small set of proof assistants, and agree on the same set of foundational proof systems they apply.

Since the Automath project [de Bruijn, 1994], proof assistants have become more and more popular. Widely-used proof assistants include Coq [The Coq Development Team, 2024], Agda [The Agda Development Team, 2024], HOL [The HOL Development Team, 2024], and Lean [The Lean Development Team, 2024].

Proof assistants allowed for gaining trust in complex mathematical proofs, like the proof of the four-color theorem [Gonthier et al., 2008] or the Feit–Thompson odd order theorem [Gonthier et al., 2013]. Proof assistants also allowed for both developing and gaining trust in programs. CompCert [Leroy, 2024], a verified compiler for a large subset of C, was the first realistic verified compiler—it is a large and complex software, and its verification was a milestone. Numerous success stories followed, like the seL4 secure microkernel [Klein et al., 2009], or the CakeML compiler [Kumar et al., 2014].

Mechanizing Program Logics The Iris Separation Logic framework, which we use, is mechanized in Coq [Krebbers et al., 2017]. Let us describe the "standard recipe" to define an Iris-based mechanized program logic, a recipe that we are going to follow in this thesis.

There are three steps. First, one needs to model the programming language in Coq. This can be done by the means of a *deep embedding*: programs are represented by syntactic constructs, and a binary relation between syntactic programs define the *semantics* of the language, by relating two programs if the first can evolve in the second. Second, one needs to prove a set of *reasoning rules*, in Coq using Iris. These rules allow for reasoning on how a program evolves over time, without actually executing it. Third, one needs to prove the *adequacy theorem* of the program logic, which asserts that the correct applications of the reasoning rules on a program guarantees some property on its semantics. For example, almost every program logic guarantees *safety*, that is, a verified program never crashes.

To verify a program, the user of the logic needs to follow three other steps. First, the user writes a deeply embedded program in Coq. Second, the user writes the specification of the program as a Separation Logic triple in Iris. Third, the user applies the reasoning rules, in Coq, to verify that the program is correct, that is, the triple is true.

In the end, what needs to be trusted? First, we need to trust the fact that the underlying logic of Coq itself is correct. This logic, a variant of the Calculus of Inductive Constructions (CIC), is well-understood, stable, and accepted by the community. Second, we need to trust that Coq correctly applies the rules of its underlying logic. This is less clear, and bugs in Coq are regularly found. Yet, it is less likely to be a bug in Coq than in a human-written proof. Recently, the MetaCoq project [Sozeau et al., 2020, 2023] helped to gain confidence both in the underlying logic of Coq and its implementation. Indeed, MetaCoq started verifying an implementation of Coq in Coq itself, with as little axioms as possible. Third, we need to trust that the programming language we model—its syntax and its semantics—is a faithful representation of reality. In our case, the axiomatization of the behavior of the GC

is a crucial aspect, which we describe later on (§2.1, §4.6). Assuming these three points, one can be sure that the result being proved with a mechanized program logic is correct.

1.3 Verification of Resource Bounds

This thesis focuses on verifying resource bounds, and in particular heap space bounds. Let us briefly cover existing works.

The Resource Meter Approach Assuming that one is able to tell where in the code the resource of interest is consumed and produced, and how much of it is consumed or produced, reasoning about resource consumption can be reduced to reasoning about safety. To do so, one can construct a variant of the program that is instrumented with a *resource meter*, that is, a global variable whose value indicates what amount of the resource remains available. In this instrumented program, one places assertions that cause a runtime failure if the value of the meter becomes negative. If one can verify that the instrumented program has no runtime failure (that is, is safe), then one has effectively established a bound on the resource consumption of the original program.

The principle of a resource meter has been exploited in many articles, using various frameworks for establishing safety. For instance, Crary and Weirich [2000] exploit a dependent type system; Aspinall et al. [2007] exploit a VDM-style program logic; Carbonneaux et al. [2015] exploit a Hoare logic; He et al. [2009] exploit Separation Logic. The manner in which one reasons about the value of the meter depends on the chosen framework. In the most straightforward approach, the value of the meter is explicitly described in the pre- and postcondition of every function. This is the case, for instance, in He et al.'s work [2009], where two distinct meters are used to measure stack space and heap space. In a more elaborate approach [Atkey, 2011], which is made possible by Separation Logic, the meter is not regarded as an integer value but as a bag of *credits* that can be individually *owned*. Time credits remove the need to refer to the absolute value of the meter: instead, the specification of a function may indicate that this function requires a number of credits and produces a number of credits.

Credits were extensively used to verify *time* bounds, using *time credits*. This approach of credits was first implemented in Separation Logic by Charguéraud and Pottier [2019]. The idea of time credits dates back to Atkey [2011] and Pilkiewicz and Pottier [2011]. The thesis of Guéneau [2019] demonstrates advanced practical applications with fractional and negative time credits. The idea of time credits is appealing for reasoning about heap space: can we just use *space credits*? This thesis answers by the positive, and underlines the challenges of this approach.

Verification of Heap Space Bounds, with Manual Memory Management In a language with manual memory management, it is easy to tell where heap space is consumed and produced: an allocation instruction consumes the amount of space that it receives as an argument; a deallocation instruction recovers the space occupied by the heap block that is about to be deallocated.

In such a setting, traditional Separation Logic extended with space credits, can be used to establish verified heap space bounds. To the best of our knowledge, such a variant of Separation Logic does not exist in the literature. However, Hofmann's work on the typed programming language LFPL [2000] can be viewed as a precursor of this idea: LFPL has explicit allocation and deallocation, which consume and produce values of a linear type, written $\diamond$, whose inhabitants behave very much like space credits.

Verification of Heap Space Bounds, with Tracing Garbage Collection In the presence of tracing garbage collection, there is no memory deallocation instruction. Thus, it is

not evident at which program points space can be reclaimed. The GC can be invoked at arbitrary points in time, and may deallocate any subset of the unreachable blocks. (Recall that an unreachable block is a block that is not reachable from any *root* via a *path* in the heap.) Thus, reasoning about heap space in the presence of garbage collection requires reasoning about roots and unreachability.

Madiot and Pottier [2022] make a first step towards addressing this problem. They extend Separation Logic with several concepts. To keep track of free space, they use space credits. They view memory deallocation as a *logical operation*: it is up to the person who verifies the program to decide at which points this operation must be used and which memory blocks must be *logically deallocated*. This decision is subject to a proof obligation: a memory block can be logically deallocated only if it is unreachable. Unfortunately, the concept of unreachability is not local: that is, this concept cannot be easily expressed in terms of Separation Logic assertions. Therefore, Madiot and Pottier rephrase this proof obligation as follows: a memory block can be logically deallocated if it has no predecessors and is not a root. To record the predecessors of every memory block, they use *pointed-by* assertions [Kassios and Kritikos, 2013]. To record which blocks are roots, they focus their attention on a low-level language, where the stack is explicitly represented in the heap as a collection of "stack cells". Then, a block is a root if and only if it is a stack cell.

In work published during this thesis [Moine et al., 2023], we scale Madiot and Pottier's ideas up to a high-level language, where the stack is implicit. We introduce *Stackable* assertions to implicitly record which memory locations are "invisible roots", that is, which memory locations are roots because they appear in some indirect caller's stack frame.

Neither of these articles focuses on concurrency. Madiot and Pottier [2022] technically support concurrency, but only for a low-level language with stack variables explicitly allocated in the heap, and without any concurrent example covered. Moine et al. [2023] do not support it. By design, their *Stackable* assertion keeps track of a single stack. Extending it with support for multiple stacks is a priori not straightforward.

1.4 Contributions and Overview

We now present a high-level view of the contributions we make in this thesis, and give an overview of each chapter of this document.

Contribution 1: IrisFit, a Concurrent Separation Logic for Heap Space The main contribution of this thesis is IrisFit, the first program logic that allows establishing *safety*, *functional correctness*, and *worst-case heap space bounds* properties of concurrent programs, in the presence of garbage collection. Moreover, IrisFit allows *compositional reasoning*, that is, verifying each component in isolation.

IrisFit follows the *resource meter approach* and comes with space credits to keep track of the free space. We make use and enhance the pointed-by-heap assertion [Madiot and Pottier, 2022] to keep track of heap-to-heap pointers and we introduce the pointed-by-thread assertion to keep track of stack-to-heap pointers. Together, these assertions allow for proving that a location is *unreachable*. When the user is able to do so, we propose a *ghost deallocation rule*, following Madiot and Pottier [2022]. This rule intuitively consumes the pointed-by-thread and pointed-by-heap assertions showing unreachability and produces space credits.

Compared with standard Separation Logic, in order to keep track of the reachability of memory location, reasoning rules of IrisFit come with additional proof obligations. However, we are able to offer to the user simplified reasoning rules *via* two mechanisms we introduce. First, thanks to a dedicated *mode*, by pledging over a delimited syntactical scope not to deallocate any heap block, the user gets back almost a standard Separation Logic. Second, the user may record that some obligations were already satisfied thanks to a *souvenir*, dropping these obligations the next time they appear during the proof.

Contribution 2: Protected Sections and Polling Points Unless some care is taken, concurrent programs under garbage collection can have a suboptimal worst-case heap space complexity, that is, worse complexity than one might naively imagine. We propose *programming language features* that let the programmer *eliminate* some of the worst-case scenarios that we have discovered. The new features that we propose include *protected sections*—sections of the code where garbage collection is globally disabled—and *polling points*—instructions that block the current thread if garbage collection has been requested by one or more other threads. The programmer can exploit the presence of protected sections to obtain *improved* worst-case heap space complexity bounds. Polling points are meant to be automatically inserted by the compiler.

Our protected sections and polling points are inspired by mechanisms found in real-world language implementations, such as Ocaml 5's "safe points". However, we believe that our design is better behaved and introduces an important distinction between a construct that is inserted by the programmer and that is required to ensure good worst-case heap space complexity (namely, protected sections), and a construct that can be automatically inserted by the compiler and that is required to ensure liveness (namely, polling points).

We equip IrisFit with reasoning rules for protected sections, allowing a form of logical deallocation "in advance". These rules allow for logically deallocating a location that is reachable from "current" protected sections but with the proof obligation to show that this location is unreachable by the time the protected sections end.

Contribution 3: Soundness of IrisFit We prove the soundness of IrisFit. More specifically, we establish both a *safety* theorem, which guarantees that a verified program cannot crash and in particular cannot exceed the heap space bound, and a *liveness* theorem, which guarantees that, provided enough polling points are present, no thread can be forever blocked by a memory allocation request.

Contribution 4: Understanding the Space Consumption and Impact of Closures under Garbage Collection In most of functional programming languages, such as OCaml, functions with free variables are compiled to *closures* [Landin, 1964; Appel, 1992]. In usual program verification, closures receive no particular treatment. However, a closure is a heap-allocated block that points to the code of the function and the values of its free variables. Consequently, closures consume space and participate in the reachability of their environment: they thus need to be handled with care when reasoning about heap space (§2.6). Closures are pervasive in functional programs: understanding their space consumption is important. In order to reason about closures, we program "closure conversion", the compilation pass which transforms the high-level notation of a function with free variables into low-level code allocating and manipulating the closure heap block. We then prove reasoning rules for closure creation and call. We first propose a low-level interface for closures, revealing the actual code of the closure. From this interface, we derive a high-level interface, which hides the closure's code and only reveals its specification.

Contribution 5: Case Studies To illustrate IrisFit, we propose a range of case studies. For each case study, we establish not only functional correctness, but also heap space bounds of the various related functions.

First, IrisFit allows reasoning on sequential programs—that is, programs that do not make use of concurrency primitives. We verify an implementation of immutable linked lists and several functions over this data structure. In particular, we verify an implementation of the concatenation of lists in continuation-passing-style, showcasing closures. To demonstrate the compositionality of our approach, we verify several implementations of stacks, and explain how to compose them. We showcase IrisFit's ability to reason about circular data structures through the verification of a circular singly-linked list.

Second, IrisFit allows reasoning on subtle concurrent programs. We first propose an encoding of the fetch-and-add (FAA) primitive with protected sections, as an instructive exercise. We verify a concurrent monotonic counter as a pair of closures sharing mutable reference. We propose a library for a form of structured concurrency called async/finish. We then present two case studies of *lock-free* data structures, that is, data structures that can be used concurrently without blocking. Namely, we verify the correctness and establish heap space bounds for Treiber's stack [1986] and Michael and Scott's queue [1996]. These two data structures were already proved correct using Separation Logic and Iris [Krebbers et al., 2017; Vindum and Birkedal, 2021]. The novelty of our contribution is their heap space bounds. In particular, in order to establish their intuitive heap space bounds, we show how these two data structures need to be fixed with protected sections.

Structure of This Document In Chapter 2, we introduce in more detail the high-level ideas of LambdaFit, the language we study, and IrisFit, the program logic we equip our language with. Next, in Chapter 3, we explain on the particular example of Treiber's stack why protected sections are needed to guarantee the intuitive worst-case heap space bound of the functions of this data structure. After these intuitions and motivations, we present in Chapter 4 the formal syntax and semantics of LambdaFit. We then dive into IrisFit: in Chapter 5, we showcase the assertions of our program logic, and in Chapter 6 we present the related reasoning rules. We next justify the soundness of our approach. In Chapter 7, we comment on our safety and liveness theorems, and how they both follow from a *core soundness* theorem. In Chapter 8, we sketch the proof of this core soundness theorem. After that, in Chapter 9, we explain our encoding of closures and its associated reasoning rules. In Chapter 10, we present the technique of triples *with souvenir*, relieving the user of some proof obligations that were satisfied in the past. We then present our sequential case studies in Chapter 11 and showcase our concurrent case studies in Chapter 12. In Chapter 13, we cover the related work. Finally, in Chapter 14, we conclude with perspectives for future work.

1.5 Research Output: Publications and Mechanization

Publications The present thesis extends two articles.

- *A High-Level Separation Logic for Heap Space under Garbage Collection*,
  Alexandre Moine, Arthur Charguéraud, and François Pottier,
  Proceedings of the ACM on Programming Languages (issue POPL), 2023.

- *Will it Fit? Verifying Heap Space Bounds of Concurrent Programs under Garbage Collection with Separation Logic*,
  Alexandre Moine, Arthur Charguéraud, and François Pottier,
  Submitted, 2024.

The POPL article presents a Separation Logic for heap space under garbage collection for *sequential* programs. This article introduces a particular assertion for tracking roots in a modular way, and specifies and verifies closures based on closure conversion. This article also proposes the technique of triples with souvenir and illustrates the logic with a range of sequential case studies including lists, stacks, and a "counter" object implemented with two closures sharing a private mutable reference.

The submitted "Will it Fit?" article scales the earlier approach to a concurrent setting, introduces protected sections and polling points, and provides a range of concurrent case studies including Treiber's lock-free stack.

The present thesis borrows text from these two articles that are co-authored with my two advisors. Material for which I am the sole author includes parts of this introduction (§1.1,

§1.2, §1.4), a detailed comment on the safety and liveness proofs (§7, §8), additional case studies (§11.1, §11.5, §12.6), and the conclusion (§14).

Based on the ideas presented in this thesis, I led a collaboration with Sam Westrick and Stephanie Balzer to propose a Separation Logic for proving *disentanglement*, a property of parallel programs restricting the set of locations that parallel tasks are allowed to reach (§13.7). This work is not covered in the present document but was published separately in the article *DisLog: A Separation Logic for Disentanglement* [Moine, Westrick and Balzer, 2024].

Mechanization All of the results presented in this thesis are mechanized using the Coq proof assistant and the Iris framework. I am the sole author of this mechanization, which is freely available [Moine, 2024]. In detail, the syntax and semantics of our language, the validity of the reasoning rules of our program logic, the safety theorem and the liveness theorem, and our case studies are all machine-checked. More information about our mechanization may be found in the conclusion section (§14.1).

Chapter 2

Key Ideas

The Jamaicans (1967).
Ba Ba Boom.

LambdaFit is a call-by-value λ-calculus with dynamic memory allocation, mutable state, shared-memory concurrency, and tracing garbage collection. Its syntax and semantics are standard, save for a few original aspects.

First, LambdaFit exhibits a number of non-standard features related with memory management. Its operational semantics defines the concept of a *root* and has explicit garbage collection steps (§2.1). Furthermore, its operational semantics is parameterized with a *maximum heap size*. A memory allocation request that would cause this limit to be exceeded is *blocking* (§2.2). There is a notion of *protected section* where garbage collection cannot take place (§2.3) and a notion of *polling point*, an instruction that blocks the current thread if garbage collection has been requested by other threads (§2.4). After detailing these aspects, we give a high-level overview of how IrisFit provides reasoning rules for all these constructs (§2.5).

Second, LambdaFit is restricted to closed functions, also known as *code pointers*. We encode closures as heap-allocated objects that store code and data (§2.6).

2.1 Roots and Garbage Collection

To be able to talk and reason about the heap space complexity of LambdaFit programs, we must first equip LambdaFit with a semantics where garbage collection is explicit. Garbage collection [Jones et al., 2012] deallocates some or all unreachable memory blocks, where a block is *reachable* if there exists a path from some *root*, through the heap, to this block. Thus, the semantics of LambdaFit, and the notion of heap space complexity, depend on an answer to the question: what is a root?

How can the intuitive concept of a root be formally defined in the setting of a small-step, substitution-based operational semantics? Before addressing this question, let us recall a few fundamental aspects of such a semantics. In an *operational* semantics, a program state, which represents the state of a running program, is a syntactic object. Here, because we are interested in concurrent programs with dynamic memory allocation, a program state includes a thread pool (a list of threads) and a heap (a finite map of memory locations to memory blocks). In a *small-step* semantics, the manner in which the program state evolves over time is described by a reduction relation, that is, a binary relation on program states. In a *substitution-based* semantics, within the thread pool, each running thread is represented as a closed term, that is, a term without free variables. The reduction rules ensure that, whenever the scope of a variable is entered, a closed value is substituted for this variable. Thus, a closed term that represents a running thread describes both the code that this thread is about to execute and the data to which this thread has access. In particular, a memory location ℓ is a closed value, and a closed term that represents a running thread can contain memory locations.

In such a setting, what is a root? A simple, commonly agreed-upon answer is: *a root is a memory location ℓ that appears in at least one running thread t.* By this, we mean that the closed term t, which represents one of the currently running threads, contains one or more occurrences of the memory location ℓ.

This convention is known as the *free variable rule* (FVR) [Felleisen and Hieb, 1992; Morrisett et al., 1995]. Intuitively, the FVR states that the (computable) set of memory blocks that are reachable by the locations that appear in threads are a conservative approximation of the (uncomputable) set of memory blocks that might be accessed in the future by any of the threads. However, one must keep in mind that the FVR is not a static approximation of the dynamic semantics. Instead, the FVR is part of the definition of the dynamic semantics. It defines the concept of root, which in turn is used to define reachability and garbage collection.

The reader may wonder whether real-world programming languages respect the FVR. As far as we know, many real-world implementations of garbage-collected languages, such as OCaml, SML, Haskell, Scala, Java, and more, are meant to respect the FVR. Unfortunately, this intention is often undocumented. A prominent example of a compiler that explicitly respects the FVR is the CakeML verified compiler. Gómez-Londoño et al. [2020] and Gómez-Londoño and Myreen [2021] prove that the CakeML compiler respects a cost model that is defined at the level of the intermediate language DataLang that includes a form of the FVR.

2.2 Maximum Heap Size and Blocking Memory Allocation

The *default* operational semantics of LambdaFit is parameterized by a maximum heap size S, and is designed in such a way that the heap size always remains less than or equal to S. This property, which is stated by Lemma 2 (§4.9), is enforced as follows. Let us say that a memory allocation request is *large* if it would cause the heap size to exceed S, that is, if the sum of the current heap size and the number of requested words exceeds S. Otherwise, let us say that the allocation is *small*. Then, a large memory allocation instruction is not allowed to proceed: it is *blocked*. Once garbage collection takes place and is able to free enough space in the heap, this memory allocation instruction may become small, therefore unblocked. Polling points (§2.4) are another kind of instruction that can be blocked.

By blocking large memory allocation instructions, we ensure that one kind of undesirable behavior, namely *growing the heap too large*, is eliminated a priori. Two kinds of undesirable behavior remain permitted by the operational semantics, namely *crashes* and *deadlocks*: a thread can crash or become forever blocked. Under certain assumptions about the placement of polling points, our program logic statically guarantees that these undesirable behaviors cannot arise: this is stated by our *safety* and *liveness* theorems (Theorems 1 and 2).

An alternative approach would be to adopt a simpler *oblivious* operational semantics, where no instruction is ever blocked and where there is no space limit. Then, a different kind of undesirable behavior, namely *deadlocks*, is eliminated a priori. The undesirable behaviors that remain permitted by the operational semantics are *crashes* and *growing the heap too large*. In such a setting, our program logic, which is parameterized by an initial amount of available space S, statically provides the following guarantees: first, no thread can crash; second, when every thread is outside a protected section, the live heap space is bounded by S. We define this alternative operational semantics and establish this result: this is our *core soundness* theorem (Theorem 3). We use this theorem as a stepping stone in the proof of Theorems 1 and 2.

In the oblivious semantics an instruction is never blocked, whereas in the default semantics the same instruction can be blocked, the oblivious semantics is a superset of the default semantics. A program has a wider set of possible behaviors in the oblivious semantics than in the default semantics. This is why, with respect to the default semantics, our program logic is able to offer a stronger static guarantee. Indeed, with respect to the default semantics, it guarantees that the heap size *never* exceeds S, whereas with respect to the oblivious semantics it guarantees that *when every thread is outside a protected section* the live heap space is at most S.

In summary, there is a choice between two operational semantics for LambdaFit. This choice influences which undesirable behavior is eliminated a priori and which ones are elim-

inated by the program logic. Because the two semantics are not equivalent (one is a strict subset of the other), this choice is not just a matter of presentation: by choosing the more complex and more restrictive semantics, we can offer a simpler and stronger static guarantee.

2.3 Protected Sections

We equip LambdaFit with *protected sections*, that is, sections of the code where garbage collection *cannot* take place. As long as *any* thread is inside a protected section, garbage collection is disabled. Thus, if some thread is blocked by a large memory allocation request (§2.2), then this thread must wait until the GC has been allowed to run, which cannot take place until every thread is outside a protected section.

A protected section is explicitly delimited by two special instructions, enter and exit, which mark the beginning and end of the section. A single well-balanced construct "protected $\{t\}$" would be insufficiently flexible, because a protected section typically has one entry point and multiple exit points. This is illustrated by our implementation of Treiber's stack enhanced with protected sections (Figure 3).

Protected sections are subject to two restrictions. First, they cannot be nested. Second, a protected section must not contain a memory allocation instruction, a "fork" instruction,[1] a polling point (§2.4), or a function call.[2] These restrictions ensure that a protected section cannot contain a blocking instruction and can be exited in a bounded number of steps. The syntax of LambdaFit does not enforce these restrictions; however, violating them causes a runtime error, and is statically forbidden by our program logic.

Decorating a program with protected sections reduces the set of its possible behaviors: indeed, as long as one thread is inside a protected section, garbage collection cannot take place, so any thread that is in need of a large allocation must wait. Therefore, decorating a program with protected sections can only reduce its worst-case heap space complexity. This phenomenon is illustrated by the example of Treiber's stack (§3).

2.4 Polling Points

The combination of blocking memory allocations (§2.2) and protected sections (§2.3) potentially creates deadlocks, endangering *liveness*: that is, for some programs, there exist adversarial schedules where a large memory allocation request is blocked forever because the GC can never run. For example, imagine that thread A is blocked by a large memory allocation request while threads B and C both are in an infinite loop whose body contains a protected section. Then, the scheduler can interleave threads B and C in such a way that at all times one of them is inside a protected section, thereby forever disabling garbage collection and blocking thread A. We wish to forbid this scenario and to formally establish a liveness guarantee of the form: *always, eventually, every thread can make progress* (Theorem 2).

To this end, we equip LambdaFit with *polling points*. A polling point is a synchronization instruction, a form of barrier. A thread may proceed past a polling point only if no large memory allocation request is currently outstanding. In other words, if any thread is currently blocked by a large memory allocation request, then no thread can move past a polling point. A polling point must not appear inside a protected section.

By inserting sufficiently many polling points into a program, one can ensure that every memory allocation request is eventually satisfied. Indeed, as soon as one thread is blocked on

[1] In our operational semantics, "fork" does not allocate any memory in the heap. We could technically allow "fork" inside a protected section without breaking any of our results. In the real world, though, "fork" is likely to allocate memory. Because we forbid memory allocation inside a protected section, it seems natural to disallow "fork" inside protected sections as well.

[2] Because loops are encoded as recursive functions, forbidding function calls inside protected sections also forbids loops inside protected sections.

a large memory allocation request, every thread must eventually reach a polling point or a large memory allocation request, where it, too, becomes blocked. At this point, since neither polling points nor memory allocation instructions can appear inside a protected section, every thread must be outside a protected section. Thus, garbage collection can, and must, take place. If enough space becomes available—which our program logic statically guarantees!—then all outstanding memory allocation requests can be satisfied.

In the scenario outlined above, provided a polling point is inserted in the loops of both thread B and thread C, these two threads must eventually reach a polling point, where they become blocked. The only permitted step is then a garbage collection step, which is expected to free up enough memory to satisfy thread A's large allocation request. Consequently, all three threads become unblocked.

In principle, polling points could be manually inserted by the programmer, but that would be tedious. In practice, we expect a compiler to automatically insert polling points where needed. In §7.2, we prove that a particular polling point insertion strategy, inspired by that of the OCaml 5 compiler, does indeed insert enough polling points to guarantee liveness.

2.5 A Concurrent Separation Logic for Heap Space

This thesis presents IrisFit, a concurrent Separation Logic for LambdaFit. IrisFit shares many features with pre-existing Separation Logics. The behavior of a program fragment is described by a *triple*, an assertion whose parameters include a precondition (an assertion that describes the initial state), the program fragment of interest, and a postcondition (an assertion that describes the final state). In IrisFit, a triple also includes a thread identifier, as the logic assigns a unique name to each thread. A rich vocabulary of logical connectives, including *points-to* assertions, *separating conjunction*, and many more, is used to construct assertions, which encode both *knowledge* of the current state and *permission to update* this state in certain ways.

What sets IrisFit apart from traditional Separation Logics? IrisFit borrows ideas from previous Separation Logics equipped with support for reasoning about heap space in the presence of garbage collection [Madiot and Pottier, 2022; Moine et al., 2023] and scales them up to a concurrent setting. *Space credits* keep track of available space and serve as permissions to allocate memory. Furthermore, several kinds of assertions record which memory locations are reachable and in what way they can be reached. *Pointed-by-heap* assertions [Madiot and Pottier, 2022] keep track of predecessors of each location in the heap. *Pointed-by-thread* assertions (new in this thesis) keep track of the threads in which each location is a root. Like previous logics [Madiot and Pottier, 2022; Moine et al., 2023], IrisFit features a *ghost deallocation rule*. Because the programming language does not have an explicit memory deallocation instruction, it is up to the user of the logic to decide where to apply this rule. This rule requires proof that the memory block of interest is unreachable. This proof takes the form of pointed-by-heap and pointed-by-thread assertions, which are consumed; space credits are produced in their stead. A novelty of our approach is that logical deallocation *does not require or consume the points-to assertion.*

A crucial novel aspect of IrisFit is its ability to take advantage of protected sections while reasoning. Indeed, IrisFit offers a relaxed way of keeping track of roots inside protected sections. Ordinarily, pointed-by-thread assertions record which locations are roots, and as long as a location is a root, this location cannot be logically deallocated. Inside a protected section, however, an exception to this regime is made: the logic keeps track of a set of *temporary* roots. The user can turn an ordinary root into a temporary root (and vice-versa). The logic requires that, by the time the protected section ends, no temporary roots remain. Thus, by that time, every temporary root must no longer be a root (or must have been turned back into an ordinary root). Crucially, inside a protected section, the condition under which logical deallocation is permitted is: *if a location ℓ is not an ordinary root in any*

thread, and if ℓ has no live heap predecessors, then it can be logically deallocated. In other words, even though physical garbage collection is disabled inside protected sections, logical deallocation remains permitted, and is oblivious to the existence of temporary roots.[3] Finally, perhaps surprisingly, because the points-to assertion survives logical deallocation and enables read and write access, *a temporary root that has already been logically deallocated can still be accessed* before the protected section ends. This pattern appears while verifying lock-free data structures (§12.5).

2.6 Closures

To model the space complexity of programs that involve closures [Landin, 1964; Appel, 1992], we must somehow reflect the fact that a closure is a heap-allocated object. It has an address, a size, and may hold pointers to other objects. Thus, a closure has both direct and indirect impacts on space complexity: it occupies some space; and, by pointing to other objects, it keeps these objects live (reachable), preventing the GC from reclaiming the space that they occupy.

Therefore, we cannot use the standard small-step and substitution-based semantics of the λ-calculus, where a λ-abstraction is a value that does not have an address or a size. Instead, two approaches come to mind. One approach is to view a λ-abstraction as a primitive expression (not a value) whose evaluation causes the allocation of a closure. Another approach is to adopt a restricted calculus that offers only closed functions (as opposed to λ-abstractions with free variables) and to *define* closure construction and closure invocation as *macros*, or canned sequences of instructions, on top of this restricted calculus. As shown by Paraskevopoulou and Appel [2019], these two approaches yield the same space cost model. Furthermore, provided suitable syntax is chosen, the end user does not see the difference: it is just a matter of presentation in the metatheory.

We choose the second approach, because we find it simpler. In so doing, we follow Gómez-Londoño et al. [2020], who define the CakeML cost model at the level of DataLang, the language that serves as the target of closure conversion.

Thus, we equip LambdaFit with *closed functions*, which we also refer to as *code pointers*. We write $\mu_{\mathsf{ptr}} f.\, \lambda\vec{x}.\, t$ for a (recursive, multi-argument) closed function, and write $(v\ \vec{u})_{\mathsf{ptr}}$ for the invocation of the code pointer v with arguments $\vec{u}$. LambdaFit does not have primitive closures. This allows us to present a program logic for LambdaFit and to establish the soundness of this logic without worrying about closures. Once this is done, we define *closure construction* $\mu_{\mathsf{clo}} f.\, \lambda\vec{x}.\, t$ and *closure invocation* $(\ell\ \vec{u})_{\mathsf{clo}}$ as macros, and we extend our program logic with high-level reasoning rules for closures (§9). This allows end users to reason about these macros without expanding them and without even knowing how they are defined. In summary, LambdaFit can macro-express closures, and our logic allows reasoning about closures in the same way as if they were primitive constructs.

Our construction of closures as macros is the same as in our previous paper [Moine et al., 2023]. Our treatment of closures in the logic, however, has been generalized to multiple threads and simplified by describing closures via persistent predicates (§9.3, §9.5).

[3] Because the GC cannot run while any thread is inside a protected section, it cannot observe the existence of a temporary root. Therefore, there is no reason why the existence of a temporary root should prevent logical deallocation.

CHAPTER 3

WHY TREIBER'S STACK NEEDS PROTECTED SECTIONS

Gottschalk, L. M. (1857).
Souvenir de Porto-Rico, marche des Gibaros.

To motivate the interest of protected sections for establishing space bounds, we use the example of Treiber's stack, a lock-free, linearizable stack [Treiber, 1986]. We first present a naive implementation of this data structure without protected sections (§3.1). We point out that this implementation has an unsatisfying worst-case heap space complexity: there are scenarios where a successful pop operation does not allow any memory cell to be freed (§3.2). All memory *can* eventually be recovered, but this requires waiting until all threads have completed their interaction with the stack. This situation is unpleasant: pop cannot be given a simple logical specification of the form "a successful pop frees up one list cell worth of heap space". We show that, by annotating the code with protected sections, one can eliminate these undesirable scenarios and obtain the desired specification (§3.3). Near the end of this paper (§12.5), we present the details of how we formally establish this specification in IrisFit.

3.1 Naive Implementation of Treiber's Stack

Treiber's stack is implemented as a mutable reference to an immutable linked list, whose head corresponds to the top of the stack. Pseudo-code is presented in Figure 1.

The function call `create()` creates a new stack, represented as a fresh reference to an empty list `nil`. The `nil` value takes up no heap space: it is in fact an integer value.

The functions `push` and `pop` make crucial use of the atomic *compare-and-swap* (CAS) instruction. Each of them is implemented as a "CAS loop": it prepares an operation and attempts to atomically commit this operation using a CAS instruction. If the CAS succeeds, the function returns; otherwise, the loop continues with another attempt. Here, each loop is encoded as a tail-recursive function.

The function `push s v` inserts a new element `v` in a stack `s`. First, `s` is dereferenced (line 4) so as to obtain the address `h` of the head of the linked list. Then, a new list cell `h'` is allocated (line 5). The "data" and "tail" fields are initialized with `v` (line 6) and `h` (line 7). Then, a CAS instruction attempts to update the content of `s` from `h` to `h'` (line 8). If this attempt

```
1  let create () = ref nil
2
3  let rec push s v =
4    let h = !s in
5    let h' = new_cell () in
6    set_data h' v;
7    set_tail h' h;
8    if compare_and_swap s h h'
9    then ()
10   else push s v
```

```
11 let rec pop s =
12   let h = !s in
13   if is_nil h
14   then pop s
15   else
16     let h' = tail h in
17     if compare_and_swap s h h'
18     then data h
19     else pop s
```

Figure 1: An unsafe-for-space implementation of Treiber's stack

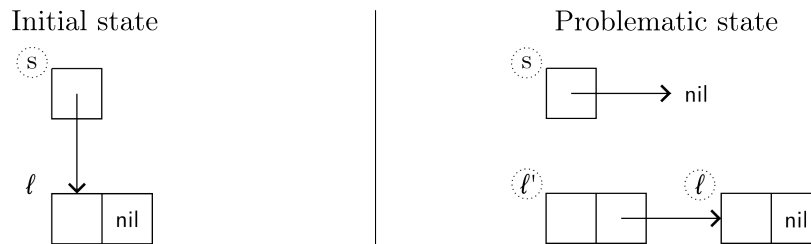

Figure 2: Initial and problematic states of the scenario.
Boxes are blocks whose location appears at their top left. Circled locations indicate roots.

is successful, `push` returns (line 9); otherwise, it means that a concurrent `push` or `pop` has succeeded. In this case, another attempt is made (line 10).

The function `pop s` extracts the top element of the stack `s`. First, the head `h` of the linked list is read (line 12). If the list is empty, `pop` makes another attempt (line 14), waiting for the stack to become nonempty. Otherwise, the "tail" field of the cell `h` is read so as to obtain the address `h'` of the next list cell (line 16). Then, a CAS instruction attempts to update the content of `s` from `h` to `h'` (line 17). If this attempt is successful, `pop` reads the "data" field of the cell `h` and returns its value (line 18); otherwise, it means that a concurrent `push` or `pop` has succeeded. In this case, another attempt is made (line 19).

Treiber's stack is *linearizable* [Herlihy and Wing, 1990], in the sense that `push` and `pop` atomically take effect at a certain point between the function call and return.

3.2 Space Usage of Treiber's Stack without Protected Sections

What is the space usage of `push` and `pop`? Let us write W for the number of memory words occupied by one list cell. A successful `push` operation consumes W memory words, as it allocates one single list cell. Symmetrically, a successful `pop` operation should intuitively free up W memory words. Indeed, the list cell being extracted from the list becomes unused, so one might hope that the GC could reclaim it.

However, this intuition is false: when `pop` returns, although the extracted list cell is indeed unused, it is not necessarily unreachable. Indeed, the extracted list cell might still be a root of other threads that are still in the process of executing a `push` or `pop` operation (which is about to fail) on the exact same cell. This issue leads to a problematic worst-case space complexity: a thread that holds a list cell as a root causes all descendants of this cell to remain reachable.

A Problematic Scenario and a Solution Here is a problematic scenario where a cell extracted by a successful `pop` remains reachable by other threads, preventing its immediate reclamation. Figure 2 pictures the initial state and the problematic state of the scenario. Suppose that the stack `s` consists of a single list cell whose address is ℓ. Suppose that thread A attempts to push a new value onto `s`, while thread B attempts to pop a value off `s`. Thread A starts making progress while thread B is asleep. Thread A begins to execute `push`. At line 4, its local variable `h` is bound to the address ℓ. At line 5, it allocates a new list cell at address ℓ'; its local variable `h'` is bound to ℓ'. At line 7, the "tail" field of the new cell is set to ℓ. Then, suppose thread A falls asleep. thread B wakes up and successfully pops one value off the stack. The reference `s` now stores the value `nil`. The cell ℓ has been extracted by `pop` and is no longer logically part of the stack. The cell ℓ' has not yet been inserted by `push` and is not logically part of the stack.

Because the cell ℓ has been extracted by a `pop` operation that has successfully completed, one might expect this cell to be now unreachable. However, this is not the case. Thread A has fallen asleep between lines 7 and 8. At this point, the local variables `h` and `h'` are still needed in the future: they occur on line 8. Therefore, the locations ℓ and ℓ' are *roots* in thread A. Besides, even if ℓ was not a root, it would still be reachable via the root ℓ', since the "tail"

```
1  let create () = ref nil
2
3  let rec push s v =
4    let h' = new_cell () in
5    set_data h' v;
6    enter; let h = !s in
7    set_tail h' h;
8    if compare_and_swap s h h'
9    then exit
10   else (exit; push s v)
```

```
11 let rec pop s =
12 enter; let h = !s in
13 if is_nil h
14 then (exit; pop s)
15 else
16   let h' = tail h in
17   if compare_and_swap s h h'
18   then (let v = data h in exit; v)
19   else (exit; pop s)
```

Figure 3: A safe-for-space implementation of Treiber's stack
Protected section entry and exit points are highlighted.

field of the cell ℓ' contains the pointer ℓ. This is problematic: a cell that has been extracted by `pop` is still reachable after `pop` has returned. So, *if the GC is invoked at this point, it cannot collect this cell.* Therefore, it is impossible to claim (and to prove) that `pop` frees up W words of memory!

How can this problem be addressed? A possible approach is to somehow forbid this undesirable behavior. For example, forbidding thread A from falling asleep at this particular point, between lines 7 and 8, might come to mind, but does not seem practical. Instead, we remark that *blocking garbage collection while thread A is asleep at this point* solves the problem, too. If some other thread signals that it needs memory, then, instead of immediately invoking the GC, we suggest to first wait until thread A wakes up, executes the CAS instruction at line 8, and reaches line 10. Recall the scenario that we are considering: thread B has successfully executed `pop` after the location ℓ was read from `s` by thread A at line 4. Therefore, the CAS instruction in thread A must fail, and thread A must reach line 10. By this time, the variables `h` and `h'` are no longer needed, so the locations ℓ and ℓ' are no longer roots. Moreover, ℓ' does not appear in the heap at all, and ℓ can be reached only via ℓ': therefore, both ℓ and ℓ' are unreachable. If the GC is now allowed to run, then it can reclaim these cells. In this approach, one *can* hope to prove that "`pop` frees up W words of memory", in the sense that "once `pop` has returned, as soon as garbage collection is allowed to take place, W words of memory will be freed up".

3.3 Space Usage of Treiber's Stack with Protected Sections

We introduce protected sections in Treiber's stack to prevent garbage collection between the moment a thread reads the address of the head cells and the moment the CAS operation is executed.

The modified pseudo-code that we propose appears in Figure 3. With respect to the original code in Figure 1, two main changes are made. First, protected sections, delimited by `enter` and `exit` instructions, are inserted into `push` and `pop`. Second, the allocation of a new list cell in `push` must be anticipated (moved higher up in the code), because memory allocations are forbidden inside protected sections (§2.3). The protected sections in Figure 3 are placed in such a way that, outside these sections, no list cell that is part of the data structure is a root. Therefore, when garbage collection takes place, necessarily at the time when no thread is inside a protected section, it is the case that no internal list cell is a root. This guarantee is strong enough to allow us to prove that "`pop` frees up W words of memory". Intuitively, the list cell addresses that are read inside protected sections can be registered in our logic as temporary roots, allowing for their logical deallocation after a successful pop operation. More details about this statement and about its proof are given in (§12.5).

Syntax and Semantics of LambdaFit

Janequin, C. (1537).
Le chant des oyseaulx.

In this chapter, we formally present the syntax of LambdaFit (§4.1) and its small-step reduction relations. We begin with our model of memory, that is, our view of the heap as a collection of memory blocks, and our notion of heap size (§4.2). We define thread pools and configurations (§4.3). Then, we introduce a series of reduction relations which, together, form the dynamic semantics of LambdaFit. The *head reduction* relation (§4.4) describes one elementary step of computation by one thread. The *step* relation (§4.5) allows head reduction to take place under an evaluation context. It represents one step of computation by one thread. The *garbage collection* relation (§4.6) describes the effect of the GC on the heap. The *action* relation (§4.7) and the *main reduction* relation (§4.9) describe the evolution of a complete system. There, each step is either a garbage collection step or a step of one thread. The main reduction relation is obtained from the action relation by restricting it to a subset of *enabled* actions (§4.8). This chapter covers the *default* operational semantics of LambdaFit, with blocking instructions and which gets stuck when the initial maximum heap size is exceeded (§2.2). The *oblivious* operational semantics, without blocking instructions and which ignores space constraints, is presented later on (§7.3).

4.1 Syntax

The syntax of LambdaFit appears in Figure 4. A *value* v is a piece of data that fits in one word of memory. A value can be the unit value $()$, a Boolean value b, an integer value z, a memory location ℓ (drawn from an infinite set $\mathcal{L}$), or a code pointer $\mu_{\mathsf{ptr}} f.\, \lambda\vec{x}.\, t$. Such a code pointer is a closed, recursive, multi-argument function. The side condition $fv(t) \subseteq \{f\} \cup \vec{x}$ ensures that the function is closed: that is, the only variables that may appear in the body of the function are f (a self-reference, allowing the function to invoke itself) and $\vec{x}$ (the formal parameters of the function).

The syntax of terms (also known as expressions) includes a number of standard sequential constructs, such as sequencing, conditionals, code pointer invocations, and primitive operations. The heap allocation expression $\mathsf{alloc}\ n$ allocates a fresh memory block of size n and returns its address. The field at offset i in the memory block at address x is read by the "load" expression $x[i]$ and written by the "store" expression $x[i] \leftarrow y$.

Two standard concurrency-related constructs are "fork" and CAS. The expression $\mathsf{fork}\, t$ spawns a new thread whose code is t. The compare-and-swap expression $\mathsf{CAS}\, \ell[i]\, v\, v'$ atomically loads a value from block ℓ at offset i, compares this value with v, and, in case they are equal, overwrites this value with v'. Its Boolean result indicates whether the write took place.

The instructions enter and exit mark the beginning and end of a protected section (§2.3). The poll instruction is a polling point (§2.4).

Primitives	$\odot ::= \&\& \mid \parallel \mid + \mid - \mid \times \mid \div \mid =$			
Values	$v, w ::= () \mid b \in \{\mathsf{false}, \mathsf{true}\} \mid z \in \mathbb{Z} \mid \ell \in \mathcal{L} \mid \mu_{\mathsf{ptr}} f.\, \lambda \vec{x}.\, t$ where $fv(t) \subseteq \{f\} \cup \vec{x}$			
Terms	$t, u ::= v$	*value*	$t[t]$	*heap load*
	x	*variable*	$t[t] \leftarrow t$	*heap store*
	$\mathsf{let}\, x = t \,\mathsf{in}\, t$	*sequencing*	$\mathsf{fork}\, t$	*thread creation*
	$\mathsf{if}\, t \,\mathsf{then}\, t \,\mathsf{else}\, t$	*conditional*	$\mathsf{CAS}\, t[t]\, t\, t$	*compare-and-swap*
	$(t\ \vec{u})_{\mathsf{ptr}}$	*code pointer invocation*	enter	*entering a protected section*
	$t \odot t$	*primitive operation*	exit	*exiting a protected section*
	$\mathsf{alloc}\, t$	*heap allocation*	poll	*polling point*
Contexts	$K ::= \mathsf{let}\, x = \square \,\mathsf{in}\, t$	$\mid \mathsf{if}\, \square \,\mathsf{then}\, t \,\mathsf{else}\, t$	$\mid \square \odot t$	$\mid v \odot \square$
	$\mathsf{alloc}\, \square$	$\mid \square[t]$	$\mid v[\square]$	$\mid \square[t] \leftarrow t$
	$v[\square] \leftarrow t$	$\mid v[v] \leftarrow \square$	$\mid (\square\ \vec{u})_{\mathsf{ptr}}$	$\mid (v\ (\vec{v} \mathbin{+\!\!+} \square \mathbin{+\!\!+} \vec{u}))_{\mathsf{ptr}}$
	$\mathsf{CAS}\, \square[t]\, t\, t$	$\mid \mathsf{CAS}\, v[\square]\, t\, t$	$\mid \mathsf{CAS}\, v[v]\, \square\, t$	$\mid \mathsf{CAS}\, v[v]\, v\, \square$
Statuses	$g ::= \mathsf{In} \mid \mathsf{Out}$			

Figure 4: LambdaFit: syntax

4.2 Memory Blocks, Stores, and Heap Size

A *memory block* is either a tuple of values, written $\vec{v}$, or a special deallocated block, written $\lightning$. A *store* σ (or *heap*) is a finite map of locations to memory blocks. We write $\emptyset$ to denote the empty store.

Our semantics does not recycle memory locations. When a heap block at address ℓ is reclaimed by the GC, the store is updated with a mapping of ℓ to $\lightning$. The address ℓ continues to exist and is never re-used. Naturally, in an implementation, memory locations would be recycled. However, we work at a higher level of abstraction. The reasoning rules of our program logic guarantee that a memory allocation always produces a fresh address. One could in principle prove that our semantics is equivalent to a lower-level semantics where locations are recycled. The argument would be that LambdaFit does not offer any means for observing a location's actual number, preventing the user from testing if a location was reused or not. We have not done such a proof.

We assume that the space usage (in words) of a block of n fields is $size(n)$, where $size$ is a mathematical function of $\mathbb{N}$ to $\mathbb{N}$. If, for instance, every memory block is preceded by a one-word header, then the function $size$ would be defined by $size(n) = n + 1$. LambdaFit and IrisFit are independent of the definition of $size$. For our case studies (§11, §12), we chose $size(n) = n$. We write $size(\vec{v})$ as a shorthand for $size(n)$, where n is the length of the list $\vec{v}$. We define $size(\lightning)$ to be 0, reflecting the fact that a deallocated block occupies no space.

We define the size of a store σ as the sum of the sizes of its blocks. Thus, we do not measure the physical size of the heap, that is, how much memory has been borrowed from the operating system. Instead, we measure the total size of the memory blocks that are currently allocated. We ignore fragmentation.

4.3 Thread Pools and Configurations

A *thread* t is just a term. A thread's *status* g is either In or Out. The status records whether the thread is currently inside or outside a protected section. A *thread pool* θ is a list of pairs (t, g) of a thread t and its status g. A *thread identifier* π is an integer index into a thread pool.

A *configuration* c is a pair (θ, σ) of a thread pool θ and a store σ. The *initial configuration* for a program t consists of a thread pool that contains just the thread (t, Out) and the

HeadLetVal
$$\mathsf{let}\ x = v\ \mathsf{in}\ t \mathbin{/} g \mathbin{/} \sigma \xrightarrow{\mathsf{head}} [v/x]t \mathbin{/} g \mathbin{/} \sigma \mathbin{/} \varepsilon$$

HeadCall
$$\frac{v = \mu_{\mathsf{ptr}} f.\, \lambda \vec{x}.\, t \qquad |\vec{x}| = |\vec{w}|}{(v\ \vec{w})_{\mathsf{ptr}} \mathbin{/} \mathsf{Out} \mathbin{/} \sigma \xrightarrow{\mathsf{head}} [v/f][\vec{w}/\vec{x}]t \mathbin{/} \mathsf{Out} \mathbin{/} \sigma \mathbin{/} \varepsilon}$$

HeadIfTrue
$$\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \mathbin{/} g \mathbin{/} \sigma \xrightarrow{\mathsf{head}} t_1 \mathbin{/} g \mathbin{/} \sigma \mathbin{/} \varepsilon$$

HeadIfFalse
$$\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2 \mathbin{/} g \mathbin{/} \sigma \xrightarrow{\mathsf{head}} t_2 \mathbin{/} g \mathbin{/} \sigma \mathbin{/} \varepsilon$$

HeadEnter
$$\mathsf{enter} \mathbin{/} \mathsf{Out} \mathbin{/} \sigma \xrightarrow{\mathsf{head}} () \mathbin{/} \mathsf{In} \mathbin{/} \sigma \mathbin{/} \varepsilon$$

HeadExit
$$\mathsf{exit} \mathbin{/} \mathsf{In} \mathbin{/} \sigma \xrightarrow{\mathsf{head}} () \mathbin{/} \mathsf{Out} \mathbin{/} \sigma \mathbin{/} \varepsilon$$

HeadPrim
$$\frac{v_1 \odot v_2 \xrightarrow{\mathsf{pure}} v}{v_1 \odot v_2 \mathbin{/} g \mathbin{/} \sigma \xrightarrow{\mathsf{head}} v \mathbin{/} g \mathbin{/} \sigma \mathbin{/} \varepsilon}$$

HeadAlloc
$$\frac{\ell \notin dom(\sigma) \qquad 0 < n \qquad \sigma' = [\ell := ()^n]\sigma}{\mathsf{alloc}\ n \mathbin{/} \mathsf{Out} \mathbin{/} \sigma \xrightarrow{\mathsf{head}} \ell \mathbin{/} \mathsf{Out} \mathbin{/} \sigma' \mathbin{/} \varepsilon}$$

HeadLoad
$$\frac{\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{\ell[i] \mathbin{/} g \mathbin{/} \sigma \xrightarrow{\mathsf{head}} v \mathbin{/} g \mathbin{/} \sigma \mathbin{/} \varepsilon}$$

HeadStore
$$\frac{\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}| \qquad \sigma' = [\ell := [i := v]\vec{w}]\sigma}{\ell[i] \leftarrow v \mathbin{/} g \mathbin{/} \sigma \xrightarrow{\mathsf{head}} () \mathbin{/} g \mathbin{/} \sigma' \mathbin{/} \varepsilon}$$

HeadCASFailure
$$\frac{\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}| \qquad \vec{w}(i) \neq v}{\mathsf{CAS}\ \ell[i]\ v\ v' \mathbin{/} g \mathbin{/} \sigma \xrightarrow{\mathsf{head}} \mathsf{false} \mathbin{/} g \mathbin{/} \sigma \mathbin{/} \varepsilon}$$

HeadCASSuccess
$$\frac{\sigma(\ell) = \vec{w} \qquad 0 \le i < |\vec{w}| \qquad \vec{w}(i) = v \qquad \sigma' = [\ell := [i := v']\vec{w}]\sigma}{\mathsf{CAS}\ \ell[i]\ v\ v' \mathbin{/} g \mathbin{/} \sigma \xrightarrow{\mathsf{head}} \mathsf{true} \mathbin{/} g \mathbin{/} \sigma' \mathbin{/} \varepsilon}$$

HeadPoll
$$\mathsf{poll} \mathbin{/} \mathsf{Out} \mathbin{/} \sigma \xrightarrow{\mathsf{head}} () \mathbin{/} \mathsf{Out} \mathbin{/} \sigma \mathbin{/} \varepsilon$$

HeadFork
$$\mathsf{fork}\ t \mathbin{/} \mathsf{Out} \mathbin{/} \sigma \xrightarrow{\mathsf{head}} () \mathbin{/} \mathsf{Out} \mathbin{/} \sigma \mathbin{/} t$$

Figure 5: The head reduction relation

StepHead
$$\frac{t \mathbin{/} g \mathbin{/} \sigma \xrightarrow{\mathsf{head}} t' \mathbin{/} g' \mathbin{/} \sigma' \mathbin{/} t^?}{t \mathbin{/} g \mathbin{/} \sigma \xrightarrow{\mathsf{step}} t' \mathbin{/} g' \mathbin{/} \sigma' \mathbin{/} t^?}$$

StepCtx
$$\frac{t \mathbin{/} g \mathbin{/} \sigma \xrightarrow{\mathsf{step}} t' \mathbin{/} g' \mathbin{/} \sigma' \mathbin{/} t^?}{K[t] \mathbin{/} g \mathbin{/} \sigma \xrightarrow{\mathsf{step}} K[t'] \mathbin{/} g' \mathbin{/} \sigma' \mathbin{/} t^?}$$

Figure 6: The step relation

Edge
$$\frac{\sigma(\ell) = \vec{w} \qquad \vec{w}(i) = \ell'}{\ell \rightsquigarrow_\sigma \ell'}$$

GC
$$\frac{dom(\sigma') = dom(\sigma) \qquad \begin{array}{l} \forall \ell.\ \ell \in dom(\sigma) \implies \\ \left\{ \begin{array}{l} \quad \sigma'(\ell) = \sigma(\ell) \\ \vee\ \sigma'(\ell) = \lightning\ \wedge\ \neg\,(\exists r \in R,\ \ r \rightsquigarrow^*_\sigma \ell) \end{array} \right. \end{array}}{R \vdash \sigma \xrightarrow{\mathsf{gc}} \sigma'}$$

Figure 7: The garbage collection relation

empty store $\emptyset$. We write $init(t)$ for this initial configuration. We define the heap size of a configuration as the size of its store: $size((\theta, \sigma)) = size(\sigma)$.

4.4 The Head Reduction Relation

The *head reduction* relation $t \,/\, g \,/\, \sigma \xrightarrow{\mathsf{head}} t' \,/\, g' \,/\, \sigma' \,/\, t^?$ describes an evolution of the term t with status g and store σ to a term t' with status g' and store σ', optionally forking off a new thread $t^?$. The metavariable $t^?$ denotes an optional term: it is either a term t or ε, which means that no thread was forked off.

The head reduction relation describes how an instruction is executed under the assumption that this instruction is *enabled*, that is, not blocked. The definition of enabled instructions, which describes under what conditions an instruction is blocked, is given later on (§4.8).

The head reduction relation is defined by the rules in Figure 5.

HeadLetVal, HeadIfTrue, HeadIfFalse, HeadPrim are standard.

HeadLoad, HeadStore, HeadCASSuccess and HeadCASFailure, which describe memory accesses, are also standard. These rules require that the memory location ℓ be valid: this is expressed by the premise $\sigma(\ell) = \vec{w}$. Furthermore, they require the integer value i to be a valid index into the memory block at address ℓ: this is expressed by the premise $0 \leq i < |\vec{w}|$. We write $\vec{w}(i)$ for the i-th value in the sequence $\vec{w}$, and $[i := v]\vec{w}$ for the sequence obtained by updating the sequence $\vec{w}$ at index i with the value v. We write $[\ell := \vec{w}]\sigma$ for the store obtained by updating the store σ at address ℓ with the block $\vec{w}$. Hence, $[\ell := [i := v]\vec{w}]\sigma$ describes an update of the i-th field of the block at location ℓ.

HeadEnter and HeadExit cause the thread to change its status from Out to In and vice-versa. By design, no reduction rule describes the effect of enter when the thread's status is In or the effect of exit when the thread's status is Out. Such a situation is considered a runtime error: the thread is *stuck*.

HeadCall, HeadAlloc, HeadFork, and HeadPoll require the thread's status to be Out. Thus, inside a protected section, a function call, a memory allocation request, a "fork" instruction, or a polling point causes a runtime error (§2.3). Aside from this, HeadCall and HeadFork are standard. HeadAlloc allocates a block of n fields at a fresh memory location and initializes each field with a unit value. We write $()^n$ for a sequence of n unit values. HeadPoll indicates that a polling point is a no-operation: poll acts as a form of barrier (§4.8), and is otherwise effectless.

4.5 The Step Relation

The *step* relation has the same shape as the head reduction relation (§4.4). It takes the form $t \,/\, g \,/\, \sigma \xrightarrow{\mathsf{step}} t' \,/\, g' \,/\, \sigma' \,/\, t^?$. It is inductively defined by the rules StepHead and StepCtx in Figure 6. These rules allow one head reduction step under a stack of evaluation contexts. An *evaluation context* K is a term with a hole written $\square$ at depth exactly 1. The syntax of evaluation contexts, presented in Figure 4, dictates a left-to-right, call-by-value evaluation strategy. We write $K[t]$ for the term obtained by filling the hole of the evaluation context K with the term t.

4.6 The Garbage Collection Relation

Several concepts related with garbage collection are defined in Figure 7. The *edge* relation $\ell \rightsquigarrow_\sigma \ell'$, defined by the rule Edge, means that the block at location ℓ contains a pointer

$$\text{ActionThread}\quad \frac{\theta(\pi) = (t, g) \qquad t \,/\, g \,/\, \sigma \xrightarrow{\mathsf{step}} t' \,/\, g' \,/\, \sigma' \,/\, t^? \qquad \theta' = [\pi := (t', g')]\theta +\!\!+ [(t^?, \mathsf{Out})]}{(\theta, \sigma) \xrightarrow{\mathsf{action}}_\pi (\theta', \sigma')}$$

$$\text{ActionGC}\quad \frac{\mathit{locs}(\theta) \vdash \sigma \xrightarrow{\mathsf{gc}} \sigma' \qquad \sigma \neq \sigma'}{(\theta, \sigma) \xrightarrow{\mathsf{action}}_{\mathsf{gc}} (\theta, \sigma')}$$

Figure 8: The action relation

to location ℓ'.[1] When this relation holds, we say that ℓ is a *predecessor* of ℓ'. The *reachability* relation $\ell \rightsquigarrow^*_\sigma \ell'$ is the reflexive-transitive closure of the edge relation.

The *garbage collection* relation $R \vdash \sigma \xrightarrow{\mathsf{gc}} \sigma'$, defined by the rule GC, describes the effect of the GC. This relation means that a garbage collection phase can transform the store σ into a store σ', while respecting the set of roots R, a set of memory locations. This relation is non-deterministic: the GC may reclaim any unreachable memory block, but need not reclaim every such block. According to the first premise of the rule GC, the stores σ and σ' have the same domain: garbage collection does not create or destroy any memory locations. According to the second premise, at each memory location ℓ, either nothing happens ($\sigma'(\ell) = \sigma(\ell)$) or a memory block becomes deallocated ($\sigma'(\ell) = \lightning$). The second case is permitted only if ℓ is not reachable from any of the roots in the set R.

4.7 The Action Relation

The relations defined so far describes how a thread makes a step (§4.5) and how the GC makes a step (§4.6). We now define a relation that interleaves these two kinds of steps. It is a labeled transition relation: each step is labeled with an *action* a, which is either a thread identifier π or the fixed token "gc". The *action* relation $c \xrightarrow{\mathsf{action}}_a c'$ relates two configurations c and c' and is labeled with an *action*. It is defined by the two rules in Figure 8. ActionThread allows a step by one thread whose identifier is π. This thread evolves from (t, g) to (t', g'): the thread pool is updated accordingly. The heap, which is shared between all threads, evolves from σ to σ'. A new thread $t^?$ possibly appears: if so, the thread pool is extended with the new entry $(t^?, \mathsf{Out})$. ActionGC describes a garbage collection step. The roots provided to the GC are $\mathit{locs}(\theta)$, that is, the locations that occur in the thread pool: this is the FVR (§2.1). The side condition $\sigma \neq \sigma'$ prevents the GC from stuttering. We would otherwise not be able to prove that every thread is eventually able to make progress (Theorem 2).

4.8 Enabled Actions

Two LambdaFit instructions possibly have a blocking behavior: a large memory allocation instruction is blocking (§2.2); if a large memory allocation request is outstanding, then a polling point is blocking (§2.4). Furthermore, while any thread is inside a protected section, garbage collection is disabled (§2.3). To reflect these aspects, we now wish to define under what conditions an action is *enabled* (allowed to proceed) or *disabled* (blocked).

The distinction between *small* and *large* memory allocation requests depends on the maximum heap size S (§2.2): Therefore, the notion of enabled action depends on the parameter S, and so does the main reduction relation, which is defined in the next subsection (§4.9).

To define enabled actions, a few auxiliary predicates are needed. They appear in Figure 9.

The proposition $\mathsf{IsAlloc}\ n\ t$ means that the next instruction of the thread t is "$\mathsf{alloc}\ n$". In other words, this thread is now requesting a new memory block of n fields. Similarly, the proposition $\mathsf{IsPoll}\, t$ means that the next instruction of the thread t is "poll".

[1] A value either *is* a location or contains no location at all. Thus, in Edge, we write just $\vec{w}(i) = \ell'$ instead of the seemingly more general condition $\ell' \in \mathit{locs}(\vec{w}(i))$.

$$\textsc{IsAllocHead}\quad \textsf{IsAlloc}\ n\ (\textsf{alloc}\ n)$$

$$\textsc{IsAllocCtx}\quad \frac{\textsf{IsAlloc}\ n\ t}{\textsf{IsAlloc}\ n\ (K[t])}$$

$$\textsc{IsPollHead}\quad \textsf{IsPoll}\ \textsf{poll}$$

$$\textsc{IsPollCtx}\quad \frac{\textsf{IsPoll}\ t}{\textsf{IsPoll}\ (K[t])}$$

$$\textsc{AllocFits}\quad \frac{\forall n.\ \textsf{IsAlloc}\ n\ t \implies \mathit{size}(\sigma) + n \le S}{\textsf{AllocFits}\ \sigma\ t}$$

$$\textsc{EveryAllocFits}\quad \frac{\forall t\ g.\ (t, g) \in \theta \implies \textsf{AllocFits}\ \sigma\ t}{\textsf{EveryAllocFits}\ (\theta, \sigma)}$$

$$\textsc{EnabledThread}\quad \frac{c = (\theta, \sigma) \qquad \theta(\pi) = (t, g) \qquad \textsf{AllocFits}\ \sigma\ t \qquad \textsf{IsPoll}\ t \implies \textsf{EveryAllocFits}\ c}{\textsf{Enabled}\ c\ \pi}$$

$$\textsc{AllOutside}\quad \frac{\forall t\ g.\ \ (t, g) \in \theta \implies g = \textsf{Out}}{\textsf{AllOutside}\ (\theta, \sigma)}$$

$$\textsc{EnabledGC}\quad \frac{\textsf{AllOutside}\ c}{\textsf{Enabled}\ c\ \textsf{gc}}$$

Figure 9: Enabled actions and auxiliary predicates

$$\textsc{EnabledAction}\quad \frac{\textsf{Enabled}\ c\ a \qquad c \xrightarrow{\textsf{action}}_a c'}{c \xrightarrow{\textsf{enabled action}}_a c'}$$

$$\textsc{Main}\quad \frac{c \xrightarrow{\textsf{enabled action}}_a c'}{c \xrightarrow{\textsf{main}} c'}$$

Figure 10: The main reduction relation

The proposition $\textsf{AllocFits}\ t\ \sigma$ means that, if the next instruction in thread t is an allocation request, then it is a small one: that is, there is currently enough free space in the store σ to satisfy it. When this is the case, we say that *thread t fits*. The proposition $\textsf{EveryAllocFits}\ c$ means that, in the configuration c, every thread fits.

The proposition $\textsf{Enabled}\ c\ a$ means that, in the configuration c, action a is enabled. It is defined by the rules EnabledThread and EnabledGC in Figure 9. For a thread π to be enabled, it must be the case that (1) thread π fits and (2) if thread π is at a polling point then every thread fits. For garbage collection to be enabled, it must be the case that every thread is currently outside a protected section.

The following simple lemma states that if every thread fits then every action is enabled. It is used in the proof of our liveness theorem (§7.2).

Lemma 1 (All Enabled)**.** *If $\textsf{EveryAllocFits}\ c$ holds, then, for every thread identifier π that is valid with respect to the configuration c, $\textsf{Enabled}\ c\ \pi$ holds.*

4.9 The Main Reduction Relation

The auxiliary relation $c \xrightarrow{\textsf{enabled action}}_a c'$, defined in Figure 10, is the restriction of the action relation to enabled actions. The *main reduction* relation $c \xrightarrow{\textsf{main}} c'$ is obtained from this auxiliary relation by abstracting away the action a. Thus, a step in the main reduction relation corresponds to an enabled action by some thread or by the GC.

By design of our semantics, the maximum heap size S is never exceeded. This is an immediate consequence of the fact that large memory allocation requests are blocked.

Lemma 2 (Heap Size)**.** *If $\mathit{size}(c) \le S$ and $c \xrightarrow{\textsf{main}} c'$ then $\mathit{size}(c') \le S$.*

This lemma is not used anywhere; it serves to document the design of the semantics.

Chapter 5

Program Logic: Assertions

Pink Floyd (1970).
Atom Heart Mother.

This chapter offers an overview of the various kinds of assertions that play a role in IrisFit. We introduce the syntax of each assertion, its intuitive meaning, and the ghost reasoning rules that help understand this meaning, such as splitting and joining rules. We informally explain the life cycle of each assertion: where it typically appears, where it is exploited, and where it is consumed. A presentation of the reasoning rules for terms is deferred to the following chapter (§6).

We begin with a presentation of triples (§5.1) and ghost updates (§5.2). Then, we briefly present the standard points-to assertion (§5.3), the novel "*sizeof*" assertion (§5.4), and space credits (§5.5). We then devote our attention to the assertions that record reachability or unreachability information, namely the pointed-by-heap assertion (§5.6), the novel pointed-by-thread assertion (§5.7), the novel "*inside*" and "*outside*" assertions (§5.8), and deallocation witnesses (§5.9). Finally, we explain liveness-based cancellable invariants (§5.10), a useful idiom that expresses that a certain invariant holds as long as a certain location is live.

IrisFit is a variant of the Iris program logic [Jung et al., 2018b, §6–7] and is built on top of the Iris base logic [Jung et al., 2018b, §5]. We write Φ for an assertion, $\ulcorner P \urcorner$ for a pure assertion, $\Phi * \Phi'$ for a separating conjunction, and $\Phi \mathrel{-\!\!*} \Phi'$ for a separating implication. We express the logical equivalence of two assertions as $\Phi \equiv \Phi'$. A postcondition Ψ is a function of a value to an assertion: in other words, it is the form $\lambda v.\, \Phi$.

5.1 Triples

A triple takes the form $\{\Phi\}\ \pi\colon t\ \{\Psi\}$. Its intuitive meaning is that if the store satisfies the assertion Φ then it is safe for thread π to execute the term t; furthermore, if and when this computation terminates and produces a value v, then the store satisfies the assertion $\Psi\ v$. A triple is a *persistent* assertion. Persistent assertions form a subclass of assertions [Jung et al., 2018b, §2.3]. Once a persistent assertion holds, it holds forever. In particular, persistent assertions are duplicable.

Even though the main reduction relation (§4.9) is parameterized with a maximum heap size S, the meaning of triples is independent of S. Indeed, triples are internally defined in terms of the *oblivious* reduction relation (§7.3), which does not depend on S. Therefore, none of the reasoning rules mentions S. Our program logic is compositional: each program component can be verified in isolation and without knowledge of S.

Formally, a triple is also parameterized by a *mask* [Jung et al., 2018b, §2.2]. Masks prevent the user from opening an invariant twice. As our treatment of invariants and masks is standard, we omit masks everywhere except for the formal definition of triples, which we present later on (§8.1).

We write $\{\Phi\}\ \pi : t\ \{\lambda\ell.\, \Phi'\}$, where the metavariable ℓ denotes a memory location, as syntactic sugar for $\{\Phi\}\ \pi\colon t\ \{\lambda v.\, \exists\ell.\, \ulcorner v = \ell \urcorner * \Phi'\}$. We adopt the convention that multi-line assertions are implicitly joined by a separating conjunction.

$$\textsc{Consequence}\quad \frac{\Phi \;{}^{\pi}\!\!\Rrightarrow^{locs(t)}\; \Phi' \qquad \{\Phi'\}\; \pi\colon t\; \{\Psi'\} \qquad \forall v.\; \Psi'\, v \;{}^{\pi}\!\!\Rrightarrow^{locs(v)}\; \Psi\, v}{\{\Phi\}\; \pi\colon t\; \{\Psi\}} \qquad \textsc{Frame}\quad \frac{\{\Phi\}\; \pi\colon t\; \{\Psi\}}{\{\Phi \;*\; \Phi'\}\; \pi\colon t\; \{\lambda v.\, \Psi\, v \;*\; \Phi'\}}$$

Figure 11: Structural reasoning rules

5.2 Ghost Updates

Iris features *ghost state* and *ghost updates* [Jung et al., 2018b, §5.4]. A ghost update is written $\Phi \Rrightarrow \Phi'$. It is an assertion, which means that (up to an update of the ghost state) the assertion Φ can be transformed into Φ'.

In IrisFit, it is sometimes necessary for a ghost update to refer to "the identifier of the current thread" or to "the roots of the current thread". For this purpose, we introduce a *custom ghost update*, written $\Phi \;{}^{\pi}\!\!\Rrightarrow^{V}\; \Phi'$, whose extra parameters are a thread identifier π and a set of memory locations V. It is strictly weaker than a standard ghost update: the law $(\Phi \Rrightarrow \Phi') \mathrel{-\!\!*} (\Phi \;{}^{\pi}\!\!\Rrightarrow^{V}\; \Phi')$ is valid.

Custom ghost updates are exploited in the Consequence rule, which appears in Figure 11. This rule allows strengthening the precondition and weakening the postcondition of a triple. Updating the precondition requires a custom ghost update where the parameter V is instantiated with $locs(t)$. Indeed, this set represents the roots at the point where this update takes place. Updating the postcondition requires a custom ghost update where V instantiated with $locs(v)$, where v denotes the value of the term t. Indeed, these are the roots at the point where that update takes place.

When a custom ghost update is independent of the parameters π and V, we omit them: we write $\Phi \Rrightarrow \Phi'$ for $\forall \pi\, V.\; \Phi \;{}^{\pi}\!\!\Rrightarrow^{V}\; \Phi'$. Examples of custom ghost updates appear in Figures 15, 16, and 17 and are discussed in the following sections.

The Frame rule, also shown in Figure 11, retains its standard form.

5.3 Points-to Assertions

IrisFit features standard points-to assertions of the form $\ell \mapsto_p \vec{w}$, where p is either a fraction in the semi-open interval $(0, 1]$ or the *discarded fraction* $\square$ [Vindum and Birkedal, 2021]. In the latter case, the points-to assertion is persistent.

Rules Points-to assertions can be split and joined in the usual way, and a points-to assertion that carries a fraction p can be permanently transformed into one that carries the discarded fraction $\square$. We do not show these standard rules.

Life cycle A points-to assertion appears when a memory block is allocated. It is required (and possibly updated) when this block is accessed by a load, store, or CAS instruction (§6.2). It is *not* required or consumed when this block is logically deallocated (§6.1). This is an original feature of IrisFit. In particular, keeping access to the points-to assertion *after* logical deallocation allows, within a protected section, recovering the space associated to a location while allowing further reads and writes to this location (§12.5, §12.6).

5.4 Sizeof Assertions

The assertion *sizeof* $\ell\, n$ means that there is or there used to be a block of size n at address ℓ. It is persistent: indeed, once the size of a block has been fixed, it can never be changed.

$$
\begin{array}{rcll}
\ell \mapsto_p \vec{w} & -\!\!* & \ell \mapsto_p \vec{w} \;*\; \textit{sizeof}\,\ell\,(\textit{size}(\vec{w})) & \textsc{SizeOfPointsTo} \\
\textit{sizeof}\,\ell\,n \;*\; \textit{sizeof}\,\ell\,m & -\!\!* & \ulcorner n = m \urcorner & \textsc{SizeOfConfront} \\
\textit{sizeof}\,\ell\,n & & \text{is persistent} & \textsc{SizeOfPersist}
\end{array}
$$

Figure 12: Reasoning rules of the "*sizeof*" assertion

$$
\begin{array}{rcll}
\ulcorner \text{True} \urcorner & \Rrightarrow & \Diamond 0 & \textsc{ZeroSC} \\
\Diamond(n_1 + n_2) & \equiv & \Diamond n_1 \;*\; \Diamond n_2 & \textsc{SplitJoinSC}
\end{array}
$$

Figure 13: Reasoning rules for space credits

Rules Two reasoning rules allow introducing and exploiting "*sizeof*" assertions (Figure 12). SizeOfPointsTo creates a "*sizeof*" assertion out of a points-to assertion. SizeOfConfront states that two "*sizeof*" assertions for the same address must agree on the size of the block at this address.

Life cycle The "*sizeof*" assertion is produced by SizeOfPointsTo. This assertion is consulted by the logical deallocation rules (§6.1, §6.6) to determine the number of space credits that must be produced.

5.5 Space Credits

To reason about free space, we use *space credits* [Madiot and Pottier, 2022; Moine et al., 2023]. The assertion $\Diamond n$ denotes the unique ownership of n space credits. It can be understood as a permission to allocate n words of memory. At a lower level of understanding, this assertion means that n memory words *are currently free or can be freed* by the GC *once it is given a chance to run*. This interpretation of space credits is the same as the earlier papers cited above; however, in these previous papers, garbage collection was allowed to take place at any time, whereas in the present paper, garbage collection is enabled only when all threads are outside protected sections.

Space credits are measured using non-negative *rational* numbers, similarly as in our earlier paper Moine et al. [2023]. Of course, a physical word of memory cannot be split, so the total number of space credits in existence is a natural number; so are the numbers involved in the reasoning rules for memory allocation and deallocation. Still, rational numbers appear essential in certain amortized complexity analyses, as illustrated by the example of chunked stacks [Moine et al., 2023]. Rational credits also appear in amortized *time* complexity analyses [Charguéraud and Pottier, 2019; Mével et al., 2019].

Rules Figure 13 presents two basic reasoning rules about space credits. ZeroSC asserts that zero credits can be forged out of thin air. SplitJoinSC asserts that space credits can be split and joined.

Life cycle Space credits are consumed by memory allocation (§6.2) and produced by logical deallocation (§6.1). Because there is no way of creating space credits out of nothing, a program or program component is usually verified under the assumption that a number of space credits are provided. This is apparent in the statement of our safety theorem (§7.1). This theorem states that, if a program is verified under the precondition $\Diamond S$, then setting the maximum heap size to S allows this program to be safely executed.

$$(\ell \hookleftarrow_{q_1} L_1 \ * \ \ell \hookleftarrow_{q_2} L_2) \ \twoheadrightarrow \ \ell \hookleftarrow_{q_1+q_2} (L_1 \uplus L_2) \qquad \textsc{JoinPBHeap}$$

$$\ell \hookleftarrow_{q_1+q_2} (L_1 \uplus L_2) \ \twoheadrightarrow \ (\ell \hookleftarrow_{q_1} L_1 \ * \ \ell \hookleftarrow_{q_2} L_2) \quad \text{if } \begin{cases} q_1 = 0 \Rightarrow \mathsf{NoPositive}(L_1) \\ q_2 = 0 \Rightarrow \mathsf{NoPositive}(L_2) \end{cases} \qquad \textsc{SplitPBHeap}$$

$$\ell \hookleftarrow_q L \ \twoheadrightarrow \ \ell \hookleftarrow_q (L \uplus \{+\ell'\}) \quad \text{if } q > 0 \qquad \textsc{CovPBHeap}$$

Figure 14: Reasoning rules for the pointed-by-heap assertion

5.6 Pointed-By-Heap Assertions

Our *pointed-by-heap* assertions are the "pointed-by" assertions of our earlier paper [Moine et al., 2023]. They generalize the "backpointer" mechanism introduced by Kassios and Kritikos [2013]. The longer name "pointed-by-heap" avoids confusion with our novel "pointed-by-thread" assertions (§5.7). To make this thesis self-contained, we recall what form these assertions take, what they mean, and what purpose they serve.

A *pointed-by-heap* assertion for the location ℓ' keeps track of a multiset L of predecessors of ℓ' (§4.6). It takes the form $\ell' \hookleftarrow_q L$, where L is a signed multiset of locations and q is a possibly-null fraction, that is, a rational number in the closed interval $[0;1]$.

Signed multisets Signed multisets [Hailperin, 1986], also known as *generalized sets* [Whitney, 1933; Blizard, 1990] or *hybrid sets* [Loeb, 1992], are a generalization of multisets: they allow an element to have *negative* multiplicity. A signed multiset is a total function of elements to $\mathbb{Z}$. The disjoint union operation $\uplus$ is the pointwise addition of multiplicities. We write $+x$ for a positive occurrence of x and $-x$ for a negative occurrence of x. For example, $\{+x;+x\} \uplus \{-x\}$ is $\{+x\}$. We write $\mathsf{NoNegative}(L)$ when no element has negative multiplicity in L. Symmetrically, we write $\mathsf{NoPositive}(L)$ when no element has positive multiplicity in L.

Possibly-Null Fractions In traditional Separation Logics with fractional permissions [Boyland, 2003; Bornat et al., 2005], a fraction is a rational number in the semi-open interval $(0,1]$. If there exists a share that carries the fraction 1, then no other shares can separately exist. With *possibly-null fractions*, the fraction 0 is allowed, so a full pointed-by-heap assertion $\ell' \hookleftarrow_1 L$ does *not* exclude the existence of a separate pointed-by-heap assertion with fraction zero, say $\ell' \hookleftarrow_0 L'$.

Nevertheless, we enforce the following *null-fraction invariant*: in a pointed-by-heap assertion $\ell' \hookleftarrow_q L$, *if the fraction q is 0, then no location can have positive multiplicity in L*; or, in short, $q = 0$ implies $\mathsf{NoPositive}(L)$.

Signed multisets and possibly-null fractions allow us to use the assertion $\ell' \hookleftarrow_0 \{-\ell\}$ as *a permission to remove one occurrence of ℓ from the predecessors of ℓ'*. The assertion $\ell' \hookleftarrow_0 \{-\ell\}$ allows formulating the reasoning rule for store instructions (§6.2) in a simpler way than would otherwise be possible.

Over-Approximation of Live Predecessors We say that a location ℓ is *dead* if it has been allocated and logically deallocated already (§5.9). We say that it is *live* if it has been allocated but not logically deallocated yet.

The true purpose of pointed-by-heap assertions is to keep track of *live* predecessors. A dead predecessor is irrelevant: increasing its multiplicity in a multiset of predecessors is sound; decreasing it is sound, too. As far as live predecessors are concerned, only over-approximation is permitted. Increasing the multiplicity of a live predecessor is sound; decreasing it is not.

In light of this, and in light of the null-fraction invariant, a *full* pointed-by-heap assertion $\ell' \hookleftarrow_1 L$, where the fraction is 1, guarantees that the multiset L contains *all live predecessors* of the location ℓ'. In particular, the assertion $\ell' \hookleftarrow_1 \emptyset$ guarantees that ℓ' has

$$
\begin{array}{rcll}
\ell \Leftarrow_{p_1+p_2} (\Pi_1 \cup \Pi_2) & \equiv & (\ell \Leftarrow_{p_1} \Pi_1 \; * \; \ell \Leftarrow_{p_2} \Pi_2) & \textsc{FracPBThread} \\
\ell \Leftarrow_p \Pi_1 & -\!\!* & \ell \Leftarrow_p (\Pi_1 \cup \Pi_2) & \textsc{CovPBThread} \\
\ulcorner \ell \notin V \urcorner \; * \; \ell \Leftarrow_p \{\pi\} & {}^{\pi}\!\Rrightarrow^{V} & \ell \Leftarrow_p \emptyset & \textsc{TrimPBThread}
\end{array}
$$

Figure 15: Reasoning rules for the pointed-by-thread assertion

no live predecessors. Such full knowledge of the live predecessors is required by the logical deallocation rule (§6.1, §6.6).

Rules Pointed-by-heap assertions obey the splitting, joining, and weakening rules in Figure 14. JoinPBHeap joins two pointed-by-heap assertions by adding the fractions q_1 and q_2 and by adding the signed multisets L_1 and L_2. In the reverse direction, SplitPBHeap splits a pointed-by-heap assertion. Its side condition ensures that the null-fraction invariant is preserved. CovPBHeap asserts that a pointed-by-heap assertion (whose fraction is nonzero) is covariant in its multiset: that is, over-approximating the multiset of predecessors is sound. It is a direct consequence of SplitPBHeap, instantiated with $q_2 \triangleq 0$ and $L_2 \triangleq \{-\ell'\}$. In the reverse direction, the rule CleanPBHeap, which is discussed later on (§5.9), allows removing a dead predecessor from a multiset of predecessors.

Life cycle A full pointed-by-heap assertion for the location ℓ appears when this location is allocated. Fractional pointed-by-heap assertions are required, updated, and produced by store instructions. For example, consider a store instruction that updates the field $\ell[i]$ and overwrites the value ℓ'_1 with the value ℓ'_2. The reasoning rule for this instruction (§6.2) requires a pointed-by-heap assertion $\ell'_2 \leftarrow_q \emptyset$, which it transforms into $\ell'_2 \leftarrow_q \{+\ell\}$. Furthermore, the pointed-by-heap assertion $\ell'_1 \leftarrow_0 \{-\ell\}$ is produced. A full pointed-by-heap assertion for the location ℓ is consumed when ℓ is logically deallocated.

Notation We define a generalized pointed-by-heap assertion $v \leftarrow_q L$ whose first argument is a value, as opposed to a memory location. If v is a location ℓ', then this assertion is defined as $\ell' \leftarrow_q L$. Otherwise, it is defined as $\ulcorner\text{True}\urcorner$. Furthermore, we write $v \leftarrow_q^{0} L$ for the assertion $\ulcorner q > 0 \urcorner * v \leftarrow_q L$. This notation is used in the reasoning rule Store (§6.2), among other places.

5.7 Pointed-By-Thread Assertions

The pointed-by-heap assertions presented in the previous section record *which heap blocks* contain pointers to a location ℓ. This information is useful but is not sufficient for our purposes. The logic must also record *which threads* have access to ℓ, that is, in which threads ℓ is a root. For this purpose, we introduce two distinct yet cooperating mechanisms. The first mechanism, presented here, is the pointed-by-thread assertion. The second mechanism, presented next (§5.8), is the "*inside*" assertion. When the fact that ℓ is a root in thread π is recorded by a pointed-by-thread assertion, we say that ℓ is an *ordinary root* in thread π; when this fact is recorded by an "*inside*" assertion, we say that ℓ is a *temporary root* in thread π. The motivation for this distinction has been presented earlier (§2.3, §3).

A *pointed-by-thread* assertion takes the form $\ell \Leftarrow_p \Pi$, where p is a fraction in the semi-open interval $(0; 1]$ and Π is a set of thread identifiers. These assertions intuitively generalize the *Stackable* assertions of our earlier paper [Moine et al., 2023] to a multi-threaded setting.

A *full* pointed-by-thread assertion $\ell \Leftarrow_1 \Pi$, where the fraction is 1, guarantees that Π is the set of *all* threads in which ℓ is an ordinary root. Such full knowledge is required by the logical deallocation rule (§6.1, §6.6).

$$\begin{array}{rclr}
\mathit{inside}\,\pi\,T \;*\; \mathit{outside}\,\pi & \twoheadrightarrow & \ulcorner\mathrm{False}\urcorner & \textsc{InsideNotOutside} \\
\mathit{inside}\,\pi\,T \;*\; \ell \Lleftarrow_p \{\pi\} & \Rrightarrow & \mathit{inside}\,\pi\,(T \cup \{\ell\}) \;*\; \ell \Lleftarrow_p \emptyset & \textsc{AddTemporary} \\
\mathit{inside}\,\pi\,T \;*\; \ell \Lleftarrow_p \emptyset & \Rrightarrow & \mathit{inside}\,\pi\,(T \setminus \{\ell\}) \;*\; \ell \Lleftarrow_p \{\pi\} & \textsc{RemTemporary} \\
\mathit{inside}\,\pi\,T & {}^{\pi}\Rrightarrow^{V} & \mathit{inside}\,\pi\,(T \cap V) & \textsc{TrimInside}
\end{array}$$

Figure 16: Reasoning rules for "*inside*" and "*outside*" assertions

Rules Figure 15 presents the splitting, joining, weakening, and trimming rules associated with the pointed-by-thread assertion. FracPBThread allows splitting and joining pointed-by-thread assertions. CovPBThread asserts that a pointed-by-thread assertion is covariant in the set Π: that is, over-approximating Π is sound. TrimPBThread allows *trimming* a pointed-by-thread assertion, that is, removing the thread identifier π from a pointed-by-thread assertion for the location ℓ, provided it is evident that ℓ is no longer a root in thread π. This rule is expressed as a custom ghost update: it transforms $\ell \Lleftarrow_p \{\pi\}$ into $\ell \Lleftarrow_p \emptyset$, provided ℓ is not a member of the set V, which denotes the set of roots of the thread π (recall §5.2). The condition $\ell \notin V$ means indeed that ℓ is not a root in thread π.

A curious reader may wonder whether and why TrimPBThread remains sound in combination with the Bind rule. Indeed, Bind lets the user focus on a subterm, therefore implies that the set V is a strict *subset* of the set of all roots of the current thread. This aspect is explained later on (§6.4).

Life cycle A full pointed-by-thread assertion $\ell \Lleftarrow_1 \{\pi\}$ appears when a location ℓ is allocated by a thread π. A fractional pointed-by-thread assertion is ordinarily required and updated by load instructions: when a thread π obtains the location ℓ as the result of a load instruction, an assertion $\ell \Lleftarrow_p \emptyset$ is updated to $\ell \Lleftarrow_p \{\pi\}$. If the thread π is currently outside a protected section, such an update is mandatory. If the thread π is currently inside a protected section, then it can be avoided by recording ℓ as a temporary root (§6.3). Once ℓ is no longer a root in any thread, TrimPBThread can be used to obtain $\ell \Lleftarrow_1 \emptyset$, which is consumed by the logical deallocation of ℓ.

Notation We define a generalized pointed-by-thread assertion $v \Lleftarrow_p \Pi$, whose first argument is a value, as opposed to a memory location. If v is a location ℓ, then this assertion is defined as $\ell \Lleftarrow_p \Pi$. Otherwise, it is defined as $\ulcorner\mathrm{True}\urcorner$. Besides, we write an iterated conjunction of pointed-by-thread assertions under the form $M \Lleftarrow \Pi$, where M is a finite map of memory locations to fractions and Π is a set of thread identifiers. This assertion is defined by the following equation: $M \Lleftarrow \Pi \;\triangleq\; \bigstar_{(\ell,p)\in M}\,(\ell \Lleftarrow_p \Pi)$.

5.8 Inside and Outside Assertions

The assertion $\mathit{outside}\,\pi$ means that the thread π is currently outside a protected section. The assertion $\mathit{inside}\,\pi\,T$ means that thread π is currently inside a protected section and that the set of its temporary roots (§2.5) is T. The set T is a set of memory locations.

Rules Figure 16 presents a number of reasoning rules related to "*inside*" and "*outside*" assertions. InsideNotOutside states that a thread cannot be both inside and outside a protected section. AddTemporary converts an ordinary root to a temporary root. The pointed-by-thread assertion $\ell \Lleftarrow_p \{\pi\}$ is transformed to $\ell \Lleftarrow_p \emptyset$; meanwhile, ℓ is added to the set of temporary roots carried by the "*inside*" assertion. In the reverse direction, RemTemporary converts a temporary root to an ordinary root. TrimInside trims the set of temporary roots by removing any locations that are no longer roots in the current thread. It is analogous to TrimPBThread.

$$
\begin{array}{rcll}
\dagger\,\ell & \Rrightarrow & \ell' \hookleftarrow_0 \{-\ell\} & \textsc{CleanPBHeap} \\
\dagger\,\ell \;*\; \ell \hookleftarrow^{o}_{q} L & \Rrightarrow & \ulcorner\text{False}\urcorner & \textsc{DeadPBHeap} \\
\dagger\,\ell \;*\; \ell \Lleftarrow_p \Pi & \Rrightarrow & \ulcorner\text{False}\urcorner & \textsc{DeadPBThread} \\
\ulcorner \ell \in V \urcorner \;*\; \dagger\,\ell \;*\; \mathit{outside}\,\pi & {}^{\pi}\Rrightarrow^{V} & \ulcorner\text{False}\urcorner & \textsc{NoDanglingRootOut} \\
\ulcorner \ell \in (V \setminus T) \urcorner \;*\; \dagger\,\ell \;*\; \mathit{inside}\,\pi\,T & {}^{\pi}\Rrightarrow^{V} & \ulcorner\text{False}\urcorner & \textsc{NoDanglingRootIn} \\
 & \dagger\ell \;\text{ is persistent} & & \textsc{DeadPersist}
\end{array}
$$

Figure 17: Reasoning rules for deallocation witnesses

Life cycle The assertion $\mathit{outside}\,\pi$ appears when thread π is created and is consumed when this thread terminates. This will be visible in the statement of Theorem 1, which describes the creation and termination of the main thread, and in the reasoning rule for "fork" instructions (§6.2). The assertion $\mathit{outside}\,\pi$ is required and preserved by the instructions that must not appear inside a protected section, namely memory allocations, function calls, "fork" instructions, and polling points. Entering a protected section transforms $\mathit{outside}\,\pi$ into $\mathit{inside}\,\pi\,\emptyset$; exiting a protected section causes the reverse transformation.

5.9 Deallocation Witnesses

The persistent assertion $\dagger\,\ell$ is a *deallocation witness* for the location ℓ. This assertion guarantees that ℓ has been logically deallocated, that is, ℓ is dead.

The fact that ℓ is dead implies that ℓ cannot be reached from an ordinary root. However, this does not imply that ℓ is unreachable: indeed, this location could still be reachable via a temporary root.

The assertion $\dagger\,\ell$ can be read as a permission to remove ℓ from the multiset of predecessors carried by a pointed-by-heap assertion. Indeed, the purpose of pointed-by-heap assertions is to keep track of live predecessors (§5.6).

A deallocation witness $x \not\mapsto$ appears in Incorrectness Separation Logic [Raad et al., 2020]. Contrary to us, this assertion is non-persistent. In RustBelt [Jung et al., 2018a] an ended lifetime κ is denoted with a persistent *dead token* written $[\dagger\kappa]$. Persistent deallocation witnesses appear in Madiot and Pottier's work [2022] and in our earlier paper [Moine et al., 2023]. These two papers do not have protected sections, therefore have no distinction between ordinary and temporary roots. There, a dead location is unreachable.

Rules Figure 17 presents reasoning rules for deallocation witnesses. CleanPBHeap requires a deallocation witness for ℓ and produces $\ell' \hookleftarrow_0 \{-\ell\}$, allowing ℓ to be removed from the predecessors of an arbitrary location ℓ'. DeadPBHeap and DeadPBThread reflect the fact that logical deallocation consumes full pointed-by-heap and pointed-by-thread assertions. Therefore, the assertions $\dagger\,\ell$ and $\ell \hookleftarrow_q L$ cannot coexist, except in the special case where q is zero, and the assertions $\dagger\,\ell$ and $\ell \Lleftarrow_p \Pi$ cannot coexist. However, in contrast with our earlier work [Madiot and Pottier, 2022; Moine et al., 2023], our deallocation witness *is* compatible with the points-to assertion. Indeed, our logical deallocation rule does not consume the points-to assertion. NoDanglingRootOut and NoDanglingRootIn both state that a dead location cannot be an ordinary root. A dead location can, however, be a temporary root: indeed, our logical deallocation rule allows deallocating a temporary root (§6.1).

5.10 Liveness-Based Cancellable Invariants

An Iris *invariant* [Jung et al., 2018b, §2.2] is written in the form $\boxed{\Phi}$.[1] It is a persistent assertion, whose meaning is that the assertion Φ in the rectangular box holds at all times. The assertion Φ itself is usually not persistent. An invariant can be temporarily *accessed* so as to gain access to the assertion Φ.

A *cancellable invariant* [Jung et al., 2018b, §7.1.3] is an invariant that comes with a teardown mechanism, allowing the user to recover ownership of the assertion Φ once the invariant is canceled. This is a one-shot mechanism: once a cancellable invariant is torn down, it cannot be restored. Naturally, accessing a cancellable invariant requires proving that this invariant has not been torn down already.

In IrisFit, a form of *liveness-based cancellable invariants* (LCIs, for short) naturally arises. An LCI is tied to a memory location ℓ, and remains in force as long as this location is live. When the location ℓ is logically deallocated, all LCIs associated with ℓ are implicitly torn down. Therefore, to access an LCI associated with the location ℓ, one must prove that this location is still live: that is, one must prove that $\dagger\,\ell$ implies $\ulcorner\text{False}\urcorner$. This can be done using any of the rules DeadPBHeap, DeadPBThread, NoDanglingRootOut, and NoDanglingRootIn in Figure 17. When the location ℓ is logically deallocated, the assertion Φ can be recovered at the same time. We have used LCIs to reason about closures (§9.4), about Treiber's stack (§12.5) and Michael and Scott's queue (§12.6).

The implementation of LCIs is simple. A liveness-based cancellable invariant tied to the location ℓ, whose content is the assertion Φ, is just $\boxed{\dagger\,\ell \vee \Phi}$, that is, a plain Iris invariant whose content is the disjunction $\dagger\,\ell \vee \Phi$. By proving that $\dagger\,\ell$ is contradictory, the user excludes the left-hand disjunct, therefore obtains access to Φ. In particular, when one is about to logically deallocate ℓ, the assertion $\ell \Lleftarrow \emptyset$ is at hand, so $\dagger\,\ell$ is excluded. One can therefore open the invariant, extract Φ, deallocate ℓ, and close the invariant by supplying $\dagger\,\ell$, keeping Φ. This technique is a somewhat unusual variation on the "golden idol" technique [Kaiser et al., 2017], with the persistent assertion $\dagger\,\ell$ in the role of the "bag of sand".

[1] Formally, an invariant also carries a *namespace*, a technicality that prevents the user from accessing the invariant twice and obtaining two copies of Φ at the same time. For simplicity, we hide namespaces here.

Program Logic: Reasoning Rules

Rachmaninoff, S. (1934).
Rhapsody on a Theme of Paganini.

In this chapter, we present the reasoning rules of IrisFit. Because most of our design is guided by the desire for a flexible logical deallocation rule, we begin with a presentation of this rule, in the simplified case where a single memory location is deallocated (§6.1). Then, we present the reasoning rules for terms (§6.2), devoting special attention to protected sections (§6.3) and to the Bind rule, whose form is non-standard (§6.4). The standard statement of the Bind rule can be recovered when the user enters a restricted mode where certain rules are disabled (§6.5). Finally, we present the general form of the logical deallocation rule, which can deallocate cycles (§6.6).

6.1 Logical Deallocation

As in the previous papers by Madiot and Pottier [2022] and Moine et al. [2023], a key aspect of IrisFit is to provide a *logical deallocation* rule. This rule produces space credits: by logically deallocating a memory block, the user recovers the space credits that were consumed when this block was allocated. It can be applied to a memory location ℓ as soon as one can prove that this memory location is eligible for collection *during the next garbage collection phase.*

As in the previous work cited above, *if ℓ is unreachable* then it can be logically deallocated. Furthermore, what is new in this thesis, *if ℓ is reachable only via temporary roots* (that is, via roots that will disappear by the time all protected sections are exited), then it can also be logically deallocated.

This reasoning rule may seem surprising, as it involves a form of anticipation: it exploits the fact that ℓ will be eligible for collection *once all protected sections have been exited*, yet it produces space credits *immediately*, at the point where the rule is applied. Intuitively, this is safe because a space credit serves to justify an allocation and (by design of our operational semantics) a large allocation request blocks until all protected sections have been exited. Hence, by the time extra free space is needed, any location that has been logically deallocated is effectively unreachable.

In §6.6, we present the general form of the logical deallocation rule, which can deallocate multiple memory locations at once, even if they form a cycle. Here, we present FreeOne, a simplified rule that is also useful in practice and that deallocates a single location ℓ:

$$\mathit{sizeof}\,\ell\,n \ * \ \ell \leftarrow\!\!\dashv_1 \emptyset \ * \ \ell \Leftarrow\!\!\dashv_1 \emptyset \quad \Rrightarrow \quad \Diamond\mathit{size}(n) \ * \ \dagger\,\ell \qquad \textsc{FreeOne}$$

FreeOne is expressed as a ghost update. It consumes three assertions: the "*sizeof*" assertion $\mathit{sizeof}\,\ell\,n$, the pointed-by-heap assertion $\ell \leftarrow\!\!\dashv_1 \emptyset$, and the pointed-by-thread assertion $\ell \Leftarrow\!\!\dashv_1 \emptyset$. The assertion $\mathit{sizeof}\,\ell\,n$ indicates that the memory block at address ℓ has size n. The assertion $\ell \leftarrow\!\!\dashv_1 \emptyset$ guarantees that ℓ has no predecessor in the heap, that is, no memory block contains the pointer ℓ. The assertion $\ell \Leftarrow\!\!\dashv_1 \emptyset$ guarantees that ℓ is not an ordinary root of any thread: that is, if ℓ is a root at all in a thread π, then it must be a temporary root for this thread (§2.5, §5.8). Together, the last two assertions imply that ℓ will be eligible for collection in the next garbage collection phase.

On the right-hand side of the ghost update, FreeOne produces two assertions, namely the recovered space credits $\Diamond n$ and the deallocation witness $\dagger\,\ell$. As noted earlier (§5.9), the latter

IfTrue
$$\frac{\{\Phi\}\ \pi\colon t_1\ \{\Psi\}}{\{\Phi\}\ \pi\colon \mathsf{if\ true\ then}\ t_1\ \mathsf{else}\ t_2\ \{\Psi\}}$$

IfFalse
$$\frac{\{\Phi\}\ \pi\colon t_2\ \{\Psi\}}{\{\Phi\}\ \pi\colon \mathsf{if\ false\ then}\ t_1\ \mathsf{else}\ t_2\ \{\Psi\}}$$

LetVal
$$\frac{\{\Phi\}\ \pi\colon [v/x]t\ \{\Psi\}}{\{\Phi\}\ \pi\colon \mathsf{let}\ x = v\ \mathsf{in}\ t\ \{\Psi\}}$$

Prim
$$\frac{v_1 \odot v_2 \xrightarrow{\mathsf{pure}} w}{\{\ulcorner \mathrm{True} \urcorner\}\ \pi\colon v_1 \odot v_2\ \{\lambda v.\ \ulcorner v = w \urcorner\}}$$

CallPtr
$$\frac{v = \mu_{\mathsf{ptr}} f.\ \lambda \vec{x}.\ t \qquad |\vec{x}| = |\vec{w}| \qquad \{\mathit{outside}\,\pi \ * \ \Phi\}\ \pi\colon [v/f][\vec{w}/\vec{x}]t\ \{\Psi\}}{\{\mathit{outside}\,\pi \ * \ \Phi\}\ \pi\colon (v\ \vec{w})_{\mathsf{ptr}}\ \{\Psi\}}$$

Val
$$\{\ulcorner \mathrm{True} \urcorner\}\ \pi\colon v\ \{\lambda v'.\ \ulcorner v' = v \urcorner\}$$

Poll
$$\{\mathit{outside}\,\pi\}\ \pi\colon \mathsf{poll}\ \{\lambda().\ \mathit{outside}\,\pi\}$$

Alloc
$$\frac{0 < n}{\left\{\begin{matrix} \Diamond n \\ \mathit{outside}\,\pi \end{matrix}\right\}\ \pi\colon \mathsf{alloc}\ \mathit{size}(n)\ \left\{\lambda \ell.\ \begin{matrix} \ell \mapsto_1 ()^n \\ \ell \mapsfrom_1 \emptyset \\ \ell \Mapsfrom_1 \{\pi\} \\ \mathit{outside}\,\pi \end{matrix}\right\}}$$

Load
$$\frac{0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{\left\{\begin{matrix} \ell \mapsto_p \vec{w} \\ v \Mapsfrom_{p'} \emptyset \end{matrix}\right\}\ \pi\colon \ell[i]\ \left\{\lambda v'.\ \begin{matrix} \ulcorner v' = v \urcorner \\ \ell \mapsto_p \vec{w} \\ v \Mapsfrom_{p'} \{\pi\} \end{matrix}\right\}}$$

Store
$$\frac{0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{\left\{\begin{matrix} \ell \mapsto_1 \vec{w} \\ v' \mapsfrom_q^{>0} \emptyset \end{matrix}\right\}\ \pi\colon \ell[i] \leftarrow v'\ \left\{\lambda().\ \begin{matrix} \ell \mapsto_1 [i := v']\vec{w} \\ v' \mapsfrom_q^{>0} \{+\ell\} \\ v \mapsfrom_0 \{-\ell\} \end{matrix}\right\}}$$

CASSuccess
$$\frac{0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{\left\{\begin{matrix} \ell \mapsto_1 \vec{w} \\ v' \mapsfrom_q^{>0} \emptyset \end{matrix}\right\}\ \pi\colon \mathsf{CAS}\ \ell[i]\ v\ v'\ \left\{\lambda b.\ \begin{matrix} \ulcorner b = \mathsf{true} \urcorner \\ \ell \mapsto_1 [i := v']\vec{w} \\ v' \mapsfrom_q^{>0} \{+\ell\} \\ v \mapsfrom_0 \{-\ell\} \end{matrix}\right\}}$$

CASFailure
$$\frac{0 \le i < |\vec{w}| \qquad \vec{w}(i) \ne v}{\{\ell \mapsto_p \vec{w}\}\ \pi\colon \mathsf{CAS}\ \ell[i]\ v\ v'\ \left\{\lambda b.\ \begin{matrix} \ulcorner b = \mathsf{false} \urcorner \\ \ell \mapsto_p \vec{w} \end{matrix}\right\}}$$

Fork
$$\frac{\mathit{dom}(M) = \mathit{locs}(t) \qquad (\forall \pi'.\ \{\mathit{outside}\,\pi' \ * \ M \Mapsfrom \{\pi'\} \ * \ \Phi\}\ \pi'\colon t\ \{\lambda().\ \mathit{outside}\,\pi'\})}{\{\mathit{outside}\,\pi \ * \ M \Mapsfrom \{\pi\} \ * \ \Phi\}\ \pi\colon \mathsf{fork}\ t\ \{\lambda().\ \mathit{outside}\,\pi\}}$$

Figure 18: Syntax-directed reasoning rules, without Bind and rules for protected sections

assertion is a permission to remove ℓ from the predecessor multisets of other locations. Thus, by iterated application of FreeOne, acyclic chains of unreachable blocks can be logically deallocated.

FreeOne can be applied to a reachable location if this location is a temporary root inside a protected section. Our logic thereby allows such a location to be read or written *post mortem*, after it has been logically deallocated. This is made possible by the fact that the points-to assertion survives logical deallocation. This pattern appears, for example, in the verification of Treiber's stack (§12.5).

Our logical deallocation rule differs from the one proposed in our earlier paper [Moine et al., 2023]. Indeed, while their rule consumes a points-to assertion for the location ℓ, ours does not. The points-to assertion is not needed to guarantee that the location is unreachable, nor is it needed to prevent a location from being deallocated twice. The size of the deallocated block is obtained in this thesis from the "*sizeof*" assertion, whereas in the previous paper this assertion did not exist, so the size was obtained from a points-to assertion.

6.2 Reasoning Rules for Terms

Figure 18 presents most of the syntax-directed reasoning rules of IrisFit, except for the rules that are specific to protected sections and the Bind rule, which are presented later on (§6.3, §6.4). In every rule, the thread identifier π represents the current thread, that is, the thread about which one is reasoning (§5.1).

IfTrue, IfFalse, LetVal, Prim and Val are standard rules.

CallPtr governs calls to (recursive, closed) functions, also known in this thesis as code pointers. Its only unusual aspect is the presence of the assertion $\mathit{outside}\,\pi$, which ensures that the current thread is currently outside a protected section. The presence of this assertion forbids function calls inside protected sections.

Similarly, Poll forbids polling points inside a protected section. Outside of this aspect, a polling point is a no-operation.

Alloc exhibits three differences with the allocation rule of Separation Logic. First, it requires and consumes $\mathit{size}(n)$ space credits, so as to pay for the space occupied by the new block. Second, the presence of the assertion $\mathit{outside}\,\pi$ forbids allocation inside a protected section. Third, in addition to a points-to assertion for the new block, allocation produces pointed-by-heap and pointed-by-thread assertions. These assertions indicate that there is initially no pointer from the heap to the new block, and that this new block is a root for the current thread (and only for this thread).

As in standard Separation Logic, Load requires a (fractional) points-to assertion for the memory location ℓ that is accessed. Furthermore, it requires a pointed-by-thread assertion $v \Lleftarrow_p \emptyset$ for the value v that is read from memory. This assertion is updated to $v \Lleftarrow_p \{\pi\}$, reflecting the fact that the value v becomes a root for the current thread.

As in standard Separation Logic, Store requires a full points-to assertion $\ell \mapsto_1 \vec{v}$ and produces an updated assertion $\ell \mapsto_1 [i := v']\vec{v}$. Furthermore, it performs bookkeeping of predecessor multisets, so as to reflect the fact that the value v that was stored in the field $\ell[i]$ is overwritten with the value v'. First, to reflect the *creation* of an edge from ℓ to the value v', an assertion of the form $v' \hookleftarrow_q \emptyset$ is changed to $v' \hookleftarrow_q \{+\ell\}$. Here, because ℓ has positive multiplicity in $\{+\ell\}$, the null-fraction invariant requires that q be positive; it cannot be 0. Second, to reflect the *deletion* of an edge from ℓ to the value v, the assertion $v \hookleftarrow_0 \{-\ell\}$ appears in the postcondition. As explained earlier (§5.6), this assertion is a permission to remove one occurrence of ℓ from a multiset of predecessors of v.

CASSuccess is similar to Store, but returns the Boolean value true rather than the unit value. Because a failed CAS does not modify the heap or create a new root, CASFailure is standard.

Fork reasons about the operation of spawning a new thread whose code is the term t. This operation must take place outside a protected section. Its impact on roots is as follows. Suppose, for a moment, that $\mathsf{fork}\,t$ is the last instruction in the parent thread. Then, the locations that occur in the term t cease to be roots of the parent thread π and become roots of the child thread π'. The reasoning rule reflects this intuition by updating a group of pointed-by-thread assertions. The iterated pointed-by-thread assertion $M \Lleftarrow \{\pi\}$ is taken away from the parent thread, and the updated assertion $M \Lleftarrow \{\pi'\}$ is transmitted to the child thread. M is a map of locations to fractions, whose domain is the set $\mathit{locs}(t)$. This is a form of *trimming*, similar in effect to the rules TrimPBThread and TrimInside.

If $\mathsf{fork}\,t$ is *not* the last instruction in the parent thread, then the user must use the reasoning rules Bind and Fork in combination. The interaction between the Bind rule and the "trimming" rules is discussed later on (§6.4, §6.5).

Still looking at Fork, an arbitrary assertion Φ is transmitted from the parent thread to the child thread. The assertion $\mathit{outside}\,\pi'$ is made available in the child thread, reflecting the fact that a new thread initially runs outside a protected section. The child thread t must be verified with the nontrivial postcondition $\mathit{outside}\,\pi'$, thereby disallowing a thread to terminate while inside a protected section.

Enter
$\{outside\,\pi\}\ \pi\colon \mathsf{enter}\ \{\lambda().\ inside\,\pi\,\emptyset\}$

Exit
$\{inside\,\pi\,\emptyset\}\ \pi\colon \mathsf{exit}\ \{\lambda().\ outside\,\pi\}$

LoadInside
$$\frac{0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{\left\{\begin{array}{c}\ell \mapsto_p \vec{w}\\ inside\,\pi\,T\end{array}\right\}\ \pi\colon \ell[i]\ \left\{\lambda v'.\ \begin{array}{c}\ulcorner v' = v \urcorner\ *\ \ell \mapsto_p \vec{w}\\ inside\,\pi\,(locs(v) \cup T)\end{array}\right\}}$$

StoreDead
$$\frac{0 \le i < |\vec{w}|}{\left\{\begin{array}{c}\ell \mapsto_1 \vec{w}\\ \dagger\,\ell\end{array}\right\}\ \pi\colon \ell[i] \leftarrow v'\ \ \{\lambda().\ \ell \mapsto_1 [i := v']\vec{w}\}}$$

CASSuccessDead
$$\frac{0 \le i < |\vec{w}| \qquad \vec{w}(i) = v}{\left\{\begin{array}{c}\ell \mapsto_1 \vec{w}\\ \dagger\,\ell\end{array}\right\}\ \pi\colon \mathsf{CAS}\,\ell[i]\,v\,v'\ \left\{\lambda b.\ \begin{array}{c}\ulcorner b = \mathsf{true} \urcorner\\ \ell \mapsto_1 [i := v']\vec{w}\end{array}\right\}}$$

Figure 19: Reasoning rules: protected-section-specific rules

In our Coq formalization, the postconditions of many reasoning rules contain a *later credit* [Spies et al., 2022]. Later credits play a role in eliminating the "later" modality. They are orthogonal to the main concern of this thesis, namely the analysis of space complexity, so we hide them in the presentation of our reasoning rules. We do explain how later credits are used in our case study of the async-finish library (§12.4).

6.3 Reasoning about Protected Sections

Within a protected section, the reasoning rules presented in the previous section (§6.2) can still be used, except for CallPtr, Alloc, and Poll, which require the assertion *outside* π. In addition, a number of reasoning rules, shown in Figure 19, specifically concern protected sections.

Enter allows entering a protected section. This rule transforms the assertion *outside* π into the assertion *inside* $\pi\,\emptyset$, thereby witnessing that the current thread is now inside a protected section and has no temporary roots.

Conversely, Exit allows exiting a protected section. By consuming the assertion *inside* $\pi\,\emptyset$, this rule requires the user to prove that the current thread has no remaining temporary roots.

LoadInside allows reading a value v from a location ℓ in the heap. The locations that appear in the value v become temporary roots of the current thread: the assertion *inside* $\pi\,T$ is updated to *inside* $\pi\,(T \cup locs(v))$. In contrast with Load, no pointed-by-thread assertion is required or updated. In fact, the location ℓ or some locations in the set $locs(v)$ might be logically deallocated already.

StoreDead allows writing a logically deallocated block. The rule requires and updates a points-to assertion. A deallocation witness $\dagger\,\ell$ is also required. Compared with Store, no pointed-by-heap assertion is required or updated. Indeed, there is no need to do so. Pointed-by-heap assertions keep track of which blocks are reachable via ordinary roots; but, because the block at address ℓ is logically deallocated, it is not reachable via ordinary roots. This is reminiscent of CleanPBHeap.

Although StoreDead does not require an "*inside*" assertion, it can be used only inside a protected section. Indeed, the rule applies to a store instruction $\ell[i] \leftarrow v'$, where the address ℓ occurs. This means that ℓ is a root, yet ℓ is also logically deallocated. This is possible only if the current thread is currently inside a protected section. Indeed, outside a protected section, a logically deallocated location cannot be a root (NoDanglingRootOut, §5.9).

CASSuccessDead is analogous to StoreDead. It concerns a successful CAS instruction on a logically deallocated location. Because a failed CAS does not write anything, the rule CASFailure can be applied to a logically deallocated location without change.

BIND

$$\frac{dom(M) = locs(K) \qquad \{\Phi\}\ \pi\colon t\ \{\Psi'\} \qquad \forall v.\ \{M \Lleftarrow \{\pi\}\ *\ \Psi'\,v\}\ \pi\colon K[v]\ \{\Psi\}}{\{M \Lleftarrow \{\pi\}\ *\ \Phi\}\ \pi\colon K[t]\ \{\Psi\}}$$

Figure 20: Reasoning rules: the BIND rule

6.4 Reasoning under Evaluation Contexts

A proof in Separation Logic is traditionally carried out under an unknown context. That is, one reasons about a term t without knowing in what evaluation context K this term is placed. There are specific points in the proof where this unknown context grows and shrinks. As an archetypal example, consider the sequencing construct $\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2$. To reason about this construct, one first focuses on the term t_1, thereby temporarily forgetting the frame $\mathsf{let}\,x = \Box\,\mathsf{in}\,t_2$, which is pushed onto the unknown context. After the verification of t_1 is completed, this focusing step is reversed: the frame $\mathsf{let}\,x = \Box\,\mathsf{in}\,t_2$ is popped and one continues with the verification of t_2. These focusing and defocusing steps are described by the "BIND" rule [Jung et al., 2018b, §6.2].

In our setting, however, a complication arises. An evaluation context contains memory locations. When one applies the BIND rule, so as to temporarily forget about this evaluation context, one must still somehow record that these locations are roots. We use pointed-by-thread assertions for this purpose.

Suppose we wish to decompose the sequence $\mathsf{let}\,x = t_1\,\mathsf{in}\,t_2$ into a subterm t_1 and an evaluation context $\mathsf{let}\,x = \Box\,\mathsf{in}\,t_2$. For simplicity, let us further assume that $locs(t_2)$ is a singleton set $\{\ell\}$. This implies that, while t_1 is being executed, the location ℓ is a root. In this specific case, our BIND rule takes the following form:

PARTICULAR CASE OF BIND

$$\frac{locs(t_2) = \{\ell\} \qquad \{\Phi\}\ \pi\colon t_1\ \{\Psi'\} \qquad \forall v.\ \{\ \ell \Lleftarrow_p \{\pi\}\ \ *\ \Psi'\ v\}\ \pi\colon [v/x]t_2\ \{\Psi\}}{\{\ \ell \Lleftarrow_p \{\pi\}\ \ *\ \Phi\}\ \pi\colon \mathsf{let}\,x = t_1\,\mathsf{in}\,t_2\ \{\Psi\}}$$

What is unusual, compared with the standard BIND rule of Separation Logic, is that the fractional pointed-by-thread assertion $\ell \Lleftarrow_p \{\pi\}$ (highlighted in the above rule) is required in the beginning, taken away from the user while focusing on the term t_1, and given back to the user once she is done reasoning about t_1 and ready to reason about t_2. In other words, this assertion is *forcibly framed out* while reasoning about t_1.

The assertion $\ell \Lleftarrow_p \{\pi\}$ records that ℓ is a root in thread π. By taking it away from the user and by giving it back once she is done reasoning about t_1, we ensure that the information that "ℓ is a root in thread π" is carried up to this point and cannot be prematurely destroyed.

What could go wrong if we did not do this? Then, the user would be allowed to keep the *full* pointed-by-thread assertion $\ell \Lleftarrow_1 \{\pi\}$ while reasoning about t_1. Technically, the user would do so by instantiating Φ with $\ell \Lleftarrow_1 \{\pi\}$ in the BIND rule. Then, the user would focus on establishing the first premise, $\{\ell \Lleftarrow_1 \{\pi\}\}\ \pi\colon t_1\ \{\Psi'\}$. Now suppose $\ell \notin locs(t_1)$, that is, ℓ does not occur in t_1. Then, the user could apply TRIMPBTHREAD to transform the assertion $\ell \Lleftarrow_1 \{\pi\}$ into $\ell \Lleftarrow_1 \emptyset$. Oops! The assertion $\ell \Lleftarrow_1 \emptyset$ means that ℓ *is not* a root. Yet ℓ really *is* still a root, as it occurs in the evaluation context that has been abstracted away, namely $\mathsf{let}\,x = \Box\,\mathsf{in}\,t_2$.

Besides TRIMPBTHREAD, two reasoning rules, namely FORK and TRIMINSIDE, involve a form of "trimming" of sets of thread identifiers. The soundness of these rules relies on the fact that BIND forcibly frames out fractional pointed-by-thread assertions.

The general form of our BIND rule, shown in Figure 20, extends this idea to an arbitrary evaluation context K, in which an arbitrary number of locations may occur. Then, for every location in $locs(K)$, a fractional pointed-by-thread assertion is forcibly framed out.

$$\frac{\{\Phi\}\ \text{✖}/\pi\colon t\ \{\Psi\}}{\{\Phi\}\ m/\pi\colon t\ \{\Psi\}}\ \textsc{SwitchMode} \qquad \frac{\{\Phi\}\ \text{✖}/\pi\colon t\ \{\Psi'\} \qquad \forall v.\ \{\Psi'\ v\}\ m/\pi\colon K[v]\ \{\Psi\}}{\{\Phi\}\ m/\pi\colon K[t]\ \{\Psi\}}\ \textsc{BindNoTrim}$$

Figure 21: Reasoning rules: additional mode-specific rules

6.5 Locally Trading Trimming for a Simpler and More Powerful Bind Rule

Forcing pointed-by-thread assertions to be framed out at each application of Bind is cumbersome, and can be restrictive, as there are situations where no pointed-by-thread assertion is at hand. (An example appears later on in this section.) Fortunately, such forced framing is unnecessary if the user promises not to exploit any of the trimming rules TrimPBThread, Fork and TrimInside. Thus, we introduce a mode that the user may choose to enter at any time, in which the trimming rules are disabled and, in exchange, a simpler, more powerful Bind rule is made available.

We parameterize IrisFit triples with a *mode* m, which is either the normal mode ✂ or the "no trim" mode ✖. Thus, in general, our triples have the form $\{\Phi\}\ m/\pi : t\ \{\Psi\}$, and our custom ghost update has the form $\Phi\ {}^{\pi}\!\!\Rrightarrow^{V}_{m}\ \Phi'$. All of the reasoning rules presented so far are polymorphic in the mode, except for the trimming rules TrimPBThread, Fork, and TrimInside, which are disabled in "no trim" mode. The public specification of a function is always stated in the normal mode. The "no trim" mode is intended for local use, inside the body of a function. The "no trim" mode is an adaptation of the "NoFree" mode of our earlier paper [Moine et al., 2023].

Figure 21 presents two new reasoning rules, SwitchMode and BindNoTrim, which allow entering "no trim" mode and taking advantage of it.

When read from bottom to top, SwitchMode lets the user locally enter "no trim" mode, whenever she so wishes, in a subproof. When read from to top to bottom, this rule asserts that if a triple holds in "no trim" mode then it also holds in normal mode. Indeed, every reasoning rule that is available in "no trim" mode is available in normal mode as well.

BindNoTrim is the standard Bind rule of Separation Logic, but imposes a switch to "no trim" mode ✖ in its left-hand premise. Thus, unlike our Bind rule, it does *not* force pointed-by-thread assertions to be framed out. Because of this, it must disable the trimming rules while the user reasons about the subterm t.

We remark that, inside a protected section, one can switch to "no trim" mode without loss of expressive power. Indeed, there, the trimming rules are never needed. Fork is forbidden inside protected sections; the effect of TrimPBThread can be simulated by AddTemporary; and all uses of TrimInside can be postponed until the protected section is about to be exited.

At a high level, BindNoTrim is needed for reasoning about code that, within a protected section, reads or writes in a location after it has been logically deallocated. Indeed, in this case, Bind can be too restrictive. To illustrate this case, consider the following code, where we assume that the location r is not accessible via the heap and is not known to any thread other than the current thread:

$$\mathsf{enter}\,;\ (\mathsf{let}\ x = t\ \mathsf{in}\ x + r[0])\,;\ \mathsf{exit}$$

Just after entering the protected section, the user may wish to logically deallocate r, in order to recover the corresponding space credits without waiting for the end of the protected section. In this case, just after entering the protected section, she would use AddTemporary to obtain a pointed-by-thread assertion $r \Lleftarrow \emptyset$, then use FreeOne to logically deallocate r, consuming this pointed-by-thread assertion. Thereafter, the user may wish to decompose the

$$\ulcorner\text{True}\urcorner \twoheadrightarrow \emptyset \clubsuit^{0} \emptyset \qquad \textsc{CloudEmpty}$$

$$\begin{array}{r} P \clubsuit^{n} D \;*\; \mathit{sizeof}\,\ell\,m \\ \ell \Lleftarrow_1 \emptyset \;*\; \ell \hookleftarrow_1 L \;*\; \ulcorner\mathsf{NoNegative}(L)\urcorner \end{array} \twoheadrightarrow (P \cup L) \clubsuit^{(n+m)} (D \cup \{\ell\}) \qquad \textsc{CloudAdd}$$

$$\ulcorner P \subseteq D\urcorner \;*\; P \clubsuit^{n} D \Rrightarrow \Diamond n \;*\; \mathop{\Huge *}_{\ell \in D} \dagger\ell \qquad \textsc{CloudFree}$$

Figure 22: Reasoning rules: logical deallocation

let construct. Yet, the BIND rule cannot be used, as it would require a (fractional) pointed-by-thread assertion for r, which no longer exists, because the fraction 1 was consumed by FREEONE. Fortunately, BINDNOTRIM is applicable.

6.6 Logical Deallocation of Cycles

Figure 22 presents our rules for deallocating an unreachable heap *fragment*, as opposed to a single location. This fragment may contain an arbitrary number of heap blocks, which may point to each other in arbitrary ways. In particular, these pointers may form one or more cycles.

These rules make use of the "cloud" assertion $P \clubsuit^{n} D$, whose parameters P (for "predecessors") and D (for "domain") are sets of locations, and whose parameter n is a natural integer. This assertion means that the memory blocks at locations D have total size n, that the locations D are not roots in any thread, and that these locations can be reached only via the locations P. We refer to P also as the *entry points* of the cloud.

If $P \subseteq D$ holds, then the locations in the set D are reachable only via D itself. In other words, the set D is closed under predecessors. This means that the locations in the set D are in fact *unreachable*, and can safely be logically deallocated. This explains the side condition $P \subseteq D$ in the logical deallocation rule CLOUDFREE. We do not require $P \subseteq D$ to hold at all times: while constructing large "cloud" assertions out of smaller "cloud" assertions, one must allow the sets P and D to be unrelated.

Figure 22 presents two cloud construction rules as well as the logical deallocation rule, which consumes a cloud.

Out of nothing, CLOUDEMPTY creates an empty cloud $\emptyset \clubsuit^{0} \emptyset$.

CLOUDADD adds the memory block at location ℓ to an existing cloud $P \clubsuit^{n} D$. This consumes the full pointed-by-thread assertion $\ell \Lleftarrow_1 \emptyset$, which guarantees that ℓ is not a root in any thread, and the full pointed-by-heap assertion $\ell \hookleftarrow_1 L$, which guarantees that L contains all of the predecessors of the location ℓ in the heap. A "*sizeof*" assertion determines the size m of the memory block at address ℓ. CLOUDADD produces an extended cloud, where L is added to the cloud's entry points, m is added to the cloud's size, and ℓ is added to the cloud's domain.

CLOUDFREE logically deallocates a cloud that is closed under predecessors, that is, a cloud such that $P \subseteq D$ holds. The "cloud" assertion is consumed. In exchange for it, the rule produces n space credits, where n is the size of the cloud. Furthermore, it produces a deallocation witness for every location in the cloud.

The rule FREEONE that was presented earlier (§6.1) is derived from the rules in Figure 22.

Chapter 7

Safety, Liveness and Core Soundness Theorems

Dave Brubeck Quartet (1959).
Take Five.

In this chapter, we state a safety theorem and a liveness theorem about programs that have been verified using IrisFit.

The *safety* theorem (§7.1) guarantees that no thread crashes. More precisely, it states that if a thread is enabled (in the sense of §4.8), then this thread is not stuck: either it has reached a value or it can make a step.

The *liveness* theorem (§7.2) guarantees that no thread can be blocked forever. More precisely, under the assumption that there is a polling point in front of every function call, we prove that every thread is eventually enabled. Furthermore, we prove that inserting a polling point in front of every function call preserves safety. Thus, after a source program without polling points has been verified with IrisFit, one can let a compiler automatically insert polling points, and obtain both safety and liveness for this instrumented program.

We also present a *core soundness* theorem (§7.3), from which the previous theorems follow. The core soundness theorem spells out the guarantee that is offered by IrisFit when a LambdaFit program is executed under a simplified *oblivious semantics* that has neither garbage collection nor blocking instructions.

We then sketch the proof of these theorems. First, we show that the core soundness theorem implies the safety theorem (§7.4). Second, we show that the safety theorem implies the liveness theorem (§7.5). Chapter 8 is dedicated to the proof of the core soundness theorem itself. The proofs of the lemmas involved are sometimes fairly technical; when omitted, these proofs can be found in our mechanization [Moine, 2024].

7.1 Safety

A concurrent Separation Logic typically comes with a safety guarantee, formulated in the form: "*no thread can crash*". A slightly more precise statement is: "*always, every thread is not stuck*". In other words, in every reachable configuration of the system, every thread either has terminated or is able to make a reduction step. A thread that has not reached a value and is unable to make a step is *stuck*: by convention, this is considered an undesirable situation, akin to a crash.

In our setting, however, this statement must be amended, because LambdaFit has blocking instructions. A blocking instruction is sometimes *disabled* (§4.8), therefore unable to make a step; yet, this situation is not considered a crash.

First, our amended safety guarantees that: "always, every thread *that is enabled* is not stuck". A thread that is not enabled is considered blocked: this is a normal situation. Second, our amended safety guarantees two additional properties which together implies that reachable configurations are not globally stuck, that is, "always, at least one thread or the GC can take a step".

Figure 23 presents the formal definitions we need for our safety theorem. First, the property NotStuck c π asserts that the thread π is not stuck in the configuration c: either the thread associated to π in θ has reached a value and is outside a protected section (Not-

NotStuckVal
$$\frac{\theta(\pi) = (v, \mathsf{Out})}{\mathsf{NotStuck}\ (\theta, \sigma)\ \pi}$$

NotStuckStep
$$\frac{c \xrightarrow{\text{enabled action}}_{\pi} c'}{\mathsf{NotStuck}\ c\ \pi}$$

Safe
$$\frac{\forall \pi.\ \mathsf{Enabled}\ c\ \pi \implies \mathsf{NotStuck}\ c\ \pi}{\mathsf{Safe}\ c}$$

GloballyNotStuck
$$\frac{(\exists c'.\ (\theta, \sigma) \xrightarrow{\text{main}} c') \ \vee\ (\forall t\, g.\ (t, g) \in \theta \implies \exists v.\ t = v)}{\mathsf{GloballyNotStuck}\ (\theta, \sigma)}$$

Always
$$\frac{\forall c'.\ c \xrightarrow{\text{main}}{}^{*} c' \implies P\ c'}{\mathsf{Always}\ P\ c}$$

PollAndAllocAreOut
$$\frac{\forall t\, g.\ (t, g) \in \theta\ \wedge\ (\mathsf{IsPoll}\ t \vee \exists n.\ \mathsf{IsAlloc}\ n\ t) \implies g = \mathsf{Out}}{\mathsf{PollAndAllocAreOut}\ (\theta, \sigma)}$$

GCCanMakeEveryAllocFit
$$\frac{\mathsf{AllOutside}\ c \implies (\ \mathsf{EveryAllocFits}\ c \vee \exists c'.\ c \xrightarrow{\text{enabled action}}_{\text{gc}} c'\ \wedge\ \mathsf{EveryAllocFits}\ c'\ )}{\mathsf{GCCanMakeEveryAllocFit}\ c}$$

StronglySafe
$$\frac{\mathsf{Safe}\ c \qquad \mathsf{PollAndAllocAreOut}\ c \qquad \mathsf{GCCanMakeEveryAllocFit}\ c}{\mathsf{StronglySafe}\ c}$$

Figure 23: Predicates used to state the soundness theorem

StuckVal), or the thread π is enabled and can take an action step (NotStuckStep). The property Safe c asserts that all threads that are enabled are not stuck (Safe). We can paraphrase the Safe c property by saying that every thread is either not stuck, or not enabled.

The proposition Always P c asserts that the execution of c reaches only configurations that satisfy the predicate P (Always). A notable configuration is the *initial configuration* of a program t, which is defined as the thread pool containing solely the thread t outside a protected section, and the empty store.

We could state a theorem asserting that Always Safe $(init(t))$ holds for a program t verified in IrisFit. However, we aim for a stronger theorem Always StronglySafe $(init(t))$, where StronglySafe extends Safe with two additional properties presented in Figure 23: the properties PollAndAllocAreOut and GCCanMakeEveryAllocFit. The property PollAndAllocAreOut c asserts that, for every thread in c, if its next instruction is a polling point or an allocation, then this thread is outside a protected section. The property GCCanMakeEveryAllocFit c asserts that, if all threads are outside protected sections, then either every allocation fits, or there exists a step of the GC after which every allocation fits.

The proposition StronglySafe c is a conjunction of Safe c and the above two properties (StronglySafe). Our soundness theorem asserts that a program t verified with our program logic with a precondition containing S space credits and the outside assertion is always strongly safe, starting from the initial configuration associated with t.

Theorem 1 (Safety)**.** *Assume that, for every thread identifier π, the following triple holds:*

$$\{\Diamond S * outside\ \pi\}\ \pi : t\ \{\lambda_.\ outside\ \pi\}$$

Then Always StronglySafe $(init(t))$ holds.

Proof. The proof of this theorem is sketched in §7.3. □

Figure 23 also presents the GloballyNotStuck c property (GloballyNotStuck). This property asserts that the configuration $c = (\theta, \sigma)$ either can take a main step—that is, either a thread or the GC can take a step—or every thread has reached a value. The following lemma shows that a strongly safe configuration is globally not stuck, and is used for the proof of the liveness theorem (§7.5).

Lemma 3 (Globally Not Stuck). *If StronglySafe c holds then GloballyNotStuck c holds.*

Proof. If all threads in c consist of a value, GloballyNotStuck c holds. Otherwise, there exists one thread, call its identifier π, that is not a value.

There are two cases, depending on whether EveryAllocFits c holds.

First case, EveryAllocFits c holds. By applying Lemma 1 to this assumption, we deduce that all threads are enabled. Hence, in particular, the thread π is enabled. By definition, the assumption StronglySafe c implies Safe c, which implies that the thread π can take a step. Therefore, GloballyNotStuck c holds.

Second case, $\neg$EveryAllocFits c holds. There are two subcases, depending on whether AllOutside c holds. First subcase, AllOutside c holds. By definition, StronglySafe c implies GCCanMakeEveryAllocFit c, which states that either every allocation fits, or the GC can take a step after which every allocation fits. Since we are in case that not every allocation fits, we deduce that the GC can take a step, hence GloballyNotStuck c holds. Second subcase, $\neg$AllOutside c holds, hence there exists a thread π' inside a protected section. By definition, the assumption StronglySafe c implies PollAndAllocAreOut c. Hence, because π' is inside a protected section, it cannot be at a polling point or an allocation. Therefore Enabled $c\,\pi'$ holds. From Safe c, we deduce that NotStuck $c\,\pi'$ holds, hence GloballyNotStuck c holds. □

Interestingly, the mere Safe c property does not suffice to ensure that c is globally not stuck. Indeed, the property Safe c trivially holds if no thread of c is enabled.

We now pose an auxiliary lemma asserting that Always is monotonic.

Lemma 4. *If, for every configuration c, $P_1\,c$ implies $P_2\,c$, then, for every configuration c, Always P_1 c implies Always P_2 c.*

Hence, if a program t is verified in IrisFit, Safety Theorem 1 shows that it is always strongly safe, and by combining lemmas Lemma 3 and Lemma 4, we can deduce that it is safe and globally not stuck.

The reader may wonder why we mention the stronger StronglySafe property if, in the end, the property of interest is the weaker GloballyNotStuck. The reason is *compositionality*: as we will see in §7.5, we show that the insertion of polling points preserves StronglySafe, and hence ensures GloballyNotStuck by Lemma 3. Therefore, we can use Safety Theorem 1 to prove that if a program t is verified in IrisFit, then, even after inserting polling points, the program t reaches only globally not stuck configurations. A direct proof that the insertion of polling points preserves GloballyNotStuck seems more difficult.

7.2 Liveness

The Safety Theorem 1 guarantees that no thread can crash, but allows a thread to become blocked. Therefore, a liveness guarantee is also desirable: one would like to be assured that *always, every thread is eventually enabled*. In other words, there is no execution scenario where certain threads remain blocked forever, in the sense that they never become enabled after some point.

In fact, we are able to offer a stronger guarantee: we prove that *always, eventually, every allocation fits*. In other words, in every execution scenario, infinitely often, the system reaches a point where no allocation request is blocked due to a lack of memory. This property is indeed stronger, because it captures the fact that, at a certain point, *every* thread is enabled at once. (By Lemma 1, the property *always, eventually, every allocation fits* implies that *always, eventually, all threads are enabled* at the same time; which, in turn, implies that *always, every thread is eventually enabled.*)

However, our liveness guarantee is subject to a condition: the program must contain *enough polling points*. To see why this is necessary, imagine a program where thread A is blocked on a large allocation request and thread B is running in an infinite loop, without

$$\frac{P\ c}{\mathsf{AfterAtMost}\ n\ P\ c}\ \textsc{HoldsNow} \qquad \frac{n > 0 \qquad (\exists c'.\ c \xrightarrow{\mathsf{main}} c') \qquad (\forall c'.\ c \xrightarrow{\mathsf{main}} c' \implies \mathsf{AfterAtMost}\ (n-1)\ P\ c')}{\mathsf{AfterAtMost}\ n\ P\ c}\ \textsc{HoldsAfter} \qquad \frac{\mathsf{AfterAtMost}\ n\ P\ c}{\mathsf{Eventually}\ P\ c}\ \textsc{Eventually}$$

Figure 24: Predicates used to state the liveness theorem

allocating memory or encountering a polling point. Then, there exists a scenario where thread B runs forever, the GC is never invoked, and thread A never becomes unblocked. Thus, the desired liveness property does not hold. However, suppose that a polling point is inserted in the loop: thread B is not allowed to proceed past this polling point. Then, in every scenario, a garbage collection step eventually takes place, at which time both thread A and thread B become unblocked.

How can one tell whether a program has enough polling points? Or, in other words, where polling points must be inserted so that the program has enough polling points? We propose a simple approach, which is to *insert a polling point in front of every function call.*[1] This insertion strategy ensures that every thread must reach a polling point in a bounded number of steps. Prior to inserting polling points, we require the program to be in administrative normal form (ANF), as defined below.

Definition 1 (Administrative Normal Form)**.** *A program t is in administrative normal form if, in every function call of t, the function itself and the actual arguments are variables or values, as opposed to arbitrary expressions.*

The ANF condition guarantees that the polling point that is inserted in front of the function call is executed *after* the actual arguments have been computed and *just before* the function is invoked. Otherwise, inserting a polling in front of function calls is not safe. For example, the program $\mathsf{enter};\ (f\ \mathsf{exit})_{\mathsf{ptr}}$ is always safe (provided that f is safe), but the program $\mathsf{enter};\ \mathsf{poll};\ (f\ \mathsf{exit})_{\mathsf{ptr}}$ is not, as it contains a polling point inside a protected section.

Up to this administrative ANF condition, we prove that our polling point insertion strategy preserves safety and ensures liveness. We refer to this polling point insertion strategy as *addpp*.

Definition 2 (addpp)**.** *Let t be a term. The term addpp(t) is obtained by inserting a polling point in front of every function call in the term t.*

Figure 24 introduces a few auxiliary predicates that appear in the statement of the liveness theorem. The proposition $\mathsf{AfterAtMost}\ n\ P\ c$ means that, out of the configuration c, *every* execution path reaches a configuration that satisfies P in at most n steps. The proposition $\mathsf{AfterAtMost}\ n\ P\ c$ is inductively defined by the rules HoldsNow and HoldsAfter. HoldsAfter guarantees not only that the predicate continues to hold after any possible step, but also that there exists such a step. The proposition $\mathsf{Eventually}\ P\ c$ means that in a bounded number of steps, out of the configuration c, *every* execution path reaches a configuration that satisfies P. It is defined by the rule Eventually, via an existential quantification over n. (The explicit depth bound n provides a stronger guarantee than just the plain inductive. Indeed, AfterAtMost is *infinitely branching* due to the non-determinism of allocation, and one

[1] LambdaFit does not have loops: instead, loops must be simulated via tail-recursive functions. Thus, inserting a polling point in front of every function call effectively implies inserting a polling point inside every loop as well. Incidentally, because function calls are forbidden inside protected sections, a polling point is never inserted into a protected section, satisfying our restriction that polling points in protected sections are forbidden. Our polling point insertion strategy is loosely inspired by the (undocumented) polling point insertion strategy of the OCaml compiler. The OCaml compiler inserts a polling point at the beginning of every function (except possibly small leaf functions), inside every loop, and views memory allocation instructions as polling points.

$$\frac{c \xrightarrow{\mathsf{action}}_{\pi} c'}{c \xrightarrow{\mathsf{oblivious}} c'}\ \textsc{Oblivious} \qquad \frac{\theta(\pi) = (v, \mathsf{Out})}{\mathsf{NotStuckOblivious}\,(\theta,\sigma)\,\pi}\ \textsc{NotStuckObliviousVal} \qquad \frac{c \xrightarrow{\mathsf{action}}_{\pi} c'}{\mathsf{NotStuckOblivious}\,c\,\pi}\ \textsc{NotStuckObliviousStep}$$

Figure 25: The oblivious reduction relation and associated predicates

cannot extract a depth bound from an infinitely branching inductive [Bertot and Castéran, 2004].)

Our final theorem states that if the program t has been verified using IrisFit, under the exact same conditions as in the Safety Theorem 1, then the program $addpp(t)$, in which enough polling points have been inserted, is safe and live.

Theorem 2 (Combined Safety and Liveness after Polling Point Insertion)**.** *Suppose that the term t is in administrative normal form. Assume that, for every thread identifier π, the following triple holds:*

$$\{\Diamond S * outside\,\pi\}\ \pi : t\ \{\lambda_.\ outside\,\pi\}$$

Let t' stand for the term $addpp(t)$. Then, both of the following propositions hold:

1. *Always StronglySafe $(init(t'))$*
2. *Always (Eventually EveryAllocFits) $(init(t'))$.*

Proof. The proof of this theorem is sketched in §7.5. □

This statement reflects how we envision the practical use of IrisFit. We expect the user to verify a program t in which polling points have not yet been inserted. Thus, the user need not know where polling points will be placed; in fact, the user need not be aware of polling points at all. As explained earlier, the uninstrumented verified program t enjoys safety but not liveness. Nevertheless, the theorem guarantees that, once enough polling points have been inserted, the program becomes safe and live.

7.3 The Oblivious Semantics and the Core Soundness

A provocative yet fundamental remark is that IrisFit has nothing to do with garbage collection. Indeed, its deallocation rule is purely logical. More generally, its reasoning rules are independent of *when* garbage collection takes place, or *whether* it takes place at all. In reality, IrisFit is concerned with the *live heap space* of a program, that is, the sum of the sizes of the reachable blocks.

The Safety Theorem 1 and the Combined Safety and Liveness Theorem 2 both follow from a *core soundness* result, which is expressed with respect to the *oblivious semantics*, an alternative semantics in which no garbage collection takes place and no instructions are blocking (§2.2). This core soundness theorem states that IrisFit offers safety and maximum live heap space guarantees.

This oblivious semantics takes the form of an *oblivious reduction* relation $c \xrightarrow{\mathsf{oblivious}} c'$, defined by the rule Oblivious in Figure 25. The relation $c \xrightarrow{\mathsf{oblivious}} c'$ holds if and only if one thread π of c takes a step. There are three interesting facts about the oblivious relation. First, no instruction is blocking: instructions are always enabled. Second, the oblivious relation does *not* depend on a parameter S: the size of the heap does not matter. Third, there is no garbage collection step: there is no need to collect space.

The transitive closure of the oblivious reduction relation interleaves the actions of all threads in arbitrary ways.

In this setting, we must redefine what it means for a thread to be "not stuck". The proposition $\mathsf{NotStuckOblivious}\, c\, \pi$, also defined in Figure 25, serves this purpose. A thread is not stuck if either it has reached a value outside a protected section, or it can make a step.

Let us write $livespace(R, \sigma)$ for the total size of the fragment of the store σ that is reachable from the roots R. Let us write $livespace(c)$ for the live heap space of the configuration $c = (\theta, \sigma)$. It is defined by $livespace((\theta, \sigma)) = livespace(locs(\theta), \sigma)$.

Our core soundness theorem states that, for a program t verified with S space credits, in every configuration that is reachable (with respect to the oblivious semantics), the following two properties hold. First, no thread is stuck. Second, if every thread is outside a protected section, then the live heap size is at most S.

Theorem 3 (Core Soundness)**.** *Assume that, for every thread identifier π, the following triple holds.*

$$\{\Diamond S * outside\, \pi\}\ \pi\colon t\ \{\lambda_.\ outside\, \pi\}$$

Then, for every configuration c such that $init(t) \xrightarrow{\mathsf{oblivious}}{}^{*} c$,

1. *for every identifier π of a thread in c, the property $\mathsf{NotStuckOblivious}\, c\, \pi$ holds;*
2. *$\mathsf{AllOutside}\, c$ implies $livespace(c) \le S$.*

Proof. The proof of this theorem is sketched in §8. □

This statement may seem surprisingly weak, as it offers no guarantee about $livespace(c)$ at a time where $\mathsf{AllOutside}\, c$ does not hold, that is, at a time where at least one thread is inside a protected section. Moreover, this statement offers a safety guarantee; it does not offer any liveness guarantee. Nevertheless, as we will see, this core soundness theorem is sufficiently strong to derive both the Safety Theorem 1 and the Combined Safety and Liveness Theorem 2.

7.4 Deriving Safety from Core Soundness

We begin with some auxiliary lemmas of which we omit the proofs for brevity.

The following lemma connects a reduction chain of the main reduction relation (with garbage collection) to a reduction chain of the oblivious reduction relation (without garbage collection), with a single garbage collection step "at the end". Intuitively, because the oblivious semantics has no space limit, the GC can wait.

Lemma 5. *If $c \xrightarrow{\mathsf{main}}{}^{*} (\theta, \sigma)$ then, there exists a store σ_0 such that $c \xrightarrow{\mathsf{oblivious}}{}^{*} (\theta, \sigma_0)$ and $locs(\theta) \vdash \sigma_0 \xrightarrow{\mathsf{gc}} \sigma$.*

The following lemma connects the notion of "not being stuck" from the core soundness theorem with the one of the safety theorem, with a possible garbage collection.

Lemma 6. *If $locs(\theta) \vdash \sigma_0 \xrightarrow{\mathsf{gc}} \sigma$ and if $\mathsf{Enabled}\, (\theta, \sigma)\, \pi$, then $\mathsf{NotStuckOblivious}\, (\theta, \sigma_0)\, \pi$ implies $\mathsf{NotStuck}\, (\theta, \sigma)\, \pi$.*

The following lemma asserts that a garbage collection always preserves or reduces the size of a store.

Lemma 7. *If $R \vdash \sigma \xrightarrow{\mathsf{gc}} \sigma'$, then $size(\sigma') \le size(\sigma)$.*

We then define the *full garbage collection* $collect(R, \sigma)$ of a store σ from a set of roots R.

Definition 3 (collect)**.** *We write $collect(R, \sigma)$ for the store that is obtained by re-binding to $\lightning$ every location in σ that is unreachable from R.*

The following lemma asserts that the live heap space corresponds to the size of the heap after a full garbage collection.

Lemma 8. *livespace*$(R, \sigma) =$ *size*(*collect*(R, σ))

The following lemma asserts that the *collect* function is monotonically increasing with respect to the GC pre-order.

Lemma 9. *If* $R \vdash \sigma \xrightarrow{\mathsf{gc}} \sigma'$ *then* $R \vdash collect(R, \sigma) \xrightarrow{\mathsf{gc}} collect(R, \sigma')$

We are now ready to sketch the proof of Safety Theorem 1, assuming that the Core Soundness Theorem 3 holds.

Proof. Let t be a program such that the premise of Safety Theorem 1 holds, that is, (Fact 1) for every thread identifier π, the triple $\{\Diamond S * outside\,\pi\}\ \pi : t\ \{\lambda_.\ outside\,\pi\}$ holds.

Our goal is to prove Always StronglySafe $(init(t))$. By definition (Always), let us consider a configuration (θ, σ) such that we have (Fact 2) $init(t) \xrightarrow{\mathsf{main}}{}^{*} (\theta, \sigma)$. Our goal is now to establish StronglySafe (θ, σ).

We are going to instantiate the core soundness theorem a first time. To that end, we need to exhibit a reduction chain of the oblivious reduction relation $\xrightarrow{\mathsf{oblivious}}$. We apply Lemma 5 to Fact 2. This application provides us with a store σ_0 such that (Fact 3) $init(t) \xrightarrow{\mathsf{oblivious}}{}^{*} (\theta, \sigma_0)$ and (Fact 4) $locs(\theta) \vdash \sigma_0 \xrightarrow{\mathsf{gc}} \sigma$.

We now invoke the Core Soundness Theorem 3 with Facts 1 and 3, and obtain in particular its first consequence (Fact 5): for every thread π of θ, NotStuckOblivious $(\theta, \sigma_0)\ \pi$ holds.

Recall that our goal is StronglySafe (θ, σ) which consists of the conjunction of the three propositions: Safe (θ, σ), PollAndAllocAreOut (θ, σ) and GCCanMakeEveryAllocFit (θ, σ).

First, we prove Safe (θ, σ). In this endeavor, we establish NotStuck $(\theta, \sigma)\ \pi$ for every π in c. This property follows from applying Lemma 6 to Facts 4 and 5.

Second, we prove PollAndAllocAreOut (θ, σ). We make use of the semantics of LambdaFit. Let us suppose a thread t' that is facing an allocation or a polling point. We know from Fact 5 that NotStuckOblivious holds, hence t' can take a step in the oblivious semantics. Recall that this semantics allows the evaluation of allocations and polling points only outside protected sections. Therefore, t' must be outside a protected section.

Third, we prove GCCanMakeEveryAllocFit (θ, σ). This property assumes that (Fact 6) all threads of (θ, σ) are outside and asserts that either every allocation fits, or there exists a step of the GC after which every allocation fits. We distinguish two cases. If EveryAllocFits (θ, σ) holds, then the result holds. Otherwise, we show that after a full garbage collection, every allocation fits: that is, we show that EveryAllocFits $(\theta, collect(R, \sigma))$. By definition (EveryAllocFits), let t' be a thread of (θ, σ) and n a natural number such that IsAlloc $n\ t'$ holds. By definition (AllocFits), we have to prove that $size(collect(locs(\theta), \sigma)) + n \leq S$.

In order to do so, we are going to apply again the Core Soundness Theorem 3, but "a step after" the configuration (θ, σ). First, we make use of Fact 5 to obtain a new threadpool θ' and a new store σ' corresponding to the reduction of t' from the configuration (θ, σ). Thus, we know that (Fact 7) $size(\sigma') = size(\sigma_0) + n$, and that (Fact 8) $init(t) \xrightarrow{\mathsf{oblivious}}{}^{*} (\theta, \sigma_0) \xrightarrow{\mathsf{oblivious}} (\theta', \sigma')$. We now use the Core Soundness Theorem 3 with Facts 1 and 8. Its second consequence is $livespace(locs(\theta'), \sigma') \leq S$. From Lemma 8, we deduce that $size(collect(locs(\theta'), \sigma')) \leq S$. Making use of Fact 7, a simple induction allows us to transform this hypothesis into (Fact 9) $size(collect(locs(\theta), \sigma_0)) + n \leq S$. Indeed, only an allocation of size n was made between σ_0 and σ'. Aside, we know from Lemma 9 that $locs(\theta) \vdash collect(locs(\theta), \sigma_0) \xrightarrow{\mathsf{gc}} collect(locs(\theta), \sigma)$. From Lemma 7, we deduce that (Fact 10) $size(collect(locs(\theta), \sigma)) \leq size(collect(locs(\theta), \sigma_0))$.

Hence, by combining Facts 9 and 10, we have that $size(collect(locs(\theta), \sigma)) + n \leq S$. □

7.5 Deriving Liveness from Safety

We now sketch the proof of the Combined Safety and Liveness Theorem 2 assuming that the Safety Theorem 1 holds.

$$\frac{P\ c}{\mathsf{CrashOrAfterAtMost}\ n\ P\ c}\ \textsc{CrashOrHoldsNow} \qquad \frac{n > 0 \qquad (\forall c'.\ c \xrightarrow{\mathsf{main}} c' \implies \mathsf{CrashOrAfterAtMost}\ (n-1)\ P\ c')}{\mathsf{CrashOrAfterAtMost}\ n\ P\ c}\ \textsc{CrashOrHoldsAfter}$$

$$\frac{\mathsf{CrashOrAfterAtMost}\ n\ P\ c}{\mathsf{CrashOrEventually}\ P\ c}\ \textsc{CrashOrEventually}$$

Figure 26: Predicates for the liveness condition

Let us recall the hypotheses of Theorem 2: let t be a term in administrative normal form such that $\{\Diamond S * \mathit{outside}\ \pi\}\ \pi : t\ \{\lambda_.\ \mathit{outside}\ \pi\}$ holds for every π. By the Safety Theorem 1, we know that Always StronglySafe $(\mathit{init}(t))$ holds, that is, t is strongly safe.

7.5.1 Strong Safety Preservation

Our goal is to deduce the first point of Theorem 2, that is, Always StronglySafe $(\mathit{init}(\mathit{addpp}(t)))$ holds. We enunciate a lemma allowing us to conclude directly. The following lemma asserts that, if t is in administrative normal form, then the strong safety of $\mathit{addpp}(t)$ amounts to the strong safety of t.

Lemma 10 (Strong Safety Preservation)**.** *Suppose that t is in administrative normal form. Always StronglySafe* $(\mathit{init}(t))$ *implies Always StronglySafe* $(\mathit{init}(\mathit{addpp}(t)))$

Proof. The goal is Always StronglySafe $(\mathit{init}(\mathit{addpp}(t)))$. By definition (Always), let us consider a configuration (θ, σ) such that we have $\mathit{init}(\mathit{addpp}(t)) \xrightarrow{\mathsf{main}}{}^{*} (\theta, \sigma)$. Our goal is to prove StronglySafe (θ, σ). The main difficulty lies in characterizing the "shape" of the thread pool θ and of the store σ.

We show that every thread of θ is *almost* the result of a call to *addpp* on some term. Indeed, there are two cases for a thread t' in θ: either there exists a term t_0 such that $t' = \mathit{addpp}(t_0)$, or t' is facing a function call and a polling point was "just" eliminated. In fact, there are two cases: either t' is of the form $\mathsf{let}\ _ = ()\ \mathsf{in}\ (t'_1\ \vec{ts})_{\mathsf{ptr}}$ (because the polling point reduced to the unit value), or t' is of the form $(t'_1\ \vec{ts})_{\mathsf{ptr}}$ (because the substitution of the unit value just occurred). In both cases, thanks to the ANF hypothesis, we know that t'_1 and $\vec{ts}$ consist only of values.

We also show that every code pointer occurring in the store σ results in a call to *addpp* on some other term.

With this precise shape of θ and σ at hand, we reconstruct a thread pool θ_o and a store σ_o such that each thread of θ corresponds to *almost* the result of a call to *addpp* on a thread of θ_0 and the store σ corresponds to a call to *addpp* on every code pointer of σ_o. We then show that $\mathit{init}(t) \xrightarrow{\mathsf{main}}{}^{*} (\theta_o, \sigma_o)$. From the hypothesis that Always StronglySafe $(\mathit{init}(t))$, we deduce that StronglySafe (θ_o, σ_o). It is then a formality to show our goal, that is, StronglySafe (θ, σ). ☐

7.5.2 A Generic Liveness Condition

Our goal is to deduce the second point of Theorem 2, asserting that every allocation will eventually fit, that is, Always (Eventually EveryAllocFits) $(\mathit{init}(\mathit{addpp}(t)))$ holds. Although the theorem fixes a specific polling point insertion strategy, namely *addpp*, we do in fact support other strategies. We present below a sufficient condition that guarantees liveness, supposing that the program is strongly safe.

Let us now explain the idea behind our generic liveness condition. In the definition of the property AfterAtMost $n\ P\ c$ (Figure 24), the rule HoldsAfter requires that the configuration c can take a proper step, guaranteeing that the universal quantification over possible

next steps is not vacuously true. However, recall that our Safety Theorem 1 guarantees in particular that any reachable configuration is globally not stuck (Lemma 3). Hence, if the program is verified in IrisFit, there is no need to manually verify this additional step existence in the premise of HoldsAfter.

We hence propose a new property $\mathsf{CrashOrAfterAtMost}\, n\, P\, c$, presented in Figure 26, which is similar to $\mathsf{AfterAtMost}\ n\ P\ c$, except that $\mathsf{CrashOrAfterAtMost}$ does not rule out globally stuck configurations, that is, configurations in which no thread nor the GC can take a step. The property $\mathsf{CrashOrAfterAtMost}\, n\, P\, c$ ensures that either P will be true after at most n steps of computation, or the program terminates or crashes before reaching n steps of computation. Indeed, the premise of CrashOrHoldsAfter does not require the existence of a possible step. Similarly to $\mathsf{Eventually}\ P\ c$, we introduce the property $\mathsf{CrashOrEventually}\, P\, c$.

We can hence propose a generic liveness condition that "splits the proof burden" between safety and liveness.

Lemma 11 (Liveness Condition)**.** *If $\mathsf{Always}\ \mathsf{GloballyNotStuck}\ (init(t))$ holds, and if $\mathsf{Always}\,(\mathsf{CrashOrEventually}\,\mathsf{EveryAllocFits})\ (init(t))$ holds, then $\mathsf{Always}\,(\mathsf{Eventually}\,\mathsf{EveryAllocFits})\ (init(t))$ holds too.*

Recall our goal to deduce the second point of Theorem 2, that is, we want to prove $\mathsf{Always}\ (\mathsf{Eventually}\,\mathsf{EveryAllocFits})\ (init(t'))$, where t' stands for $addpp(t)$. We make use of Lemma 11. We have to prove $\mathsf{Always}\ \mathsf{GloballyNotStuck}\ (init(t'))$. Thanks to the facts that $\mathsf{StronglySafe}$ implies $\mathsf{GloballyNotStuck}$ (Lemma 3) and that Always is monotonic (Lemma 4), we reduce the goal to proving $\mathsf{Always}\ \mathsf{StronglySafe}\ (init(t'))$. This goal follows from the fact that t is strongly safe and that strong safety is preserved by $addpp$ (Lemma 10).

Hence, the goal is to prove $\mathsf{Always}\ (\mathsf{CrashOrEventually}\,\mathsf{EveryAllocFits})\ (init(addpp(t)))$. This result is stated for reference as the following lemma

Lemma 12 (Satisfaction of the Liveness Condition after Polling Point Insertion)**.** *For every t, $\mathsf{Always}\,(\mathsf{CrashOrEventually}\,\mathsf{EveryAllocFits})\ (init(addpp(t)))$ holds.*

Interestingly, this lemma holds *without precondition.* At a very high level, the proof sketch follows the proof of Lemma 10: we first characterize which kind of configurations can be reached from $init(addpp(t))$ (this time, without an ANF hypothesis). For every reachable configuration, we then exhibit an upper bound after which $\mathsf{EveryAllocFits}$ holds, or the program crashed. The details can be found in our mechanization [Moine, 2024].

Chapter 8

Proof of the Core Soundness Theorem

Beethoven, L. (1795).
Rondo a capriccio "Rage Over a Lost Penny".

The goal of this chapter is to sketch the proof of the Core Soundness Theorem 3, from which we have derived the Safety Theorem 1 and the Combined Safety and Liveness Theorem 2. In this endeavor, we devote our attention to the internals of IrisFit. First, we define the Separation Logic triple in terms of a weakest precondition modality (§8.1). Then, we present auxiliary definitions (§8.2–§8.3) which appear in the definition of the key *state interpretation predicate*, itself giving meaning to assertions (§8.4). We finally sketch the proof of the core soundness theorem (§8.5).

8.1 Definition of the Weakest Precondition Modality

In Iris-based program logics, the triple is usually defined in terms of a *weakest precondition* (WP) modality [Jung et al., 2018b]. In these standard approaches, the WP takes the form $\mathsf{wp}\,\mathcal{E}\,t\,\Psi$, where $\mathcal{E}$ is a *mask* (we explain masks afterwards), t is a term and Ψ a postcondition. The assertion $\mathsf{wp}\,\mathcal{E}\,t\,\Psi$ means that it is safe to execute t and that, if this execution terminates on a value v, then $\Psi\,v$ holds. We follow this standard approach and define our triple in terms of a WP. Our WP takes the form $\mathit{wp}\,\mathcal{E}\,m\,\pi\,t\,\Psi$, where m is the mode (§6.5) and π is a ghost thread identifier (§5.1). The Separation Logic triple $\{\Phi\}\ m/\pi\colon t\ \{\Psi\}_{\mathcal{E}}$ with a mask as an additional formal parameter is then defined in a standard manner as $\Box(\Phi \mathrel{-\!\!*} \mathit{wp}\,\mathcal{E}\,m\,\pi\,t\,\Psi)$. The $\Box$ modality ensures that triples are persistent, which is not the case of the WP modality.

Masks prevent from opening the same invariant twice. Intuitively, a mask $\mathcal{E}$ syntactically records the invariants that the user has not yet opened. In particular, the mask $\top$ allows for opening any of the existing invariants, and $\emptyset$ prevents the opening of any invariant. Additionally, masks appear as parameters of ghost updates for two purposes. First, opening an invariant is a ghost update, and one needs to check if the current mask $\mathcal{E}$ allows it. Second, after such an opening, the name of the invariant must be removed from the current mask. Formally, we write a ghost update as $\Phi\ {}_{\mathcal{E}_1}\!\Rrightarrow_{\mathcal{E}_2} \Phi'$ where $\mathcal{E}_1$ is the mask before the ghost update takes place, and $\mathcal{E}_2$ the mask after the ghost update took place. In fact, this binary ghost update is defined in terms of a lower level *unary ghost update modality*, written ${}_{\mathcal{E}_1}\!\models\!\!\!\Rrightarrow_{\mathcal{E}_2} \Phi'$. The standard Iris library [The Iris Developement Team, 2024] defines $\Phi\ {}_{\mathcal{E}_1}\!\Rrightarrow_{\mathcal{E}_2} \Phi'$ as $\Phi \mathrel{-\!\!*} {}_{\mathcal{E}_1}\!\models\!\!\!\Rrightarrow_{\mathcal{E}_2} \Phi'$. Our treatment of masks is standard. Hence, apart from the formal definition of our WP, we entirely omit masks. We refer the interested reader to our mechanization [Moine, 2024].

Figure 27 first defines the property $\mathit{reducible}\,t\,g\,\sigma$, which asserts that the thread t with status g and store σ can take a step. Figure 27 then presents the formal definition of our WP modality. It makes use of a *state interpretation* predicate, written *interp*, which links the physical state to the ghost state. The overall form of the definition of our WP is standard. At a high level, it consists of a guarded fixpoint and asserts that the considered term is either a value that satisfies the postcondition, or is reducible—that is, can take a step—and for every possible term it reduces to, the WP continues to hold. Moreover, if a thread is forked, its WP holds, too. We next comment in detail the definition of the assertion $\mathit{wp}\,\mathcal{E}\,\pi\,m\,t\,\Psi$ presented in Figure 27.

$$
\begin{array}{ll}
 & \mathit{reducible}\; t\, g\, \sigma \;\triangleq\; \exists t'\, g'\, \sigma'\, t^?.\; t \,/\, g \,/\, \sigma \xrightarrow{\mathsf{step}} t' \,/\, g' \,/\, \sigma' \,/\, t^? \\
(1) & \mathit{wp}\, \mathcal{E}\, m\, \pi\, t\, \Psi \;\triangleq\; \forall N\, \kappa\, \omega\, \sigma\, M\, g. \\
(2) & \quad \ulcorner \kappa(\pi) = (M, \mathit{locs}(t)) \wedge \omega(\pi) = g \urcorner \mathrel{-\!\!*} \mathit{interp}\, N\, m\, \kappa\, \omega\, \sigma \;\; {}_{\mathcal{E}}{\Rrightarrow}_{\emptyset} \\
(3) & \qquad \ulcorner \exists v.\, t = v \urcorner \;*\; {}_{\emptyset}{\mid\!\Rrightarrow}_{\mathcal{E}} (\mathit{interp}\, N\, m\, \kappa\, \omega\, \sigma \;*\; \Psi\, t) \\
(4) & \quad\;\; \vee\; \ulcorner \mathit{reducible}\; t\, g\, \sigma \urcorner \;* \\
(5) & \qquad\; \forall t'\, g'\, \sigma'\, t^?.\; \ulcorner t \,/\, g \,/\, \sigma \xrightarrow{\mathsf{step}} t' \,/\, g' \,/\, \sigma' \,/\, t^? \urcorner \mathrel{-\!\!*} \pounds 1 \mathrel{-\!\!*} \triangleright\, {}_{\emptyset}{\mid\!\Rrightarrow}_{\mathcal{E}} \\
(6) & \qquad\quad \mathit{wp}\, \mathcal{E}\, m\, \pi\, t'\, \Psi \;* \\
(7) & \qquad\quad \mathsf{let}\; \kappa' \triangleq [\pi := (M, \mathit{locs}(t'))]\kappa\; \mathsf{in} \\
(8) & \qquad\quad \mathsf{let}\; \omega' \triangleq [\pi := g']\omega\; \mathsf{in} \\
(9) & \qquad\qquad \ulcorner t^? = \varepsilon \urcorner \;*\; \mathit{interp}\, N\, m\, \kappa'\, \omega'\, \sigma' \\
(10) & \qquad\quad \vee\; \ulcorner t^? \neq \varepsilon \urcorner \;*\; \mathsf{let}\; N' \triangleq N + 1\; \mathsf{in} \\
(11) & \qquad\qquad \mathit{interp}\, N'\, m\, ([N' := (\mathit{locs}(t^?), \emptyset)]\kappa')\, ([N' := \mathsf{Out}]\omega')\, \sigma' \;* \\
(12) & \qquad\qquad \mathit{wp}\, \top\, m\, N'\, t^?\, (\lambda_.\; \mathit{outside}\, N')
\end{array}
$$

Figure 27: Definition of the weakest precondition (WP) modality
In this figure, m is a mode, π a thread identifier, t a term, Ψ a postcondition, N the maximal thread identifier in circulation, κ a roots map, ω a status map, σ a store, M a map of invisible roots, g a status, and $t^?$ a potentially forked thread. The resource $\pounds 1$ is a later credit.

The WP definition universally quantifies several variables (line 1). The variable N represents the maximal thread identifier in circulation. It is used to determine the ghost thread identifier of new threads. The variable κ represents the *roots map*. It associates to each thread identifier the tuple of its *map of invisible roots* (the roots of the evaluation context), and the *set of visible roots*. The map of invisible roots is a map that associates, to each location in the set of invisible roots, the fraction of pointed-by-heap assertion given by the user during an application of Bind. The set of visible roots represents the set of roots in the term under focus, after applications of Bind. The variable ω represents the *status map*, it associates to each thread is status. The variable σ represents the store. The variable M represents the map of invisible roots of π. The variable g represents the status of π.

Next, the WP requires that the variables are coherent (line 2). First, it assumes that κ associates to π the tuple M and $\mathit{locs}(t)$. Second, it also assumes that ω associates to π the status g. Third, the WP requires the state interpretation predicate to hold, with parameters the maximal thread identifier, the mode, the roots map, the status map and the store.

Then, the WP expresses a disjunction over two cases. The first case (line 3) asserts that t is a value, and that the state interpretation predicate holds and that the postcondition holds on the said value. The second case (lines 4–12) is more complex. First, it asserts that the term t with status g and store σ is reducible. Second, the WP quantifies over the possible term t', status g', store σ' and potentially forked thread $t^?$ to which the previous configuration can take a step (line 5). The WP also mentions a later credit $\pounds 1$ [Spies et al., 2022]. This later credit appears in the postcondition of our reasoning rules (§6.2). The definition then asserts, below a later modality, that the WP for t' continues to hold with the same postcondition (line 6), and that two cases are possible. First (line 9), if no thread is forked ($t^? = \varepsilon$), the state interpretation predicate is returned with the roots map and status map properly updated. Second, (line 10) if a thread is forked ($t^? \neq \varepsilon$), the state interpretation predicate is also returned properly updated for both the parent thread, and the child thread, the latter being assigned the thread identifier N'. Moreover, the WP of the child thread must hold too, and return the *outside* N' assertion if it terminates (line 12).

It remains to define the predicate *interp*, which we do in the next sections.

8.2 Auxiliary Definitions

We now define the auxiliary properties we need to state the state interpretation predicate.

8.2.1 General Definitions

We start with the notion of valid and deallocated locations.

Definition 4 (Valid locations)**.** *We say that a location ℓ is* valid *in store σ if ℓ is bound in σ, that is, if ℓ has been allocated, regardless of whether it has been subsequently deallocated.*

Definition 5 (Deallocated locations)**.** *We say that a location ℓ is* deallocated *in a store σ when $\sigma(\ell) = \lightning$.*

Next, we define the notion of a *closed* store with respect to a set of locations R. Intuitively, it means that every location reachable from R are valid. The semantics of LambdaFit ensures that at any point of the evaluation of a program, the current store is closed with respect to the roots of the threads. In the following definition, we write $successors(\sigma, \ell)$ to denote the set of locations found in the block associated to ℓ in σ.

Definition 6 (Closedness)**.** *A store σ is closed with respect to a set of roots R, which we write $\mathsf{closed}\, R\, \sigma$, if:*

- *All the locations in R are valid, that is, $R \subseteq dom(\sigma)$.*
- *Successors are valid, that is, for any valid location ℓ, $successors(\sigma, \ell) \subseteq dom(\sigma)$.*

The oblivious semantics (§7.3) does not deallocate any block. Yet, the user logically deallocates blocks by the mean of the FreeOne rule. Hence, it is useful to introduce a distinction between the physical store σ that exists at runtime with the oblivious semantics and the *logical store* τ that the programmer has in mind when carrying-out proofs [Madiot and Pottier, 2022]. In the physical store, blocks are never deallocated. In the logical store, the user of the logic explicitly deallocates blocks.

Given a set of roots R, the physical store σ and the logical store τ are related: for each location, either their content coincides or the location was deallocated in the logical store. Additionally, if a location in τ points to a non-deallocated block, all its successors are non-deallocated. The roots themselves are non-deallocated in τ.

Definition 7 (Linkedness)**.** *Two stores σ and τ are* linked *with respect to a set of locations R, written $\mathsf{linked}\, R\, \sigma\, \tau$ if:*

- *For any location ℓ that is valid in both σ and τ, either $\sigma(\ell) = \tau(\ell)$ or ℓ is deallocated in τ.*
- *For any location ℓ and block $\vec{w}$, if $\tau(\ell) = \vec{w}$ then no location in $\vec{w}$ is deallocated in τ.*
- *The locations in R are not deallocated in τ.*

To track temporary roots, we use a map written η. This map associates every thread with either a set of locations (the set of temporary roots) if the thread is inside a protected section, and to the special token ε when the thread is outside a protected section.

Definition 8 (Synchronization of the status map and map of temporary roots)**.** *A status map ω is synchronized with the map of temporary roots η, written $\mathsf{syncout}\, \omega\, \eta$ if, every thread identifier π, $\omega(\pi) = \mathsf{In}$ if and only if $\eta(\pi) \neq \varepsilon$.*

8.2.2 Pointed-by-Heap Store

To define pointed-by-heap assertions, we follow Madiot and Pottier [2022] and introduce a *pointed-by-heap store* (which Madiot and Pottier call a predecessor map), written α, from locations to (unsigned) multisets of locations. Intuitively, the pointed-by-heap store α describes an over-approximation of the transposed graph of a logical store τ. Madiot and Pottier define the consistency between a pointed-by-heap store and a logical store as follows.

Definition 9 (Consistence)**.** *A pointed-by-heap store α is* consistent *with a logical store τ written $\mathsf{consistent}\,\tau\,\alpha$, if:*

- *The domain of α corresponds to the non-deallocated locations:* $dom(\tau) \setminus \mathsf{deallocated}(\tau) = dom(\alpha)$, *where* $\mathsf{deallocated}(\tau) = \{\ell \mid \tau(\ell) = \lightning\}$.
- *For any location ℓ, the locations of $\alpha(\ell)$ are valid in τ.*
- *For any two locations ℓ and ℓ' in $dom(\alpha)$, the multiplicity of ℓ' in $successors(\tau, \ell)$ is less than or equal to the multiplicity of ℓ in $\alpha(\ell')$.*

Recall that IrisFit tolerates leftover pointed-by-heap assertions of the form $\ell \hookleftarrow_0 L$, where ℓ is deallocated and L contains only negative elements. The specificity of these null fractions is that they may remain even after a deallocation: null fractions are not gathered at the point of logical deallocation. To cater for these leftover assertions, we introduce a *leftover map* μ, a map of locations to signed multisets where all elements have a nonpositive occurrence. In the end, pointed-by-heap assertions are justified by the union $\alpha \cup \mu$.

Definition 10 (Strong Consistence)**.** *A store τ, a pointed-by-heap store α, and a leftover map μ are* strongly consistent, *written as a ternary predicate $\mathsf{stronglyConsistent}\,\tau\,\alpha\,\mu$, if:*

- *τ and α are consistent: $\mathsf{consistent}\,\tau\,\alpha$*
- $dom(\mu) = dom(\tau)$
- *Locations in μ have a nonpositive multiplicity: for every location ℓ, the multiplicity of each element in $\mu(\ell)$ is nonpositive.*
- *Non-zero elements in the codomain of μ must be themselves deallocated: for any two locations ℓ and ℓ', if ℓ' has a non-zero multiplicity in $\mu(\ell)$, then $\tau(\ell') = \lightning$.*

8.2.3 Pointed-by-Thread Store

To give meaning to pointed-by-thread assertions, we introduce the *pointed-by-thread store*, written ρ, which is a map from locations to sets of thread identifiers. This pointed-by-thread store can be understood as the transposed graph of the roots map κ, minus the elements from the map of temporary roots. Indeed, recall that temporary roots are not tracked by the pointed-by-thread discipline. In the definition below, we abuse notation and write $\kappa(\pi)$ for the union of the domain of the map of invisible roots and the set of visible roots.

Definition 11 (Faithfulness)**.** *A pointed-by-thread store ρ is* faithful *to a roots map κ, a logical store τ and a map of temporary roots η, written $\mathsf{faithful}\,\kappa\,\tau\,\eta\,\rho$ when:*

- $dom(\rho) \subseteq dom(\tau)$
- *if a location ℓ is deallocated in τ, then it is not in the domain of ρ*
- *for every thread π and location ℓ such that $\ell \in (\kappa(\pi) \setminus \eta(\pi))$ we have that $\pi \in \rho(\ell)$*

8.3 Resource Algebras

In Iris, the content of ghost cells should belong to a *camera*, which corresponds, roughly speaking, to a "step-indexed resource algebra". The concept of resource algebras is itself a generalization of Partial Commutative Monoids (PCMs) [Jung et al., 2018b]. As step-indexing is not relevant to the definition of our assertions, we simplify the intuition of the following resource algebras. Each resource algebra comes with a binary *composition law* written $x \cdot y$ for x and y members of the resource algebra. Moreover, resource algebras come with a notion of *validity*, a unary predicate that distributes over the composition law (that is, if $x \cdot y$ is valid, both x and y are valid). Crucially, ghost cells contain only valid elements.

Definition 12 (Signed multisets with generalized fraction)**.** *The structure "Signed Multisets with Generalized Fraction (SMGF) over a countable set $\mathcal{L}$" is the resource algebras whose elements are of $\mathbb{Q} \times \mathsf{SMultiset}(\mathcal{L})$, where for every element (q, X), if $q = 0$, then X contains only elements with a nonpositive multiplicity. The composition law is defined as $(q_1, X_1) \cdot (p_z, X_2) \triangleq (q_1 + q_2, X_1 \uplus X_2)$. Valid elements are those with a fraction in the interval* $[0; 1]$.

We next briefly review the standard resource algebra we use to set up the ghost state. They are all mentioned by Jung et al. [2018b], except for the set resource algebra, the rational numbers resource algebra, and the option resource algebra, which ship with the Iris mechanization [The Iris Developement Team, 2024].

Authoritative The resource algebra $\mathit{Auth}(A)$ describes the *authoritative resource algebra* over the resource algebra A. This resource algebra gives access to $\bullet x$, the *authoritative* ownership of x, and $\circ y$, the *fragmentary* ownership of y. Together, these two resources entail that $y \preccurlyeq x$, which means that there exists an element z of the algebra such that $x = y \cdot z$. We omit the composition law and validity definition.

Agreement The resource algebra $\mathit{Ag}(A)$ allows multiple parties to agree on a particular member of A. Members of this resource algebras are either $\mathsf{ag}(x)$ for some $x \in A$ or a bottom value $\lightning$. The composition law sends every composition to $\lightning$, except for the composition $\mathsf{ag}(x) \cdot \mathsf{ag}(x)$ which is sent to $\mathsf{ag}(x)$. The only valid element is $\mathsf{ag}(x)$.

Sum The resource algebra $A_1 +_{\lightning} A_2$ describes the sum of the resource algebras A_1 and A_2. Members of this resource algebra can either be $\mathsf{inl}(x_1)$ for some $x_1 \in A_1$, $\mathsf{inr}(x_2)$ for some $x_2 \in A_2$, or a bottom value $\lightning$. The composition law sends two members of the same algebra to their underlying composition, and every other combination to $\lightning$. The element $\mathsf{inl}(x_1)$ is valid if x_1 is valid, and similarly $\mathsf{inr}(x_2)$ is valid if x_2 is valid. The bottom value $\lightning$ is not valid.

Product The resource algebra $A_1 \times A_2$ describes the product between the resource algebra A_1 and A_2. The composition law composes independently each member of the pair.

Rational numbers Rational numbers $\mathbb{Q}$ define a resource algebra with the composition law being the addition. Making use of the validity predicate, we then restrict the set X of possible fractions. We abuse notations and write $\mathbb{Q} \cap X$ to denote the resource algebra whose elements are in $\mathbb{Q}$ and validity is restricted to X. For example, standard Separation Logic fractions corresponds to $\mathbb{Q} \cap (0; 1]$.

Fractional Algebra We define the resource algebra $\mathit{Frac}(A)$ as $((\mathbb{Q} \cap (0; 1]) \times A)$. The resource algebra $\mathit{Frac}(A)$ intuitively corresponds to the "fractional" version of A, where each element comes with a particular fraction.

Set The resource algebra $\mathit{SetMono}(A)$ describes the sets of elements of X, with the union as the composition law. Every element of this resource algebra is valid. This algebra is monotonic in the sense that, because the composition law is the union and not the disjoint union, once an element is added to a ghost cell with the resource algebra $\mathit{SetMono}(A)$, it cannot be removed.

Option The resource algebra $\mathit{Option}(A)$ describes the resource algebra A with an additional unit value ε. The composition law of $\mathit{Option}(A)$ is the composition law of A with ε as an additional unit. The unit value ε is valid, and the validity of other elements amounts to their validity in A.

Finite map The resource algebra $X \rightarrow_{\text{fin}} A$ describes the resource algebra of finite maps from the type X to the resource algebra A. The composition law is the union, composing the elements of two pre-existing keys. A finite map is valid if and only if all its values are valid.

8.4 State Interpretation and Definition of Assertions

We now introduce the ghost cells used to define our assertions. Each ghost cell is equipped with a resource algebra. We realize thread identifiers with natural numbers $\mathbb{N}$.

Definition 13 (Ghost state). *We introduce four ghost cells γ_d, γ_h, γ_t and γ_s equipped with the following resource algebras.*

Ghost cell	*Used to define*	*Associated resource algebra*
γ_d	*Space Credits*	$Auth(\mathbb{Q} \cap [0, \infty))$
γ_h	*Pointed-by-heap*	$Auth(\mathcal{L} \rightarrow_{fin} \mathsf{SMGF}(\mathcal{L}))$
γ_t	*Pointed-by-thread, deallocation witness*	$Auth(\mathcal{L} \rightarrow_{fin} Frac(SetMono(\mathbb{N})) +_{\lightning} Ag(\mathbb{1}))$
γ_s	*Inside and outside assertions*	$Auth(\mathbb{N} \rightarrow_{fin} Option(SetMono(\mathcal{L})))$

Our resource algebras are all *authoritative*: the state interpretation holds the authoritative resource, while various assertions are defined as fragmentary resources. Space credits are non-negative rational numbers. Pointed-by-heap assertions are represented with a map from locations to signed multisets with generalized fractions of locations (Definition 12). Pointed-by-thread assertions and the deallocation witness are represented using the same ghost cell: indeed the two are mutually exclusive, a location is either a root for a (possibly empty) set of threads, or logically deallocated.[1] Finally, "*outside*" and "*inside*" assertions are defined using a map from thread identifiers to an optional set of locations. The case ε is used to assert that the thread is outside a protected section, the other case being used to track temporary roots.

Figure 28 unveils the definitions occurring in our state interpretation predicate. Figure 29 presents the definition of the assertions of IrisFit. These definitions are mostly standard, except for the later line of the definition of *interp*, which is related to the trimming rules. We comment on this novelty below. In these definitions, given a fraction p and a map m from X to A, we write $p.m$ for the map $\{(k, (p, v)) \mid (k, v) \in m\}$. Given a set s, we write $[s := \mathsf{inr}(\mathsf{ag}(\mathbb{1}))]$ the map whose keys are in s and whose values are all equal to the agreement over the unit $\mathbb{1}$. We next comment on how the definitions of Figure 28 give meanings to the assertions defined in Figure 29.

Points-to assertions The assertion *store* σ gives meaning to the points-to assertion. To realize points-to assertions, Iris defines a certain piece of ghost state, defines an assertion *Heap* σ that ties a store σ to this ghost state, and defines the base points-to assertion *pointsto* $\ell\, p\, blk$ in terms of this ghost state [Jung et al., 2018b, §6.3.2]. This machinery is implemented within the Iris `gen_heap` library [The Iris Developement Team, 2024], which we build on. Moreover, the `gen_heap` library allows for associating persistent information to locations via a mechanism of "*meta*" assertions. In our case, we associate to each location ℓ the number of fields of its associated block. Indeed, Figure 29 defines the assertion *sizeof* $\ell\, n$ as *meta* $\ell\, n$. (We omit the *namespace* parameter of the "*meta*" assertions allowing the user to associate multiple persistent data to a location.) The definition of our points-to assertion $\ell \mapsto_p \vec{v}$ is *pointsto* $\ell\, p\, \vec{v} * $ *sizeof* $\ell\, (size(\vec{v}))$. This definition allows the user to extract a *sizeof* assertion from a points-to assertion without requiring access to the state interpretation predicate, via SizeOfPointsTo.

[1] We could have used another ghost cell for deallocation witnesses.

$$
\begin{aligned}
\mathit{store}\,\sigma &\triangleq \mathit{Heap}\,\sigma \;*\; \mathop{\Large\ast}_{(\ell,\vec{w})\in\sigma} \mathit{meta}\,\ell\,(\mathit{size}(\vec{w}))\\
\mathit{pbh}\,\tau &\triangleq \exists \alpha\,\mu.\,\ulcorner \mathsf{stronglyConsistent}\,\tau\,\alpha\,\mu \urcorner \;*\; \boxed{\bullet\,(1.\alpha \cup 0.\mu)}^{\gamma_h}\\
\mathit{pbt}\,\kappa\,\tau\,\eta &\triangleq \exists \rho\,\xi.\,\ulcorner \mathsf{faithful}\,\kappa\,\tau\,\eta\,\rho \;\wedge\; \xi \subseteq \mathit{dom}(\tau) \urcorner \;*\; \boxed{\bullet\,(\mathsf{inl}.1.\rho \cup [\xi := \mathsf{inr}(\mathsf{ag}(\mathbb{1}))])}^{\gamma_t}\\
\mathit{protected}\,\eta &\triangleq \boxed{\bullet\,\eta}^{\gamma_s}\\
\mathit{spacecredits}\,\tau &\triangleq \boxed{\bullet\,(S - \mathit{size}(\tau))}^{\gamma_d}
\end{aligned}
$$

$\mathit{interp}\,N\,m\,\kappa\,\omega\,\sigma \triangleq \exists \tau\ \eta.$

$\ulcorner \mathit{dom}(\kappa) = \mathit{dom}(\omega) = \mathit{dom}(\eta) = \{0..N\} \urcorner$	$*$	// agreement on thread ids
$\ulcorner \mathit{dom}(\sigma) = \mathit{dom}(\tau) \urcorner$	$*$	// agreement on heap locations
$\ulcorner \mathsf{closed}\,(\mathit{roots}(\kappa))\,\sigma \urcorner$	$*$	// physical heap is closed
$\ulcorner \mathsf{linked}\,(\mathit{logicalroots}(\kappa,\eta))\,\sigma\,\tau \urcorner$	$*$	// physical and logical heaps are linked
$\ulcorner \mathit{size}(\tau) \le S \urcorner$	$*$	// size of the logical heap is bounded
$\ulcorner \mathsf{syncout}\,\omega\,\eta \urcorner$	$*$	// statuses are synchronized
$\mathit{store}\,\sigma \;*\; \mathit{pbh}\,\tau \;*\; \mathit{pbt}\,\kappa\,\tau\,\eta$	$*$	// ghost resources
$\mathit{protected}\,\eta \;*\; \mathit{spacecredits}\,\tau$	$*$	
if $m = ✂$		
then $\mathop{\Large\ast}_{(\pi,(M,_))\in\kappa} M \Lleftarrow \{\pi\}$		// invisible roots
else $\ulcorner \mathrm{True} \urcorner$		

Figure 28: Definition of the state interpretation predicate

$$
\begin{aligned}
\ell \mapsto_p \vec{v} &\triangleq \mathit{pointsto}\,\ell\,p\,\vec{v} \;*\; \mathit{sizeof}\,\ell\,(\mathit{size}(\vec{v}))\\
\ell \hookleftarrow_q L &\triangleq \boxed{\circ\,[\ell := (q, L)]}^{\gamma_h}\\
\ell \Lleftarrow_p \Pi &\triangleq \exists \Pi'.\,\ulcorner \Pi' \subseteq \Pi \urcorner \;*\; \boxed{\circ\,[\ell := \mathsf{inl}(p, \Pi')]}^{\gamma_t}\\
M \Lleftarrow \Pi &\triangleq \mathop{\Large\ast}_{(\ell,p)\in M} \ell \Lleftarrow_p \Pi\\
\dagger\,\ell &\triangleq \boxed{\circ\,[\ell := \mathsf{inr}(\mathsf{ag}(\mathbb{1}))]}^{\gamma_t}
\end{aligned}
$$

$$
\begin{aligned}
\Diamond c &\triangleq \boxed{\circ\,c}^{\gamma_d}\\
\mathit{sizeof}\,\ell\,n &\triangleq \mathit{meta}\,\ell\,n\\
\mathit{outside}\,\pi &\triangleq \boxed{\circ\,[\pi := \varepsilon]}^{\gamma_s}\\
\mathit{inside}\,\pi\,\Pi &\triangleq \boxed{\circ\,[\pi := \Pi]}^{\gamma_s}
\end{aligned}
$$

Figure 29: Definition of assertions

Pointed-by-heap assertions The assertion $\mathit{pbh}\,\tau$ gives meaning to the pointed-by-heap assertion. It quantifies over a pointed-by-heap store α and a leftover store μ, which must be strongly consistent with the logical store τ. The authoritative assertion $\boxed{\bullet\,(1.\alpha \cup 0.\mu)}^{\gamma_h}$ is linked to the definition of pointed-by assertions. The map $1.\alpha \cup 0.\mu$ is the union of the the map α where every value is given the fraction 1 and the and the map μ where every value is given the fraction 0. Recall from §8.2.2 that the pointed-by-heap store α contains the predecessors of allocated predecessors. The leftover map μ has only nonpositive signed multisets, with fraction 0, and records every leftover assertion. We define the pointed-by-heap assertion by $\ell \leftarrow_q L \triangleq \boxed{\circ\,[\ell := (q, L)]}^{\gamma_h}$. In particular, if $q = 1$, then one can deduce that L is an over-approximation of the predecessors of ℓ.

Pointed-by-thread and deallocation witness assertions The assertion $\mathit{pbt}\,\kappa\,\tau\,\eta$ gives meaning to the pointed-by-thread assertion, as well as the deallocation witness. It existentially quantifies over a pointed-by-thread store ρ and a leftover set of location ξ, and asserts that the former is faithful to the latter with the map of temporary roots η (Definition 11). It also requires the leftover set to be included in the domain of the logical store. The authoritative assertion $\boxed{\bullet\,(\mathsf{inl}.1.\rho \cup [\xi := \mathsf{inr}(\mathsf{ag}(\mathbb{1}))])}^{\gamma_t}$ is linked to the definition of pointed-by-thread assertions as well as the deallocation witnesses. The map $\mathsf{inl}.1.\rho \cup [\xi := \mathsf{inr}(\mathsf{ag}(\mathbb{1}))]$ is the union of the map ρ where every value is given the fraction 1 and put in the left part of a sum, and of the map whose keys are the elements of ξ and value $\mathsf{ag}(\mathbb{1})$ put in the right part of a sum. Figure 29 defines the pointed-by-thread assertion $\ell \Leftarrow_p \Pi$ as $\exists \Pi'.\ \ulcorner \Pi' \subseteq \Pi \urcorner * \boxed{\circ\,[\ell := \mathsf{inl}(p, \Pi')]}^{\gamma_t}$, that is, the assertion $\ell \Leftarrow_p \Pi$ asserts the ownership of some thread-predecessors of ℓ, included in Π. This definition allows us to easily derive CovPBThread. We then define the deallocation witness $\dagger\ell$ as $\boxed{\circ\,[\ell := \mathsf{inr}(\mathsf{ag}(\mathbb{1}))]}^{\gamma_t}$. These definitions allow for easily deriving DeadPBThread, since $\ell \Leftarrow_p \Pi$ excludes $\dagger\,\ell$ as the former is a member of the left part of a sum and the latter a member of the right part of a sum.

Inside and outside assertions The assertion $\mathit{protected}\,\eta$ gives meaning to the assertions $\mathit{inside}\,\pi\,T$ and $\mathit{outside}\,\pi$ assertions from the map of temporary roots η. Indeed, the assertion $\mathit{protected}\,\eta$ is defined as the authoritative ownership $\boxed{\bullet\,\eta}^{\gamma_s}$. Then, Figure 29 defines both assertions $\mathit{inside}\,\pi\,T$ and $\mathit{outside}\,\pi$ as fragmentary ownership of η.

Space credits The assertion $\mathit{spacecredits}\,\tau$ gives meaning to space credits. Indeed, we define the assertion $\mathit{spacecredits}\,\tau$ as $\boxed{\bullet\,(S - \mathit{size}(\tau))}^{\gamma_d}$, that is, the authoritative ownership of the number of space credits in circulation: the maximal heap size S minus the size of the current logical store. We then define space credits as fragmentary ownership of this total amount and pose $\lozenge c \triangleq \boxed{\circ\,c}^{\gamma_d}$.

On roots Recall that a roots map κ is a map from thread identifiers to a pair of a map M of locations to fractions (the invisible roots of the thread) and a set S of locations (the visible roots of the thread). Given a map of temporary roots η, that is, a map from a thread identifier to a set of locations T, we define the set of all roots of all threads minus their temporary roots as

$$\mathit{logicalroots}(\kappa, \eta) \triangleq \bigcup_{(\pi,(M,S)) \in \kappa,\ (\pi,T) \in \eta} ((\mathit{dom}(M) \cup S) \setminus T)$$

We then define the set of all roots as $\mathit{roots}(\kappa) \triangleq \mathit{logicalroots}(\kappa, \emptyset)$.

Definition of the state interpretation Figure 28 finally defines the state interpretation predicate itself $\mathit{interp}\,N\,m\,\kappa\,\omega\,\sigma$, where N is the maximal thread identifier in circulation, m is the current mode, κ is the roots map, ω the status map and σ the physical store. The

definition quantifies existentially over the logical store τ and the map of temporary roots η. The state interpretation first asserts that the domains of the roots map κ, the status map ω and the map of temporary roots η are all equal to the set of thread identifiers from 0 to N. The predicate then asserts that the physical store σ and logical store τ have the same domain. The state interpretation records that the physical store is *closed* (Definition 6) with respect to all the roots of κ. It also records that the physical store is linked with the logical store, with the set of roots $\mathit{logicalroots}(\kappa, \eta)$ (Definition 7). Then, the state interpretation records that the logical store τ has a size less than or equal to the maximum size S. It also records that the status map ω is synchronized with the map of temporary roots η (Definition 8). The definition then records the ghost state associated to the points-to, the pointed-by-heap, and the pointed-by-thread assertions, as well as the "*inside*" and "*outside*" assertions and space credits.

Innovative Aspect of the State Interpretation Predicate The last three lines of the definition of *interp* are innovative. If the mode m is ✂ then the state interpretation holds the fractions of pointed-by-thread assertions associated to the invisible map from the roots map κ. These fractions of pointed-by-thread assertions are exactly the ones given by the user while applying Bind.

The fact that the interpretation predicate holds a fraction of pointed-by-thread assertions of invisible roots allows for verifying "trimming" rules (that is, TrimPBThread, Fork and TrimInside, as explained in §6.4). Let us focus on TrimPBThread. Recall that this reasoning rule allows for updating an assertion $\ell \Leftarrow\!\!\!\!\!\!\Leftarrow_p \{\pi\}$ into $\ell \Leftarrow\!\!\!\!\!\!\Leftarrow_p \emptyset$ if ℓ is not a root of the term under focus in π. The proof of TrimPBThread looks as follows. Either ℓ is not a root of the evaluation context of π, and the trimming can be done without further work. Otherwise, ℓ is a root of the evaluation context of π. In this case, there is a corresponding assertion $\ell \Leftarrow\!\!\!\!\!\!\Leftarrow_{p'} \{\pi\}$ stored inside the state interpretation predicate! We hence use FracPBThread to update $\ell \Leftarrow\!\!\!\!\!\!\Leftarrow_p \{\pi\} * \ell \Leftarrow\!\!\!\!\!\!\Leftarrow_{p'} \{\pi\}$ into $\ell \Leftarrow\!\!\!\!\!\!\Leftarrow_p \emptyset * \ell \Leftarrow\!\!\!\!\!\!\Leftarrow_{p'} \{\pi\}$.

This approach also motivates the mode ✖: if the user did not provide pointed-by-thread assertions while applying Bind, our proof for TrimPBThread does not stand anymore!

Definition of the parameterized ghost update In IrisFit, several reasoning rules are expressed as a *parameterized ghost update* $\Phi \ {}^{\pi}\!\!\Rrightarrow^{V} \Phi'$, applying only on a thread π with visible roots V. As explained in §6.5, this ghost update is also parameterized by a mode m. The parameterized ghost update $\Phi \ {}^{\pi}\!\!\Rrightarrow^{V}_{m} \Phi'$ is defined as a primitive ghost update $\Rrightarrow$ allowing temporary access to the state interpretation predicate.

$$\Phi \ {}^{\pi}\!\!\Rrightarrow^{V}_{m} \Phi' \ \triangleq\ \forall N\, \kappa\, \omega\, \sigma.\ \ulcorner \mathsf{snd}(\kappa(\pi)) = V \urcorner \ *\ \Phi \ *\ \mathit{interp}\, N\, m\, \kappa\, \omega\, \sigma \Rrightarrow \Phi' \ *\ \mathit{interp}\, N\, m\, \kappa\, \omega\, \sigma$$

8.5 Proving the Core Soundness Theorem

In order to prove the Core Soundness Theorem 3, we imitate the soundness (or *adequacy*) proof of the standard Iris WP [Jung et al., 2018b, §6.4], as our WP is structurally similar. We sketch the proof below.

Recall that a thread pool θ is a list of pairs of a thread and a status. In the following, we use the index in the list as a thread identifier. We define the *concrete roots map*, written $\mathit{rootsmap}(\theta)$ as the map associating to each thread t in θ the tuple of the empty map and the set $\mathit{locs}(t)$. We define the *concrete status map*, written $\mathit{statusmap}(\theta)$ as the map associating each thread with its status. We finally define the *concrete state interpretation* of a thread pool θ and a store σ written $\mathit{interp}_0\,(\theta, \sigma)$, as follows:

$$\mathit{interp}_0\,(\theta, \sigma) \ \triangleq\ \mathit{interp}\,(|\theta|)\,(✂)\,(\mathit{rootsmap}(\theta))\,(\mathit{statusmap}(\theta))\,\sigma$$

Preserve
$$\ulcorner(\theta_1,\sigma_1) \xrightarrow{\mathsf{oblivious}}{}^{n} (\theta_2,\sigma_2)\urcorner \ * \ £n \quad interp_0\,(\theta_2,\sigma_2) \ * \ wps\,\theta_1 \quad {}_{\top,\emptyset}{\Rrightarrow}^{n}_{\emptyset,\top} \quad interp_0\,(\theta_2,\sigma_2) \ * \ wps\,\theta_2$$

Progress
$$\ulcorner\theta(t,g)=\pi\urcorner \ * \ interp_0\,(\theta,\sigma) \ * \ wps\,\theta \quad {}_{\emptyset}{\Rrightarrow}_{\top} \quad \ulcorner\mathsf{NotStuckOblivious}\,(\theta,\sigma)\,\pi\urcorner$$

LiveSpace
$$\ulcorner\mathsf{AllOutside}\,(\theta,\sigma)\urcorner \ * \ interp_0\,(\theta,\sigma) \quad -\!\!* \quad \ulcorner livespace(locs(\theta),\sigma) \leq S\urcorner$$

InterpInit
$$\ulcorner locs(t)=\emptyset\urcorner \quad \Rrightarrow \quad \exists\gamma_d\,\gamma_h\,\gamma_t\,\gamma_s.\ interp_0\,([(t,\mathsf{Out})],\emptyset) \ * \ \Diamond S \ * \ outside\,\pi_0$$

Figure 30: Lemmas for the proof of the Core Soundness Theorem

Let θ be a thread pool, we write $wps\,\theta$ for the iterated conjunction of the WP of each thread in θ. Precisely:

$$wps\,\theta \quad \triangleq \quad \mathop{\Huge *}_{0\leq\pi<|\theta|\ \wedge\ \theta(\pi)=(t,_)} wp\ \top\ \pi\ ✂\ t\,(\lambda_.\ outside\,\pi)$$

We now present the key lemmas used in our proof of the core soundness theorem. These lemmas are gathered in Figure 30.

Preserve intuitively asserts that reduction steps preserve the state interpretation and the WP. The premise of the rule requires that $c_1 \xrightarrow{\mathsf{oblivious}}{}^{n} c_2$, that is, there is a reduction of length n between the configuration c_1 and c_2. The rule also consumes $£n$, the concrete interpretation predicate and the WPs. The assertion $\Phi\,{}_{\top,\emptyset}{\Rrightarrow}^{n}_{\emptyset,\top}\,\Phi'$ is an abbreviation for the mouthful assertion $\Phi \quad {}_{\top}{\Rrightarrow}_{\emptyset}\ {}_{\emptyset}{\Rrightarrow}^{n}_{\emptyset}\ {}_{\emptyset}{\Rrightarrow}_{\top} \quad \Phi'$, where ${}_{\emptyset}{\Rrightarrow}^{n}_{\emptyset}$ represents n ghost updates ${}_{\emptyset}{\Rrightarrow}_{\emptyset}$ separated by one later modality $\triangleright$. In short, ${}_{\top,\emptyset}{\Rrightarrow}^{n}_{\emptyset,\top}$ allows first for a ghost update removing the possibility of opening invariants, second for n ghost updates without invariants, and third for a last ghost update restoring the ability to open invariants. After all these ghost updates, Preserve asserts that the concrete state interpretation of (θ,σ) holds, and the WPs of the threads of θ_2 hold too.

Progress asserts that, for a configuration (θ,σ), if the concrete state interpretation holds as well as the WPs, then no thread of the configuration is stuck. This is a direct consequence of the definition of the WP.

LiveSpace is key to our endeavor: if all threads are outside protected sections, then the state interpretation predicate asserts that the live heap space is bounded by the maximal heap size.

InterpInit asserts that we can construct the adequate state interpretation predicate for the initial configuration of a single thread outside a protected section, with an empty store, under the assumption that this thread contains no location. This lemma produces the adequate state interpretation, the initial amount of space credits $\Diamond S$ and the outside assertion for the initial thread of identifier $\pi_0 = 0$. In the statement of this lemma, we make explicit the existential quantification of the ghost cells parameterizing all the logic and described in §8.4. Indeed, its InterpInit that "allocates" these names, which are implicit parameters elsewhere.

The last piece of the puzzle is the soundness theorem of Iris with later credits [Spies et al., 2022]. We present here the theorem as it appears in Iris' mechanization [The Iris Developement Team, 2024].

Theorem 4 (Adequacy of Iris with later credits)**.** *Let ϕ be a pure proposition. If $£n \quad {}_{\top}{\Rrightarrow}_{\emptyset}\ {}_{\emptyset}{\Rrightarrow}^{n}_{\emptyset}\ \ulcorner\phi\urcorner$ holds in Iris with later credits, then ϕ holds.*

We now sketch the proof of the Core Soundness Theorem 3, assembling building blocks. We restate the theorem below, with a more precise statement, revealing ghost cells and the need for a *closed term*.

Theorem 5 (Core Soundness, precise). *Let t be a closed term. Assume that the following assertion holds*

$$\forall \gamma_d\, \gamma_h\, \gamma_t\, \gamma_s\, \pi.\ \{\Diamond S * \mathit{outside}\,\pi\}\ \pi \colon t\ \{\lambda_.\ \mathit{outside}\,\pi\}$$

Then, for every configuration c such that $\mathit{init}(t) \xrightarrow{\mathsf{oblivious}}{}^{*} c$,

1. *for every identifier π of a thread in c, the property $\mathsf{NotStuckOblivious}\, c\, \pi$ holds;*
2. *$\mathsf{AllOutside}\, c$ implies $\mathit{livespace}(c) \leq S$.*

Proof. Let $c = (\theta, \sigma)$ such that $([(t, \mathsf{Out})], \emptyset) \xrightarrow{\mathsf{oblivious}}{}^{*} (\theta, \sigma)$. The reduction chain hypothesis $([(t, \mathsf{Out})], \emptyset) \xrightarrow{\mathsf{oblivious}}{}^{*} (\theta, \sigma)$ guarantees that there exists a natural number n such that $([(t, \mathsf{Out})], \emptyset) \xrightarrow{\mathsf{oblivious}}{}^{n} (\theta, \sigma)$.

Let us select the goal (1). Let π be a valid index in θ. We apply Theorem 4, instantiating ϕ with $\mathsf{NotStuckOblivious}\,(\theta, \sigma)\,\pi$. We now switch to an Iris proof and face the following goal:

$$\begin{array}{c} \pounds n\ * \\ (\forall \gamma_d\, \gamma_h\, \gamma_t\, \gamma_s\, \pi.\ \{\Diamond S * \mathit{outside}\,\pi\}\ \pi \colon t\ \{\lambda_.\ \mathit{outside}\,\pi\}) \end{array} {}_{\top}\!\!\Rrightarrow_{\emptyset}\ {}_{\emptyset}\!\!\Rrightarrow^{n}_{\emptyset} \ulcorner \mathsf{NotStuckOblivious}\,(\theta, \sigma)\,\pi \urcorner$$

We use InterpInit to allocate the ghost cells names, the concrete state interpretation, the initial space credits, and the "*outside*" assertion for the initial thread identifier $\pi_0 = 0$. We specialize the universal quantification in front of the triple with our fresh ghost cells names and the initial thread identifier. We now face the goal:

$$\begin{array}{c} \pounds n\ *\ \mathit{interp}_0\,([(t, \mathsf{Out})], \emptyset)\ * \\ \Diamond S\ *\ \mathit{outside}\,\pi_0\ * \\ \{\Diamond S\ *\ \mathit{outside}\,\pi_0\}\ \pi \colon t\ \{\lambda_.\ \mathit{outside}\,\pi_0\} \end{array} \quad {}_{\top}\!\!\Rrightarrow_{\emptyset}\ {}_{\emptyset}\!\!\Rrightarrow^{n}_{\emptyset}\ \ulcorner \mathsf{NotStuckOblivious}\,(\theta, \sigma)\,\pi \urcorner$$

By definition, we have that $\{\Diamond S * \mathit{outside}\,\pi_0\}\ \pi \colon t\ \{\lambda_.\ \mathit{outside}\,\pi_0\}$ unfolds to the assertion $\Box(\Diamond S * \mathit{outside}\,\pi_0 \twoheadrightarrow \mathit{wp}\,\pi ✂ t\,(\lambda_.\ \mathit{outside}\,\pi_0))$. We remove the persistence modality and make use of the initial space credits and $\mathit{outside}\,\pi_0$ assertions to satisfy the precondition of the wand, and now face the goal:

$$\begin{array}{c} \pounds n\ * \\ \mathit{interp}_0\,([(t, \mathsf{Out})], \emptyset)\ *\ \mathit{wp}\,\pi ✂ t\,(\lambda_.\ \mathit{outside}\,\pi) \end{array} \quad {}_{\top}\!\!\Rrightarrow_{\emptyset}\ {}_{\emptyset}\!\!\Rrightarrow^{n}_{\emptyset}\ \ulcorner \mathsf{NotStuckOblivious}\,(\theta, \sigma)\,\pi \urcorner$$

Knowing that $([(t, \mathsf{Out})], \emptyset) \xrightarrow{\mathsf{oblivious}}{}^{n} (\theta, \sigma)$, the premise of the above entailment is exactly the one of Preserve, with thread pool $[(t, \mathsf{Out})]$ and store $\emptyset$. We make use of Preserve as well as the various rules of connectives, and we are left to prove that:

$$\mathit{interp}_0\,(\theta, \sigma) * \mathit{wps}\,\theta\ \ {}_{\emptyset}\!\!\Rrightarrow_{\top} \ulcorner \mathsf{NotStuckOblivious}\,(\theta, \sigma)\,\pi \urcorner$$

We conclude with Progress.

The proof of the goal (2) is entirely similar, except that we instantiate ϕ with the space inequality $\mathit{livespace}(\mathit{locs}(\theta), \sigma) \leq S$ and that, at the end, we use LiveSpace to conclude on the live heap space. ∎

CHAPTER 9

CLOSURES

Poulenc, F. (1962).
Clarinet Sonata.

As explained earlier (§2.6), LambdaFit does not have primitive closures. Instead, we define *closure construction* $\mu_{\mathsf{clo}} f.\, \lambda \vec{x}.\, t$ and *closure invocation* $(\ell\ \vec{u})_{\mathsf{clo}}$ as macros, which expand to sequences of primitive LambdaFit instructions. These macros implement *flat closures* [Appel, 1992, Chapter 10]. That is, a closure is represented as a record whose fields store a code pointer (at offset 0) and a series of values (at offset 1 and beyond). The implementation of these macros (§9.2) is the same as in our earlier paper [Moine et al., 2023]. Our reasoning rules for closure construction, invocation, and deallocation are improved versions of the rules presented in our earlier paper [Moine et al., 2023]. The main improvement is that the assertions that describe closures are now *persistent*. From an end user's point of view, this makes closures much easier to work with. Internally, this is made possible by using *liveness-based cancellable invariants* (§5.9).

Our reasoning rules for closures are abstract and do not reveal *how* closures are implemented. They reveal only how much space a closure occupies and which pointers it keeps live. A user can apply these rules without knowing how closures are internally represented.

Our construction of the reasoning rules for closures is in two layers. First, we introduce a low-level assertion $\mathit{Closure}\, E\, f\, \vec{x}\, t\, \ell$, which asserts that, at location ℓ in the heap, one finds a closure that behaves like the function $\mu f.\lambda \vec{x}.\, t$ under the environment E. Crucially, in this assertion, the term $\mu f.\lambda \vec{x}.\, t$ *can* have free variables, whose values are given by E. This assertion does not reveal how a closure is represented in memory, but does reveal its code. We give an overview of this low-level API (§9.3), then describe some details of its implementation (§9.4). Second, we define a high-level assertion $\mathit{Spec}\, n\, E\, P\, \ell$, which describes the behavior of a closure in a more abstract way. It asserts that, at location ℓ, one finds a closure that corresponds to a n-ary function, whose behavior is described by the predicate P, and whose environment is E. The exact type and meaning of P are explained later on; roughly speaking, it is a Hoare triple. Although the environment E does not participate in the description of the behavior of the closure, it remains needed in order to reason about the pointers that it contains and about the size of the closure block. We give an overview of this high-level API (§9.5), then describe its implementation (§9.6). Only the high-level layer is exposed to the end user; the low-level layer remains internal.

9.1 Environments

We write $\mathit{fvclo}(f, \vec{x}, t)$ for a list of the free variables of the function $\mu f.\lambda \vec{x}.\, t$, that is, for a list of the variables in the set $\mathit{fv}(t) \setminus \{f, \vec{x}\}$. The order in which the variables occur in this list is not relevant, but is fixed: this is reflected in the fact that fvclo is a function of f, $\vec{x}$, and t.

An environment E is a list of pairs (v, q) of a value v and a nonzero fraction q. This fraction is used in a pointed-by-heap assertion, as follows: we write $E \hookleftarrow L$ for the conjunction $\mathop{\Large\ast}_{(v,q)\in E}\, v \hookleftarrow_q L$. The assertion $E \hookleftarrow L$ can be understood as a collective fractional pointed-by-heap assertion that covers every memory location that occurs in the environment E.

The length and order of the list E are intended to match the length and order of the list $\mathit{fvclo}(f, \vec{x}, t)$. An environment E is not a runtime object: it is a mathematical object that we use as a parameter of the predicates $\mathit{Closure}$ and Spec.

Closure construction:
$\mu_{\mathsf{clo}} f.\, \lambda \vec{x}.\, t \triangleq$
$\quad \mathsf{let}\, f = \mathsf{alloc}\ (n+1)\, \mathsf{in}$
$\quad f[0] \leftarrow \mathit{codeclo}(f, \vec{x}, t);$
$\quad f[i+1] \leftarrow y_i; \quad \#\ \textit{for each } i \textit{ in } [0, n)$
$\quad f$

Closure code pointer:
$\mathit{codeclo}(f, \vec{x}, t) \triangleq$
$\quad \mu_{\mathsf{ptr}}\, _.\, \lambda(f :: \vec{x}).$
$\quad\quad \mathsf{let}\, y_i = f[i+1]\, \mathsf{in} \quad \#\ \textit{for each } i \textit{ in } [0, n)$
$\quad\quad t$

Closure invocation:
$(v\ \vec{w})_{\mathsf{clo}} \triangleq$
$\quad (v[0]\ (v :: \vec{w}))_{\mathsf{ptr}}$

Side condition:
$\mathit{fvclo}(f, \vec{x}, t) = [y_0; \ldots; y_{n-1}]$

Figure 31: Macros for closure construction and invocation

MkClo

$$\frac{\vec{y} = \mathit{fvclo}(f, \vec{x}, t) \qquad E = \mathit{zip}\ \vec{v}\ \vec{q} \qquad |\vec{v}| = |\vec{y}| \qquad f \notin \vec{x}}{\left\{\begin{matrix}\Diamond(\mathit{size}(1+|E|)) \ * \ \mathit{outside}\, \pi \\ E \leftarrowtail \emptyset\end{matrix}\right\} \ \pi\colon [\vec{v}/\vec{y}]\, (\mu_{\mathsf{clo}} f.\, \lambda \vec{x}.\, t) \ \left\{\lambda \ell.\ \begin{matrix}\mathit{outside}\, \pi \ * \ \mathit{Closure}\, E\, f\, \vec{x}\, t\, \ell \\ \ell \Lleftarrow \{\pi\} \ * \ \ell \leftarrowtail \emptyset\end{matrix}\right\}}$$

CallClo

$$\frac{\begin{matrix}\vec{y} = \mathit{fvclo}(f, \vec{x}, t) \qquad E = \mathit{zip}\ \vec{v}\ \vec{q} \qquad |\vec{x}| = |\vec{w}| \\ \mathit{locs}(\vec{v}) = \mathit{dom}(M) \qquad \{\mathit{outside}\, \pi \ * \ M \Lleftarrow \{\pi\} \ * \ \Phi\}\ \pi\colon [\vec{v}/\vec{y}][\ell/f][\vec{w}/\vec{x}]t\ \{\Psi\}\end{matrix}}{\{\mathit{Closure}\, E\, f\, \vec{x}\, t\, \ell \ * \ \mathit{outside}\, \pi \ * \ M \Lleftarrow \{\pi\} \ * \ \Phi\}\ \pi\colon (\ell\ \vec{w})_{\mathsf{clo}}\ \{\Psi\}}$$

$$\mathit{Closure}\, E\, f\, \vec{x}\, t\, \ell \ * \ \ell \leftarrowtail \emptyset \ * \ \ell \Lleftarrow \emptyset \ \Rrightarrow \ \Diamond(\mathit{size}(1+|E|)) \ * \ \dagger \ell \ * \ E \leftarrowtail \emptyset \qquad \text{CloFree}$$

$$\mathit{Closure}\, E\, f\, \vec{x}\, t\, \ell \ \text{ is persistent} \qquad \text{CloPersist}$$

Figure 32: Low-level API for closures

9.2 Closure Implementation

The definitions of the closure macros $\mu_{\mathsf{clo}} f.\, \lambda \vec{x}.\, t$ and of $(\ell\ \vec{v})_{\mathsf{clo}}$ appear in Figure 31. Both macros generate LambdaFit syntax: that is, the result of their expansion is a LambdaFit expression. We write $t_1\, ;\, t_2$ is as sugar for $\mathsf{let}\, x = t_1\, \mathsf{in}\, t_2$ where $x \notin \mathit{fv}(t_2)$.

The code produced by the macro $\mu_{\mathsf{clo}} f.\, \lambda \vec{x}.\, t$ allocates a block of size $n+1$, stores a code pointer in the first field, stores the values currently bound to the variables $y_0, \ldots, y_{n-1}$ in the remaining fields, and returns the address of this block. The variables $y_0, \ldots, y_{n-1}$ are the free variables of the function $\mu f.\lambda \vec{x}.\, t$, that is, $\mathit{fvclo}(f, \vec{x}, t)$.

The code pointer is produced by the auxiliary macro $\mathit{codeclo}(f, \vec{x}, t)$. It is a closed function whose parameters are f (the closure itself) followed with $\vec{x}$. This function loads the values stored in the closure and binds them to the variables $y_0, \ldots, y_{n-1}$ before executing the body t.

The code produced by the closure invocation macro $(v\ \vec{v})_{\mathsf{clo}}$ first fetches the code pointer that is stored in the first field of the closure, then invokes this code pointer, passing it the closure v itself as well as the actual arguments $\vec{v}$.

9.3 Low-Level Closure API

Our low-level reasoning rules for closures, shown in Figure 32, involve the predicate *Closure*, describing the layout of a closure in memory. Its definition appears in the next section (§9.4).

The rule MkClo specifies a closure construction operation. The term, which is written $[\vec{v}/\vec{y}]\, \mu_{\mathsf{clo}} f.\, \lambda \vec{x}.\, t$, is the application of the substitution $[\vec{v}/\vec{y}]$ to the closure construction macro $\mu_{\mathsf{clo}} f.\, \lambda \vec{x}.\, t$. In this substitution, the variables $\vec{y}$ are the free variables of the function $\mu f.\lambda \vec{x}.\, t$. The reason why we must be prepared to reason about a term of this form is that the premise of LetVal gives rise to substitutions which (after being propagated down) become blocked in front of the *opaque* macro $\mu_{\mathsf{clo}} f.\, \lambda \vec{x}.\, t$. The values $\vec{v}$ that appear in this substitution are

the values "captured" by the closure, that is, the values that are stored in the closure when it is constructed.

In the second premise of MkClo, an environment E is built by pairing up the values $\vec{v}$ with nonzero fractions $\vec{q}$. Then, according to the precondition in MkClo, closure construction consumes $E \hookleftarrow \emptyset$. In other words, for each memory location that occurs in E, it consumes a fractional pointed-by-heap assertion. This records the fact that there exists a pointer from the closure to each such memory location.

According to the precondition in MkClo, closure construction consumes $\mathit{size}(1 + |E|)$ space credits, reflecting the space needed to store a code pointer and the values $\vec{v}$.

Because closure construction involves an allocation, MkClo requires the thread π to be outside a protected section.

According to the postcondition in MkClo, closure construction produces a memory location ℓ. Pointed-by-heap and pointed-by-thread assertions for this memory location are produced, indicating that it is fresh. Furthermore, the assertion $\mathit{Closure}\, E\, f\, \vec{x}\, t\, \ell$, which guarantees that there is a well-formed closure at address ℓ, is also produced. In this thesis, in contrast with our earlier work [Moine et al., 2023], this assertion is *persistent* [Jung et al., 2018b, §2.3]. This means that the knowledge that there is a closure at address ℓ can be shared without any restriction. The pointed-by-heap and pointed-by-thread assertions $\ell \Lleftarrow \{\pi\} * \ell \hookleftarrow \emptyset$ are *not* persistent. Indeed, these assertions allow deallocating the closure, and our program logic ensures that every object is deallocated at most once.

The rule CallClo closely resembles the rule CallPtr for primitive function calls (Figure 18). One difference is that CallClo requires the assertion $\mathit{Closure}\, E\, f\, \vec{x}\, t\, \ell$, which describes the closure. Another difference is that, whereas a primitive function $\mu_{\mathsf{ptr}} f.\, \lambda\vec{x}.\, t$ must be closed, a general function can have a nonempty list of free variables $\vec{y}$, an alias for $\mathit{fvclo}(f, \vec{x}, t)$. In the last premise of CallClo, which requires reasoning about the function's body, the variables $\vec{y}$ are replaced with the values $\vec{v}$ captured at closure construction time, which are recorded in the environment E.

The precondition of CallClo requires a pointed-by-thread assertion $M \Lleftarrow \{\pi\}$, where the domain of the map M includes all of the locations that appear in $\vec{v}$, that is, all of the locations that appear in the closure's environment. This assertion is not consumed: it appears again in the precondition of the triple that forms the last premise of CallClo. In other words, it is transmitted from the caller to the callee. The presence of this assertion is imposed to us by the fact that, when the closure is invoked, these values are read from memory: the load instructions that appear in the definition of $\mathit{codeclo}(f, \vec{x}, t)$ in Figure 31 require pointed-by-thread assertions for the values that are read. If desired, the pointed-by-thread assertion $M \Lleftarrow \{\pi\}$ can be transmitted back from the callee to the caller via a suitable instantiation of the postcondition Ψ. Alternatively, it may be consumed by the callee to justify a logical deallocation operation.

Together, the rules MkClo and CallClo express the correctness of our closure construction and invocation macros. They guarantee that a closure at address ℓ constructed by $[\vec{v}/\vec{y}]\, \mu_{\mathsf{clo}} f.\, \lambda\vec{x}.\, t$, when invoked with actual arguments $\vec{w}$, behaves indistinguishably from the term $[\vec{v}/\vec{y}][\ell/f][\vec{w}/\vec{x}]t$. This is the operational behavior that is expected of a closure.

CloFree logically deallocates a closure. It resembles FreeOne, but, instead of a "*sizeof*" assertion, requires the abstract assertion $\mathit{Closure}\, E\, f\, \vec{x}\, t\, \ell$. Like FreeOne, it produces space credits and a deallocation witness for the closure. Furthermore, CloFree lets the user recover the pointed-by-heap assertion $E \hookleftarrow \emptyset$, thereby undoing the effect of MkClo.

9.4 Low-Level Closure API: Implementation Details

Figure 33 presents the internal definition of the assertion $\mathit{Closure}\, E\, f\, \vec{x}\, t\, \ell$. It records two pure facts: the name f is disjoint from the parameters $\vec{x}$ and the length of the environment E matches the number of free variables of the closure.

$$\begin{array}{rcl} \mathit{Closure}\,E\,f\,\vec{x}\,t\,\ell & \triangleq & \ulcorner f \notin \vec{x} \;\wedge\; |E| = |\mathit{fvclo}(f,\vec{x},t)| \urcorner \;*\; \\ & & \ell \mapsto_{\square} (\mathit{codeclo}(f,\vec{x},t) :: \mathit{map}\,\mathit{fst}\,E) \;*\; \\ & & \boxed{\dagger\ell \vee E \hookleftarrow \{+\ell\}} \end{array}$$

Figure 33: Definition of the predicate *Closure*

$$\textsc{MkSpec}\quad \frac{\begin{array}{c}\vec{y} = \mathit{fvclo}(f,\vec{x},t) \qquad E = \mathit{zip}\;\vec{v}\;\vec{q} \qquad |\vec{v}| = |\vec{y}| \qquad f \notin \vec{x} \qquad n = |\vec{x}| \\ \forall \vec{w}.\; \square\big(\mathit{Spec}\,n\,E\,P\,\ell \twoheadrightarrow P\,\ell\,\vec{w}\,([\vec{v}/\vec{y}][\ell/f][\vec{w}/\vec{x}]t)\big)\end{array}}{\left\{\begin{array}{c}\Diamond(\mathit{size}(1+|E|)) \;*\; \mathit{outside}\,\pi \\ E \hookleftarrow \emptyset\end{array}\right\} \;\pi\colon [\vec{v}/\vec{y}]\,(\mu_{\mathsf{clo}} f.\,\lambda\vec{x}.\,t)\; \left\{\lambda\ell.\;\begin{array}{c}\mathit{outside}\,\pi \;*\; \mathit{Spec}\,n\,E\,P\,\ell \\ \ell \Lleftarrow \{\pi\} \;*\; \ell \hookleftarrow \emptyset\end{array}\right\}}$$

$$\textsc{CallSpec}\quad \frac{\begin{array}{c}E = \mathit{zip}\;\vec{v}\;\vec{q} \qquad \mathit{dom}(M) = \mathit{locs}(\vec{v}) \qquad |\vec{w}| = n \\ \big(\forall u.\; P\,\ell\,\vec{w}\,u \twoheadrightarrow \{\mathit{outside}\,\pi \;*\; M \Lleftarrow \{\pi\} \;*\; \Phi\}\;\pi\colon u\;\{\Psi\}\big)\end{array}}{\{\mathit{Spec}\,n\,E\,P\,\ell \;*\; \mathit{outside}\,\pi \;*\; M \Lleftarrow \{\pi\} \;*\; \Phi\}\;\pi\colon (\ell\;\vec{w})_{\mathsf{clo}}\;\{\Psi\}}$$

$$\begin{array}{rcll} \square\big(\forall \vec{w}\,t.\; P_1\,\ell\,\vec{w}\,t \twoheadrightarrow P_2\,\ell\,\vec{w}\,t\big) \;*\; \mathit{Spec}\,n\,E\,P_1\,\ell & \twoheadrightarrow & \mathit{Spec}\,n\,E\,P_2\,\ell & \textsc{SpecWeak} \\ \mathit{Spec}\,n\,E\,P\,\ell \;*\; \ell \hookleftarrow \emptyset \;*\; \ell \Lleftarrow \emptyset & \Rrightarrow & \Diamond(\mathit{size}(1+|E|)) \;*\; \dagger\ell \;*\; E \hookleftarrow \emptyset & \textsc{SpecFree} \\ & \mathit{Spec}\,n\,E\,P\,\ell \text{ is persistent} & & \textsc{SpecPersist} \end{array}$$

Figure 34: High-level API for closures

Then, a points-to assertion states that the location ℓ points to a block of size $1 + |E|$, whose first field contains the code of the closure, $\mathit{codeclo}(f, \vec{x}, t)$, and whose remaining fields contain the values recorded in the environment E. Because this points-to assertion carries a *discarded fraction* $\square$ [Vindum and Birkedal, 2021], it is a *persistent points-to* assertion. This reflects the fact that the closure is immutable.

The last component in this definition is a liveness-based cancellable invariant (§5.10): a persistent assertion that we can tear down and regain full ownership when we deallocate ℓ.

Since every assertion involved in its definition is persistent, the assertion $\mathit{Closure}\,E\,f\,\vec{x}\,t\,\ell$ is itself persistent.

The liveness-based cancellable invariant contains the pointed-by-heap assertion $E \hookleftarrow \{+\ell\}$, which means that every memory location in E is pointed to by the closure. In the proof of the reasoning rule CloFree, we tear down the liveness-based cancellable invariant, and gain back the assertion $E \hookleftarrow \{+\ell\}$. Because ℓ is now dead, we use the CleanPBHeap rule to change $E \hookleftarrow \{+\ell\}$ into $E \hookleftarrow \emptyset$. This explains how, in the proof of CloFree, we are able to produce the assertion $E \hookleftarrow \emptyset$.

9.5 High-Level Closure API

The user of a program logic is ultimately interested in the specification of a function, not in the details of its implementation. Yet, the predicate $\mathit{Closure}\,E\,f\,\vec{x}\,t\,\ell$ reveals the code of the closure. As a result, a user naturally wishes to hide this information via an existential quantification over this code. This pattern is common enough and technical enough that we offer a higher-level API where this existential quantification is built in. To this end, we introduce the assertion $\mathit{Spec}\,n\,E\,P\,\ell$ (defined further on in §9.6), where n is the arity of the function, E is the environment of the closure, P describes the behavior of the closure, and ℓ is the location of the closure in memory.

Like the *Closure* predicate (§9.3, §9.4), and unlike the *Spec* predicate presented in our previous paper [Moine et al., 2023], the predicate *Spec* is persistent. This enables a better

$$
\begin{array}{l}
\mathit{Spec}\,n\,E\,P\,\ell \;\triangleq \\
\quad \exists\, f\,\vec{x}\,t\,P'. \\
\qquad \ulcorner |\vec{x}| = n \urcorner \;\;*\;\; \mathit{Closure}\,E\,f\,\vec{x}\,t \;\;* \\
\qquad \text{let } \vec{v} = \mathit{map}\,\mathit{fst}\,E \text{ in} \\
\qquad \text{let } \vec{y} = \mathit{fvclo}(f, \vec{x}, t) \text{ in} \\
\qquad \text{let } \mathit{body}\,\vec{w} = [\vec{v}/\vec{y}][\ell/f][\vec{w}/\vec{x}]t \text{ in} \\
\qquad \triangleright\Box(\forall \vec{w}.\ \mathit{Spec}\,n\,E\,P'\,\ell \;\;-\!\!*\;\; P'\,\ell\,\vec{w}\,(\mathit{body}\,\vec{w})) \;\;* \\
\qquad \triangleright\Box(\forall \vec{w}\,u.\ P'\,\ell\,\vec{w}\,u \;\;-\!\!*\;\; P\,\ell\,\vec{w}\,u)
\end{array}
$$

Figure 35: Definition of the predicate *Spec*

separation of concerns between the persistent assertion $\mathit{Spec}\,n\,E\,P\,\ell$, which views the closure as an eternal service provider, and the affine assertion $\ell \Lleftarrow \{\pi\} * \ell \hookleftarrow \emptyset$, which views the closure as an object in memory, allowing it to participate in the object graph and (at some point) to be logically deallocated.

Figure 34 presents the reasoning rules associated with the *Spec* predicate. Let us first examine the rule CallSpec. In many ways, this rule is the same as the low-level rule CallClo. The main difference is that, to prove that the call $(\ell\;\vec{w})_{\mathsf{clo}}$ admits the postcondition Ψ, the user must check that the entailment $\forall u.\ P\,\ell\,\vec{w}\,u \mathrel{-\!\!*} \{\mathit{outside}\,\pi * M \Lleftarrow \{\pi\} * \Phi\}\ \pi\colon u\ \{\Psi\}$ holds. Intuitively, u denotes the instantiated function body that was visible in CallClo; however, this function body is now abstracted away by the universal quantification over u. The predicate P represents the specification of the function, and is typically instantiated with a triple. For example, in the specification of a closure of arity 1 whose effect is to increment a reference r that it receives as an argument, the predicate P takes the form: $\lambda \ell\,\vec{w}\,u.\ \forall\, r\,n.\ \ulcorner \vec{w} = [r] \urcorner \mathrel{-\!\!*} \{r \mapsto [n]\}\ \pi\colon u\ \{\lambda().\ r \mapsto [n+1]\}$. In short, the user must prove an entailment stating that the specification needed by the caller follows from the specification P.

Let us now consider the rule MkSpec. It is again quite similar to the low-level rule MkClo. The premise on the second line ensures that P is a valid description of the behavior of the function body, whose concrete form $[\vec{v}/\vec{y}][\vec{w}/\vec{x}]t$ is visible. In comparison with the low-level API (§9.3), the work of reasoning about the function body is shifted from the closure invocation site to the closure construction site. Moreover, while establishing $P\,\ell\,\vec{w}\,([\vec{v}/\vec{y}][\ell/f][\vec{w}/\vec{x}]t)$, the user is allowed to assume $\mathit{Spec}\,n\,E\,P\,\ell$: this allows verifying recursive calls.

The rule SpecWeak is a consequence rule: it allows weakening the assertion $\mathit{Spec}\,n\,E\,P_1\,\ell$ into $\mathit{Spec}\,n\,E\,P_2\,\ell$, under the hypothesis that P_1 is stronger than P_2.

The rule SpecFree is similar to the rule CloFree.

9.6 High-Level Closure API: Implementation Details

Figure 35 presents the definition of the assertion $\mathit{Spec}\,n\,E\,P\,\ell$. This is a guarded recursive definition: *Spec* appears (under a "later" modality) in its own definition. The definition is existentially quantified over the code of the closure, represented by f, $\vec{x}$, and t. It is also existentially quantified over a predicate P' that is required to be stronger than P. This lets us establish SpecWeak.

Chapter 10

Triples with Souvenir

Yom (2016).
The Old Man.

In this chapter, we introduce *triples with souvenir*, a syntactic sugar that allows for simpler reasoning rules—in particular, a simpler Bind rule—while reasoning about code that lies outside a protected section. We first present the reasoning rules of triples with souvenir (§10.1), then cover how they are defined (§10.2).

10.1 Those Who Cannot Remember the Past Are Condemned to Repeat It

IrisFit, as presented until this point, can be cumbersome to use, for two unrelated reasons.

One reason is that the user must give up pointed-by-thread assertions at each application of Bind, even in the common case where such a fraction has been framed already at a previous application of Bind, which encloses the current application. This obligation to split off and give up pointed-by-thread assertions becomes especially heavy when a variable x denotes a location and has a long *live range*, that is, when this location remains a root throughout a long sequence of instructions. In such a situation, at each point in the sequence, the user is required to split off and give up a fractional pointed-by-thread assertion for x. The problem is partly mitigated by the "no trim" mode ✖ (§6.5). However, this mode is designed for very local use, and cannot be exploited if trimming is needed.

A second reason is that, typically, the large majority of instructions are placed outside protected sections. Yet, the user must provide the assertion $\mathit{outside}\,\pi$ at each application of the *outside rules* Alloc, CallPtr, Fork, Poll, MkSpec, and CallSpec. This is not difficult, but the presence of this assertion creates visual clutter in pre- and postconditions.

To alleviate both problems at once, we introduce *triples with souvenir*, following our earlier work [Moine et al., 2023]. A triple with souvenir takes the form $[R]\,\{\Phi\}\,\pi\colon t\,\{\Psi\}$, where R is a set of locations for which the user has already given up a pointed-by-thread assertion. Recording this *souvenir* (or remembrance) relieves the user from the obligation of giving up another pointed-by-thread assertion at future applications of the Bind rule. Furthermore, a triple with souvenir implicitly carries an $\mathit{outside}\,\pi$ assertion: this allows for more concise statements of the outside rules.

For each reasoning rule in Figure 18, we provide a new rule (not shown) that operates on triples with souvenir and that is polymorphic in R. This is done simply by inserting $[R]$

BindWithSouvenir
$$\frac{\mathit{dom}(M) = \mathit{locs}(K) \setminus R \qquad [R \cup \mathit{locs}(K)]\,\{\Phi\}\,\pi\colon t\,\{\Psi'\} \qquad \forall v.\ [R]\,\{M \Lleftarrow \{\pi\} \;*\; \Psi'\,v\}\,\pi\colon K[v]\,\{\Psi\}}{[R]\,\{M \Lleftarrow \{\pi\} \;*\; \Phi\}\,\pi\colon K[t]\,\{\Psi\}}$$

AddSouvenir
$$\frac{[\{\ell\} \cup R]\,\{\Phi\}\,\pi\colon t\,\{\Psi\}}{[R]\,\{\ell \Lleftarrow_p \{\pi\} \;*\; \Phi\}\,\pi\colon t\,\{\lambda v.\,\ell \Lleftarrow_p \{\pi\} \;*\; \Psi\,v\}}$$

ForgetSouvenir
$$\frac{R' \subseteq R \qquad [R']\,\{\Phi\}\,\pi\colon t\,\{\Psi\}}{[R]\,\{\Phi\}\,\pi\colon t\,\{\Psi\}}$$

Figure 36: Key reasoning rules for triples with souvenir

$$\begin{array}{l}[R]\,\{\Phi\}\,\pi\colon t\,\{\Psi\} \triangleq \\ \quad \forall M.\quad R = dom(M) \implies \\ \qquad \{\Phi \;*\; outside\,\pi \;*\; M \Lleftarrow \{\pi\}\}\;\pi\colon t\;\{\lambda v.\,\Psi\, v \;*\; outside\,\pi \;*\; M \Lleftarrow \{\pi\}\}\end{array}$$

Figure 37: Definition of triples with souvenir

in front every triple that appears in the rule. We do not provide new reasoning rules for protected sections, as triples with souvenir are applicable only outside protected sections.

The new reasoning rules that make use of souvenirs appear in Figure 36. BindWith-Souvenir is what we aimed for: it is our motivation for introducing triples with souvenir. It closely resembles Bind, but does not require the user to give up pointed-by-thread assertions for the locations that are already part of the souvenir R. The first premise requires the domain of M (a map of locations to nonzero fractions) to cover all roots of the evaluation context K, except those that are already in the souvenir R. In other words, *if a location already appears in R then there is no need to again split off and give up a pointed-by-thread assertion for this location.* Furthermore, BindWithSouvenir augments the current souvenir by changing R to $R \cup locs(K)$ in its second premise. Thus, nested applications of this rule do not require repeatedly and redundantly giving up pointed-by-thread assertions. The rule AddSouvenir extends the current souvenir with a location ℓ. This requires the user to frame out (that is, temporarily give up) a pointed-by-thread assertion for ℓ. The rule ForgetSouvenir shrinks the current souvenir.

By exploiting triples with souvenir, each of the outside rules mentioned above can be given a more concise statement. For example, the reasoning rule Poll can be more concisely formulated as PollWithSouvenir:

Poll
$\{outside\,\pi\}\;\pi\colon \mathsf{poll}\;\{\lambda().\;outside\,\pi\}$

PollWithSouvenir
$[R]\{\ulcorner\mathrm{True}\urcorner\}\;\pi\colon \mathsf{poll}\;\{\lambda().\;\ulcorner\mathrm{True}\urcorner\}$

10.2 Internals of Souvenirs

The definition of triples with souvenir appears in Figure 37. A triple with souvenir $[R]\,\{\Phi\}\,\pi\colon t\,\{\Psi\}$ is expressed as an ordinary triple where the assertions $outside\,\pi$ and $M \Lleftarrow \{\pi\}$ are framed out. That is, these assertions appear in the pre- and postcondition, so they are required and preserved, but they are not made available to a user who views a triple with souvenir as an abstract assertion. The domain of the map M is the set R: this ensures that, for every location in this set, a fractional pointed-by-thread assertion is indeed framed out.

A triple with souvenir describes a piece of code whose execution begins and ends outside a protected section: it cannot be used to describe a code fragment that lies inside a protected section. To establish a triple with souvenir about a whole protected section, the user must unfold the definition of triples with souvenir and drop down to the level of standard triples. Then, all of the reasoning rules for standard triples are applicable.

In our mechanization [Moine, 2024], we use a more general triple that allows both "no trim" mode (§6.5) without a souvenir and normal mode with a souvenir. This general triple always frames out an "*outside*" assertion. In our case studies, this is the triple that we use most of the time.

Chapter 11

Sequential Case Studies

Vaughan, S. (1954).
Lullaby of Birdland.

Before diving into the deep waters of concurrent cases studies (§12), we first showcase *sequential* cases studies. We start by presenting our generic approach for representing containers (§11.1). Then, we explain how to reason about linked lists reversal and concatenation (§11.2). We next focus on a continuation-passing-style (CPS) implementation of the concatenation of lists, illustrating the use of closures (§11.3). We then present a specification for a stack abstract data type (§11.4). The first implementation demonstrates a usage of our linked lists. The second implementation relies on a mutable array. The third implementation is a generic construction of a stack as a stack of stacks. It demonstrates modular reasoning as well as an amortized space complexity analysis that exploits rational space credits. We conclude this chapter by commenting on an implementation of circular singly-linked lists, showcasing the reasoning about cycles in IrisFit and specifically their deallocation (§11.5).

We present here "sequential" case studies in the sense that it is unsafe for two concurrent threads to use two functions of the API without proper synchronization. However, our specifications allow for two threads to synchronize (via a lock or another mechanism) and use one after the other two functions of the API.

In this chapter and the following one, for each case study, we present the code, the specification, and a few insights from the proof. These insights contain Iris proof details and can be skipped by non-experts.

For establishing concrete heap bounds, we pose that a block of n fields is represented by n memory words, that is, we pose $\mathit{size}(n) = n$. Another practical choice, such as $\mathit{size}(n) = n+1$, would only affect the constant values that appear behind diamond symbols in specifications.

11.1 Containers: A Generic Approach

The following cases studies (§11.4–11.5), as well as Treiber's stack (§12.5) and Michael and Scott's queue (§12.6) showcase *containers*, that is, data structures representing collections of objects. All of these case studies follow the same approach, which we summarize in this section: this approach envisions how we specify and verify containers in IrisFit.

First, as in standard Separation Logic, a container is a value and has a *model*: the mathematical data structure represented by the container. For example, if the container implements a (possibly mutable) list, the model is a mathematical list of the values being stored. The *representation predicate* is a Separation Logic assertion that relates the container to its model.

In our setting, the model contains not only values, but also two fractions per value: one for an associated pointed-by-thread assertion and one for an associated pointed-by-heap assertion. Following our example of a list, the model is a mathematical list of triples (v, p, q) of a value v and two positive fractions p and q.

Each time a value v is added to the container, the user provides a fraction p of the associated pointed-by-thread assertion $v \Lleftarrow_p \{\pi\}$ as well as a (positive) fraction q of the associated pointed-by-heap assertion $v \hookleftarrow^0_q \emptyset$. The user gets back these two assertions when the value is removed from the container. What are these assertions used for?

First, the pointed-by-thread assertion allows the functions of the container's API to manipulate the stored values. Indeed, recall that loading a value from the heap requires updating a

fraction of its pointed-by-thread assertion (Load). Moreover, Bind frames pointed-by-thread assertions of locations of the evaluation context. Hence, we embark on the representation predicate a fraction of pointed-by-thread for each value being stored in the container. This approach is more convenient than requiring the user to provide pointed-by-threads assertions for all the values being loaded in the premise of each specification of the API.

Second, the pointed-by-heap assertion allows not revealing the internal pointers of the data structure. We already followed this approach for closures (§9.2). Indeed, the values being stored inside a container are pointed by internal heap blocks of the data structure, and this information must be recorded inside a pointed-by-heap assertion. Yet, we do not want to reveal the exact location of internal pointers of the data structure: after all, they are *internal*. Hence, we rather embark on the representation predicate a fraction of pointed-by-heap for each value, used to record the internal heap blocks pointing to the value.

We note that our approach has two limitations. First, it leads the user to manipulate a lot of fractions. Indeed, the user must provide two fractions per value inserted in the container, which can be tedious to provide in practice. One way to overcome this limitation in practice is to equip every value with the same fraction of pointed-by-thread and -heap, but it leads to a lack of expressivity: if a fraction $1/n$ is fixed *a-priori*, then the user will not be able to insert the same value more than n times in the container. Second, our approach does not scale well with persistent containers. Indeed, in our approach, "peeking" a value from a container (that is, loading the value without removing it) updates the model of the container. Indeed, the "peek" operation needs to update a fraction of the pointed-by-thread of a loaded value. This fraction must be given to the user (in order for trimming the set of predecessors threads, for example), but another fraction must also be kept inside the representation predicate to allow for another peek later on. This reason explains why our representation predicates for closures (§9.3, §9.5) do not follow the general recipe we describe here.

11.2 Linked Lists and Linked List Reversal

In this section, we present our encoding of lists. We also comment on the implementation and specification of two standard functions on lists: *rev_append* and *rev*.

Code The constructor "nil" is implemented by the unit value (). The constructor "cons" is implemented by a block of size 2, whose first offset stores the head, and the second offset the tail. This representation is faithful to space consumption in OCaml: constant constructors are compiled into integers, which are not heap allocated.

We define the list API with functions *nil*, *is_nil*, *head*, *tail* and *cons* in Figure 38.

We make use of this API to implement *rev_append* and *rev*. This function *rev_append* expects two lists xs and ys and returns a list whose elements are the elements of xs in reverse order followed by the elements of ys. This function *rev_append* is recursive and is defined by induction over its first argument xs. The function *rev* expects a list xs and returns a new list with the element of xs in reverse order. The definition of *rev* makes a direct use of *rev_append*.

Specifications The assertion *list* L xs asserts that xs represents a well-formed linked list whose logical model is L, a mathematical list of triples of a value v and two positive fractions p and q, following our general recipe for containers (§11.1).

We omit the specifications of base functions on lists. The specifications of *rev_append* and *rev* appear in Figure 38. For each of these two functions, we propose two specifications, depending on whether their first argument should be kept or can be logically deallocated. Moreover, these four specifications are independent of the choice of the current thread π, reflecting the fact that these specifications can be used by arbitrary threads.

$$
\begin{array}{l}
\mathit{nil} \triangleq \mu_{\mathsf{ptr}}_.\ \lambda[].\ () \\
\mathit{is_nil} \triangleq \mu_{\mathsf{ptr}}_.\ \lambda[xs].\ xs = () \\
\mathit{head} \triangleq \mu_{\mathsf{ptr}}_.\ \lambda[xs].\ xs[0] \\
\mathit{tail} \triangleq \mu_{\mathsf{ptr}}_.\ \lambda[xs].\ xs[1] \\
\mathit{cons} \triangleq \mu_{\mathsf{ptr}}_.\ \lambda[v, xs]. \\
\quad \mathsf{let}\ ys = \mathsf{alloc}\ 2\ \mathsf{in} \\
\quad ys[0] \leftarrow v\ ;\ ys[1] \leftarrow xs\ ;\ ys
\end{array}
\qquad
\begin{array}{l}
\mathit{rev_append} \triangleq \mu_{\mathsf{ptr}} f.\ \lambda[xs;\ ys]. \\
\quad \mathsf{if}\ (\mathit{is_nil}\ xs)_{\mathsf{ptr}}\ \mathsf{then}\ ys\ \mathsf{else} \\
\quad\quad \mathsf{let}\ x = (\mathit{head}\ [xs])_{\mathsf{ptr}}\ \mathsf{in} \\
\quad\quad \mathsf{let}\ xs' = (\mathit{tail}\ [xs])_{\mathsf{ptr}}\ \mathsf{in} \\
\quad\quad \mathsf{let}\ ys' = (\mathit{cons}\ [x;\ ys])_{\mathsf{ptr}}\ \mathsf{in} \\
\quad\quad (f\ [xs';\ ys'])_{\mathsf{ptr}} \\
\\
\mathit{rev} \triangleq \mu_{\mathsf{ptr}}_.\ \lambda[xs]. \\
\quad \mathsf{let}\ ys = (\mathit{nil}\ [])_{\mathsf{ptr}}\ \mathsf{in} \\
\quad (\mathit{rev_append}\ [xs;\ ys])_{\mathsf{ptr}}
\end{array}
$$

$$
[\mathit{locs}(xs)] \left\{\begin{array}{c} \Diamond(2 \times |L_x|) \\ \mathit{list}\ L_x\ xs\ *\ \mathit{list}\ L_y\ ys \\ ys \Lleftarrow \{\pi\}\ *\ ys \mapsfrom \emptyset \end{array}\right\}\ \pi\colon (\mathit{rev_append}\ [xs;\ ys])_{\mathsf{ptr}}\ \left\{\begin{array}{c} \mathit{list}\ L_x\ xs \\ \lambda zs.\ \mathit{list}\ (\mathsf{rev}(\tfrac{1}{2}L_x) \mathbin{+\!\!+} L_y)\ zs \\ zs \Lleftarrow \{\pi\}\ *\ zs \mapsfrom \emptyset \end{array}\right\}
$$

$$
[\emptyset] \left\{\begin{array}{c} \mathit{list}\ L_x\ xs\ *\ \mathit{list}\ L_y\ ys \\ xs \Lleftarrow \{\pi\}\ *\ xs \mapsfrom \emptyset \\ ys \Lleftarrow \{\pi\}\ *\ ys \mapsfrom \emptyset \end{array}\right\}\ \pi\colon (\mathit{rev_append}\ [xs;\ ys])_{\mathsf{ptr}}\ \left\{\lambda zs.\ \begin{array}{c} \mathit{list}\ (\mathsf{rev}(L_x) \mathbin{+\!\!+} L_y)\ zs \\ zs \Lleftarrow \{\pi\}\ *\ zs \mapsfrom \emptyset \end{array}\right\}
$$

$$
[\mathit{locs}(xs)] \left\{\begin{array}{c} \Diamond(2 \times |L_x|) \\ \mathit{list}\ L_x\ xs \end{array}\right\}\ \pi\colon (\mathit{rev}\ [xs])_{\mathsf{ptr}}\ \left\{\begin{array}{c} \mathit{list}\ L_x\ xs \\ \lambda zs.\ \ \mathit{list}\ (\mathsf{rev}(\tfrac{1}{2}L_x))\ zs \\ zs \Lleftarrow \{\pi\}\ *\ zs \mapsfrom \emptyset \end{array}\right\}
$$

$$
[\emptyset] \left\{\begin{array}{c} \mathit{list}\ L_x\ xs \\ xs \Lleftarrow \{\pi\}\ *\ xs \mapsfrom \emptyset \end{array}\right\}\ \pi\colon (\mathit{rev}\ [xs])_{\mathsf{ptr}}\ \left\{\lambda zs.\ \begin{array}{c} \mathit{list}\ \mathsf{rev}(L)\ zs \\ zs \Lleftarrow \{\pi\}\ *\ zs \mapsfrom \emptyset \end{array}\right\}
$$

Figure 38: Code and specification of linked list reversal

We first comment on the specifications for *rev_append*. The first specification of the call $(\mathit{rev_append}\ [xs; ys])_{\mathsf{ptr}}$ allows the caller to retain the root xs: this is expressed by a souvenir on $\mathit{locs}(xs)$ (recall that xs is either a location or the unit value, hence $\mathit{locs}(xs)$ produces either the singleton holding the location or the empty set). The first specification asserts that *rev_append* has *linear* heap space complexity: it requires $2 \times |L|$ space credits. The second specification requires the caller to provide (and give up) a unique pointer to xs and asserts that *rev_append* has *constant* heap space complexity. Indeed, in this case, *rev_append* requires *zero* space credits because, at each step, one cell of the list xs can be logically freed before one new list cell is allocated. Both specifications require a unique pointer on ys, that is its full pointed-by-thread and its full empty pointed-by-heap assertions. This is necessary because ys becomes a suffix of the list that is returned by *rev_append*. If the caller was allowed to keep a copy of the pointer ys, then this copy would become a pointer from the outside to an internal cell, a situation which our definition of *list* forbids.

The two specifications differ slightly in their postconditions. The postcondition of the second specification describes the output list as $\mathit{list}\ (\mathsf{rev}(L_x) \mathbin{+\!\!+} L_y)\ zs$. The postcondition of the first specification is more complex because the values contained in the input list xs become shared between the input list xs and the output list zs. We express this by splitting fractions: $\frac{1}{2}L_x$ denotes a copy of the list L_x where the fraction associated with every value has been halved.

The two specifications for *rev* that appear at the bottom of Figure 38 are then obtained by specializing ys to nil.

Proof Insights The assertion $\mathit{list}\ L_x\ xs$ is defined in Figure 39 This predicate describes a list *without sharing*, that is, each cons block is the only one to point to its tail.

The definition of $\mathit{list}\ L\ xs$ has a standard overall structure [Reynolds, 2002]. The novelty comes with the presence of pointed-by-thread and -heap assertions. The assertion $\mathit{list}\ L\ xs$ is

$$
\begin{array}{l}
\mathit{list}\ L\ xs \quad \triangleq \\
\qquad \ulcorner L = [\,] \ \wedge\ xs = ()\urcorner \\
\quad \vee\ \exists v\, p\, q\, L'\, xs'.\ \ulcorner L = (v,p,q) :: L'\urcorner \ * \\
\qquad xs \mapsto [v; xs'] \ * \ v \Leftarrow_p \emptyset \ * \ v \hookleftarrow^0_q \{+xs\} \ * \\
\qquad xs' \Leftarrow \emptyset \ * \ xs' \hookleftarrow \{+xs\} \ * \ \mathit{list}\ L'\ xs'
\end{array}
$$

Figure 39: Internals of linked lists

defined by induction over L. If $L = [\,]$, this predicate boils down to the pure assertion $xs = ()$. Otherwise, if $L = (v, p, q) :: L'$, the predicate begins with the points-to assertion $xs \mapsto [v; xs']$, which describes a 2-field cell. The value xs' of the following cell is existentially quantified. Moreover, the assertion contains the pointed-by-thread $v \Leftarrow_p \emptyset$ and pointed-by-heap $v \hookleftarrow^0_q \{+xs\}$ of the stored value v. Then, the assertion contains the full pointed-by-thread $xs' \Leftarrow \emptyset$ and pointed-by-heap $xs' \hookleftarrow \{+xs\}$ of the next cell xs'. The pointed-by-thread is empty (xs' is internal and is not a root of any thread), and the pointed-by-heap contains only xs as a predecessor. These two assertions mean that there are no other pointers (from the heap or the stack) to xs'. The definition concludes with the recursive ownership of the list *list* $L'\ xs'$.

With this definition, there can be no direct pointers from the outside to an internal cell. The ability to express this property is unusual: indeed, via points-to assertions, traditional Separation Logic can express *unique ownership*, that is, control who may dereference a pointer; however, it cannot express the fact that a pointer is unique. The predicate *list* in traditional Separation Logic does forbid two valid linked lists from sharing a suffix, but does *not* rule out the existence of a rogue pointer (without any access permission) from the outside into a linked list.

11.3 Continuation-Passing Style

To demonstrate our ability to reason about use of closures, we present a function that constructs the concatenation of two linked lists and is written in continuation-passing-style (CPS). This style leads to a nontrivial chain of closures which are allocated and deallocated over the time of a recursion. More precisely, The continuations involved in this example are one-shot, that is, called only once. They are self-destructing continuations: in our proofs, we logically deallocate them as soon as they are invoked.

Code Our implementation appears in Figure 40. The main function, *append*, expects two linked lists xs and ys. It first allocates a (recursive) closure *aux*, described below, which closes over ys. Then, it invokes this closure, with a closure for the identity function as a continuation.

The function *aux* expects two arguments: a linked list xs and a continuation (a closure) k. If xs is nil, then it applies the closure k to the linked list ys. Otherwise, it allocates a new closure k', whose purpose is to "cons" the element x in front of the linked list produced by the concatenation of xs' and ys. The closure k' captures the values of k, x and xs. After allocating this closure, *aux* invokes itself with arguments xs' and k'.

Specifications Like *rev_append* and *rev* (§11.2), *append* admits two specifications presented in Figure 40, which differ in their assumption about xs. If the linked list xs comes with a full empty pointed-by-thread and pointed-by-heap, then it can be logically deallocated, which pays for the space occupied by the new list that is constructed; otherwise, this space must be paid for. Besides, internally, *append* needs a certain amount of temporary storage, whose size is linear in the length of the list xs, and which is released when *append* returns.

$$
\begin{array}{l}
\mathit{append} \triangleq \mu_{\mathsf{ptr}_}.\, \lambda[xs; ys]. \\
\quad \mathsf{let}\ \mathit{aux} = \mu_{\mathsf{clo}} f.\, \lambda[xs; k]. \\
\quad\quad \mathsf{if}\ (\mathit{is_nil}\ [xs])_{\mathsf{ptr}}\ \mathsf{then}\ (k\ [ys])_{\mathsf{clo}}\ \mathsf{else} \\
\quad\quad\quad \mathsf{let}\ x = (\mathit{head}\ [xs])_{\mathsf{ptr}}\ \mathsf{in} \\
\quad\quad\quad \mathsf{let}\ xs' = (\mathit{tail}\ [xs])_{\mathsf{ptr}}\ \mathsf{in} \\
\quad\quad\quad \mathsf{let}\ k' = \mu_{\mathsf{clo}_}.\, \lambda[r]. \\
\quad\quad\quad\quad \mathsf{let}\ p = (\mathit{cons}\ [x; r])_{\mathsf{ptr}}\ \mathsf{in}\ (k\ [p])_{\mathsf{clo}}\ \mathsf{in} \\
\quad\quad\quad (f\ [xs'; k'])_{\mathsf{clo}}\ \mathsf{in} \\
\quad \mathsf{let}\ \mathit{id} = \mu_{\mathsf{clo}_}.\, \lambda[x].\, x\ \mathsf{in} \\
\quad (\mathit{aux}\ [xs; \mathit{id}])_{\mathsf{clo}}
\end{array}
$$

$$
[\mathit{locs}(xs)] \left\{ \begin{array}{c} \Diamond(2 \times 3 \times |L_x| + 3) \\ \mathit{list}\ L_x\ xs \ * \ \mathit{list}\ L_y\ ys \\ ys \Lleftarrow \{\pi\} \ * \ ys \hookleftarrow \emptyset \end{array} \right\} \pi \colon (\mathit{append}\ [xs; ys])_{\mathsf{ptr}} \left\{ \lambda zs. \begin{array}{c} \Diamond(3 \times |L_x| + 3) \\ \mathit{list}\ (\tfrac{1}{2} L_x)\ xs \\ \mathit{list}\ (\tfrac{1}{2} L_x +\!\!+ L_y)\ zs \\ zs \Lleftarrow \{\pi\} \ * \ zs \hookleftarrow \emptyset \end{array} \right\}
$$

$$
[\emptyset] \left\{ \begin{array}{c} \Diamond(3 \times |L_x| + 3) \\ \mathit{list}\ L_x\ xs \ * \ \mathit{list}\ L_y\ ys \\ xs \Lleftarrow \{\pi\} \ * \ xs \hookleftarrow \emptyset \\ ys \Lleftarrow \{\pi\} \ * \ ys \hookleftarrow \emptyset \end{array} \right\} \pi \colon (\mathit{append}\ [xs; ys])_{\mathsf{ptr}} \left\{ \lambda zs. \begin{array}{c} \Diamond(3 \times |L_x| + 3) \\ \mathit{list}\ (L_x +\!\!+ L_y)\ zs \\ zs \Lleftarrow \{\pi\} \ * \ zs \hookleftarrow \emptyset \end{array} \right\}
$$

Figure 40: Code and specification of linked list concatenation in continuation-passing style

This temporary storage is described by the space credits that appear both in the precondition and in the postcondition.

The number $3 \times |L_x|$ that appear in these specifications, where $|L_x|$ denotes the length of the linked list L_x, corresponds to the space usage of the linked chain of continuations that is formed in the heap. In the first triple, an additional $2 \times |L_x|$ credits are needed, because of the allocation of new linked list cells. One credit is used by the identity closure. Another two credits are used by the closure *aux*.

Proof Insights This example serves mainly as an exercise of our reasoning rules on closures. Proof details can be found in our mechanization [Moine, 2024].

11.4 Sequential Stacks

We verify three implementations of sequential stacks: an unbounded-capacity mutable stack implemented as a linked list, a bounded-capacity stack implemented as an array, and a functor that constructs a stack of stacks. We show that they all satisfy the same specifications, up to a potential capacity bound and their space complexity.

Code Our stacks are *potentially bounded*, meaning that some implementations can store only a fixed amount of elements. Each of our three implementations provides a function *create* to allocate a new stack, *push* to push elements onto a non-full stack, and *pop* to pop elements from a non-empty stack. Two additional operations (not shown) allow testing whether a stack is empty and testing whether a stack is full.

The first implementation is an unbounded stack as a reference on an immutable list.

The second implementation is a bounded stack as a mutable pair where one field holds the logical size of the stack and one field holds a pointer to a (fixed-capacity) array. Every unused cell in this array is filled with a unit value.

$$\{\Diamond A\} \quad \pi\colon (\mathit{create}\ [])_{\mathsf{ptr}} \quad \left\{\lambda \ell.\ \begin{array}{c}\mathit{stack}\,\ell\,[] \\ \ell \Lleftarrow \{\pi\} \ * \ \ell \hookleftarrow \emptyset\end{array}\right\}$$

$$[\{\ell\}] \left\{\begin{array}{c}\ulcorner |L| < C \urcorner \\ \mathit{stack}\,\ell\,L \ * \ \Diamond B \\ v \Lleftarrow_p \{\pi\} \ * \ v \hookleftarrow^0_q \emptyset\end{array}\right\} \pi\colon (\mathit{push}\ [\ell; v])_{\mathsf{ptr}} \quad \{\lambda().\ \mathit{stack}\,\ell\,((v,p,q) :: L)\}$$

$$[\{\ell\}] \{\mathit{stack}\,\ell\,((v,p,q) :: L)\} \quad \pi\colon (\mathit{pop}\ [\ell])_{\mathsf{ptr}} \quad \left\{\lambda v.\ \begin{array}{c}\mathit{stack}\,\ell\,L \ * \ \Diamond B \\ v \Lleftarrow_p \{\pi\} \ * \ v \hookleftarrow^0_q \emptyset\end{array}\right\}$$

$$\mathit{stack}\,\ell\,L \ * \ \ell \Lleftarrow \emptyset \ * \ \ell \hookleftarrow \emptyset \qquad \Rrightarrow \qquad \Diamond(A + B \times |L|) \ * \mathop{\Large *}_{(v,p,q)\in L} (v \Lleftarrow_p \emptyset \ * \ v \hookleftarrow^0_q \emptyset)$$

Figure 41: Specification of possibly-bounded sequential stacks
In this figure, A, B and C are constants determined from the implementation.

The third implementation is generic: it is a functor that expects two implementations of stacks, say X*-stacks* and Y*-stacks*, and produces a new implementation, say Z*-stacks*. A Z-stack is implemented as a pair made of (1) a nonempty Y*-stack* storing the elements at the top of the stack, and (2) a X*-stack* of full Y*-stacks*, storing all the remaining elements. As a result, a Z-stack is bounded if and only if both X and Y stacks are bounded. By applying the functor to our previous two implementations of stacks as arrays and stacks as linked lists, one obtains a time- and space-efficient implementation of *chunked stacks*, that is, linked lists of fixed-capacity arrays.

We omit the exact code of our implementations for brevity. The interested reader may find it in our mechanization [Moine, 2024].

Specifications Figure 41 presents the common interface of all our stacks. This interface is parameterized with a capacity C, which is either an integer or $+\infty$, the latter denoting an unbounded stack. The interface is also parameterized with two constants: A is the number of credits required to allocate an empty stack, and B is the number of credits required by a push operation.

The specifications rely on the abstract predicate $\mathit{stack}\,L\,\ell$, which asserts that at address ℓ there is a valid stack whose elements are described by the mathematical list L. Following our recipe for containers (§11.1), L is a list of triples of value and two nonzero fractions. According to our specifications, *create* consumes A space credits and produces a fresh empty stack; *push* consumes B space credits and a fractional handle for the value that is inserted into the stack; *pop* gives up these assertions. In addition, *push* requires the number of elements in the stack to be less than the stack's capacity C. This requirement is trivially satisfied if C is $+\infty$. Finally, the logical deallocation of a stack allows recovering all of the space occupied by the stack, namely $A + B \times |L|$ space credits, where $|L|$ is the number of elements of the stack.

Our three implementations of stacks (not shown) differ in their space complexity. Each of them is verified with respect to a particular instantiation of the parameters A, B, and C.

Recall that the first implementation consists of a reference on an immutable list. The reference occupies 1 word, an empty list occupies 0 word, and each list cell occupies 2 words. This stack has unbounded capacity. Hence, this implementation satisfies our common interface for the parameters $A = 1$, $B = 2$, and $C = +\infty$.

Recall that the second implementation consists of a mutable pair of an array and an offset. Let us call T the fixed capacity of the array. This implementation satisfies our interface with creation cost $A = T + 1$, insertion cost $B = 0$, and bounded capacity $C = T$.

Recall that our third implementation is generic: it is a functor that expects two implementations of stacks, X-stacks and Y-stacks, and produces a new implementation, Z-stacks. To simplify the explanations, we assume that Y-stacks are bounded—an assumption that our formalization does not make. Let us write $X.A$ and $X.B$ and $X.C$ for the space complexity

```
copy2 ≜ μptr_. λ[s; t].
  t[0] ← s[0] ; t[1] ← s[1]

ccons ≜ μptr_. λ[v; c].
  let xs = c[0] in
  if xs = () then
    let ys = alloc 2 in
    ys[0] ← v ; ys[1] ← ys
    c[0] ← ys
  else
    let ys = alloc 2 in
    (copy2 [xs; ys])ptr
    xs[0] ← v ; xs[1] ← ys

cnext ≜ μptr_. λ[c].
  let xs = c[0] in
  if xs = () then () else c[0] ← xs[1]

cnil ≜ alloc 1

cuncons ≜ μptr_. λ[c].
  let xs = c[0] in
  let v = xs[0] in let ys = xs[1] in
  (if xs = ys then c[0] ← () else (copy2 [ys; xs])ptr) ;
  v

cappend ≜ μptr_. λ[cxs; cys].
  let ys = cys[0] in
  if ys = () then () else
  let xs = cxs[0] in
  cxs[0] ← ys ;
  if xs = () then () else
  let v = xs[0] in let xs' = xs[1] in
  (copy2 [ys; xs])ptr ;
  ys[0] ← v ; ys[1] ← xs'
```

$$[\{c\}]\ \{\mathit{clist}\ L\ c\} \quad \pi\colon (\mathit{cnext}\ [c])_{\mathsf{ptr}} \quad \{\lambda().\ \mathit{clist}\ (\mathsf{rot}_1(L))\ c\}$$

$$[\{c\}] \left\{ \begin{array}{c} \mathit{clist}\ L_1\ c_1 \ * \ \mathit{clist}\ L_2\ c_2 \\ c_2 \Lleftarrow \{\pi\} \ * \ c_2 \leftarrow \emptyset \end{array} \right\} \pi\colon (\mathit{cappend}\ [c_1; c_2])_{\mathsf{ptr}} \left\{ \lambda().\ \begin{array}{c} \Diamond 1 \\ \mathit{clist}\ (L_1 +\!\!+ L_2)\ c_1 \end{array} \right\}$$

Figure 42: Code and specification of circular singly-linked lists

parameters of X-*stacks*, and likewise for Y-*stacks*. We formally establish that our Z-*stacks* have creation cost $A = X.A + Y.A + 2$, insertion cost $B = Y.B + (Y.A + X.B)/Y.C$, and capacity $C = X.C \times (1 + Y.C)$. The insertion cost is of particular interest. An empty Y-*stack* is allocated and pushed on the X-*stack* only every $Y.C$ *push* operations on the Z-*stack*: this explains the fractional cost $(Y.A + X.B)/Y.C$. Obtaining this bound requires rational space credits and an amortized analysis, which involves defining a suitable potential function and saving space credits in the definition of *stack* for Z-stacks.

By applying the functor to our previous two implementations of stacks as arrays and stacks as linked lists, one obtains a time- and space-efficient implementation of *chunked stacks*, that is, linked lists of fixed-capacity arrays.

11.5 A Circular Singly-Linked List

To demonstrate our ability to reason about circular data structures, we specify and verify circular singly-linked lists. Although circular doubly-linked lists are more useful in practice, verifying circular singly-linked lists is already a nontrivial and an interesting exercise which captures the difficulty in terms of pointers and space usage. We posit that the specifications and proofs of circular doubly linked-list can follow the same pattern.

Code We encode circular lists as a reference on either the unit value (), denoting an empty circular list, or on a circular chain of 2-cell blocks. As for non-circular lists (§11.2), the first offset of this block stores a value and its second offset stores the next block. If any, we refer to the first list cell pointed by the main reference as the "focus" of the list. Our implementation of circular lists is non-persistent. In particular, there is no sharing possible and every operation requires the full ownership of its arguments.

$$
\begin{array}{l}
\mathit{seg}\,L\,xs\,ys \;\triangleq\; \exists v\,p\,q\,L'\,xs'. \\
\quad \ulcorner L = (v,p,q) :: L' \urcorner \;*\; xs \mapsto [v; xs'] \;*\; v \hookleftarrow^{\geq 0}_{q} \{+xs\} \;*\; v \Leftarrow\!\dashv_{p} \emptyset \;* \\
\quad xs' \Leftarrow\!\dashv_{\frac{1}{2}} \emptyset \;*\; xs' \hookleftarrow_{\frac{1}{2}} \{+xs\} \;* \\
\quad (\ulcorner L' = [] \wedge xs' = ys \urcorner \;\vee\; xs' \Leftarrow\!\dashv_{\frac{1}{2}} \emptyset \;*\; xs' \hookleftarrow_{\frac{1}{2}} \emptyset \;*\; \mathit{seg}\,L'\,xs'\,ys) \\
\mathit{clist}\,L\,c \;\triangleq \\
\qquad \ulcorner L = [] \urcorner \;*\; c \mapsto [()] \\
\quad \vee\; \ulcorner L \neq [] \urcorner \;*\; \exists xs.\, c \mapsto [xs] \;*\; xs \Leftarrow\!\dashv_{\frac{1}{2}} \emptyset \;*\; xs \hookleftarrow_{\frac{1}{2}} \{+c\} \;*\; \mathit{seg}\,L\,xs\,xs
\end{array}
$$

Figure 43: Internals of circular singly-linked lists

The code appears on top of Figure 42. The auxiliary function copy_2 takes two 2-cell blocks s and t, and copies the content of s, the source, to t, the target. The function *cnil* takes no argument and returns a 1-cell block pointing to unit, that is, an empty circular list. The function *ccons* takes a value v and a list c and adds a new list cell in focus. The function *cuncons* takes a non-empty list c, removes the focused cell, and returns the value its store. These three functions—*cnil*, *ccons* and *cuncons*—form a "stack" API.

Interestingly, circular lists come with two additional functions with constant time complexity, improving the usually linear time complexity of their implementation on non-circular lists. The function *cnext* takes a circular list as an argument and rotate its content of one block. If the list is empty, it is left untouched. Otherwise, the current focus is switched to its successor. The function *cappend* concatenates in place two circular lists and stores the result in the first argument.

Specifications Our specifications make use of a representation predicate $\mathit{clist}\,L\,c$, asserting that the circular list c has model L, a mathematical list of triples of a value v and two positive fractions p and q. The functions with a stack API—*cnil*, *ccons* and *cuncons*—obey the specifications presented in Figure 41, where *cnil* is *create*, *ccons* is *push*, and *cuncons* is *pop*. The predicate *stack* is *clist*, the cost of creation A is 1—the space occupied by a reference—and the cost of a cell B is 2—the space occupied by a 2-cell block.

Crucially, our circular list also obeys the logical deallocation specification of stacks presented in Figure 41. The user can logically deallocate the circular list as soon as its sole entry point, the reference, is unreachable.

The specifications of *cnext* and *cappend* at the bottom of Figure 42. The specification of $(\mathit{cnext}\ [c])_{\mathsf{ptr}}$ is a souvenir on c and requires that c represents a circular list of model L. The function call updates the model from L to $\mathsf{rot}_1(L)$, where $\mathsf{rot}_1([]) = []$ and $\mathsf{rot}_1(x :: L) = L +\!\!+ [x]$. The specification of $(\mathit{cappend}\ [c_1; c_2])_{\mathsf{ptr}}$ is a souvenir on c_1. The precondition consumes the assertions witnessing that c_1 is a circular list of model L_1 and c_2 is a circular list of model L_2. The precondition also consumes the full pointed-by-thread and empty pointed-by-heap if c_2. The postcondition produces one space credit corresponding to the reference c_2, which is logically deallocated, as well as the representation predicate of c_1 which now represents $L_1 +\!\!+ L_2$.

Proof Insights Figure 43 presents the definitions of our assertions. First, the assertion $\mathit{seg}\,L\,xs\,ys$ asserts that xs represents a non-empty segment of a list whose first block is xs and last block is ys. This predicate is a variation of the *list* predicate (§11.2). However, *seg* stores only half of the pointed-by-thread and pointed-by-heap assertions of the focused cell. The definition of $\mathit{clist}\,L\,c$ appears after and cases over whether the model L is nil. If $L = []$, then c points to the unit value. If $L \neq []$, then there exists a focused cell xs that forms a

circular fragment of model L. Moreover, the other halves of the pointed-by-thread and -heap assertions of this focused cell are stored.

The point of interest of these proofs is the logical deallocation. To logically deallocate a list c of model L, we first logically deallocate the reference c. If the list is empty, the proof is finished. Otherwise, if the list is non-empty, we have to logically deallocate a cycle. By induction over L, we construct a "could" assertion (§6.6). Then, because the cloud has no other entry point than the reference c, we use CloudFree and recover space credits and the empty pointed-by-thread and -heap assertions of the stored values.

Concurrent Case Studies

Schubert, F. (1824).
String Quartet No. 14 "Death and the Maiden".

We showcase the features of IrisFit for concurrency via a series of representative case studies. We first present *logically atomic triples* [da Rocha Pinto et al., 2014; Jung et al., 2015], a standard way of specifying operations on concurrent data structures (§12.1), and explain how to combine them with souvenirs. We begin our case studies with an encoding of the fetch-and-add operation in LambdaFit; the encoding considered makes use of protected sections (§12.2). Then, we present an implementation of a concurrent counter object, implemented as a pair of closures that share an internal reference (§12.3). We continue with a library for async/finish parallelism, which we encode on top of our implementation of fetch-and-add (§12.4). Last but not least, we present two lock-free data structures. First, we present our version of Treiber's stack (§12.5), which exploits protected sections, along the lines sketched earlier (§3). Second, our version of Michael and Scott's queue (§12.6), a concurrent lock-free queue [Michael and Scott, 1996] which we equip with protected sections.

12.1 Atomic Triples

Our specifications for fetch-and-add (§12.2), for Treiber's stack (§12.5) and for Michael and Scott's queue (§12.6) involve *logically atomic triples*, also known simply as *atomic triples* [da Rocha Pinto et al., 2014; Jung et al., 2015]. In our work, an atomic triple takes the form:

$$[R] \left\langle \frac{\Phi_{private}}{\forall \vec{x}.\ \Phi_{public}} \right\rangle \pi \colon t \left\langle \frac{\lambda v.\ \Phi'_{private}}{\Phi'_{public}} \right\rangle$$

The parameter R between square brackets is a souvenir (§10). We construct our atomic triples on top of our triples with souvenir in the same way that atomic triples are usually constructed on top of ordinary triples. Intuitively, atomic triples with a souvenir, written $[R]$, are atomic triples whose private pre- and postconditions are extended with pointed-by-thread assertions covering R (that is, $M \Lleftarrow \{\pi\}$ with $R = dom M$) and with the assertion *outside* π.

The private precondition $\Phi_{private}$ and the private postcondition $\lambda v.\ \Phi'_{private}$ play the same role as the precondition and postcondition of a standard triple. The private precondition is given up by thread π when the execution of the term t begins; the private postcondition is gained by thread π when the execution of the term t ends. They are *private* in the sense that they are invisible to other threads.

The characteristic feature of atomic triples is the presence of a public precondition Φ_{public} and of a public postcondition Φ'_{public}. An atomic triple guarantees that the public precondition Φ_{public} continuously holds until a certain point in time, the *linearization point* [Herlihy and Wing, 1990], where it is atomically transformed into the public postcondition Φ'_{public} [Birkedal et al., 2021]. Technically, an atomic triple involves a quantification over a list of variables $\vec{x}$, which scopes over Φ_{public}, $\Phi'_{private}$, and Φ'_{public}. The existentially quantified public precondition $\exists \vec{x}.\ \Phi_{public}$ continuously holds until the linearization point is reached. There, a specific instantiation of the variables $\vec{x}$ becomes fixed. For this specific choice of $\vec{x}$, the public precondition is transformed into the public postcondition Φ'_{public}, and the value v that is eventually returned satisfies $\Phi'_{private}$.

Interaction of Atomic Triples with Invariants Atomic triples can interact with invariants. Let us recall how invariants are used without atomic triples, expose the problem atomic triples address, and how we are going to write our specifications.

An invariant (§5.10) is a persistent assertion written $\boxed{\Phi}$ whose meaning is that the assertion Φ in the rectangular box holds at all times. Such an invariant allows in particular to share Φ among threads. Indeed, the protected resource Φ can be temporarily accessed while reasoning on *atomic* expressions—that is, expressions that evaluate to a value in a single step of computation, like a load, a store, or a CAS, for example. By "temporarily accessed", we mean that the user can *open* an invariant before reasoning on an atomic expression, adding the assertion $\triangleright \Phi$ to the context. The assertion extracted from an invariant is guarded by a *later* $\triangleright$ modality, which avoids paradoxes with high-order ghost state [Jung et al., 2018b, §5.5]. The user then has the obligation to *close* the invariant after reasoning on the atomic expression, by giving back the exact same assertion $\triangleright \Phi$ (an assertion entailed by Φ), hence restoring the invariant.

One limitation of the API of invariants we presented above is that an invariant can be accessed only while reasoning on an atomic step of computation. Hence, a Separation Logic triple specifying a concurrent method with several steps of computation is not satisfactory, as the user will not be able to use invariants to store shared resources. Atomic triples come to save the day. Indeed, an atomic triple makes the promise that the public precondition is updated into the public postcondition *atomically*. At the Iris level, this promise means that the user can open an invariant in the public precondition and has the obligation to close it in the atomic postcondition, *as if* the expression was atomic.

The later modality that appears while opening an invariant is tedious to eliminate in practice. Indeed, the user can eliminate a later modality only by matching up with an actual computation step. Thanks to later credits (§6.2), one may use an earlier computation step to "pay" for the elimination of one later modality. Our async/finish library (§12.4) presents a use case. Yet, later modalities are difficult to deal with, and pollute specifications. Thankfully, later modalities can often *be removed for free*. Indeed, there exists a large class of *timeless* assertions [Jung et al., 2018b, §5.7] for which we have $\triangleright \Phi \Rrightarrow \Phi$. Intuitively, these are the assertions that do not depend on the so-called step-index. Most of the "ground" assertions: the points-to, the pointed-by-thread, the pointed-by-heap, the deallocation witness, and space credits are timeless. Timelessness is preserved by most of the connectives, except for the later modality, invariants, various fancy updates, and triples themselves. Because working with timeless assertions is convenient, we design the assertions that are meant to be shared within invariants (for example, representation predicates of data structures) to be timeless assertions.

12.2 Fetch-and-Add

The "fetch-and-add" (FAA) operation atomically increments the content of an integer reference and returns the previous content of the reference. Although this operation is commonly provided in hardware, implementing it in LambdaFit is a fairly instructive exercise. Indeed, this code and its proof offer a typical example of the use of protected sections.

Code In our setting, FAA takes three parameters: an address l, an offset i, and the desired increment n, an integer value. We encode FAA as a tail-recursive function whose body contains a CAS instruction enclosed in a protected section. The code is shown in Figure 44. The recursive function is named f; its parameters are l, i and n. Initially, the content of the memory at address l and offset i is loaded into the variable m. Then, a protected section is entered, and a CAS instruction attempts to update the content of the memory from m to $m + n$. In case of success, the protected section is exited and the value m is returned. In case of failure, the protected section is also exited, and a recursive call is performed, so as to try again.

$$\begin{aligned}
&\mathit{faa} \triangleq \mu_{\mathsf{ptr}} f.\, \lambda[l, i, n].\\
&\quad \mathsf{let}\; m = l[i]\; \mathsf{in}\\
&\quad \mathsf{enter}\,;\, \mathsf{if}\; \mathsf{CAS}\; l[i]\; m\; (m+n)\\
&\quad \mathsf{then}\; (\mathsf{exit}\,;\, m)\\
&\quad \mathsf{else}\; (\mathsf{exit}\,;\, (f\; [l, i, n])_{\mathsf{ptr}})
\end{aligned}$$

FAA

$$[\emptyset] \left\langle \frac{\ell \Leftarrow_p \{\pi\}}{\forall \vec{v}\, m.\ \ulcorner \vec{v}(i) = m \urcorner \ * \ \ell \mapsto \vec{v}} \right\rangle \pi \colon (\mathit{faa}\ [\ell, i, n])_{\mathsf{ptr}} \left\langle \frac{\lambda m'. \qquad \ulcorner m' = m \urcorner}{\ell \mapsto ([i := (m+n)]\vec{v}) \ * \ \ell \Leftarrow_p \emptyset} \right\rangle$$

Figure 44: Code and specification of fetch-and-add

Thanks to the protected section, as soon as the CAS instruction succeeds, the memory location l can be considered as a temporary root, as opposed to an ordinary root. Indeed, as soon as CAS succeeds, it is known that the first branch of the conditional construct will be taken, so the protected section will be exited via the first exit instruction, where l is no longer a root.

Consider the equivalent body of FAA without protected sections whose last instructions are $\mathsf{if}\; \mathsf{CAS}\; l[i]\; m\; (m+n)\; \mathsf{then}\; m\; \mathsf{else}\; (f\; [l, i, n])_{\mathsf{ptr}}$. If the CAS succeeds, the expression reduces to $\mathsf{if}\; \mathsf{true}\; \mathsf{then}\; m\; \mathsf{else}\; (f\; [l, i, n])_{\mathsf{ptr}}$. However, in this expression, l would still be considered a root (that is to say, an ordinary root), because it occurs inside the "else" branch, and according to the FVR (§2.1), every location that occurs in the code that lies ahead is a root. This is an issue in the case where another thread reads the just-committed value stored in l, decide it does not need l anymore, and would like to reuse its space. We present such a scenario in our async/finish case study (§12.4).

Specification Our specification of FAA appears at the bottom of Figure 44. The private precondition consumes a pointed-by-thread assertion for the location ℓ, carrying some fraction p and the current thread identifier π. The public precondition requires that ℓ points to a block $\vec{v}$ and that the value stored at offset i in this block be m. The private postcondition asserts that the result of FAA is m. The public postcondition asserts that FAA atomically updates m into $m + n$. Crucially, it also produces an updated pointed-by-thread assertion for ℓ, carrying the same fraction p and an *empty* set of thread identifiers. This postcondition means that as soon as the linearization point is reached, ℓ is not a root in the thread π any more. Capturing this property will reveal crucial for reasoning about our async/finish library (§12.4).

Proof Insights We next explain how we use the reasoning rules of protected sections (Figure 19) for verifying that FAA obeys its specification. Upon entering the protected section, we use ENTER and transform the assertion $\mathit{outside}\,\pi$ into the assertion $\mathit{inside}\,\pi\,\emptyset$. Then, we face the CAS instruction, a possible linearization point. We open the public precondition, and gain the points-to assertion for ℓ. By case analysis on the value that is currently stored at address l and offset i, we consider the case where CAS succeeds and the case where it fails. We do so before reasoning on the if statement. Let us focus on the case where it succeeds. We use CASSUCCESS, which updates the points-to assertion, and effectively execute the linearization point. At this point, the atomic triple requires us to prove that the public postcondition holds. Using ADDTEMPORARY, we make ℓ a temporary root: applying this rule changes the assertions $\ell \Leftarrow_p \{\pi\}$ and $\mathit{inside}\,\pi\,\emptyset$ into $\ell \Leftarrow_p \emptyset$ and $\mathit{inside}\,\pi\,\{\ell\}$. By giving up the points-to and pointed-by-thread assertions, we fulfill the public postcondition. Then, we use IFTRUE and enter the first branch of the "if" statement. There, TRIMINSIDE lets us

$$
\begin{array}{ll}
\begin{array}{l}
\mathit{ref} \triangleq \mu_{\mathsf{ptr}_}.\,\lambda[x].\\
\quad \mathsf{let}\ r = \mathsf{alloc}\ 1\ \mathsf{in}\\
\quad r[0] \leftarrow x\,;\ r\\
\mathit{pair} \triangleq \mu_{\mathsf{ptr}_}.\,\lambda[x,y].\\
\quad \mathsf{let}\ r = \mathsf{alloc}\ 2\ \mathsf{in}\\
\quad r[0] \leftarrow x\,;\ r[1] \leftarrow y\,;\ r
\end{array}
&
\begin{array}{l}
\mathit{ignore} \triangleq \mu_{\mathsf{ptr}_}.\,\lambda[x].\,()\\
\mathit{create} \triangleq \mu_{\mathsf{ptr}_}.\,\lambda[].\\
\quad \mathsf{let}\ r = (\mathit{ref}\ [0])_{\mathsf{ptr}}\ \mathsf{in}\\
\quad \mathsf{let}\ i = \mu_{\mathsf{clo}_}.\,\lambda_.\,(\mathit{ignore}\ [(\mathit{faa}\ [r,0,1])_{\mathsf{ptr}}])_{\mathsf{ptr}}\ \mathsf{in}\\
\quad \mathsf{let}\ g = \mu_{\mathsf{clo}_}.\,\lambda_.\,r[0]\ \mathsf{in}\\
\quad (\mathit{pair}\ [i,g])_{\mathsf{ptr}}
\end{array}
\end{array}
$$

$$(\mathit{counter}\ i\ g\ (p_1+p_2)\ (n_1+n_2)) \quad \equiv \quad (\mathit{counter}\ i\ g\ p_1\ n_1 \ * \ \mathit{counter}\ i\ g\ p_2\ n_2)$$

$$[\emptyset]\ \{\Diamond 7\}\ \pi \colon (\mathit{create}\ [])_{\mathsf{ptr}}\ \left\{\lambda\ell.\ \exists i\,g.\ \begin{array}{c} \ell \mapsto [i;g] \ * \ \mathit{counter}\ i\ g\ 1\ 0\\ \ell \Lleftarrow \{\pi\} \ * \ \ell \hookleftarrow \emptyset\\ i \Lleftarrow \emptyset \ * \ i \hookleftarrow \{+\ell\}\\ g \Lleftarrow \emptyset \ * \ g \hookleftarrow \{+\ell\} \end{array}\right\}$$

$$[\emptyset]\ \{\mathit{counter}\ i\ g\ p\ n\} \quad \pi \colon (i\ [])_{\mathsf{clo}} \quad \{\lambda().\ \mathit{counter}\ i\ g\ p\ (n+1)\}$$

$$[\emptyset]\ \{\mathit{counter}\ i\ g\ p\ n\} \quad \pi \colon (g\ [])_{\mathsf{clo}} \quad \left\{\lambda m.\ \begin{array}{c} \ulcorner n \le m \ \land\ (p = 1 \implies n = m) \urcorner\\ \mathit{counter}\ i\ g\ p\ n \end{array}\right\}$$

$$\begin{pmatrix} \mathit{counter}\ i\ g\ 1\ n\\ i \Lleftarrow \emptyset \ * \ i \hookleftarrow \emptyset\\ g \Lleftarrow \emptyset \ * \ g \hookleftarrow \emptyset \end{pmatrix} \quad \Rrightarrow \quad (\Diamond 5)$$

Figure 45: Code and specification of a concurrent monotonic counter

change the assertion $\mathit{inside}\ \pi\ \{\ell\}$ to $\mathit{inside}\ \pi\ \emptyset$. This allows us to exit the protected section using Exit. We finish the proof with Val.

12.3 A Concurrent Counter Object

Our next example is a concurrent monotonic "counter" object, whose internal state is stored in a mutable reference, and whose access is mediated by a pair of closures: a closure i which *increments* the counter; a closure g which *gets* its current value. This is an example of a procedural abstraction [Reynolds, 1975], also known as an *object*: indeed, "an object is a value exporting a procedural interface to data or behavior" [Cook, 2009]. Crucially, a counter can be used concurrently by several threads.

Code The top of Figure 45 presents the code that we verify. The function call $(\mathit{ref}\ [x])_{\mathsf{ptr}}$ allocates a mutable reference, that is, a block of size 1. The function call $(\mathit{pair}\ [x,y])_{\mathsf{ptr}}$ allocates a mutable pair, that is, a block of size 2. The function call $(\mathit{ignore}\ [x])_{\mathsf{ptr}}$ ignores its argument and returns the unit value. The function call $(\mathit{create}\ [])_{\mathsf{ptr}}$ returns a fresh "counter", that is, a pair of two closures i and g. Both closures point to an internal reference r, which is initialized to the value 0. The closure i uses our fetch-and-add function (§12.2) and ignores its result.

Specifications Figure 45 presents the specification of our concurrent counter. It is inspired by a specification that appears in lecture notes [Birkedal and Bizjak, 2023, §8.7]. It relies on an abstract assertion $\mathit{counter}\ i\ g\ p\ n$ where i is the location of the "increment" closure, g is the location of the "get" closure, $p \in (0;1]$ is a fraction that represents a *share* of the ownership of the counter, and n, a natural number, represents a *past contribution* to the current value of the counter. If p is 1 then the contribution n is in fact the current value of the counter.

The equivalence rule in Figure 45 shows that "*counter*" assertions can be split and joined; both the fraction and the contribution are then split or joined by addition. This allows a counter to be used in a concurrent setting: the user can split the "*counter*" predicate into

$$\begin{array}{rcl}
\mathit{counterInv}\,\gamma\,\ell & \triangleq & \boxed{\exists m.\ \ell \mapsto [m] \ * \ \boxed{\bullet\,(1,m)}^{\gamma}} \\
\mathit{share}\,\gamma\,p\,n & \triangleq & \boxed{\circ\,(p,n)}^{\gamma} \\
\mathit{incrspec}\,\gamma\,\ell & \triangleq & \lambda__t.\ \forall \pi\,p\,n. \\
\multicolumn{3}{l}{\quad [\{\ell\}]\,\{\mathit{counterInv}\,\gamma\,\ell \ * \ \mathit{share}\,\gamma\,p\,n\}\,\pi\!:t\,\{\lambda().\ \mathit{share}\,\gamma\,p\,(n+1)\}} \\
\mathit{getspec}\,\gamma\,\ell & \triangleq & \lambda__t.\ \forall \pi\,p\,n. \\
\multicolumn{3}{l}{\quad [\{\ell\}]\,\{\mathit{counterInv}\,\gamma\,\ell \ * \ \mathit{share}\,\gamma\,p\,n\}\,\pi\!:t\,\{\lambda m.\ \ulcorner n \leq m \ \wedge\ (p = 1 \implies n = m)\urcorner \ * \ \mathit{share}\,\gamma\,p\,n\}} \\
\mathit{counter}\,p\,n\,i\,g & \triangleq & \exists \gamma\,\ell. \\
\multicolumn{3}{l}{\quad \mathit{meta}\,i\,\gamma \ * \ \mathit{counterInv}\,\gamma\,\ell \ * \ \mathit{share}\,\gamma\,p\,n \ * \ \ell \Lleftarrow_p \emptyset \ *} \\
\multicolumn{3}{l}{\quad \mathit{Spec}\,0\,[(\ell,1/2)]\,(\mathit{incrspec}\,\gamma\,\ell)\,i \ * \ \mathit{Spec}\,0\,[(\ell,1/2)]\,(\mathit{getspec}\,\gamma\,\ell)\,g}
\end{array}$$

Figure 46: Internals of the concurrent counter

several parts and give a part to each participating thread. In the end, the user can gather all the parts, draw conclusions about the final value of the counter, and logically deallocate it.

The specification of $(\mathit{create}\ [])_{\mathsf{ptr}}$ states that this call consumes 7 space credits (1 credit for the shared reference, 2 credits for each closure, and 2 credits for the pair). It returns a pair ℓ of two locations i and g such that $\mathit{counter}\ i\ g\ 1\ 0$ holds. This assertion captures the full ownership of the counter, and specifies that its current value is 0.

Figure 45 also shows the specifications of calls to i and g. Both calls require an assertion of the form $\mathit{counter}\ i\ g\ p\ n$. The postcondition of a call to the "increment" closure contains an updated assertion $\mathit{counter}\ i\ g\ p\ (n{+}1)$. The postcondition of a call to the "get" closure contains an unmodified "*counter*" assertion. Furthermore, it guarantees that the natural number m that is returned by this call is no less than the past contribution n and, in the case where p is 1, is equal to the past contribution.

Last, Figure 45 shows the reasoning rule for deallocating a counter. This rule requires full ownership of the counter as well as pointed-by-heap and pointed-by-thread assertions for the closures i and g, with fraction 1 and empty sets—this witnesses that both closures are unreachable. In exchange, the rule produces 5 spaces credits. The 2 credits corresponding to the pair produced by *create* can be recovered independently.

Proof Insights Figure 46 presents the internals of our counter, based on standard ghost state [Birkedal and Bizjak, 2023, §8.7] and on our *Spec* predicates (§9.5). The assertion $\mathit{counterInv}\,\gamma\,\ell$ is an invariant describing the content of the shared reference ℓ, which must point to a natural number m. This number m is authoritatively registered with fraction 1 inside the ghost cell γ, which is equipped with the resource algebra $\mathit{Auth}(\mathit{Frac}(\mathbb{N}))$, the authoritative fractional resource algebra of natural numbers, with the addition as binary operation.

The assertion $\mathit{share}\,\gamma\,p\,n$ represents a share n with fraction p of the natural number stored inside the shared reference ℓ. The share is materialized by the fragmentary ownership of the pair (p,n) in the ghost cell γ. Thanks to these definitions, we have the following two rules:

$$\begin{array}{rcl}
\mathit{share}\,\gamma\,(p_1+p_2)\,(n_1+n_2) & \equiv & \mathit{share}\,\gamma\,p_1\,n_1 \ * \ \mathit{share}\,\gamma\,p_2\,n_2 \\
\boxed{\bullet\,(1,m)}^{\gamma} \ * \ \mathit{share}\,\gamma\,p\,n & -\!\!* & \ulcorner n \leq m \ \wedge\ (p = 1 \implies n = m)\urcorner
\end{array}$$

The first of these rules allows for splitting and joining a share, and the second allows for deducing the postcondition of the get closure. Then we present the two specifications of the increment and get closures, namely the predicates $\mathit{incrspec}\,\gamma\,\ell$ and $\mathit{getspec}\,\gamma\,\ell$. These two "specifications" are meant to be plugged inside a *Spec* predicate, and are hence functions. Here, these functions ignore their two firsts parameters: the first parameter, the location of the closure, is not needed as the two closures are not recursive, and the second parameter, the list of actual arguments of the closure, is not needed as the two functions take no arguments.

The third argument t represents the body of the closure. The two specifications are then a triple with souvenir over t.

The last piece of the puzzle is the definition of $counter\ i\ g\ p\ n$. This definition existentially quantifies over the name of the ghost cell γ and the location of the shared reference ℓ. The definition makes use of a meta assertion $meta\ i\ \gamma$ to permanently tie the ghost cell γ to the location of the increment closure i. The definition then asserts that ℓ is a valid logical counter via the invariant $counterInv\ \gamma\ \ell$. The definition also asserts the ownership of a share n with fraction p, and stores an empty pointed-by-thread assertion for ℓ with fraction p. This latter assertion $\ell \Leftarrow\!\!\!|_p\ \emptyset$ allows satisfying the side condition of the reasoning rule for closure call (CallSpec), which requires updating pointed-by-thread assertions of locations from the closure's environment. The pointed-by-thread assertion of ℓ is trimmed by the time the closure call end, in order to restore the assertion *share*.

Moreover, the assertion $counter\ i\ g\ p\ n$ contains *Spec* assertions for the two closures i and g. The first argument of these *Spec* assertions is the arity of the closure they describe; in both cases, this arity is 0. The second argument is the environment of the closures. Each one maps the shared location ℓ to the fraction $\frac{1}{2}$, which means that each closure owns one half of the pointed-by-heap assertion for the location ℓ. The third argument is the actual specification of the closure, and the last the location of the closure itself.

12.4 An Async/Finish Library

The async/finish paradigm was introduced in X10 [Charles et al., 2005], as a generalization of the spawn/sync mechanism of Cilk [Blumofe et al., 1995], spawn/sync itself being a generalization of the binary fork/join paradigm. The async/finish paradigm allows spawning an arbitrary number of tasks before waiting at a common join point. More precisely, the construct "*async*" allows spawning new tasks, whereas "*finish*" performs synchronization: it blocks until all previously spawned tasks terminate. In this section, we show how to encode these two constructs in LambdaFit using a shared mutable reference that is updated using a fetch-and-add operation (§12.2). We then provide specifications in IrisFit, and show that the space credits associated to the shared reference can be recovered as soon as "*finish*" returns.[1] A strength of our specification is that it allows for *nested* spawns: a spawned task can itself spawn tasks.

Code The code of our async/finish library is presented in the top part of Figure 47. The library uses a reference that we call the *session*. A session is a channel through which tasks communicate. It stores the number of currently running tasks.

The function $(create\ [])_{\mathsf{ptr}}$ returns a fresh session, with zero running tasks.

The function $(async\ [l, f])_{\mathsf{ptr}}$ expects a session l and a closure f as arguments. It first atomically increments the session, hence recording the existence of a new running task, then forks off a thread that invokes the closure f with no arguments. When this invocation terminates, it atomically decrements the session, thereby recording that this task is finished.

The function $(finish\ [l])_{\mathsf{ptr}}$ consists of an active waiting loop. This loop ends when it observes that the session contains the value 0, which guarantees that all previously spawned tasks have terminated.

Specifications We present the specification of our async/finish library at the bottom of Figure 47.

According to AFCreate, $(create\ [])_{\mathsf{ptr}}$ consumes one space credit, which corresponds to the space occupied by the session, and returns a location ℓ such that $AF\ \ell$ holds. This

[1] That is to say, as soon as every task reaches the linearization point of the fetch-and-add operation to signal that it is done. A task may still execute instructions past the linearization point before actually terminating.

$$\begin{array}{l} \mathit{create} \triangleq \mu_{\mathsf{ptr}}_.\ \lambda[].\\ \quad (\mathit{ref}\ [0])_{\mathsf{ptr}}\\ \mathit{async} \triangleq \mu_{\mathsf{ptr}}_.\ \lambda[l,f].\\ \quad (\mathit{faa}\ [l,0,1])_{\mathsf{ptr}}\,;\\ \quad \mathsf{fork}\,((f\ [])_{\mathsf{clo}}\,;\,(\mathit{ignore}\ [(\mathit{faa}\ [l,0,-1])_{\mathsf{ptr}}])_{\mathsf{ptr}}) \end{array}$$

$$\begin{array}{l} \mathit{finish} \triangleq \mu_{\mathsf{ptr}} f.\ \lambda[l].\\ \quad \mathsf{if}\ l[0]=0\\ \quad \mathsf{then}\ ()\\ \quad \mathsf{else}\ (f\ [l])_{\mathsf{ptr}} \end{array}$$

AFCreate
$$[\emptyset]\{\Diamond 1\}\ \pi\colon (\mathit{create}\ [])_{\mathsf{ptr}}\ \{\lambda\ell.\ \mathit{AF}\,\ell \ * \ \ell \Leftarrow_{\frac{1}{2}} \{\pi\} \ * \ \ell \hookleftarrow_1 \emptyset\}$$

AFAsync
$$\frac{\forall \pi'.\quad [\{\ell\}]\{f \Leftarrow_p \{\pi'\} \ * \ \Phi\}\ \pi'\colon (f\ [])_{\mathsf{clo}}\ \{\lambda().\ \Psi\}}{[\{\ell\}]\{\mathit{AF}\,\ell \ * \ f \Leftarrow_p \{\pi\} \ * \ \Phi\}\ \pi\colon (\mathit{async}\ [\ell,f])_{\mathsf{ptr}}\ \{\lambda().\ \mathit{spawned}\,\ell\,\Psi\}}$$

AFFinish
$$[\emptyset]\{\mathit{AF}\,\ell \ * \ \ell \Leftarrow_{\frac{1}{2}} \{\pi\}\}\ \pi\colon (\mathit{finish}\ [\ell])_{\mathsf{ptr}}\ \{\lambda().\ \mathit{finished}\,\ell\}$$

FinishedSpawned
$$\mathit{finished}\,\ell \ * \ \mathit{spawned}\,\ell\,\Psi \quad \Rrightarrow \quad \Psi$$

FinishedFree
$$\mathit{finished}\,\ell \ * \ \ell \hookleftarrow_1 \emptyset \quad \Rrightarrow \quad \Diamond 1$$

AFPersistent
$\mathit{AF}\,\ell$ is persistent

FinishedPersistent
$\mathit{finished}\,\ell$ is persistent

Figure 47: Code and specification of an async/finish library

persistent assertion guarantees that ℓ is a session. The postcondition also provides pointed-by-thread and pointed-by-heap assertions for the location ℓ. The pointed-by-heap assertion carries the fraction $\frac{1}{2}$; the other half is hidden from the user.

The specification of $(\mathit{async}\ [\ell,f])_{\mathsf{ptr}}$ is stated as a triple featuring a souvenir on ℓ. This means that, for the duration of this call, ℓ is a root. The precondition requires ℓ to be a session. A fractional pointed-by-thread assertion for the closure f, as well as an arbitrary assertion Φ, are consumed and transmitted to the new task, which invokes the closure f. The premise of the rule AFAsync requires the user to prove that, under an arbitrary thread identifier π', this invocation is safe and satisfies some postcondition Ψ. The postcondition of $(\mathit{async}\ [\ell,f])_{\mathsf{ptr}}$ provides a witness that this task was spawned, in the form of the assertion $\mathit{spawned}\,\ell\,\Psi$. This assertion is not persistent: it can be understood as a unique permission to collect Ψ once the task is finished.

The specification of f in the premise of AFAsync is again a triple with a souvenir of ℓ. This formulation allows f to itself use *async*. Using an ordinary triple there would place a stronger requirement on f and would forbid the use of *async* inside f.

According to AFFinish, $(\mathit{finish}\ [\ell])_{\mathsf{ptr}}$ consumes the pointed-by-thread assertion that was produced by *create*. This forbids any further use of the session ℓ: indeed, both AFAsync and AFFinish require a pointed-by-thread assertion for ℓ.[2] The postcondition contains the persistent assertion $\mathit{finished}\,\ell$, which witnesses that this session has been ended.

The ghost update FinishedSpawned states that if the witness $\mathit{finished}\,\ell$ is at hand then the assertion $\mathit{spawned}\,\ell\,\Psi$ can be converted to Ψ. This reflects the idea that if the session associated with ℓ has been ended, then all of its tasks must have terminated: so, a permission to collect Ψ can indeed be converted to Ψ. The ghost update FinishedFree states that if the session has ended then abandoning the pointed-by-heap assertion for ℓ allows recovering the space credit associated with the session ℓ.

[2] In the case of AFAsync, this is implicit in the fact that the conclusion of the rule is a triple with a souvenir on ℓ.

Proof Insights The assertion $AF\,\ell$ is internally defined as an Iris invariant. Among other things, this invariant imposes a protocol on the pointed-by-thread assertion for the session ℓ. Initially, the invariant contains a pointed-by-thread assertion carrying the fraction $\frac{1}{2}$ and an empty set; the other half is given to the user by AFCREATE. Each spawned task gets a fraction of this assertion: indeed, spawning a task involves "fork", and our FORK rule requires updating a pointed-by-thread assertion so as to reflect the fact that ℓ is a root of the new thread. When a task signals that it is finished, it surrenders its fractional pointed-by-thread assertion, carrying an *empty* set of thread identifiers. Hence, once every task has terminated, the invariant again contains $\ell \Leftarrow\!\!\dashv_{\frac{1}{2}} \emptyset$.

How and when exactly does a task signal that it is finished? This is done via a fetch-and-add (FAA) operation, which decrements the count of active tasks, and takes effect precisely at the linearization point of this FAA operation. Hence, as soon as this linearization point is reached, the invariant requires this task to surrender its fractional pointed-by-thread assertion. Fortunately, our specification of FAA (§12.2) allows this: the pointed-by-thread assertion $\ell \Leftarrow\!\!\dashv_p \emptyset$ appears in the public postcondition in FAA.

The absence of a "later" modality in front of Ψ in FINISHEDSPAWNED may seem surprising to an expert reader. Indeed, because the assertion Ψ has transited through an invariant, one might expect it to be guarded by such a modality. The usual way to eliminate a "later" modality is through a physical step, yet this rule is a ghost update. Fortunately, IrisFit supports and takes advantage of *later credits* (§6.2). A later credit is a piece of ghost state that is produced by a physical step and that can later be used to eliminate a "later" modality. With each spawned task, we are able to internally associate one later credit, which we obtain from the function call $(async\ [\ell, f])_{\mathsf{ptr}}$. By exploiting this later credit, we can eliminate the "later" modality in front of Ψ before giving this assertion back to the user.

12.5 Treiber's Stack

Code The code that we verify is the code of Figure 3, translated to LambdaFit syntax. Recall that Treiber's stack consists of a reference on an immutable list. In LambdaFit, we encode a reference with a block of size 1. As for sequential lists (§11.2), an empty list is represented by (), the unit value. A list cell is a block of size 2 whose first offset stores the content of the cell, and the second offset stores the successor of the list.

For simplicity, we specify and verify only `create`, `push`, `pop` and logical deallocation. The functions `push`, `pop` are encoded as a "CAS loop", looping if contention occurs. However, it is possible to distinguish separated functions `try_push` and `try_pop`: each one tries to commit the corresponding operation, and returns an error code if contention occurs rather than looping. More precisely, `try_pop` returns either the popped value or an error code inside a heap-allocated sum. The specification of `try_pop` would hence require additional space credits accounting for the space required by the sum.

Specifications Figure 48 presents our specification of Treiber's stack. The stack is described in terms of the representation predicate $stack\,\ell\,L$, where ℓ is the location of the stack and L is its mathematical model. This model is a list of triples (v, p, q) of a value v and two positive fractions p and q, following our recipe for containers (§11.1).

The assertion $stack\,\ell\,L$ is not fractional: it represents the full ownership of the stack. STACKTIMELESS witnesses that $stack\,\ell\,L$ is a timeless assertion—that is, it can be extracted from an invariant without a later modality. This timeless property is important, as the assertion $stack\,\ell\,L$ is meant to be shared among threads within an invariant.

According to STACKCREATE, creating a new stack consumes one space credit. This is the size of the reference that holds the address of the top list cell. The result is a fresh location ℓ that represents an empty stack.

StackCreate

$$[\emptyset]\{\Diamond 1\}\ \pi\colon (\mathit{create}\ [])_{\mathsf{ptr}}\ \{\lambda \ell.\ \mathit{stack}\,\ell\,[] \ * \ \ell \Leftarrow\!\!\dashv \{\pi\} \ * \ \ell \leftarrow\!\!\dashv \emptyset\}$$

StackPush

$$[\{\ell\}] \left\langle \frac{\Diamond 2 \ * \ v \Leftarrow\!\!\dashv_p \{\pi\} \ * \ v \leftarrow\!\!\dashv^{\geqslant 0}_q \emptyset}{\forall L. \qquad \mathit{stack}\,\ell\,L} \right\rangle \pi\colon (\mathit{push}\ [\ell; v])_{\mathsf{ptr}} \left\langle \frac{\lambda (). \qquad \ulcorner \mathrm{True} \urcorner}{\mathit{stack}\,\ell\,((v,p,q) :: L)} \right\rangle$$

StackPop

$$[\{\ell\}] \left\langle \frac{\ulcorner \mathrm{True} \urcorner}{\forall v\,p\,q\,L.\ \ \mathit{stack}\,\ell\,((v,p,q) :: L)} \right\rangle \pi\colon (\mathit{pop}\ [\ell])_{\mathsf{ptr}} \left\langle \frac{\lambda w. \quad \ulcorner w = v \urcorner \ * \ v \Leftarrow\!\!\dashv_p \{\pi\}}{\mathit{stack}\,\ell\,L \ * \ \Diamond 2 \ * \ v \leftarrow\!\!\dashv^{\geqslant 0}_q \emptyset} \right\rangle$$

StackFree

$$\mathit{stack}\,\ell\,L \ * \ \ell \Leftarrow\!\!\dashv \emptyset \ * \ \ell \leftarrow\!\!\dashv \emptyset \quad \Rrightarrow \quad \Diamond(1 + 2 \times |L|) \ * \ \mathop{\Large *}_{(v,p,q)\in L} (v \Leftarrow\!\!\dashv_p \emptyset \ * \ v \leftarrow\!\!\dashv^{\geqslant 0}_q \emptyset)$$

StackTimeless

$\mathit{stack}\,\ell\,L$ is timeless

Figure 48: Specification of Treiber's stack

The specification of $(\mathit{push}\ [\ell; v])_{\mathsf{ptr}}$, expressed by StackPush, is an atomic triple with a souvenir on ℓ. The private precondition requires two space credits, which is the size of a new list cell, as well as fractional pointed-by-heap and pointed-by-thread assertions for the value v that is pushed onto the stack. Together, the public pre- and postconditions indicate that the model of the stack is atomically updated from L to $(v, p, q) :: L$ at the linearization point.

The specification of $(\mathit{pop}\ [\ell])_{\mathsf{ptr}}$, expressed by StackPop, is also an atomic triple with a souvenir on ℓ. The public pre- and postconditions indicate that the model of the stack is atomically updated from $(v, p, q) :: L$ to L. Furthermore, according to the public postcondition, two space credits are produced, as well as a pointed-by-heap assertion for v carrying an empty multiset of predecessors, is produced, as a pointer from the stack to v has been destroyed.

Our specification of "*pop*" exhibits a certain asymmetry: whereas the space credits and the pointed-by-heap assertion appear in the *public* postcondition, which means that they are produced at the linearization point, the pointed-by-thread assertion appears in the *private* postcondition. which means that it is produced when the function returns. The space credits and the pointed-by-heap assertion can be produced because, as soon as the linearization point occurs, we are able to logically deallocate the list cell and to argue that a pointer from the stack to v has been destroyed. However, the pointed-by-thread assertion cannot be surrendered as part of the public postcondition, because the value v is read from the heap *after* the linearization point has been passed.

Last, StackFree logically deallocates a possibly nonempty stack. The assertion $\mathit{stack}\,\ell\,L$, as well as empty pointed-by-thread and pointed-by-heap assertions for ℓ, are consumed. A number of space credits are produced, which reflects the overall size occupied by the stack data structure in the heap: one credit for the toplevel reference, plus two credits per list cell. The pointed-by-thread and pointed-by-heap assertions associated with every triple (v, p, q) in the stack are also produced. In the common case where L is an empty list, this rule can be significantly simplified.

We show that the specifications presented in Figure 48 entails the specifications for sequential stacks presented earlier (§11.4), with the parameters $A = 1$, $B = 2$, and $C = +\infty$. The instantiation demonstrates that our concurrent specification refines our sequential specification. In short, we prove that Treiber's concurrent lock-free stack can be used sequentially.

Proof Insights As argued earlier (§3), the main difficulty of the proof is to produce space credits when a "*pop*" operation succeeds. Producing these credits requires logically deallo-

cating the list cell, name it ℓ, that is being extracted. This logical deallocation requires exhibiting both an empty pointed-by-thread assertion $\ell \Lleftarrow \emptyset$ and an empty pointed-by-heap assertion $\ell \hookleftarrow \emptyset$ for this cell. Yet, neither of these assertions is easy to obtain. Indeed, internal list cells may be roots of ongoing concurrent "*push*" or "*pop*" operations, compromising the availability of an empty pointed-by-thread. Moreover, an internal list cell may be pointed by an internal block of an ongoing "*push*", compromising the availability of an empty pointed-by-heap.

Let us discuss the pointed-by-thread assertion $\ell \Lleftarrow \emptyset$ first. The difficulty is that "*push*" and "*pop*" are *invisible readers* [Alistarh et al., 2018]: these operations read the top of the stack (that is, the address of a list cell) without synchronization. Such a read normally requires updating a pointed-by-thread assertion for the cell whose address is thus obtained. However, here, we do not wish to record that this cell is pointed to by the current thread. Fortunately, these reads occur inside protected sections. Hence, we use LoadInside, which updates an "*inside*" assertion instead of a pointed-by-thread assertion. This allows the stack's invariant to keep an *empty* pointed-by-thread assertion, at all times, for every list cell. Such an invariant allows a successful "*pop*" operation to extract this empty pointed-by-thread assertion out of the invariant. Maintaining empty pointed-by-thread assertions for locations that are acquired only inside protected sections is a typical idiom.

Next, let us discuss the pointed-by-heap $\ell \hookleftarrow \emptyset$ assertion. Here, the difficulty is that a list cell ℓ may be pointed to by a new cell ℓ' that has just been allocated by an ongoing "*push*" operation. This scenario was discussed earlier (§3.2). Hence, each ongoing "*push*" needs to register the list cell ℓ' this thread constructs as a predecessor of the internal list cell ℓ the location ℓ' points to within the pointed-by-heap assertion of ℓ. Now, how can "*pop*" obtain the assertion $\ell \hookleftarrow_1 \emptyset$ that is required to allow logical deallocation? We answer this question via an original technique that we dub *logical deallocation by proxy*: the thread that successfully pops the list cell ℓ also takes care of logically deallocating the predecessor cells ℓ' that have been allocated by ongoing "*push*" operations.[3] This technique reminds of (physical) concurrent helping between threads [Herlihy and Shavit, 2012, §6.4], when a thread helps another thread to make progress.

The logical deallocation of the locations ℓ and ℓ' is made possible by the protected section in "*push*". This approach has a somewhat unexpected consequence: in the proof of "*push*", it may be the case that the cell ℓ' has been logically deallocated by another thread, yet "*push*" still needs to access this cell. Fortunately, IrisFit allows this: for example, the proof of "*push*" makes use of the rule StoreDead.

Proof Details The definition of *stack* $\ell\, L$ and its auxiliary assertions appears in Figure 49. The idea of our invariant is to offer a way for threads to register predecessors of internal list cells using ghost state. In this endeavor, the pointed-by-heap assertion of each list cell is split into two halves. One is kept inside the representation predicate of the list, registering the pointer from the potential predecessor cell. The other half is kept inside a dedicated predicate *cells* for threads to register predecessors. These predecessors are registered inside a ghost multiset G of pairs of locations (ℓ, ℓ'): the list cell ℓ being pointed-by the private block ℓ' of an ongoing "*push*". For each private block ℓ', there is a most one pair (ℓ, ℓ') in G.

In detail, our definitions in Figure 49 make use of two ghost cells. First, γ_1 is equipped with the resource algebra $\mathit{Auth}(\mathit{SetMono}(\mathcal{L}))$, the authoritative resource algebra over a monotonic (that is, ever-growing) set of locations. This ghost cell keeps track of the list cells of the stack. Its fragmentary elements are persistent. Second, γ_2 is equipped with the resource algebra $\mathit{Auth}(\mathit{Multiset}(\mathcal{L} \times \mathcal{L}))$, the authoritative resource algebra of a multiset of pairs of locations. This ghost cell keeps track of potential private locations of every ongoing "*push*" operation pointing to list cells.

[3] Note that these ongoing "*push*" operations will fail, because the top list cell that they have previously observed has been replaced.

$$
\begin{aligned}
\mathit{reg}\,\gamma_1 x &\triangleq \ulcorner x = () \urcorner \vee \boxed{\circ\,\{x\}}^{\gamma_1} \\
\mathit{innerList}\,\gamma_1\,x\,L &\triangleq \\
&\quad \ulcorner L = [] \;\wedge\; x = () \urcorner \\
&\vee\; \exists v\,p\,q\,x'\,L'.\, \ulcorner L = (v,p,q) :: L' \urcorner \;*\; x \mapsto_{\Box} [v; x'] \;* \\
&\quad v \hookleftarrow^{0}_{q} \{+x\} \;*\; v \Leftarrow_p \emptyset \;* \\
&\quad x' \Leftarrow_1 \emptyset \;*\; \mathit{reg}\,\gamma_1 x' \;*\; x' \hookleftarrow_{\frac{1}{2}} \{+x\} \;* \\
&\quad \mathit{innerList}\,\gamma_1\,x'\,L' \\
\mathit{inner}\,\gamma_1\,\ell\,L &\triangleq \exists x.\, \ell \mapsto [x] \;*\; x \Leftarrow_1 \emptyset \;*\; \mathit{reg}\,\gamma_1 x \;*\; x \hookleftarrow_{\frac{1}{2}} \{+\ell\} \;*\; \mathit{innerList}\,\gamma_1\,x\,L \\
\mathit{volatile}\,\ell'\,n\,\Phi &\triangleq (\mathit{sizeof}\,\ell'\,n \;*\; \ell' \Leftarrow_1 \emptyset \;*\; \ell' \hookleftarrow_1 \emptyset) \vee (\dagger\,\ell' \;*\; \Diamond n \;*\; \Phi) \\
\mathit{outsiders}\,\gamma_1\,\gamma_2\,G &\triangleq \boxed{\bullet\,G}^{\gamma_2} \;*\; \mathop{\Huge *}_{(\ell,\ell') \in G} \mathit{volatile}\,\ell'\,2\,(\dagger\,\ell) \\
\mathit{cells}\,\gamma_1\,G &\triangleq \exists A.\, \boxed{\bullet\,A}^{\gamma_1} \;*\; \mathop{\Huge *}_{\ell \in A} \big(\dagger\,\ell \vee \exists L.\, \ell \hookleftarrow_{\frac{1}{2}} L \;*\; \ulcorner \forall \ell'.\, \ell' \,\#\, L \leq (\ell,\ell') \,\#\, G \urcorner \big) \\
\mathit{stack}\,\ell\,L &\triangleq \exists \gamma_1\,\gamma_2\,G.\, \mathit{meta}\,\ell\,(\gamma_1,\gamma_2) \;*\; \mathit{innerList}\,\gamma_1\,\ell\,L \;*\; \mathit{cells}\,\gamma_1\,G \;*\; \mathit{outsiders}\,\gamma_1\,\gamma_2\,G
\end{aligned}
$$

Figure 49: Internals of Treiber's stack

The assertion $\mathit{reg}\,\gamma_1 x$ asserts that either x is the empty list (denoted by the unit value), or x is registered as a list cell in γ_1. This is a persistent assertion.

The assertion $\mathit{innerList}\,\gamma_1\,x\,L$ asserts that x represents the internal list of the Treiber's stack. This assertion is a variation of the list representation predicate (§11.2, Figure 39), where the points-to is made immutable, and only half of the pointed-by-heap assertion of the cell is stored. The assertion $\mathit{reg}\,\gamma_1 x$ witnesses that the cell x is registered as a list cell.

The assertion $\mathit{inner}\,\gamma_1\,\ell\,L$ is just a small wrapper around the main reference ℓ over the underlying list x.

The assertion $\mathit{cells}\,\gamma_1\,G$ stores half of the pointed-by-heap assertion of every list cell. These cells are represented by the set A, stored authoritatively in γ_1. Then, we constrain the multiplicity of elements of A in the multiset G. We write $x\#X$ for the multiplicity of the element x in the multiset X. For each cell of A, either it is deallocated, or there exists a multiset of predecessors L such that the multiplicity of every ℓ' in L is less than or equal to the multiplicity of the pair (ℓ,ℓ') in the multiset G. This fact records that every predecessor ℓ' of ℓ are correctly registered in G (We use multisets even if the multiplicities are always 1 or 0, because proving that predecessors are all distinct is more work than just handling the general case.)

The assertion $\mathit{volatile}\,\ell'\,n\,\Phi$ is a high-level combinator, which asserts either (1) the ownership of the resources needed to deallocate ℓ', or (2) the proof that ℓ' was deallocated as well as the ownership of Φ. Indeed, the assertion $\mathit{volatile}\,\ell'\,n\,\Phi$ is defined as a disjunction, with either (1) the "*sizeof*" assertion of ℓ' as well as its empty pointed-by-heap and -thread assertions, or (2) the deallocation witnesses of ℓ', its associated space credits, and Φ.

The assertion $\mathit{outsiders}\,\gamma_1\,\gamma_2\,G$ is where the magic happens. This assertion stores authoritatively G inside γ_2. Then, for every pair of a list cell ℓ and a private block ℓ', the assertion *outsiders* asserts that ℓ' is volatile, and if ℓ' was deallocated, then ℓ was deallocated too.

The assertion $\mathit{stack}\,\ell\,L$ gather the pieces by existentially quantifying over the two names of the two ghost cells and the ghost multiset G, fixing the two ghost names with a meta assertion and asserting the ownership of the assertions described above.

A thread executing "*push*" must register the private list cell ℓ' pointing to a cell ℓ by adding a pair (ℓ,ℓ') in G. This forces the thread to give up the empty pointed-by-thread and -heap assertions of ℓ' in order to satisfies $\mathit{volatile}\,\ell'\,2\,(\dagger\,\ell)$. To witness its addition to multiset G, the thread obtains the fragmentary assertion $\boxed{\circ\,\{+(\ell,\ell')\}}^{\gamma_2}$. Moreover, at each step, the thread executing "*push*" has to deal with the two alternatives presented in the

assertion $volatile\,\ell'\,2\,(\dagger\,\ell)$: either ℓ' is still allocated, or it was logically deallocated by another thread and so do ℓ.

When a thread successfully pops a cell ℓ from the list, this thread gains back the empty pointed-by-thread assertion of ℓ with fraction 1 and half of the pointed-by-heap assertion of ℓ from the assertion *inner*. The other half lies in *cells*, and is not empty: it may contain predecessors, all registered in G. Thanks to the assertion *outsiders*, we are able to logically deallocate all these predecessors, materializing our "logical deallocation by proxy". With both empty halves of the pointed-by-heap assertion of ℓ, and its empty pointed-by-thread, we can finally logically deallocate ℓ.

12.6 Michael and Scott's Queue

Michael and Scott's queue is a lock-free and linearizable queue [Michael and Scott, 1996]. Originally, the queue proposed by Michael and Scott is represented by a non-empty list, with two distinguished cells: the *sentinel*, a cell whose successor contains the next element to dequeue, and the *tail*, a cell which conceptually contains the last element to dequeue. Note the use of the word "conceptually": the particularity of Michael and Scott's queue is that the tail is allowed to *lag behind*, that is, to point to some cell between the sentinel and the effective last cell.

Due to this particularity, the verification of Michael and Scott's queue is non-trivial. In the Separation Logic world, the first proof was conducted by Vindum and Birkedal [2021]. They prove that Michael and Scott's fine-grained lock-free queue refines a coarse-grained queue protected with a lock. As we will see, we reuse a large part of their proof.

In the presence of tracing garbage collection, the code of Michael and Scott's queue needs two modifications.

First, the queue exhibits the same issue as Treiber's stack and as every lock-free data structure: functions of the API are invisible readers—that is, they do not synchronize with other threads before reading an internal location of the data structure. This absence of synchronization implies that, without a protected section, a thread can hold an internal location of the data structure as a root for an arbitrary amount of time, preventing the deallocation of the said location. We show how protected sections fix this issue.

Second, the direct translation of the original code of Michael and Scott in a garbage-collected memory setting has a space leak. Indeed, in the original code, the sentinel cell usually contains the last dequeued value. This fact is not an issue in the manual memory management setting of Michael and Scott. Indeed, the original code ensures to never access this garbage value stored in the sentinel cell, allowing the user to manually deallocate this value, and create a "safe" dangling pointer. However, in the setting of tracing garbage collection and without modification of the code, the sentinel block keeps reachable the last dequeued value. This leak was also identified by the developers of the `saturn` library, which implements in OCaml standard lock-free data structures [Karvonen, 2023a]. To fix the leak, we need to add an additional write (in order to overwrite the value that has just been dequeued) after a successful dequeue. This modification has an impact on the proof: the argument of Vindum and Birkedal [2021] is based on the fact that the underlying list is immutable (except for the last cell). With our additional write, the list is no longer immutable, so the argument needs to be adapted.

Interestingly, the code that Vindum and Birkedal [2021] verify as well as the implementation of Michael and Scott's queue in `saturn` at the time of writing [The saturn Developement Team, 2024], does not suffer from the space leak: for reasons unrelated to space usage, they introduce an indirection between each cell of the data structure. This indirection solves the matter. Indeed, the mutable pair representing the queue now stores pointers to list cells. When a dequeue operation is performed, the sentinel is updated to store the pointer to the next list cell, and not the direct location of the former first cell of the list. Hence, this former

```
create ≜ μptr_. λ[].
  let c = alloc 2 in
  (pair [c, c])ptr

enqueue ≜ μptr f. λ[q, v].
  let c = alloc 2 in
  c[0] ← v ;
  enter ;
  let t = q[1] in
  let x = t[1] in
  if x = () then
    if CAS t[1] () c
    then (CAS q[1] t c ; exit)
    else (exit ; (f [q, v])ptr)
  else
    (CAS q[1] t x ; exit ; (f [q, v])ptr)
```

```
dequeue ≜ μptr f. λ[q].
  enter ;
  let s = q[0] in
  let t = q[1] in
  let x = s[1] in
  if s = t then
    if x = ()
    then (exit ; (f [q])ptr)
    else (CAS q[1] t x ; exit ; (f [q])ptr)
  else
    if CAS q[0] s x
    then (let v = x[0] in x[0] ← () ; exit ; v)
    else (exit ; (f [q])ptr)
```

QueueCreate
$$\{\Diamond 4\}\ \pi\colon \mathit{create}\ ()\ \{\lambda \ell.\ \mathit{queueInv}\,\ell \ *\ \mathit{queue}\,\ell\,[]\ *\ \ell \Lleftarrow \{\pi\}\ *\ \ell \mapsfrom \emptyset\}$$

QueueEnqueue
$$[\{\ell\}]\left\langle \frac{\mathit{queueInv}\,\ell \ *\ \Diamond 2 \ *\ v \Lleftarrow_p \{\pi\}\ *\ v \mapsfrom^{0}_{q} \emptyset}{\forall L.\qquad \mathit{queue}\,\ell\, L} \right\rangle \pi\colon (\mathit{enqueue}\ [\ell, v])_{\mathsf{ptr}} \left\langle \frac{\lambda().\qquad \ulcorner \mathrm{True} \urcorner}{\mathit{queue}\,\ell\,(L +\!\!+ [(v,p,q)])} \right\rangle$$

QueueDequeue
$$[\{\ell\}]\left\langle \frac{\mathit{queueInv}\,\ell}{\forall v\,p\,q\,L.\ \ \mathit{queue}\,\ell\,((v,p,q) :: L)} \right\rangle \pi\colon (\mathit{dequeue}\ [\ell])_{\mathsf{ptr}} \left\langle \frac{\lambda w.\ \ulcorner w = v \urcorner\ *\ v \Lleftarrow_p \{\pi\}\ *\ v \mapsfrom^{0}_{q} \emptyset}{\mathit{queue}\,\ell\,L\ *\ \Diamond 4} \right\rangle$$

QueueFree
$$\mathit{queueInv}\,\ell\ *\ \mathit{queue}\,\ell\,L\ *\ \ell \Lleftarrow \emptyset\ *\ \ell \mapsfrom \emptyset \quad \Rrightarrow \quad \Diamond(4 + 2 \times |L|)\ *\ \mathop{\Large *}_{(v,p,q)\in L} (v \Lleftarrow_p \emptyset\ *\ v \mapsfrom^{0}_{q} \emptyset)$$

QueueInvPersistent
$\mathit{queueInv}\,\ell$ is persistent

QueueTimeless
$\mathit{queue}\,\ell\,L$ is timeless

Figure 50: Code and specification of Michael and Scott's queue

list cell can be garbage collected, as well as the value it points-to if needed. However, the "indirection" solution is unsatisfactory: indirections are usually costly.

Code Our implementation of Michael and Scott's queue appears in the upper part of Figure 50. Operationally, as previously explained, the elements of the queue are stored in a mostly immutable non-empty list. Again, the empty list is represented with the unit value () and a list cell with a block of size 2. Moreover, we do not specify or verify `try_enqueue` and `try_dequeue` variants.

The queue itself consists of a mutable pair whose two fields store the address of two (non-necessarily distinct) cells of the underlying list: the *sentinel* and the *tail*—as previously explained the latter does not necessarily correspond to the last cell of the list, has the tail may be *lagging behind*. At all times, the sentinel is a cell storing a garbage value (which must be not allocated on the heap, we chose the unit value), and whose successor is either empty (if the queue is empty), or another list cell storing the first element of the queue. The tail points to a list cell between the head and the last cell of the list.

The function call *create* () allocates an empty queue. First, a list cell c is created, containing no value nor successor: both fields are initialized to () by "alloc". The initial state of the queue, a mutable pair, is returned: the sentinel (the first component of the pair) is c, and the tail (the second component of the pair) is also c.

The function call $(enqueue\ [q, v])_{\mathsf{ptr}}$ adds the value v at the end of the queue q. First, a new list cell c is created, with content set to v. Then, a protected section is entered. Indeed, the next instruction acquires the tail t, which is an internal cell. Then, the successor of t is acquired and named x. If $x = ()$, we know that t was indeed the last list cell. Hence, a CAS is attempted to update the successor of t from () to c.

First, if the CAS succeeds, the enqueue operation is a success, but the tail is now lagging behind: it points to t, whereas the last list cell is now c. Another CAS is made to attempt to update the tail pointer from t to c, the new last pointer. Whatever the result, the protected section is exited and the function terminates.

Second, if the CAS fails, another thread interfered with the queue: another attempt to enqueue is made with a recursive call. If $x \neq ()$, it means that the original tail t was lagging behind. An effort is made to advance the queue pointer with a CAS, and another attempt to enqueue is made with a recursive call.

The function call $(dequeue\ [q])_{\mathsf{ptr}}$ removes an element from the queue q, and actively loops if the queue is empty. A protected section is first entered. The sentinel s, the tail t and the successor x of the sentinel (the cell containing the first element of the queue) are acquired.

Then a test is performed on whether s is equal to $= t$.

If so, either the tail is lagging behind (that is, $x \neq ()$), or the queue is empty. If the tail is lagging, an attempt is made to advance it. In both cases, the protected section is exited, and another attempt is made.

If s is distinct from t, a CAS is made to attempt to update the sentinel from s to x. If it succeeds, the value stored in x is loaded and overwritten. Then the protected section is exited, and the value is returned. If the CAS fails, the protected section is exited and another attempt is made. The additional write overwriting x prevents a space leak. We note that this write is not needed if the value being dequeued is not allocated on the heap.

Specifications The lower part of Figure 50 presents our specifications of Michael and Scott's queue. The queue is described in terms of two abstract assertions. First, $queueInv\ \ell$ asserting that ℓ represents a valid queue. This first assertion is persistent (QueueInvPersistent). Second, $queue\ \ell\ L$ asserting that the current content of the queue ℓ is L. This model follows our recipe for containers (§11.1) and consists of a list of triples (v, p, q) of a value v and two positive fractions p and q. This second assertion is not persistent, but is timeless (QueueTimeless). This is important: the assertion $queue\ \ell\ L$ is meant to be shared within an invariant.

According to QueueCreate, creating a new queue consumes four space credits: two for the initial list cell and two for the mutable pair. The result is a fresh location ℓ that represents a valid queue with an empty content.

The specification of $(enqueue\ [\ell, v])_{\mathsf{ptr}}$, expressed by QueueEnqueue, is an atomic triple with a souvenir on ℓ. The private precondition requires that ℓ be a valid queue. It also requires two space credits to account for the new list cell, as well as fractional pointed-by-heap and pointed-by-thread assertions for the value v that is enqueued Together, the public precondition and postcondition indicate that the model of the queue is atomically updated from L updated to $(v, p, q) +\!\!+ [L]$ at the linearization point.

The specification of $(dequeue\ [\ell])_{\mathsf{ptr}}$, expressed by QueueDequeue, is also an atomic triple with a souvenir on ℓ. The private precondition requires that ℓ be a valid queue. The public precondition and postcondition indicate that the model of the queue is atomically updated from $(v, p, q) :: L$ to L. Furthermore, according to the public postcondition, at the linearization point, two space credits are produced. The empty pointed-by-heap and the pointed-by-thread assertion of the value being dequeued are returned in the public postcon-

dition. Contrary to Treiber's stack, the pointed-by-heap assertion cannot be returned in the public postcondition: the additional write that removes the heap edge between the sentinel cell and the dequeued value occurs *after* the linearization point. We note that, adding complexity to the specification, a fraction of the pointed-by-heap assertion could be produced in both the public and the private postconditions.

QUEUEFREE concludes the reasoning rules for Michael and Scott's queue. This rule logically deallocates a (possibly nonempty) queue. The assertions $\mathit{queueInv}\,\ell$ and $\mathit{queue}\,\ell\,L$ are consumed, as well as well as empty pointed-by-thread and -heap assertions for ℓ. A number of space credits are produced, which reflect the overall size occupied by the queue data structure in the heap: two credits for the mutable pair, two credits for the sentinel cell, and two credits per remaining list cell. The pointed-by-thread and -heap assertions associated with every triple (v, p, q) in the queue are also produced.

We note a small difference between the specifications of Michael and Scott's queue and Treiber's stack (§12.5). Indeed, for Treiber's stack, there was no need for a "*stackInv*" assertion. This is due to a limitation of atomic triples, in which we lose access to the whole public precondition after the commit point. However, for Michael and Scott's queue, the structure is modified after the commit point, to advance a possibly-lagging tail. To allow for verifying this pattern, we must somehow keep access to the data structure after the commit point, which we achieve through the additional invariant "*queueInv*". Hiding this invariant *queueInv* inside the assertion *queue* is not possible, as it would make the assertion *queue* not timeless.

Proof Insights For functional correctness, we reuse a large part of the argument of Vindum and Birkedal [2021]. Their argument is based on *reachability* between list cells: the sentinel can reach the tail, and the tail can reach the last cell. Vindum and Birkedal represent these reachability invariants using persistent points-to assertions. Yet, our points-to assertions cannot be persistent, since a write operation must be made when a cell becomes a sentinel after a successful dequeue operation, to avoid a space leak. Thankfully, this write operation impacts only the data of the list cell, and not the overall linkage between cells. Hence, by making use of ghost state, we "cut" the points-to assertions of list cells into *per-field* permissions: one ephemeral part for the data field (the first offset), and one persistent part for the successor field (the second offset). The part of the proof concerning functional correctness is then almost identical to the proof of Vindum and Birkedal.

Concerning space reasoning, we make use of the reasoning rules of protected sections to logically deallocate the ex-sentinel after a successful dequeue. This is roughly the same approach as for Treiber's stack: thanks to protected sections, we logically deallocate a cell even if it is a root inside a protected section of an ongoing enqueue or dequeue operation. Interestingly, there is no need for the "logical deallocation by proxy" technique of Treiber's stack: because enqueuing is made at the end of the list structure, a new cell has no allocated successor—it points to "nil". This contrasts with Treiber's stack, in which new cells are pushed at the beginning of the list, and hence point to the first block of the list.

Last, one of the proofs of Vindum and Birkedal makes use of a prophecy variable [Jung et al., 2020] due to a "free" operation. As they verify in another proof, such a prophecy variable is not needed in the presence of a GC. Even in the presence of protected sections, we do not need a prophecy variable either.

Chapter 13

Related Work

Queen (1975).
Bohemian Rhapsody.

13.1 Polling Points

A stop-the-world event may be viewed as an asynchronous interruption: a thread that requests garbage collection stops the execution of all other threads. Such an interruption can be implemented using hardware interrupts, but this scheme can be expensive and non-portable [Feeley, 1993]. Another approach is to let the compiler insert explicit tests for interruptions into the code. These tests appear in the literature under various names, including *polling points* [Feeley, 1993], *GC points* [Agesen, 1998], *yield points* [Lin et al., 2015], and *safe points* [Sivaramakrishnan et al., 2020]. Let us refer to them collectively as *safe points*. Safe points are typically inserted by the compiler in such a way that no computation can run forever without encountering a safe point. When a thread encounters a safe point, it tests whether some other thread has requested garbage collection. If so, it pauses and passes control to the runtime system. Once all threads have paused in this way, the runtime system performs a global garbage collection phase.

Safe points are used in the Jalapeño/Jikes RVM [Alpern et al., 1999, 2005] and in OCaml 5 [Sivaramakrishnan et al., 2020]. The existence of safe points is not revealed to the programmer, who is not expected to know about their existence and is given no means of controlling their placement. As an experimental feature, the OCaml 5 compiler does offer a `[@poll error]` attribute [Jaffer, 2021]. This attribute is placed on a function definition. An attempt by the compiler to insert a safe point into a function that carries this attribute causes a compile-time error. This lets the programmer check that a function body does not contain any safe point, therefore is (de facto) a protected section. At this time, there is not a clear consensus whether this feature is useful and corresponds to the needs of expert programmers.

Safe points, as described above, and polling points, as proposed in this thesis, are two related yet distinct concepts. Indeed, in our view, safe points play two distinct roles. On the one hand, they are *polling points*, in the sense of this thesis: they are points where a thread must stop and allow garbage collection to take place if it has been requested. On the other hand, at the same time, they are delimiters (that is, starting points and ending points) of *protected sections*: indeed, *the GC cannot run unless every thread has reached a safe point.* We believe that our design, where protected sections and polling points are separate concepts, is better behaved. In particular, it enjoys *monotonicity* properties: inserting a new polling point, creating a new protected section, or enlarging an existing protected section *restricts* the set of possible behaviors of the program.[1] In contrast, in a setting where only a “safe point” construct is offered by the language, inserting a new safe point creates one more program point where the GC is allowed to run, therefore can *enlarge* the set of possible behaviors of the program and compromise the program’s worst-case heap space complexity. In short, in such a setting, automated safe point insertion is arguably unsafe!

[1] Polling points must be inserted only outside protected sections. In our setting, inserting a new polling point does not create a new opportunity for the GC to run, because outside protected sections, the GC is everywhere allowed to run.

In our approach, the user *explicitly* inserts enough protected sections to (verifiably) obtain the desired worst-case heap space complexity, then lets the compiler *implicitly* insert enough polling points to guarantee liveness, without endangering the program's space complexity. This is expressed by Theorem 2.

13.2 Protected Sections

In the production systems that we are aware of, the concept that seems closest to our protected sections appears in the .NET runtime system, where it was introduced in 2015, with performance in mind [Lander, 2015]. The API of the `GC` module [Microsoft, 2024] provides a method `TryStartNoGCRegion(Int64)` and a method `EndNoGCRegion()`. A "NoGC region" is not quite a protected section in our sense, though, as allocation is permitted inside a "NoGC region". The integer parameter of the method `TryStartNoGCRegion` is a request for a certain amount of free heap space: garbage collection takes place at this point so as to guarantee that this much free space exists. Allocation requests within the "NoGC region" are then served out of this pre-allocated free space. However, if the runtime system runs out of free space while some thread is inside a "NoGC region", then garbage collection will take place.

Beside performance, another possible motivation for temporarily disabling garbage collection is safety. Feeley [1993, §1.2.1] discusses why "critical sections"—sections in which the GC must not run—may be needed for safety reasons. He takes the example of a store instruction that stores a 64-bit pointer into memory and that is decomposed into two 32-bit stores. In between the two 32-bit stores, the memory is in an inconsistent state and must not be read by the GC.

To the best of our knowledge, our paper is the first where a notion of protected section is introduced for complexity reasons, that is, with the aim of guaranteeing tighter worst-case heap space complexity bounds.

13.3 Reasoning about Space without a GC

Hofmann [1999, 2003] introduces space credits in the setting of an affine type system for the λ-calculus. Hofmann [2000] and Aspinall and Hofmann [2002] adapt the idea to LFPL, a first-order functional programming language without GC and with explicit destructive pattern matching. There, a value of type $\Diamond$ exists at runtime and can be understood as a pointer to a free block in the heap. Subsequent work aims at automating space complexity analyses. In particular, Hofmann and Jost [2003] propose an affine type system where types carry space credits. Hofmann and Jost [2006] and Hofmann and Rodriguez [2009, 2013] analyze a variant of Java where garbage collection has been replaced with explicit deallocation. RaML [Hoffmann et al., 2012a,b, 2017] analyzes a fragment of OCaml, also without GC and with explicit destructive pattern matching. Niu and Hoffmann [2018] present a type-based amortized space analysis for a pure, first-order programming language where destructive pattern matching can be applied to shared objects, an unusual feature. Their system performs significant over-approximations: when a data structure becomes shared, the logic charges the cost of creating a copy of this data structure. As far as we understand, this analysis can be used to reason in a sound yet very conservative way about a programming language with GC. Kahn and Hoffmann [2021] present a system that is equipped with more flexible typing rules than its predecessors and can thus derive tighter resource consumption bounds. Hoffmann and Jost [2022] offer a survey of two decades of work on automated amortized resource analysis (AARA).

Nguyen et al. [2007] propose an automated verification system based on Separation Logic. They allow user-defined inductive predicates, which can be indexed with sizes (often the depth of the data structure). They are not concerned about heap space. He et al. [2009] re-use an existing Separation Logic-based program verifier, Hip/Sleek, to reason about stack and heap

space. They consider a C-like imperative language with explicit deallocation instructions. To reason about space, they instrument the program with two global variables `stk` and `heap` of type *int*, which represent the available space in the stack and the heap, respectively.

Following the ideas of LFPL, Lorenzen et al. [2023] introduce a calculus with "reuse" credits. Explicit destructive pattern matching produces reuse credits, which can be used to satisfy a new allocation. Because they want to reuse space *in place*, they need to cater for fragmentation: reuse credits need to describe contiguous memory. This contiguity requirement is guaranteed by the fact reuse credits cannot be joined. The goal of Lorenzen et al. [2023] is to statically detect *fully in-place* functions—that is, functions that do not need to allocate new memory. This includes, for example, functions that reuse the heap space occupied by their arguments.

Chin et al. [2005, 2008] present a type system that automatically keeps track of data structure sizes. The type system incorporates an alias analysis, which distinguishes between shared and unique objects and allows unique objects to be explicitly deallocated. Shared objects can never be logically deallocated. Specifications indicate how much memory a method may need (a high-water mark) and how much memory it releases, in terms of the sizes of the arguments and results.

Compared with type systems, program logics offer weaker automation but greater expressiveness. Aspinall et al. [2007] propose a VDM-style program logic, where postconditions depend not only on the pre-state, post-state, and return value, but also on a cost. Atkey [2011] extends Separation Logic with an abstract notion of resource, such as time or space, and introduces an assertion that denotes the ownership of a certain amount of resources.

All of the work cited above concerns languages with explicit memory deallocation, where there is no need to reason about unreachability. Reasoning about unreachability in the setting of a static analysis or program logic is a central challenge.

13.4 Reasoning about Space with a GC

In the setting of Java bytecode, Albert et al. [2007, 2013] infer recurrence equations that describe the heap space consumption of a first-order method, expressed as a function of the sizes of its arguments. Their system relies on an external analysis that infers object lifetimes and determines when objects can be deallocated. Their system characterizes several quantities at a program point via recurrence equations: these include total memory allocation, live memory (that they call "active memory"), and peak heap space consumption. Albert et al. [2015] explored how to scale their approach to more generic "non-cumulative resources" (that is, resources that can be consumed and produced), like heap and stack space. Albert et al. [2019] propose an extension for concurrent distributed systems.

Also in the setting of Java, Braberman et al. [2006, 2008] and Garbervetsky et al. [2011] synthesize a formula that bounds the amount of memory allocated by a method, as a function of its parameters. The tool of Garbervetsky et al. [2011] first infers scope-based memory regions and then infers their sizes. They do not support recursion.

Hur et al. [2011] propose a Separation Logic for the combination of a low-level language with explicit deallocation and a high-level language with a GC. Their concern is that when the location ℓ is deallocated by the GC, the assertion $\ell \mapsto v$ cease to be valid in the standard model of Separation Logic in which the points-to assertion describes the physical state. In order to solve the issue, they make use of the indirection of a *logical state*, with respect to which the points-to assertion is defined. They then enforce the invariant that the physical state and the logical state coincide on the reachable fragment. We follow a similar approach (§8). Hur et al. [2011] allow the GC invariant to be temporarily broken within "GC-unsafe" sections. Contrary to us, they are not interested in space complexity.

Madiot and Pottier [2022] and Moine et al. [2023] propose Separation Logics that allow reasoning about space in the presence of a GC.

The logic presented by Madiot and Pottier [2022] concerns a low-level language with explicit stack cells. Its reasoning rules are intended to support concurrency, but the paper does not provide any case study.

The logic presented in our previous paper [Moine et al., 2023] concerns a high-level language, where the call stack is implicit, but is restricted to a sequential setting. This paper also introduces support for closures. The logic relies on a distinction between *visible roots*—the roots of the term under focus—and *invisible roots*—the roots of the evaluation context. The logic keeps track of invisible roots using a *Stackable* assertion, and introduces the idea that *Stackable* assertions must be "forcibly framed out" at applications of the Bind rule. We re-use this idea in our own Bind rule (§6.4), but replace *Stackable* assertions with pointed-by-thread assertions, which are better suited to a concurrent setting. In so doing, we remove the distinction between visible roots and invisible roots, which does not seem to make sense in a concurrent setting; our pointed-by-thread assertions keep track of all (ordinary) roots. In contrast, Moine et al. [2023] do not keep track of visible roots via an a dedicated assertion: indeed, in their setting, it suffices to inspect the term under focus to determine the set of visible roots. This allows them to offer a standard Load rule, whereas our Load rule updates a pointed-by-thread assertion for the value that is loaded (§6.2).

Our mechanization [Moine, 2024] includes an encoding inside IrisFit of our previous logic for sequential programs [Moine et al., 2023]. This encoding demonstrates that our concurrent program logic can be used to reason about sequential programs with no overhead.

13.5 Space-Related Results for Compilers

Paraskevopoulou and Appel [2019] prove that, in the presence of a GC, closure conversion is safe for space: that is, it does not change the space consumption of a program. They view closure conversion as a transformation from a CPS-style λ-calculus into itself. This calculus is equipped with two different environment-based big-step operational semantics. The "source" semantics implicitly constructs a closure for each function definition by capturing the relevant part of the environment and storing it in the heap. The "target" semantics performs no such construction: it requires every function to be closed. In either semantics, the roots are defined as the locations that occur in the environment. Up to the stylistic difference between a substitution-based semantics and an environment-based semantics, this definition is equivalent to the "free variable rule" (FVR) [Morrisett et al., 1995].

Besson et al. [2019] prove that (an enhanced version of) CompCert [Leroy, 2024] preserves memory consumption when compiling C programs.

In a sequential setting, Gómez-Londoño et al. [2020] prove that the CakeML compiler respects a cost model that is defined at the level of the intermediate language DataLang, which serves as the target of closure conversion. Our cost model is analogous to theirs. Our work and theirs are complementary: whereas they prove that the CakeML compiler respects the DataLang cost model, we show how to establish space complexity bounds about source programs, based on a similar cost model. One could in principle adapt IrisFit to DataLang. Then, one would be able to use IrisFit to establish a space complexity bound about a source CakeML program, to compile this program down to machine code using the CakeML compiler, and to obtain a machine-checked space complexity guarantee about the compiled code.

13.6 Safe Memory Reclamation Schemes

Manual memory management can be so difficult in a concurrent setting that programmers often rely on semi-automatic *safe memory reclamation* (SMR) schemes. Two main families exist, namely hazard pointers [Michael, 2004; Michael et al., 2023] and read-copy-update (RCU) [McKenney, 2004; McKenney et al., 2023]. The two families offer roughly similar APIs. First, the user declares *hazardous* locations for a delimited scope. While it is marked

hazardous, a location is not deallocated. Second, the user can *retire* a location to indicate that this location is no longer needed. The SMR implementation deallocates a retired location once it is not marked hazardous by any thread.

RCU seems particularly close to our concept of a protected section. Indeed, RCU declares *every* pointer hazardous inside a certain section of the code. Yet, there is not a perfect analogy between the two. Indeed, garbage collection provides a strong guarantee: *no dangling pointer can exist*. SMR schemes, on the contrary, tolerate dangling pointers. Hence, with RCU, a location that the code mentions, but without reading or writing it, does not need to be protected. For example, the "*push*" operation of Treiber's stack does *not* need an RCU section [Jung et al., 2023, mechanization], whereas the "*pop*" operation does need one. Indeed, the *push* operation never accesses the content of an internal list cell. Hence, it is not dangerous if such a location is deallocated in the meantime.

Equipping SMR schemes with abstract Separation Logic specifications and verifying them has long been a challenge. Treiber's stack has been the first data structure based on hazard pointers to be verified. This task was tackled several times using different variants of Concurrent Separation Logic [Parkinson et al., 2007; Fu et al., 2010]. Tofan et al. [2011] verify Treiber's stack both with hazard pointers and with garbage collection (though without a heap space complexity analysis). They show that a large part of the main invariant can be shared between the two proofs. Gotsman et al. [2013] provide the first general framework for verifying programs using SMR schemes in Separation Logic, making use of temporal logic reasoning. Jung et al. [2023] provide a more abstract framework, where temporal reasoning is replaced with ownership arguments. Their work unveils a close relationship between RCU and garbage collection. Indeed, RCU allows accessing any location that was *not* retired when the current RCU section was entered. (There is a loose analogy with our liveness-based cancellable invariants: to access such an invariant, one must eliminate the case where ℓ has been logically deallocated.) To prove that a location is *not* retired at a certain point in time, Jung et al. [2023] express the topology of data structures using pointed-by-heap assertions, which they borrow from our prior paper [Moine et al., 2023].

Outside the Separation Logic world, Meyer and Wolff [2019] propose an API for SMR schemes in the form of an observer automaton, inspired by the temporal reasoning of Gotsman et al. [2013]. Meyer and Wolff [2019] make use of the observer automaton to decorrelate the verification of lock-free data structures from the SMR implementation.

13.7 Disentanglement

Disentanglement [Raghunathan et al., 2016; Guatto et al., 2018; Westrick et al., 2020] is a property of parallel programs which intuitively asserts that "parallel tasks remain oblivious to each other allocations". Disentangled programs can be equipped with efficient memory management. Indeed, if the program is disentangled, then one can supply each parallel task with its own local heap. Each task can then allocate and reclaim memory independently of other tasks. This approach allows for faster memory management than for arbitrary concurrent programs, where the GC must take into account every running thread. Recall for example that in OCaml5 threads synchronize before running the GC. Arora et al. [2021] present a provably-efficient memory manager for disentangled programs, and Arora et al. [2023] show how to handle entanglement at runtime without losing too much efficiency.

Related to our work, the semantics of disentanglement [Westrick et al., 2020, 2022] follows the free variable rule: the informal idea "parallel tasks remain oblivious to each other allocations" is formalized by the following invariant: if a location is a root (that is, a free variable) of a task, then this root must have been allocated either by the task itself or by one of its parent in the so-called *parallel task tree*. With our experience of IrisFit, the fact that disentanglement relies on the free variable rule led to a collaboration with Sam Westrick and Stephanie Balzer to propose a Separation Logic for verifying that a program is disentangled [Moine et al., 2024].

Chapter 14

Conclusion

Piazzolla, A. (1970).
Las Cuatro Estaciones Porteñas, Primavera Porteña.

We have presented LambdaFit, a lambda-calculus with shared-memory concurrency and tracing garbage collection. In particular, LambdaFit is equipped with protected sections, a new, realistic construct that programmers can and sometimes must exploit to ensure that fine-grained concurrent data structures have the desired worst-case heap space complexity. We believe that protected sections are a necessary part of a concurrent programmer's toolbox, and that they should be considered for inclusion in high-level languages.

Furthermore, we have presented IrisFit, a Concurrent Separation Logic with space credits, which allows expressing and verifying worst-case heap space bounds about LambdaFit programs. IrisFit features pointed-by-heap and pointed-by-thread assertions, which offer a compositional means of keeping track of the various ways through which a memory block is reachable. These assertions can be used to prove that a block is unreachable, or more accurately, that by the time the garbage collector is allowed to run, this block will be unreachable. IrisFit provides special treatment of temporary roots within protected sections and is thereby able to take advantage of protected sections to establish stronger worst-case heap space bounds.

In particular, IrisFit provides an answer to our motivating question (§1):

How to prove heap space bounds for concurrent programs under tracing garbage collection?

This thesis shows that IrisFit satisfies three desired criteria.

First, IrisFit is expressive and allows mimicking intuitive reasoning on heap space, thanks in particular to logical deallocation. Indeed, logical deallocation permits reasoning as if space is available as soon as a location becomes unreachable, while the GC has perhaps not yet run. IrisFit also supports other intuitive patterns, ranging from amortized analysis with rational space credits (§5.5) to logical deallocation of inner cells of lock-free data structures before they are truly unreachable, thanks to protected sections (§6.3).

Second, IrisFit is usable in practice. For example, we verify that an implementation of closure conversion is correct and we derive reasoning rules for closures (§9). We also tackle a wide variety of case studies, including standard sequential data structures (§11.5, §11.4), CPS definitions (§11.3), and sequential circular lists (§11.5). Moreover, we show that IrisFit allows establishing modular bounds at every level of abstraction: instruction, functions, and modules (§11.4). We also emphasize the purposefulness of IrisFit for fine-grained concurrent programs, including a concurrent counter implemented as a pair of closures (§12.3), an async/finish library (§12.4) and lock-free data structures such as Treiber's stack (§12.5) and Michael and Scott's queue (§12.6).

Third, IrisFit, its soundness proof, and all its case studies are entirely mechanized. We comment on the mechanization in the next section.

14.1 Mechanization

All of our results are mechanized in the Coq proof assistant using the Iris library [Jung et al., 2018b] and its dedicated Proof Mode [Krebbers et al., 2018]. Our definitions and proofs are

available in electronic form [Moine, 2024]. Discounting blank lines and comments, the definition of LambdaFit and of its oblivious semantics occupy roughly 2700 lines of code (LOC); the construction of IrisFit, including the reasoning rules and the core soundness theorem, represent 9300LOC; the definition of the default semantics of LambdaFit and the proof of the safety and liveness theorems take up 4400LOC; and the verification of the case studies represents 7200LOC. In addition to these numbers, we encapsulate the mechanization of possibly-null fractions and signed multisets in a library of 1700LOC. Additionally, we re-use as a library about 1900LOC of proofs establishing results about reachability within the heap from Madiot and Pottier [2022]. We provide tactics that facilitate reasoning with IrisFit and achieve a basic level of automation thanks to the Diaframe library [Mulder et al., 2022].

14.2 Perspectives

In this section, we present venues for future work. We first focus on direct future work for IrisFit (§14.2.1), then we present connections with OCaml (§14.2.2) and finish with broader perspectives (§14.2.3).

14.2.1 Improvements and Extensions of IrisFit

Improved User Experience Our mechanization of IrisFit comes with basic automation, but a lot remains to be done to improve the user's experience. First, we use Diaframe [Mulder et al., 2022] to a fraction of its power. We would like to use the tool more thoroughly in order to automatize basic goals. We would also like to use Diaframe's features for concurrency. These features include the support for logical atomic triples [Mulder and Krebbers, 2023] and the support for "connections", improving the automation of goals with disjunctions, which often appear while reasoning with invariants [Mulder et al., 2023]. Another avenue for improvement is the automation of goals related to sets of locations (for example, in the premise of BindWithSouvenir and TrimPBThread). We currently rely on the great `set_solver` tactic [The Coq-std++ Team, 2023]. Yet, for large terms, computation time can be a bottleneck and we would like to improve the situation.

Additional Case Studies While we verify several case studies, there is room for more. In particular, we would like to apply IrisFit to larger (in terms of number of LOC) examples as well as subtler concurrent examples. For the latter, Harris's list [Harris, 2001] and multi-CAS algorithms such as RDCSS [Harris et al., 2002] are interesting candidates.

A General Approach for Sharing A function that allocates a data structure similar to one of its argument usually come with two variants of its specifications. Either the said argument is unreachable from other parts of the program, and the space of the argument can be reused for allocating the new data structure, or the argument may be reachable, and space credits must be required in the precondition to satisfy the allocations. Yet, this approach lacks generality and has two drawbacks. First, this approach leads to an explosion of possible specifications. For example in §11.3, we show two specifications for list append, depending on whether the first list being appended has predecessors or not. But there are more possible scenarii, that often appear in functional programming: indeed, only a segment of the list may be shared. In this case, space credits must be required only for non-shared list cells. Second, this approach duplicates the proof effort: the proof of the specification logically deallocating an unreachable argument differs from the proof of a specification with provisioned space credits.

We would like to understand what could be a unique specification that fits all use cases, and how to prove the program correct in one pass. If these two goals prove impossible, we would like to devise a mechanism to generate and automatically verify a family of specifications.

Immutable Data Structures At present, IrisFit offers no special support for immutable data structures: every memory block is considered mutable by default, and it is up to the user to exploit the logical tools offered by Iris, such as invariants, to indicate that a memory block is immutable. In this thesis, we have done so in the special case of closures (§9): we have been able to describe the behavior of a closure via a *persistent* predicate, while still allowing for its deallocation. We would like to investigate whether this approach can be extended to all immutable data structures. If this approach can be extended, the user would benefit from less bookkeeping of fractions of pointed-by-heap assertions.

Coarse-Grained Specifications Specifications in IrisFit are *fine-grained*: they mention exactly, via the pointed-by-heap assertion, which location is pointed by which location and with which fraction. For containers (§11.1), we are able to temper this fine graininess and hide the exact predecessors of elements by capturing a fraction of their pointed-by-heap assertion inside the representation predicate. Yet, the need to precisely keep track of predecessors of locations, either by mentioning explicitly these predecessors or by fragmenting the pointed-by-heap assertion of the location, is tedious and unneeded in most cases. Indeed, programmers often think of coarser *regions*, portions of the heap storing multiple blocks which may all point to each other. At a high-level, only the reachability between regions matters. We currently propose an API for dead and unreachable regions (§6.6). We would like to enhance this support to live regions, add reasoning rules to reason about the reachability between regions, and relax existing reasoning rules while mutating blocks inside a particular region. The literature on *region-based* memory management [Tofte and Talpin, 1997; Tofte et al., 2004] and related type systems [Fluet et al., 2006; Birkedal et al., 2012; Elsman, 2023] may be of some inspiration.

In detail, we envision *logical regions* that the user could allocate and fill with locations while reasoning. Within a region, a location is assumed to be pointed by every other location of the region. With such an approach, one could design a StoreWithin reasoning rule which can be used when a location is stored inside a block that lives in the same region. Such a rule would only require the user to prove that the stored location and the block are indeed in the same region, via perhaps a persistent witness. Interestingly, this StoreWithin rule would not mention the pointed-by-heap assertion. Then, one would have, to keep track of the entry-points and exit-points of regions with new assertions. We speculate that keeping track of these high-level entry and exit points will be simpler than mentioning every particular heap link with the pointed-by-heap assertion.

Asymptotic Space Reasoning Specifications in IrisFit mention exact space bounds. With large data structures, for which the exact bound matters, this is fine. However, for smaller auxiliary data structures, programmers often think of *asymptotic space complexities*, and not the exact bound. Yet, formalizing asymptotic complexities is not an easy job. This problem was studied by Guéneau [2019] in the context of verifying time complexity bounds with time credits. We wonder to what extent we could reuse parts of Guéneau's approach for space. One challenge is that *constant factors matter for space credits*. For example, specifying the constructor of a pair as consuming a constant amount of space credits lacks information. Indeed, how to satisfy this constant amount of space credits? Reusing the space occupied by an unreachable list cell is safe, whereas reusing the space of an unreachable reference is not. Indeed, even if these two structures occupy a constant amount of heap space, a list cell occupies two memory words (similarly to a pair) whereas a reference occupies a single word.

IrisFit as a Foundation for Type Systems In §13, we covered a large number of *automated* approaches for the inference and verification of heap space bounds, often using type systems. An interesting venue for future work is to investigate if IrisFit can be used to provide foundational guarantees to these approaches. Indeed, this is the technique of *semantic typ-*

```
let before_pattern_matching_compilation (f:unit -> unit) (p: (int*int)) : int =
    match p with (x,_) -> f (); x

let after_pattern_matching_compilation (f:unit -> unit) (p: (int*int)) : int =
    f (); (fst p)
```

Figure 51: Pattern matching may extend the lifetime of variables

ing [Timany et al., 2024], one could encode and prove correct in IrisFit the reasoning rules applied by automated systems.

14.2.2 Links with OCaml

Protected Sections It would be interesting to offer and document protected sections to OCaml users. As we previously discussed (§13.2), we do not foresee any difficulty in order to implement protected sections by relying on OCaml's safe points. Indeed, it suffices to insert safe points outside of protected sections to guarantee the validity of heap space bounds established with IrisFit.

Protected sections are not only interesting from the point of view of space consumption. For example, in OCaml, hardware threads are mapped to *domains*, and only one lightweight thread (that is, a user-level thread, called `pthread` in the C/C++ world) can execute per domain. Lightweight threads are useful in practice: for example, one can install a lightweight thread per domain to handle asynchronous I/O. Interestingly, within a single domain, the OCaml runtime system interrupts lightweight threads *only at safe points*. Hence, as identified by Karvonen [2023b], one could use the control of safe points placement (via the limited existing `[@poll error]`, or via the proposed protected sections) to ensure that some portion of code is executed atomically with respect to other lightweight threads on the same domain. Such a use of protected sections is interesting, as actual synchronization primitives like CAS are expensive, whereas plain reads and writes are much cheaper. Hence, within a single domain, one can implement an efficient CAS as

```
let lightweight_compare_and_set r seen v =
    enter; if !r = seen then (l := v; exit; true) else (exit; false)
```

Hence, if multiple lightweight threads need to interact on the same data structure, one could use "atomic" lightweight thread operations like `lightweight_thread_compare_and_set` to improve efficiency.

Unsafe for Space Code Transformations At the time of writing, OCaml performs code transformations that *may extend the reachability of heap objects*. These transformations are clearly "unsafe for space": by extending the reachability of a heap object, the compiler pushes back the time by which this object will be collected by the GC.

One motivation to extend the lifetime of locations is to limit the number of simultaneously active variables by postponing field access as much as possible.

The compilation of pattern matching is a good example, where the compiler may extend the lifetime of matched variables. Consider the two functions presented in Figure 51. They are compiled into the same assembly code. Yet, `before_pattern_matching_compilation` may have a lower heap space bound than `after_pattern_matching_compilation`. Indeed, in the former, the pair `p` is unreachable just after the pattern matching and can hence be collected by the GC *before* the call to `f`, which can reuse the space of the pair. However, in the latter, `p` is a root of the evaluation context while executing `f`, and the space of `p` cannot be reused.

Another example of extended lifetimes appears with closures, where the environment is not immediately loaded. Instead, the closure block, storing the environment, is kept reachable and loaded when needed.

```
let before_dce (r : bool ref) : bool =
    let w = Weak.ref r in
    let x = Weak.get w in
    if true then (x <> None) else !r
```

```
let after_dce (r : bool ref) : bool =
    let w = Weak.ref r in
    let x = Weak.get w in
    (x <> None)
```

Figure 52: Impact of dead code elimination (DCE) on weak pointers

```
let before_cse (p : (int * int)) : int =
    let w = Weak.ref p in
    let x = fst p in
    if Weak.get w <> None then x+fst p else x
```

```
let after_cse (p : (int * int)) : int =
    let w = Weak.ref p in
    let x = fst p in
    if Weak.get w <> None then x+x else x
```

Figure 53: Impact of common sub-expression elimination (CSE) on weak pointers

Ideally, a compiler should document more precisely the list of its transformations that may extend the reachability of a heap object. Moreover, the compiler should provide a way of disabling such optimizations, in order to generate compiled code that respects space bounds established at the source level.

14.2.3 Broader Perspectives

Weak Pointers and Ephemerons Weak pointers [Jones et al., 2012, §12.2] are pointers that are *not* followed by the GC: they do not participate in the reachability of the value they point to. As a consequence, the GC can deallocate the content of a weak pointer, even if the weak pointer is reachable. Hence, dereferencing a weak pointer either fails and returns a special value `None` showing that the GC deallocated the pointed value, or succeeds and returns the said pointed value. Ephemerons [Hayes, 1997] generalize weak pointers. They are implemented in OCaml and in Haskell. An ephemeron is a memory block of size 2, storing a *key* and a *value*. The ephemeron weakly points to its key—that is, an ephemeron does not participate in the reachability of its key. Crucially, the GC considers reachable the value only if the key is considered reachable.

Ephemerons are useful in practice as they allow for “adding information” to an object without participating in its reachability. Ephemerons are often used to implement *weak hash tables*, where each pair of a key and a value is implemented with an ephemeron. Weak hash tables are themselves used to implement *hash-consing*, a crucial technique to prevent space-usage blowups of functional programs.

Yet, weak pointers and ephemerons all the more are difficult to reason about. Indeed, certain transformations of the compiler *may shorten the reachability of heap objects*. While shortening the lifetime of a heap object is not an issue for heap space bounds, this is problematic for weak pointers: the programmer could argue at the source level that the value of a weak pointer is reachable (and hence that dereferencing the weak pointer succeeds), but this reachability argument could be broken by the compiler!

Transformations shortening reachability are common, and include for example dead code elimination (DCE) and common sub-expression elimination (CSE).

For dead code elimination, consider the contrived example presented in Figure 52. The source program is on the left and the transformed program is on the right. In the source program, a weak reference `w` is created on the parameter `r`, and immediately after, the content of the weak reference is loaded and named `x`. Because `r` is mentioned in the dead “else” branch of the following conditional, one can be sure that `x` is not `None` (since such a case would show that the GC deallocated `r`). Yet, because the “else” branch is dead (that is, will never be executed), the elimination of dead code transforms the source program on the left of Figure 52 into the version on the right. However, in this transformed program, `r` is unreachable by the time `Weak.get` is executed. Hence the GC could have deallocated `r`, and the variable `x` may hence be `None`.

For common sub-expression elimination (CSE), let us consider the example presented in Figure 53, which we simplify from Donnelly et al. [2006, §5]. Again, the source program is on the left and the transformed program is on the right. A weak reference on the argument `p`, a pair, is made, then the first component of the pair is named `x`. Then a test is made to check if the GC deallocated `p`. This cannot be the case, as `p` is a root of the "then" branch. Hence the source program always returns `2`. Yet, the compiler might transform the code by observing that `fst p` appears two times in the code, and by replacing the last occurrence by `x`. The resulting code appears on the right side of Figure 53. It is not possible anymore to guarantee that the program returns `2`, as the GC may have deallocated `p` by the time `Weak.get` is executed.

The only existing formal semantics for weak pointers that we are aware of [Donnelly et al., 2006] resorts to asserting that dereferencing a weak pointer may always fail. This semantics is unsatisfactory: it does not allow verifying the correctness of hash-consed-based equality, for example. Let us sketch the issue. The core of the approach of hash-consing is to share structurally-equal objects. This is done by maintaining a weak hash set (a weak hash table with no values) of every object ever allocated. When the user wants to allocate a new heap object, they first test whether the object is in the hash set and still allocated, or not. If yes, the existing version is returned, and otherwise, the object is indeed allocated and stored. The set of hash-consing is weak as otherwise, the set would maintain reachable every object ever allocated. By having a weak hash set, the GC can still collect unused objects. Hash-consing ensures that at most one version of each object is in circulation. The user can hence reduce testing the structural equality of two objects by testing the equality of their addresses. However, with semantics where dereferencing a weak pointer can always fail, the whole approach falls down. Indeed, it would mean that we have to consider that the GC deallocated some object, and hence allocate a new version of it, while the original object is still in use elsewhere in the code.

Programmers make use of weak pointers and ephemerons, and play with their strange behavior. For example, [Peyton Jones et al., 1999, §5.4] propose a code that follows the pattern below:

```
let weird (r:int ref) : int =
    let w = Weak.ref r in
    if Weak.get = None then !r else 42
```

In this function, a weak reference `w` is created on the argument `r`. Then, a test is made on the result of `Weak.get w`. Interestingly, the branch handling the case `None` mentions `r`. Hence, we can be sure that `Weak.get w` does not return `None`, since `r` appears in the continuation. But this means that the said branch is dead, and eliminating it removes the mention of `r`, which in turn allows `Weak.get w` to return `None`, rendering the branch live! This example is a more evolved version of the impact of dead code elimination that we mentioned above. Quoting Peyton Jones et al. [1999], "this sort of weirdness is typical of weak pointers".

It could be interesting to investigate how to provide a satisfying semantics for weak pointers and ephemerons. After devising such a semantics, we would like to propose a program logic allowing the verification of programs making use of these constructs. An interesting goal would be to make use of space credits to specify and verify a hash-consing that internally makes us of ephemerons.

Finalizers A *finalizer* (or *finaliser*) [Jones et al., 2012, §12.1] is a method that is called by the GC when it deallocates a particular location. Finalizers may be useful when the program also manipulates manually managed memory. Through a finalizer, the programmer can manually deallocate memory at the same time as some automatically managed memory.

Similarly to weak pointers and ephemerons, finalizers allow for observing the reachability of the locations they are attached to, and their semantics is not well understood. We would like to investigate this topic. We would also like to propose reasoning rules for finalizers.

Indeed, finalizers are strange beasts: they are often given as input *the very object being deallocated by the GC*. Hence, finalizers have access to an "unreachable" location. Thus, finalizers can potentially *resurrect* this unreachable location, for example, by writing it in a reachable heap cell. Resurrecting a location is arguably a bad practice: it breaks the intuitive reasoning that an unreachable collection will eventually be collected. To prevent this practice, one could imagine making use of an approach similar to protected sections: by the time the finalizer ends, the user should prove that the location it was given as an argument is unreachable.

Finalizers are also not easy to reason about because the programmer cannot predict when a finalizer will run, as the programmer does not control when the GC runs. This fact complicates a hypothetical API for a finalizer: when should its precondition be satisfied? Indeed, in some subtle scenarii, the finalizer may depend on resources that do not exist by the time the finalizer is installed. This is morally correct, as we have the guarantee that the finalizer will not run until the associated location is again unreachable. Another question is: when does the postcondition of a finalizer should be made available to the user? As there is no way to observe the GC in LambdaFit, it seems difficult to answer this question. If we add a primitive calling the GC in the language, one could imagine a reasoning rule updating an assertion $\dagger\,\ell$ as well as an assertion asserting that a finalizer with postcondition Ψ was installed on ℓ into the assertion Ψ. Interestingly, with weak pointers, one could observe that a location was deallocated by the GC, and deduce that the associated finalizer must have run and obtain its associated postcondition.

In part because finalizers are so difficult to reason about, they were deprecated in Java 18 [Christian and Marks, 2021]. To this date, finalizers are still offered by Haskell and OCaml, for example.

Safe Encapsulation IrisFit allows for reasoning about unreachability. By making use of the pointed-by-heap and pointed-by-thread assertions, one can prove that some location cannot be accessed outside a particular abstraction boundary: the location is *encapsulated*, or *private*. (This notion supposes that the language does not permit forging locations out of thin air.) We identify two cases where one could want such encapsulation results, either using IrisFit directly, adapting a relational logic for establishing contextual refinements such as ReLoc [Frumin et al., 2021], or applying the "Theorems for free" of Separation Logic [Birkedal et al., 2021].

The first case consists of data structures relying on a private "dummy" element, generally a fresh location, that is used to fill empty slots. Yet, the correctness of the approach rests on the fact that the user has no access to this dummy element. Otherwise, the user could insert the dummy element into the data structure, and one would not be able to distinguish between an empty slot from a non-empty one. Currently, the general approach is to resort in revealing the existence of the dummy element in the specifications, such that the user is able to prove that the values being inserted are distinct from it. For example, in other work [Moine et al., 2024, §6.3], we present a lock-free set making use of a dummy element. Leaning out details about concurrency, these specifications involve a predicate $set\,\ell\,d\,S$ asserting that the location ℓ represents a set S with a user-chosen dummy element d, passed as an argument when creating the set. The specification of this set looks like:

SetCreate

$$\{\ulcorner \mathrm{True} \urcorner\}\ \mathsf{create}\,d\ \{\lambda \ell.\ set\,\ell\,d\,\emptyset\}$$

SetInsert

$$\frac{x \neq d}{\{set\,\ell\,d\,S\}\ \mathsf{insert}\,\ell\,x\ \{\lambda().\ set\,\ell\,d\,(S \cup \{x\})\}}$$

Notice the premise of SetInsert, requiring that the element being inserted is not the dummy element d. One solution to hide this dummy element could be to store its points-to assertion inside the representation predicate, and update SetInsert to frame an arbitrary fraction of the points-to assertion of the element being inserted. By using the fact that the full points-to assertion of a location excludes the existence of other points-to assertions of the same

location, one could deduce that the value being inserted cannot be the dummy element. Yet, this solution is unsatisfactory, as the points-to assertion of the element being inserted may be hard to exhibit for the user (especially in a concurrent setting, where points-to assertions may reside inside an invariant).

A second case consists of *robustness* properties: for safety reasons, one may want to verify that a data structure does not leak an internal reference, and hence that this reference cannot be modified by any client of the data structure. This is a "light" safety property it makes the hypothesis that the client is well-formed and cannot break abstraction boundaries (for example, in OCaml, a "well-formed" client cannot use `Obj.magic`, the primitive casting a value of any type to any other type). The *Necessity* approach [Mackay et al., 2022] for proving such robustness properties is parametric with respect to a technique for proving "assertion encapsulation". Even if their formal setting is different than ours, it would be interesting to investigate if IrisFit could be used to prove such properties.

Bibliography

Mozart, W. A. (1787).
Don Giovanni, "Madamina, il catalogo è questo".

Ole Agesen. 1998. *GC Points in a Threaded Environment.* Technical Report SMLI TR-98-70. Sun Microsystems, Inc. https://dl.acm.org/doi/10.5555/974974

Elvira Albert, Jesús Correas, and Guillermo Román-Díez. 2019. Peak resource analysis of concurrent distributed systems. *Journal of Systems and Software* 149 (2019), 35–62. https://doi.org/10.1016/j.jss.2018.11.018

Elvira Albert, Jesús Correas Fernández, and Guillermo Román-Díez. 2015. Non-cumulative Resource Analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, Christel Baier and Cesare Tinelli (Eds.). Springer, 85–100. https://doi.org/10.1007/978-3-662-46681-0_6

Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. 2007. Heap space analysis for Java bytecode. In *International Symposium on Memory Management.* 105–116. http://cliplab.org/papers/jvm-heap-ismm07.pdf

Elvira Albert, Samir Genaim, and Miguel Gómez-Zamalloa. 2013. Heap space analysis for garbage collected languages. *Science of Computer Programming* 78, 9 (2013), 1427–1448. https://doi.org/10.1016/j.scico.2012.10.008

Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2018. ThreadScan: Automatic and Scalable Memory Reclamation. *ACM Trans. Parallel Comput.* 4, 4, Article 18 (May 2018). https://doi.org/10.1145/3201897

Bowen Alpern, C. Richard Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark F. Mergen, Janice C. Shepherd, and Stephen E. Smith. 1999. Implementing Jalapeño in Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* 314–324. https://doi.org/10.1145/320384.320418

Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria A. Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark F. Mergen, J. Eliot B. Moss, Ton Anh Ngo, Vivek Sarkar, and Martin Trapp. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Syst. J.* 44, 2 (2005), 399–418. https://doi.org/10.1147/sj.442.0399

Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press. http://www.cambridge.org/9780521033114

Jatin Arora, Sam Westrick, and Umut A. Acar. 2021. Provably Space-Efficient Parallel Functional Programming. *Proceedings of the ACM on Programming Languages* 5, POPL (jan 2021). https://doi.org/10.1145/3434299

Jatin Arora, Sam Westrick, and Umut A. Acar. 2023. Efficient Parallel Functional Programming with Effects. *Proceedings of the ACM on Programming Languages* 7, PLDI (jun 2023). https://doi.org/10.1145/3591284

David Aspinall, Lennart Beringer, Martin Hofmann, Hans-Wolfgang Loidl, and Alberto Momigliano. 2007. A program logic for resources. *Theoretical Computer Science* 389, 3 (2007), 411–445. https://doi.org/10.1016/j.tcs.2007.09.003

David Aspinall and Martin Hofmann. 2002. Another Type System for In-Place Update. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 2305)*. Springer, 36–52. https://homepages.inf.ed.ac.uk/da/papers/readonly/readonly.pdf

Robert Atkey. 2011. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science* 7, 2:17 (2011), 1–33. https://lmcs.episciences.org/685/pdf

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Springer. https://www.labri.fr/perso/casteran/CoqArt/coqartF.pdf

Frédéric Besson, Sandrine Blazy, and Pierre Wilke. 2019. CompCertS: a Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics. *Journal of Automated Reasoning* 63, 2 (2019), 369–392. https://doi.org/10.1007/s10817-018-9496-y

Lars Birkedal and Aleš Bizjak. 2023. Lecture notes on Iris: Higher-order concurrent separation logic. (2023). https://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf Unpublished.

Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. 2021. Theorems for free from separation logic specifications. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473586

Lars Birkedal, Filip Sieczkowski, and Jacob Thamsborg. 2012. A Concurrent Logical Relation. In *Computer Science Logic (Leibniz International Proceedings in Informatics, Vol. 16)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 107–121. https://doi.org/10.4230/LIPIcs.CSL.2012.107

Wayne D. Blizard. 1990. Negative membership. *Notre Dame Journal of Formal Logic* 31, 3 (1990), 346–368. https://doi.org/10.1305/ndjfl/1093635499

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: an efficient multithreaded runtime system. In *Principles and Practice of Parallel Programming (PPoPP)*. ACM, 207–216. https://doi.org/10.1145/209936.209958

Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission accounting in separation logic. In *Principles of Programming Languages (POPL)*. 259–270. http://www.cs.ucl.ac.uk/staff/p.ohearn/papers/permissions_paper.pdf

John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis Symposium (SAS) (Lecture Notes in Computer Science, Vol. 2694)*. Springer, 55–72. https://doi.org/10.1007/3-540-44898-5_4

Víctor A. Braberman, Federico Javier Fernández, Diego Garbervetsky, and Sergio Yovine. 2008. Parametric prediction of heap memory requirements. In *International Symposium on Memory Management*. 141–150. https://dl.acm.org/doi/10.1145/1375634.1375655

Víctor A. Braberman, Diego Garbervetsky, and Sergio Yovine. 2006. A Static Analysis for Synthesizing Parametric Specifications of Dynamic Memory Consumption. *Journal of Object Technology* 5, 5 (2006), 31–58. https://www.jot.fm/issues/issue_2006_06/article2.pdf

Stephen Brookes. 2007. A semantics for concurrent separation logic. *Theoretical Computer Science* 375, 1–3 (2007), 227–270. https://doi.org/10.1016/j.tcs.2006.12.034

Stephen Brookes and Peter W. O'Hearn. 2016. Concurrent separation logic. *SIGLOG News* 3, 3 (2016), 47–65. http://siglog.hosting.acm.org/wp-content/uploads/2016/07/siglog_news_9.pdf

Quentin Carbonneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-end verification of stack-space bounds for C programs. In *Programming Language Design and Implementation (PLDI)*. 270–281. http://flint.cs.yale.edu/flint/publications/veristack.pdf

Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional certified resource bounds. In *Programming Language Design and Implementation (PLDI)*. 467–478. https://www.cs.yale.edu/homes/hoffmann/papers/amort_imp15.pdf

Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning* 62, 3 (March 2019), 331–365. http://cambium.inria.fr/~fpottier/publis/chargueraud-pottier-uf-sltc.pdf

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 519–538. https://doi.org/10.1145/1094811.1094852

Wei-Ngan Chin, Huu Hai Nguyen, Corneliu Popeea, and Shengchao Qin. 2008. Analysing memory resource bounds for low-level programs. In *International Symposium on Memory Management*. 151–160. https://www7.in.tum.de/~popeea/research/memory.ismm08.pdf

Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin C. Rinard. 2005. Memory Usage Verification for OO Programs. In *Static Analysis Symposium (SAS) (Lecture Notes in Computer Science, Vol. 3672)*. Springer, 70–86. https://doi.org/10.1007/11547662_7

Brent Christian and Stuart Marks. 2021. JEP 421: Deprecate Finalization for Removal. https://openjdk.org/jeps/421

Edmund M Clarke, Thomas A Henzinger, Helmut Veith, Roderick Bloem, et al. 2018. *Handbook of model checking*. Vol. 10. Springer.

George E. Collins. 1960. A method for overlapping and erasure of lists. *Commun. ACM* 3, 12 (1960), 655–657. https://doi.org/10.1145/367487.367501

William R. Cook. 2009. On understanding data abstraction, revisited. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 557–572. http://www.cs.utexas.edu/~wcook/Drafts/2009/essay.pdf

Karl Crary and Stephanie Weirich. 2000. Resource bound certification. In *Principles of Programming Languages (POPL)*. 184–198. http://www.cs.cornell.edu/talc/papers/resource_bound/res.pdf

Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 2014. TaDA: A Logic for Time and Data Abstraction. In *European Conference on Object-Oriented Programming (ECOOP) (Lecture Notes in Computer Science, Vol. 8586)*, Richard E. Jones (Ed.). Springer, 207–231. https://vtss.doc.ic.ac.uk/publications/daRochaPinto2014TaDA.pdf

N.G. de Bruijn. 1994. The Mathematical Language Automath, its Usage, and Some of its Extensions. In *Selected Papers on Automath*, R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 133. Elsevier, 73–100. https://doi.org/10.1016/S0049-237X(08)70200-3

Paulo Emílio de Vilhena and François Pottier. 2021. A Separation Logic for Effect Handlers. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021). http://cambium.inria.fr/~fpottier/publis/de-vilhena-pottier-sleh.pdf

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *Principles of Programming Languages (POPL)*. 287–300. http://cs.au.dk/~birke/papers/views.pdf

Kevin Donnelly, J. J. Hallett, and Assaf J. Kfoury. 2006. Formal semantics of weak references. In *International Symposium on Memory Management*. 126–137. https://doi.org/10.1145/1133956.1133974

Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. 2014. The Matter of Heartbleed. In *Internet Measurement Conference (IMC)*. https://doi.org/10.1145/2663716.2663755

EATCS. 2016. Gödel Prize Citation. https://eatcs.org/index.php/component/content/article/1-news/2280-2016-godel-prize.

Martin Elsman. 2023. Garbage-Collection Safety for Region-Based Type-Polymorphic Programs. *Proceedings of the ACM on Programming Languages* 7, PLDI (jun 2023). https://doi.org/10.1145/3591229

Marc Feeley. 1993. Polling Efficiently on Stock Hardware. In *Functional Programming Languages and Computer Architecture (FPCA)*. 179–190. https://doi.org/10.1145/165180.165205

Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science* 103, 2 (1992), 235–271. https://www2.ccs.neu.edu/racket/pubs/tcs92-fh.pdf

Christian Ferdinand, Reinhold Heckmann, and Bärbel Franzen. 2006. Static Memory and Timing Analysis of Embedded Systems Code. In *European Symposium on Verification and Validation of Software Systems*. https://www.absint.com/aiT_StackAnalyzer.pdf

Jean-Christophe Filliâtre. 2011. Deductive software verification. *Software Tools for Technology Transfer* 13, 5 (2011), 397–403. https://doi.org/10.1007/s10009-011-0211-0

R. W. Floyd. 1967. Assigning meanings to programs. In *Mathematical Aspects of Computer Science (Proceedings of Symposia in Applied Mathematics, Vol. 19)*. American Mathematical Society, 19–32. https://people.eecs.berkeley.edu/~necula/Papers/FloydMeaning.pdf

Matthew Fluet, Greg Morrisett, and Amal Ahmed. 2006. Linear Regions Are All You Need. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 3924)*. Springer, 7–21. http://ttic.uchicago.edu/~fluet/research/substruct-regions/ESOP06/esop06.pdf

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2018. ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency. In *Logic in Computer Science (LICS)*. 442–451. https://iris-project.org/pdfs/2018-lics-reloc-final.pdf

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Logical Methods in Computer Science* 17, 3 (2021). https://arxiv.org/abs/2006.13635v3

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *International Conference on Concurrency Theory (CONCUR) (Lecture Notes in Computer Science, Vol. 6269)*. Springer, 388–402. https://doi.org/10.1007/978-3-642-15375-4_27

Diego Garbervetsky, Sergio Yovine, Víctor A. Braberman, Martín Rouaux, and Alejandro Taboada. 2011. Quantitative dynamic-memory analysis for Java. *Concurrency and Computation Practice and Experience* 23, 14 (2011), 1665–1678. https://doi.org/10.1002/cpe.1656

Alejandro Gómez-Londoño and Magnus O. Myreen. 2021. A flat reachability-based measure for CakeML's cost semantics. In *Implementation of Functional Languages (IFL)*. 1–9. https://doi.org/10.1145/3544885.3544887

Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. 2020. Do you have space for dessert? A verified space cost semantics for CakeML programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 204:1–204:29. https://doi.org/10.1145/3428272

Georges Gonthier et al. 2008. Formal proof–the four-color theorem. *Notices of the AMS* 55, 11 (2008), 1382–1393. https://www.ams.org/notices/200811/tx081101382p.pdf

Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, et al. 2013. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving (ITP)*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer, 163–179. https://doi.org/10.1007/978-3-642-39634-2_14

Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. 2007. Local Reasoning for Storable Locks and Threads. In *Asian Symposium on Programming Languages and Systems (APLAS) (Lecture Notes in Computer Science, Vol. 4807)*. Springer, 19–37. http://dx.doi.org/10.1007/978-3-540-76637-7_3

Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. 2013. Verifying Concurrent Memory Reclamation Algorithms with Grace. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 249–269. https://software.imdea.org/~gotsman/papers/recycling-esop13.pdf

Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut A. Acar, and Matthew Fluet. 2018. Hierarchical memory management for mutable state. In *Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/3178487.3178494

Armaël Guéneau. 2019. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs*. Ph. D. Dissertation. Université de Paris. https://tel.archives-ouvertes.fr/tel-02437532

Theodore Hailperin. 1986. Formalization of Boole's Logic. In *Boole's Logic and Probability*. Studies in Logic and the Foundations of Mathematics, Vol. 85. Elsevier, 135–172. https://www.sciencedirect.com/science/article/pii/S0049237X08702477

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *International Conference on Distributed Computing (DISC)*. Springer, 300–314. https://www.cl.cam.ac.uk/research/srg/netos/papers/2001-caslists.pdf

Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-word Compare-and-Swap Operation. In *Distributed Computing*, Dahlia Malkhi (Ed.). Springer, 265–279. https://www.cl.cam.ac.uk/research/srg/netos/papers/2002-casn.pdf

Barry Hayes. 1997. Ephemerons: A New Finalization Mechanism. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Mary E. S. Loomis, Toby Bloom, and A. Michael Berman (Eds.). ACM, 176–183. https://doi.org/10.1145/263698.263733

Guanhua He, Shengchao Qin, Chenguang Luo, and Wei-Ngan Chin. 2009. Memory Usage Verification Using Hip/Sleek. In *Automated Technology for Verification and Analysis (ATVA) (Lecture Notes in Computer Science, Vol. 5799)*. Springer, 166–181. https://dro.dur.ac.uk/6241/

Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc.

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580. http://doi.acm.org/10.1145/363235.363259

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012a. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems* 34, 3 (2012), 14:1–14:62. https://www.cs.cmu.edu/~janh/assets/pdf/HoffmannAH10.pdf

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012b. Resource Aware ML. In *Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 7358)*. Springer, 781–786. http://dx.doi.org/10.1007/978-3-642-31424-7_64

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *Principles of Programming Languages (POPL)*. 359–373. http://www.cs.cmu.edu/~janh/papers/HoffmannDW17.pdf

Jan Hoffmann and Steffen Jost. 2022. Two decades of automatic amortized resource analysis. *Mathematical Structures in Computer Science* 32, 6 (2022), 729–759. https://doi.org/10.1017/S0960129521000487

Martin Hofmann. 1999. Linear Types and Non-Size-Increasing Polynomial Time Computation. In *Logic in Computer Science (LICS)*. 464–473. https://doi.org/10.1109/LICS.1999.782641

Martin Hofmann. 2000. A type system for bounded space and functional in-place update. *Nordic Journal of Computing* 7, 4 (2000), 258–289. http://www.dcs.ed.ac.uk/home/mxh/nordic.ps.gz

Martin Hofmann. 2003. Linear types and non-size-increasing polynomial time computation. *Information and Computation* 183, 1 (2003), 57–85. https://doi.org/10.1016/S0890-5401(03)00009-9

Martin Hofmann and Steffen Jost. 2003. Static prediction of heap space usage for first-order functional programs. In *Principles of Programming Languages (POPL)*. 185–197. http://www2.tcs.ifi.lmu.de/~jost/research/POPL_2003_Jost_Hofmann.pdf

Martin Hofmann and Steffen Jost. 2006. Type-Based Amortised Heap-Space Analysis. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 3924)*. Springer, 22–37. https://www2.tcs.ifi.lmu.de/~jost/research/hofmann_jost_esop06_postfinal.pdf

Martin Hofmann and Dulma Rodriguez. 2009. Efficient Type-Checking for Amortised Heap-Space Analysis. In *Computer Science Logic (Lecture Notes in Computer Science, Vol. 5771)*. Springer, 317–331. https://doi.org/10.1007/978-3-642-04027-6_24

Martin Hofmann and Dulma Rodriguez. 2013. Automatic Type Inference for Amortised Heap-Space Analysis. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 593–613. https://doi.org/10.1007/978-3-642-37036-6_32

Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2011. Separation Logic in the Presence of Garbage Collection. In *Logic in Computer Science (LICS)*. 247–256. http://people.mpi-sws.org/~dreyer/papers/gcsl/paper.pdf

Sadiq Jaffer. 2021. OCaml Compiler Pull Request 10462: Add [@poll error] attribute. https://github.com/ocaml/ocaml/pull/10462.

Richard Jones, Antony Hosking, and Eliot Moss. 2012. *The Garbage Collection Handbook*. Chapman and Hall/CRC. https://gchandbook.org/

Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. 2023. Modular Verification of Safe Memory Reclamation in Concurrent Separation Logic. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023), 828–856. https://doi.org/10.1145/3622827

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 66:1–66:34. https://people.mpi-sws.org/~dreyer/papers/rustbelt/paper.pdf

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 2020. The Future is Ours: Prophecy Variables in Separation Logic. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 45:1–45:32. https://plv.mpi-sws.org/prophecies/paper.pdf

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: monoids and invariants as an orthogonal basis for concurrent reasoning. In *Principles of Programming Languages (POPL)*. 637–650. http://plv.mpi-sws.org/iris/paper.pdf

David M. Kahn and Jan Hoffmann. 2021. Automatic amortized resource analysis with the quantum physicist's method. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–29. https://doi.org/10.1145/3473581

Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. 2017. Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris. In *European Conference on Object-Oriented Programming (ECOOP)*. 17:1–17:29. https://people.mpi-sws.org/~dreyer/papers/iris-weak/paper.pdf

Vesa Karvonen. 2023a. Saturn Issue 63: Michael_scott_queue space safety. https://github.com/ocaml-multicore/saturn/issues/63.

Vesa Karvonen. 2023b. Using [@poll error] attribute to implement systhread safe data structures. https://discuss.ocaml.org/t/using-poll-error-attribute-to-implement-systhread-safe-data-structures/12804

Ioannis T. Kassios and Eleftherios Kritikos. 2013. A Discipline for Program Verification Based on Backpointers and Its Use in Observational Disjointness. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 7792)*. Springer, 149–168. https://doi.org/10.1007/978-3-642-37036-6_10

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 207–220. https://doi.org/10.1145/1629575.1629596

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 77:1–77:30. https://doi.org/10.1145/3236772

Robert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Principles of Programming Languages (POPL)*. http://cs.au.dk/~birke/papers/ipm-conf.pdf

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *Principles of Programming Languages (POPL)*. 179–192. https://cakeml.org/popl14.pdf

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (1979). https://doi.org/10.1109/TC.1979.1675439

Rich Lander. 2015. Announcing .NET Framework 4.6. https://devblogs.microsoft.com/dotnet/announcing-net-framework-4-6/.

Peter J. Landin. 1964. The Mechanical Evaluation of Expressions. *Computer Journal* 6, 4 (Jan. 1964), 308–320. https://doi.org/10.1093/comjnl/6.4.308

Gérard Le Lann. 1997. An analysis of the Ariane 5 flight 501 failure-a system engineering perspective. In *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems*. 339–346. https://doi.org/10.1109/ECBS.1997.581900

Xavier Leroy. 2024. The CompCert C compiler. http://compcert.org/.

Nancy G. Leveson and Clark S. Turner. 1993. An investigation of the Therac-25 accidents. *Computer* 26, 7 (1993), 18–41. https://doi.org/10.1109/MC.1993.274940

Yi Lin, Kunshan Wang, Stephen M. Blackburn, Antony L. Hosking, and Michael Norrish. 2015. Stop and go: understanding yieldpoint behavior. In *International Symposium on Memory Management*. 70–80. https://doi.org/10.1145/2754169.2754187

Daniel Loeb. 1992. Sets with a negative number of elements. *Advances in Mathematics* 91, 1 (1992), 64–74. https://www.sciencedirect.com/science/article/pii/0001870892900119

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. 2023. FP^2: Fully in-Place Functional Programming. *Proceedings of the ACM on Programming Languages* 7, ICFP (Aug. 2023), 275–304. https://doi.org/10.1145/3607840

Julian Mackay, Susan Eisenbach, James Noble, and Sophia Drossopoulou. 2022. Necessity specifications for robustness. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (oct 2022). https://doi.org/10.1145/3563317

Jean-Marie Madiot and François Pottier. 2022. A Separation Logic for Heap Space under Garbage Collection. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 718–747. http://cambium.inria.fr/~fpottier/publis/madiot-pottier-diamonds-2022.pdf

John McCarthy. 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Commun. ACM* 3, 4 (April 1960), 184–195. https://doi.org/10.1145/367177.367199

Paul McKenney, Michael Wong, Maged M. Michael, Andrew Hunter, Daisy Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, Erik Rigtorp, Tomasz Kamiński, Olivier Giroux, David Vernet, and Timur Doumler. 2023. Read-Copy Update (RCU). https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2545r4.pdf

Paul E. McKenney. 2004. *Exploiting deferred destruction: an analysis of read-copy-update techniques in operating system kernels*. Ph. D. Dissertation. Oregon Health & Science University. http://www.rdrop.com/~paulmck/RCU/RCUdissertation.2004.07.14e1.pdf

Roland Meyer and Sebastian Wolff. 2019. Decoupling lock-free data structures from memory reclamation for static analysis. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019). https://doi.org/10.1145/3290371

Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

Maged M. Michael and Michael L. Scott. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Symposium on Principles of Distributed Computing (PODC)*. ACM, 267–275. https://doi.org/10.1145/248052.248106

Maged M. Michael, Michael Wong, Paul McKenney, Andrew Hunter, Daisy Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, and Mathias Stearn. 2023. Hazard Pointers for C++26. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2530r3.pdf

Microsoft. 2024. Documentation of the GC class of the .NET 8.0 framework. https://learn.microsoft.com/en-us/dotnet/api/system.gc?view=net-8.0

Alexandre Moine. 2024. Formal Verification of Heap Space Bounds under Garbage Collection - Mechnization. https://github.com/nobrakal/irisfit The last commit at the time of writing is archived at https://archive.softwareheritage.org/swh:1:snp:8c4b8d3adc356c2b9c1f370c2c4e4f191f9eeff0;origin=https://github.com/nobrakal/irisfit.

Alexandre Moine, Arthur Charguéraud, and François Pottier. 2023. A High-Level Separation Logic for Heap Space under Garbage Collection. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 718–747. https://doi.org/10.1145/3571218

Alexandre Moine, Sam Westrick, and Stephanie Balzer. 2024. DisLog: A Separation Logic for Disentanglement. *Proceedings of the ACM on Programming Languages* 8, POPL, Article 11 (jan 2024). https://doi.org/10.1145/3632853

Francis L. Morris and Clifford B. Jones. 1984. An Early Program Proof by Alan Turing. *Annals of the History of Computing* 6, 2 (1984), 139–143. https://doi.org/10.1109/MAHC.1984.10017

J. Gregory Morrisett, Matthias Felleisen, and Robert Harper. 1995. Abstract Models of Memory Management. In *Functional Programming Languages and Computer Architecture (FPCA)*. 66–77. https://www.cs.cmu.edu/~rwh/papers/gc/fpca95.pdf

Ike Mulder, Łukasz Czajka, and Robbert Krebbers. 2023. Beyond Backtracking: Connections in Fine-Grained Concurrent Separation Logic. *Proceedings of the ACM on Programming Languages* 7, PLDI (jun 2023). https://doi.org/10.1145/3591275

Ike Mulder and Robbert Krebbers. 2023. Proof Automation for Linearizability in Separation Logic. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (apr 2023). https://doi.org/10.1145/3586043

Ike Mulder, Robbert Krebbers, and Herman Geuvers. 2022. Diaframe: automated verification of fine-grained concurrent programs in Iris. In *Programming Language Design and Implementation (PLDI)*. 809–824. https://doi.org/10.1145/3519939.3523432

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2019. Time credits and time receipts in Iris. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 11423)*. Springer, 1–27. http://cambium.inria.fr/~fpottier/publis/mevel-jourdan-pottier-time-in-iris-2019.pdf

Glen Mével, Jacques-Henri Jourdan, and François Pottier. 2020. Cosmo: A Concurrent Separation Logic for Multicore OCaml. *Proceedings of the ACM on Programming Languages* 4, ICFP (June 2020). http://cambium.inria.fr/~fpottier/publis/mevel-jourdan-pottier-cosmo-2020.pdf

Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. 2007. Automated Verification of Shape and Size Properties Via Separation Logic. In *Verification, Model Checking and Abstract Interpretation (VMCAI) (Lecture Notes in Computer Science, Vol. 4349)*. Springer, 251–266. https://dro.dur.ac.uk/6213/

Yue Niu and Jan Hoffmann. 2018. Automatic Space Bound Analysis for Functional Programs with Garbage Collection. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR) (EPiC Series in Computing, Vol. 57)*. 543–563. https://easychair.org/publications/paper/dcnD

Peter W. O'Hearn. 2007. Resources, Concurrency and Local Reasoning. *Theoretical Computer Science* 375, 1–3 (May 2007), 271–307. http://www.cs.ucl.ac.uk/staff/p.ohearn/papers/concurrency.pdf

Peter W. O'Hearn. 2019. Separation logic. *Commun. ACM* 62, 2 (2019), 86–95. https://doi.org/10.1145/3211968

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic (Lecture Notes in Computer Science, Vol. 2142)*. Springer, 1–19. http://www0.cs.ucl.ac.uk/staff/p.ohearn/papers/localreasoning.pdf

Zoe Paraskevopoulou and Andrew W. Appel. 2019. Closure conversion is safe for space. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 83:1–83:29. https://doi.org/10.1145/3341687

Matthew Parkinson. 2010. The Next 700 Separation Logics. In *Verified Software: Theories, Tools, Experiments*, Gary T. Leavens, Peter O'Hearn, and Sriram K. Rajamani (Eds.). Springer, 169–182. https://doi.org/10.1007/978-3-642-15057-9_12

Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. 2007. Modular verification of a non-blocking stack. In *Principles of Programming Languages (POPL)*. 297–302. https://doi.org/10.1145/1190216.1190261

Simon Peyton Jones, Simon Marlow, and Conal Elliott. 1999. Stretching the storage manager: weak pointers and stable names in Haskell. In *Implementation of Functional Languages (IFL) (LNCS)*. Springer. https://www.microsoft.com/en-us/research/publication/stretching-the-storage-manager-weak-pointers-and-stable-names-in-haskell/

Benjamin C. Pierce. 2002. *Types and Programming Languages*. MIT Press.

Alexandre Pilkiewicz and François Pottier. 2011. The essence of monotonic state. In *Types in Language Design and Implementation (TLDI)*. http://cambium.inria.fr/~fpottier/publis/pilkiewicz-pottier-monotonicity.pdf

Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 12225)*. Springer, 225–252. https://plv.mpi-sws.org/ISL/

Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy E. Blelloch. 2016. Hierarchical memory management for parallel programs. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2951913.2951935

John C. Reynolds. 1975. *User-defined Types and Procedural Data Structures as Complementary Approaches to Data Abstraction*. Technical Report 1278. Carnegie Mellon University. http://repository.cmu.edu/compsci/1278/

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*. 55–74. http://www.cs.cmu.edu/~jcr/seplogic.pdf

Xavier Rival and Kwangkeun Yi. 2020. *Introduction to static analysis: an abstract interpretation perspective*. Mit Press. https://mitpress.mit.edu/9780262043410/introduction-to-static-analysis/

K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting Parallelism onto OCaml. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 113:1–113:30. https://doi.org/10.1145/3408995

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *Journal of Automated Reasoning* (Feb. 2020). https://doi.org/10.1007/s10817-019-09540-0

Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Botsch Nielsen, Nicolas Tabareau, and Théo Winterhalter. 2023. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. (April 2023). https://inria.hal.science/hal-04077552 Unpublished.

Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2022. Later credits: resourceful reasoning for the later modality. *Proceedings of the ACM on Programming Languages* 6, ICFP (2022), 283–311. https://doi.org/10.1145/3547631

Kasper Svendsen and Lars Birkedal. 2014. Impredicative Concurrent Abstract Predicates. In *European Symposium on Programming (ESOP) (Lecture Notes in Computer Science, Vol. 8410)*. Springer, 149–168. http://cs.au.dk/~birke/papers/icap-conf.pdf

The Agda Development Team. 2024. The Agda Programing Language. https://wiki.portal.chalmers.se/agda/pmwiki.php

The Coq Development Team. 2024. The Coq Proof Assistant. http://coq.inria.fr/

The Coq-std++ Team. 2023. Coq-std++: An extended standard library for Coq. https://gitlab.mpi-sws.org/iris/stdpp

The HOL Development Team. 2024. The HOL Theorem Prover. https://hol-theorem-prover.org/

The Iris Developement Team. 2024. The Iris Project. https://gitlab.mpi-sws.org/iris/iris/ The last commit at the time of writing is archived at https://archive.softwareheritage.org/swh:1:snp:a3b01e150fc67626d9c4082c0b205863017382c6;origin=https://gitlab.mpi-sws.org/iris/iris.

The Lean Development Team. 2024. The Lean Theorem Prover. https://leanprover-community.github.io/

The saturn Developement Team. 2024. Michael and Scott's queue implementation in saturn. https://archive.softwareheritage.org/swh:1:cnt:2c60ec92098f9f95e72debec5a85155e8e0bb28e;origin=https://github.com/ocaml-multicore/saturn;path=/src_lockfree/michael_scott_queue.ml

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2024. A Logical Approach to Type Soundness. (2024). https://iris-project.org/pdfs/2024-jacm-logical-type-soundness.pdf To appear in Jounral of the ACM.

Bogdan Tofan, Gerhard Schellhorn, and Wolfgang Reif. 2011. Formal Verification of a Lock-Free Stack with Hazard Pointers. In *Theoretical Aspects of Computing (ICTAC) (Lecture Notes in Computer Science, Vol. 6916)*. Springer, 239–255. https://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/docId/55403

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher-Order and Symbolic Computation* 17, 3 (Sept. 2004), 245–265. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and Computation* 132, 2 (1997), 109–176. http://www.irisa.fr/prive/talpin/papers/ic97.pdf

R. Kent Treiber. 1986. Systems programming: Coping with parallelism. https://dominoweb.draco.res.ibm.com/reports/rj5118.pdf

Alan Turing. 1949. Checking a Large Routine. *Report of a Conference on High Speed Automatic Calculating Machines* (1949), 67–69. See the corrected and commented version by Francis L. Morris and Clifford B. Jones: An Early Program Proof by Alan Turing. Annals of the History of Computing 6, 2 (1984).

U.S.-Canada Power System Outage Task Force. 2004. Final report on the August 14, 2003 blackout in the United States and Canada : causes and recommendations. https://www3.epa.gov/region1/npdes/merrimackstation/pdfs/ar/AR-1165.pdf

Viktor Vafeiadis and Matthew Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *International Conference on Concurrency Theory (CONCUR)*, Luís Caires and Vasco T. Vasconcelos (Eds.). Springer, 256–271. https://people.mpi-sws.org/~viktor/papers/concur2007-marriage.pdf

Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue. In *Certified Programs and Proofs (CPP)*. 76–90. https://cs.au.dk/~birke/papers/2021-ms-queue-final.pdf

Sam Westrick, Jatin Arora, and Umut A. Acar. 2022. Entanglement Detection with Near-Zero Cost. *Proceedings of the ACM on Programming Languages* 6, ICFP (aug 2022). https://doi.org/10.1145/3547646

Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2020. Disentanglement in Nested-Parallel Programs. *Proceedings of the ACM on Programming Languages* 4, POPL (jan 2020). https://doi.org/10.1145/3371115

Hassler Whitney. 1933. Characteristic Functions and the Algebra of Logic. *Annals of Mathematics* 34, 3 (1933), 405–414. http://www.jstor.org/stable/1968168