THÈSE

pour l'obtention du titre de

DOCTEUR

de l'

UNIVERSITÉ PARIS. DIDEROT (PARIS 7)

École doctorale 386 — Sciences Mathématiques de Paris-Centre

spécialité

INFORMATIQUE

Les objets en C++ : sémantique formelle mécanisée et compilation vérifiée

Mechanized Formal Semantics and Verified Compilation for C++ Objects

---

par

Tahina RAMANANANDRO

soutenue le

10 janvier 2012

à l'

École normale supérieure

devant le Jury composé de

Résumé

C++ est un des langages de programmation les plus utilisés en pratique, y compris pour le logiciel embarqué critique. C'est pourquoi la vérification de programmes écrits en C++ devient intéressante, en particulier via l'utilisation de méthodes formelles. Pour cela, il est nécessaire de se fonder sur une sémantique formelle de C++. De plus, une telle sémantique formelle peut être validée en la prenant comme base pour la spécification et la preuve d'un compilateur C++ réaliste, afin d'établir la confiance dans les techniques usuelles des compilateurs C++. Dans cette thèse, nous nous focalisons sur le modèle objet de C++.

Nous proposons une sémantique formelle de l'héritage multiple en C++ comprenant les structures imbriquées à la C, sur laquelle s'appuie notre étude de la représentation concrète des objets avec optimisations des bases vides, à travers des conditions suffisantes que nous prouvons correctes vis-à-vis des accès aux champs et des opérations polymorphes. Puis nous spécifions un algorithme de représentation en mémoire fondé sur l'ABI pour Itanium, et une extension de cet algorithme avec optimisations des champs vides, et nous prouvons qu'ils satisfont nos conditions. Nous obtenons alors un compilateur vérifié et réaliste d'un sous-ensemble de C++ vers un langage à trois adresses et accès mémoire de bas niveau.

Rajoutant à notre sémantique la construction et la destruction d'objets, nous étudions leurs interactions avec l'héritage multiple. Cela nous permet de formaliser la gestion de ressources, notamment le principe RAII (*resource acquisition is initialization*) via l'ordre de construction et destruction des sous-objets. Nous étudions aussi les effets sur les opérations polymorphes telles que la sélection de fonction virtuelle pendant la construction et la destruction, en généralisant la notion de type dynamique. Nous obtenons alors un compilateur vérifié pour notre sémantique étendue, notamment en prouvant la correction de l'implémentation des changements de types dynamiques. Toutes nos spécifications et preuves sont formalisées en Coq.

Abstract

C++ is one of the most widely used programming languages in practice, including for embedded critical software. Thus, it becomes interesting to apply formal methods to programs written in C++. To this end, it is necessary to rely on a formal semantics of C++. Moreover, such a formal semantics can be validated as a basis to the specification and proof of a verified realistic compiler for C++ to gain confidence in the implementation techniques of mainstream C++ compilers. In this thesis, we focus on the C++ object model.

We formally specify C++ multiple inheritance with C-style embedded structures, leading us to study the concrete representation of objects with empty base optimizations. We propose a set of sufficient layout conditions, and we show that they are sound with respect to field accesses and polymorphic operations. We then specify a realistic layout algorithm based on the Common Vendor ABI for Itanium, and an extension performing empty member optimizations, and we prove that they satisfy our conditions. We obtain a verified realistic compiler from a subset of C++ to a 3-address language with low-level memory accesses.

Extending our semantics with object construction and destruction, we study their intrications with multiple inheritance. This leads us to formalize resource management, namely "resource acquisition is initialization" through the subobject construction and destruction order. We also study the impact on polymorphic operations such as virtual function dispatch during construction and destruction, by generalizing the notion of dynamic type. We obtain a verified compiler for our extended semantics, in particular by verifying the implementation of dynamic type changes. All our specifications and proofs are carried out with Coq.

Table des matières

Ordre ! Discipline ! Rigueur !

Remerciements

Je tiens à remercier en tout premier lieu mon maître de thèse Xavier Leroy. C'est lui qui m'a transmis tous ses éléments de rigueur scientifique, dans la droite lignée de mes anciens professeurs d'informatique et de mathématiques du lycée Kléber de Strasbourg (notamment M. Pister). C'est lui qui, conformément aux dires de mon prédécesseur, m'a donné toutes les clés pour transformer le plomb en or, notamment en matière de rédaction. C'est lui, enfin, qui m'a donné le goût de l'ambition en me montrant ma valeur réelle et en me permettant de publier, avec succès, à l'une des conférences les plus prestigieuses dans notre domaine.

I am very thankful to my reviewers Thomas Jensen and Michael Norrish. Their reviews helped me greatly and significantly improve the quality of this thesis. Their attention to the slightest detail to help the reader handle this quite long document really impressed me, in addition to their courage and enthusiasm, as well as their expertise on mechanically formalizing real-world standards, which also guided me one step beyond CompCert.

Je remercie les membres de l'équipe Parasol (Texas A&M University). En premier lieu, Gabriel Dos Reis m'a ouvert les yeux sur le monde réel de C++, et m'a donné tout le courage pour l'affronter, grâce à son optimisme sans limites qui m'a permis de construire ce « pont » entre la vérification formelle d'une part, et le développement de compilateurs réalistes d'autre part, deux mondes en tension permanente autour de la conception du Standard C++. À ce titre, je lui tire mon chapeau. *I naturally bow to Bjarne Stroustrup and his all-time fellow Lawrence Rauchberger, for they utmostly honoured me by the confidence and the hopes they granted me to formalize C++ multiple inheritance. Hopefully I deserve their attention.*

Je remercie l'ensemble de mes camarades de laboratoire des équipes Gallium, Moscova, Sanskrit et Contraintes, pour ces années fructueuses passées ensemble. Notamment, Sandrine Blazy, Zaynah Dargaye, Damien Doligez, Alain Frisch, Gérard Huet, Benoît Razet et Jean-Baptiste Tristan m'ont guidé tout au long de mes (petits ou grands) pas dans le monde de la sémantique des langages de programmation. Je n'oublie pas Arthur Charguéraud, pour son inestimable expérience de Coq ; ni Nicolas Pouillard, Luc Maranget, Jean-Jacques Lévy ou Sylvain Soliman, qui tous ensemble avons tant de fois refait le monde dans le prolongement direct de ma thèse, ce qui a eu des impacts non négligeables sur ceux-ci (ma thèse d'abord, le monde ensuite).

Je remercie Xavier Clerc, Steven Gay, Thierry Martinez, Gabriel Scherer, et également Denise Maurice (de l'équipe Secret), qui m'ont apporté une aide conséquente pour assurer la clarté du présent tapuscrit (notamment le tutoriel), jusque dans la critique des exemples que j'ai choisis. Tous méritent une part des félicitations adressées par mes rapporteurs.

Je remercie l'équipe Marelle, notamment Philippe Audébaud, Yves Bertot, Laurent Théry et Laurence Rideau, qui m'ont initié à Coq, qui m'ont montré la voie. Je leur dois tout.

I wish to thank Mark Batty, with whom we shared our views on the practices of the C++ Standard Committee, to keep our works realistic, so that they may complete each other.

Je remercie chaleureusement, pour leur participation à mon jury de thèse qui a été un grand honneur pour moi, Roberto Di Cosmo et Claude Marché, mais aussi Patrick Cousot, qui a toujours su nous épauler, mes camarades et moi, tout au long de notre scolarité à Ulm et au-delà, jusqu'à ses interventions pour me permettre de soutenir au sein de l'École normale supérieure, là où tout commença en 2004, et là où tout finira désormais.

Je remercie également les autres membres de l'équipe Sémantique et Interprétation Abstraite, à Ulm, qui me recueillirent ainsi les bras grand ouverts alors que la fin approchait. Spécialement, je remercie Xavier Rival, qui, grâce à ses précieux conseils, m'a guidé pendant la finalisation de ma thèse et de ma soutenance. À ce titre, je remercie également le personnel administratif du Département d'Informatique de l'ENS : Michelle Angély, Lise-Marie Bivard, Joëlle Isnard et Valérie Mongiat, ainsi que Sylvia Imbert qui les a précédées ; mais aussi Régine Guittard, de l'école doctorale 386.

Je remercie toutes celles et tous ceux qui m'ont épaulé durant toutes ces années depuis 2004 et ma scolarité à Ulm jusqu'à ma soutenance. La liste serait bien trop longue, j'espère qu'on ne m'en voudra pas trop. Notamment le forum des élèves (et anciens) de l'ENS, qui m'a permis de me compter parmi les *geeks* après mes longues nuits blanches sous `jonque` ou passées à jouer à des parties endiablées d'Arcanoïd au sein dudit club après des soirées animées par les clubs Animescens, BD-thèque, Cirque, Jeux, Sporz, ou encore le Département Clandestin de Lagodéblatératique au sein du 3^{e} Rataud. L'enthousiasme de Sekhmet, Subbak et `a3_nm`, ainsi que de notre regretté Jacen Solo, a joué un rôle décisif dans mes choix. Tous, des 1988 aux 2011, m'ont donné la joie de jouer et troller pendant comme avant ma thèse, me rendant plus efficace malgré tous les efforts du club Inutile. Je remercie donc globalement tout le COF, et l'ENS Ulm en général, mon *alma mater*, qui m'a offert les plus belles années de ma vie.

Je remercie aussi le Raton-Laveur, qui m'a grandement aidé à me sevrer de mon besoin irrépressible de lignes de Coq, mais aussi à préparer mon pot de thèse depuis plusieurs semestres sous le haut patronage de ses goûteurs impitoyables, insatiables et parfois même insatisfiables.

Je remercie tous ceux-là, Ulmiens et Lyonnais, ainsi que d'autres encore que je n'ai pas cités ici, qui, par le jeu, les blagues nacszwxkes, jdmn et autres facéties aléatoires (ou pas), ont su préserver le sens de ma vie comme je préserve le sens des programmes.

Je remercie aussi mes amis malgaches d'Île-de-France, Rouen, Toulouse ou ailleurs, notamment Misa et les Rano, pour leur disponibilité pendant ces années de thèse.

Je remercie surtout mes oncles, tantes et cousins de Digne-les-Bains, Nancy, Cachan ou même Madagascar, qui m'ont sagement et efficacement accompagné pendant toutes mes années d'études. Aucune distance, euclidienne, SNCF ou autre, ne fut un obstacle pour eux.

Et enfin, surtout, je remercie mes parents. Leur indéfectible soutien (notamment culinaire) ne peut être décrit en un nombre fini de mots, et je voudrais leur dire combien je suis fier, et combien ils peuvent également être fiers, du résultat que nous obtenons désormais, et que constitue le présent tapuscrit, fruit de plus d'un quart de siècle d'éducation rigoureuse et raisonnée, toujours dans cet esprit de **progrès** qui est le leur et qui meut tout bon Malgache. *Misaotra indrindra anareo amin'ireo zavatra rehetra nataonareo ho ahy. Tsy ho adinoko mandrakizay.*

Vous allez tous me manquer désormais. C'est pourquoi c'est à vous toutes et tous, sans exception, que je dédie le présent tapuscrit.

Au revoir, et bonne continuation. *Veloma daholy, ary soava dia.*

Chapitre 0

Aperçu

This chapter is intended for French-speaking readers who cannot read English. English-speaking readers can skip it to the actual beginning of this thesis, at Part ◯ (p. 29)

Ce chapitre est destiné aux lecteurs francophones non-anglophones. Il constitue un résumé de la présente thèse, décrivant dans leur contexte nos contributions scientifiques et en présentant le bilan général.

0.1 Contexte

0.1.1 Vérification de logiciels

L'informatique est de plus en plus présente dans les systèmes dits *critiques*, tels que les transports (chemins de fer [67], aéronautique [55], aérospatiale), les instruments militaires, ou les dispositifs médicaux. Dans de tels domaines critiques, la moindre défaillance logicielle peut causer des dommages extrêmement coûteux, voire mortels. Considérons par exemple l'appareil de radiothérapie Therac-25 : entre 1985 et 1987, aux États-Unis et au Canada, les patients traités par cette machine ont subi des surdosages massifs de rayonnements ayant entraîné la mort d'au moins 6 d'entre eux [53]. Pour expliquer ces surdosages, il faut examiner les deux modes de fonctionnement de cet appareil : soit par de faibles radiations directement envoyées sur le patient, soit par de fortes radiations envoyées indirectement à travers une cible intermédiaire. En réalité, le logiciel embarqué dans l'appareil n'a pas pu activer correctement le bon mode, ce qui a conduit à ces surdosages. De telles surexpositions peuvent même résulter d'une mauvaise gestion des erreurs au niveau de composants logiciels considérés comme moins critiques tels que l'interface utilisateur mise à disposition de l'opérateur humain pour contrôler l'appareil. C'est ce qui s'est produit au Panama avec un autre appareil de radiothérapie entre 2000 et 2002 : l'interface utilisateur n'a pas contrôlé la validité des données entrées par les médecins, ce qui a entraîné la mort d'au moins 17 patients [18].

Heureusement, les défaillances logicielles ne sont pas systématiquement mortelles. Cependant, elles peuvent entraîner des pertes financières importantes, de l'ordre de 370 millions de dollars américains dans le cas de l'échec du vol inaugural d'Ariane 5 en 1996 : un dépassement de capacité dans un calcul d'entiers [28] a perturbé le fonctionnement du logiciel embarqué, entraînant la perte de la fusée et de sa charge, le vaisseau Cluster du programme commun Agence spatiale europénne – NASA, dont la perte a par la suite entraîné 4 années de retard dans le développement du projet.

Dans son étude [27], Dershowitz a établi une liste de défaillances logicielles parmi les plus coûteuses (en termes financiers et humains) de l'histoire du logiciel embarqué critique. Éviter de telles défaillances devient donc la motivation principale de la recherche d'un moyen d'établir la confiance dans le logiciel critique. L'approche habituellement adoptée dans l'industrie du logiciel embarqué critique inclut le *test* [46], qui consiste à faire tourner le logiciel dans des environnements de test fictifs pour découvrir des erreurs logicielles, ou *bogues*. L'industrie fait également appel à la *relecture de code manuelle* telle que l'inspection de Fagan [31]. En pratique, le test et la relecture de code se sont révélés relativement efficaces à ce jour : parmi les accidents mortels dans l'ère (actuelle) de l'aviation assistée par électronique, aucun n'est dû à une défaillance logicielle. Cet état de fait est rendu possible, en partie par les règlementations officielles telles que DO-178B [65] qui normalisent et rendent systématiques ces procédures de test et de relecture de code en vue de la certification de logiciels aéronautiques. Cependant, le temps et le prix de telles procédures de test et de relecture de code se révèlent extrêmement élevés. De surcroît, il est difficile de garantir que les tests couvrent tous les cas possibles. En outre, il est de plus en plus douteux que ces méthodes passent à l'échelle au regard de l'augmentation continue de la taille et de la complexité des systèmes critiques : alors que la taille du code embarqué dans un Airbus A320 approche les 10 mégaoctets, celle d'un A380 se chiffre en plusieurs centaines de mégaoctets [34].

Une alternative prometteuse ou complémentaire au test est la *vérification de logiciels* en faisant appel aux *méthodes formelles*, afin de réduire les coûts des procédures de test et d'accroître la confiance dans les programmes. La vérification de logiciels permet aux programmeurs d'établir des propriétés de haut niveau sur leurs programmes et ce *statiquement*, c'est-à-dire sans avoir à les exécuter effectivement. Parmi de telles propriétés de haut niveau, on peut vouloir établir l'absence d'erreurs à l'exécution, l'inaccessibilité d'états déclarés « impossibles », ou encore la correction fonctionnelle vis-à-vis d'une spécification donnée.

Les logiciels peuvent être vérifiés en utilisant un large éventail de méthodes formelles. Le *model-checking* [21] consiste en l'exploration énumérative ou symbolique de l'ensemble des états accessibles par toutes les exécutions du programme. Une telle exploration repose sur un modèle du programme, et est menée par des outils entièrement automatiques tels que Alloy [6, 44]. L'*interprétation abstraite* [23] calcule une approximation de l'ensemble des valeurs que peut prendre chaque variable du programme (par exemple, en considérant des intervalles d'entiers). De tels calculs sont menés par des outils dédiés appelés *analyseurs statiques* tels que Astrée [1], qui a été utilisé pour vérifier, de manière entièrement automatique, la sûreté des accès mémoire et des calculs flottants dans le code – écrit en langage C – des systèmes de contrôle de l'Airbus A340.

La *preuve de programmes par vérification déductive* est une méthode générale permettant aux programmeurs de prouver des propriétés de haut niveau sur leurs programmes en les annotant avec des formules logiques (préconditions, postconditions, invariants de boucles). À partir de ces annotations, un outil automatique, appelé *générateur de conditions de vérification*, tel que Frama-C [5] ou Why [8], génère des lemmes appelés *obligations de preuve*, telles que la preuve des propriétés de haut niveau sur le programme se ramène à la résolution des obligations de preuve en utilisant des *prouveurs automatiques de théorèmes* tels que Z3 [26], ou des *assistants de preuve interactive* tels que Coq [4].

Finalement, la *génération de logiciels vérifiés* consiste à créer automatiquement un programme à partir de preuves (manuelles ou automatiques) de théorèmes dans un modèle abstrait, par exemple en utilisant la méthode B dite de *raffinement* utilisée pour le système de signalisation Meteor SAET du métro 14 [67], ligne entièrement automatique du réseau de Paris. De manière similaire, le mécanisme d'*extraction* [51, 52] de Coq [4] génère automatiquement des programmes exécutables à partir de spécifications et preuves en Coq.

0.1.2 Sémantique formelle

Quelle que soit la méthode formelle utilisée, la vérification de logiciels nécessite de comprendre le sens exact des programmes. Pour cela, il est nécessaire de définir la *sémantique* des langages de programmation, c'est-à-dire le sens des expressions et autres tournures du langage. Le plus souvent, la sémantique d'un langage de programmation est définie par une description informelle, sous la forme d'un document de standardisation ou de spécification écrit en langue naturelle (cas de C, C++, Java, ECMAScript), ou même seulement par une implémentation de référence (cas de Perl, Ruby, Caml, Haskell).

De telles descriptions informelles ne sont pas des bases suffisantes pour assurer la correction des logiciels critiques, car elles peuvent faire l'objet d'ambiguïtés dans les interprétations, ou de comportements manquants car non spécifiés. Par conséquent, il est nécessaire de définir mathématiquement les spécifications afin qu'elles soient aussi précises que possible. Cela nous amène à introduire le concept de *sémantique formelle* d'un langage de programmation, fondée sur de solides bases mathématiques. Il existe plusieurs formalismes mathématiques [77] permettant d'établir la sémantique formelle d'un langage : la *sémantique axiomatique*, ajoutant à la logique mathématique des règles pour prouver les assertions sur l'état d'exécution du programme ; la *sémantique dénotationnelle* interprétant les expressions et autres tournures du langage par des objets mathématiques ; ou la *sémantique opérationnelle* définissant les différentes étapes possibles de l'exécution d'un programme par une relation de transition entre les états d'exécution.

De nombreuses sémantiques formelles existent, mais elles ne concernent que des langages abstraits, ou des sous-ensembles jouets de langages existants, souvent dans un but de recherche. Il est surprenant que de telles sémantiques formelles soient si peu courantes parmi les langages de programmation effectivement utilisés en pratique. En effet, un des rares exemples connus est celui du langage fonctionnel Standard ML [61], et ce pourrait être le seul. Cependant, la précision exhaustive des sémantiques formelles, qui permet d'éviter les ambiguïtés et les situations non spécifiées, fait de ces sémantiques des systèmes formels de grande taille, sur lesquels il devient difficile de raisonner à la main, et donc préférable de mécaniser les raisonnements avec des assistants de preuve tels que Coq [4]. Le défi PoplMark [13] soutient cette aspiration croissante à la mécanisation de la métathéorie des langages de programmation.

0.1.3 Compilation vérifiée

En général, les logiciels formellement vérifiés le sont au niveau de leur code source, lisible par un opérateur humain, mais non au niveau du code effectivement exécuté par l'ordinateur. Ce code est lui-même obtenu à partir du code source par un programme appelé *compilateur*, auquel il est donc également nécessaire de faire confiance : il ne doit pas changer le sens du programme source en produisant le programme compilé.

En effet, les bogues dans les compilateurs peuvent avoir de lourdes conséquences sur les programmes compilés. En 2011, peu après la sortie du langage Java 7, Apache et Oracle ont mis en garde leurs utilisateurs contre un bogue important dans le compilateur Java 7 fourni par Oracle [66]. Ce compilateur effectuait des optimisations qui introduisaient des erreurs dans la compilation de certaines boucles. De telles erreurs se sont répercutées dans de célèbres logiciels très utilisés en pratique tels que Apache Lucene Core [33]. En outre, les bogues dans les compilateurs ne sont pas rares, comme l'ont montré Yang et al. [86], qui ont trouvé, grâce à des tests aléatoires massifs, plus de 325 bogues inconnus dans des compilateurs C parmi les plus courants tels que GCC ou LLVM.

Pour éviter de tels bogues introduits par des compilateurs incorrects, il existe une solution, proposée et utilisée dans l'aéronautique, prescrite par les règlementations officielles DO-178B [65] : écrire un compilateur tel que la relecture manuelle simultanée du code compilé et du code source permet d'arguer que le sens du programme source n'a pas été changé par le compilateur. Cependant, une telle analyse est effectuée entièrement à la main et de manière empirique. En outre, une telle procédure empêche le compilateur d'effectuer des optimisations intelligentes sur le code source [15, §2.1]. De surcroît, cette méthode, utilisée en pratique pour la compilation de C vers un langage assembleur, est en revanche difficile à adapter pour un langage de plus haut niveau (tel qu'un langage orienté objet, ou un langage fonctionnel) : en effet, de tels langages présentent un modèle mémoire fort différent de celui du langage cible et de la machine elle-même. C'est pourquoi il est nécessaire de vérifier aussi le compilateur lui-même.

L'étude de Dave [24] résume les principaux travaux de recherche sur la vérification de compilateurs jusqu'en 2003. Parmi ces travaux, certains peuvent être considérés comme pionniers. Dès 1963, McCarthy [57] a souligné le besoin de vérifier les compilateurs. Il y apporta la première réponse partielle en prouvant, pour la première fois, la correction d'un compilateur pour des expressions arithmétiques [58]. Cependant, cette preuve était menée sur papier. Par la suite, Milner et Weyrauch [60] ont donc apporté une solution plus précise en prouvant un compilateur similaire, mais pour la première fois en utilisant un système logique mécanisé. En 1989, Moore [62] a réalisé la preuve mécanisée d'un compilateur plus réaliste à partir d'un langage de type assembleur.

Pour prouver la correction d'un compilateur, on peut le voir comme un programme ordinaire et prouver sa correction fonctionnelle vis-à-vis de la spécification suivante :

Étant donné un code source, si le compilateur produit un code compilé, alors toute spécification respectée par le code source est respectée par le code compilé.

Cependant, il existe une différence fondamentale entre un compilateur et un programme ordinaire : la spécification d'un compilateur inclut en réalité un moyen de décrire les comportements à la fois du code source et du code compilé. Plus exactement, donner une spécification à un compilateur nécessite de spécifier formellement les langages source et cible, c'est-à-dire d'en donner une syntaxe et une sémantique formelles. Ainsi, la vérification de compilateurs est une motivation supplémentaire à la formalisation de la sémantique des langages de programmation.

Parmi les exemples de compilateurs formellement vérifiés, citons Jinja [47], d'un sous-ensemble de Java vers le code-objet (*bytecode*) JVM, écrit et prouvé en Isabelle; et surtout CompCert [2, 50, 49], de C vers les assembleurs PowerPC, ARM et Intel x86, écrit et prouvé en Coq [4]. Ces deux compilateurs formellement vérifiés sont obtenus par extraction.

Notre travail est fondé sur le compilateur vérifié CompCert. CompCert est une chaîne de plusieurs passes de compilation, dont la plupart est formellement vérifiée. Certaines passes sont directement prouvées au moyen de preuves de théorèmes, tandis que d'autres sont prouvées par *validation vérifiée* [82] : leur algorithme effectif n'est pas lui-même prouvé, mais, pour chaque compilation, le code source et le résultat de la passe de compilation sont contrôlés par un validateur qui, lui, est formellement vérifié pour assurer qu'il n'accepte que des couples source et résultat ayant effectivement la même sémantique.

0.1.4 Le langage C++

Cette thèse traite de la sémantique formelle et de la compilation vérifiée d'un sous-ensemble du langage C++. C++ [30, 79, 40, 42, 43] est un langage orienté objet créé en 1981 par Bjarne Stroustrup sous le nom de « *C with classes* ». En effet, C++ était initialement conçu comme une extension de C avec un modèle objet essentiellement fondé sur les classes à la Simula. C++ offre un modèle objet exceptionnellement riche : héritage multiple partagé ou répété, sélection (*dispatch*) dynamique de méthodes, et transtypage (*cast*) dynamique. C++ combine donc de manière unique des outils pour la programmation système avec néanmoins de puissants mécanismes d'abstraction.

À première vue, C++ peut paraître complexe. Toutefois, il est l'un des langages de programmation les plus utilisés en pratique, y compris dans les domaines du logiciel embarqué critique, par exemple par des entreprises telles que Lockheed Martin. Cependant, de telles entreprises formulent des lignes de conduite [55] que leurs ingénieurs en programmation doivent suivre : ils doivent abandonner certaines fonctionnalités de C++ telles que les exceptions. De telles restrictions permettent d'arguer moralement que la sémantique des programmes ainsi écrits est bien définie. De plus, elles sont censées faciliter la relecture et l'analyse de code.

Les motivations de telles restrictions, cependant, ne sont que d'ordre grossièrement moral. C'est pourquoi notre but est de donner un cadre formel à une partie de ces restrictions en spécifiant formellement un sous-ensemble de C++ et en prouvant un compilateur vérifié pour ce sous-ensemble. La formalisation d'un tel sous-ensemble de C++ permettra de faire confiance non seulement au compilateur, mais aussi à la spécification du langage, en donnant un cadre formel plus fort que les standards C++ écrits en langue naturelle. Dans notre travail, nous nous focalisons sur le modèle objet de C++, incluant l'héritage multiple, la construction et destruction d'objets, et la gestion de ressources.

Dans le chapitre 2, nous proposons un tutoriel, certes informel, mais aussi complet que possible, sur le modèle objet de C++. Le chapitre 3 introduit les notations mathématiques formelles utilisées tout au long de cette thèse, y compris le formalisme utilisé pour décrire les sémantiques opérationnelles de nos langages. Dans notre travail, nous suivons une méthode de compilation vérifiée fondée sur la preuve de théorèmes à la CompCert, présentée formellement en Appendice.

0.2 Contributions scientifiques

Le but de notre travail est de formaliser la sémantique du modèle objet de C++, incluant l'héritage multiple et la construction et destruction d'objets, et de valider cette sémantique par un compilateur vérifié apportant la preuve de la correction des techniques habituelles utilisées par les compilateurs les plus courants. Nous avons mécanisé toutes nos sémantiques formelles et nos preuves à l'aide de l'assistant de preuve Coq [4]. L'intégralité de notre développement Coq est disponible en ligne [71]. Son architecture est décrite dans l'Appendice A. Ainsi, notre travail propose une justification formelle de notre affirmation suivante :

Thèse :
La confiance dans la sémantique et la compilation du modèle objet de C++ peut s'établir formellement.

Pour cela, nous traitons les deux problèmes que nous considérons comme les plus importants :

Partie I. La représentation concrète des objets en mémoire, ou *layout*, permettant les accès aux champs, les transtypages (*casts*) statiques et la sélection (*dispatch*) dynamique de méthodes en temps et accès mémoire constants, suivant les techniques habituelles des compilateurs les plus courants.

Partie II. La construction et destruction des objets : le mécanisme de construction est conçu pour établir, au moment de la création d'un objet, des invariants sur celui-ci en initialisant ses champs, et au besoin en acquérant des ressources. Inversement, sa destruction doit donc libérer ces ressources et retransformer cet objet en mémoire brute. Nous étudions les effets de l'héritage multiple en C++ sur ces mécanismes.

La Figure 1.1 (p. 37) résume les langages et passes de compilation traités dans notre travail.

0.2.1 Représentation concrète des objets en mémoire (*layout*)

Chapitre 4. Nous formalisons la sémantique de l'héritage multiple en C++, y compris l'héritage partagé et l'héritage répété, les accès aux champs d'un objet, les appels de fonctions virtuelles, les transtypages (*casts*) statiques et dynamiques. Notre travail étend le langage CoreC++ défini par Wasserrab et al. [85, 84] en formalisant les deux notions de sous-objets d'héritage et sous-objets d'agrégation dus aux structures imbriquées à la C : nous obtenons alors un langage que nous appelons s++, pour « CoreC++ avec structures imbriquées ».

Chapitre 5. Puis nous spécifions formellement une famille d'algorithmes permettant de représenter concrètement les objets C++ en mémoire. Nous donnons un ensemble de conditions suffisantes sur les résultats calculés par un tel algorithme. Nos conditions prennent en compte les *optimisations des bases vides* afin de réduire l'espace mémoire occupé par les sous-objets de type classe vide au sein d'un objet. Nous prouvons que nos conditions satisfont la « propriété de bonnes variables » et préservent l'identité des sous-objets, permettant ainsi de modéliser correctement les opérations orientées objet telles que les accès aux champs, les transtypages et les tests d'égalité entre pointeurs vers sous-objets.

Chapitre 6. Puis nous spécifions et prouvons formellement la correction de deux implémentations réalistes de représentation concrète des objets en C++. Le premier algorithme est inspiré de l'ABI pour Itanium [22], que GNU GCC a adapté à d'autres plateformes. Dans le second algorithme, nous proposons des optimisations supplémentaires pour les membres vides. En réalité, nous prouvons alors que ces deux algorithmes satisfont nos conditions suffisantes.

Chapitre 7. Finalement, pour tout algorithme satisfaisant nos conditions suffisantes, nous spécifions et prouvons formellement un compilateur de s++ vers un langage à trois adresses et accès mémoire de bas niveau, implémentant les opérations polymorphes (telles que le *dispatch* de méthodes virtuelles) à l'aide de structures de données telles que les *tables virtuelles* utilisées par la plupart des compilateurs C++ courants. Comme notre langage cible est une variante du langage intermédiaire Cminor de CompCert, nous l'appelons donc Vcm, pour « Cminor avec tables virtuelles ».

Nos résultats sont mis en perspective au Chapitre 8.

Nous les avons publiés dans un article [72] à la conférence POPL 2011.

0.2.2 Construction et destruction des objets

Chapitre 9. Nous spécifions formellement la sémantique de la construction et destruction d'objets en C++ : nous introduisons alors un langage que nous appelons κ++ (κ désignant la présence de « constructeurs »). En guise de fondements à notre sémantique, nous introduisons la notion d'*états de construction* d'un objet, marquant l'évolution des processus de construction et destruction pour chaque objet. Les états de construction nous permettent de considérer la notion de *type dynamique généralisé* d'un objet, pour clarifier le comportement du *dispatch* dynamique de méthodes pendant la construction ou la destruction : nous arguons qu'il ne doit pas utiliser des parties non encore initialisées (ou déjà détruites) d'un objet. Notre sémantique formelle a eu un impact sur le développement et l'évolution du Standard C++.

Chapitre 10. Puis nous étudions quelques propriétés de notre sémantique, liées à la notion C++ de *durée de vie* d'un objet, et à la gestion de ressources suivant le paradigme *RAII* (*resource acquisition is initialization*). Nous clarifions les interactions entre l'héritage multiple et la construction et destruction d'objets. En particulier, nous décrivons l'évolution des états de construction et nous étudions son impact sur le type dynamique généralisé d'un objet, en soulignant les moments où celui-ci change.

Chapitre 11. Puis, forts de nos résultats, nous spécifions et prouvons formellement un compilateur de κ++ vers un langage à trois adresses et accès mémoire de bas niveau. Nous remarquons que les tables virtuelles d'un objet doivent changer pendant la construction et destruction, exactement aux moments où son type dynamique généralisé change. C'est pourquoi nous procédons en deux passes de compilation.

D'abord, nous compilons κ++ vers un sur-ensemble de s++ que nous munissons d'une opération supplémentaire pour changer explicitement le type dynamique généralisé d'un objet et de tous ses sous-objets. Nous appelons donc ce langage intermédiaire Ds++ pour « s++ avec modification du type dynamique généralisé ». Ainsi, dans Ds++, les états de construction ne sont plus nécessaires.

Enfin nous arguons que les tables virtuelles utilisées durant la construction et destruction, appelées *tables virtuelles de construction*, doivent elles-mêmes être stockées dans des tables, appelées *tables de tables virtuelles*. Alors, notre seconde passe peut compiler Ds++ vers un langage à trois adresses, accès mémoire de bas niveau, tables virtuelles et tables de tables virtuelles. Nous appelons donc notre langage cible CVcm, pour « Vcm avec tables virtuelles de construction ».

Nous mettons nos résultats en perspective au Chapitre 12.

Nous avons publié nos résultats des chapitres 9 et 10 dans un article [73] à la conférence POPL 2012, ainsi qu'une modification du Standard C++ [45] ; d'autres propositions de modification sont en attente.

Finalement, le Chapitre 13 conclut et ouvre des perspectives générales vers une sémantique formelle et un compilateur vérifié pour C++ tout entier.

0.3 Bilan

0.3.1 L'expérience Coq

À première vue, formaliser le langage C++ avec un assistant de preuve tel que Coq peut paraître effrayant et décourageant, au vu de la complexité tant décriée du modèle objet de C++ et de l'importante quantité de détails à prendre en compte dans la formalisation : rien ne peut être considéré comme « trivial et laissé au lecteur ».

En réalité, ce n'est pas le cas. En effet, la sémantique formelle de κ++, incluant la construction et destruction d'objets, s'écrit en seulement 900 lignes de Coq. Nous croyons même qu'elle pourrait se décrire de façon encore plus succincte en adoptant un point de vue plus général que l'héritage (par exemple en unifiant la construction des scalaires et celle des objets — ce qui nous permettrait de prendre en charge les tableaux de scalaires). Cela montre que les spécifications Coq sont maniables, grâce au langage de spécification Gallina, remarquable pour sa clarté et ses fondements mathématiques précis tels que le Calcul des Constructions Inductives.

En revanche, les scripts de preuve, de leur côté, totalisent environ 80000 lignes de Coq, semblent souvent répétitifs (nous croyons qu'ils pourraient être réécrits en reconnaissant des astuces de preuve souvent utilisées, telles que des preuves par symétrie, et en les factorisant par des tactiques Ltac), et leur revérification par `coqc` prend plus de deux heures et demie sur un Pentium Core Duo cadencé à 2 GHz et consomme la moitié des 4 Go de mémoire vive. Après des tests informels, nous prétendons que ces problèmes pourraient être essentiellement dus à l'implémentation du système Coq lui-même, au niveau des dépliages de définitions pendant les unifications de typage. Toutefois, on pourrait également expliquer ces chiffres élevés par notre choix délibéré d'automatiser nos preuves aussi peu que possible, en nous limitant à l'automatisation des raisonnements en logique du premier ordre, voire propositionnelle (en utilisant la tactique `tauto`), ou l'arithmétique entière (`omega`) : aucun de nos lemmes n'est intégré dans une base de `Hint` pour l'automatisation. Notre choix a pour but de comprendre quels lemmes sont utilisés dans nos preuves, et quand ils le sont, afin de pouvoir par la suite expliquer nos preuves au niveau le plus haut possible, ce que nous avons réalisé dans le présent document.

0.3.2 Impacts pratiques

Notre travail nous a permis de trouver des erreurs et incohérences dans le Standard C++03 [42]. Notamment, le programmeur peut appeler une fonction virtuelle pendant la construction des champs d'un objet, mais le manque de symétrie dans le Standard nous a paru très surprenant, en ce sens qu'il laissait non spécifiés de tels appels durant la destruction des mêmes champs. Ce problème a été corrigé [45] dans le Standard C++11 [43]. D'autres problèmes ont été soumis au comité de standardisation C++ pour être pris en compte dans de futures versions du langage.

Du point de vue pratique, notre travail nous a également permis d'expliquer des bogues connus dans des compilateurs C++ tels que Microsoft Visual C++ 7.0 ou Borland C++ Builder 5.x [38]. Ces bogues, dus à des optimisations trop agressives, violent le principe d'identité des sous-objets en présence de bases ou membres vides.

Finalement, nous avons prouvé un algorithme de représentation concrète des objets en C++ couvrant la presque totalité de l'ABI d'Itanium, maintenant très utilisée sous une forme adaptée par GNU GCC sur des plateformes courantes. Nous avons seulement omis une optimisation controversée des bases primaires virtuelles, que le consortium développant l'ABI considère cependant comme un « défaut de conception » de l'ABI [22].

0.3.3 Impacts potentiels

En plus des impacts immédiats sur le Standard C++, notre travail peut être considéré comme une description alternative de C++ pour mieux comprendre le modèle objet de C++. Notre sémantique formelle valide a posteriori les principes de conception orientés objet de C++ et les exigences du Standard telles que le principe d'identité des sous-objets (pour rendre compte formellement de l'héritage répété en C++), ou du *dispatch* de fonction virtuelle pendant la construction (pour modéliser correctement le principe RAII de gestion de ressources). De plus, notre approche fondée sur la compilation vérifiée donne des bases solides aux techniques d'implémentation habituelles utilisées par les compilateurs C++ les plus courants (dont GNU GCC) : optimisation des bases vides, tables virtuelles, tables de tables virtuelles.

Inversement, nous croyons que notre travail de description formelle d'un sous-ensemble orienté objet de C++ peut servir de base à l'application de méthodes formelles sur des programmes écrits en C++. Grâce à notre sémantique formelle, une approche prometteuse pourrait être l'-analyse statique de programmes par interprétation abstraite [23]. Une telle méthode reposerait sur un interprète abstrait implémentant directement les règles de la sémantique opérationnelle de κ++. En revanche, l'application de notre sémantique formelle à la vérification déductive de programmes à la Frama-C [5] ou Why [8] pourrait exiger du travail supplémentaire. Une piste à explorer pourrait être le développement d'une logique à la Hoare pour l'héritage multiple. Une telle logique pourrait être fondée sur la logique de séparation développée par Luo et al. [56] pour raisonner sur les accès aux champs en présence d'héritage multiple. À partir de tels systèmes logiques, nous espérons que notre travail pourra permettre de spécifier des préconditions et des postconditions sur les constructeurs et destructeurs d'une classe, ainsi que sur les fonctions virtuelles pendant la construction et la destruction. Elles pourraient être exprimées directement en termes de nos états de construction d'objets.

0.3.4 Travaux futurs

Essentiellement, notre travail peut être étendu de deux façons. Tout d'abord, des optimisations supplémentaires pour la compilation de C++ pourraient être traitées, notamment, la prise en compte des bases primaires virtuelles. On pourrait également étudier encore plus en détail les structures de tables virtuelles et tables de tables virtuelles utilisées pour l'implémentation des opérations polymorphes, notamment leur représentation concrète — incluant aussi une implémentation concrète plus réalistes de `dynamic_cast` utilisant les structures d'identification de type à l'exécution (RTTI, *run-time type identification*). Cela ouvrirait également la voie à la formalisation des *thunks*, optimisant les appels aux fonctions virtuelles. Cependant, de telles représentations et optimisations dépendent fortement de la plate-forme cible ; de plus, si ces optimisations étaient effectuées dans le cadre de CompCert au niveau du langage intermédiaire CVcm ou Cminor, il faudrait alors veiller à ne pas perdre leurs bénéfices dans les passes suivantes de compilation (*backend*) vers l'assembleur. (En pratique, contrairement à CompCert, des compilateurs tels que GNU GCC génèrent directement du code assembleur, sans passer par des langages intermédiaires.)

Au-delà des optimisations, nous pensons que notre travail peut être étendu du point de vue de la formalisation de la sémantique, vers C++ tout entier suivant un certain nombre de directions. Notamment, la formalisation de la sémantique de copie de C++ nécessiterait d'autoriser la destruction des objets temporaires après l'appel d'un constructeur. Cela nous permettrait alors de prendre en charge le passage d'arguments par valeur (par copie, en utilisant le constructeur de copie), ainsi que les fonctions renvoyant des structures.

D'un point de vue plus général, des étapes supplémentaires plus lointaines restent nécessaires à la formalisation de C++ tout entier : les patrons (*templates*) suivant les travaux en Isabelle de Siek et Taha [76], ou encore la concurrence suivant les travaux en Isabelle/HOL de Batty et al. [14]. La formalisation de C++ en HOL4 par Norrish [64] se veut proche du Standard C++, ainsi elle pourrait servir de base solide à l'extension de notre travail. En effet, elle couvre les références, mais aussi les exceptions, qui requièrent notamment que si la construction d'un sous-objet d'un objet échoue, alors tous les sous-objets construits jusque-là au sein de cet objet doivent être détruits. Dans un premier temps, cependant, on pourrait mettre les exceptions de côté et chercher plutôt à couvrir seulement le sous-ensemble de C++ décrit par les lignes de conduites formulées par Lockheed Martin [55]. Il resterait alors à traiter l'accessibilité (public/privé), les `const`, et surtout la surcharge.

La lutte des classes
commence à la base.

N.P.

Part ◯

Preliminaries

Chapter 1

Introduction

1.1 Context

1.1.1 Software verification

Software is becoming ubiquitous, and in particular in critical systems such as transportation (railways [67], avionics [55] or space), military applications, or medical devices. In such critical domains, any bug may lead to very costly damage, and even to loss of human lives. Between 1985 and 1987, massive overdoses of radiation by the Therac-25 radiotherapy machine caused at least 6 patient deaths [53]. The machine could actually be operated in two radiation modes: direct low-energy radiations to the patient on the one hand, and indirect high-energy radiations through an intermediate target on the other hand. The software embedded in the radiotherapy machine failed to activate the correct mode. Overexposition to radiations may also result from improper error handling by the user interface: the software embedded in another radiotherapy machine would not check for the validity of the data entered by the physicians, which led to at least 17 deaths in Panama between 2000 and 2002 [18].

Even though bugs in critical software are not necessarily fatal, they can incur important financial losses, such as in the 1996 Ariane 5 failed maiden flight, due to an integer overflow [28], leading to the loss of the rocket and its payload (Cluster spacecraft from European Space Agency and NASA), worth a total US$370 million; moreover, the loss of the spacecraft led to a 4-year delay for the completion of the Cluster spatial mission.

A survey by Dershowitz [27] lists the costliest and deadliest software bugs in the history of critical software. Avoiding those critical bugs is the main motivation for finding a way to trust critical software. The standard approach adopted in the industry of critical embedded software includes *software testing* [46] performed by running the software on "dry-run" test cases to discover bugs; and manual *code reviews* such as Fagan inspection [31]. In practice, software testing and code reviews have been relatively effective so far, with no known casualties in fly-by-wire airplanes due to software errors. This is partly due to their systematization by the DO-178B official regulations for the certification of avionics software [65]. However, software testing turns out to be very costly and time-consuming. Moreover, it is difficult to guarantee that test cases cover all possible cases. Furthermore, the scalability of software testing is seriously challenged by the ever increasing size and complexity of critical software: the code size of Airbus A320 control software reaches a total 10 megabytes; this size amounts to hundreds of megabytes for the Airbus A380 [34].

A promising alternative or complement to testing is *software verification* through *formal methods*, intended to both reduce testing costs and increase trust in programs. Software verification allows programmers to establish high-level properties on their programs *statically*, i.e. without actually executing them. Examples of such high-level properties include absence of run-time errors, unreachability of states deemed “impossible”, and functional correctness with respect to a given specification.

Software verification can be achieved through several formal methods. *Model-checking* [21] consists in an enumerative or symbolic exploration of all reachable states of all program executions. Such an exploration often relies on a model of the actual program, and is performed by fully automated tools such as Alloy [6, 44]. *Abstract interpretation* [23] computes an approximation of the ranges of values that the variables of a program can take (e.g. integer intervals). Such computations are performed by dedicated tools called *static analyzers* such as the fully automated Astrée [1] used on the actual C code of the control systems of Airbus A340 and A380 aircraft to prove the safety of their memory accesses and floating-point computations.

Program proof by deductive verification is a general-purpose method allowing programmers to prove high-level properties on programs by annotating them with logical formulae (preconditions, postconditions, loop invariants). From those annotations, an automated tool called a *verification condition generator* such as Frama-C [5] or Why [8] generates lemmata called *proof obligations*, such that proving the high-level properties on the program boils down to *discharging* the proof obligations, i.e. solving them using *automated theorem provers* such as Z3 [26], or *proof assistants* such as Coq [4].

Finally, *generation of verified software* consists in automatically extracting a program from (manual or automated) proofs of theorems carried within an abstract model, for instance thanks to the B *refinement* method [9, 10] used for the signalling system of the fully automated Paris Metro line 14 (Meteor SAET[1]) [67]. Likewise, the *extraction* mechanism [51, 52] of Coq [4] automatically generates executable programs from Coq specifications and proofs.

1.1.2 Formal semantics

Software verification, regardless of the method actually used, requires to understand the exact meaning of programs. To this end, it is necessary to define the *semantics* of programming languages, i.e. the meaning of language constructs. Most often, the semantics of programming languages are defined through informal descriptions, such as standards or specifications written in natural language (C, C++, Java, ECMAScript), or even only a reference implementation (Perl, Ruby, Caml, Haskell).

Such informal descriptions are not enough to ground the correctness of critical software, as they can suffer from interpretation ambiguities or missing unspecified behaviours. Consequently, it is necessary to define mathematically-precise specifications. This leads to the concept of *formal semantics* of a programming language, based on solid mathematical grounds. The formal semantics of a language can be described in several mathematical forms [77], including *axiomatic semantics* extending the mathematical logic with rules to prove assertions about the program execution state, *denotational semantics* interpreting language constructs by mathematical objects, and *operational semantics* defining the possible execution steps through a transition relation between execution states.

1. *Système automatisé d'exploitation des trains*, Automatized system for train operation

Many formal semantics are known for calculi or small language subsets, often for academic purposes, but are surprisingly rare among widely used languages. The Standard ML [61] functional language is perhaps the only example of realistic programming languages defined by a formal semantics. However, due to their comprehensive precision to avoid ambiguities and unspecified cases, formal semantics are big formal systems, making it desirable to *mechanize* them using proof assistants such as Coq [4]. The PoplMark challenge [13] underpins this desire of mechanizing the metatheory of programming languages.

1.1.3 Verified compilation

Formal verification of software is generally conducted on the source, human-readable code of the program, but not the actual code executed by the machine, obtained by the compiler. Thus, it is necessary to also trust the compiler: it must not change the meaning of the source program when producing the compiled program.

Indeed, compiler bugs can have serious consequences. In 2011, shortly after the release of Java 7, Apache and Oracle warned of a serious bug in the Oracle Java 7 compiler [66]. This compiler introduced optimizations that miscompiled some loops, which introduced bugs in well-known and widely used software such as the Apache Lucene Core plain text search engine [33]. Moreover, compiler bugs are not rare: Yang et al. [86] found more than 325 unknown bugs in mainstream C compilers such as GCC or LLVM thanks to massive random testing.

One solution proposed in avionics to avoid such bugs introduced by unsound compilers, as prescribed by the DO-178B official regulations [65], is to write a compiler such that manual analysis of the source and compiled code allows to argue that the meaning of the source program was not changed by the compiler. However, such analysis is done entirely manually and empirically. Moreover, this prevents the compiler from performing clever optimizations [15, §2.1]. Furthermore, this method is difficult to adapt to high-level languages (such as object-oriented languages, or functional languages), for which the memory model is by far different from the actual memory model of the machine. This is why it is necessary to also verify the compiler itself.

Dave [24] provides a representative survey of works on compiler verification until 2003. A couple of those works may be regarded as seminal. The need for compiler verification has been pointed out by John McCarthy as early as 1963 [57]. He partially answers his own concern by the first paper-and-pencil proof of a compiler for arithmetic expressions [58]. However, a more accurate solution has been seminally brought by Milner and Weyrauch [60], who proved a similar compiler not on paper, but for the first time using a mechanical logic system. In 1989, a more realistic compiler from an assembly-like language has been mechanically proved by Moore [62].

One solution to prove the soundness of a compiler is to see it like an ordinary program and to prove its functional correctness with respect to the following specification:

Given a source code, if the compiler produces a compiled code, then any specification met by the source code is met by the produced code.

What makes the difference between a compiler and an ordinary program is that the specification of a compiler actually includes a way of describing the behaviours of both the source and the compiled code. More exactly, specifying a compiler requires to formally specify the source and target languages, i.e. to formally describe their syntaxes, and their semantics. As such, compiler verification is an additional motivation to formalizing the semantics of programming languages.

Notable examples of formally verified compilers include Jinja [47] from a subset of Java to the JVM, written and proved in Isabelle [7]; and most importantly CompCert [2, 50, 49] from C to PowerPC, ARM or Intel x86 assembly languages, written and proved in Coq [4]. Those two formally verified compilers are obtained by extraction.

Our work is based on the CompCert verified compiler. CompCert is a chain of several compilation passes, most of which are formally verified. Some passes are proved using direct theorem proving. Other passes are tackled through *verified translation validation* [82]: they actually use an unproved compilation pass, but, for each compilation, the source code and the result of the compilation pass are checked by a formally verified validator determining whether they have the same semantics.

1.1.4 The C++ language

This thesis deals with formal semantics and verified compilation for a subset of the C++ language. C++ [30, 79, 40, 42, 43] is an object-oriented language created in 1981 by Bjarne Stroustrup as “C with classes”: initially, C++ was meant to extend C with an object model essentially based on Simula-like classes. C++ provides an exceptionally rich object model featuring multiple inheritance with both shared and repeated inheritance, dynamic function dispatch, and dynamic cast. C++ also features exceptions, and templates (as a compile-time expansion for generic programming), leading to a unique combination of support for systems programming providing nevertheless powerful abstraction mechanisms.

Despite its apparent complexity, C++ is one of the most widely used languages, even in critical embedded software, e.g. by companies such as Lockheed Martin. However, such companies edict guidelines [55] for their programmers, instructing them to drop some features of C++ such as exceptions. Such restrictions allow to morally assess that the semantics of their programs are well-defined; moreover, they are designed to ease code review and analysis.

Such guidelines only provide moral rationales. Thus, we aim at giving a formal assessment to part of those guidelines by formally specifying a subset of C++ and proving a verified compiler for this subset. Formally specifying a subset of C++ does not only give confidence on the compiler, but also on the language specification itself, by providing a formal background stronger than the prose standards defining C++. In this work, we focus on the C++ object model, including multiple inheritance, object construction and destruction, and resource management.

Chapter 2 exposes an informal, yet comprehensive tutorial on the C++ object model. Chapter 3 introduces the formal mathematical notations used throughout this thesis, including the underlying formalism used for describing the operational semantics of programming languages. The theorem-proving method for compiler verification based on CompCert and followed by our work is formally presented in Appendix B.

1.2 Summary of the contributions

Our work aims at formalizing the semantics of the C++ object model, including multiple inheritance and object construction and destruction, and validating this semantics by a verified compiler assessing the soundness of the mainstream techniques used in practice by most real-world compilers. We have mechanized all our formal semantics, specifications and proofs using the Coq [4] proof assistant. Our whole Coq development is available at [71]. Its architecture is summarized in Appendix A. As such, our work proposes a formal answer to our following claim:

Thesis:
The semantics and compilation of the C++ object model can be formally trusted.

To this end, we tackle the most two important issues for our goal:

Part I. the *object layout* problem: how objects are represented in concrete memory in a practical way allowing field accesses, static casts and virtual method dispatch in constant time and memory accesses.

Part II. *object construction and destruction*: the C++ object construction mechanism is meant to establish invariants on an object by initializing its fields, which may need to acquire resources, upon its creation. So, its destruction counterpart must turn the object back to raw memory by releasing those resources. We study the impact of C++ multiple inheritance on those mechanisms.

Figure 1.1 (p. 37) summarizes all languages and compilation passes of our work.

1.2.1 Object layout

Chapter 4. We formalize the semantics of C++ multiple inheritance including shared and repeated inheritance, scalar field accesses, virtual function calls, static and dynamic casts, and adding C-style embedded structures and structure arrays, an aggregation mechanism distinct from inheritance. This work, an extension to the CoreC++ language by Wasserrab et al. [85, 84], formalizes the notion of subobjects due to both inheritance and aggregation, leading to a language which we called s++, standing for "CoreC$\underline{++}$ with embedded $\underline{s}$tructures".

Chapter 5. Then, we formally specify a family of C++ concrete object layout algorithms, through a set of sufficient conditions on their computed results. Those conditions take *empty base optimizations* into account, to minimize the memory footprint of subobjects of empty class types within objects. We prove that those conditions satisfy the good variable property and preserve the identity of subobjects, allowing to correctly model object-oriented operations such as field accesses, casts and pointer equality tests.

Chapter 6. Then, we formally prove the correctness of two realistic object layout implementations. The first algorithm is inspired from the Common Vendor ABI for Itanium [22], which GNU GCC also reuses and adapts for other platforms. In the second algorithm, we propose further empty base optimizations. Actually, we prove that those two algorithms meet the soundness conditions of our considered family of algorithms.

Chapter 7. Finally, for any layout algorithm complying with our sufficient conditions, we formally specify and verify a compiler from s++ to a language featuring low-level memory accesses and *virtual tables*, a special data structure used by most present-day C++ compilers and designed to implement polymorphic operations such as virtual method dispatch, or dynamic cast. We call this language Vcm, which stands for

"Cminor with virtual tables", as Vcm is a variant of the Cminor [2, 49, 16] intermediate language of CompCert.

We discuss our results in Chapter 8.

Those results have been published in a POPL 2011 article [72].

1.2.2 Object construction and destruction

Chapter 9. We formally specify the semantics of C++ object construction and destruction, leading to a language we call κ++ (κ standing for "constructors"). This semantics is based on *construction states*, which mark the evolution of the construction and destruction process for each object. Thanks to construction states, we define the notion of the *generalized dynamic type* of an object, which is used to clarify the behaviour of virtual method dispatch during object construction and destruction, arguing that dispatch prevents from using parts of an object that are not constructed yet (or already destructed). We also discuss the impact of our semantics on the evolution of the C++ Standard.

Chapter 10. Then, we study some properties of this semantics, leading to results related to the C++ notion of *object lifetime* and *RAII (resource acquisition is initialization).* We clarify the intricacies between inheritance and object construction and destruction. In particular, we describe the evolution of construction states, and we study its impact on the generalized dynamic type of an object, to point out its changes.

Chapter 11. Then, we apply all our results to build a compiler from κ++ to a language featuring low-level memory accesses. We point out the fact that the virtual tables of an object have to change during their construction or destruction, actually at the same time as the generalized dynamic type changes. Thus, our compilation goes through two passes.

First, κ++ is compiled to a superset of s++ featuring an additional operation to explicitly change the generalized dynamic type. Thus, we call this intermediate language Ds++, which stands for "s++ with set generalized dynamic type". Thus, in Ds++, construction states are no longer required.

Then, we argue that virtual tables used during construction and destruction, called construction virtual tables, have to be stored in tables themselves, called *virtual table tables.* Thus, the second pass compiles Ds++ to a language featuring low-level memory accesses, virtual tables and virtual table tables. Thus, we call our target language CVcm, which stands for "Vcm with construction virtual tables".

We discuss our results in Chapter 12.

The results of Chapters 9 and 10 have been published in a POPL 2012 article [73], as well as a successful request of modification of the C++ Standard [45]; some other requests are pending.

Finally, Chapter 13 concludes and offers some general perspectives from our work, towards a formal semantics and verified compiler for full-fledged C++.

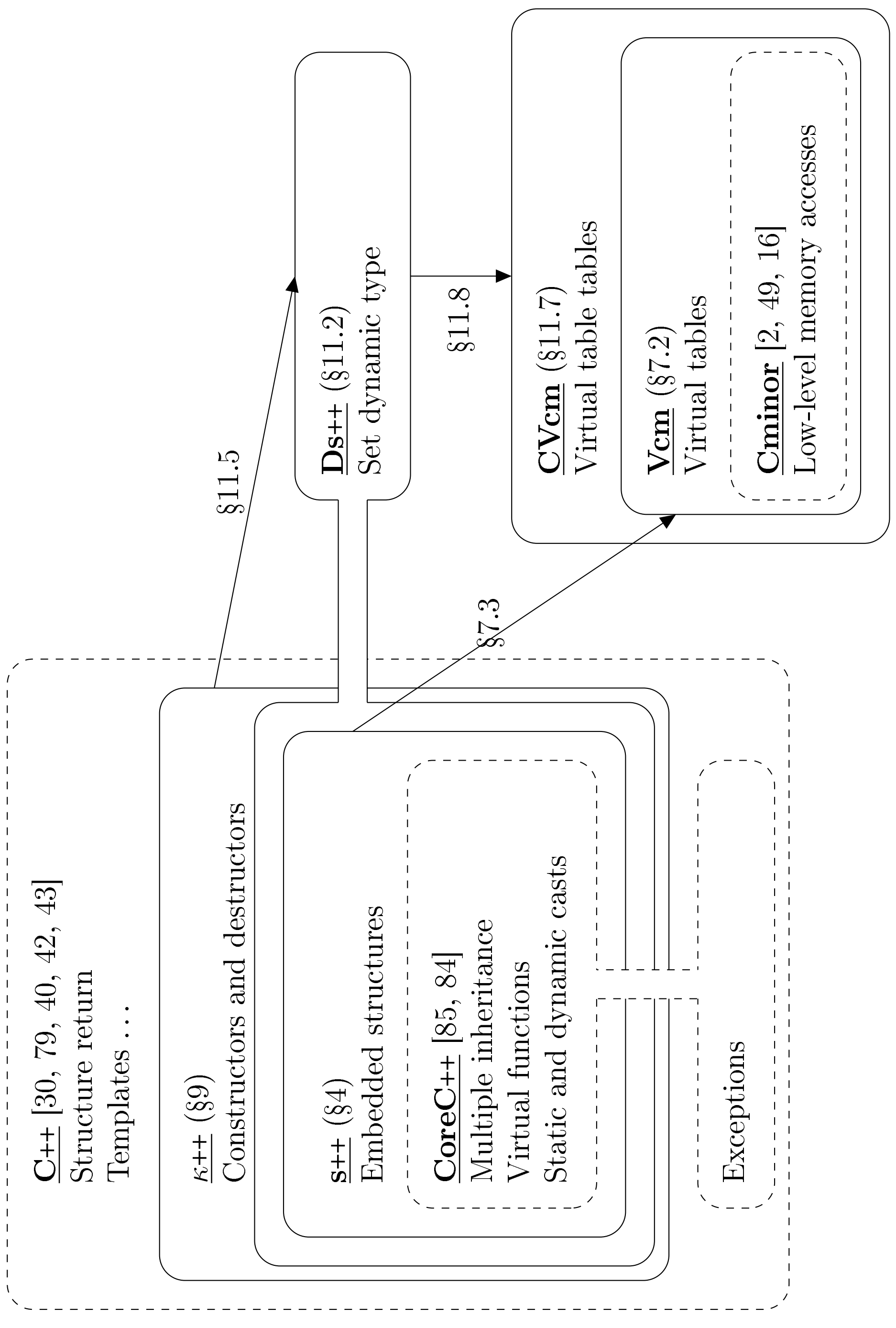


Figure 1.1 – Languages, features and compilation passes

Chapter 2

Tutorial: the C++ object model

In this chapter, we describe object-oriented programming, and in particular the C++ object model, focusing on multiple inheritance and object construction and destruction, through practical examples modeling real-world situations.

2.1 Classes and instances. Aggregation (*has-a*)

Consider an *alarm clock*. Its purpose is to help its user wake up by producing a loud sound at a specified time.

An electricity-powered alarm clock (cf. Figure 2.1 p. 39) can draw its electric current through a *plug*, which can be *plugged* or not. Once plugged, it can be controlled through a *control switch*, which can be turned on or off. When it is both plugged and turned on, it simply shows the current time to the user. To make it really work, the user has to provide the *alarm time* when the alarm should ring. The user then makes the alarm clock *wait* for the indicated time before actually ringing. But the user can also test the ring.

One can say that an alarm clock has the following attributes:

– a plug, which can be plugged or not,
– a control switch, which can be turned on or not,
– and the time when it should ring.

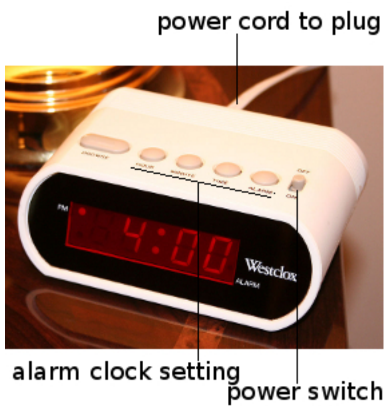


Figure 2.1: An alarm clock. Image from Wikimedia Commons, public domain.

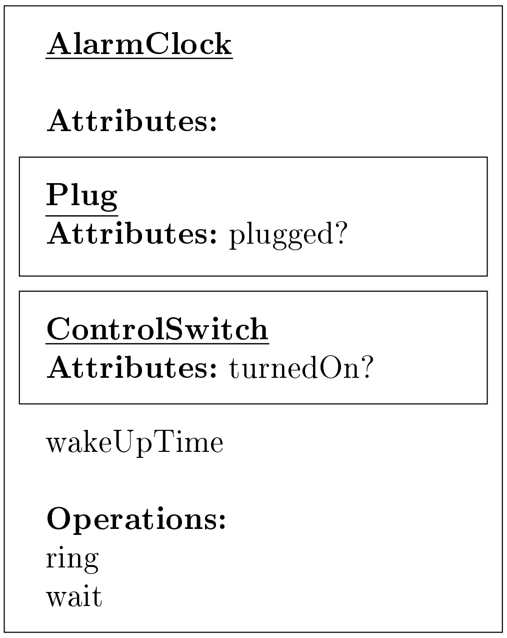
Besides, the user can perform operations on the alarm clock: make it ring, or make it wait for a time before ringing.

In C++ object-oriented programming, a programmer designs a *class* to represent alarm clocks: attributes of an alarm clock object are declared as *fields* of the class, also called *data members*, whereas operations on objects are declared as *methods* of the class, also called *class member functions*. Data members and class member functions are called *class members*.

While the *values* of attributes are local to an alarm clock object, the attributes and operations are declared at the class level. Thus, in C++, it is not possible to know the attributes and operations on an object, unless this object is known to be an *instance* of some class: in the latter case, the programmer gains access to all[1] fields and methods of the class on this object, which represent the attributes and operations of the object.

Thus, an alarm clock can be seen as an instance of a class, say `AlarmClock`. defined as follows[2]:

```
#include <ctime>

struct Plug {
  bool plugged;
};
struct ControlSwitch {
  bool turnedOn;
};
struct AlarmClock {
  /* Attributes of an alarm clock are
     data members of class AlarmClock */
  Plug plug;
  ControlSwitch controlSwitch;
  time_t wakeupTime;

  /* Operations on an alarm clock are
     class member functions of AlarmClock */
  void ring();
  void wait();
};
```



An alarm clock *has* a plug and a control switch: each instance of `AlarmClock` *embeds* an instance of `Plug` and an instance of `ControlSwitch`.

Thus, two different alarm clocks `a1` and `a2`, but still instances of the same `AlarmClock` class, may be made available to a (very sleepy) user through the following code:

```
AlarmClock a1;
AlarmClock a2;
```

1. C++ defines the notions of *visibility* (`public`, `private`, ...) of class members, to allow or forbid accessing them from outside class member functions. We do not cover this issue in this thesis, so we consider that everything is `public`. This explains why, throughout this thesis, we use `struct` rather than `class`.

2. `time_t` is the type name for dates and times, which are expressed as seconds since January 1st, 1970. It is equivalent to some integer type. It is defined in the C++ `ctime` standard library, along with its relevant system calls.

Each alarm clock embeds a plug and a control switch. Those components, which are *aggregation subobjects*, may be accessed as follows:

```
a1.plug.plugged = true;
a2.plug.plugged = true;
a1.controlSwitch.turnedOn = true;
```

In this example, the user plugged the plugs of both `a1` and `a2`, and turned on the control switch of `a1` only, forgetting `a2`.

Then, the action of making an alarm clock wait until the given time before ringing may be defined as the following `wait` class member function of class `AlarmClock`: if the alarm clock is plugged and turned on, wait until reaching[3] the wake-up time provided by the user, then ring.

```
void AlarmClock::wait() {
  if(this->plug.plugged && this->controlSwitch.turnedOn) {
    while(time(NULL) < this->wakeupTime) {};
    this->ring();
  }
}
```

Within the function body, the alarm clock object on which the `wait` action is performed is referred to by a pointer called `this`. The properties and actions on the current object may be used through this pointer.

Once a proper value is given to the `wakeupTime` data member of alarm clock `a1`, the user may put the alarm clock at work:

```
 a1.wait();
```

2.2 Inheritance (*is-a*). Virtual functions

2.2.1 Virtual functions: overriding and dispatch

Assume that an ordinary alarm clock rings by buzzing. Now suppose that we want to model *musical alarm clocks*, which play a music tone instead of buzzing.

Like an alarm clock, a musical alarm clock may be plugged or unplugged, turned on or off, and may be given the time when to ring. Conceptually, a musical alarm clock *is* a particular kind of alarm clock, so any musical alarm clock may be seen as an alarm clock from outside. This is called a *subtyping* relation: "musical alarm clock" is a subtype of "alarm clock".

But, contrary to ordinary alarm clocks, the user may choose the music tone of a musical alarm clock, say among a finite number of predefined tones. This needs an additional `tone` data member. Then, the `ring()` class member function has to be redefined in another way, so that a musical alarm clock, even if seen from outside as an ordinary alarm clock, will play a musical tone whenever asked to ring.

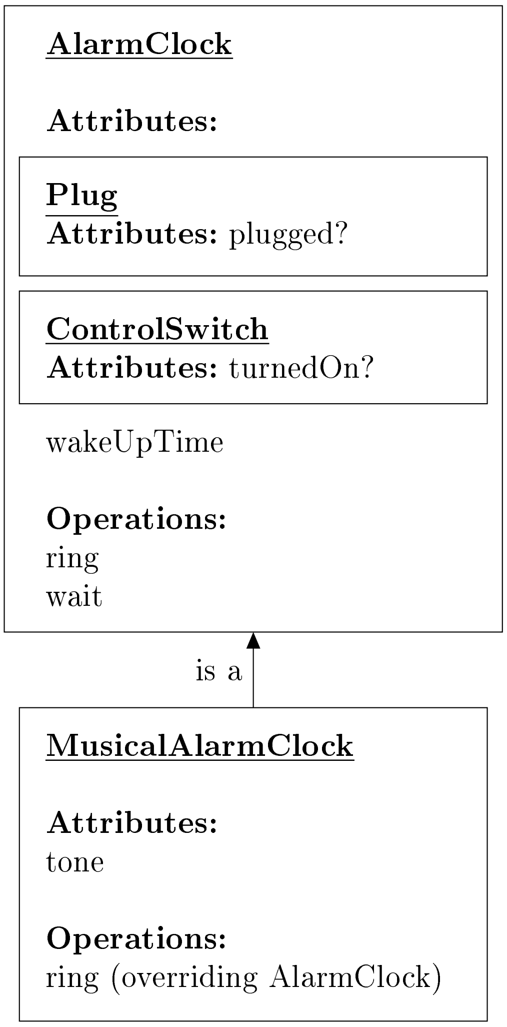
To this purpose, C++ offers the mechanism of *inheritance*. In our example, if we want to define a `MusicalAlarmClock` class to represent musical alarm clocks, then we shall:

3. `time(NULL)` is a system call retrieving the current date and time.

– declare that `MusicalAlarmClock` is a subtype of `AlarmClock`: we say that `MusicalAlarmClock` *derives from* `AlarmClock`, or that `AlarmClock` is a *base class* (or *base* for short) of `MusicalAlarmClock`;
– declare that the `ring` class member function of `AlarmClock` may be redefined by classes derived from `AlarmClock`. Such a class member function is said to be a *virtual function*
– redefine `ring` for the `MusicalAlarmClock`: we say that the new definition of `ring` for `MusicalAlarmClock` *overrides* the old definition for `AlarmClock`.

This leads to the following code:

```
struct AlarmClock {
  Plug plug;
  ControlSwitch controlSwitch;
  time_t wakeupTime;

  /* virtual = may be overridden
     in derived classes */
  virtual void ring();
  void wait();
};

  /* base classes = subtyping */
struct MusicalAlarmClock: AlarmClock
{

  /* additional attribute to allow
     the user to choose the musical tone */
  int tone;

  /* overriding = redefinition */
  void ring();
}
```



As `MusicalAlarmClock` derives from `AlarmClock`, it is said to *inherit* its data members (plug, control switch, wake-up time) from `AlarmClock`, so that it is not necessary to redeclare them. The "alarm clock" point of view of a musical alarm clock, which is the restriction of a musical alarm clock to the attributes and actions of an alarm clock, is called an *inheritance subobject* of the musical alarm clock.

The following code illustrates the use of inheritance in practice:

```
MusicalAlarmClock   ma;
MusicalAlarmClock* pma = &ma;  /* create pointer to the object */
AlarmClock*         pa = pma;  /* implicit conversion thanks to subtyping */
pa->ring();                    /* actually calls pma->MusicalAlarmClock::ring() */
```

As we can see, a musical alarm clock may be seen as an alarm clock through an implicit subtyping conversion, called *implicit cast*. But subtyping only changes the external view of the object, not its actual behaviour: a musical alarm clock seen as an ordinary alarm clock keeps behaving like a musical alarm clock, thus the overriding `ring()` redefinition is actually called. The process of selecting the right function to call is named *virtual function dispatch*.

In particular, thanks to the inheritance mechanism of function overriding, waiting for the alarm to ring, as implemented in `AlarmClock`, will also call the overriding `ring()` redefinition of `MusicalAlarmClock` upon wake-up time. This shows that there is no need to redefine the `wait()` class member function. Then, when calling the `wait()` function from a musical alarm clock:

```
ma.wait();
```

Then, the `this` pointer within the body of `wait()` must refer to the "alarm clock" point of view of the musical alarm clock. This requires that an *adjustment*, i.e. a conversion from musical alarm clock to alarm clock, be automatically performed during the call, before entering the function body.

This example points out the fact that inheritance is not only subtyping, but also impacts the actual behaviour of objects.

2.2.2 Casts

We have seen that it is possible to view an object from the point of view of some subtype of its actual type, thanks to implicit casts. But in some cases, C++ may allow the programmer to retrieve the original type of an object from one of its subtypes, through specific explicit casts.

Dynamic cast C++ makes it possible to know whether an alarm clock is a musical alarm clock. To this purpose, C++ provides the *dynamic cast* operator:

```
AlarmClock* pa;
...
MusicalAlarmClock* pm = dynamic_cast<MusicalAlarmClock*>(pa);
if(pm != NULL) {
  /* we know that pa is actually a musical alarm clock,
     pm is pa from the musical point of view */
  ...
} else {
  /* pa is not a musical alarm clock */
  ...
}
```

If `pa` actually refers to a musical alarm clock, then the dynamic cast will succeed and return the actual musical alarm clock to the programmer. But if `pa` is actually a genuine alarm clock, then the dynamic cast will fail and make the programmer aware of the failure by giving a null pointer.

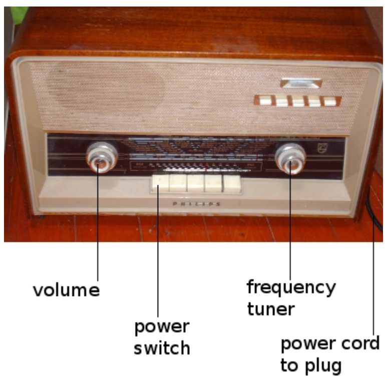


Figure 2.2: A radio receiver. Image from Wikimedia Commons, public domain.

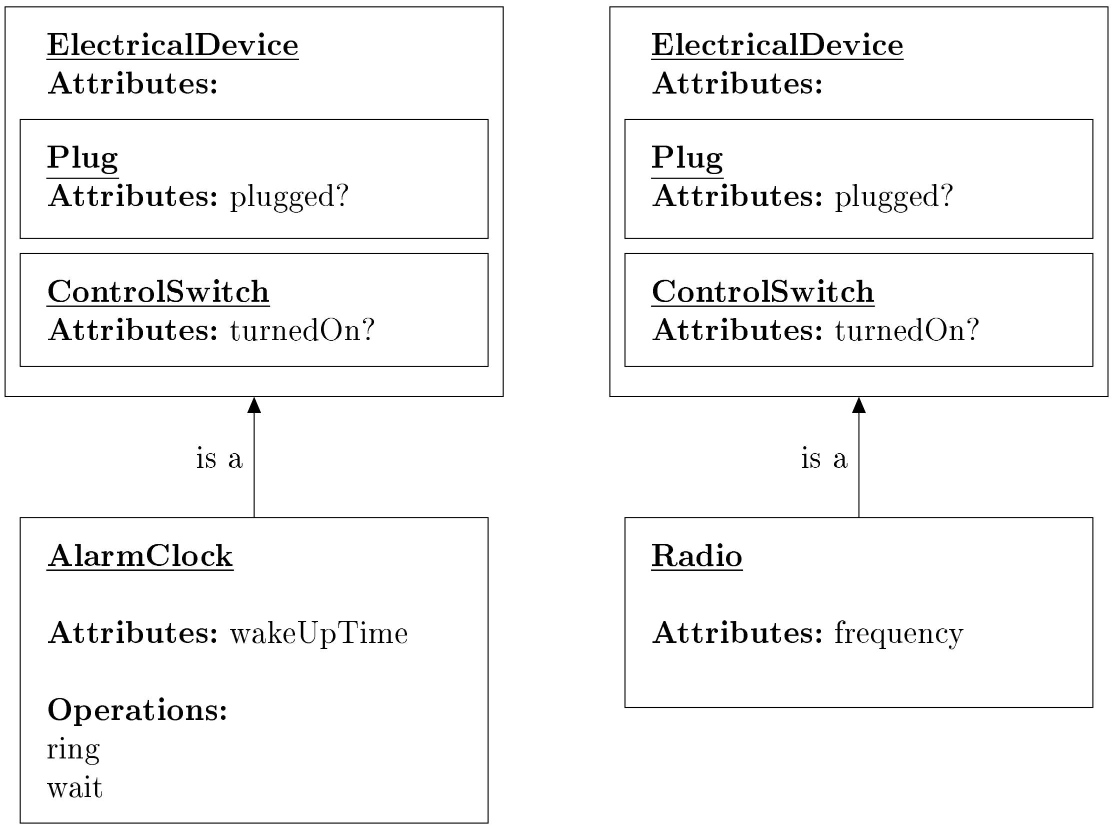
2.2.3 Inheritance for the purpose of subtyping

In C++, inheritance is the only way to achieve subtyping[4]. Thus, inheritance may occur even if there are no class member functions to override. Consider for instance a radio receiver (cf. Figure 2.2 p. 44). Like an alarm clock, it is an electrical device having a control switch and a plug. But it also provides an own attribute, its *frequency*, allowing the user to choose the radio station listened to.

We can define the type of electrical devices, containing a control switch and a plug. Then we can say that radios and alarm clocks are electrical devices defining each their own attributes and operations:

```
struct ElectricalDevice {
  Plug plug;
  ControlSwitch controlSwitch;
};

struct AlarmClock: ElectricalDevice {
  time_t wakeupTime;
  virtual void ring();
  void wait();
};

struct Radio: ElectricalDevice {
  float frequency; /* in MHz, floating-point number */
};
```

4. C++ offers a way to convert an object of some type to another type, by defining a *type conversion operator*, which may be used implicitly. However, such operators are defined by the programmer, by contrast to subtyping, which is part of the language semantics of C++. This conceptually means that the convertibility of some data to some other type does not mean that the data actually *are* of that type. Thus, convertibility is not subtyping.



However, for implementational reasons, C++ defines dynamic cast only if the "from" class, or one of its bases, declares at least a virtual function. It is impossible to dynamically cast from a radio, or an electrical device.

2.3 Multiple inheritance

Now consider a *clock radio* (cf. Figure 2.3 p. 46), which is both a radio and an alarm clock, such that:

- either the radio and the alarm can be turned on or off independently of the other, through the corresponding control switch
- if the radio is turned off, then ringing the alarm actually turns on the radio instead of buzzing

Then, the two questions are: is a clock radio a radio? Yes, as it can be used exactly like a radio. Is a clock radio an alarm clock? Yes, as it can be used exactly like an alarm clock.

Thus, a clock radio may be seen as an instance of a `ClockRadio` class inheriting from two classes at the same time: `Radio` and `AlarmClock`. This is called *multiple inheritance*. Moreover, `ClockRadio` redefines the `ring()` virtual function accordingly:

```
struct ClockRadio: Radio, AlarmClock {
  void ring();
};
```

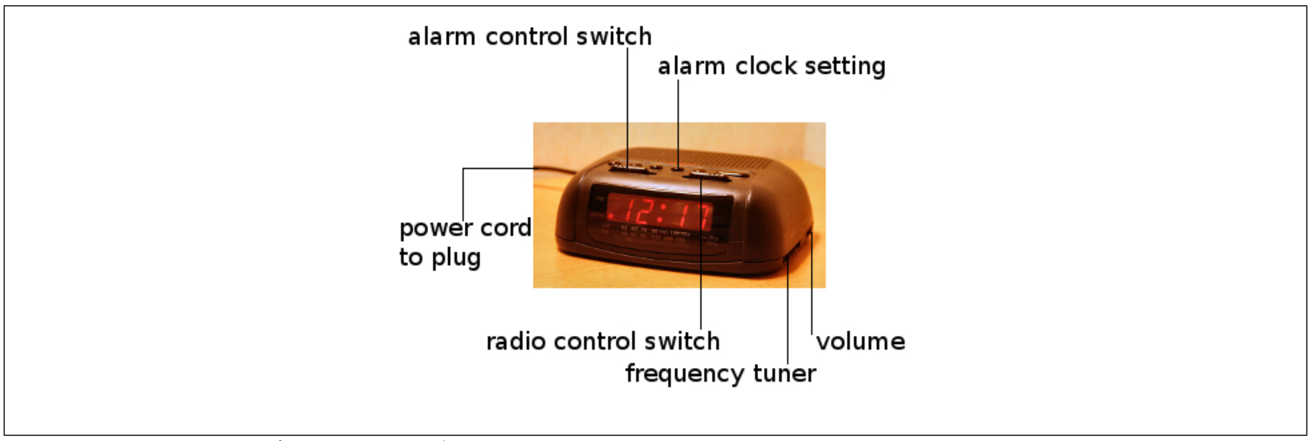
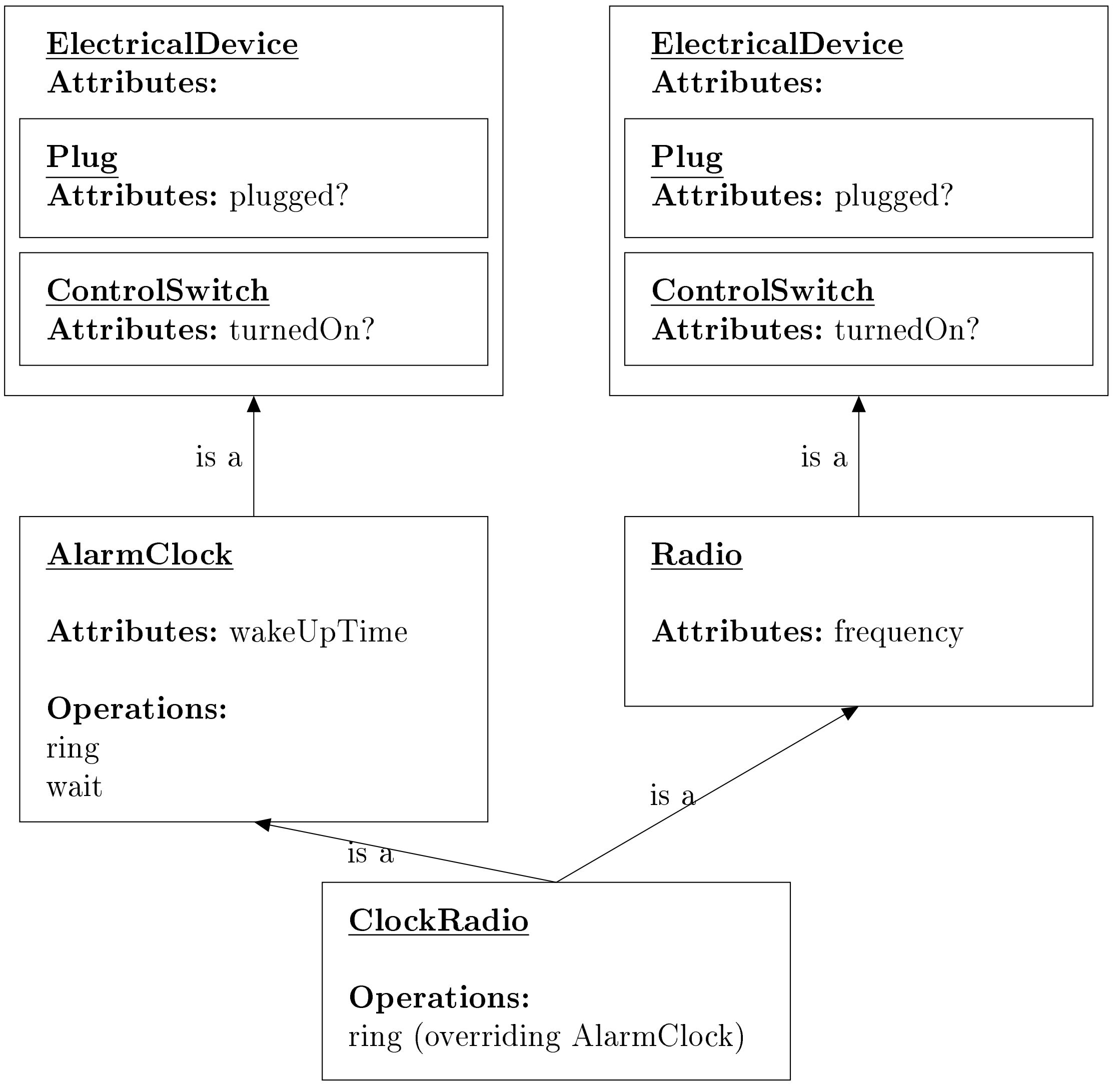


Figure 2.3: A clock radio. Image from Wikimedia Commons, public domain.



This inheritance scheme, or *hierarchy*, makes `ClockRadio` inherit from two classes, each of which inherits from a common `ElectricalDevice` class. In C++, this means that a clock radio is considered to be an electrical device in two *distinct* ways. This allows distinguishing between the control switch of the electrical device of the "radio" point of view, and the control switch of the electrical device of the “alarm clock” point of view. This is called *repeated inheritance* , as within a clock radio, there are two distinct “electrical device” inheritance subobjects.

To refer to the control switch of the radio point of view, explicit casts are necessary. Thus, under this inheritance scheme, the desired ringing protocol of switching on the radio may be implemented by overriding the `ring` virtual function in `ClockRadio` as follows:

```
void ClockRadio::ring() {
  if(!((Radio*)this)->controlSwitch.turnedOn) {
    ((Radio*)this)->controlSwitch.turnedOn = true;
  } else {
    this->AlarmClock::ring(); /* bypasses overriding */
  }
}
```

In case the radio is already turned on, the second option chosen actually performs the “legacy” `ring()` operation from the alarm clock point of view. This is done in C++ through an *explicit qualification*, performing a *non-virtual function call* bypassing inheritance: the exact function asked for is called.

2.3.1 Ambiguous subobjects

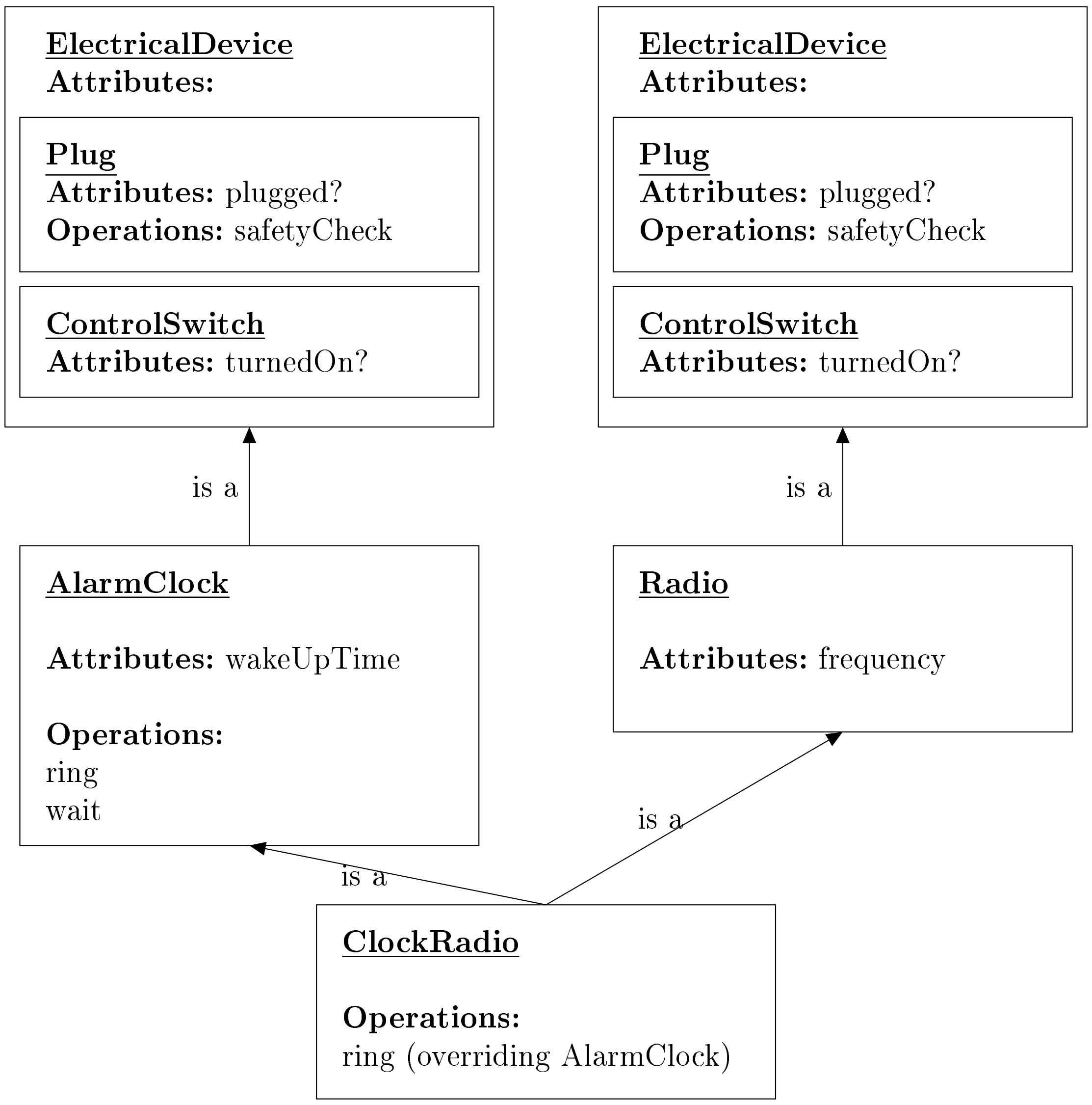
Implicit subtype cast is *ambiguous*. The following code refuses to compile:

```
ClockRadio ra;
ra.plug.plugged = true; /* ERROR: ambiguous implicit cast of ra
                           to ElectricalDevice */
```

To refer to either control switch, it is necessary to explicitly provide the point of view: convert `this` (referring to the current clock radio instance) to the correct subtype, through an *explicit cast*. Hence, in the redefinition of `ring`, `(Radio*)this` converts to the radio subtype, to reach the relevant control switch.

Ambiguous virtual functions Assume, for this section, that an electrical device has a *safety check* operation, which, for instance, checks whether the current drawn from the plug is not too high, and whether the control switch is correctly turned on or off (and not in an intermediate state). Such an operation is emulated through the `safetyCheck()` class member function in `ElectricalDevice`, which can be overridden in derived classes:

```
struct ElectricalDevice {
  Plug plug;
  ControlSwitch controlSwitch;
  virtual bool safetyCheck();
};
```



Performing such check on a radio (or an alarm clock) becomes possible:

```
Radio* pr;
...
pr->safetyCheck();
```

However, for a clock radio, which inherits twice from `ElectricalDevice`, it is impossible to know which safety check operation will be selected. A compiler will consequently reject the following code:

```
ClockRadio* pra;
...
pra->safetyCheck();
```

Thus, there are two solutions:
– either cast to `Radio` or `AlarmClock` to explicitly choose which `safetyCheck` function to call:

```
ClockRadio* pra;
...
Radio* pr = pra;
AlarmClock* pa = pra;
pr->safetyCheck(); /* calls ElectricalDevice::safetyCheck()
                      seen from Radio
                      (or Radio::safetyCheck() if overridden) */
pa->safetyCheck(); /* calls ElectricalDevice::safetyCheck()
                      seen from AlarmClock
                      (or AlarmClock::safetyCheck() if overridden) */
```

– or explicitly override `safetyCheck` in `AlarmClock`. For instance:

```
bool ClockRadio::safetyCheck() {
  return this->Radio::safetyCheck() && this->Alarm::safetyCheck();
};
```

In this particular example, those two function calls are not ordinary function calls: as they are explicitly qualified, they *bypass* inheritance, and they force calling the functions actually defined in the specified classes. Thus, this definition performs the safety check operation specific to the radio, then, in case of success, performs the safety check operation specific to the alarm clock.

2.3.2 Cross cast

Besides casts to derived, C++ dynamic cast offers a powerful way to navigate within a class hierarchy: given an alarm clock, it is possible to know whether the object is also a radio, without exposing to the programmer the intermediate steps (for instance, determining first whether the object is a clock radio). This is called *cross cast*: cast between two classes which are not base classes of each other in either way. Then, internally, this operation performs the following steps:

– view the object from its *most derived* point of view
– then, determine if there is exactly one way to see the object as an instance of the class to cross-cast to. The cast succeeds if, and only if, this is true.

Again, dynamic cast makes the programmer aware of the failure by returning a null pointer.

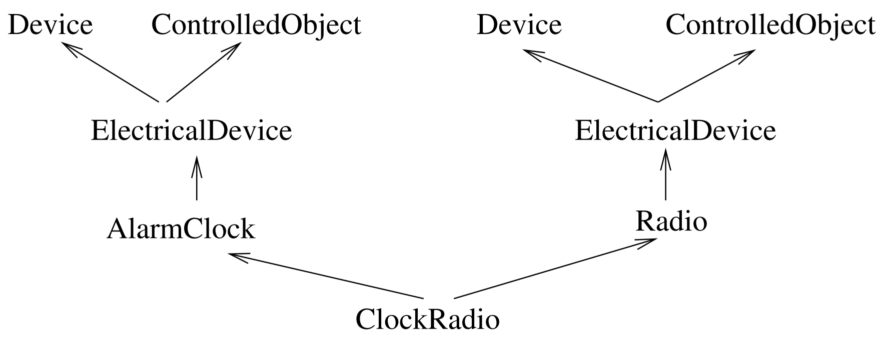
```
AlarmClock* pa;
...
Radio* pr = dynamic_cast<Radio*>(pa);
```

This dynamic cast succeeds if `pa` refers to a clock radio, because there is exactly one way to see a clock radio as a radio. However, it fails for a genuine alarm clock, which is in no way a radio. Thus, cross cast depends on the most-derived view of the object.

Moreover, ambiguity leads to another case of dynamic cast failure. Consider, for this section, that an electrical device itself is both a device and also a controlled object (indeed, there may be controlled objects that have a control switch but are not devices, such as an elevator door), thus inheriting from two base classes:

```
struct Device {
  virtual bool safetyCheck();
};
```

```
struct ControlledObject {
  ControlSwitch controlSwitch;
};
struct ElectricalDevice: Device, ControlledObject {
  Plug plug;
};
```



Then, in the following code:

```
ClockRadio* pra;
...
Radio*            pr  = pra;  /* OK, non-ambiguous base */
Device*           pd  = pr;   /* OK, non-ambiguous base */
ControlledObject* pc  = dynamic_cast<ControlledObject*>(pd);
                              /* NULL: ambiguous cross cast */
```

Dynamic cross cast fails, as there are two ways of seeing a clock radio as a controlled object: either as a radio, or as an alarm clock.

2.4 Virtual inheritance

There is a design problem in this inheritance hierarchy so far: a clock radio may be seen as an electrical device in two distinct ways, each of which has its own plug. But in reality, a clock radio should have only one plug, shared between the radio and the alarm clock features.

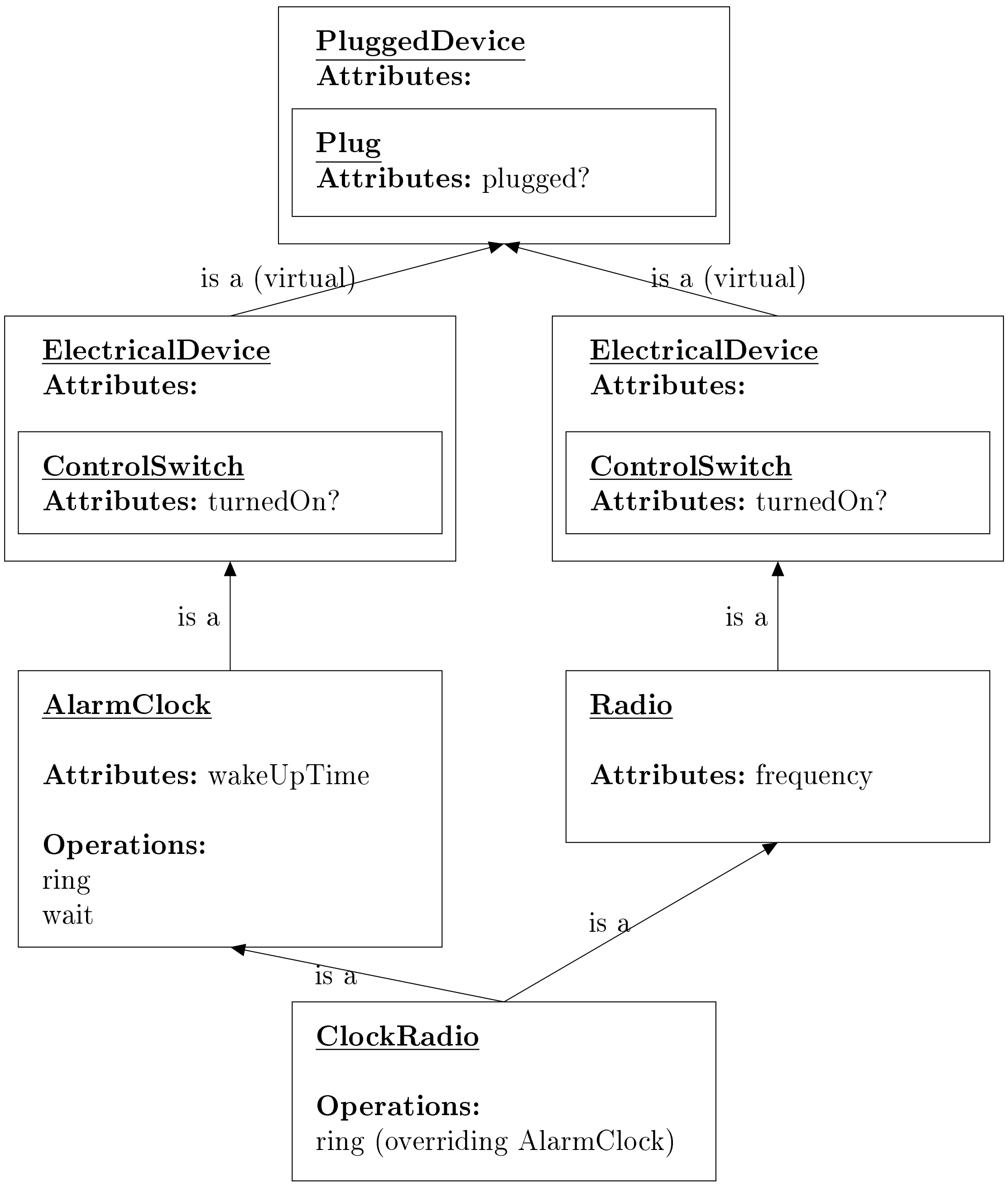
C++ offers a way to design the inheritance hierarchy to ensure sharing inheritance subobjects: *shared inheritance*, also known as *virtual inheritance*. (Thus, by contrast, repeated inheritance is also dubbed as *non-virtual inheritance*).

2.4.1 Virtual base classes

In our example, our clock radio has two control switches, but it should only have one plug. This may be reflected on the definition of an electrical device. An `ElectricalDevice` may be seen as a particular kind of `PluggedDevice`, where a plugged device provides a plug, and an electrical device additionally provides a control switch. In our case, we want that, even though a class may inherit from `ElectricalDevice` in several different ways, those ways should only provide one single way of inheriting from `PluggedDevice`. To this purpose, C++ offers the option of declaring `PluggedDevice` a *virtual* base of `ElectricalDevice`:

```
struct PluggedDevice {
  Plug plug;
};
struct ElectricalDevice: virtual PluggedDevice {
  ControlSwitch controlSwitch;
};
```

Then, there is nothing to change at the level of radios, alarm clocks, or clock radios, and the inheritance graph becomes as follows:



2.4.2 Casts

Cast to virtual base The main principle of virtual inheritance is the following: if a class has a virtual base, then accessing this virtual base from any of its derived classes does not depend on the way to access it. In other words, in our example, the following two `PluggedDevice` objects:

```
ClockRadio* pra;
...
Radio*         pr  = pra;
AlarmClock*    pa  = pra;
PluggedDevice* pp1 = pr;
PluggedDevice* pp2 = pa;
```

are actually the same object. In particular, the radio and alarm clock features of a clock radio actually share the same plug.

Thus, in particular, the following cast:

```
PluggedDevice* pp = pra;
```

succeeds, because the way to reach the virtual base class is irrelevant.

Cast from a virtual base to derived classes Conversely, a cast from a virtual base to some derived class is mostly not possible. For instance, casting from `PluggedDevice` to `ElectricalDevice` will not be possible for an `AlarmClock`, which is an electrical device in two distinct ways. Thus, such casts have to be actually considered as ordinary cross casts.

```
ClockRadio         ra;
ClockRadio*       pra = &ra;
PluggedDevice*    pp  = pra; /* OK, only one plug */
ElectricalDevice* pe  = dynamic_cast<ElectricalDevice*>(pp)
                             /* NULL: two possible (radio and alarm clock) views */
Radio*            pr  = dynamic_cast<Radio*>(pp)
                             /* OK: ClockRadio is a Radio in exactly one way */
```

2.4.3 Virtual functions: final overrider, domination, delegation to sister class

Now consider that a plugged device has a `safetyCheck()` operation (ensuring, for instance, that not too much electric current is drawn from the plug), that may be overridden by derived classes:

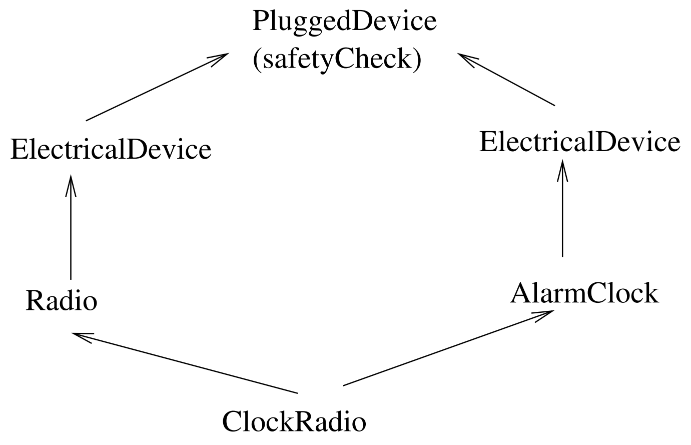
```
struct PluggedDevice {
  Plug plug;
  virtual void safetyCheck();
};
```

Then, consider that a safety check is performed on a clock radio, in the following sense:

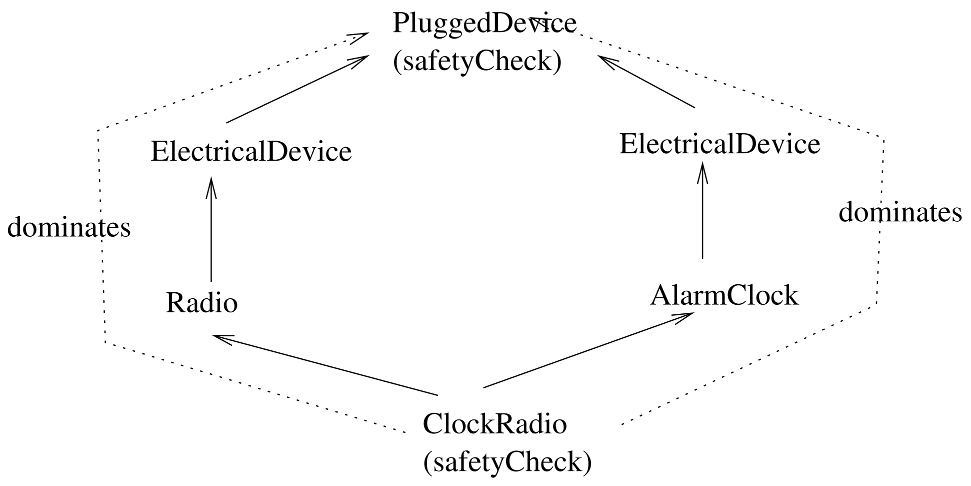
```
ClockRadio      ra;
ClockRadio*    pra = &ra;
PluggedDevice* pp  = pra;
pp->safetyCheck();
```

There are several cases:

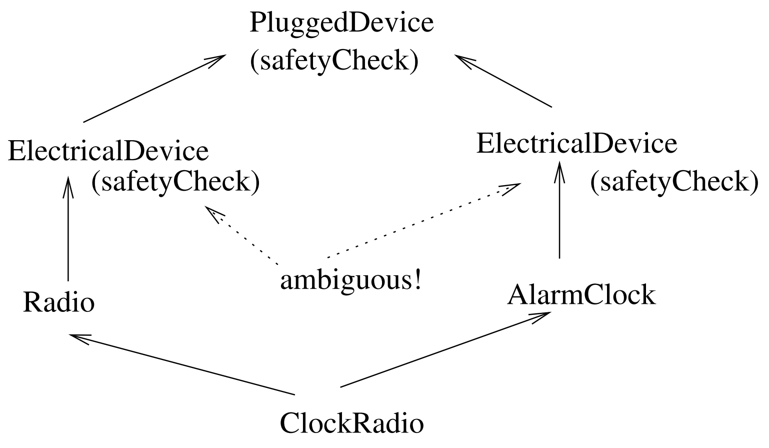
Not overridden We know that there is only one view of a radio alarm as a plugged device. Thus, if `safetyCheck` is not overridden in any derived class, then the original function within `PluggedDevice` will be called.



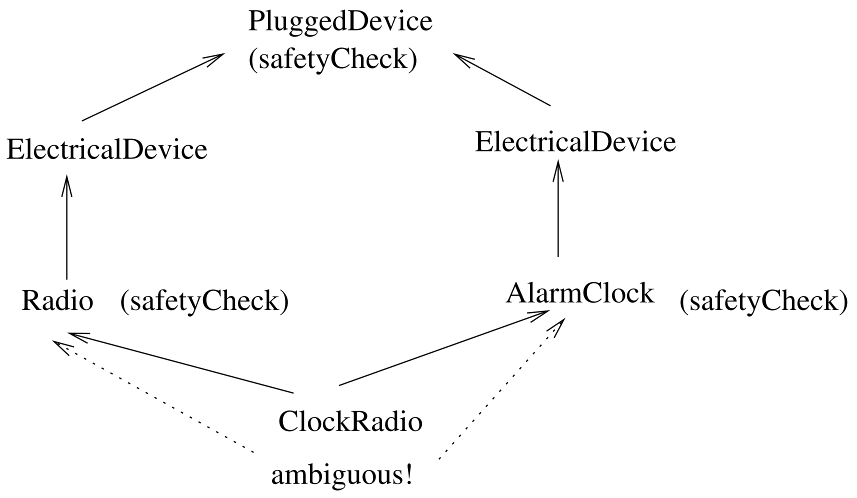
Overridden in `ClockRadio` If the clock radio redefines its own safety check, then there is no ambiguity, and, similarly to non-virtual inheritance, the safety check of the clock radio will be performed.



Overridden in `ElectricalDevice` By contrast, there are two possible ways of seeing a clock radio as an electrical device. However, contrary to repeated inheritance, those two points of view share the same `PluggedDevice` point of view, so it is impossible to know how to choose the adequate electrical device point of view to perform the safety check.



Overridden in `Radio` and `AlarmClock` Similarly, if the two base classes of `ClockRadio` each override the safety check operation, then there is no way to know how to choose between the radio and the alarm clock.

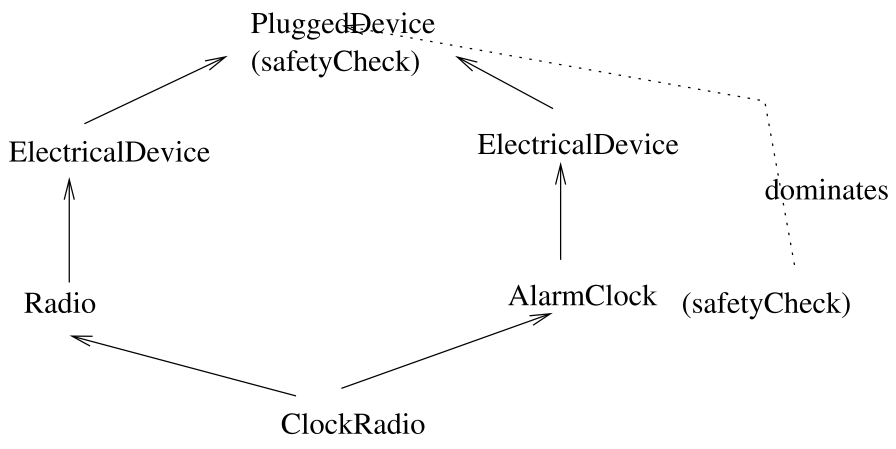


Overridden only in `AlarmClock` This is the most interesting case. For a clock radio, there are two candidates: either the original `PluggedDevice` safety check, or the `AlarmClock` safety check.

If the original `PluggedDevice` safety check were chosen, then, conceptually, the check would be incomplete, by not taking into account the alarm clock feature. This would be contradictory with the fact that, for a genuine alarm clock, the alarm clock safety check would be chosen.

To help towards a choice, C++ introduces the notion of *dominance*. Roughly speaking, whenever a virtual function call occurs on a virtual base, the dispatch examines all possible views of subtypes (taking repeated inheritance into account) of the most-derived class that share this common virtual base, and that override the virtual function. Then, if there is one candidate of which all other candidates are subtypes, then this candidate is said to *dominate*.

In our example, of the two candidates, `AlarmClock` dominates because `PluggedDevice` is a subtype of `AlarmClock`. Then, it will be chosen: such a choice is called the *final overrider* for `PluggedDevice::safetyCheck()` within a clock radio.



The most interesting thing is that this only final overrider will be called *independently* of the point of view chosen on a clock radio, as soon as `PluggedDevice` is a virtual base of this point of view. More concretely, if the safety check is performed from the `Radio` point of view of a clock radio:

```
Radio* pr = pra;
pr->safetyCheck();
```

Then, as `PluggedDevice` is a virtual base of `Radio`, its final overrider will be chosen: the `AlarmClock` candidate! This is called the *delegation to sister class*. Conceptually, choosing the final overrider ensures that the safety check is as complete as possible.

2.5 Construction and destruction

On top of multiple inheritance, C++ provides the programmer with notions of *object construction*. This mechanism makes the initialization of object fields easier. Moreover, along with *object destruction*, C++ allows the programmer to relate object lifetime with resource management (e.g. files, locks, etc.)

2.5.1 The lifetime of objects

C++ relates the *lifetime* of objects with their scope in the program. In the following excerpt:

```
{
  Radio ra;
  ...
}
```

The statement block creates an instance of the `Radio` class, and this instance is destroyed once the block exits.

2.5.2 Object initialization

C++ allows to execute a specific piece of code whenever an instance of some class is created. Such a piece of code is called a *constructor* of the class. A constructor is called explicitly when requesting the creation of an instance. Thus, it may have some arguments. Constructors are meant to correctly *initialize* objects, by giving values to their fields. For instance, when a `Radio` instance is created, its constructor can initialize the frequency of the radio with its argument:

```
struct Radio: ElectricalDevice {
  double frequency;
  Radio(double initFreq): frequency(initFreq) {}
};
```

2.5.3 RAII: Resource acquisition is initialization

Symmetrically to construction, C++ allows to execute a specific piece of code whenever an instance of some class is destructed. Such a piece of code is called the *destructor* of the class. By contrast to constructors, when a block defining an object is left, the corresponding destructor is implicitly called, so there may be only one destructor per class, and it accepts no arguments.

Constructors and destructors are meant to model a paradigm of resource management, called *resource acquisition is initialization* (RAII): roughly speaking, an object creation can take a resource, which will be released upon object destruction.

For instance, we build a high-level object model for writing data to a file, such that:

- an instance of `OutputFile` is associated to a physical file in the file system, whose name is given when requesting the creation of the `OutputFile` instance;
- when creating a `OutputFile` instance, the associated physical file is opened for writing operations
- when destroying a `OutputFile` instance, the associated physical file is closed

The `OutputFile` class, along its constructors and destructors, may be implemented as follows[5]:

```
#include <cstdio>

struct OutputFile {
  FILE* fileHandler;                          /* Physical file handler  */

  OutputFile(char* name) {                    /* Constructor */
    fileHandler = fopen(name, "w");           /* Perform system call */
  }

  ~OutputFile() {                             /* Destructor */
    fclose(fileHandler);                      /* Perform system call */
  }

  virtual void write(char* stringToWrite) {   /* File write */
    fputs(fileHandler, stringToWrite);        /* Perform system call */
```

5. We consider strings (`char*`) as scalar values, as well as physical file handlers (`FILE*`). Then, the system calls for physical files are as follows: `FILE* fopen(char*, "w")` opens a file in write-only mode, `int fputs(FILE*, char*)` writes a string into a file opened in write mode, and `int fclose(FILE*)` closes it. Those system types and calls are defined in the C++ `cstdio` standard library. Moreover, we perform no error handling: this issue requires exceptions, which are not treated in this thesis, as discussed in Section 13.2.1 (p. 327)

```
  }
};
```

Such a class allows hiding low-level implementation details (e.g. file handler and system calls), so that the following code actually produces a file named “toto” containing the string “Hello world!”:

```
main () {
  {
    OutputFile f = OutputFile("toto");  /* explicit constructor call,
                                           opens file "toto" */
    f.write("Hello world!");
  }                                     /* upon block exit,
                                           implicit destructor call,
                                           closes file "toto" */
}
```

When `f` enters its scope, a `OutputFile` instance is created and bound to it, thus the constructor specified by the programmer is called, actually opening the physical file.

Conversely, once `f` has left its scope, the `OutputFile` instance bound to it is destroyed, thus its destructor is called, actually closing the physical file.

Thus, resource management may be lifted to the level of the language: a file is opened if, and only if, its corresponding `OutputFile` instance is in the scope. This paradigm is called *resource acquisition is initialization (RAII).*

This is why, actually, it is worth saying that an object is *destructed* (rather than “destroyed”) upon scope exit: this ensures the safety property that an object must not survive its scope. This situation is adequate with the meaning of the verb *to destruct*: “to destroy for safety purposes”.

2.5.4 Construction and destruction order

Assume now that we need to write data to a file on a non-cooperative device allowing access by only one process at a time (e.g. a tape). Then, we have to put a lock on the device before opening the file, and release the lock after closing the file. Assume that there exists a class `DeviceLock` such that any instance locks a device on its creation, and releases the lock on its destruction:

```
struct DeviceLock {
  DeviceLock(char* device); /* constructor takes the lock */
  ~DeviceLock();            /* destructor releases the lock */
};
```

Then, writing on a file on a locked device may be implemented by the following `LockedDeviceFile` class, having two fields: the device lock, and the file to write.

```
struct LockedDeviceFile {
  /* WARNING: this order is important */
  DeviceLock deviceLock;
  File       file;
  LockedDeviceFile(char* device, char* fileName):
    /* order is irrelevant here */
    File(fileName),
    DeviceLock(device)
  {}

  ~LockedDeviceFile() {}
};
```

Its constructor takes two arguments: the device to lock and the name of the file to write. This constructor explicitly calls the constructors for each of its fields. Then, C++ guarantees that the fields are constructed by the corresponding explicit calls in the constructor of `LockedDeviceFile`. Moreover, they are constructed in the order of declaration of the fields, not of the constructor calls. The following class ensures that the device is locked before opening the file.

Conversely, C++ guarantees that, after executing the body of the `~LockedDeviceFile()` destructor, the device lock and the file are destructed upon destruction of a `LockedDeviceFile`, in the reverse order of field declaration. This ensures that the lock is released from the device only once the file has been closed.

This example illustrates the C++ principle enforcing two objects to destruct in the reverse order of their construction.

2.5.5 Inheritance

Now consider creating a HTML file. A HTML file is a file, which must begin with `<html>` and end with `</html>`. Moreover, a HTML file should not contain raw text: any text should be surrounded by tags such as `<p>`... `</p>`[6] We would like to design a `HTMLFile` class, such that the following example:

```
main () {
  {
    HTMLFile f = HTMLFile("index.html");
    f.write("hello world!");
    f.write("42");
  }
}
```

produces a file named `index.html` containing:

```
<html>
<p>hello world!</p>
<p>42</p>
</html>
```

`HTMLFile` would hide any implementation details (system calls, HTML tags) from the user, who would thus be provided a high level of abstraction.

6. For simplicity, we omit the transformation of special characters, in particular < and >. We also omit the `<head>` and `<body>` structure, and in particular the mandatory `<title>` tag.

Construction of Base Classes One approach is to enforce writing the header `<html>` upon creation of the file, and the footer `</html>` upon its destruction.

This may be realized with the help of the following `HeaderedFooteredFile` class, deriving from `OutputFile`, and equipped with a constructor taking two extra arguments, the header and the footer:

```
struct HeaderedFooteredFile: OutputFile {
  char* foot;                        /* Keep footer for later use      */

  HeaderedFooteredFile(char* name, char* header, char* footer):
                                     /* Constructor                    */
    OutputFile(name),                /* Initialize base class:
                                        open underlying file           */
    foot(footer)                     /* Initialize field to keep footer
                                        for later use                  */
  {
    this->write(header);             /* Write the header               */
  }

  ~HeaderedFooteredFile() {          /* Destructor                     */
    this->write(foot);               /* Write the footer               */
  }
};
```

When a `HeaderedFooteredFile` is created, its constructor is called, performing the following operations in this order:

1. open the underlying file, by initializing the `OutputFile` base class by calling its constructor;
2. then, keep the footer for use upon file closure, by initializing the `foot` field;
3. finally, write the header to the underlying file, by executing the constructor body

Conversely, the destructor of the underlying `OutputFile` base class is implicitly called only once the destructor of `HeaderedFooteredFile` exits: this ensures both that the file is actually closed, and that the footer is actually written before closing the file.

As an HTML file is a file with the specific header `<html>` and the specific footer `</html>`, we define a `HTMLFile` class deriving from `HeaderedFooteredFile`, calling its constructor with the appropriate arguments:

```
struct HTMLFile: HeaderedFooteredFile {
  HTMLFile(char* name): HeaderedFooteredFile(name, "<html>", "</html>") {}
  ...
}
```

Virtual functions during construction To ensure that each output to an HTML file be surrounded by appropriate `<p>...</p>` tags, we override the `write` virtual function, calling the `OutputFile::write` function of the underlying file to write the surrounding tags:

```
struct HTMLFile: HeaderedFooteredFile {
  HTMLFile(char* name): HeaderedFooteredFile(name, "<html>", "</html>") {}

  void write(char* stringToWrite) {
    this->OutputFile::write("<p>");
    this->OutputFile::write(stringToWrite);
    this->OutputFile::write("</p>");
  }
}
```

Now the question is: when creating a HTML file, which `write` function is called within the constructor of `HeaderedFooteredFile` to write the header? Logically, it must not be the function of the most-derived class `HTMLFile`: the `<html>` header must not be surrounded by `<p>`... `</p>` tags. It is necessarily the function of the underlying `OutputFile` base class.

This is a general point in C++: during the construction of a base class, any derived classes are ignored until the constructor exits. This is a contrast to other languages such as Java, which would invariably use the `write` method defined in the most-derived `HTMLFile` class.

The same principle holds for destruction: when disposing of a `HTMLFile`, the `</html>` footer is written using the `write` function of the underlying file, not of the most-derived `HTMLFile` class.

C++ also requires that *indirect* calls to virtual functions (i.e. calls to virtual functions from outside the constructor body) during construction also follow this paradigm. That is, if the constructor for `HeaderedFooteredFile` is defined as follows, then it still has to give the same result.

```
void writeToHFFile(HeaderedFooteredFile* hf, char* stringToWrite) {
  hf->write(stringToWrite);
}

HeaderedFooteredFile(char* name, char* header, char* footer):
  OutputFile(name),
  foot(footer)
{
  writeToHFFile(this, header); /* performs indirect call to write() */
}
```

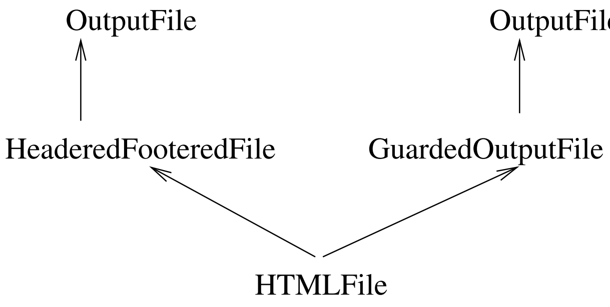
2.5.6 Virtual inheritance

The object model for an HTML file can still be improved. Indeed, the fact that every output must be surrounded by tags is a priori independent of the fact that the entire file must be surrounded by a header and a footer. Therefore, we define a `GuardedOutputFile` class ensuring that any output to the file is surrounded by a preface and a postface:

```
struct GuardedOutputFile: OutputFile {
  char* pre;
  char* post;
  GuardedOutputFile(char* name, char* preface, char* postface):
    OutputFile(name),
    pre(preface),
    post(postface)
  {}

  ~GuardedOutputFile() {}

  void write(char* stringToWrite) {
    this->OutputFile::write(pre);
    this->OutputFile::write(stringToWrite);
    this->OutputFile::write(post);
  }
};
```

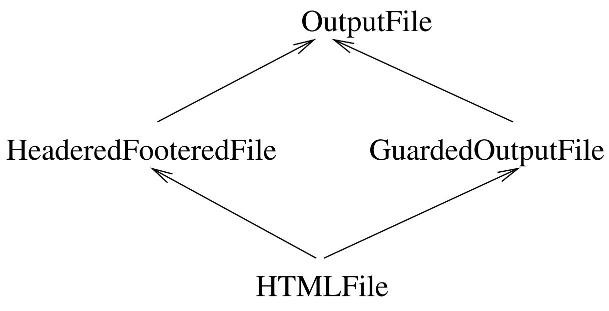
Now a HTML file inherits from both a `HeaderedFooteredFile` and a `GuardedOutputFile`:

```
struct HTMLFile: HeaderedFooteredFile, GuardedOutputFile {
  HTMLFile(char* name):
    HeaderedFooteredFile(name, "<html">, "</html>"),
    GuardedOutputFile(name, "<p>", "</p>")
  {}
};
```



However, the two base classes must agree on the output file to write to: it must not be opened twice. Repeated inheritance would lead to two different views of a HTML file as a file, which would incorrectly open the file twice, once for each view. We therefore use virtual inheritance:

```
struct HeaderedFooteredFile: virtual OutputFile {...};
struct GuardedOutputFile: virtual OutputFile {...};
/* class definitions remain otherwise unchanged */
```



C++ enforces that virtual base classes are initialized (and destroyed) only once. `GuardedOutputFile` and `HeaderedFooteredFile` may virtually inherit from `OutputFile`, to ensure that the constructor of `OutputFile` is called only once. To ensure this, the constructor of the virtual base class is called directly from the most-derived class, and the virtual base constructor calls from base classes are ignored. Thus, defining:

```
struct HTMLFile: HeaderedFooteredFile, GuardedOutputFile {
  HTMLFile(char* name):
    OutputFile(name),
    HeaderedFooteredFile(name, "<html">, "</html>"),
    GuardedOutputFile(name, "<p>", "</p>")
  {}
};
```

ensures that the `OutputFile` virtual base class is initialized before the other base classes, and the `OutputFile` constructor calls within `HeaderedFooteredFile` and `GuardedOutputFile` are ignored, so their `name` arguments are actually useless for a HTML file.

2.5.7 Summary of construction and destruction principles

Those simple examples illustrate the need for the following C++ principles for managing object construction and destruction in the presence of C++ multiple inheritance:

- Virtual bases (and their non-virtual bases) are constructed before the non-virtual bases of a most-derived object
- An object first constructs its bases
- Fields are constructed after bases
- Two bases or fields, are constructed in their declaration order
- Two bases or fields are destructed in the reverse order of their construction

Although those examples cover all those principles, they do not cover the interactions between them. However, such interactions are not infrequent in practice: the C++ `iostream` standard library defines a much more developed and precise object model for input and output streams. This object model makes use of all the paradigms (multiple inheritance, RAII, ...) presented in this chapter, all at the same time. That points out the importance of precisely clarifying the intricacies between the different features provided by C++ multiple inheritance and object construction and destruction.

Chapter 3

Setting and notations

In this chapter, we introduce general-purpose mathematical notations used throughout this thesis, along with the formalism of small-step semantics for programming languages.

3.1 Overall notations

This section introduces general-purpose mathematical notations and conventions used throughout this thesis.

Specific notations will be introduced and described following the course of this thesis. However, for a quick reference at a glance, all notations are gathered in the index of notations (p. 357). Most notations are inspired from Coq [4]

Typographical conventions We use fixed-size serif font for language (syntactic) constructs as in `setDynType`, by contrast to variable-size sans-serif font for abstract (semantic) concepts as in $\mathsf{setDynType}$.

Booleans Booleans (set $\mathbb{B}$) are written true and false.

Integers

- To avoid notation clashes, the set of strictly positive integers will be written $\mathbb{N}^{>0}$.
- $\mathsf{S} : \mathbb{N} \to \mathbb{N}$ denotes the successor function among nonnegative integers.
- $[a, b]$ denotes the “closed” integer interval, i.e. the interval from a to b both included:

$$[a, b] \underset{\text{def.}}{=\!=} \{x \in \mathbb{Z} : a \leq x \leq b\}$$

- $[a, b)$ denotes the “semi-open” interval, i.e. the integer interval from a to b, including a but excluding b:

$$[a, b) \underset{\text{def.}}{=\!=} \{x \in \mathbb{Z} : a \leq x < b\}$$

Sets

- $\varnothing$ is the empty set
- $x \in S$, or $S \ni x$, denotes the fact that x is an element of the set S
- $A \subseteq B$, or $B \supseteq A$, denotes the fact that every element of A is an element of B
- we retain the usual notations $\cap, \cup$ for set intersection and union

- for any sets A, B: $A \# B$ if, and only if, A and B are disjoint, i.e. $A \cap B = \varnothing$.
- for any sets A, B: $A \uplus B$ is the union $A \cup B$ but **assuming** that A and B are disjoint (otherwise, $A \uplus B$ is undefined).
- consequently, in an enumerative description of a set $\{a, b, c\}$, we do **not** assume $a \neq b$, $b \neq c$ or $c \neq a$. Those conditions are enforced by writing $\{a\} \uplus \{b\} \uplus \{c\}$ instead.
- the set of all subsets of S is written $\mathcal{P}(S)$.

Functions If $f : A \to B$ is a function, then, for any $a \in A$ and $b \in B$, $f[a \leftarrow b]$ is the function where a becomes associated to b whereas any other image is unchanged:

$$f[a \leftarrow b] : \begin{array}{lcc} A & \to & B \\ a & \mapsto & b \\ a' \neq a & \mapsto & f(a') \end{array}$$

Optional values and partial functions

- for any set S such that $\bot \notin S$, we pose $S^? \overset{}{\underset{\text{def.}}{=\!=}} S \uplus \{\bot\}$. The value $\bot$ is said to be *undefined.*
- Thus, a partial function f from A to B may be seen as a total function from A to $B^?$, where an undefined $f(a)$ shall actually take value $\bot$.

By convention, any set S is assumed to not contain $\bot$, unless S can be written $T^?$ for some T. In grammar definitions, if s is a syntactic category, then $s^?$ represents zero or one occurrence of s (or *optional* s)

Lists

- ϵ is the empty list
- $a :: q$ is the list starting with an element a and continuing with the tail list q
- for any set S, we write $S^\star$ the set of the lists of S. In grammar definitions, if s is a syntactic category, then $s^\star$ represents any finite number of occurrences of s
- $x \in l$, or $l \ni x$, denotes the fact that x is an element of the list l

Operations over lists

- $+\!\!+$ is the *append* (list concatenation) operator: for any list l, we have $\epsilon +\!\!+ l \underset{\text{def.}}{=\!=} l$ and $(a :: q) +\!\!+ l \underset{\text{def.}}{=\!=} a :: (q +\!\!+ l)$.
- $\mathsf{filter}_f(l)$ is the list l from which all elements a such that $f(a) \neq \mathsf{true}$ have been removed, recursively defined as follows: $\mathsf{filter}_f(\epsilon) = \epsilon$ and $\mathsf{filter}_f(a :: q) = \begin{cases} a :: \mathsf{filter}_f(q) & \text{if } f(a) = \mathsf{true} \\ \mathsf{filter}_f(q) & \text{otherwise} \end{cases}$
- $\mathsf{map}[f](l)$ is the list l where each element a has been replaced with its image $f(a)$. That is, $\mathsf{map}[f](\epsilon) = \epsilon$ and $\mathsf{map}[f](a :: q) = f(a) :: \mathsf{map}[f](q)$.
- $\mathsf{rev}(l)$ is the *reverse* of the list l, i.e. the list of elements of l given in the reverse order of l. That is, $\mathsf{rev}(\epsilon) = \epsilon$ and $\mathsf{rev}(a :: q) = \mathsf{rev}(q) +\!\!+ a :: \epsilon$.
- $\mathsf{length}(l)$ is the length of a list l: $\mathsf{length}(\epsilon) \underset{\text{def.}}{=\!=} 0$ and $\mathsf{length}(a :: l') \underset{\text{def.}}{=\!=} 1 + \mathsf{length}(l')$
- $+\!\!+'$ is the operator "append without duplicate", defined as $l_1 +\!\!+' l_2 = l_1 +\!\!+ \mathsf{filter}_{x \mapsto x \notin l_1}(l_2)$
- first is a function defined on non-empty lists, such that $\mathsf{first}(a :: l') \underset{\text{def.}}{=\!=} a$ for all a, l'.
- last is a function defined on non-empty lists, computing their last elements: $\mathsf{last}(a :: \epsilon) \underset{\text{def.}}{=\!=} a$ and $\mathsf{last}(a :: b :: l') \underset{\text{def.}}{=\!=} \mathsf{last}(b :: l')$ for all a, b, l'.

Finite maps For any two sets A, B, we write $f : A \nrightarrow B$ for a partial function f from A to B with a *finite domain*: $\{a \in A : f(a) \neq \bot\}$ is finite.[1]

In practice (e.g. in our Coq development), such a function is represented as a finite list of tuples, or *association list* : $f \in (A \times B)^\star$, such that, $(a, b) \in f$ if, and only if, $f(a) = b \neq \bot$. This implies that a finite map must be entirely computable within a finite amount of time.

A finite map is *empty* if, and only if, its domain is empty (i.e. it maps anything to $\bot$). It is then written $\emptyset$

Records A *record* can be considered as a tuple whose components are named: that is, a tuple with projection functions. A record type *Record* defined as follows:

$$\begin{array}{l} \mathit{Record} = \\ \{ \\ \qquad \begin{array}{ccc} c_1 : & V_1 & ; \\ \vdots & \vdots & ; \\ c_n : & V_n & ; \end{array} \\ \} \end{array}$$

can be seen as follows:

$$\begin{array}{lcl} \mathit{Record} & \overset{}{\underset{\text{def.}}{=\!=}} & V_1 \times \cdots \times V_n \\ c_1 & : & ((v_1, \ldots, v_n) \in \mathit{Record}) \mapsto (v_1 \in V_1) \\ \vdots & & \vdots \\ c_n & : & ((v_1, \ldots, v_n) \in \mathit{Record}) \mapsto (v_n \in V_n) \end{array}$$

If r is a record, then:

- the value $c(r)$ of its component c is written $r.c$
- $r[c \leftarrow v]$ denotes a new record whose value of component c has become v whereas all other components remain unchanged:

$$\begin{aligned} r[c \leftarrow v].c &= v \\ r[c \leftarrow v].c' &= r.c' && (c' \neq c) \end{aligned}$$

- In particular, if $r.c$ is a function from A to B, then $r[c(a) \leftarrow b]$ denotes a new record verifying:

$$\begin{aligned} r[c(a) \leftarrow b].c(a) &= b \\ r[c(a) \leftarrow b].c(a') &= r.c(a') && (a' \neq a) \\ r[c(a) \leftarrow b].c' &= r.c' && (c' \neq c) \end{aligned}$$

In other words:

$$r[c(a) \leftarrow b] = r[c \leftarrow r.c[a \leftarrow b]]$$

1. Symbol $\nrightarrow$ is borrowed from the Z notation [11, 78, 41].

Infinite streams Let S be a set. An infinite stream of elements of S is coinductively written $s :::: \mathfrak{S}$ where $s \in S$ and $\mathfrak{S}$ is an infinite stream of elements of S.

Then, we recursively define the concatenation $l +\!\!\!+\!\!\!+ \mathfrak{S}$ of a finite list l of elements of S and an infinite stream $\mathfrak{S}$ of elements of S as follows:

$$\epsilon +\!\!\!+\!\!\!+ \mathfrak{S} \underset{\text{def.}}{=\!=} \mathfrak{S} \qquad\qquad (s :: l) +\!\!\!+\!\!\!+ \mathfrak{S} \underset{\text{def.}}{=\!=} s :::: (l +\!\!\!+\!\!\!+ \mathfrak{S})$$

3.2 Small-step operational semantics

We present here the formal guidelines allowing us to define, for each language that we shall consider in our work, a syntax and a small-step operational semantics [2]. For formal definitions of program behaviours and semantics preservation as foundations to the proofs of correctness of compilers, please refer to Appendix B (p. 337).

3.2.1 Observational semantics of traces

To relate two languages through a verified compiler, we must focus on a common, language-independent subset of their semantics. Indeed, the two languages may act completely differently in internal memory representation, but the results expected by the end user should not be affected by such differences.

To this purpose, we focus on the *observational semantics* [3] of languages: during its execution, a program produces an *event trace*, representing a (finite or infinite) sequence of inputs/outputs, and, if it terminates, a return value.

We consider a compiler to *preserve the semantics* of languages if the compiled program produces the same event traces and the same return value as the source program. Internals such as memory usage or time consumption are not considered part of the observational semantics (unless they are accurately and accordingly modeled in the trace semantics, which is out of the scope of our work). More formally:

Hypothesis 3.2.1. *Throughout this thesis, we assume the existence of a set $\mathfrak{E}$ of* events*, and a set $\mathcal{Z}$ of* return values*.*

Then, the trace semantics will be shared among all language semantics considered here:

***Definition* 3.2.1.** *A* finite trace *is a (finite) list of events.*
An infinite trace *is an infinite stream of events.*

3.2.2 Transition system, programming language

***Definition* 3.2.2.** *A* transition system *is a tuple $(\mathfrak{S}, (\rightarrow, \mathfrak{I}, \mathfrak{F}))$ where:*

- *$\mathfrak{S}$ is the set of* execution states *(or* states *for short)*
- *$\rightarrow \subseteq (\mathfrak{S} \times \mathfrak{E}^? \times \mathfrak{S})$ is the* transition relation *or* step relation*. A member $(s_1, \mathfrak{e}^?, s_2)$ of this relation, written $s_1 \underset{\mathfrak{e}^?}{\rightarrow} s_2$, is called a* valid transition step*, a* transition step*, or a* transition *for short. The transition is said to be* silent *if $\mathfrak{e}^? = \bot$, and then it is simply written $s_1 \rightarrow s_2$. Otherwise, if $\mathfrak{e}^? = \mathfrak{e} \neq \bot$, then the transition is said to* produce *the event $\mathfrak{e}$*

2. Coq development: theory `Smallstep`.
3. Coq development: theory `Events`.

- $\mathfrak{I} \subseteq \mathfrak{S}$ *is the set of* initial states*, assumed non-empty*
- $\mathfrak{F} \subseteq (\mathfrak{S} \times \mathcal{Z})$ *is the set of* final states*, each of which is associated with a* return value*. If* $(s, z) \in \mathfrak{F}$*, then* s *is assumed to be* stuck*: there can be no event* $\mathfrak{e}^?$ *and no state* s' *such that* $s \xrightarrow[\mathfrak{e}^?]{} s'$*.*

$(\rightarrow, \mathfrak{I}, \mathfrak{F})$ *is called the* operational semantics *(or* semantics *for short) of the transition system.*

Any final state is assumed to be stuck, but the converse is false: not all stuck states are final.

***Notation* 3.2.3.** *For any transition system* $(\mathfrak{S}, (\rightarrow, \mathfrak{I}, \mathfrak{F}))$ *and any return value* z*,* $\mathfrak{F}_z$ *denotes the set of all final states with return value* z*:*

$$\mathfrak{F}_z \overset{}{\underset{\text{def.}}{=\!=}} \{s : (s, z) \in \mathfrak{F}\}$$

***Definition* 3.2.4.** *A* programming language*, or* language *for short, is a tuple* $(\mathbb{P}, \mathbb{S})$ *where:*
- $\mathbb{P}$ *is the set of all programs written in this language. A description of* $\mathbb{P}$ *(e.g. a grammar) is called a* syntax *of the language*
- $\mathbb{S}$ *is a partial function defined on programs, such that for any program* $P \in \mathbb{P}$ *such that* $\mathbb{S}(P)$ *is defined (such a program is said to be* well-formed*), then* $\mathbb{S}(P)$ *is a transition system, called the* meaning *(or* semantics*) of* P*. The function* $\mathbb{S}$ *itself is called the* semantics *of the language.*

3.2.3 Sequences of transition steps

***Definition* 3.2.5.** *Let* $(\mathfrak{S}, (\rightarrow, \mathfrak{I}, \mathfrak{F}))$ *be a transition system.*

Inductively, for any states $s_0, s_2 \in \mathfrak{S}$ *and any finite trace* $\mathfrak{t} \in \mathfrak{E}^\star$*, we write* $s_0 \xrightarrow[\mathfrak{t}]{+} s_2$ *when* s_2 *is reachable from* s_0 *through a finite non-zero number of transitions producing trace* $\mathfrak{t}$ *according to the following rules:*

$$\frac{s_0 \rightarrow s_2}{s_0 \xrightarrow[\epsilon]{+} s_2} \qquad \frac{s_0 \xrightarrow[\mathfrak{e}]{} s_2 \qquad \mathfrak{e} \neq \bot}{s_0 \xrightarrow[\mathfrak{e}::\epsilon]{+} s_2} \qquad \frac{s_0 \xrightarrow[\mathfrak{t}_1]{+} s_1 \xrightarrow[\mathfrak{t}_2]{+} s_2}{s_0 \xrightarrow[\mathfrak{t}_1 +\!\!+ \mathfrak{t}_2]{+} s_2}$$

Then, we say that s_2 *is* finitely reachable *(or* reachable*) from* s_0*, written* $s_0 \xrightarrow[\mathfrak{t}]{\star} s_2$*, if and only if* $(s_0, t) = (s_2, \epsilon)$ *or* $s_0 \xrightarrow[\mathfrak{t}]{+} s_2$*.*

3.2.4 Built-in types, values and operations

All languages considered in this thesis manipulate *built-in types* (for instance `int`, `char`, `double`, ...) and *built-in values* (for instance −18, `'c'`, 3.1415926535897932, ...).

***Notation* 3.2.6.** *Let BuiltinTypes be the set of* built-in types*, and Builtin be the set of* built-in values*.*

We then assume the existence of built-in operators to manipulate them, such as integer or floating-point arithmetics.

Hypothesis 3.2.2. *Let Op be the set of* built-in operators*. Then, for each built-in operator $op \in Op$, we assume that there exists a* built-in operation $[op] : \mathit{Builtin}^\star \to \mathcal{P}(\mathfrak{E}^? \times \mathit{Builtin}^?)$*, which is also called the* semantics *of op. If $[op](B) \ni (\mathfrak{e}^?, b)$, then B is the list of the* arguments *passed to the operator op, and the operation is said to* produce *an event $\mathfrak{e}^?$ (possibly) and (possibly) a* return value *b.*

In all our languages, evaluation of built-in operators shall be the only possible non-silent transition steps. That is, no artifacts observable by the end user shall be caused by operations related to C++ multiple inheritance (casts, virtual function calls, ...)

Part I

Verification of C++ object layout

Chapter 4

The semantics of C++ multiple inheritance

In this chapter, we formalize the notions of C++ objects and subobjects in the presence of multiple inheritance and structure fields [1]. This leads us to present a language, which we call s++, featuring scalar and structure field and array accesses, static and dynamic casts, and virtual function dispatch.

Our starting points are Rossie et al. [74], who first studied C++ multiple inheritance with both virtual and non-virtual inheritance; and Wasserrab et al. [85], who formalized their approach in Isabelle to define a language called *CoreC*++ featuring scalar fields, static and dynamic casts, and virtual function dispatch. We extend those works by featuring fields of structure and structure array types, also known as *embedded structures* and embedded structure arrays.

4.1 Classes and subobjects

4.1.1 Class hierarchy

***Definition* 4.1.1.** *A* class definition *is composed of:*

– *the declarations of* class member functions*, or* methods*, with their names, argument types and return type, and an indication of whether they are virtual or not. (However, no code is provided.)*
– *the declaration of* data members*, or* fields*, which can be either a scalar (as in [85]) or an embedded structure*
– *the names of the direct (virtual or non-virtual) bases of the class*

A class hierarchy *is a finite map from class names to class definitions.*

1. Coq development: theory `Cplusconcepts`.

$t \in \mathit{ScalarType}$	$::=$	$\mathit{BuiltinType}$	*Built-in type*
	$\mid$	$\mathit{ClassName}*$	*Pointer to an object of class ClassName*
n	$\in$	$\mathbb{Z}$	*Structure array size*
$\mathit{msig} \in \mathit{MethodSig}$	$::=$	$\mathit{ScalarType}^{?}\mathit{MethodName}(\mathit{ScalarType}^{\star})$	*Class member function (method) signature*
$\mathit{MethodDecls}$	$=$	$\mathit{MethodSig} \mathrel{+\!\!\!\!\twoheadrightarrow} \mathbb{B}$	*Method virtualness*
$\mathit{fsig} \in \mathit{FieldSig}$	$::=$	`scalar` $\mathit{ScalarType}\ \mathit{FieldName};$	*Data member*
	$\mid$	`struct` $\mathit{ClassName}[n]\ \mathit{FieldName};$	*(field)*
Base	$::=$	$\mathit{ClassName} \mid$ `virtual` $\mathit{ClassName}$	*Direct bases*
$\mathit{ClassDef}$	$::=$	$\mathit{Base}^{\star}\{\mathit{FieldSig}^{\star}\ \mathit{MethodDecls}\};$	*Class definition*
$\mathit{Hierarchy}$	$=$	$\mathit{ClassName} \mathrel{+\!\!\!\!\twoheadrightarrow} \mathit{ClassDef}$	*Class hierarchy*

A class member function (a.k.a. method) is characterized by its name and the types of its arguments.

A data member (a.k.a. field) is characterized by its name and type. A class may declare two fields with the same name as long as they have different types. As regards non-scalar fields, we only consider structure array fields: a structure field of type `C` can be seen as a structure array field of type `C[1]`, with only one cell.

***Notation* 4.1.2.** *Given a class C, we write:*

- $\mathcal{DNV}(C) \in \mathit{ClassName}^{\star}$ *the list of its direct non-virtual bases*
- $\mathcal{DV}(C) \in \mathit{ClassName}^{\star}$ *the list of its direct virtual bases*
- $\mathcal{D}(C) \in ($`virtual`$^{?}\ \mathit{ClassName})^{\star}$ *the list of its direct (virtual or non-virtual) bases*
- $\mathcal{F}(C) \in \mathit{FieldSig}^{\star}$ *the list of its (scalar or structure array) fields*
- $\mathcal{M}(C) \in \mathit{MethodDecls}$ *its methods (a function retrieving* true *if the class member function is virtual,* false *if the class member function is non-virtual, and $\perp$ if C does not define the method).*

We write $\mathcal{C}$ for the set of defined classes, i.e. the set of class names having a class definition through the hierarchy map.

Example 4.1.1. Consider for instance the following C++ code:

```
struct A            { int i; virtual void f(float); };
struct B: virtual A { C* c;  };
struct C: A, B      { float j; A a[2]; B b; void f(float); };
```

Then, it corresponds to the following hierarchy:

$$
\begin{array}{lcll}
\mathit{FieldName} & = & \{i, c, j, a, b\} & \\
\mathit{ClassName} & = & \{A, B, C\} & \\
\\
\mathcal{DNV}(A) & = & \epsilon & \\
\mathcal{DV}(A) & = & \epsilon & \\
\mathcal{D}(A) & = & \epsilon & \\
\mathcal{F}(A) & = & (\mathtt{scalar}\ \mathtt{int}\ i) :: \epsilon & \\
\mathcal{M}(A) & : & f(\mathtt{float}) \mapsto \mathsf{true} & \text{matches } \mathtt{virtual\ void\ f(float);} \\
\\
\mathcal{DNV}(B) & = & \epsilon & \\
\mathcal{DV}(B) & = & A :: \epsilon & \\
\mathcal{D}(B) & = & (\mathtt{virtual}\ A) :: \epsilon & \\
\mathcal{F}(B) & = & (\mathtt{scalar}\ C\ c) :: \epsilon & \text{matches } \mathtt{C*\ c;} \\
\mathcal{M}(B) & = & \emptyset & B \text{ declares no methods} \\
\\
\mathcal{DNV}(C) & = & A :: B :: \epsilon & \\
\mathcal{DV}(C) & = & \epsilon & \\
\mathcal{D}(C) & = & A :: B :: \epsilon & \\
\mathcal{F}(C) & = & (\mathtt{scalar}\ \mathtt{float}\ j) :: & \\
 & & (\mathtt{struct}\ A[2]\ a) :: & \text{matches } \mathtt{A\ a[2];} \\
 & & (\mathtt{struct}\ B[1]\ b) :: \epsilon & \text{matches } \mathtt{B\ b;} \\
\mathcal{M}(C) & : & f(\mathtt{float}) \mapsto \mathsf{false} & \text{matches } \mathtt{void\ f(float);} \\
\\
\mathcal{C} & = & \{A, B, C\} &
\end{array}
$$

Even though C overrides f from its base A which declares f as virtual, C is not required to declare f as virtual.

4.1.2 Inheritance paths

4.1.2.1 Non-virtual inheritance

Example 4.1.2. Consider the following code:

```
struct A             { int a; };
struct B1: A         {};
struct B2: A         {};
struct C : A         {};
struct D : B1, B2, C {};

main() {
  D    d;
  B1* b1 = (B1*) &d;
  B2* b2 = (B2*) &d;
  A * a1 = (A *) b1;
  A * a2 = (A *) b2;
}
```

$$
\begin{tikzcd}
A & A & A \\
B_1 \arrow[u] & B_2 \arrow[u] & C \arrow[u] \\
 & D \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}
$$

An instance of `D` can be seen as having *three* different "copies" of `A`: one reachable through the direct non-virtual base `B1`, one through `B2`, and one from `C`. This is called *non-virtual inheritance*, or *repeated inheritance*. Each "copy", called an *inheritance subobject* of D of static type A, has its own value for the field a. Therefore, it is necessary to distinguish those subobjects. Following Rossie et al. [74]., those subobjects can be distinguished by the *paths* through which they are reached. More precisely:

***Definition* 4.1.3 (Non-virtual path).** *A list l of class names is a* **n**on-**v**irtual inheritance path *from C to A (written $C \prec\langle l\rangle\xrightarrow{\mathcal{NV}} A$) if, and only if:*

- *either $C = A$ and $l = A :: \epsilon$. This path is called the* trivial path.
- *or there exists a non-virtual direct base B of C and a non-virtual path l' from B to A, such that $l = C :: l'$.*

$$\frac{}{C \prec\langle C :: \epsilon\rangle\xrightarrow{\mathcal{NV}} C} \qquad \frac{B \in \mathcal{DNV}_C \qquad B \prec\langle l'\rangle\xrightarrow{\mathcal{NV}} A}{C \prec\langle C :: l'\rangle\xrightarrow{\mathcal{NV}} A}$$

***Definition* 4.1.4 (Non-virtual base).** *A is a* non-virtual base *of C if, and only if, A is reachable through a non-trivial* non-virtual path *from C.*

If B is a direct non-virtual base of C, then any non-virtual base of B is an indirect *non-virtual base of C.*

In Example 4.1.2 (p. 73), A is an indirect non-virtual base of D through three *different* paths:

- $D :: C :: A :: \epsilon$,
- $D :: B_1 :: A :: \epsilon$ which corresponds to the a_1 pointer,
- and $D :: B_2 :: A :: \epsilon$ which corresponds to the a_2 pointer.

The C++ Standard [42] dictates $a_1 \neq a_2$: even though they are pointers to subobjects of the same C object, those subobjects are accessible through *different non-virtual paths*, so those pointers must be different.

Concatenation

Lemma 4.1.1. *Any non-virtual path l from any class C to any class B begins with C and ends with B.*

Lemma 4.1.2 (Non-virtual path concatenation). *If $C :: l$ is a non-virtual path from C to B and $B :: l'$ is a non-virtual path from B to A, then $C :: l +\!\!\!+\, l'$ is a non-virtual path from C to A written $(C :: l)@_{\mathsf{Repeated}}(B :: l')$.*

$$(C :: l)@_{\mathsf{Repeated}}(B :: l') \underset{\text{def.}}{=\!=} C :: l +\!\!\!+\, l' \qquad \frac{C \prec\langle C :: l\rangle\xrightarrow{\mathcal{NV}} B \prec\langle B :: l'\rangle\xrightarrow{\mathcal{NV}} A}{C \prec\langle (C :: l)@_{\mathsf{Repeated}}(B :: l')\rangle\xrightarrow{\mathcal{NV}} A}$$

4.1.2.2 Virtual inheritance

C++ features *virtual inheritance*, allowing some subobjects accessible through apparently different paths to be identical. Rossie et al. [74] give a notion of path allowing such subobjects to be represented by *the same* path.

***Definition* 4.1.5.** *A class A is a* virtual base *of C (written $C \overset{\mathcal{V}}{\to} A$) if, and only if:*
- *either A is a direct virtual base of C*
- *or there is a direct (virtual or non-virtual) base B of C such that A is a virtual base of B. In this case, A is said to be an* indirect *virtual base of C.*

$$\frac{A \in \mathcal{DV}(C)}{C \overset{\mathcal{V}}{\to} A} \qquad\qquad \frac{B \in \mathcal{DNV}(C) \cup \mathcal{DV}(C) \qquad B \overset{\mathcal{V}}{\to} A}{C \overset{\mathcal{V}}{\to} A}$$

***Definition* 4.1.6 (Inheritance path).** *Let $h \in \{\mathsf{Repeated}, \mathsf{Shared}\}$ and l be a list of class names. The pair $\sigma = (h, l)$ is an* inheritance path *from a class C to a class A (written $C \prec\!\langle \sigma \rangle\!\overset{\mathcal{I}}{\rightarrowtail} A$) if, and only if:*
- *either $h = \mathsf{Repeated}$ and l is a non-virtual path from C to A*
- *or $h = \mathsf{Shared}$ and there exists a virtual base B of C such that l is a non-virtual path from B to A.*

$$\frac{C \prec\!\langle l \rangle\!\overset{\mathcal{NV}}{\rightarrowtail} A}{C \prec\!\langle (\mathsf{Repeated}, l) \rangle\!\overset{\mathcal{I}}{\rightarrowtail} A} \qquad\qquad \frac{C \overset{\mathcal{V}}{\to} B \prec\!\langle l \rangle\!\overset{\mathcal{NV}}{\rightarrowtail} A}{C \prec\!\langle (\mathsf{Shared}, l) \rangle\!\overset{\mathcal{I}}{\rightarrowtail} A}$$

***Definition* 4.1.7.** *The path $(\mathsf{Repeated}, C :: \epsilon)$ from C to C is said to be* trivial . *Otherwise, if there is a non-trivial inheritance path from a class C to a class A, then, A is said to be a* base class, *or* base *for short, of C. Any base of a direct base of C is an* indirect *base of C.*

Example 4.1.3. Consider the following code:

```
struct A            {};
struct B1: virtual A {};
struct B2: virtual A {};
struct C : A        {};
struct D : B1, B2, C {};
```

$$\begin{tikzcd}
 & A & & A \\
B_1 \arrow[ur, dashed] & & B_2 \arrow[ul, dashed] & C \arrow[u] \\
 & & D \arrow[ull] \arrow[u] \arrow[ur] &
\end{tikzcd}$$

Then, the path from `D` to `A` through `B1` corresponds to the same subobject as the path through `B2`, because `A` is a virtual base of both `B1` and `B2`: the `A` subobject is *shared* by both classes. As `B1` and `B2` are bases of `D`, it is the case that `A` is a virtual base of `D` and this subobject is denoted by the path $(\mathsf{Shared}, A :: \epsilon)$. However, `A` is also reachable from `D` through another non-virtual path: $D :: C :: A :: \epsilon$, which denotes a different subobject.

Definition 4.1.6 (p. 75) enforces the fact that reaching a non-virtual inheritance subobject of a virtual base does not depend on how the virtual base is reached. Consequently, we have an equivalent characterization of inheritance paths:

LEMMA 4.1.3. *(h, l) is an inheritance path from C to A if, and only if:*
- *either $h = \mathsf{Repeated}$ and l is a non-virtual path from C to A*
- *or $h = \mathsf{Shared}$ and there is a virtual base B of C such that (h', l) is an inheritance path from B to A for some h'*

$$\frac{C \prec\!\langle l \rangle\!\overset{\mathcal{NV}}{\rightarrowtail} A}{C \prec\!\langle (\mathsf{Repeated}, l) \rangle\!\overset{\mathcal{I}}{\rightarrowtail} A} \qquad\qquad \frac{C \overset{\mathcal{V}}{\to} B \prec\!\langle (h', l) \rangle\!\overset{\mathcal{I}}{\rightarrowtail} A}{C \prec\!\langle (\mathsf{Shared}, l) \rangle\!\overset{\mathcal{I}}{\rightarrowtail} A}$$

Concatenation More generally, we can *concatenate* two inheritance paths:

Lemma 4.1.4. *Let C_0, C_1, C_2 be three classes. Let (h_1, l_1) be an inheritance path from C_0 to C_1 and (h_2, l_2) be a path from C_1 to C_2.*
– *If $h_2 = \mathsf{Repeated}$, then $(h_1, l_1 @_{\mathsf{Repeated}} l_2)$ is a path from C_0 to C_2*
– *If $h_2 = \mathsf{Shared}$, then (h_2, l_2) is a path from C_0 to C_2*
This path is written $\sigma_1 @ \sigma_2$.

$$(h_1, l_1)@(\mathsf{Repeated}, l_2) \underset{\text{def.}}{=\!=} (h_1, l_1 @_{\mathsf{Repeated}} l_2) \qquad (h_1, l_1)@(\mathsf{Shared}, l_2) \underset{\text{def.}}{=\!=} (\mathsf{Shared}, l_2)$$

$$\frac{C_0 \prec\!\langle (h_1, l_1) \rangle\!\overset{\mathcal{I}}{\succ} C_1 \prec\!\langle (h_2, l_2) \rangle\!\overset{\mathcal{I}}{\succ} C_2}{C_0 \prec\!\langle (h_1, l_1)@(h_2, l_2) \rangle\!\overset{\mathcal{I}}{\succ} C_2}$$

We extensively use the path concatenation operation for defining operations that allow navigating through the inheritance subobjects of an object. Such operations are called *casts*.

4.1.3 Structure array fields: array paths and generalized subobjects

Besides inheritance, C++ also provides a mechanism of *object embedding* (or *aggregation*) through structure or structure array data members. However, accessing such objects is not a matter of inheritance. To take them into account, we extend the work of Wasserrab et al. [85] with a notion of *generalized subobjects* including not only inheritance, but also structure field accesses.

Example 4.1.4. Consider for instance the following code:

```
struct X            {};
struct A : X        {};
struct B            { A fa[4]; };
struct C : virtual B {};
struct D            { C fc[5]; };
struct E : D        {};
E e[7];

X* px = (X*) &(e[2].fc[0].fa[3]);
```

The expression defining `px` can be decomposed into the more elementary parts:

```
E* pe2 = &(e[2]);    /* access to array cell */
D* pd  = (D*) pe2;   /* access to inheritance subobject */
C* pfc = pe2->pfc;   /* access to structure array field */

C* pc0 = &(pfc[0]);  /* access to array cell */
B* pb  = (B*) pc0;   /* access to inheritance subobject */
A* pfa = pb->pfa;    /* access to structure array field */

A* pa3 = &(pfa[3]);  /* access to array cell */
X* px  = (X*) pa3;   /* access to inheritance subobject */
```

That is, `px` is accessed from `e` in two stages:

- select the final array `pfa` through a sequence of "array cell selection, inheritance subobject selection, structure array field selection", called an *array path*;
- then, select the final array cell within `pfa` and the final inheritance subobject within this cell.

We formalize the notion of array paths as follows:

***Definition* 4.1.8.** *Let $C, C' \in \mathcal{C}$, $n, n' \in \mathbb{N}$, and α be a list of elements of the form (i, σ, f) where $i \in \mathbb{N}$, σ is an inheritance path and f is a field signature. We say that α is an* **a**rray path from $C[n]$ to $C'[n']$, *and we write $C[n] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\rightarrowtail} C'[n']$, if, and only if, one of these conditions holds:*

- *$C = C'$ and $n' \leq n$ and $\alpha = \epsilon$*
- *or all the following conditions hold:*
  - *there exists a cell index i within $C[n]$ (such that $i < n$)*
  - *there is a base A of C through an inheritance path σ*
  - *A has a structure array field $f =$ (`struct` $C_f[n_f]$ fname)*
  - *there is an array path α' from $C_f[n_f]$ to $C'[n']$*
  - *and $\alpha = (i, \sigma, f) :: \alpha'$*

$$\frac{n' \leq n}{C[n] \prec\!\langle\epsilon\rangle\!\overset{\mathcal{A}}{\rightarrowtail} C[n']}$$

$$\frac{0 \leq i < n \qquad C \prec\!\langle\sigma\rangle\!\overset{\mathcal{I}}{\rightarrowtail} A \qquad f = (\texttt{struct}\ C_f[n_f]\ \mathit{fname}) \in \mathcal{F}_A \qquad C_f[n_f] \prec\!\langle\alpha'\rangle\!\overset{\mathcal{A}}{\rightarrowtail} C'[n']}{C[n] \prec\!\langle(i, \sigma, f) :: \alpha'\rangle\!\overset{\mathcal{A}}{\rightarrowtail} C'[n']}$$

In Example 4.1.4 (p. 76), `pfa` corresponds to the following array path α from `E e[7]` to the structure field `A fa[4]` of B:

$$\begin{aligned} \alpha = {} & (2, (\mathsf{Repeated}, E :: D :: \epsilon), fc) \\ & :: (0, (\mathsf{Shared}, B :: \epsilon), fa) \\ & :: \epsilon \end{aligned}$$

Then, we combine an array path with an inheritance path to unambiguously designate any subobject:

***Definition* 4.1.9.** *A* generalized subobject *or* relative pointer *p of static type A from (or within) an array of type $C[n]$ (or a* relative pointer from $C[n]$ to A*) is a triple $p = (\alpha, i, \sigma)$ where:*

- *α is an array path from $C[n]$ to some $C'[n']$;*
- *i is an array cell index within $C'[n']$, such that $0 \leq i < n'$;*
- *σ is an inheritance path from C' to A.*

We write $C[n] \prec\!\langle(\alpha, i, \sigma)\rangle\!\rightarrowtail A$.

We also write for short $C[n] \prec\!\langle(i, \sigma)\rangle\!\overset{\mathcal{CI}}{\rightarrowtail} A$ to mean that σ is an A **i***nheritance path within the i-th* **c***ell of array $C[n]$.*

$$\frac{0 \leq i < n \qquad C \prec\!\langle\sigma\rangle\!\overset{\mathcal{I}}{\rightarrowtail} A}{C \prec\!\langle(i, \sigma)\rangle\!\overset{\mathcal{CI}}{\rightarrowtail} A} \qquad \frac{C[n] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\rightarrowtail} C'[n'] \prec\!\langle(i, \sigma)\rangle\!\overset{\mathcal{CI}}{\rightarrowtail} A}{C[n] \prec\!\langle(\alpha, i, \sigma)\rangle\!\rightarrowtail A}$$

In Example 4.1.4 (p. 76), `px` corresponds to the following generalized subobject from `E e[7]`:

$$(\alpha, 3, (\mathsf{Repeated}, A :: X :: \epsilon))$$

Now we formally introduce the notion of *inheritance subobject* and *most-derived object*:

***Definition* 4.1.10 (Inheritance subobject).** *Let $p_1 = (\alpha_1, i_1, \sigma_1)$ and $p_2 = (\alpha_2, i_2, \sigma_2)$ be two generalized subobjects from some $C[n]$. Then p_2 is said to be an* inheritance subobject *of p_1 if, and only if, all the following conditions hold:*

(i) they are subobjects of the same array cell: $(\alpha_1, i_1) = (\alpha_2, i_2)$, such that $C[n] \prec\!\langle\alpha_1\rangle\!\overset{\mathcal{A}}{\succ\!\to} C'[n']$ for some $C'[n']$ such that $0 \leq i_1 < n'$,

(ii) and $\sigma_2 = \sigma_1 @ \sigma'$ for some σ' such that $C' \prec\!\langle\sigma_1\rangle\!\overset{\mathcal{I}}{\succ\!\to} C_1 \prec\!\langle\sigma'\rangle\!\overset{\mathcal{I}}{\succ\!\to} C_2$ for some classes C_1 and C_2.

Condition (ii) may be also designated as: σ_2 is an inheritance subobject of σ_1

***Definition* 4.1.11 (Most-derived object).** *A* most-derived *object is an object that is not an inheritance subobject of another object.*

For instance, if $C'[n'] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\succ\!\to} C[n]$ is an array path, and $0 \leq i < n$, then $(\alpha, i, (\mathsf{Repeated}, C :: \epsilon))$, obtained by the trivial inheritance path from C, is a most-derived object. Corollary 4.1.10 (p. 79) shows that most-derived objects are necessarily of this form, under some well-formedness conditions on class hierarchies introduced in the next section.

4.1.4 Well-formed hierarchies

We expect some well-formedness and well-foundedness hypotheses [2] to hold on the class hierarchy, so as to be able to *compute*, for any class C, some data $F(C)$ depending on $F(B)$ for all classes B being bases of C or types of the structure array fields of C.

4.1.4.1 Well-defined classes

We assume that the hierarchy is *complete* in the sense of the C++ standard: all classes referred to by class definitions are correctly defined.

Hypothesis 4.1.1. *Assume the hierarchy is* well-defined*, i.e. for any defined class $C \in \mathcal{C}$, all its bases and structure fields refer to defined classes:*

$$\forall (\texttt{virtual}^?) B \in \mathcal{D}(C) : \ B \in \mathcal{C}$$

and compatible with structure array fields, i.e. such that:

$$\forall (\texttt{struct } B[n] \ f) \in \mathcal{F}(C) : \ B \in \mathcal{C}$$

2. Coq development: theory `CplusWf`.

4.1.4.2 Well-founded hierarchies

Hypothesis 4.1.2. *Assume the hierarchy is* well-founded, *i.e. the set of classes can be ordered by some well-founded relation $\prec$ compatible with inheritance, i.e. such that:*

$$\forall(\texttt{virtual}^?)B \in \mathcal{D}(C) : \ B \prec C$$

and compatible with structure array fields, i.e. such that:

$$\forall(\texttt{struct}\ B[n]\ f) \in \mathcal{F}(C) : \ B \prec C$$

In our Coq formalization, $\mathcal{C} \subseteq \mathbb{N}^{>0}$, and $\prec$ is the usual ordering relation on $\mathbb{N}^{>0}$.

***Notation* 4.1.12.** *For each class C, the set of its virtual bases is written $\mathcal{V}(C)$. Thus, as the hierarchy is well-founded, we can compute $\mathcal{V}(C)$ by well-founded induction over $\prec$:*

$$\mathcal{V}(C) \ \underset{\text{def.}}{=\!=} \ \mathcal{DV}(C) \cup \bigcup_{B \in \mathcal{D}(C)} \mathcal{V}(B)$$

Lemma 4.1.5.

$$\forall C, \forall B \in \mathcal{V}(C) : \ B \prec C$$

In particular, $C \notin \mathcal{V}(C)$.

Lemma 4.1.6. *If there exists an inheritance path from C to A, then $A \preceq C$.*

If there exists an array path from $C[n]$ to $C'[n']$, then $C' \preceq C$.

Lemma 4.1.7. *The only path from a class C to itself is the* trivial *path:*

$$(\mathsf{Repeated}, C :: \epsilon)$$

Corollary 4.1.8. *For any classes C and A, it is possible to compute the set of all inheritance paths from C to A. This set is finite.*

For any class C and any $n \in \mathbb{N}$, it is possible to compute the set of all generalized subobjects within $C[n]$. This set is finite.

Lemma 4.1.9. *For any classes C and A, if $(h, B :: l)$ is an inheritance path from C to A, then $B = C$ if and only if $h = \mathsf{Repeated}$.*

Corollary 4.1.10. *A most-derived object within an array of $C[n]$ is necessarily of the form:*

$$(\alpha, i, (\mathsf{Repeated}, C' :: \epsilon))$$

where $C[n] \prec\!\langle\alpha\rangle\!\xrightarrow{A} C'[n']$ and $0 \leq i < n'$. That is, an object is most-derived if, and only if, it is a structure array cell.

The well-formedness of the hierarchy also allows to argue on the left-hand-side regularity of path concatenation:

LEMMA 4.1.11. *Consider the inheritance paths* $D \prec\!\langle(h,l)\rangle\!\overset{\mathcal{I}}{\rightarrow} C \prec\!\langle(h_1,l_1)\rangle\!\overset{\mathcal{I}}{\rightarrow} B_1$ *and* $C \prec\!\langle(h_2,l_2)\rangle\!\overset{\mathcal{I}}{\rightarrow} B_2$.

Then, $(h,l)@(h_1,l_1) = (h,l)@(h_2,l_2)$ *implies* $(h_1,l_1) = (h_2,l_2)$.

Proof. LEMMA 4.1.1 (p. 74) immediately gives $B_1 = B_2$. First we show that $h_1 = h_2$. If, for instance, $h_1 = \mathsf{Shared}$ and $h_2 = \mathsf{Repeated}$, then $(h,l)@(h_1,l_1) = (\mathsf{Shared}, l_1)$. Let $l_1 = V :: l'$, so that V is a virtual base of C, thus $V \prec C$ per LEMMA 4.1.5 (p. 79). But on the other hand, $(h,l)@(h_2,l_2) = (h, l@_{\mathsf{Repeated}}l_2)$, so that $l@_{\mathsf{Repeated}}l_2 = V :: l'$. Thus, l starts with V. This means that l is a non-virtual path from V to C, thus $C \preceq V$ per LEMMA 4.1.6 (p. 79), which leads to a contradiction.

Now since $h_1 = h_2$, there are two cases:

- if $h_1 = \mathsf{Repeated}$, then for each $i \in \{1,2\}$: $l_i = C :: l'_i$ is a non-virtual path from C to B'_i, and we have $(h,l)@(h_i,l_i) = (h, l@_{\mathsf{Repeated}}l_i)$, which leads to $l@_{\mathsf{Repeated}}l_1 = l@_{\mathsf{Repeated}}l_2$, i.e. $l +\!\!+ l'_1 = l +\!\!+ l'_2$. It is easy to see that $+\!\!+$ is left-hand-side regular, so $l'_1 = l'_2$, which concludes.
- Otherwise, $h_1 = h_2 = \mathsf{Shared}$, then for each i, $(h,l)@(h_i,l_i) = (\mathsf{Shared}, l_i) = (h_i, l_i)$ so that $l_1 = l_2$, which concludes. □

4.2 Syntax of the s++ language

Wasserrab et al. [85] define a subset of C++ called CoreC++ to formalize C++ multiple inheritance features, in particular static and dynamic casts, and virtual function calls. We extend their language by adding structure field array accesses, to define a language that we call s++ (for a subset of C++ featuring embedded structures).

s++ is an imperative 3-address language, without embedded expressions: arguments of operations are necessarily variables. s++ features built-in operations (Section 3.2.4 p. 67) and usual structured control under the form of conditionals, sequences, infinite loops, and statement blocks with early exit (avoiding the need for *goto*-like statements, and allowing exit from infinite loops).

s++ also has object-oriented features: reading or writing a scalar field, accessing a structure field, accessing a structure array cell, pointer equality test, static and dynamic casts, and virtual function calls. But s++ also allows static (non-class-member) function calls, as well as non-virtual function calls, which are calls to a class member function bypassing dynamic dispatch, as if it were a static function.

n	$\in \mathbb{N}$	
$op, \ldots$	$\in Op$	Built-in operations
$x, \ldots$	$\in Var$	Variables
$B, C, \ldots$	$\in ClassName$	Classes
$fsig$	$\in FieldSig$	Field signatures
$msig$	$\in MethodSig$	Method signatures
$sfname$	$\in StaticFunName$	Static function names

st	$::= x' := op(x^*)$	Built-in operation
	$\mid x' := x$	Assignment between variables
	$\mid \texttt{if}\ (x)\ st_{\mathsf{true}}\ \texttt{else}\ st_{\mathsf{false}}$	Conditional
	$\mid st_1; st_2$	Statement sequence

$\mid$ **`skip`**	Do nothing
$\mid$ **`loop`** st	Infinite loop
$\mid$ **`return`** $x^?$	Return from function
$\mid \{st\}$	Statement block
$\mid$ **`exit`** n	Exit from n blocks
$\mid x'^? := sfname(x^*)$	Static function call
$\mid x'^? := x\texttt{->}C\texttt{::}msig(x^*)$	Non-virtual function call
$\mid x' := x\texttt{->}_C fsig$	Field read
$\mid x\texttt{->}_C fsig := x'$	Scalar field write
$\mid x' := \&x[x_{\mathsf{index}}]_C$	Array cell access
$\mid x' := x_1 ==_C x_2$	Pointer equality test
$\mid x' := \texttt{static\_cast}\langle B\rangle_C(x)$	Static cast
$\mid x' := \texttt{dynamic\_cast}\langle B\rangle_C(x)$	Dynamic cast
$\mid x'^? := x\texttt{->}_C msig(x^*)$	Virtual function call

In s++, object-oriented features perform no implicit casts: for instance, field accesses explicitly require that the class defining the field exactly match the static type of the object pointer. In other words, we rely on the C++ typechecker to materialize implicit casts while elaborating the source program into the s++ intermediate language. For example, under the following C++ class hierarchy:

```
struct A     { int i; } ;
struct B: A  {} ;
B* b;
```

the C++ statement `j = b->i;` is elaborated to the following s++ code, explicitly casting to `A`:

$$a = \texttt{static\_cast}\langle A\rangle_B(b);$$
$$j = a\texttt{->}_A i;$$

This example also shows that object-oriented operations are tagged by the types expected for their arguments: no static type inference is performed on s++ code.

Finally, a s++ program is composed of a class hierarchy, definitions for static (non-class-member) functions, and the codes of class member functions, with the names of the arguments for use within function bodies:

StaticFunDef	::=	$(x^*)\{st\}$	Static function
MethodDef	::=	$this\texttt{->}(x^*)\{st\}$	Class member function (method) definition

Program =
{

hierarchy	:	*Hierarchy*	; Class hierarchy
staticfuns	:	$StaticFunName \nrightarrow StaticFunDef$	; Static functions
methods	:	$ClassName \times MethodSig \nrightarrow MethodDef$	; Class method codes

}

4.3 Semantic elements

We formalized a small-step style semantics for the s++ language. Similarly to some CompCert-like intermediate languages, the execution state contains a simple "block or callframe" continuation stack to precisely model each step of computation.

4.3.1 Values

A value is either a *value of built-in type* (integer, floating-point number, etc.), a *pointer* to a generalized subobject, or a *null* pointer.

In s++, objects cannot be created ; instead, they are assumed to already exist when the program starts. Such objects are called *complete* objects. They are represented by locations written ℓ. A pointer to a generalized subobject is a tuple (ℓ, p) where p represents a generalized subobject within a complete object stored at location ℓ. Null pointers are required in order to deal with failing dynamic casts (which purposefully must not interrupt the execution of the program).

$\ell, \ldots$	$\in \Lambda$	Complete object location
Ptr	$::= (\ell, (\alpha, i, \sigma))$	Pointer to subobject
Val	$::= \mathit{Builtin}$	Value of built-in type
	$\mid \mathit{Ptr}$	Non-null pointer
	$\mid \mathsf{NULL}_C$	Null pointer of C class type

4.3.2 Execution state

An *execution state* of the small-step semantics is composed of:

- the current statement to execute
- the list of further statements to execute in the same block
- the environment (mapping of values to variables)
- the continuation stack, which is a list of frames, each frame being either:
  - leaving a block, with the further statements to execute after leaving the block
  - returning from a function, with the caller variable to store the result (if any), the caller environment, and the further statements to execute on resumption
- the class type and array size of each location of a complete object
- the value of each scalar field. A scalar field is unambiguously designated by a complete object location, a generalized subobject of some class type C within this complete object, and a scalar field signature declared in class C.

For presentation convenience, the types of complete objects and the scalar field values are grouped into a common *global state*, so that a state is written as a tuple $(st, st^*, e, \mathcal{K}, \mathcal{G})$ where st is the current statement, st^* is the list of further statements, $\mathcal{K}$ the continuation stack, and $\mathcal{G}$ the global state grouping the types of complete objects and the values of scalar fields.

$$\begin{array}{llll}
\mathit{Env} & = & x \to \mathit{Val}^{?} & \text{Environment} \\
e & ::= & \mathit{Env} & \\
\mathit{Frame} & ::= & \mathsf{Block}(st^{*}) & \text{Further statements} \\
 & & & \text{after leaving a block} \\
 & \mid & \mathsf{Callframe}(x^{?}, st^{*}, e) & \text{Return from function} \\
\mathcal{K} & ::= & \mathit{Frame}^{*} & \text{Continuation stack} \\
\mathcal{G} & = & & \\
\{ & & & \\
\quad \mathsf{LocType} & : & \Lambda \to (\mathit{ClassName} \times \mathbb{N}^{>0})^{?}\ ; & \text{Complete object types} \\
\quad \mathsf{FieldValue} & : & \mathit{Ptr} \times \mathit{FieldSig} \to \mathit{Val}^{?}\ ; & \text{Scalar field values} \\
\} & & & \\
\mathit{State} & ::= & (st, st^{*}, e, \mathcal{K}, \mathcal{G}) & \text{Execution state}
\end{array}$$

***Notation* 4.3.1.** *We write $\mathcal{G} \vdash (\ell, p) : B$ to mean that p is a generalized subobject of the complete object ℓ, and the static type of p is B. More formally:*

$$\frac{\mathcal{G}.\mathsf{LocType}(\ell) = (C, n) \qquad C[n] \prec\!\langle p\rangle\!\!\rightarrowtail B}{\mathcal{G} \vdash (\ell, p) : B}$$

We also write $\mathcal{G} \vdash \langle \ell \rangle\ C[n]$ to mean $\mathcal{G}.\mathsf{LocType}(\ell) = (C, n)$.

The latter notation allows for chaining notations: $\mathcal{G} \vdash \langle \ell \rangle\ C[n] \prec\!\langle \alpha \rangle\!\!\overset{\mathcal{A}}{\rightarrowtail} C'[n'] \prec\!\langle (i, \sigma) \rangle\!\!\overset{\mathcal{CI}}{\rightarrowtail} A$.

4.4 Semantic rules

The small-step semantics of s++ is given by the transition relation $\to$ between two transition states, defined in this section.

4.4.1 Features unrelated to C++ multiple inheritance

4.4.1.1 Structured control, variable value duplication and built-in operations

In an execution state $(st, stl, e, \mathcal{K}, \mathcal{G})$, st is the statement to run, and stl is a pipeline of pending statements *within the same block*, each pending enclosing block being represented by a frame in the continuation stack $\mathcal{K}$. However, the pipeline L is not guaranteed to be executed, in particular if the statement is `exit` or `return`.

Most structured control behaves similarly as in other CompCert-like languages: conditionals, sequences, infinite loops, and return from call (once all statements blocks within the current function have been left), as well as variable value duplication, and built-in operations (Hypothesis 3.2.2 p. 68). The corresponding rules are described below.

Structured control The conditional statement selects the next statement depending on the value of its boolean argument:

$$\frac{e(x) = b \ \in \{\mathsf{true}, \mathsf{false}\}}{\begin{array}{ll} & (\mathtt{if}(x)\ st_{\mathsf{true}}\ \mathtt{else}\ st_{\mathsf{false}},\ stl,\ e,\ \mathcal{K},\ \mathcal{G}) \\ \to & (st_b,\ stl,\ e,\ \mathcal{K},\ \mathcal{G}) \end{array}} \quad \text{(s++-if)}$$

The statement sequence feeds the pipeline:

$$\frac{}{\begin{array}{ll} & (st_1; st_2,\ stl,\ e,\ \mathcal{K},\ \mathcal{G}) \\ \to & (st_1,\ st_2 :: stl,\ e,\ \mathcal{K},\ \mathcal{G}) \end{array}} \quad \text{(s++-seq)}$$

The pipeline can be forced by `skip`:

$$\frac{}{\begin{array}{ll} & (\mathtt{skip},\ st :: stl,\ e,\ \mathcal{K},\ \mathcal{G}) \\ \to & (st,\ stl,\ e,\ \mathcal{K},\ \mathcal{G}) \end{array}} \quad \text{(s++-skip)}$$

An infinite loop feeds itself into the pipeline, then executes its body:

$$\frac{}{\begin{array}{ll} & (\mathtt{loop}\ st,\ stl,\ e,\ \mathcal{K},\ \mathcal{G}) \\ \to & (st,\ \mathtt{loop}\ st :: stl,\ e,\ \mathcal{K},\ \mathcal{G}) \end{array}} \quad \text{(s++-loop)}$$

When returning from a function, the caller is given by a Callframe on top of the continuation stack. If the caller expects a return value by giving a variable name, then this variable is updated with the actual return value.

$$\frac{e(x) = v \qquad e'' = e'[res \leftarrow v]}{\begin{array}{ll} & (\mathtt{return}\ x,\ stl,\ e,\ \mathsf{Callframe}(res, stl', e') :: \mathcal{K},\ \mathcal{G}) \\ \to & (\mathtt{skip},\ stl',\ e'',\ \mathcal{K},\ \mathcal{G}) \end{array}} \quad \text{(s++-return-arg)}$$

$$\frac{}{\begin{array}{ll} & (\mathtt{return},\ stl,\ e,\ \mathsf{Callframe}(\bot, stl', e') :: \mathcal{K},\ \mathcal{G}) \\ \to & (\mathtt{skip},\ stl',\ e',\ \mathcal{K},\ \mathcal{G}) \end{array}} \quad \text{(s++-return-no-arg)}$$

Built-in operation A built-in operation can produce an observable event:

$$\frac{\forall i, e(x_i) = v_i \qquad [op](v_1 :: \ldots :: v_n :: \epsilon) \ni (\mathfrak{e}^?, res) \qquad e' = e[x' \leftarrow res]}{\begin{array}{ll} & (x' := op(x_1, \ldots, x_n),\ stl,\ e,\ \mathcal{K},\ \mathcal{G}) \\ \underset{\mathfrak{e}^?}{\to} & (\mathtt{skip},\ stl,\ e',\ \mathcal{K},\ \mathcal{G}) \end{array}} \quad \text{(s++-builtin)}$$

Assignment between variables The value of the source variable is simply stored into the target variable:

$$\frac{e(x) = v \qquad e' = e[x' \leftarrow v]}{\begin{array}{ll} & (x' := x,\ stl,\ e,\ \mathcal{K},\ \mathcal{G}) \\ \to & (\mathtt{skip},\ stl,\ e',\ \mathcal{K},\ \mathcal{G}) \end{array}} \quad \text{(s++-var-dup)}$$

4.4.1.2 Statement blocks

Statements may be enclosed into *statement blocks*, which can terminate prematurely by executing an `exit` statement.

$$\frac{}{\begin{array}{rllll} & (\{st\}, & stl, & e, & \mathcal{K}, \ \mathcal{G}) \\ \to & (st \quad , & \epsilon, & e, & \mathsf{Block}(stl) :: \mathcal{K}, \ \mathcal{G}) \end{array}} \quad \text{(s++-block)}$$

`exit` n leaves n enclosing blocks.

$$\frac{}{\begin{array}{rlllll} & (\texttt{exit}\ 0, & stl, & e, & \mathcal{K}, & \mathcal{G}) \\ \to & (\texttt{skip} \quad , & stl, & e, & \mathcal{K}, & \mathcal{G}) \end{array}} \quad \text{(s++-exit-0)}$$

$$\frac{}{\begin{array}{rlllll} & (\texttt{exit}\ (\mathsf{S}\ n), & stl, & e, & \mathsf{Block}(stl') :: \mathcal{K}, & \mathcal{G}) \\ \to & (\texttt{exit}\ n \quad , & stl', & e, & \mathcal{K}, & \mathcal{G}) \end{array}} \quad \text{(s++-exit-S)}$$

Returning from a function first leaves all enclosed blocks.

$$\frac{}{\begin{array}{rlllll} & (\texttt{return}\ x^?, & stl, & e, & \mathsf{Block}(stl') :: \mathcal{K}, & \mathcal{G}) \\ \to & (\texttt{return}\ x^?, & stl', & e, & \mathcal{K}, & \mathcal{G}) \end{array}} \quad \text{(s++-return-block)}$$

4.4.2 Static and non-virtual function call

Static functions s++ allows to call static (non-class-member) functions [3]. Upon such a call, a new variable environment is created for the callee, to store the values of arguments. The caller, with its own environment, the further statements to execute, and maybe a variable to store the return value, are saved into a new Callframe placed on top of the continuation stack. Finally, the body statement of the function is executed in the callee:

$$\frac{\begin{array}{c} \mathsf{staticfuns}(sfname) = (varg_1, \ldots, varg_n)\{body\} \\ \forall j, e(x_j) = v_j \qquad e' = \emptyset[varg_1 \leftarrow v_1] \ldots [varg_n \leftarrow v_n] \end{array}}{\begin{array}{rlllll} & (x^? := sfname(x_1 \ldots x_n), & stl, & e\ , & \mathcal{K}, & \mathcal{G}) \\ \to & (body \quad , & \epsilon, & e', & \mathsf{Callframe}(x^?, stl, e) :: \mathcal{K}, & \mathcal{G}) \end{array}} \quad \text{(s++-static-funcall)}$$

Non-virtual function call s++ also allows calls to class member functions in a non-virtual fashion, i.e. bypassing dynamic dispatch, as in C++ with explicit qualification. Such a call actually corresponds to calling a class member function of n arguments as if it were a non-class-member function of $1 + n$ arguments, the additional argument being the object on which to call the function without dynamic dispatch.

This operation is tagged with a class name C. Contrary to C++, the requested function must be actually declared in C, and C must be exactly the static type of the object on which to perform the call: there is no implicit cast.

3. No functions were supported in our POPL 2011 paper [72], which relied on a simplified proof [70].

$$\frac{\begin{array}{c}\mathsf{methods}(C, \mathit{msig}) = \mathit{this}\texttt{->}(\mathit{varg}_1, \ldots, \mathit{varg}_n)\{\mathit{body}\} \\ \forall j, e(x_j) = v_j \qquad e(x) = v \qquad e' = \emptyset[\mathit{this} \leftarrow v][\mathit{varg}_1 \leftarrow v_1] \ldots [\mathit{varg}_n \leftarrow v_n]\end{array}}{\begin{array}{rl} & (x'^{?} := x\texttt{->}C\texttt{::}\mathit{msig}(x_1 \ldots x_n),\ \ \mathit{stl},\ \ e,\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathit{body},\ \ \epsilon,\ \ e',\ \ \mathsf{Callframe}(x'^{?}, \mathit{stl}, e) :: \mathcal{K},\ \ \mathcal{G})\end{array}}$$

(s++-non-virtual-funcall)

4.4.3 Field and array accesses, and pointer equality test

Scalar fields Reading or writing a scalar field is tagged by a class name C. The field being accessed must be explicitly declared in C, and the pointer to the object must correspond to a subobject of static type C (there is no implicit cast). The value of the scalar field is retrieved from, or written to, the $\mathcal{G}.\mathsf{FieldValue}$ entry of the global state:

$$\frac{\begin{array}{c} f = (\texttt{scalar}\ t\ \mathit{fname}) \in \mathcal{F}(C) \\ e(x) = \pi \qquad \mathcal{G} \vdash \pi : C \qquad \mathcal{G}.\mathsf{FieldValue}(\pi, f) = \mathit{res} \qquad e' = e[x' \leftarrow \mathit{res}]\end{array}}{\begin{array}{rl} & (x' := x\texttt{->}_C f,\ \ \mathit{stl},\ \ e,\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\texttt{skip},\ \ \mathit{stl},\ \ e',\ \ \mathcal{K},\ \ \mathcal{G})\end{array}}$$

(s++-field-scalar-read)

$$\frac{\begin{array}{c} e(x) = \pi \qquad \mathcal{G} \vdash \pi : C \\ f = (\texttt{scalar}\ t\ \mathit{fname}) \in \mathcal{F}(C) \qquad e(x') = \mathit{res} \qquad \mathcal{G}' = \mathcal{G}[\mathsf{FieldValue}(\pi, f) \leftarrow \mathit{res}]\end{array}}{\begin{array}{rl} & (x\texttt{->}_C f := x',\ \ \mathit{stl},\ \ e,\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\texttt{skip},\ \ \mathit{stl},\ \ e,\ \ \mathcal{K},\ \ \mathcal{G}')\end{array}}$$

(s++-field-scalar-write)

Structure fields Only scalar fields may be assigned. In true C++, assignment to a structure field is actually a call to the `operator=` class member function. We therefore assume that the elaborator produced an explicit call to this class member function, and add no specific evaluation rule for that case.

Accessing a structure array field actually produces a pointer to its first cell, without dereferencing it. It is only "pointer adjustment" without actually reading any value.

$$\frac{\begin{array}{c} e(x) = \pi = (\ell, (\alpha, i, \sigma)) \qquad \mathcal{G} \vdash \pi : C \qquad f = (\texttt{struct}\ B[n]\ \mathit{fname})) \in \mathcal{F}(C) \\ e' = e[x' \leftarrow (\ell, (\alpha +\!\!+ (i, \sigma, f) :: \epsilon, 0, (\mathsf{Repeated}, B :: \epsilon)))]\end{array}}{\begin{array}{rl} & (x' := x\texttt{->}_C f,\ \ \mathit{stl},\ \ e,\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\texttt{skip},\ \ \mathit{stl},\ \ e',\ \ \mathcal{K},\ \ \mathcal{G})\end{array}}$$

(Ds++-field-struct-point)

Array subscripting Accessing an array cell is only valid on a pointer to a most-derived object. Indeed, per COROLLARY 4.1.10 (p. 79), only most-derived objects are array cells, and the subscripting operation actually allows accessing a *sibling* array cell, as in the following C++ example:

```
A a[3];
A* a2 = &(a[2]);
A* a1 = &(a2[-1]); /* equivalent to &(a[1]) */
```

Then, likewise, it is a mere "pointer adjustment" without actually reading any value:

$$\frac{\begin{array}{c} e(x) = (\ell, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon))) \\ e(x_{\mathsf{index}}) = j \in \mathbb{Z} \qquad e' = e[x' \leftarrow (\ell, (\alpha, i + j, (\mathsf{Repeated}, C :: \epsilon)))] \end{array}}{\begin{array}{rlllll} & (x' := \&x[x_{\mathsf{index}}]_C, & stl, & e\,, & \mathcal{K}, & \mathcal{G}) \\ \to & (\mathtt{skip} & , stl, & e', & \mathcal{K}, & \mathcal{G}) \end{array}} \quad \text{(s++-array-point)}$$

Pointer equality test Comparing two pointers requires them to be of the same type C. Either may be null. Thus we need to introduce a notion of pointer typing generalized to null pointers:

***Notation* 4.4.1.** *We define the notation $\mathcal{G} \vdash \tilde{\pi} \div C$ to say that a pointer $\tilde{\pi}$ that may be* NULL *has type C:*

$$\frac{\mathcal{G} \vdash \pi : C}{\mathcal{G} \vdash \pi \div C} \qquad\qquad \frac{}{\mathcal{G} \vdash \mathsf{NULL}_C \div C}$$

Then, comparing two pointers of the same type C yields a boolean, true if and only if the two pointers are equal.

$$\frac{\begin{array}{c} \forall i \in \{1, 2\} : e(x_i) = \tilde{\pi}_i \\ \forall i \in \{1, 2\} : \mathcal{G} \vdash \tilde{\pi}_i \div C \qquad b \in \{\mathsf{true}, \mathsf{false}\} \qquad b = \mathsf{true} \Leftrightarrow \tilde{\pi}_1 = \tilde{\pi}_2 \qquad e' = e[x' \leftarrow b] \end{array}}{\begin{array}{rlllll} & (x' := x_1 ==_C x_2, & stl, & e\,, & \mathcal{K}, & \mathcal{G}) \\ \to & (\mathtt{skip} & , stl, & e', & \mathcal{K}, & \mathcal{G}) \end{array}}$$

(s++-ptreq)

4.4.4 Static cast

4.4.4.1 The two flavours of static cast

C++ gives two flavours of static cast from an object of class type B to some class type B' (cf. Wasserrab et al. [85]). We write $\mathsf{StatCast}(\sigma, B, B', \sigma')$ to say that static cast from an inheritance subobject σ of type B to B' is well-defined with σ' as the result:

– either cast from B to one of its bases B' such that there is a unique inheritance path from B to B'

$$\frac{B \prec\!\langle \sigma'' \rangle\!\overset{\mathcal{I}}{\succ} B' \qquad \forall \sigma' : B \prec\!\langle \sigma' \rangle\!\overset{\mathcal{I}}{\succ} B' \Rightarrow \sigma' = \sigma''}{\mathsf{StatCast}(\sigma, B, B', \sigma @ \sigma'')} \quad \text{(statcast-derived-to-base)}$$

– or cast from B to one of its *non-virtual* derived classes B' such that there is a unique *non-virtual* inheritance path from B' to B

$$\frac{B' \prec\!\langle (B' :: l) \rangle\!\xrightarrow{\mathcal{NV}} B \qquad \forall l'', B' \prec\!\langle (B' :: l'') \rangle\!\xrightarrow{\mathcal{NV}} B \Rightarrow l'' = l \qquad \mathsf{last}(l') = B'}{\mathsf{StatCast}((h, l' + l), B, B', (h, l'))}$$
(statcast-base-to-derived-non-virtual)

Lemma 4.4.1. *The result of a static cast, if any, is unique.*

4.4.4.2 s++ language construct

Those rules are used for defining the s++ static cast rule from B to B', which applies on a valid pointer to a generalized subobject of static type B (again, there is no implicit cast).

$$\frac{\begin{array}{c} e(x) = \pi = (\ell, (\alpha, i, \sigma)) \\ \mathcal{G} \vdash \pi : B \qquad \mathsf{StatCast}(\sigma, B, B', \sigma') \qquad e' = e[x' \leftarrow (\ell, (\alpha, i, \sigma'))] \end{array}}{\begin{array}{ll} & (x' := \mathtt{static\_cast}\langle B' \rangle_B(x), \ stl, \ e, \ \mathcal{K}, \ \mathcal{G}) \\ \rightarrow & (\mathtt{skip} \qquad\qquad\qquad\quad , \ stl, \ e', \ \mathcal{K}, \ \mathcal{G}) \end{array}}$$
(s++-statcast)

4.4.5 Dynamic cast

4.4.5.1 The three flavours of dynamic cast

C++ gives three flavours of dynamic cast from an object of type B to B'. Let σ be an inheritance subobject of type B from a most-derived C object. We write $\mathsf{DynCast}(C, \sigma, B, B', \sigma')$ (inspired from Wasserrab et al. [85]) if dynamic cast of σ from B to B' succeeds and yields the subobject σ' of the most-derived C object, and $\mathsf{DynCast}(C, \sigma, B, B', \mathsf{NULL}_{B'})$ if it fails.

– Either B' is a base of B and there is a unique inheritance path from B to B' (derived-to-base cast):

$$\frac{C \prec\!\langle \sigma \rangle\!\xrightarrow{\mathcal{I}} B \prec\!\langle \sigma'' \rangle\!\xrightarrow{\mathcal{I}} B' \qquad \sigma'' \text{ unique}}{\mathsf{DynCast}(C, \sigma, B, B', \sigma @ \sigma'')}$$
(dyncast-derived-to-base)

– Or, σ may be decomposed as a B non-virtual subobject of some B' subobject (non-virtual base-to-derived cast):

$$\frac{C \prec\!\langle (h, l') \rangle\!\xrightarrow{\mathcal{I}} B' \prec\!\langle (\mathsf{Repeated}, B' :: l) \rangle\!\xrightarrow{\mathcal{I}} B}{\mathsf{DynCast}(C, (h, l' + l), B, B', (h, l'))}$$
(dyncast-base-to-derived-non-virtual)

– Or, there is a unique B' subobject from D (*cross-cast* from B to B' via the most-derived object D):

$$\frac{C \prec\!\langle \sigma \rangle\!\xrightarrow{\mathcal{I}} B \qquad C \prec\!\langle \sigma' \rangle\!\xrightarrow{\mathcal{I}} B' \qquad \sigma' \text{ unique}}{\mathsf{DynCast}(C, \sigma, B, B', \sigma')}$$
(dyncast-crosscast)

Moreover, if none of the above is applicable, then dynamic cast must yield a null pointer. That is, dynamic cast never crashes:

$$\frac{\begin{array}{c} C \prec\!(\sigma)\!\succ\!\xrightarrow{\mathcal{I}} B \qquad \nexists!\sigma'' : B \prec\!(\sigma'')\!\succ\!\xrightarrow{\mathcal{I}} B' \\ \nexists!\sigma' : C \prec\!(\sigma')\!\succ\!\xrightarrow{\mathcal{I}} B' \qquad \nexists h, l', l : \begin{cases} \sigma = (h, l' +\!\!+\, l) \\ C \prec\!((h, l'))\!\succ\!\xrightarrow{\mathcal{I}} B' \prec\!((\mathsf{Repeated}, B' :: l))\!\succ\!\xrightarrow{\mathcal{I}} B \end{cases} \end{array}}{\mathsf{DynCast}(C, \sigma, B, B', \mathsf{NULL}_{B'})} \ (\mathsf{dyncast\text{-}fail})$$

Lemma 4.4.2. *The result of dynamic cast is unique.*

Static and dynamic casts to bases are related:

Lemma 4.4.3. *If B' is a base of B, then dynamic cast from an inheritance subobject succeeds if, and only if, static cast is well-defined, and in such case, they give the same result.*

This fact allows the elaborator to optimize away dynamic casts to bases: considering that s++ is an intermediate language, we can assume that dynamic casts to bases have been translated to static casts, leaving only dynamic casts to non-bases.

4.4.5.2 s++ language construct

The C++ Standard [42] allows dynamic casts from *dynamic classes* only[4]:

***Definition* 4.4.2.** *A class C is said to be* dynamic *(written* $\mathsf{isDynamic}(c)$*) if, and only if, at least one of the following conditions holds:*
- *C declares at least one virtual function:*
- *C has at least one virtual base*
- *C has at least one dynamic base*

$$\frac{\mathcal{M}(C)(msig) = \mathsf{true}}{\mathsf{isDynamic}(C)} \qquad \frac{C \overset{\mathcal{V}}{\to} B}{\mathsf{isDynamic}(C)} \qquad \frac{B \in \mathcal{DNV}(C) \cup \mathcal{DV}(C) \qquad \mathsf{isDynamic}(B)}{\mathsf{isDynamic}(C)}$$

If the hierarchy is well-formed, then this predicate is decidable[5].

In real-world implementations (for reasons that we shall see in the next chapter), run-time data is needed to perform dynamic-casts. Thus, following the C++ standard, we shall also restrict s++ dynamic casts to object of dynamic class types. Moreover, Lemma 4.4.3 (p. 89) allows us to restrict s++ dynamic cast to non-base classes:

$$\frac{\begin{array}{c} e(x) = \pi = (\ell, (\alpha, i, \sigma)) \qquad \mathcal{G} \vdash \langle \ell \rangle\ C_\circ[n_\circ] \prec\!\alpha\!\succ\!\xrightarrow{\mathcal{A}} C[n] \prec\!((i, \sigma))\!\succ\!\xrightarrow{\mathcal{CI}} B \\ \mathsf{isDynamic}(B) \qquad \nexists\sigma' : B \prec\!(\sigma')\!\succ\!\xrightarrow{\mathcal{I}} B' \qquad \mathsf{DynCast}(C, \sigma, B, B', \tilde{\sigma}') \\ \tilde{\pi}' = \mathsf{match}\ \tilde{\sigma}'\ \mathsf{with}\ \sigma' \mapsto (\ell, (\alpha, i, \sigma'))\ |\ \mathsf{NULL}_{B'} \mapsto \mathsf{NULL}_{B'}\ \mathsf{end} \qquad e' = e[x' \leftarrow \tilde{\pi}'] \end{array}}{\begin{array}{rl} & (x' := \texttt{dynamic\_cast}\langle B' \rangle_B(x),\ \ stl,\ \ e,\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\texttt{skip}\ ,\ \ stl,\ \ e',\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}}$$

(s++-dyncast)

4. Coq development: theory `Dynamic`.
5. Coq development: theory `DynamicWf`.

4.4.6 Virtual function call

4.4.6.1 Virtual function dispatch

Following Wasserrab et al., we write $\mathsf{VFDispatch}(C_\circ, \sigma', f, B'', \sigma'')$ to say that, if the most-derived object is of type $C_\circ$, then selecting the dispatch subobject for method f from an inheritance subobject σ' of $C_\circ$ yields the actual subobject σ'' of $C_\circ$ of static type B'' defining the actual function to call. Virtual function dispatch occurs as follows:

1. Let B be the static type of the pointer to the subobject σ' on which to operate. Then, statically choose the static resolving subobject σ_f of type B_f declaring f. This step, written $\mathsf{staticDispatch}(B, f, B_f, \sigma_f)$, actually finds a unique base class B_f of B declaring f (unless B itself declares f):

$$\frac{\begin{array}{c} B \prec\!\langle \sigma_f \rangle\!\overset{\mathcal{I}}{\rightarrowtail} B_f \\ \mathcal{M}(B_f)(f) \neq \bot \qquad \forall \sigma_2, B_2 : \begin{array}{c} B \prec\!\langle \sigma_2 \rangle\!\overset{\mathcal{I}}{\rightarrowtail} B_2 \land \mathcal{M}(B_2)(f) \neq \bot \Rightarrow \\ \exists \sigma_4 : B_f \prec\!\langle \sigma_4 \rangle\!\overset{\mathcal{I}}{\rightarrowtail} B_2 \land \sigma_2 = \sigma_f @ \sigma_4 \end{array} \end{array}}{\mathsf{staticDispatch}(B, f, B_f, \sigma_f)} \quad (\mathsf{static\text{-}dispatch})$$

   Contrary to other object-oriented operations such as field access, B is not required to declare f. However, this does not constrain real-life implementations to perform an implicit cast: this static dispatch step is mostly included in clever implementations of virtual dispatch.

2. Determine the *final overriders* for the methods. The final overriders are the inheritance subobjects σ'' of $C_\circ$ nearest to $C_\circ$ along the path $\sigma' @ \sigma_f$, as defined by the predicate $\mathsf{finalOverrider}(C_\circ, \sigma', B, f, B'', \sigma'')$ as follows:

$$\frac{\begin{array}{c} C_\circ \prec\!\langle \sigma' \rangle\!\overset{\mathcal{I}}{\rightarrowtail} B \\ \mathsf{staticDispatch}(B, f, B_f, \sigma_f) \qquad C_\circ \prec\!\langle \sigma'' \rangle\!\overset{\mathcal{I}}{\rightarrowtail} B'' \qquad \mathcal{M}(B'')(f) \neq \bot \qquad B'' \prec\!\langle \sigma''_f \rangle\!\overset{\mathcal{I}}{\rightarrowtail} B_f \\ \sigma' @ \sigma_f = \sigma'' @ \sigma''_f \qquad \forall \sigma_2, \sigma_4, B_2 : \begin{array}{c} C_\circ \prec\!\langle \sigma_2 \rangle\!\overset{\mathcal{I}}{\rightarrowtail} B_2 \prec\!\langle \sigma_4 \rangle\!\overset{\mathcal{I}}{\rightarrowtail} B'' \land \mathcal{M}(B_2)(f) \neq \bot \Rightarrow \\ (B'', \sigma'') = (B_2, \sigma_2) \end{array} \end{array}}{\mathsf{finalOverrider}(C_\circ, \sigma', B, f, B'', \sigma'')} \quad (\mathsf{final\text{-}overrider})$$

3. Finally, virtual function dispatch works only if there is a unique final overrider; and only if there is a base $B_\circ$ of $C_\circ$ declaring f virtual. This is a slight difference from [85] and [84], where all class member functions (methods) are virtual.

$$\frac{\begin{array}{c} \mathsf{finalOverrider}(C_\circ, \sigma, B, f, B'', \sigma'') \\ \forall (B', \sigma') : \mathsf{finalOverrider}(C_\circ, \sigma, B, f, B', \sigma') \Rightarrow (B', \sigma') = (B'', \sigma'') \\ B \prec\!\langle \sigma_\circ \rangle\!\overset{\mathcal{I}}{\rightarrowtail} B_\circ \qquad \mathcal{M}(B_\circ)(f) = \mathsf{true} \end{array}}{\mathsf{VFDispatch}(C_\circ, \sigma, f, B'', \sigma'')} \quad (\mathsf{virtual\text{-}dispatch})$$

4.4.6.2 s++ language construct

Finally, s++ virtual function call from an inheritance subobject σ of $C_\circ$ of static type B is defined as follows:

1. Perform the virtual function dispatch, via the VFDispatch predicate. It returns the dispatched subobject σ'' of static type B'' on which the actual function of B'' shall be called.
2. Then, the actually called function expects `this` pointing to the dispatched subobject. This requires an *adjustment* on the `this` pointer, to make it point to σ'' instead of σ.
3. Finally, pass that adjusted `this` pointer along with the arguments to the function.

$$\frac{\begin{array}{c} e(x) = \pi = (\ell, (\alpha, i, \sigma)) \qquad \mathcal{G} \vdash \langle \ell \rangle\ C[n] \prec\!\langle \alpha \rangle\!\overset{\mathcal{A}}{\rightarrowtail} C_\circ[n_\circ] \prec\!\langle (i, \sigma) \rangle\!\overset{\mathcal{CI}}{\rightarrowtail} B \\ \mathsf{VFDispatch}(C_\circ, \sigma, f, B'', \sigma'') \qquad \mathsf{methods}(B'', f) = \mathit{this}\texttt{->}(\mathit{varg}_1, \ldots, \mathit{varg}_n)\{\mathit{body}\} \\ \forall j, e(x_j) = v_j \qquad e' = \emptyset[\mathit{this} \leftarrow (\ell, (\alpha, i, \sigma''))][\mathit{varg}_1 \leftarrow v_1] \ldots [\mathit{varg}_n \leftarrow v_n] \end{array}}{\begin{array}{rlllll} & (x^? := x\texttt{->}_B f(x_1 \ldots x_n), & \mathit{stl}, & e, & & \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathit{body} & , \quad \epsilon, & e', & \mathsf{Callframe}(x^?, \mathit{stl}, e) :: & \mathcal{K}, \quad \mathcal{G}) \end{array}}$$

(s++-virtual-funcall)

Chapter 5

Formalization of object layout

Semantics of programming languages often define data types and structures at an abstract level (records, unions, arrays, classes, etc.) So, concrete implementation by compilers and interpreters have to find an adequate representation of those data types and structures in terms of low-level memory concepts (bits, pointers, etc.) Such a representation is called *layout.*

In this chapter, we formalize a family of layout algorithms for C++ objects with multiple inheritance, with some optimizations. Then, we prove that:

– any two different *scalar* fields, reachable through field or inheritance subobjects of the same object, are represented by disjoint memory zones. This allows showing the *good variable property* on concrete memory models such as CompCert [2].

– two different subobjects of the same type must be laid out at different memory addresses. This requirement is called the *object identity principle.*

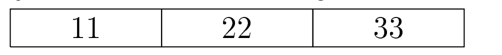
However, we shall also see that additional data are involved to implement polymorphic operations such as virtual function dispatch, dynamic casts, or even access to virtual bases. Then, we also have to prove that this additional data is stored disjointly from field data.

5.1 The object layout problem

Within a compound data structure, components are laid out at offsets relative to the start address of the memory area representing the whole structure.

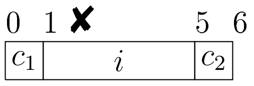
Arrays The most basic example is *arrays.* An array is an homogeneous compound data structure: all its components, called *cells*, are of the same type. Thus, in C++ as in C or Fortran, cells are laid out contiguously and consecutively in memory.

```
int i[3] = {11, 22, 33};
```



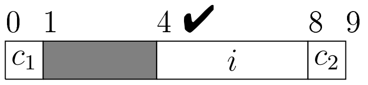
Structures But C and C++ also allow to define heterogeneous compound data structures, called `structs` (or *structures*). Within such a structure, components, called *members* (or *fields*), often are of different types. They may be still laid out consecutively in memory, such that a field within a structure is accessed through a constant offset within this structure, from the start address of the structure.

```
struct S {
  char c1;
  int  i;
  char c2;
};
```

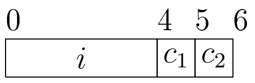


Alignment However, most architectures require that any low-level memory access be correctly *aligned*: for instance, an access to a 32-bit integer must be done at an address multiple of 4 bytes. For any elementary data types (numerical types, pointer types, etc.), a constant *alignment* is defined, which is a number such that accessing an elementary data through a memory address requires that this address be a multiple of the alignment of the corresponding elementary data type.

To ensure this constraint, a compiler may insert *padding* between two fields of a structure, to ensure that the second field be correctly aligned:



This padding space is, most often, unused. Thus, it may be relevant to reorder fields to minimize the amount of padding.



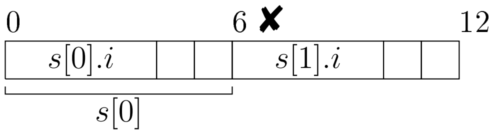
However, neither the C standard nor the C++ standard allow the compiler to reorder the fields by itself: they require fields to be laid out at increasing offsets following their declaration order. Thus, in both cases, the programmer has to be aware of this layout issue to manually determine the order of the fields in the structure. In C++ object-oriented model, this can trigger unexpected consequences at the level of the semantics itself, where some aspects depend on the field declaration order: for instance, fields are initialized in their declaration order.

So, for now we assume that the programmer has reordered the fields of `S` in the following way:

```
struct S {
  int  i;
  char c1;
  char c2;
};
```

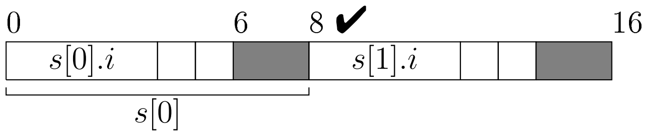
Arrays of structures Then, in C and in C++ structures may be themselves aggregated in arrays of structures:

```
S s[2];
```



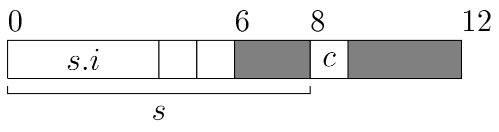
But the access to the fields of the second structure cell must still be correctly aligned. This is ensured by adding even more padding space at the end of the structure, called *tail padding.*

Then, the alignment of a structure is defined to be the least common multiple of the aligments of its fields. So, to ensure the correct alignment of the fields of each cell of a structure array, the aligment of the structure must evenly divide the total size of the structure.

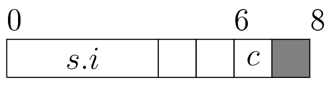


Embedded structures C and C++ allow structures to be themselves fields of other structures: this is called *embedding* (or *aggregation*).

```
struct T {
  S    s;
  char c;
};
```



Tail padding of embedded structures can incur unused space within the embedding structure. But the compiler can reuse this tail padding for the further fields of the embedding structure.



5.2 Impact of C++ multiple inheritance on data layout

5.2.1 Non-virtual inheritance

As with any other data structure, the principle is that, for any pointer to a subobject, the way to access its fields must not depend on how the subobject is reached. More generally, the implementation of operations on an object of some class type must not depend on whether it is a most-derived object or a subobject of another object.

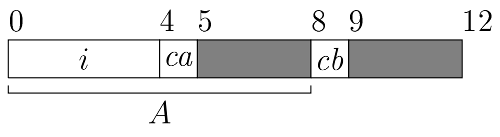
Thus, the non-virtual base classes of a class can be laid out as if they were structure fields of the class. This was the layout introduced by `cfront`.

C++ program:

```
struct A {
  int i;
  char ca;
};

struct B: A {
  char cb;
};
```

`cfront` layout

```
struct A {
  int i;
  char ca;
};

struct B {
  A a;
  char cb;
};
```

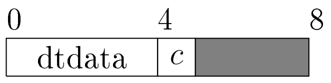


5.2.2 Dynamic type data

At the level of the abstract semantics, calling a method depends on the way the subobject is reached: this additional information is the *dynamic type* of the subobject. At the semantics level, it is actually the inheritance path from the most-derived object to the subobject.

Thus, at the implementation level, most C++ compilers introduce additional data at the beginning of some classes to store *dynamic type data*. In practice, when a virtual method is called from a subobject of some type A, most compilers such as GNU GCC perform function dispatch through a read-only *virtual function table*, or *virtual table*. Such compilers therefore store, within the subobject, a pointer to its corresponding virtual table, which is different depending on the dynamic type of the subobject. Our formalization does not depend on the actual data stored, it only fixes the size. In practice, such additional data are added in front of objects of some class type whenever this class has or inherits a virtual method.

```
struct A {
  char c;
  virtual void f();
}
```



The need for dynamic type data introduces the following requirement:

> Within an object, fields and dynamic type data of two (field or inheritance) subobjects of an object must be laid out in disjoint memory zones.

Moreover, this dynamic type data (often a pointer) also has an alignment constraint imposed by the underlying architecture, thus introducing a further requirement:

> Accesses to dynamic type data must be correctly aligned.

5.2.3 Virtual inheritance

A virtual base is shared between its derived classes. So, it would be incorrect to store a virtual base within its derived classes: such a layout would lose sharing and would result in multiple copies of the base being allocated.

Instead, it is mandatory to store a single copy of every virtual base at the level of the most-derived object only. Thus, a wise idea is to consider the *non-virtual part* of a subobject of class type C, which is composed of the fields defined in C and, recursively, the non-virtual parts of the direct non-virtual base class subobjects of C but exclude virtual bases.

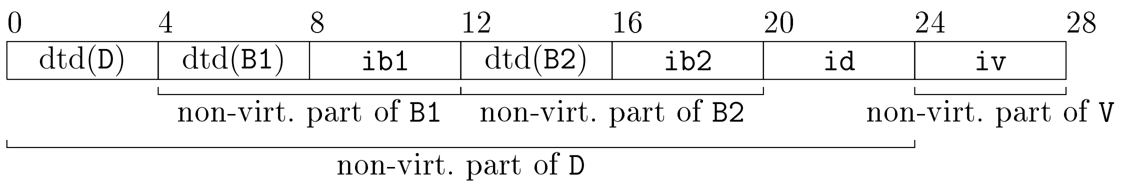
Example 5.2.1. Consider the following layout example:

Roughly speaking, the following hierarchy:

```
struct V {
  int iv;
};

struct B1: virtual V {
  int ib1;
};

struct B2: virtual V {
  int ib2;
};

struct D: B1, B2 {
  int id;
};
```

. . . may be laid out as follows:

```
struct V_non_virtual {
  int iv;
};

struct V_total {
  V_non_virtual v;
};

struct B1_non_virtual {
  int ib1;
};

struct B1_total {
  B1_non_virtual b1;
  V_non_virtual  v;
};

struct B2_non_virtual {
  int ib2;
};

struct B2_total {
  B2_non_virtual b2;
  V_non_virtual  v;
};

struct D_non_virtual {
  B1_non_virtual b1;
  B2_non_virtual b2;
  int id;
};

struct D_total {
  D_non_virtual d;
  V_non_virtual v;
};
```



Laying out the non-virtual part of a class first, before the virtual bases, allows to treat a pointer to a base class subobject of some type C in the same way as a pointer to a most-derived object of type C as regards accesses to fields and non-virtual base class subobjects.

Following this layout scheme, we have to consider not only the size of a most-derived object (which would be retrieved by the C++ `sizeof` operator), but also the size of the non-virtual part of a class.

Similarly, it is worth noting that the alignment of the non-virtual part of a class may be weaker than the alignment of a most-derived object of this class. That is, the alignment of `B_total` may be a strict multiple of the alignment of `B_non_virtual`. This leads us to distinguish two alignments for a class:

– the *alignment* of a class C, such that any instance of C be aligned at an offset multiple of this alignment
– the *non-virtual alignment* of a class C, to allow the alignment of C as a base of other classes, so that any subobject of type C be aligned at an offset multiple of this alignment

Dynamic type data In summary, a most-derived object of some type D is composed of the non-virtual part of D and the non-virtual parts of all (direct or indirect) virtual bases of D. This illustrates that the position of the virtual bases depends on the most-derived object: it is impossible to statically know the position of the virtual base class subobjects of a subobject that is not a most-derived object. Thus, at the implementation level, additional data will be necessary for a subobject to reach its virtual bases. In practice, such additional data will be read through dynamic type data. So, such dynamic type data will also be added for subobjects of classes having virtual bases.

Thus, the C++ Standard introduces the notion of *dynamic class* (*Definition* 4.4.2 p. 89). In practice, this notion corresponds to classes requiring dynamic type data. The fact that this notion is required by the Standard, illustrates that *dynamic cast* is often implemented using dynamic type data.

5.3 Optimizations

5.3.1 Dynamic type data sharing

Systematically adding dynamic type data in front of the non-virtual parts of all dynamic classes would significantly waste space.

In practice, compilers optimize the layout of objects of dynamic class type to allow sharing the dynamic data between a class and one of its direct non-virtual base class subobjects. Such a base class is called the *direct non-virtual primary base* of the class.

Then, two subobjects are allowed to share their dynamic type data if, and only if, one is a *non-virtual primary base* class subobject of another. A class B is a non-virtual primary base of a class D if, and only if, either it is the direct non-virtual primary base of D or it is a non-virtual primary base of the direct non-virtual primary base of D.

The first C++ compiler, `cfront`, already implemented this optimization.

We refine our layout scheme by formalizing the following layout for a full instance of some class C, common to many C++ compilers:

– First, the non-virtual part of C, consisting of:

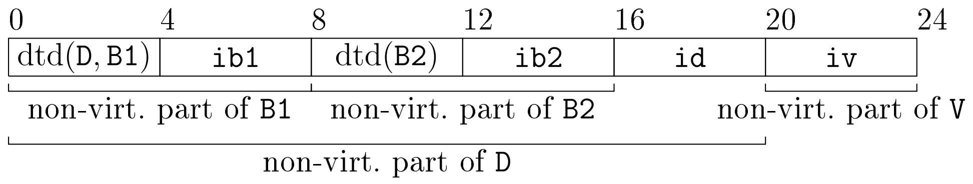
1. The non-virtual part of a dynamic non-virtual direct base A of C, if such class exists
2. If C is dynamic but such A does not exist, then some space for the *dynamic type data* for C

3. Then the non-virtual parts of other non-virtual direct bases of C
4. Then the fields of C

– Then the non-virtual parts of all virtual bases of C, so that they never overlap
The requirement on dynamic type data is then:

Within an object, the dynamic type data of two subobjects must be laid out in disjoint memory zones, unless one subobject is a non-virtual primary base of another.

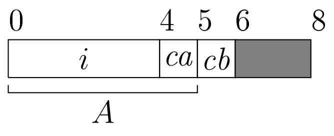
The hierarchy of Example 5.2.1 (p. 96) can be laid out in the following optimized way compliant to this requirement:



5.3.2 Reusing tail padding

The layout proposed above is the historical layout adopted by Stroustrup's `cfront`, which was a C++-to-C translator. Its main drawback is that, as non-virtual parts of subobjects are laid out like embedded structures, their tail padding is not reused. Such reuse of tail padding is perfectly safe. It can be achieved easily by defining the non-virtual size of a class as excluding its tail padding.

```
struct A {
  int i;
  char ca;
};

struct B: A {
  char cb;
};
```



5.3.3 Empty base optimization

The STL (C++ standard template library) and the Boost libraries make extensive use of *empty classes*. Consider the following function template provided by the STL to implement a sorting algorithm on arrays:

```
template <typename Ran, typename Comp>
void sort(Ran first, Ran last, Comp cmp);
```

This template is parameterized by two types, `Ran` which is the type of arrays to sort, and `Comp` which is the type of the comparator. Now we want to use this sorting algorithm on integer arrays. Thus, we instantiate `Ran` with `int*`, and we have to instantiate `Comp` for the comparator. To this end, we define the following `MyGreater` class:

```
struct MyGreater {
  typedef int first_argument_type;
  typedef int second_argument_type;
  typedef bool result_type;
  bool operator()(int i, int j) const {
    return i > j;
  }
};
```

so that sorting an array `t` of `n` integers is then performed by:

```
sort(t, t+n, MyGreater());
```

where `MyGreater()` creates an instance of the `MyGreater` class. This instance actually represents a comparison function to pass to the `sort` function. It contains no data, so we expect compilers to minimize its memory footprint.

However, in practice, the STL provides ready-to-use classes to implement such operators. For instance, the following `std::binary_function` template defines a class to represent functions with two arguments:

```
template <typename FIRST, typename SECOND, typename RESULT>
struct std::binary_function {
  typedef FIRST first_argument_type;
  typedef SECOND second_argument_type;
  typedef RESULT result_type;
};
```

so that the `MyGreater` function can inherit from this class:

```
struct MyGreater:
  std::binary_function<int,int,bool> {
  bool operator()(int i, int j) const {
    return i > j;
  }
}
```

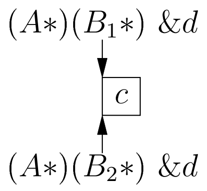
One can see that `std::binary_function` contains no data and no functions: it is completely empty, and actually serves as a tagging class for the purpose of static typing. So, we expect the compiler to minimize its footprint within a `MyGreater` instance. However, such optimizations must not contradict the object identity principle: if a class inherits from an empty class through distinct paths, those different subobjects must be still laid out at distinct offsets.

A naive approach Objects of empty class type must have a nonzero size, for instance to distinguish two distinct cells of an array of empty class objects. Consider the following hierarchy:

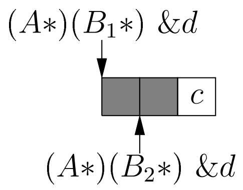
```
struct A         {};  /* empty */
struct B1: A     {};  /* empty */
struct B2: A     {};  /* empty */
struct D : B1, B2 { char c; };
```

Assume therefore `sizeof(A)` $= 1$. Then, it would be sound to have `sizeof(B1)` $= 1$ and `sizeof(B2)` $= 1$.

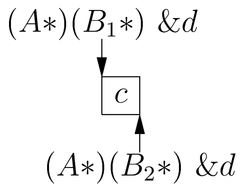
The following layout is illegal, confusing the two different `A` subobjects of `D`:



A naive compiler would lay out `B1`, `B2` and `c` in disjoint memory locations, yielding `sizeof(C)` $=$ 3.

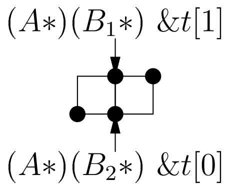


A clever optimization However, it is consistent with the Standard to allow `c` to overlap empty bases:

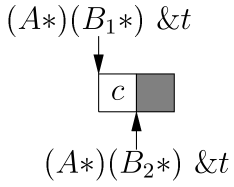


This optimization is called *empty base optimization*: empty bases can overlap the data of non-empty bases or fields. Despite this optimization, offsets for `B1` and `B2` are still distinct. However, and perhaps surprisingly, we must actually take `sizeof(D)` $= 2$. The explanation follows: consider an array of two `D` cells, then we have to ensure that the two `a1` and `a2` subobjects below be distinct:

```
D t[2];
A* a2 = (A*)(B2*)&t[0];
A* a1 = (A*)(B1*)&t[1];
```



Thus, further tail padding has to be added to `D` to avoid such illegal sharing, yielding indeed a size of 2:



So, two empty bases may overlap only with care. We shall see later in more detail the precise constraints to satisfy before an empty base can overlap other data. This is the main reason why C++ object layout cannot be mapped to C structure representations as `cfront` [80] does.

However, our work also shows that it is possible, without breaking the semantics of a C++ program, to generalize this empty base optimization to empty structure members, as suggested by Myers [63]. This is the reason why our formalization gives no precise definition for empty classes. Instead, some constraints are enforced:

Hypothesis 5.3.1. *A layout algorithm may define its own notions of* empty *and* dynamic *classes, subject to the following constraints:*

- *An empty class must have no scalar fields*
- *An empty class must have no non-empty base*
- *An empty class must have no structure field of non-empty class type (regardless, however, of the number of array cells)*
- *A dynamic class must not be empty*

Chapter 6 (p. 127) describes layout algorithms giving precise definitions for empty classes, sometimes more restrictive than necessary to lower compilation time, at the price of losing run-time optimizations on the programs being compiled.

Buggy implementations Our interest in mechanically formalizing empty base and empty member optimizations and proving their correctness stems from the presence of bugs in real-world compilers. In his argument for empty base and empty member optimizations, Myers [63] gives examples of compilers from MetroWerks (CodeWarrior C++ 4.0) and IBM that make too agressive optimizations not complying with the object identity requirement. Such aggressive optimizations were also performed by compilers such as Microsoft Visual C++ 7.1 and Borland C++ Builder 5.x [38], which erroneously identify the base class subobject `((Empty*) &d)` with the data member subobject `&(d.value)` in the following example:

```
struct Empty {};
struct Derived: Empty {
  Empty value;
};
Derived d;
```

Sizes and data sizes When dealing with empty classes that can overlap other components, considering their size or non-virtual size is not enough (as those sizes are necessary not null). We therefore introduce the notion of *data size*. Roughly speaking, the *data size* of a class is an upper bound on the sum of the sizes of all scalar fields accessible from this class (through inheritance and/or structure array field paths). The *non-virtual data size* of a class is an upper bound on the sum of the sizes of all scalar fields accessible from this class *when the first inheritance step is non-virtual inheritance* (i.e. virtual inheritance is allowed only once a structure field within a non-virtual base of C is reached). Both sums also include the sizes of the dynamic type data for all relevant bases.

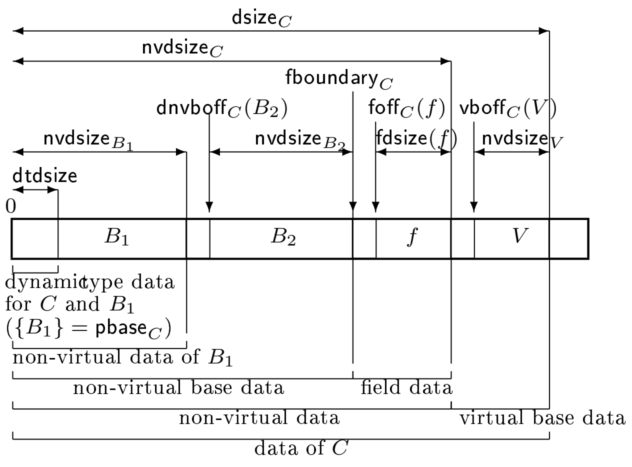
The data size and non-virtual data size are not relevant for empty classes.

5.4 Formal interface of a layout algorithm

In this section, we describe the parameters that a layout algorithm is expected to compute. Consider the following hierarchy:

```
struct X  {...};
struct V  {...};
struct B1: virtual V  {...};
struct B2: virtual V  {...};
struct B3 {};
struct C:  B1, B2, B3 { X f; };
```

Offsets and data sizes The following figure depicts the parameters related to data sizes for a most-derived C object. It illustrates that the data zones of class components (for non-empty bases and fields) must not overlap.

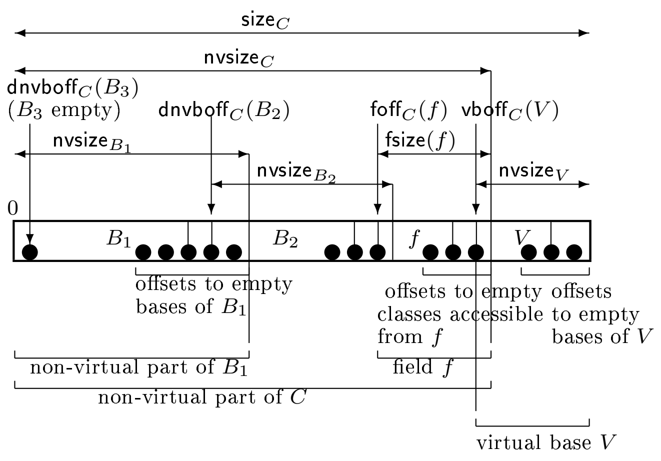


This figure introduces the following parameters:

– dnvboff$_C$: $\mathcal{DNV}(C) \nrightarrow \mathbb{N}$ is a finite map assigning to each non-virtual direct base of C an offset within a subobject of type C. This offset corresponds to the direct non-virtual subobject of type B within a subobject of type C.
– vboff$_C$: $\mathcal{V}(C) \nrightarrow \mathbb{N}$ is a finite map assigning to each virtual base of C an offset within a most-derived C object.
– fboundary$_C \in \mathbb{N}$ is the boundary between the non-virtual data for the direct non-virtual bases of C and the *non-empty* fields of C
– foff$_C$: $Sc\mathcal{F}(C) \cup St\mathcal{F}(C) \nrightarrow \mathbb{N}$ is a finite map assigning to each (scalar or structure array) field of C an offset within a subobject of type C.
– nvdsize$_C$ is the *non-virtual data size* of C
– dsize$_C$ is the *data size* of a most-derived C object

Additionally, this figure introduces the data size fdsize$_f$ of each field f of C, which can be deduced from the other parameters, as we shall see in *Definition* 5.5.6 (p. 111).

Total sizes The following figure depicts the parameters related to total sizes. It illustrates that empty components may have their offsets (represented by the bullets) lie within the memory span of other components, as long as two distinct components of the same type are laid out at distinct offsets:



This figure introduces the following parameters:

– nvsize_C is the *non-virtual size* of C

– size_C is the *size* of a most-derived C object, as would be given by the `sizeof` operator

Additionally, this figure introduces the total size fsize_f of each field f of C, which can be deduced from the other parameters, as we shall see in *Definition* 5.5.2 (p. 107).

Summary

***Definition* 5.4.1.** *A* class layout *is a tuple of class-indexed families:*

$$\begin{array}{l}(\\ \quad (\mathsf{dnvboff}_C), (\mathsf{vboff}_C), (\mathsf{fboundary}_C), (\mathsf{foff}_C), (\mathsf{nvdsize}_C), (\mathsf{dsize}_C),\\ \quad (\mathsf{nvsize}_C), (\mathsf{size}_C),\\ \quad (\mathsf{nvalign}_C), (\mathsf{align}_C)\\ )_{C\in\mathcal{C}}\end{array}$$

where, for each class C, in addition to the other parameters described before:

– $\mathsf{nvalign}_C$ *is the* non-virtual alignment *of C*

– align_C *is the* alignment *of C*

As the hierarchy is assumed to be well-founded, a class layout algorithm can incrementally compute offsets for a given class C, assuming that for all classes $B \prec C$, offsets have already been computed.

Then, given such a class layout, it is possible to compute the offset of non-virtual paths within a subobject of some class C, and the offset of any inheritance path within a full instance of C.

***Definition* 5.4.2.** *If l is a non-virtual path from some class C to some class A, then* $\mathsf{nvsoff}(l) \in \mathbb{N}$ *shall denote its offset within a subobject of type C. It is computed as follows, by structural recursion on l:*

– $\mathsf{nvsoff}(C :: \epsilon) \underset{\text{def.}}{=\!=} 0$ *($C :: \epsilon$ is the trivial non-virtual path to the subobject itself).*

– *if $l = C :: B :: l'$, then $B \in \mathcal{DNV}(C)$ and $B :: l'$ is a non-virtual path from B to A, and* $\mathsf{nvsoff}(C :: B :: l') \underset{\text{def.}}{=\!=} \mathsf{dnvboff}_C(B) + \mathsf{nvsoff}(B :: l')$.

***Definition* 5.4.3.** *If (h, l) is a path from C to A, then its offset* $\mathsf{soff}_C(h, l) \in \mathbb{N}$ *within a full instance of C is computed as follows:*

– $\mathsf{soff}_C(\mathsf{Repeated}, l) \overset{}{\underset{\text{def.}}{=\!=}} \mathsf{nvsoff}(l)$,
– $\mathsf{soff}_C(\mathsf{Shared}, l) \underset{\text{def.}}{=\!=} \mathsf{vboff}_C(B) + \mathsf{nvsoff}(l)$, *where* $B \in \mathcal{V}(C)$ *and* l *is a non-virtual path from* B *to* A

The above definition can be simplified by the convention $\mathsf{vboff}_C(C) \underset{\text{def.}}{=\!=} 0$, thanks to the well-foundedness of the hierarchy: indeed, C is not a virtual base of itself, so $\mathsf{vboff}_C(C)$ is theoretically undefined. Then, with this convention, we obtain for any path $(h, B :: l')$ from C to A:

$$\mathsf{soff}_C(h, B :: l') = \mathsf{vboff}_C(B) + \mathsf{nvsoff}(B :: l)$$

regardless of h. Remember that Lemma 4.1.9 (p. 79) makes h depend on whether $B = C$ or not. This leads us to introduce the notion of *generalized virtual bases* of a class:

***Definition* 5.4.4.** *The* generalized virtual bases *of* C *are* C *and the virtual bases of* C*. This set, written* $\tilde{\mathcal{V}}(C)$*, shall be the domain of* vboff_C*:*

$$\tilde{\mathcal{V}}(C) \underset{\text{def.}}{=\!=} \mathcal{V}(C) \cup \{C\}$$

$$\begin{array}{rccl} \mathsf{vboff}_C : & \tilde{\mathcal{V}}(C) & \to & \mathbb{N} \\ & C & \mapsto & 0 \\ & B \in \mathcal{V}(C) & \mapsto & \mathsf{vboff}_C(B) \end{array}$$

Lemma 5.4.1. *If* l' *is a non-virtual path from some class* B *to some class* A*, then casting a subobject from* B *to* A *through path* l' *is equivalent to adding the offset* $\mathsf{nvsoff}(l')$ *to the offset of the subobject. That is, for any path* σ *from* C *to* B*:*

$$\mathsf{soff}_C(\sigma @ (\mathsf{Repeated}, l')) = \mathsf{soff}_C(\sigma) + \mathsf{nvsoff}(l')$$

This shows that non-virtual derived-to-base casting does not depend on the starting subobject. However, to make a derived-to-base static cast through virtual inheritance, it is necessary to know the offset of the virtual base from the *full* object, so a subobject has to know its offset from the full object. Chen [20] mechanically formalizes a type system for C++ multiple inheritance to determine which arithmetic operations are necessary for casts. We follow a different approach by directly proving the correctness of a compiler (Chapter 7 p. 135).

Given those definitions for inheritance path offsets, we can compute the offset of an array of n' structures of type C' within an array of n structures of type C:

***Definition* 5.4.5.** *Let* C, C' *be two classes, and* n, n' *two integers. If* α *is an array path from* $C[n]$ *to* $C'[n']$*, then the offset* $\mathsf{aoff}_C(\alpha)$ *of* $C'[n']$ *within* $C[n]$ *through the array path* α *is computed as follows:*

–

$$\mathsf{aoff}_C(\epsilon) \underset{\text{def.}}{=\!=} 0$$

– *if* $\alpha = (i, \sigma, f) :: \alpha'$*, then* i *is the index in the array* $C[n]$*, so* $0 \leq i < n$*, and there is a class* B *such that* σ *is an inheritance path from* C *to* B*, and* f *is a field of* B *declared as an array of* q *structures of type* A *for some* q *and* A*, so that* α' *is an array path from* $A[q]$ *to* $C'[n']$*. Then we define:*

$$\mathsf{aoff}_C((i, \sigma, f) :: \alpha') \underset{\text{def.}}{=\!=} i \cdot \mathsf{size}_C + \mathsf{soff}_C(\sigma) + \mathsf{foff}_B(f) + \mathsf{aoff}_A(\alpha')$$

***Definition* 5.4.6.** *Let $p = (\alpha, i, \sigma)$ be a generalized subobject (or relative pointer) from an array of some type $C[n]$, where α is an array path from $C[n]$ to $C'[n']$ for some class C'. Then, the offset of p within $C[n]$ is written:*

$$\mathsf{off}(p) \underset{\text{def.}}{=\!=} \mathsf{aoff}_C(\alpha) \; + \; i \cdot \mathsf{size}_{C'} \; + \; \mathsf{soff}_{C'}(\sigma)$$

5.5 Soundness conditions

In our formalism, rather than imposing a fixed object layout as for C structures in CompCert [2], we propose some sufficient conditions [1] on object layout, which layout algorithms must abide by. Then we prove [2] that those constraints make two distinct scalar fields disjoint and two pointers to differents subobjects of the same static type point to distinct memory locations.

Throughout this section, we consider a fixed class C.

5.5.1 Total size

The total size size_C of a class C is specified as an upper bound on the sizes of all its components. In the same way, the total non-virtual size nvsize_C of a class C is an upper bound on the sizes of all its non-virtual components (the non-virtual sizes of its direct non-virtual bases, and the sizes of its fields).

Hypothesis 5.5.1. *The non-virtual size of C is an upper bound on the non-virtual sizes of all its direct non-virtual bases:*

$$\forall B \in \mathcal{DNV}_C : \mathsf{dnvboff}_C(B) + \mathsf{nvsize}_B \leq \mathsf{nvsize}_C \qquad \text{(nvsize-upper-bound-bases)}$$

Lemma 5.5.1. *If l is a non-virtual path from C to some class A, then the non-virtual part of A is included in the non-virtual part of C:*

$$\mathsf{nvsoff}(l) + \mathsf{nvsize}_A \leq \mathsf{nvsize}_C$$

Proof. By induction on the length of l.

If l is the trivial path $C :: \epsilon$, then $A = C$, $\mathsf{nvsoff}(l) = 0$ and the inequality is an equality.

If $l = C :: B :: l'$, then $B \in \mathcal{DNV}(C)$ and $B :: l'$ is a non-virtual path from B to A, and we have $\mathsf{nvsoff}(C :: B :: l') = \mathsf{dnvboff}_C(B) + \mathsf{nvsoff}(B :: l')$. By induction hypothesis, we have $\mathsf{nvsoff}(B :: l') + \mathsf{nvsize}_A \leq \mathsf{nvsize}_B$. By the above hypothesis, we have $\mathsf{dnvboff}_C(B) + \mathsf{nvsize}_B \leq \mathsf{nvsize}_C$, which concludes. □

Hypothesis 5.5.2. *The total size of C is an upper bound on the non-virtual sizes of all its generalized virtual bases (including C itself):*

$$\forall B \in \tilde{\mathcal{V}}(C) : \mathsf{vboff}_C(B) + \mathsf{nvsize}_B \leq \mathsf{size}_C \qquad \text{(fullsize-upper-bound)}$$

1. Those conditions appear in a different order than in our POPL 2011 article [72]. The index of equations (p. 353) pairs the condition numbers of our POPL 2011 article with their actual names in the present thesis.

2. Coq development: theory `LayoutConstraints`.

Lemma 5.5.2. *If (h, l) is an inheritance path (= base class subobject) from C to some class A, then the non-virtual part of A is included in C:*

$$\mathsf{soff}_C(h, l) + \mathsf{nvsize}_A \leq \mathsf{size}_C$$

Proof. Recall that $l = B :: l'$ for some class $B \in \tilde{\mathcal{V}}(C)$ and some l' such that $B :: l'$ is a non-virtual path from B to A. We then have $\mathsf{soff}_C(h, l) = \mathsf{vboff}_C(B) + \mathsf{nvsoff}(B :: l')$. (fullsize-upper-bound, p. 106) gives us $\mathsf{vboff}_C(B) + \mathsf{nvsize}_B \leq \mathsf{size}_C$. Lemma 5.5.1 (p. 106) gives us $\mathsf{nvsoff}(B :: l') + \mathsf{nvsize}_A \leq \mathsf{nvsize}_B$, which concludes. □

The non-virtual part of a class C contains all fields defined in C. To express this condition, we must define the *size* of a field.

***Notation* 5.5.1.** *We assume that there exists a function* $\mathsf{scsize} : \mathcal{A} \cup \mathcal{C} \to \mathbb{N}^{>0}$ *computing the size of a scalar value (atom, or pointer to C for some class C).*

The function scsize should not depend on the layout algorithm. For most compilers, the size of a value of type pointer to T does not depend on T. We shall see that the compiler actually also needs such a condition.

***Definition* 5.5.2.** *The* size $\mathsf{fsize}(f)$ *of a field f is defined as follows:*

$$\mathsf{fsize}(\texttt{scalar}\ t\ \mathit{fname}) \underset{\text{def.}}{=\!=} \mathsf{scsize}(t) \qquad \textit{(for a scalar field)}$$
$$\mathsf{fsize}(\texttt{struct}\ B[n]\ \mathit{fname}) \underset{\text{def.}}{=\!=} n \cdot \mathsf{size}_B \qquad \textit{(for a structure array field of } n \textit{ cells of type } B\textit{)}$$

Hypothesis 5.5.3. *The total non-virtual size of C is an upper bound on the sizes of all its fields:*

$$\forall F \in \mathcal{F}(C) : \mathsf{foff}_C(F) + \mathsf{fsize}(F) \leq \mathsf{nvsize}_C \qquad \text{(nvsize-upper-bound-fields)}$$

Lemma 5.5.3. *If α is an array path from $C[n]$ to $C'[n']$, then the designated array $C'[n']$ is "included" in $C[n]$:*

$$\mathsf{aoff}_C(\alpha) + n' \cdot \mathsf{size}_{C'} \leq n \cdot \mathsf{size}_C$$

Proof. By structural induction on α.

- If $\alpha = \epsilon$, then $C' = C$ and $n' = n$ and $\mathsf{aoff}_C(\alpha) = 0$, so the inequality is an equality.
- Otherwise, $\alpha = (i, (h, l), f) :: \alpha'$ where $0 \leq i < n$, (h, l) is an inheritance path from C to some class B, and f is a structure array of some type $A[m]$ defined in B. So, we have $\mathsf{aoff}_C(\alpha) = i \cdot \mathsf{size}_C + \mathsf{soff}_C(h, l) + \mathsf{foff}_B(f) + \mathsf{aoff}_A(\alpha')$, and :
  - by induction hypothesis, $\mathsf{aoff}_A(\alpha') + n' \cdot \mathsf{size}_{C'} \leq m \cdot \mathsf{size}_A = \mathsf{fsize}(f)$ by definition
  - $\mathsf{foff}_B(f) + \mathsf{fsize}(f) \leq \mathsf{nvsize}_B$ by (nvsize-upper-bound-fields, p. 107)
  - $\mathsf{soff}_C(h, l) + \mathsf{nvsize}_B \leq \mathsf{size}_C$ by Lemma 5.5.2 (p. 107)
  - $i + 1 \leq n$ by hypothesis

  which concludes. □

Corollary 5.5.4. *If p is a relative pointer (= generalized subobject) from $C[n]$ to some class A, then the non-virtual part of A is included in $C[n]$:*

$$\mathsf{off}_C(p) + \mathsf{nvsize}_A \leq n \cdot \mathsf{size}_C$$

Proof. Immediately follows from Lemma 5.5.2 (p. 107) and Lemma 5.5.3 (p. 107). □

5.5.2 Alignment

The non-virtual alignment of a class C is a boundary used to align C as a non-virtual base of another class. So, it has to be a multiple of the alignments of all the non-virtual components of C: the alignments of fields defined in C, and the non-virtual alignments of non-virtual bases of C.

So we must first define field alignments.

***Notation* 5.5.3.** *We assume that there exists a function* $\mathsf{scalign} : \mathcal{A} \cup \mathcal{C} \to \mathbb{N}^{>0}$ *computing the alignment of a scalar value (atom, or pointer to B for some class B).*

As with scsize, the function $\mathsf{scalign}$ should not depend on the layout algorithm.

***Definition* 5.5.4.** *The* alignment $\mathsf{falign}(f)$ *of a field f is defined as follows:*
– $\mathsf{falign}(\texttt{scalar}\ t\ \mathit{fname}) \overset{}{\underset{\text{def.}}{=\!=}} \mathsf{scalign}(t)$ *for a scalar field*
– $\mathsf{falign}(\texttt{struct}\ B[n]\ \mathit{fname}) \underset{\text{def.}}{=\!=} \mathsf{align}_B$ *for a structure array field*

Indeed, a structure array field contains “full” instances of some class B, so the field must be aligned to the whole alignment of B, not only the non-virtual alignment of B.

Hypothesis 5.5.4. *The non-virtual alignment of a class C is a multiple of the alignments of its fields. Any field f of C must be laid out with respect to its alignment:*

$$\forall F \in \mathcal{F}(C) : (\mathsf{falign}(F) \mid \mathsf{nvalign}_C) \wedge (\mathsf{falign}(F) \mid \mathsf{foff}_C(f)) \tag{falign}$$

Hypothesis 5.5.5. *The non-virtual alignment of a class C is a multiple of the non-virtual alignments of its direct non-virtual bases. Any non-virtual base B of C must be laid out with respect to its non-virtual alignment::*

$$\forall B \in \mathcal{DNV}(C) : (\mathsf{nvalign}_B \mid \mathsf{nvalign}_C) \wedge (\mathsf{nvalign}_B \mid \mathsf{dnvboff}_C(B)) \tag{nvalign}$$

Finally, the alignment of a class C synthesizes the non-virtual alignments of C and its virtual bases.

Hypothesis 5.5.6. *The alignment of a class C is a multiple of the non-virtual alignments of its generalized virtual bases (including C itself). Any virtual base B of C must be laid out with respect to its non-virtual alignment:*

$$\forall B \in \tilde{\mathcal{V}}(C) : (\mathsf{nvalign}_B \mid \mathsf{align}_C) \wedge (\mathsf{nvalign}_B \mid \mathsf{vboff}_C(B)) \tag{align}$$

Then, all those constraints allow to show that the access to a field is correctly aligned.

Lemma 5.5.5. *If l is a non-virtual path from some class C to some class A, then the access to A through l is correctly aligned:*

$$(\mathsf{nvalign}_A \mid \mathsf{nvalign}_C) \tag{i}$$

$$(\mathsf{nvalign}_A \mid \mathsf{nvsoff}(l)) \tag{ii}$$

Proof. By structural induction on l.
– If $l = \epsilon$, then $C = A$ and $\mathsf{nvsoff}(l) = 0$, which trivially concludes.

- Otherwise, $l = B :: l'$ with $B \in \mathcal{DNV}(C)$ and $\mathsf{nvsoff}(l) = \mathsf{dnvboff}_C(B) + \mathsf{nvsoff}(l')$. So:
  - $(\mathsf{nvalign}_B \mid \mathsf{nvalign}_C)$ by ($\mathsf{nvalign}$, p. 108) and $(\mathsf{nvalign}_A \mid \mathsf{nvalign}_B)$ by induction hypothesis, thus (i) by transitivity.
  - $(\mathsf{nvalign}_B \mid \mathsf{dnvboff}_C(B))$ by ($\mathsf{nvalign}$, p. 108), and $(\mathsf{nvalign}_A \mid \mathsf{nvalign}_B)$ by induction hypothesis, and $(\mathsf{nvalign}_A \mid \mathsf{nvsoff}(l'))$ by induction hypothesis, thus (ii) by transitivity and compatibility of $(. \mid .)$ with addition. □

Lemma 5.5.6. *If (h, l) is an inheritance path (= base class subobject) from some class C to some class A, then the access to A through (h, l) is correctly aligned:*

$$(\mathsf{nvalign}_A \mid \mathsf{align}_C) \qquad (i)$$

$$(\mathsf{nvalign}_A \mid \mathsf{soff}_C(h, l)) \qquad (ii)$$

Proof. Recall that $l = B :: l'$ for some class $B \in \tilde{\mathcal{V}}(C)$ and some l' such that $B :: l'$ is a non-virtual path from B to A. We then have $\mathsf{soff}_C(h, l) = \mathsf{vboff}_C(B) + \mathsf{nvsoff}(B :: l')$.
- (align, p. 108) gives us $(\mathsf{nvalign}_B \mid \mathsf{align}_C)$; part (i) of Lemma 5.5.5 (p. 108) gives us $(\mathsf{nvalign}_A \mid \mathsf{nvalign}_B)$, thus (i) by transitivity.
- (align, p. 108) also gives us $(\mathsf{nvalign}_B \mid \mathsf{vboff}_C(B))$; Lemma 5.5.5 (p. 108) gives us $(\mathsf{nvalign}_A \mid \mathsf{nvalign}_B)$ and $(\mathsf{nvalign}_A \mid \mathsf{nvsoff}(l'))$, thus (ii) by transitivity and compatibility of $(. \mid .)$ with addition. □

If C is a class and c is an array of several structures of type C, then access to the second structure item of c will be realized at low-level through adding an offset of `sizeof`(C) bytes to a pointer to c. So, the following constraint is necessary to ensure the correct alignment of such an access:

Hypothesis 5.5.7.

$$(\mathsf{align}_C \mid \mathsf{size}_C) \qquad (\mathsf{align\text{-}size})$$

This condition ensures the following property:

Lemma 5.5.7. *If l is an array path from $C[n]$ to $C'[n']$, then the designated array $C'[n']$ is correctly aligned in $C[n]$:*

$$(\mathsf{align}_{C'} \mid \mathsf{align}_C) \qquad (i)$$

$$(\mathsf{align}_{C'} \mid \mathsf{aoff}_C(l)) \qquad (ii)$$

Proof. By structural induction on l.
- If $l = \epsilon$, then $C' = C$ and $\mathsf{aoff}_C(l) = 0$, which trivially concludes.
- Otherwise, $l = (i, (h, p), f) :: l'$ where $0 \leq i < n$, (h, p) is an inheritance path from C to some class B, and f is a structure array of some type $A[m]$ defined in B, such that $\mathsf{falign}(f) = \mathsf{align}_A$. So, we have $\mathsf{aoff}_C(l) = i \cdot \mathsf{size}_C + \mathsf{soff}_C(h, p) + \mathsf{foff}_B(f)$, and:
  - $(\mathsf{align}_{C'} \mid \mathsf{align}_A)$ by induction hypothesis, and $(\underset{\mathsf{align}_A}{\underset{\shortparallel}{\mathsf{falign}(f)}} \mid \mathsf{nvalign}_B)$ by (falign, p. 108), and $(\mathsf{nvalign}_B \mid \mathsf{align}_C)$ by part (i) of Lemma 5.5.6 (p. 109), thus (i) by transitivity

– thanks to (i) and ($\mathsf{align\text{-}size}$, p. 109), we have $(\mathsf{align}_{C'} \mid i \cdot \mathsf{size}_C)$. Moreover, by induction hypothesis, $(\mathsf{align}_{C'} \mid \mathsf{align}_A)$, and $(\mathsf{align}_A \mid \mathsf{nvalign}_B)$ as above, and $(\mathsf{nvalign}_B \mid \mathsf{soff}_C(h,p))$ by part (ii) of Lemma 5.5.6 (p. 109), and $(\mathsf{align}_A \mid \mathsf{foff}_B(f))$ by (falign, p. 108), and $(\mathsf{align}'_C \mid \mathsf{aoff}_A(l'))$ by induction hypothesis, thus (ii). □

Corollary 5.5.8. *If p is a relative pointer (= generalized subobject) from $C[n]$ to some class A, then the access to A through p is correctly aligned:*

$$(\mathsf{nvalign}_A \mid \mathsf{align}_C)$$

$$(\mathsf{nvalign}_A \mid \mathsf{off}_C(p))$$

Proof. Immediately follows from Lemma 5.5.6 (p. 109) and Lemma 5.5.7 (p. 109). □

This lemma stresses out the difference of use between the alignment and the non-virtual alignment of a class: whereas full instances are aligned to the full alignment of their class, by contrast, generalized subobjects are aligned to the non-virtual alignment of their static type.

This finally allows to show a general alignment guarantee:

Theorem I.1 (Scalar field alignment). *If p is a generalized subobject of static type A from a structure array of C, and if* `scalar` $t\ f$ *is a scalar field of A of scalar type $t \in \mathcal{A} \cup \mathcal{C}$, then the access to this field is correctly aligned:*

$$(\mathsf{scalign}(t) \mid \mathsf{align}_C)$$

$$(\mathsf{scalign}(t) \mid \mathsf{off}_C(p) + \mathsf{foff}_A(f))$$

Proof. Immediately follows from Corollary 5.5.8 (p. 110) and condition (falign, p. 108). □

5.5.3 Data size

We aim at stating constraints to forbid the overlapping of different scalar data. So, we shall only consider here *non-empty* components, in the sense that roughly speaking, a component is non-empty as soon as it contains some scalar data.

Then, to express that different scalar data do not overlap, we shall not use the sizes, but the *data sizes* of the relevant components. In this section, we shall show that data sizes are not relevant for empty components.

Recall that the actual definition for an empty class is left to the layout algorithm. Based on this definition, we introduce the notion of *empty field*, which is a structure field of an empty class type, regardless of the number of its array elements:

***Notation* 5.5.5.** *The set of* empty fields *of C is the set of those fields defined in class C that are array structures of type B for some empty class B:*

$$E\mathcal{F}(C) \underset{\text{def.}}{=\!=} \{(\texttt{struct}\ B[n]\ f) \in \mathcal{F}(C) \mid B \textit{ is empty}\}$$

The set of non-empty fields of C is:

$$NE\mathcal{F}(C) \underset{\text{def.}}{=\!=} Sc\mathcal{F}(C) \cup St\mathcal{F}(C) \backslash E\mathcal{F}(C)$$

Then, we define the notion of the *data size* of a field. For a scalar field, the data size of a field is equal to its size. But for a structure array field, there are two cases:

- if the field has only one structure element of some type B, then the data size of such a field is equal to the data size of the class B.
- otherwise, it is important to note that the data of a field is considered to be a *contiguous interval.* So, any padding between two array elements is lost. However, nothing prevents us from reusing the padding of the last element of the array.

For this reason, we define the *data size* of a non-empty field as follows.

***Definition* 5.5.6.** *The* data size *of a scalar field* `scalar` *t fname is:*

$$\mathsf{fdsize}(\texttt{scalar}\ t\ \mathit{fname}) \underset{\text{def.}}{=\!=} \mathsf{scsize}(t)$$

The data size *of a non-empty structure array field* `struct` *$B[n]$fname is:*

$$\mathsf{fdsize}(\texttt{struct}\ B[n]\ \mathit{fname}) \underset{\text{def.}}{=\!=}\ (n-1)\cdot \mathsf{size}_B + \mathsf{dsize}_B$$

Then, in the same way as for the whole sizes, we introduce the notions of *non-virtual data size* (for the non-virtual part of a class) and *data size* (for the whole class), as bounds on the sizes of the class components.

***Notation* 5.5.7.** *In the same way as for fields, we shall use the following notations:*

- *$NE\mathcal{DNV}(C)$ for the set of non-empty direct non-virtual bases of C*
- *$E\mathcal{DNV}(C)$ for the set of empty direct non-virtual bases of C*
- *$NE\mathcal{V}(C)$ for the set of non-empty virtual bases of C*
- *$E\mathcal{V}(C)$ for the set of empty virtual bases of C*
- *$NE\tilde{\mathcal{V}}(C)$ for the set of non-empty generalized virtual bases of C*
- *$E\tilde{\mathcal{V}}(C)$ for the set of empty generalized virtual bases of C*

Then, the non-virtual data of a class C is divided into two parts, each being a *contiguous interval*:

- the non-virtual data of the non-empty direct non-virtual bases of C
- the data of the fields defined in C

The following hypothesis ensures that those two parts are disjoint:

Hypothesis 5.5.8. *The boundary* $\mathsf{fboundary}_C \in \mathbb{N}$ *between the non-virtual data for the direct non-virtual bases of C and the* non-empty *fields of C is such that:*

$$\forall B \in NE\mathcal{DNV}(C) : \mathsf{dnvboff}_C(B) + \mathsf{nvdsize}_B \leq \mathsf{fboundary}_C \qquad \text{(fboundary-lower-bound)}$$

$$\forall F \in NE\mathcal{F}(C) : \mathsf{fboundary}_C \leq \mathsf{foff}_C(F) \qquad \text{(foff-low-bound)}$$

In our formalization, such a boundary is included in the definition of a class layout (i.e. $\mathsf{fboundary}_C$ is a value explicitly computed by the layout algorithm).

Hypothesis 5.5.9. *The* non-virtual data size *of class C is an upper bound of the data sizes of the two parts of the non-virtual data of C (in particular, it is an upper bound of the data sizes of all non-empty fields).*

$$\mathsf{fboundary}_C \leq \mathsf{nvdsize}_C \qquad \text{(fboundary-upper-bound-bases)}$$

$$\forall F \in NE\mathcal{F}(C) : \mathsf{foff}_C(F) + \mathsf{fdsize}(F) \leq \mathsf{nvdsize}_C \qquad \text{(fboundary-upper-bound-fields)}$$

(fboundary-upper-bound-bases, p. 111) is necessary if there are no non-empty fields defined in C. Otherwise, (fboundary-upper-bound-bases, p. 111) is entailed by (fboundary-upper-bound-fields, p. 111).

Lemma 5.5.9. *If l is a non-virtual path from C to some* non-empty *class A, then the non-virtual part of A is included in the non-virtual part of C:*

$$\mathsf{nvsoff}(l) + \mathsf{nvdsize}_A \le \mathsf{nvdsize}_C$$

Proof. Same shape as the proof for Lemma 5.5.1 (p. 106), additionally using (fboundary-upper-bound-bases, p. 111). Additionally to A, all considered classes are not empty, as they have a non-empty base A. □

Hypothesis 5.5.10. *The total data size of C is an upper bound on the non-virtual data sizes of all its non-empty generalized virtual bases (including C itself, if non-empty):*

$$\forall B \in NE\tilde{\mathcal{V}}(C) : \mathsf{vboff}_C(B) + \mathsf{nvdsize}_B \le \mathsf{dsize}_C \qquad (\mathsf{datasize})$$

Lemma 5.5.10. *If (h, l) is an inheritance path (= base class subobject) from C to some class A, then the non-virtual data of A is included in C:*

$$\mathsf{soff}_C(h, l) + \mathsf{nvdsize}_A \le \mathsf{dsize}_C$$

Proof. Same shape as the proof for Lemma 5.5.2 (p. 107). □

Recall that the *data* of a structure array of n cells of type C consisting of a *contiguous interval* embedding the total sizes of the first $n-1$ cells and only the data of the last cell:

$$(n-1) \cdot \mathsf{size}_C + \mathsf{dsize}_C$$

Theorem I.2 (Non-virtual data subobject inclusion). *If $p = (l, i, \sigma)$ is a relative pointer (= generalized subobject) from $C[n]$ to some non-empty class A, then the non-virtual data of A is included in the data of $C[n]$:*

$$\mathsf{off}_C(p) + \mathsf{nvdsize}_A \le (n-1) \cdot \mathsf{size}_C + \mathsf{dsize}_C$$

Proof. By structural induction on l.
- If $l = \epsilon$, then $i \le (n-1)$ and $\mathsf{off}_C(p) = i \cdot \mathsf{size}_C + \mathsf{soff}_C(\sigma)$; moreover, σ is an inheritance path from C to A, so by Lemma 5.5.10 (p. 112), $\mathsf{soff}_C(\sigma) + \mathsf{nvdsize}_A \le \mathsf{dsize}_C$, which concludes.
- Otherwise, $l = (i', (h', p'), f') :: l'$ where $0 \le i' < n$, (h', p') is an inheritance path from C to some class B, and f' is a structure array of some type $C'[n']$ defined in B, such that (l', i, σ) is a relative pointer from $C'[n']$ to A, and $\mathsf{off}_C(p) = i \cdot \mathsf{size}_C + \mathsf{soff}_C(h, p) + \mathsf{foff}_B(f') + \mathsf{off}_{C'}(l', i, \sigma)$, and :
  - by induction hypothesis, $\mathsf{off}_{C'}(l', i, \sigma) + \mathsf{nvdsize}_A \le (n'-1) \cdot \mathsf{size}_{C'} + \mathsf{dsize}_{C'} = \mathsf{fdsize}(f')$ by definition
  - $\mathsf{foff}_B(f) + \mathsf{fdsize}(f) \le \mathsf{nvdsize}_B$ by (fboundary-upper-bound-bases, p. 111) (legal because, as A is not empty, neither is B)
  - $\mathsf{soff}_C(h, p) + \mathsf{nvdsize}_B \le \mathsf{dsize}_C$ by Lemma 5.5.10 (p. 112)
  - $i' \le n - 1$ by hypothesis

  which concludes. □

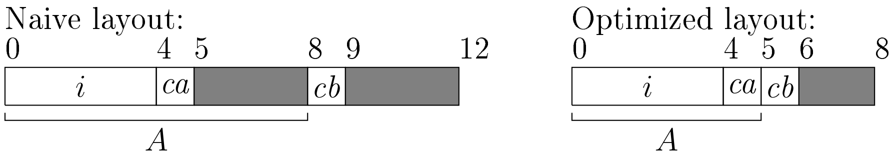
5.5.4 Non-overlapping of data

To express that two fields F_1 and F_2 defined in some class C should not overlap, we could expect their full size intervals to be disjoint:

$$[\mathsf{foff}_C(F_1), \mathsf{foff}_C(F_1) + \mathsf{fsize}(F_1)) \mathrel{\#} [\mathsf{foff}_C(F_2), \mathsf{foff}_C(F_2) + \mathsf{fsize}(F_2))$$

This condition is actually enforced by the C++ Common Vendor ABI [22, Section 4.1]. However, it impedes the reuse of tail padding, even alignment tail padding. For instance, consider the following structure:

```
struct Z {};
struct A: Z {
  int  i;
  char ca;
};
struct B {
  A a;
  char cb;
}
```



Most compilers (such as GCC) lay out `a` and `cb` completely disjointly within `B`, laying out `cb` at offset 8 (on Intel x86 32-bit platforms). This incurs the loss of unused space between the end of `a.ca` and the beginning of `cb`: 3 bytes for alignment padding.

Our formalization allows reusing this padding, by defining *data sizes* distinct from sizes, trying to exclude padding as most as possible. Then, we introduce constraints to ensure that *data* intervals be disjoint.

In our example, whereas the size of `a` is 8, its data size would be only 5 thanks to our formalization, such that `cb` would be laid out at offset 5 instead of 8.

Thus, we have to use the notion of *data size* rather than size, to express that two fields of the same class do not overlap:

Hypothesis 5.5.11. *The data of two non-empty fields defined in the same class do not overlap:*

$$\begin{aligned} &\forall F_1, F_2 \in NE\mathcal{F}(C): \\ &\quad [\mathsf{foff}_C(F_1), \mathsf{foff}_C(F_1) + \mathsf{fdsize}(F_1)) \\ \#&\quad [\mathsf{foff}_C(F_2), \mathsf{foff}_C(F_2) + \mathsf{fdsize}(F_2)) \end{aligned} \qquad (\mathsf{fields\text{-}non\text{-}overlap})$$

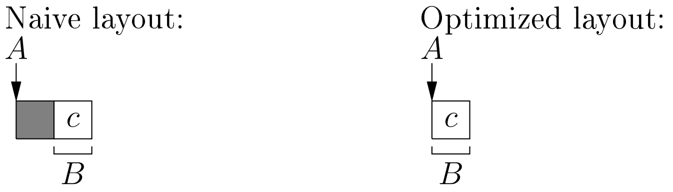
Similarly, the non-virtual data of two direct non-virtual bases of C do not overlap:

Hypothesis 5.5.12. $\forall B_1, B_2 \in \mathcal{DNV}(C)$:

$$\begin{aligned} &[\mathsf{dnvboff}_C(B_1), \mathsf{dnvboff}_C(B_1) + \mathsf{nvdsize}_{B_1}) \\ \#\ &[\mathsf{dnvboff}_C(B_2), \mathsf{dnvboff}_C(B_2) + \mathsf{nvdsize}_{B_2}) \end{aligned} \qquad (\mathsf{nvbases\text{-}non\text{-}overlap})$$

However, non-virtual sizes may overlap. Consider for instance the following hierarchy:

```
struct A       {};
struct B       {char c};
struct C: A, B {};
```



Then, the following layout (actually given by GNU GCC) complies with the above conditions:

$$\begin{aligned}
\mathsf{nvdsize}_A &= 0\\
\mathsf{nvsize}_A &= 1\\
\mathsf{foff}_B(\texttt{char}\ c) &= 0\\
\mathsf{nvdsize}_B &= 1\\
\mathsf{nvsize}_B &= 1\\
\mathsf{dnvboff}_C(A) &= 0\\
\mathsf{dnvboff}_C(B) &= 0\\
\mathsf{nvdsize}_C &= 1\\
\mathsf{nvsize}_C &= 1
\end{aligned}$$

Here, the non-virtual sizes of `A` and `B` overlap in `C`, but this does not impede the access to field `c` of `B`, as `A` is empty.

The actual purpose of distinguishing sizes from data sizes is that a pointer to an empty class can point outside of the actual data (accessible non-empty fields), but must still point inside the whole size of the object (so as to prevent it from pointing into another object that would be created independently). However, we shall see further in Section 5.5.6 (p. 122) weaker conditions to prevent two pointers to different subobjects of static type A (where A is an empty class) from pointing to the same memory location.

Lemma 5.5.11. *If l_1, l_2 are two distinct non-virtual paths from some class C to some non-empty classes B_1, B_2, then their field data zones are disjoint:*

$$\begin{array}{rl}
& [\mathsf{nvsoff}(l_1) + \mathsf{fboundary}_{B_1}, \mathsf{nvsoff}(l_1) + \mathsf{nvdsize}_{B_1})\\
\# & [\mathsf{nvsoff}(l_2) + \mathsf{fboundary}_{B_2}, \mathsf{nvsoff}(l_2) + \mathsf{nvdsize}_{B_2})
\end{array}$$

Proof. For symmetry reasons, we can assume $\mathsf{length}(l_1) \leq \mathsf{length}(l_2)$. Then, there are two cases:

- either there is a non-virtual non-trivial path $B_1 :: A :: l'$ from B_1 to B_2 such that $l_2 = l_1 @_{\mathsf{Repeated}} (B_1 :: A :: l')$. So A is a non-virtual direct base of B_1, and actually the non-virtual data of A (which includes the non-virtual data of B_2, in particular field F_2) is disjoint from the field data of B_1 (in particular field F_1) thanks to the boundary $\mathsf{fboundary}_{B_1}$.
- or there are a class A and two distinct non-virtual direct bases A_1 and A_2 of A, and a list l such that for each $i \in \{1, 2\}$, $l +\!\!+ A :: A_i :: \epsilon$ is a non-virtual path from C to A_i and $A_i :: l'_i$ is a non-virtual path from A_i to B_i for some l'_i. The non-virtual direct bases A_1 and A_2 of A have disjoint non-virtual data. The non-virtual data of A_i includes the non-virtual data of B_i, and in particular field F_i, so those fields are disjoint. □

Recall the convention $\mathsf{vboff}_C(C) \underset{\text{def.}}{=\!=} 0$. This allows to say that the non-virtual data of all non-empty generalized virtual bases of C (with C considered as a generalized virtual base of itself) are laid out one after another, in such a way that they are disjoint:

Hypothesis 5.5.13. $\forall B_1, B_2 \in NE\tilde{\mathcal{V}}(C)$: *if* B_1, B_2:

$$\begin{array}{rl}
& [\mathsf{vboff}_C(B_1), \mathsf{vboff}_C(B_1) + \mathsf{nvdsize}_{B_1})\\
\# & [\mathsf{vboff}_C(B_2), \mathsf{vboff}_C(B_2) + \mathsf{nvdsize}_{B_2})
\end{array} \qquad (\mathsf{vbases\text{-}non\text{-}overlap})$$

It follows immediately that:

Lemma 5.5.12. *If σ_1, σ_2 are distinct inheritance paths from C to some B_1, B_2, then their field data zones are disjoint:*

$$\begin{array}{rl} & \left[\mathsf{soff}_C(\sigma_1) + \mathsf{fboundary}_{B_1}, \mathsf{soff}_C(\sigma_1) + \mathsf{nvdsize}_{B_1}\right) \\ \# & \left[\mathsf{soff}_C(\sigma_2) + \mathsf{fboundary}_{B_2}, \mathsf{soff}_C(\sigma_2) + \mathsf{nvdsize}_{B_2}\right) \end{array}$$

Proof. Let $\sigma_i = (h_i, A_i :: l_i)$ for each i. If $A_1 = A_2$, then Lemma 5.5.11 (p. 114) about non-virtual paths can be reused. Otherwise, we know by (vbases-non-overlap, p. 114) that the non-virtual data of A_i are disjoint, then Lemma 5.5.9 (p. 112) and (fboundary-upper-bound-bases, p. 111) conclude. □

The conditions given so far are enough to show that two different scalar fields reachable from two base class subobjects of an object are disjoint. However, when it comes to traversing structure array fields, we must show that two fields reachable from two different cells of the same array are disjoint. So far there is no condition ensuring such a property. Indeed, there is yet no link between the data size and the size of a class: we have to explicitly constrain that:

Hypothesis 5.5.14. *For any class C:*

$$\mathsf{dsize}_C \leq \mathsf{size}_C \qquad \text{(datasize-upper-bound)}$$

Theorem I.3 (Non-overlapping of scalar fields). *If p_i are two generalized subobjects of static type B_i within a structure array of type C, and if F_i are two **scalar** fields defined in class B_i, such that $(p_1, F_1) \neq (p_2, F_2)$, then those two fields are disjoint.*

Proof. We have to reason about the *length* of the generalized subobjects:

***Notation* 5.5.8.** *The* length *of a pointer $p = (\alpha, j, \sigma)$, written* $\mathsf{plength}(p)$*, is the length of its array path α:*

$$\mathsf{plength}(p) \underset{\text{def.}}{=\!=} \mathsf{length}(\alpha)$$

For symmetry reasons, we can assume $\mathsf{plength}(p_1) \leq \mathsf{plength}(p_2)$. Reason by induction on $\mathsf{plength}(p_1)$. Then, we shall introduce an alternate representation for p_i:

Definition* 5.5.9 (Alternate representation of generalized subobjects).** ***(= alternate relative pointers) *For any generalized subobject (= relative pointer) $p = (\alpha, j, \sigma)$ from a structure array of type C to some class B, we can find $j' \in \mathbb{Z}$, an inheritance path σ' from C to some class B', and a Φ such that:*

- *either $\Phi = \bot$ and $p = (\epsilon, j', \sigma')$*
- *or there is a structure array field F' and a relative pointer p' = for some array path α' such that $\Phi = (F', (\alpha', j, \sigma))$ and $\alpha = (j', \sigma', F') :: \alpha'$ (so that $\mathsf{plength}(\alpha', j, \sigma) = \mathsf{plength}(p) - 1$ to allow induction).*

(j', σ', Φ) is called the alternate representation *of p, and we write:*

$$(\alpha, j, \sigma) \propto (j', \sigma', \Phi)$$

Lemma 5.5.13. *The alternate representation of a relative pointer p is unique:*

$$p \propto \mathfrak{p}_1 \wedge p \propto \mathfrak{p}_2 \;\Rightarrow\; \mathfrak{p}_1 = \mathfrak{p}_2$$

This alternate representation is intended for proofs about concrete object layout, by contrast to the "regular" *Definition* 4.1.9 (p. 77) which is designed for the abstract high-level semantics of C++ (especially to model casts and other dynamic operations such as virtual method dispatch). This alternate representation is trivially compatible with offsets:

Lemma 5.5.14. *Let $p \propto (j', \sigma', \Phi)$ be a relative pointer from C of static type B.*
- *if $\Phi = \bot$, then:*
$$\mathsf{off}_C(p) = j' \cdot \mathsf{size}_C + \mathsf{soff}_C(\sigma')$$
- *otherwise, σ' is an inheritance path from C to some class A and $\Phi = (F', p')$ for some structure field F' of type B' defined in A and for some relative pointer p' from B of static type B, and:*
$$\mathsf{off}_C(p) = j' \cdot \mathsf{size}_C + \mathsf{soff}_C(\sigma') + \mathsf{foff}_A(F') + \mathsf{off}_{B'}(p')$$

In particular, the following interesting lemma holds:

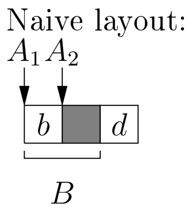
Lemma 5.5.15. *If $p \propto (j', \sigma', \Phi)$ is a relative pointer from C of static type B, such that σ' is an inheritance path from C to some class C', then the field data of p is included in the field data of C':*

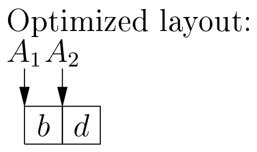
$$\begin{array}{rl} & [\mathsf{off}_C(p) + \mathsf{fboundary}_B, \mathsf{off}_C(p) + \mathsf{nvdsize}_B) \\ \subseteq & [j' \cdot \mathsf{size}_C + \mathsf{soff}_C(\sigma') + \mathsf{fboundary}_{C'}, j' \cdot \mathsf{size}_C + \mathsf{soff}_C(\sigma') + \mathsf{nvdsize}_{C'}) \end{array}$$

Then, for each i, let $p_i \propto (j'_i, \sigma'_i, \Phi_i)$. There are several cases:
- if $j'_1 \neq j'_2$, then we have two disjoint cells of an array of structures of some type C. Thanks to (datasize-upper-bound, p. 115), their data are also disjoint, which concludes.
- $j'_1 = j'_2$ and $\sigma'_1 \neq \sigma'_2$: then, thanks to the above lemma, the theorem for virtual inheritance directly applies.
- Now assume $j'_1 = j'_2$, $\sigma'_1 = \sigma'_2$.
  - If $\Phi_1 = \Phi_2$, then $F_1 \neq F_2$ are distinct fields of the same subobject, so by (fields-non-overlap, p. 113) they are disjoint.
  - Otherwise, we can assume $\Phi_2 = (F'_2, p'_2)$ (because $\mathsf{plength}(p_1) \leq \mathsf{plength}(p_2)$, so that if $\Phi_2 = \bot$, then $\Phi_1 = \bot$).
    - if $\Phi_1 = \bot$, then F_1 and F'_2 are distinct fields (because F_1 is scalar whereas F'_2 is a structure field) of the same class, so their data are disjoint.
    - Otherwise, $\Phi_1 = (F'_1, p'_1)$.
      - if $F'_1 \neq F'_2$, then they are two distinct fields of the same class, so their data are disjoint.
      - Otherwise, we may use the induction hypothesis. □

Our layout constraints allow to produce smarter layouts than GNU GCC. Indeed, fields are laid out not by their whole sizes, but only by their data sizes. This allows fields to be stored within the end of a structure array field. Consider for instance:

```
struct Z          {};
struct A1: Z      {};
struct A2: Z      {};
struct B:  A1, A2 { char b; };
struct D          { B  pb; char d; };
```





Assuming `sizeof`(`char`) $= 1$, GNU GCC gives `sizeof`$(B) = 2$, as two different offsets to subobjects of type Z have to be allotted.

But GNU GCC would give `sizeof`$(D) = 3$ by laying out j beyond the size of field c, so at offset 2, whereas our formalization allows `d` to be laid out at offset 1, that is just after the end of field `b` of `pb`, not waiting for the actual end of field `pb`, thus yielding `sizeof`$(D) = 2$. Such an optimization was proposed by [63].

In other words, our formalization generalizes empty base offsets to empty fields, thanks to the fact that the data size of a field of type $C[n]$ is $(n-1) \cdot \mathsf{size}_C + \mathsf{dsize}_C$ instead of $n \cdot \mathsf{size}_C$ as prescribed by the C++ Common Vendor ABI [22].

5.5.5 Dynamic type data

When a class has a polymorphic behaviour (e.g. virtual bases, or virtual functions), it is dynamic. Then, additional dynamic type data is needed for subobjects of such class types.

If C is dynamic, then there are two cases:

- if C has a direct non-virtual base A that is dynamic, then this base may be chosen as a *primary base* that will have offset 0 within the non-virtual part of C.
- otherwise, some space must be explicitly allocated at the beginning of C to store the dynamic type data.

This ensures that any dynamic class C has some space at the beginning of C to store the dynamic type data. If C has a primary base A, then C and A will share their dynamic type data.

Hypothesis 5.5.15. *The size of dynamic type data is written* $\mathsf{dtdsize}$*. It is positive and it does not depend on any class.*

In practice, for compilers such as GNU GCC, $\mathsf{dtdsize} =$ `sizeof`(`void`$*$) to store a pointer to a virtual table. Other compilers like Microsoft Visual C++ used $\mathsf{dtdsize} = 2 \cdot$ `sizeof`(`void`$*$) to store additional data (a pointer to the table of virtual bases) [20].

The choice of the primary base is formalized as follows:

Hypothesis 5.5.16. *There exists* $\mathsf{pbase}_C \in \mathcal{DNV}(C)^?$ *such that if* $\mathsf{pbase}_C = A \neq \perp$*, then C and A are dynamic and:*

$$\mathsf{dnvboff}_C(A) = 0 \tag{pbase}$$

In this case, A is called the primary base *of C.*

A class C having a dynamic non-virtual direct base does not necessarily have a primary class: the layout algorithm is not forced to choose any. However, as regards performance, such an algorithm could yield under-optimized layouts, with C having its own dynamic type data disjoint from all of its bases'.

We have to prove non-overlapping lemmata for dynamic type data. Indeed, we must ensure that:

- whenever a field is written to, the dynamic type data of dynamic subobjects are not altered
- whenever a subobject is being initialized, the dynamic type data of other subobjects are not altered

Disjointness of fields and dynamic type data Writing data to a scalar field must not impact the polymorphic behaviour of its holding subobject. Thus, we have to prevent such operation from overwriting dynamic type data.

Hypothesis 5.5.17. *The dynamic type data of a* dynamic *class C is disjoint from the field data of C:*

$$\mathsf{dtdsize} \leq \mathsf{fboundary}_C \qquad (\mathsf{dtdatasize\text{-}upper\text{-}bound})$$

This hypothesis along with (fboundary-upper-bound-bases, p. 111) allows to show that the dynamic type data of a class is included in its non-virtual data.

The following hypothesis makes C's dynamic type data disjoint from any of its non-empty non-virtual direct bases if C has no primary base:

Hypothesis 5.5.18. *If $\mathsf{pbase}_C = \varnothing$, then for any* non-empty *direct non-virtual base B of a dynamic class C:*

$$\mathsf{dtdsize} \leq \mathsf{dnvboff}_C(B) \qquad (\mathsf{dtdatasize\text{-}pbase})$$

Lemma 5.5.16. *If l is a non-virtual path from some dynamic class C to some non-empty class B, then the dynamic type data of C is disjoint from the field data of B. More precisely:*

$$\mathsf{dtdsize} \leq \mathsf{nvsoff}(l) + \mathsf{fboundary}_B$$

Proof. By induction on l. If $l = C :: \epsilon$ is the trivial path, then $B = C$ and (dtdatasize-upper-bound, p. 118) concludes. Otherwise, if $l = C :: B' :: l'$, then B' is a non-virtual direct base of C. There are two cases:

- If B' is the primary base of C, having offset 0 within C, then B' is dynamic and we use the induction hypothesis.
- Otherwise, it suffices to show that B' itself starts at offset at least dtdsize within C, which would conclude. There are two cases:
  - If C has no primary base, then (dtdatasize-pbase, p. 118) concludes.
  - Otherwise, if P is the primary base of C, then the data of P and B' are disjoint. However, we need an additional hypothesis to conclude:

    Hypothesis 5.5.19. *Any non-empty class A has non-zero non-virtual data size:*

    $$0 < \mathsf{nvdsize}_A \qquad (\mathsf{nonempty\text{-}nvdatasize\text{-}positive})$$

    In practice, this hypothesis makes sense, but its proof depends on the layout algorithm and, in particular, the notions of empty class and dynamic class. □

Lemma 5.5.17. *If l is a non-virtual path from some class C to some dynamic class B, then the dynamic type data of B is disjoint from the field data of C. More precisely:*

$$\mathsf{nvsoff}(l) + \mathsf{dtdsize} \leq \mathsf{fboundary}_C$$

Proof. By case analysis on l. If $l = C :: \epsilon$ is the trivial path, then $B = C$ and (dtdatasize-upper-bound, p. 118) concludes. Otherwise, if $l = C :: B' :: l'$, then B' is a non-virtual direct base of C, and:

- $\mathsf{dtdsize} \leq \mathsf{fboundary}_B$ by (dtdatasize-upper-bound, p. 118)
- $\mathsf{fboundary}_B \leq \mathsf{nvdsize}_B$ by (fboundary-upper-bound-bases, p. 111)
- $\mathsf{nvsoff}(B' :: l) + \mathsf{nvdsize}_B \leq \mathsf{nvdsize}_{B'}$ by Lemma 5.5.9 (p. 112)
- $\mathsf{dnvboff}_C(B') + \mathsf{nvdsize}_{B'} \leq \mathsf{fboundary}_C$ by (fboundary-lower-bound, p. 111)

which concludes. □

Lemma 5.5.18. *If l_1 is a non-virtual path from C to some dynamic class B_1, and if l_2 is a non-virtual path from C to some non-empty class B_2, then the field data of B_2 is disjoint from the dynamic data of B_1:*

$$\begin{array}{rl} & [\mathsf{nvsoff}(l_1), \mathsf{nvsoff}(l_1) + \mathsf{dtdsize}) \\ \# & \left[\mathsf{nvsoff}(l_2) + \mathsf{fboundary}_{B_2}(F), \mathsf{nvsoff}(l_2) + \mathsf{nvdsize}_{B_2}\right) \end{array}$$

Proof. There are three cases:

- if there is a non-virtual path p from B_1 to B_2 such that $l_2 = l_1 @_{\mathsf{Repeated}} p$, then Lemma 5.5.16 (p. 118) concludes.
- if there is a non-virtual path p from B_2 to B_1 such that $l_1 = l_2 @_{\mathsf{Repeated}} p$, then Lemma 5.5.17 (p. 118) concludes.
- Otherwise, there is a class A and a non-virtual path p from C to A and $A_1 \neq A_2 \in \mathcal{DNV}_A$ and non-virtual paths l'_1 from A_1 to B_1 and l'_2 from A_2 to B_2 such that $\forall i : l_i = p @_{\mathsf{Repeated}} (A :: A_i :: \epsilon) @_{\mathsf{Repeated}} l'_i$. As A_1 and A_2 are two distinct direct non-virtual non-empty bases of A, their data zones are disjoint, which concludes. □

Corollary 5.5.19. *If σ_1 is an inheritance path from C to some dynamic class B_1, and if σ_2 is an inheritance path from C to some non-empty class B_2, then the field data of B_2 is disjoint from the dynamic data of B_1:*

$$\begin{array}{rl} & [\mathsf{soff}_C(\sigma_1), \mathsf{soff}_C(\sigma_1) + \mathsf{dtdsize}) \\ \# & \left[\mathsf{soff}_C(\sigma_2) + \mathsf{fboundary}_{B_2}(F), \mathsf{soff}_C(\sigma_2) + \mathsf{nvdsize}_{B_2}\right) \end{array}$$

Proof. Let $\sigma_i = (h_i, B'_i :: l_i)$ for each i. Then, there are two cases:

- if $B'_1 = B'_2$, then the previous lemma immediately concludes.
- Otherwise, B'_1 and B'_2 are two distinct non-empty generalized virtual bases of C, so by (vbases-non-overlap, p. 114) their data are disjoint, which concludes thanks to Lemma 5.5.10 (p. 112) □

Theorem I.4 (Dynamic type data are disjoint from scalar fields). *If p_1, p_2 are generalized subobjects of static type B_1, B_2 within an array of C, such that B_1 defines a scalar field F and B_2 is dynamic, then F is disjoint from the dynamic type data of B_1:*

$$\begin{array}{rl} & [\mathsf{off}_C(p_1) + \mathsf{foff}_{B_1}(F), \mathsf{off}_C(p_1) + \mathsf{foff}_{B_1}(F) + \mathsf{fdsize}(F)) \\ \# & [\mathsf{off}_C(p_2), \mathsf{off}_C(p_2) + \mathsf{dtdsize}) \end{array}$$

Proof. By induction on $\mathsf{plength}(p_1)$. For each i, let $p_i \propto (j_i, \sigma_i, \Phi_i)$. Then:

- If $j_1 \neq j_2$, then we have two different cells of the array of C, so thanks to (datasize-upper-bound, p. 115) and THEOREM I.2 (p. 112), their data are disjoint.

Otherwise, let B'_i be the static type of σ_i. By LEMMA 5.5.15 (p. 116), F is included in the field data of B'_1.

- If $\Phi_2 = \bot$, then COROLLARY 5.5.19 (p. 119) concludes.

Otherwise, let $\Phi_2 = (F'_2, p'_2)$. Then, the dynamic type data of B_2 is included in the data zone of the structure field F'_2 by THEOREM I.2 (p. 112), which is itself in the field data zone of B'_2.

- If $\sigma_1 \neq \sigma_2$, then, by LEMMA 5.5.12 (p. 115), the two field data zones of B'_1 and B'_2 are disjoint, which concludes.

Otherwise, $\sigma_1 = \sigma_2$ and $B'_1 = B'_2$.

- If $\Phi_1 = \bot$, then F and F'_2 are distinct fields (because F is scalar whereas F'_2 is a structure field) of the same subobject, so by (fields-non-overlap, p. 113) their data are disjoint.

Otherwise, let $\Phi_1 = (F'_1, p'_1)$ with $\mathsf{plength}(p'_1) = \mathsf{plength}(p_1) - 1$ to allow induction.

- If $F'_1 \neq F'_2$, then they are distinct fields of the same subobject, so their data are disjoint.
- Otherwise, we may use the induction hypothesis. □

Disjointness of two dynamic type data The non-virtual data of two different non-virtual bases of C are not necessarily disjoint, because such classes can share their dynamic type data with their primary bases. As this phenomenon can propagate, it is necessary to precisely determine which subobjects will share their dynamic type data.

***Definition* 5.5.10.** *A non-virtual path l from some class B to some class A is* primary *(written* $\mathsf{isPrimaryPath}(l)$*) if, and only if, at least one of the following conditions holds:*

- *either $B = A$ and $l = B :: \epsilon$*
- *or B has a primary base B' and $l = B :: B' :: l'$ for some l', such that $B' :: l'$ is a primary path from B' to A.*

$$\frac{}{\mathsf{isPrimaryPath}(B :: \epsilon)} \qquad \frac{\mathsf{pbase}(B) = A \qquad \mathsf{isPrimaryPath}(A :: l)}{\mathsf{isPrimaryPath}(B :: A :: l)}$$

LEMMA 5.5.20. *For any non-virtual, non-primary path l from some class B to some class A, there exists a unique path written* $\mathsf{reducePath}(l)$ *and called the* reduced path *from l, such that there exists l' the longest primary path such that $l = \mathsf{reducePath}(l) @_{\mathsf{Repeated}} l'$. By convention, if l is primary, then $\mathsf{reducePath}(l) \underset{\text{def.}}{=\!=} B :: \epsilon$.*

LEMMA 5.5.21. *If l' is a primary path, then $\mathsf{reducePath}(l @_{\mathsf{Repeated}} l') = \mathsf{reducePath}(l)$.*

LEMMA 5.5.22. *For any non-virtual path l, $\mathsf{nvsoff}(l) = \mathsf{nvsoff}(\mathsf{reducePath}(l))$.*

In particular, any non-virtual path l to a dynamic class shares its dynamic type data with its reduced path $\mathsf{reducePath}(l)$.

Proof. It suffices to show that for any non-virtual paths l and l', if l' is primary, then $\mathsf{nvsoff}(l @_{\mathsf{Repeated}} l') = \mathsf{nvsoff}(l)$. By induction on l'. □

Conversely:

LEMMA 5.5.23. *If two* non-primary *non-virtual paths l_1, l_2 from some class B to some dynamic classes A_1, A_2 such that $\mathsf{reducePath}(l_1) \neq \mathsf{reducePath}(l_2)$, then their dynamic type data are disjoint.*

Proof. Pose $l'_i = \mathsf{reducePath}(l_i)$ for each i. Then, as l_1 and l_2 are not primary, their reduced paths l'_1, l'_2 are not trivial. By symmetry, we may assume $\mathsf{length}(l'_1) \leq \mathsf{length}(l'_2)$. Then, there are two cases:

- If $l'_2 = l'_1 @_{\mathsf{Repeated}} l$, then, as $l'_2 \neq l'_1$, l is not trivial. So, $l'_1 = l'' +\!\!+ A'_1 :: A_1 :: \epsilon$ and there are two cases:
  - either $l = A_1 :: A_2 :: \epsilon$, so that A_2 is a non-primary direct non-virtual base of A_1, which concludes
  - either $l = A_1 :: l''' :: A' :: A_2 :: \epsilon$ for some non-virtual base A' of A_1 such that A_2 is a non-primary direct non-virtual base of A'. Then, A_2 starts at offset at least $\mathsf{dtdsize}$ in A', which is enough to conclude. □

Corollary 5.5.24. *If two non-virtual paths l_1, l_2 from some class B to some dynamic classes A_1, A_2 such that* $\mathsf{reducePath}(l_1) \neq \mathsf{reducePath}(l_2)$*, then their dynamic type data are disjoint.*

Proof.
- If both paths were primary, then they would have the same reduced path, which is absurd.
- If (for instance) l_1 is primary and l_2 is non-primary, then A_2 starts at offset at least $\mathsf{dtdsize}$ in B, which concludes.
- Otherwise, Lemma 5.5.23 (p. 120) concludes. □

Corollary 5.5.25. *If two paths $(h_1, l_1), (h_2, l_2)$ from some class C to some dynamic classes A_1, A_2 such that* $\mathsf{reducePath}(l_1) \neq \mathsf{reducePath}(l_2)$*, then their dynamic type data are disjoint.*

Theorem I.5 (Non-overlapping of dynamic type data). *If $p_1 = (\alpha_1, j_1, (h_1, l_1))$, $p_2 = (\alpha_2, j_2, (h_2, l_2))$ are two generalized subobjects from $C[n]$ to some dynamic classes A_1, A_2 such that:*

$$(\alpha_1, j_1, (h_1, \mathsf{reducePath}(l_1))) \neq (\alpha_2, j_2, (h_2, \mathsf{reducePath}(l_2)))$$

Then their dynamic type data are disjoint.

Proof. As there is a condition on $(h_1, l_1), (h_2, l_2)$, we cannot use the alternate representation for pointers. So a direct reasoning is necessary.

- If $(\alpha_1, j_1) = (\alpha_2, j_2)$, then Corollary 5.5.25 (p. 121) concludes.
- Otherwise, if $\alpha_1 = \alpha_2$ and $j_1 \neq j_2$, then there are two different cells of the same array, so their data are disjoint.

Otherwise, assume $\alpha_1 \neq \alpha_2$. By symmetry, we may assume $\mathsf{length}(\alpha_1) \leq \mathsf{length}(\alpha_2)$. Then, we have $\alpha_2 = (j'_2, P'_2, F'_2) :: \alpha'_2$. Reason by structural induction on α_1. There are several cases:

- If $\alpha_1 = \epsilon$, then the dynamic type data targeted by p_2 is included in the data of F'_2 which is included in the field data of the subobject p_1. Then, Theorem I.4 (p. 119) concludes.
- Otherwise, $\alpha_1 = (j'_1, \sigma'_1, F'_1) :: \alpha'_1$. If $(j'_1, \sigma'_1) \neq (j'_2, \sigma'_2)$, then our generalized subobjects p_1 and p_2 are located in disjoint field zones, which concludes. Otherwise, if $F'_1 \neq F'_2$, then the data of those two fields are disjoint, which concludes. Otherwise, we may use the induction hypothesis. □

Dynamic type data alignment As with fields, access to dynamic type data must be correctly aligned.

Hypothesis 5.5.20. *We assume that there exists an alignment* $\mathsf{dtdalign} > 0$ *for dynamic type data.*

Then, the following theorem:

THEOREM I.6 (Dynamic type data alignment). *If p is a generalized subobject of static type A from a structure array of C, such that A is dynamic, then the access to the dynamic type data of A is correctly aligned:*

$$(\mathsf{dtdalign} \mid \mathsf{off}_C(p))$$

$$(\mathsf{dtdalign} \mid \mathsf{align}_C)$$

is a direct consequence of COROLLARY 5.5.8 (p. 110) requiring the following hypothesis:

Hypothesis 5.5.21. *If A is a dynamic class, then:*

$$(\mathsf{dtdalign} \mid \mathsf{nvalign}_A) \qquad (\mathsf{dtdalign\text{-}nv})$$

5.5.6 Identity of subobjects

In this section, we want to show that if $p_1 \neq p_2$ are two distinct generalized subobjects of the same static type A from the same structure array of C, then they are located at distinct offsets $\mathsf{off}_C(p_1) \neq \mathsf{off}_C(p_2)$ within the structure array.

The proof of this theorem is very different depending on whether A is empty or not.

5.5.6.1 Non-empty base offsets

If A is not empty, then ($\mathsf{nonempty\text{-}nvdatasize\text{-}positive}$, p. 118) ensures that $\mathsf{nvdsize}_A \neq 0$.

However, recall that the non-virtual data of two distinct subobjects of the same static type are not necessarily disjoint, because of their dynamic type data.

LEMMA 5.5.26. *If l_1, l_2 are distinct non-virtual paths from some class B to some non-empty class A, then* $\mathsf{nvsoff}(l_1) \neq \mathsf{nvsoff}(l_2)$.

Proof. By symmetry, we may assume $\mathsf{length}(l_1) \leq \mathsf{length}(l_2)$. Then, there are two cases:

- Case $l_2 = l_1 @_{\mathsf{Repeated}} l$ is absurd. Indeed, as l_1 and l_2 have the same static type A, we would have l trivial, so $l_2 = l_1$.
- So, l_1 and l_2 point to subobjects of some classes $B'_1 \neq B'_2$ that are distinct direct non-virtual bases of some class B'. As the non-virtual data of B'_1, B'_2 are disjoint, and the non-virtual data of A is non-null, arithmetics conclude. □

COROLLARY 5.5.27. *If σ_1, σ_2 are distinct paths from some class B to some non-empty class A, then* $\mathsf{soff}_B(\sigma_1) \neq \mathsf{soff}_B(\sigma_2)$.

THEOREM I.7 (Non-empty subobject identity). *If p_1, p_2 are distinct generalized subobjects from some class C to some non-empty class A, then* $\mathsf{off}_C(p_1) \neq \mathsf{off}_C(p_2)$.

Proof. For each i, let $p_i \propto (j_i, \sigma_i, \Phi_i)$. By symmetry, we may assume $\mathsf{plength}(l_1) \leq \mathsf{plength}(l_2)$. Reason by induction on $\mathsf{plength}(p_1)$. Then, there are several cases:

- if $j_1 \neq j_2$, then we have two distinct cells of the same array, so their data are disjoint.
- Otherwise, if $j_1 = j_2$ and $\Phi_2 = (F_2, p'_2)$, then there are two cases:
  - if $\Phi_1 = (F_1, p'_1)$, then there are three cases:
    - if $\sigma_1 = \sigma_2$, then either $F_1 = F_2$, in which case we may use the induction hypothesis, or $F_1 \neq F_2$ so those are two distinct fields of the same subobject, so their data are disjoint.
    - otherwise, the designated subobjects are located in disjoint field zones.
  - Otherwise, $\Phi_1 = \perp$. Then, there are several cases:
    - Case $\sigma_2 = \sigma_1 @_{\mathsf{Repeated}} \sigma$ is absurd (the hierarchy is well-founded).
    - Case $\sigma_1 = \sigma_2 @_{\mathsf{Repeated}} (B_2 :: B' :: l)$: in that case, whereas p_2 is located in the field data zone of B_2, p_1 is located in the data zone of B' which is a direct non-virtual base of B_2: those two data zones are disjoint
    - Otherwise, $\sigma_1 = \sigma @_{\mathsf{Repeated}} B' :: B'_1 :: l_1$ and $\sigma_2 = \sigma @_{\mathsf{Repeated}} B' :: B'_2 :: l_2$ with B'_1 and B'_2 two distinct direct non-virtual bases of B', so their data are disjoint.
- Otherwise, $j_1 = j_2$, $\Phi_2 = \perp$ and $\Phi_1 = \perp$ (as $\mathsf{plength}(p_1) \leq \mathsf{plength}(p_2)$), so Corollary 5.5.27 (p. 122) concludes. □

5.5.6.2 Empty base offsets

Unfortunately, this kind of reasoning cannot be done if A is empty. In this case, we have to explicitly enumerate the offsets to all possible subobjects of an empty static type.

***Definition* 5.5.11.** *The sets* $\mathsf{nveboffs}_C$ *of the* non-virtual empty base offsets *of C and* eboffs_C *of* empty base offsets *of C are defined as follows:*

$$\begin{aligned}
\mathsf{nveboffs}_C \overset{=\!=}{\text{def.}} \quad & \begin{cases} \{(C, 0)\} & \text{if } C \text{ is empty} \\ \varnothing & \text{otherwise} \end{cases} \\
\cup \quad & \bigcup_{B \in \mathcal{NV}(C)} \mathsf{dnvboff}_C(B) + \mathsf{nveboffs}_B \\
\cup \quad & \bigcup_{(\mathtt{struct}\ B[n]\ f) \in \mathcal{F}(C)} \bigcup_{i \in [0 \ldots n)} \mathsf{foff}_C(\mathtt{struct}\ B[n]\ f) + i \cdot \mathsf{size}_B + \mathsf{eboffs}_B
\end{aligned}$$

$$\mathsf{eboffs}_C \overset{=\!=}{\text{def.}} \bigcup_{B \in \tilde{\mathcal{V}}(C)} \mathsf{nveboffs}_B$$

In our Coq development, those two sets are defined as mutually inductive predicates. Only at the level of the algorithms, we show that we can construct for any class C two sets such that (A, o) is in either set if and only if (A, o) verifies the corresponding predicate, assuming that such sets exist for any class $B \prec C$. Indeed, we shall see that those sets are computed at the same time as class layout.

Lemma 5.5.28. $(A, o) \in \mathsf{nveboffs}_C$ *if, and only if, A is empty and there is a class B and a non-virtual path l from C to B such that at least one of the following conditions holds:*

- *either $B = A$ and $o = \mathsf{nvsoff}(l)$*

– *or there is a structure array field f of some type $B'[m']$ defined in class B and an array path α from $B'[m']$ to some $A'[n']$, a nonnegative integer $k < n'$ and an inheritance path σ from A' to A such that:*

$$o = \mathsf{nvsoff}(l) + \mathsf{aoff}_{B'}(\alpha) \ + k \cdot \mathsf{size}_{A'} \ + \mathsf{soff}_{A'}(\sigma)$$

$(A, o) \in \mathsf{eboffs}_C$ *if, and only if, A is empty and there is a class B and an inheritance path σ from C to B such that at least one of the following conditions holds:*
– *either $B = A$ and $o = \mathsf{soff}_C(\sigma)$*
– *or there is a structure array field f of some type $B'[m']$ defined in class B and an array path α from $B'[m']$ to some $A'[n']$, a nonnegative integer $k < n'$ and an inheritance path σ' from A' to A such that:*

$$o = \mathsf{soff}_C(\sigma) + \mathsf{aoff}_{B'}(\alpha) \ + k \cdot \mathsf{size}_{A'} \ + \mathsf{soff}_{A'}(\sigma')$$

Proof. $\Rightarrow$: by mutual induction on the definitions of eboffs and $\mathsf{nveboffs}$.
$\Leftarrow$: cases $B = A$ by induction on p for non-virtual inheritance, by case analysis on h where $\sigma = (h, l)$ for virtual inheritance. Other cases by induction on α. $\square$

Theorem I.8 (Empty subobject identity). *Let A be an empty class. Two different pointers of static type A, but from the same initial object, point to a different memory location, if and only if, in the construction of the sets* eboffs *and* $\mathsf{nveboffs}$*, the unions are disjoint, i.e. for all classes C, the following conditions hold:*
– $\forall B_1, B_2 \in \mathcal{DNV}(C) : B_1 \neq B_2 \Rightarrow$

$$\mathsf{dnvboff}_C(B_1) + \mathsf{nveboffs}_{B_1} \ \# \ \mathsf{dnvboff}_C(B_2) + \mathsf{nveboffs}_{B_2} \qquad (\mathsf{nveboffs\text{-}disjoint})$$

– $\forall B_1 \in \mathcal{DNV}(C), \forall(\texttt{struct } B_2[n]\ f) \in \mathcal{F}(C), \forall i \in [0 \ldots (n-1)] :$

$$\mathsf{dnvboff}_C(B_1) + \mathsf{nveboffs}_{B_1} \ \# \ \mathsf{foff}_C(\texttt{struct } B_2[n]\ f) + \ i \cdot \mathsf{size}_{B_2} \ + \mathsf{eboffs}_{B_2} \qquad (\mathsf{nveboffs\text{-}eboffs\text{-}disjoint})$$

– $\forall(\texttt{struct } B_1[n_1]\ f_1), (\texttt{struct } B_2[n_2]\ f_2) \in \mathcal{F}(C), (\texttt{struct } B_1[n_1]\ f_1) \neq (\texttt{struct } B_2[n_2]\ f_2) \Rightarrow$
$\forall i_1 \in [0 \ldots (n_1 - 1)], \forall i_2 \in [0 \ldots (n_2 - 1)] :$

$$\mathsf{foff}_C(\texttt{struct } B_1[n_1]\ f_1) + i_1 \cdot \mathsf{size}_{B_1} + \mathsf{eboffs}_{B_1} \ \# \ \mathsf{foff}_C(\texttt{struct } B_2[n_2]\ f_2) + i_2 \cdot \mathsf{size}_{B_2} + \mathsf{eboffs}_{B_2} \qquad (\mathsf{eboffs\text{-}disjoint})$$

– $\forall B_1, B_2 \in \tilde{\mathcal{V}}(C) : B_1 \neq B_2 \Rightarrow$

$$\mathsf{vboff}_C(B_1) + \mathsf{nveboffs}_{B_1} \ \# \ \mathsf{vboff}_C(B_2) + \mathsf{nveboffs}_{B_2} \qquad (\mathsf{vboff\text{-}nveboffs\text{-}disjoint})$$

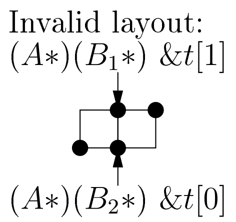
Proof. Mostly by definition of $\mathsf{nveboffs}$ and eboffs, except for two empty bases of distinct cells of a structure array. In that case, we have to use the fact that two different cells of the same array are totally disjoint (not only their data). Then, we need a further hypothesis:

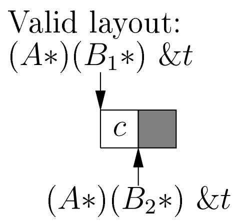
Hypothesis 5.5.22. *The non-virtual size of any class B is positive.*

$$\mathsf{nvsize}_C > 0 \qquad (\mathsf{nvsize\text{-}positive})$$

The reason why (nvsize-positive, p. 124) is required already for non-virtual sizes (not only for total sizes) may seem not obvious, but the following simple example makes it clear:

```
struct A          {};
struct B1: A      {};
struct B2: A      {};
struct C:  B1, B2 {};

C t[2];
A * a1 = (A*) (B2*) &t[0];
A * a2 = (A*) (B1*) &t[1];
```



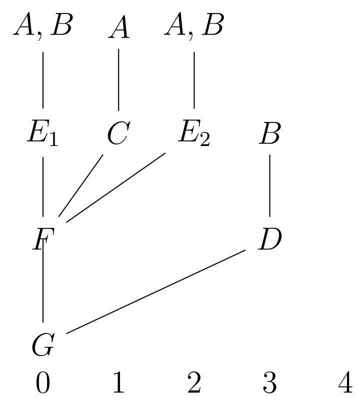


If $\mathsf{nvsize}_A = 0$, $\mathsf{nvsize}_{B_1} = 0$, and $\mathsf{nvsize}_{B_2} = 0$ were allowed, then we would not be able to distinguish pointers a_1 and a_2 above.

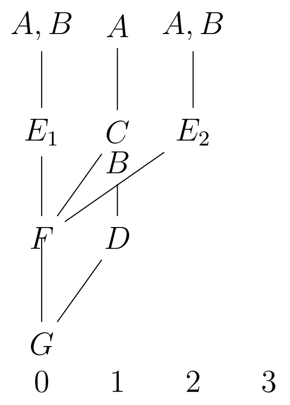
How empty base class offsets are checked In practice, following the behaviour prescribed by the C++ Common Vendor ABI [22] to compute offsets for a given class C, assuming that layout has already been computed for all bases of C and all classes that are types of some structure fields of C, components of C (base or field) are laid out one after another, and for each attempt to lay out a given component, it is checked against all components previously laid out.

Considering intervals instead of eboffs and nveboffs is not enough, even in practice. Indeed, consider the following hierarchy:

```
struct A              {};
struct B              {};
struct C:  A          {};
struct D:  B          {};
struct E1: A,  B      {};
struct E2: A,  B      {};
struct F:  E1, C, E2  {};
struct G:  F,  D      {};
```



A naive layout: laying out D within G relies on an interval-based checking, forbidding the B subobject of D to be laid out at offsets $[0 \ldots 2]$ within G.



The layout actually given by GNU GCC: laying out D within G relies on checking for the exact set of actually used offsets, allowing the B subobject of D to be laid out at offset 1 within G.

The existence of the type conflict test existence, but not the way it should work, is prescribed by the C++ Common Vendor ABI [22]). If such a test relies on interval checking, then base F of G would reserve the offset interval $[0 \ldots 2]$ for base B, so that base D of G would be laid out at offset 3. However, it would be consistent to lay out D at offset 1 within G, which GNU GCC actually does. This justifies why the exact sets of offsets have to be checked.

Chapter 6

Verification of realistic layout algorithms

In Chapter 5 (p. 93), we chose not to directly impose a layout, as CompCert does with C structures, but to set some constraints for layout algorithms, allowing them to make some optimizations that our constraints do not necessarily catch, such as field or base reordering for optimizing data alignment. The two algorithms considered in this chapter take advantage of this freedom, notably to optimize the layout of empty base classes.

The first algorithm we verify is based on the C++ Common Vendor ABI [22], a popular Application Binary Interface used by GCC and other C++ compilers (except notably Microsoft Visual C++). The second algorithm we consider extends the first algorithm with additional optimizations for empty base tail padding.

As the hierarchy is well-formed, we may prove, by well-founded induction on $\prec$ (the well-founded order on class names, cf. Section 4.1.4.2 p. 79), that those algorithms satisfy the soundness conditions of Chapter 5 (p. 93). We do not explain the technical details of those proofs, which are done in Coq; we only give here the high-level arguments used in those proofs.

6.1 An algorithm based on the C++ Common Vendor ABI

The principle of this algorithm based on the Common Vendor C++ ABI for Itanium [22] is that all bases and fields are laid out within a class such that they are mutually *totally* disjoint, not only their data. This actually incurs stronger constraints than the ones stated in Chapter 5 (p. 93). Consequently, for two such disjoint components (bases or fields), any two offsets to generalized subobjects of some class type A (empty or not) from those components are automatically distinct.

However, such a layout would be too naive for empty bases, as it would consume some space for them. For this reason, an important exception is prescribed for empty bases, which may be laid out at offset 0. In that case, explicit checks are necessary to ensure that a subobject reachable from an empty base will not conflict with another subobject of the same type reachable from another base or field.

However, to limit the amount of such explicit checks that must be performed during layout, this algorithm considers classes with fields to be non-empty. More formally:

***Definition* 6.1.1.** *A class C is* empty *if, and only if, all the following conditions hold:*
- *C has no virtual methods*
- *C has no direct virtual base*

- *C has no fields*
- *all direct non-virtual bases of C are empty*

In particular, a class having a virtual base is not empty. More generally:

Lemma 6.1.1. *Dynamic classes are not empty.*

Then, each class C is laid out through the following way, assuming that all classes B such that $B \prec C$ have been already laid out [1]:

1. If C has a dynamic direct non-virtual base B, choose B as the primary base of C, and lay out B within C at offset 0.
2. Otherwise, if C is dynamic, reserve some space for dynamic type data
3. For each direct non-virtual base B of C that is not the primary base of B, lay it out within C as follows:
   - If B is empty, try to lay it out at offset 0, unless there is a type conflict for empty bases.
   - Otherwise, try to lay out B at the current value of $\mathsf{nvdsize}_C$ so far. If there is a type conflict for empty bases, then increase by the non-virtual alignment $\mathsf{nvalign}_B$ of the base B, knowing that beyond the current value for nvsize_C, there will be no conflict (because all offsets of bases reachable from C so far are less than nvsize_C).
   - If B is not empty, update $\mathsf{nvdsize}_C$ to include the *whole* non-virtual size of the base (not only its non-virtual data size).
4. Then, for each field, lay it out at increasing offsets, making them *wholly* disjoint (not only their data).
5. Then, lay out all virtual bases of C the same way as for non-virtual bases (i.e. trying offset 0 for empty virtual bases, and making them wholly disjoint – not only their data).

During the layout of the components of C, it is also wise to construct the sets $\mathsf{nveboffs}_C$ and eboffs_C of generalized subobjects of an empty class type that are reachable from C.

However, the overall shape of the algorithm as described above and in the C++ Common Vendor ABI does not exactly state how type conflicts for empty bases should be resolved. In fact, there are only two possible conflicts:

- when trying to lay out an empty base B at offset 0, it is necessary to check that no empty base reachable through B conflict with any empty base reachable from all other non-virtual bases that have been already laid out.
- when trying to lay out a non-empty base B, or a field f, it is only necessary to check that there is no conflict with any empty base reachable from *non-virtual empty bases* laid out so far. Indeed, this algorithm lays out all non-empty components in such way that two distinct non-empty components are *totally* disjoint (not only their data).

We now formalize this layout algorithm:

The main algorithm The general shape of the algorithm is as follows. Starting with the choice of the primary base and/or the allocation of dynamic type data, it then lays out the remaining components of the class, first the non-virtual bases (other than the primary base), then the fields, and finally the virtual bases.

This algorithm computes the set eboffs'_C of empty subobjects that are subobjects of empty bases. So, to check whether an empty subobject is laid out without conflict, there are two cases:

1. Coq development: theory `CommonVendorABI`.

– either this empty subobject is a subobject of an empty base, then its offset is in eboffs'_C on which the check is directly performed
– or this empty subobject is a subobject of a non-empty base B, then its offset lies within the non-virtual size of B. But B is laid out in such a way that its non-virtual size is totally disjoint from the non-virtual sizes of other non-empty objects (thus including their empty bases), as well as from the offsets of empty subobjects of empty bases. So it is not necessary to collect the offsets of empty subobjects of non-empty bases, as they are covered by the whole memory spans of non-empty bases.

```
 1: $\mathsf{nvdsize}_C \leftarrow 0$,
    $\mathsf{pbase}_C, \mathsf{nveboffs}_C, \mathsf{eboffs}'_C \leftarrow \varnothing$,
    $\mathsf{dnvboff}_C, \mathsf{foff}_C, \mathsf{vboff}_C : \varnothing \to \mathbb{Z}$,
    $\mathsf{nvsize}_C, \mathsf{nvalign}_C \leftarrow 1$
 2: if $C$ is dynamic then
 3:   if $\exists B$ dynamic direct non-virtual base of $C$ then
 4:     $\mathsf{pbase}_C \leftarrow \{B\}$
 5:     $\mathsf{dnvboff}_C(B) \leftarrow 0$
 6:     $\mathsf{nveboffs}_C \leftarrow \mathsf{nveboffs}_B$
 7:     $\mathsf{nvdsize}_C \leftarrow \mathsf{nvsize}_B$
 8:     $\mathsf{nvsize}_C \leftarrow \mathsf{nvsize}_B$
 9:     $\mathsf{nvalign}_C \leftarrow \mathsf{nvalign}_B$
10:   else {there is no dynamic direct non-virtual base of $C$}
11:     $\mathsf{nvdsize}_C \leftarrow \mathsf{dtdsize}$
12:     $\mathsf{nvsize}_C \leftarrow \mathsf{dtdsize}$
13:     $\mathsf{nvalign}_C \leftarrow \mathsf{dtdalign}$
14:   end if
15: end if
16: lay out direct non-virtual bases $B$ of $C$ such that $\mathsf{pbase}_C \neq \{B\}$
17: $\mathsf{fboundary}_C \leftarrow \mathsf{nvdsize}_C$
18: lay out fields of $C$
19: $\mathsf{eboffs}_C \leftarrow \mathsf{nveboffs}_C$
20: $\mathsf{dsize}_C \leftarrow \mathsf{nvdsize}_C$
21: $\mathsf{size}_C \leftarrow \mathsf{nvsize}_C$
22: $\mathsf{align}_C \leftarrow \mathsf{nvalign}_C$
23: lay out virtual bases of $C$
24: $\mathsf{vboff}_C(C) \leftarrow 0$
25: $\mathsf{size}_C \leftarrow \min\{o \mid (\mathsf{align}_C \mid o) \wedge \mathsf{size}_C \leq o\}$
```

Direct non-virtual bases

```
1: for all $B$ direct non-virtual base of $C$ such that $\mathsf{pbase}_C \neq \{B\}$ do
2:   if $B$ is empty and $\mathsf{nveboffs}_B \# \mathsf{nveboffs}_C$ then
3:     $\mathsf{dnvboff}_C(B) \leftarrow 0$
4:   else
5:     $\mathsf{dnvboff}_C(B) \leftarrow \min\{o \mid (\mathsf{nvalign}_B \mid o) \wedge \mathsf{nvdsize}_C \leq o\}$
6:     while $\mathsf{dnvboff}_C(B) < \mathsf{nvsize}_C \wedge \neg \mathsf{dnvboff}_C(B) + \mathsf{nveboffs}_B \# \mathsf{eboffs}'_C$ do
7:       $\mathsf{dnvboff}_C(B) \leftarrow \mathsf{dnvboff}_C(B) + \mathsf{nvalign}_B$
8:     end while
```

```
 9:     end if
10:     $\mathsf{nveboffs}_C \leftarrow \mathsf{nveboffs}_C \cup \mathsf{dnvboff}_C(B) + \mathsf{nveboffs}_B$
11:     if $B$ is empty then
12:         $\mathsf{eboffs}'_C \leftarrow \mathsf{eboffs}'_C \cup \mathsf{dnvboff}_C(B) + \mathsf{nveboffs}_B$
13:     else
14:         $\mathsf{nvdsize}_C \leftarrow \mathsf{dnvboff}_C(B) + \mathsf{nvsize}_B$
15:     end if
16:     $\mathsf{nvsize}_C \leftarrow \max(\mathsf{nvsize}_C, \mathsf{dnvboff}_C(B) + \mathsf{nvsize}_B)$
17:     $\mathsf{nvalign}_C \leftarrow \mathrm{lcm}(\mathsf{nvalign}_C, \mathsf{nvalign}_B)$
18: end for
```

Line 6 imposes that the whole memory span of the object be disjoint from both:

- the whole memory spans of the non-empty bases of C, including the locations of their empty base subobjects, by checking on $\mathsf{nvdsize}_C$ (because the non-virtual data size of C includes the whole memory spans of its non-empty non-virtual bases)
- and the memory locations of empty subobjects that are subobjects of empty bases of C, by checking on eboffs'_C

Line 12 collects the offsets of empty subobjects that are subobjects of empty direct non-virtual bases only.

Line 14 imposes every non-empty direct non-virtual base B to have its whole non-virtual part (not only its data) included in the non-virtual *data* of C.

Thanks to this requirement, conflict checks for empty bases are limited to those offsets reachable only through empty direct bases (line 6). Indeed, in the “else” case of lines 5–8, B is laid out wholly (not only by its data) disjointly from all other non-empty non-virtual bases laid out so far. So, in particular, offsets to empty bases are disjoint.

Fields In this algorithm, no field is considered empty. Moreover, fields are laid out disjointly from the whole memory spans of the direct non-virtual bases of C. So the only requirement is to check whether a structure field overlaps memory locations of empty bases of empty direct non-virtual bases of C.

```
 1: for all $f$ field of $C$ do
 2:     $\mathsf{foff}_C(f) \leftarrow \min\{o \mid (\mathsf{falign}(f) \mid o) \wedge \mathsf{nvdsize}_C \leq o\}$
 3:     if $f$ is a structure field $(fid, B, n)$ then
 4:         while $\neg \bigcup_{0 \leq j < n} \mathsf{foff}_C(f) + j \cdot \mathsf{size}_B + \mathsf{eboffs}_B \;\#\; \mathsf{eboffs}'_C$ do
 5:             $\mathsf{foff}_C(f) \leftarrow \mathsf{foff}_C(f) + \mathsf{align}_B$
 6:         end while
 7:         $\mathsf{nveboffs}_C \leftarrow \mathsf{nveboffs}_C \cup \bigcup_{0 \leq j < n} \mathsf{foff}_C(f) + j \cdot \mathsf{size}_B + \mathsf{eboffs}_B$
 8:     end if
 9:     $\mathsf{nvdsize}_C \leftarrow \mathsf{foff}_C(f) + \mathsf{fsize}(f)$
10:     $\mathsf{nvsize}_C \leftarrow \max(\mathsf{nvsize}_C, \mathsf{foff}_C(f) + \mathsf{fsize}(f))$
11:     $\mathsf{nvalign}_C \leftarrow \mathrm{lcm}(\mathsf{nvalign}_C, \mathsf{falign}(f))$
12: end for
```

Virtual bases The layout of the non-virtual parts of (generalized) virtual bases is similar to the layout of the non-virtual parts of direct non-virtual bases. In particular, the data size of a most-derived C object covers the whole memory span of the non-virtual parts of non-empty virtual bases.

```
1: for all $B$ virtual base of $C$ do
2:   if $B$ is empty and $\mathsf{nveboffs}_B \# \mathsf{eboffs}_C$ then
3:     $\mathsf{vboff}_C(B) \leftarrow 0$
4:   else
5:     $\mathsf{vboff}_C(B) \leftarrow \min\{o \mid (\mathsf{align}_B \mid o) \land \mathsf{dsize}_C \leq o\}$
6:     while $\mathsf{vboff}_C(B) < \mathsf{size}_C \land \neg \mathsf{vboff}_C(B) + \mathsf{nveboffs}_B \# \mathsf{eboffs}'_C$ do
7:       $\mathsf{vboff}_C(B) \leftarrow \mathsf{vboff}_C(B) + \mathsf{nvalign}_B$
8:     end while
9:   end if
10:  $\mathsf{eboffs}_C \leftarrow \mathsf{eboffs}_C \cup \mathsf{vboff}_C(B) + \mathsf{nveboffs}_B$
11:  if $B$ is empty then
12:    $\mathsf{eboffs}'_C \leftarrow \mathsf{eboffs}'_C \cup \mathsf{vboff}_C(B) + \mathsf{nveboffs}_B$
13:  else
14:    $\mathsf{dsize}_C \leftarrow \mathsf{vboff}_C(B) + \mathsf{nvsize}_B$
15:  end if
16:  $\mathsf{size}_C \leftarrow \max(\mathsf{size}_C, \mathsf{vboff}_C(B) + \mathsf{size}_B)$
17:  $\mathsf{align}_C \leftarrow \mathrm{lcm}(\mathsf{align}_C, \mathsf{nvalign}_B)$
18: end for
```

Summary

Lemma 6.1.2. *This algorithm ensures that for any class C, for any non-empty non-virtual bases B_1, B_2 of C, for any fields F_1, F_2 declared in C, and for any non-empty generalized virtual bases V_1, V_2 of C:*

$$\mathsf{nvdsize}_C \leq \mathsf{nvsize}_C \qquad \begin{array}{rl} & [\mathsf{dnvboff}_C(B_1), \mathsf{dnvboff}_C(B_1) + \mathsf{nvsize}(B_1)) \\ \# & [\mathsf{dnvboff}_C(B_2), \mathsf{dnvboff}_C(B_2) + \mathsf{nvsize}(B_2)) \end{array}$$

$$\begin{array}{rl} & [\mathsf{foff}_C(F_1), \mathsf{foff}_C(F_1) + \mathsf{fsize}(F_1)) \\ \# & [\mathsf{foff}_C(F_2), \mathsf{foff}_C(F_2) + \mathsf{fsize}(F_2)) \end{array} \qquad \begin{array}{rl} & [\mathsf{vboff}_C(V_1), \mathsf{vboff}_C(V_1) + \mathsf{nvsize}(V_1)) \\ \# & [\mathsf{vboff}_C(V_2), \mathsf{vboff}_C(V_2) + \mathsf{nvsize}(V_2)) \end{array}$$

That is, two distinct sibling non-empty components are wholly disjoint (not only their data).

As a corollary:

Theorem I.9 (Correctness of the Common Vendor ABI layout algorithm). *This algorithm satisfies the soundness conditions of Chapter 5 (p. 93).*

6.2 An optimized algorithm: CCCPP

One obvious drawback of the previous algorithm is that the alignment tail padding of non-empty bases is never reused to store the data of subsequent non-empty components of an object. In this section, we propose an optimized algorithm, which we call CCCPP (standing for "CompCert C Plus Plus"), where the data of a class only includes the data of its bases, without their tail padding. Thus, tail paddings may be reused as long as there is no conflict on empty bases. However, we can no longer rely on the sizes of bases to reason about those conflicts. Consequently, we have to collect the offsets of all empty bases during the computation of layout.

Moreover, we also allow fields of empty class types to be laid out in an optimized way. To this purpose, we tailor the notion of empty class as follows:

***Definition* 6.2.1.** *A class C is* empty *if, and only if, all the following conditions hold:*
- *C has no virtual methods*
- *C has no direct virtual base*
- *C has no scalar fields*
- *if C has a structure array field of some type B, then B is empty*
- *all direct non-virtual bases of C are empty*

Given the definition of dynamic classes taken for our algorithms, this definition for empty bases is the smallest possible satisfying the constraints of Hypothesis 5.3.1 (p. 102).

Lemma 6.2.1. *Dynamic classes are not empty.*

In particular, a class having an empty virtual base is not empty.

The main algorithm The general shape of the algorithm[2] is similar to the previous one, except when choosing a primary base: in that case, after the choice of the primary base but before the layout of the other direct non-virtual bases, the non-virtual data size is updated to the non-virtual *data* size of the primary base (line 7), not its whole non-virtual size.

```
 1: $\mathsf{nvdsize}_C \leftarrow 0$,
    $\mathsf{pbase}_C, \mathsf{nveboffs}_C \leftarrow \varnothing$,
    $\mathsf{dnvboff}_C, \mathsf{foff}_C : \varnothing \to \mathbb{Z}$,
    $\mathsf{vboff}_C : C \mapsto 0$,
    $\mathsf{nvsize}_C, \mathsf{nvalign}_C \leftarrow 1$
 2: if $C$ is dynamic then
 3:    if $\exists B$ dynamic direct non-virtual base of $C$ then
 4:       $\mathsf{pbase}_C \leftarrow \{B\}$
 5:       $\mathsf{dnvboff}_C(B) \leftarrow 0$
 6:       $\mathsf{nveboffs}_C \leftarrow \mathsf{nveboffs}_B$
 7:       $\mathsf{nvdsize}_C \leftarrow \mathsf{nvdsize}_B$
 8:       $\mathsf{nvsize}_C \leftarrow \mathsf{nvsize}_B$
 9:       $\mathsf{nvalign}_C \leftarrow \mathsf{nvalign}_B$
10:    else {there is no dynamic direct non-virtual base of $C$}
11:       $\mathsf{nvdsize}_C \leftarrow \mathsf{dtdsize}$
12:       $\mathsf{nvsize}_C \leftarrow \mathsf{dtdsize}$
13:       $\mathsf{nvalign}_C \leftarrow \mathsf{dtdalign}$
14:    end if
15: end if
16: lay out direct non-virtual bases $B$ of $C$ such that $\mathsf{pbase}_C \neq \{B\}$
17: $\mathsf{fboundary}_C \leftarrow \mathsf{nvdsize}_C$
18: lay out fields of $C$
```

2. Coq development: theory `CCCPP`.

```
19: $\mathsf{eboffs}_C \leftarrow \mathsf{nveboffs}_C$
20: $\mathsf{dsize}_C \leftarrow \mathsf{nvdsize}_C$
21: $\mathsf{size}_C \leftarrow \mathsf{nvsize}_C$
22: $\mathsf{align}_C \leftarrow \mathsf{nvalign}_C$
23: lay out virtual bases of $C$
24: $\mathsf{size}_C \leftarrow \min\{o \mid (\mathsf{align}_C \mid o) \wedge \mathsf{size}_C \leq o \wedge \mathsf{dsize}_C \leq o\}$
```

Direct non-virtual bases The CCCPP algorithm considers more layout possibilities than the Common Vendor ABI. Indeed, the final choice of the offset is done by successive checks for empty base offset conflicts, by a systematic comparison against eboffs_C (line 7). This systematic comparison is necessary as those checks can no longer rely on the sizes of subobjects. Finally, the only difference between laying out an empty and a non-empty direct base is the starting point (lines 3–6): for an empty base, all offsets starting from 0 are tried; by contrast, for a non-empty base, offsets starting from $\mathsf{nvdsize}_C$ (maybe adding alignment padding) are tried, thus ensuring that the data of non-empty direct non-virtual bases are disjoint.

Another difference with the Common Vendor ABI algorithm is line 12, taking into account only the data part of non-empty bases, thus excluding its tail padding, which is then allowed for reuse.

```
 1: for all $B$ direct non-virtual base of $C$ such that $\mathsf{pbase}_C \neq \{B\}$ do
 2:   if $B$ is empty then
 3:     $\mathsf{dnvboff}_C(B) \leftarrow 0$
 4:   else
 5:     $\mathsf{dnvboff}_C(B) \leftarrow \min\{o \mid (\mathsf{nvalign}_B \mid o) \wedge \mathsf{nvdsize}_C \leq o\}$
 6:   end if
 7:   while $\mathsf{dnvboff}_C(B) < \mathsf{nvsize}_C \wedge \neg \mathsf{dnvboff}_C(B) + \mathsf{nveboffs}_B \# \mathsf{eboffs}_C$ do
 8:     $\mathsf{dnvboff}_C(B) \leftarrow \mathsf{dnvboff}_C(B) + \mathsf{nvalign}_B$
 9:   end while
10:   $\mathsf{nveboffs}_C \leftarrow \mathsf{nveboffs}_C \cup \mathsf{dnvboff}_C(B) + \mathsf{nveboffs}_B$
11:   if $B$ is not empty then
12:     $\mathsf{nvdsize}_C \leftarrow \mathsf{dnvboff}_C(B) + \mathsf{nvdsize}_B$
13:   end if
14:   $\mathsf{nvsize}_C \leftarrow \max(\mathsf{nvsize}_C, \mathsf{dnvboff}_C(B) + \mathsf{nvsize}_B)$
15:   $\mathsf{nvalign}_C \leftarrow \mathrm{lcm}(\mathsf{nvalign}_C, \mathsf{nvalign}_B)$
16: end for
```

Fields Here, contrary to the Common Vendor ABI, we distinguish between empty and non-empty fields. In the latter case, fields are laid out by making their data part disjoint only, thus allowing to reuse their tail padding (line 14). This approach follows the same pattern as for laying out the direct non-virtual bases.

```
1: for all $f$ field of $C$ do
2:   $\mathsf{foff}_C(f) \leftarrow \min\{o \mid (\mathsf{falign}(f) \mid o) \wedge \mathsf{nvdsize}_C \leq o\}$
3:   if $f$ is a structure field $(fid, B, n)$ then
4:     if $B$ is empty then
5:       $\mathsf{foff}_C(f) \leftarrow 0$
6:     else
7:       $\mathsf{foff}_C(f) \leftarrow \min\{o \mid (\mathsf{align}_B \mid o) \wedge \mathsf{nvdsize}_C \leq o\}$
```

```
 8:       end if
 9:       while $\neg \bigcup_{0 \leq j < n} \mathsf{foff}_C(f) + j \cdot \mathsf{size}_B + \mathsf{eboffs}_B \# \mathsf{eboffs}_C$ do
10:          $\mathsf{foff}_C(f) \leftarrow \mathsf{foff}_C(f) + \mathsf{align}_B$
11:       end while
12:       $\mathsf{nveboffs}_C \leftarrow \mathsf{nveboffs}_C \cup \bigcup_{0 \leq j < n} \mathsf{foff}_C(f) + j \cdot \mathsf{size}_B + \mathsf{eboffs}_B$
13:       if $B$ is not empty then
14:          $\mathsf{nvdsize}_C \leftarrow \mathsf{foff}_C(f) + \mathsf{fdsize}(f)$
15:       end if
16:    else {$f$ is a scalar field}
17:       $\mathsf{nvdsize}_C \leftarrow \mathsf{foff}_C(f) + \mathsf{fdsize}(f)$
18:    end if
19:    $\mathsf{nvsize}_C \leftarrow \max(\mathsf{nvsize}_C, \mathsf{foff}_C(f) + \mathsf{fsize}(f))$
20:    $\mathsf{nvalign}_C \leftarrow \mathrm{lcm}(\mathsf{nvalign}_C, \mathsf{falign}(f))$
21: end for
```

Virtual bases The layout algorithm has the same pattern as for laying out the direct non-virtual bases.

```
 1: for all $B$ virtual base of $C$ do
 2:    if $B$ is empty then
 3:       $\mathsf{vboff}_C(B) \leftarrow 0$
 4:    else
 5:       $\mathsf{vboff}_C(B) \leftarrow \min\{o \mid (\mathsf{align}_B \mid o) \wedge \mathsf{dsize}_C \leq o\}$
 6:    end if
 7:    while $\mathsf{vboff}_C(B) < \mathsf{size}_C \wedge \neg\mathsf{vboff}_C(B) + \mathsf{nveboffs}_B \# \mathsf{eboffs}_C$ do
 8:       $\mathsf{vboff}_C(B) \leftarrow \mathsf{vboff}_C(B) + \mathsf{nvalign}_B$
 9:    end while
10:    $\mathsf{eboffs}_C \leftarrow \mathsf{eboffs}_C \cup \mathsf{vboff}_C(B) + \mathsf{nveboffs}_B$
11:    if $B$ is not empty then
12:       $\mathsf{dsize}_C \leftarrow \mathsf{vboff}_C(B) + \mathsf{nvdsize}_B$
13:    end if
14:    $\mathsf{size}_C \leftarrow \max(\mathsf{size}_C, \mathsf{vboff}_C(B) + \mathsf{size}_B)$
15:    $\mathsf{align}_C \leftarrow \mathrm{lcm}(\mathsf{align}_C, \mathsf{nvalign}_B)$
16: end for
```

Theorem I.10 (Correctness of the optimized CCCPP layout algorithm). *This algorithm satisfies the soundness conditions of Chapter 5 (p. 93).*

The proof is simpler than that of Theorem I.9 (p. 131), as the CCCPP algorithm is much closer to the soundness conditions than the Common Vendor ABI algorithm.

Chapter 7

Application of verified object layout to a verified compiler

In this chapter, we define and formally verify a simple compiler for the s++ language of Chapter 4 (p. 71), taking advantage of the object layout framework from Chapter 5 (p. 93), but independent of the algorithm actually chosen, as long as it complies with the soundness conditions of the framework (Chapter 6 p. 127 shows that such algorithms exist in practice).

To this purpose, we specify a target language, called Vcm, featuring low-level memory accesses and a specific data structure to handle polymorphic operations, called *virtual tables*. Then, we build a compiler from s++ to Vcm, and we prove that the compiler preserves the semantics of programs through a compilation invariant.

7.1 Virtual tables

Whenever a polymorphic operation (dynamic cast, virtual function call, access to virtual base) is performed on an object, this operation is implemented by looking up its dynamic type, which is stored at offset 0 before any subsequent object data. Dynamic type data can be represented in many different forms (as shown by Driesen et al. [29] through their extensive survey of virtual function dispatch implementations), but, most often (such as in the GNU GCC compiler), it is a pointer to a table stored at a read-only memory location, called *virtual table*, containing all the information required to implement polymorphic operations.

Virtual function dispatch and `this` pointer adjustment Consider the following C++ code:

```
struct V1 {
  virtual void f();
  int iv1;
};
struct V2 {
  virtual void f();
  int iv2;
};
struct D: virtual V1, virtual V2 {
  void f() {
    this->i++;
  }
};

D d;
V1* v1 = (V1*)&d;
V2* v2 = (V2*)&d;
v1->f();
v2->f();
```

Those two calls to the `f` virtual function choose the `D::f` overriding definition in `D`. Within this function, the `this` pointer refers to the whole `D` object. At the level of function calls, `v1` and `v2` must be adjusted to the `D` object. This is called *`this` pointer adjustment*.

At the implementation level, adjustment is performed by subtracting some offset from the pointer used for the call. This offset, as well as the actual function to be called, is known only at run time, so it has to be stored somewhere in the virtual table.

Concretely, there are several solutions. The simplest solution is to store two pieces of data for each virtual function: the pointer to the function body, and the offset for `this` pointer adjustment. Another solution is to store only a pointer to a function which performs itself the adjustment. Such a function is called *thunk*. However, our formalization does not cover the concrete implementation, and models the contents of virtual tables only in an abstract way.

Notice that the adjustments are different from `V1` than from `V2`. This illustrates that different virtual tables are needed for each inheritance subobject of a given class.

Summary of the contents of virtual tables To sum up, for any class C, a virtual table for a C object (regardless of whether the object is a most-derived object, or inheritance subobject of another object) contains:

- the offset to each virtual base subobject of C
- for each well-defined dynamic cast, the offset to the corresponding objects
- for each virtual function having a final overrider for C, the pointer to the actually dispatched function as well as the offset for the `this` pointer adjustment.

Whereas such data are always present in the virtual tables for any C object, their contents may vary depending on the actual most-derived class D and the inheritance path σ from D to C.

We only give an abstract model of the contents of virtual tables, leaving aside their concrete implementation.

Sharing We saw in Section 5.3.1 (p. 98) that an object may share its dynamic type data with its primary bases. This implies that the virtual table of a class must also feature the necessary information of the virtual tables of its primary bases. Such information related to the primary base must be stored with respect to the structure of the virtual table of the primary base as this virtual table were stand-alone.

To avoid unnecessary duplication, the virtual tables of a class can share relevant pieces of information with the virtual tables of its primary bases. Consider the following example:

```
struct B {
  virtual void f();
};
struct D: B {
  void f();         /* overrides B::f */
  virtual void g();
};
```

In this single inheritance example, an instance of `D` will share its dynamic type data with its `B` subobject, so that the single virtual table entry (shared between `B` and `D`) corresponding to the `f` virtual function will accordingly contain a pointer to the `D::f` function overridden by `D`. As `B` is laid out at offset 0 within `D`, the `this` pointer adjustment to perform will be the same. This illustrates that a single virtual table can be used for both `D` and its primary base `B`.

7.2 The Vcm target language

7.2.1 Syntax

The Vcm target language is a variant of CompCert Cminor [2, 49, 16]. It features low-level memory accesses, pointer arithmetic, and virtual tables in an abstract representation. Hence its name: Vcm stands for "Cminor with virtual tables".

Like s++, Vcm is a 3-address language, without complex expressions: arguments of operations are necesarily variables. Vcm features built-in operations, and the usual structured control: conditionals, sequences, infinite loops, and statement blocks with early exit (avoiding the need for *goto*-like statements, and allowing exit from infinite loops). Moreover, Vcm also features function calls.

Unlike s++, Vcm also features low-level memory accesses: memory loads, memory stores, at memory addresses obtained by pointer arithmetic. However, in Vcm, object-oriented operations are no longer primitive, and class hierarchies are no longer needed. Instead, and additionally to CompCert Cminor, Vcm features accesses to virtual tables, to retrieve offsets to virtual bases, offsets to subobjects for dynamic casts, and pointers to dispatched virtual functions, along with their `this` pointer adjustment offsets. Those features are enough to implement C++ virtual function dispatch; however, to allow for further back-end optimizations, Vcm features *thunk calls*, a compound operation that performs virtual table accesses, `this` pointer adjustment and function call at the same time.

7.2.1.1 Memory chunks

The Vcm low-level memory accesses (load, store) are parameterized by the *memory chunk*, which expresses the expected kind of the value to read from memory or to write to memory:

T	: *Chunk*	Type of data writable to memory (chunk)
Chunk ::=	*BuiltinType*	Built-in type (*Notation* 3.2.6 p. 67)
	| `Ptr`	Pointer to memory
	| `Fptr`	Pointer to function
	| `Vptr`	Pointer to virtual table

7.2.1.2 Virtual table type definition

A virtual table type declares all entries that can be defined in virtual tables of this type. From the C++ point of view, a virtual table type $\tau \in \mathit{VTableType}$ corresponds to a class definition, but limited to the virtual bases, the virtual functions and the results of dynamic cast operations. *VTableType* is left as a parameter of the semantics of Vcm.

A virtual table type τ can be said to have a *parent* virtual table type $\tau' \in \mathit{VTableType}$: a virtual table of type τ must then contain all entries declared in the type τ', allowing for virtual table sharing: from the C++ point of view, this corresponds to the *primary base.*

$\mathit{VTableEntry}$	$::=$	$\texttt{vboff}\ B$	Virtual base offset
	$\mid$	$\texttt{dyncast}\ X$	Dynamic cast
	$\mid$	$\texttt{disp}\ F$	Virtual function dispatch

$$\mathit{VTableTypeDefs} = \mathit{VTableType} \nrightarrow \mathit{VTableTypeDef}$$

$\mathit{VTableTypeDef}$ =
{

parent	:	$\mathit{VTableType}^?$	; Parent virtual table (for sharing)
entries	:	$\mathit{VTableEntry}^*$	; Valid entries

}

7.2.1.3 Statements

The syntax of Vcm statements follows:

$st ::= \texttt{if}\ (x)\ st_{\mathsf{true}}\ \texttt{else}\ st_{\mathsf{false}}$	Conditional
$\mid st_1; st_2$	Statement sequence
$\mid \texttt{skip}$	Do nothing
$\mid \texttt{loop}\ st$	Loop
$\mid \{st\}$	Statement block
$\mid \texttt{exit}\ n$	Leaving n blocks
$\mid x' := x$	Variable value duplication
$\mid x' := op(x^\star)$	Built-in operation
$\mid \texttt{return}\ x^?$	Return from function
$\mid x' := \mathit{fname}(x^\star)$	Function call
$\mid x' := (*x)(x^\star)$	Function call through function pointer
$\mid x' := *_{\mathit{Chunk}}(x + \delta)$	Memory read at a constant δ offset from pointer x
$\mid *_{\mathit{Chunk}}(x + \delta) := x'$	Memory write at a constant δ offset from pointer x
$\mid x := x' + \delta$	Constant offset shift
$\mid x := x' + x_{\mathsf{i}} \cdot \delta$	Variable offset shift by a constant factor
$\mid x' := x_1 == x_2$	Pointer comparison
$\mid x' := \texttt{NULL}$	Null pointer
$\mid x' := \texttt{vboff}\langle B\rangle_\tau(x)$	Offset to virtual base
$\mid x' := \texttt{dyncastdef}\langle X\rangle_\tau(x)$	Is dynamic cast defined?
$\mid x' := \texttt{dyncastoff}\langle X\rangle_\tau(x)$	Dynamic cast offset

$\mid x' := \mathtt{dispfunc}\langle F\rangle_\tau(x)$	Virtual function dispatch
$\mid x' := \mathtt{dispoff}\langle F\rangle_\tau(x)$	`this` pointer adjustment for virtual function dispatch
$\mid x'^{?} := x\texttt{->}_\tau F(x_1, \ldots, x_n)$	Thunk call

There are three ways to call a function: by explicitly giving its name; through a function pointer; or through a *thunk call*. The latter is a compound operation to call a function by finding its function pointer and the `this` pointer adjustment in a virtual table.

7.2.1.4 Virtual tables

A virtual table is composed of three finite maps: virtual base offsets, dynamic cast offsets, and virtual function dispatch. The latter maps each function (for which dispatch is well-defined) to a function pointer and the corresponding `this` pointer adjustment.

$VTables = VTableName \twoheadrightarrow VTable$

$VTable =$

{

type	:	$VTableType$	; Type of virtual table (for abstract vtable layout)
vboff	:	$VBOffRequest \twoheadrightarrow \mathbb{Z}$	; Virtual base offsets
dyncast	:	$DynCastRequest \twoheadrightarrow \mathbb{Z}$	; Dynamic cast offsets
disp	:	$DispRequest \twoheadrightarrow FuncName \times \mathbb{Z}$	; Function pointer and `this` pointer adjustment for virtual function dispatch

}

VTableName, *VBOffRequest*, *DynCastRequest* and *DispRequest* are left as parameters of the semantics of Vcm.

7.2.1.5 Program

A program is composed of function definitions, the declaration of virtual table types and the contents of virtual tables:

$Func ::= (x^*)\{st\}$ Function definition

$Program =$

{

funcs	:	$FuncName \twoheadrightarrow Func$	; Functions
vtabletypes	:	$VTableTypeDefs$	; Virtual table types (for abstract vtable layout)
vtables	:	$VTables$	; Virtual tables

}

7.2.2 Memory model

The low-level memory model of Vcm is inspired from the CompCert memory model [16]. Memory is organized in several *memory blocks*. Each block is a finite array of bytes. Values are stored within one block, spanning an interval of one or several bytes.

7.2.2.1 Memory operations

The memory model defines the following operations: retrieve the size or alignment of a memory chunk (which are actually platform-specific parameters rather than operations on a memory state), load value from memory, store value to memory, and retrieve the size of a memory block.

$$
\begin{array}{rrll}
\multicolumn{2}{l}{\mathit{MemSpec} \quad =} & & \\
\multicolumn{2}{l}{\{} & & \\
\mathsf{chunksize} : & & \mathit{Chunk} \to \mathbb{N}^{>0} & ;\ \text{Size of data chunk} \\
\mathsf{chunkalign} : & & \mathit{Chunk} \to \mathbb{N}^{>0} & ;\ \text{Alignment of data chunk} \\
\mathsf{load} : & & \mathit{Mem} \times \mathit{Chunk} \times \mathit{MemBlock} \times \mathbb{Z} \to \mathit{Value}^{?} & ;\ \text{Memory load} \\
\mathsf{store} : & & \mathit{Mem} \times \mathit{Chunk} \times \mathit{MemBlock} \times \mathbb{Z} \times \mathit{Value} \to \mathit{Mem}^{?} & ;\ \text{Memory store} \\
\mathsf{blocksize} : & & \mathit{Mem} \times \mathit{MemBlock} \to \mathbb{N}^{?} & ;\ \text{Size of a memory block} \\
\multicolumn{2}{l}{\}} & &
\end{array}
$$

7.2.2.2 Values

Values A value is either a *value of built-in type* (integer, floating-point number, etc.), a pointer to a memory location (which is the pair of a memory block and an integer offset within this block), a null pointer, a pointer to a function, or a pointer to a virtual table:

$$
\begin{array}{lll}
\mathit{Val} ::= & \mathit{Builtin} & \text{Value of built-in type} \\
 & \mid @(b, o) & \text{Pointer to memory location at offset } o \text{ within block } b \\
 & \mid \mathsf{NULL} & \text{Null pointer} \\
 & \mid \&\mathit{FuncName} & \text{Pointer to function} \\
 & \mid \&\mathit{VTableName} & \text{Pointer to virtual table}
\end{array}
$$

7.2.2.3 Value typing

To write a value to memory, it must correspond to the chunk specified for the store operation. Thus, we define the relation $v : T$ denoting the fact that value v corresponds to the chunk T:

$$
\frac{\mathit{Builtin} \in \mathit{BuiltinType}}{\mathit{Builtin} : \mathit{BuiltinType}} \qquad \frac{}{@(b, o) : \texttt{Ptr}} \qquad \frac{}{\mathsf{NULL} : \texttt{Ptr}} \qquad \frac{}{\&\mathit{FuncName} : \texttt{Fptr}}
$$

$$
\frac{}{\&\mathit{VTableName} : \texttt{Vptr}}
$$

7.2.2.4 Axioms for memory load/store

We axiomatize the behaviour of memory operations as follows (CompCert [16] provides a memory model that satisfies those axioms):

– A successfully written value may be immediately read back:

$$
\frac{\mathsf{store}(\mathfrak{M}, T, b, o, v) = \mathfrak{M}' \neq \bot}{\mathsf{load}(\mathfrak{M}', T, b, o) = v} \qquad \text{(Vcm-mem-load-store-same)}
$$

– A memory write has no impact on reading from disjoint memory locations. Actually, this rule justifies the need for theorems of non-overlap such as Theorem I.3 (p. 115) or Theorem I.4 (p. 119), to argue for the correctness of field write.

$$\frac{b \neq b' \vee [o, o + \mathsf{chunksize}(T)) \;\#\; [o', o' + \mathsf{chunksize}(T')) \qquad \mathsf{store}(\mathfrak{M}, T', b', o') = \mathfrak{M}' \neq \bot}{\mathsf{load}(\mathfrak{M}', T, b, o) = \mathsf{load}(\mathfrak{M}, T, b, o)}$$

(Vcm-mem-load-store-other)

– A memory write is successful if the value is of the corresponding chunk, the range of bytes accessed is within the bounds of an existing block, and the offset within the block is correctly aligned. Actually, this rule justifies the need for theorems of alignment such as Theorem I.1 (p. 110) to argue for the correctness of field write.

$$\frac{\mathsf{blocksize}(\mathfrak{M}, b) = sz \neq \bot \qquad 0 \leq o \qquad \begin{array}{c} v : T \\ o + \mathsf{chunksize}(T) < sz \end{array} \qquad (o \mid \mathsf{chunkalign}(T))}{\mathsf{store}(\mathfrak{M}, T, b, o, v) \neq \bot}$$

(Vcm-mem-store-some)

– Storing a value into memory has no impact on the sizes of memory blocks:

$$\frac{\mathsf{store}(\mathfrak{M}, T, b, o, v) = \mathfrak{M}' \neq \bot}{\mathsf{blocksize}(\mathfrak{M}', b) = \mathsf{blocksize}(\mathfrak{M}, b)}$$

(Vcm-mem-store-blocksize)

In Vcm, memory blocks cannot be created; instead, they are assumed to already exist when the program starts.

7.2.3 Execution state

An *execution state* of the small-step semantics is composed of:
– the current statement to execute,
– the list of further statements to execute in the same block,
– the environment (mapping of values to variables),
– the continuation stack, which is a list of frames, each frame being either of:
  – leaving a block, with the further statements to execute after leaving the block,
  – returning from a function, with the caller variable to store the result (if any), the caller environment, and the further statements to execute on resumption
– the memory state

$\mathfrak{M}$	$\in$	*Mem*	Memory state
e	:	$\mathit{Var} \rightharpoonup \mathit{Val}^?$	Environment
Frame	::=	$\mathsf{Block}(st^*)$	Further statements after leaving a block
	|	$\mathsf{Callframe}(x^?, st^*, e)$	Return from function
$\mathcal{K}$	::=	Frame^*	Continuation stack
State	::=	$(st, st^*, e, \mathcal{K}, \mathfrak{M})$	Execution state

7.2.4 Semantic rules

The small-step semantics of Vcm is given by the transition relation $\to$ between two transition states, defined in this section.

7.2.4.1 Structured control, variable value duplication, built-in operations and statement blocks

Most structured control behaves similarly as in other CompCert-like languages: conditionals, sequences, infinite loops, and return from call (once all statements blocks within the current function have been left), as well as variable value duplication, and built-in operations (Hypothesis 3.2.2 p. 68). Vcm reuses the corresponding rules of s++ defined in Section 4.4.1.1 (p. 83). Vcm also reuses the s++ rules corresponding to statement blocks defined in Section 4.4.1.2 (p. 85).

7.2.4.2 Function call

Vcm defines functions. Function call is performed in two steps. First, the function to be called has to be selected. We write $e \vdash \varphi \rightsquigarrow \mathit{fname}$ to define function selection, depending on the kind of function call:

– either the function is given by its name:

$$\frac{}{e \vdash \mathit{fname} \rightsquigarrow \mathit{fname}} \qquad \text{(Vcm-call-select-static)}$$

– or the function is called through a function pointer, so its name is given by the (pointer) value of the variable:

$$\frac{e(x) = \&\mathit{fname}}{e \vdash *x \rightsquigarrow \mathit{fname}} \qquad \text{(Vcm-call-select-dynamic)}$$

Then, once the name of the function is known, arguments are passed and a new call frame is pushed onto the stack, and the function body is ready to execute:

$$\frac{\begin{array}{cc} e \vdash \varphi \rightsquigarrow \mathit{fname} & \mathsf{funcs}(\mathit{fname}) = (\mathit{varg}_1, \dots, \mathit{varg}_n)\{\mathit{st}\} \\ \forall i, e(x_i) = v_i & e' = \varnothing[\mathit{varg}_1 \leftarrow v_1] \dots [\mathit{varg}_n \leftarrow v_n] \end{array}}{\begin{array}{lllll} & (\varphi(x_1, \dots, x_n), & \mathit{stl}, & e, & \mathcal{K}, \quad \mathfrak{M}) \\ \to & (\mathit{st} \qquad\qquad\quad , & \epsilon, & e', & \mathsf{Callframe}(x'^?, \mathit{stl}, e) :: \mathcal{K}, \quad \mathfrak{M}) \end{array}} \qquad \text{(Vcm-call)}$$

There is also a third kind of function call: *thunk call*. It will be defined later, in Section 7.2.4.4 (p. 143).

7.2.4.3 Memory accesses and pointer arithmetics

Memory accesses Memory load (resp. store) reads from (resp. writes to) memory using the load (resp. store) memory model operation. Their success or failure are determined by the axioms of Section 7.2.2.4 (p. 140)

$$\frac{e(x) = @(b, o) \qquad \mathsf{load}(\mathfrak{M}, \mathit{Chunk}, b, o + \delta) = v \neq \bot \qquad e' = e[x' \leftarrow v]}{\begin{array}{llllll} & (x' := *_{\mathit{Chunk}}(x + \delta), & \mathit{stl}, & e, & \mathcal{K}, & \mathfrak{M}) \\ \to & (\mathtt{skip} \qquad\qquad\quad , & \mathit{stl}, & e', & \mathcal{K}, & \mathfrak{M}) \end{array}} \qquad \text{(Vcm-load)}$$

$$\frac{e(x) = @(b, o) \qquad e(x') = v \neq \bot \qquad \mathsf{store}(\mathfrak{M}, \mathit{Chunk}, b, o + \delta, v) = \mathfrak{M}' \neq \bot}{\begin{array}{llllll} & (*_{\mathit{Chunk}}(x + \delta) := x', & \mathit{stl}, & e, & \mathcal{K}, & \mathfrak{M}) \\ \to & (\mathtt{skip} \qquad\qquad\quad , & \mathit{stl}, & e, & \mathcal{K}, & \mathfrak{M}') \end{array}} \qquad \text{(Vcm-store)}$$

Pointer arithmetic Vcm defines shifting operations on pointers. They shift a pointer by an offset within the same memory block (it is impossible to retrieve a pointer to a different block): either a constant offset, or a variable offset multiplied by a constant factor. Moreover, the equality between two pointers to memory may be tested. Finally, it is always possible to retrieve a null pointer.

$$\frac{e(x) = @(b, o) \qquad e' = e[x' \leftarrow @(b, o+\delta)]}{\begin{array}{ll} & (x' := x + \delta, \ \ stl, \ \ e, \ \ \mathcal{K}, \ \ \mathfrak{M}) \\ \to & (\texttt{skip} \qquad\ \ , \ \ stl, \ \ e', \ \ \mathcal{K}, \ \ \mathfrak{M}) \end{array}} \quad \text{(Vcm-shift-const)}$$

$$\frac{e(x) = @(b, o) \qquad e(x_{\mathsf{i}}) = i \in \mathbb{Z} \qquad e' = e[x' \leftarrow @(b, o + i \cdot \delta)]}{\begin{array}{ll} & (x' := x + x_{\mathsf{i}} \cdot \delta, \ \ stl, \ \ e, \ \ \mathcal{K}, \ \ \mathfrak{M}) \\ \to & (\texttt{skip} \qquad\qquad\ , \ \ stl, \ \ e', \ \ \mathcal{K}, \ \ \mathfrak{M}) \end{array}} \quad \text{(Vcm-shift-const-factor)}$$

$$\frac{e(x_2) = @(b_2, o_2) \qquad res \in \mathbb{B} \qquad \begin{array}{l} e(x_1) = @(b_1, o_1) \\ res = \mathsf{true} \Leftrightarrow (b_1, o_1) = (b_2, o_2) \end{array} \qquad e' = e[x' \leftarrow res]}{\begin{array}{ll} & (x' := x_1 == x_2, \ \ stl, \ \ e, \ \ \mathcal{K}, \ \ \mathfrak{M}) \\ \to & (\texttt{skip} \qquad\qquad\ \ , \ \ stl, \ \ e', \ \ \mathcal{K}, \ \ \mathfrak{M}) \end{array}} \quad \text{(Vcm-ptreq)}$$

$$\frac{e' = e[x' \leftarrow \mathsf{NULL}]}{\begin{array}{ll} & (x' := \texttt{NULL}, \ \ stl, \ \ e, \ \ \mathcal{K}, \ \ \mathfrak{M}) \\ \to & (\texttt{skip} \qquad\ \ , \ \ stl, \ \ e', \ \ \mathcal{K}, \ \ \mathfrak{M}) \end{array}} \quad \text{(Vcm-null)}$$

7.2.4.4 Virtual tables and thunk call

Finally, it remains to define the operations on virtual tables: retrieving the function pointer, or the `this` pointer adjustment, for virtual function dispatch; determining whether a dynamic cast is defined, and, if so, retrieving the corresponding offset; retrieving the offset of a virtual base. Moreover, an additional function call is defined: *thunk call*, using virtual tables.

Valid access to information from virtual tables All those requests for information rely on the underlying type system for virtual tables. Operations are parameterized by a virtual table type τ such that the requested information is expected by the definition of τ. Then, access to such information is valid only if the actual type τ' of the virtual table pointer given by the value of the variable x is a *subtype* of τ, written $\tau \leq \tau'$. This subtyping relation is the reflexive and transitive closure of the virtual table type parentship relation.

$$\frac{\mathsf{vtabletypes}(\tau_2).\mathsf{parent} = \tau_1}{\tau_1 < \tau_2} \quad \text{(Vcm-vtype-lt-step)}$$

$$\frac{\tau_1 < \tau_2 \qquad \tau_2 < \tau_3}{\tau_1 < \tau_3} \quad \text{(Vcm-vtype-lt-trans)}$$

$$\frac{\tau_1 < \tau_2}{\tau_1 \leq \tau_2} \quad \text{(Vcm-vtype-le-step)}$$

$$\frac{}{\tau \leq \tau} \qquad \text{(Vcm-vtype-le-refl)}$$

Retrieving the offset of a virtual base from the virtual table succeeds only if the requested virtual base is declared as a potential entry in the type declaration of τ, and if the actual type τ' of the virtual table is a subtype of τ:

$$\frac{\begin{array}{c} e(x) = \&\mathit{vname} \qquad \mathsf{vtables}(\mathit{vname}) = V \\ V.\mathsf{vboff}(B) = \mathit{off} \qquad \tau \leq V.\mathsf{type} \qquad (\mathtt{vboff}\ B) \in \mathsf{vtabletypes}(\tau).\mathsf{entries} \qquad e' = e[x' \leftarrow \mathit{off}] \end{array}}{\begin{array}{rl} & (x' := \mathtt{vboff}\langle B\rangle_\tau(x),\ \ \mathit{stl},\ \ e,\ \ \mathcal{K},\ \ \mathfrak{M}) \\ \to & (\mathtt{skip}\ ,\ \ \mathit{stl},\ \ e',\ \ \mathcal{K},\ \ \mathfrak{M}) \end{array}}$$

(Vcm-vboff)

Similarly for retrieving function dispatch information. The two operations retrieving the function pointer on the one hand, and the `this` pointer adjustment offset on the other hand, are separate:

$$\frac{\begin{array}{c} e(x) = \&\mathit{vname} \qquad \mathsf{vtables}(\mathit{vname}) = V \qquad V.\mathsf{disp}(F) = (\mathit{fname}, \_) \\ \tau \leq V.\mathsf{type} \qquad (\mathtt{disp}\ F) \in \mathsf{vtabletypes}(\tau).\mathsf{entries} \qquad e' = e[x' \leftarrow \&\mathit{fname}] \end{array}}{\begin{array}{rl} & (x' := \mathtt{dispfunc}\langle F\rangle_\tau(x),\ \ \mathit{stl},\ \ e,\ \ \mathcal{K},\ \ \mathfrak{M}) \\ \to & (\mathtt{skip}\ ,\ \ \mathit{stl},\ \ e',\ \ \mathcal{K},\ \ \mathfrak{M}) \end{array}} \qquad \text{(Vcm-dispfunc)}$$

$$\frac{\begin{array}{c} e(x) = \&\mathit{vname} \qquad \mathsf{vtables}(\mathit{vname}) = V \\ V.\mathsf{disp}(F) = (\_, \mathit{off}) \qquad \tau \leq V.\mathsf{type} \qquad (\mathtt{disp}\ F) \in \mathsf{vtabletypes}(\tau).\mathsf{entries} \qquad e' = e[x' \leftarrow \mathit{off}] \end{array}}{\begin{array}{rl} & (x' := \mathtt{dispoff}\langle F\rangle_\tau(x),\ \ \mathit{stl},\ \ e,\ \ \mathcal{K},\ \ \mathfrak{M}) \\ \to & (\mathtt{skip}\ ,\ \ \mathit{stl},\ \ e',\ \ \mathcal{K},\ \ \mathfrak{M}) \end{array}}$$

(Vcm-dispoff)

For dynamic casts, a separate operation is necessary to know whether dynamic cast succeeds or fails.

$$\frac{\begin{array}{c} e(x) = \&\mathit{vname} \\ \mathsf{vtables}(\mathit{vname}) = V \qquad b \in \{\mathsf{true}, \mathsf{false}\} \qquad b = \mathsf{false} \Leftrightarrow V.\mathsf{dyncast}(X) = \bot \\ \tau \leq V.\mathsf{type} \qquad (\mathtt{dyncast}\ X) \in \mathsf{vtabletypes}(\tau).\mathsf{entries} \qquad e' = e[x' \leftarrow b] \end{array}}{\begin{array}{rl} & (x' := \mathtt{dyncastdef}\langle X\rangle_\tau(x),\ \ \mathit{stl},\ \ e,\ \ \mathcal{K},\ \ \mathfrak{M}) \\ \to & (\mathtt{skip}\ ,\ \ \mathit{stl},\ \ e',\ \ \mathcal{K},\ \ \mathfrak{M}) \end{array}} \qquad \text{(Vcm-dyncastdef)}$$

$$\frac{\begin{array}{c} e(x) = \&\mathit{vname} \qquad \mathsf{vtables}(\mathit{vname}) = V \qquad V.\mathsf{dyncast}(X) = \mathit{off} \\ \tau \leq V.\mathsf{type} \qquad (\mathtt{dyncast}\ X) \in \mathsf{vtabletypes}(\tau).\mathsf{entries} \qquad e' = e[x' \leftarrow \mathit{off}] \end{array}}{\begin{array}{rl} & (x' := \mathtt{dyncastoff}\langle X\rangle_\tau(x),\ \ \mathit{stl},\ \ e,\ \ \mathcal{K},\ \ \mathfrak{M}) \\ \to & (\mathtt{skip}\ ,\ \ \mathit{stl},\ \ e',\ \ \mathcal{K},\ \ \mathfrak{M}) \end{array}} \qquad \text{(Vcm-dyncastoff)}$$

Thunk call This function call performs the following operations in one step:

1. read the virtual table pointer from x
2. adjust x by the `this` pointer adjustment offset retrieved from the virtual table

3. then call the function given by the function pointer retrieved from the virtual table

$$\frac{\begin{array}{c} e(x) = @(b,o) \qquad \mathsf{load}(\mathfrak{M}, \mathtt{Vptr}, b, o) = \&\mathit{vname} \\ \mathsf{vtables}(\mathit{vname}) = V \qquad V.\mathsf{disp}(F) = (\mathit{fname}, \mathit{off}) \qquad \tau \le V.\mathsf{type} \\ (\mathtt{disp}\ F) \in \mathsf{vtabletypes}(\tau).\mathsf{entries} \qquad \mathsf{funcs}(\mathit{fname}) = (\mathit{this}, \mathit{varg}_1, \ldots, \mathit{varg}_n)\{\mathit{st}\} \\ \forall i, e(x_i) = v_i \qquad e' = \varnothing[\mathit{this} \leftarrow @(b, o + \mathit{off})][\mathit{varg}_1 \leftarrow v_1] \ldots [\mathit{varg}_n \leftarrow v_n] \end{array}}{\begin{array}{rlllll} & (x'^? := x\texttt{->}_\tau F(x_1, \ldots, x_n), & \mathit{stl}, & e, & \mathcal{K}, & \mathfrak{M}) \\ \to & (\mathit{st}, & \epsilon, & e', & \mathsf{Callframe}(x'^?, \mathit{stl}, e) :: \mathcal{K}, & \mathfrak{M}) \end{array}}$$

(Vcm-thunkcall)

This construct is meant to consider Vcm as a source language for further compilation to a lower-level language where the concrete representation of virtual tables would be made explicit. Then, this compound construct would facilitate a clever compilation of function calls using virtual tables (e.g. by thunks).

However, in our formalization, we leave open the choice of compiling virtual function calls through thunk calls or by explicitly and separately retrieving the relevant information from the virtual table.

7.3 A compiler from s++ to Vcm

Now that the semantics of the Vcm target language has been defined, we can compile s++ programs into Vcm. Our compiler, described in this section, transforms class member functions of n arguments into ordinary functions of $1 + n$ arguments, the additional argument corresponding to the `this` pointer. We make use of the layout algorithm to implement casts and field accesses, and we compile C++ virtual function dispatch, dynamic casts, and accesses to virtual bases, through accesses to a read-only structure called *virtual tables*. To this purpose, we show how to populate those read-only structures at the level of the Vcm program.

***Notation* 7.3.1.** *If st is a s++ statement, then the corresponding Vcm compiled statement will be written* $[\![st]\!]$

***Notation* 7.3.2 (Compilation of variable names).** *The variable names in the target Vcm program are of the two forms:*

- $\overline{x}$ *for any variable* x *of the source s++ program;*
- $\underline{y}$ *for any variable introduced by the compiler*

We assume that those notations are injective and that $\overline{x} \neq \underline{y}$.

***Notation* 7.3.3 (Compilation of function names).** *The function names of the target Vcm program are of the two forms:*

- $\overline{f}$ *for any static function* f *of the source s++ program;*
- $\underline{(B, msig)}$ *for any (virtual or non-virtual) method signature msig of a class member function declared in B. Such a name may be computed by* name mangling *(encoding the declaring class and argument types within the new name).*

We assume that those notations are injective and that $\overline{f} \neq \underline{(B, msig)}$.

***Definition* 7.3.4.** *The types of s++ scalar values are mapped to their corresponding Vcm memory chunks by the following* $[\![\cdot]\!]$ *function:*

$$[\![\mathit{BuiltinType}]\!] \underset{\text{def.}}{=\!=} \mathit{BuiltinType} \qquad\qquad [\![C*]\!] \underset{\text{def.}}{=\!=} \mathtt{Ptr}$$

Hypothesis 7.3.1. *The size and alignment of a scalar data of type t, given by platform-dependent parameters of the layout algorithm, matches its chunk size and alignment at the level of Vcm memory model:*

$$\mathsf{scsize}(t) = \mathsf{chunksize}([\![t]\!]) \qquad\qquad \mathsf{scalign}(t) = \mathsf{chunkalign}([\![t]\!])$$

Hypothesis 7.3.2. *The size and alignment of dynamic type data given by platform-dependent parameters of the layout algorithm, match the corresponding size and alignment at the level of Vcm memory model:*

$$\mathsf{dtdsize} = \mathsf{chunksize}(\mathtt{Vptr}) \qquad\qquad \mathsf{dtdalign} = \mathsf{chunkalign}(\mathtt{Vptr})$$

7.3.1 Virtual tables

We have seen that a virtual table is necessary for any inheritance subobject that is not a primary base class subobject of another distinct object. In this section, we investigate which virtual tables to construct, and how to construct them.

7.3.1.1 What is an adjustment ?

Virtual tables contain `this` pointer adjustment offsets for virtual function dispatchs, and adjustment offsets for dynamic casts.

More generally, if a s++ operation transforms (in either way) a subobject σ to another subobject σ' of the same most-derived object D, then, at the low level of the Vcm compiled program, this is translated by adding the adjustment offset $\mathsf{soff}_D(\sigma') - \mathsf{soff}_D(\sigma)$ to the pointer to the subobject.

7.3.1.2 Which virtual tables are necessary?

It can be easily seen that:

Lemma 7.3.1. *The offset of a subobject is the same of the offset of its reduced path*
More formally, if $D[n] \prec\!\langle(\alpha, i, \sigma)\rangle\!\succ B$*, then:*

$$\mathsf{off}_D(\alpha, i, \sigma) = \mathsf{off}_D(\alpha, i, \mathsf{reducePath}(\sigma))$$

Consequently, a subobject has the same offset as all its primary base subobjects.

Conversely:

Lemma 7.3.2. *An inheritance path* $\sigma = (h, l)$ *from* D *to* C *is not a primary subobject of another distinct subobject if, and only if,* $l = \mathsf{reducePath}(l)$*. Moreover, those are the only paths of the form* $(h', \mathsf{reducePath}(l'))$ *for some* (h', l')*.*

Proof. We have $l = \mathsf{reducePath}(l)@(\mathsf{Repeated}, l')$ where l' is primary, so the equality holds if and only if l' is trivial, which concludes. □

So, we claim that it suffices to construct the virtual tables for inheritance subobjects of the form (h, l) such that $l = \mathsf{reducePath}(l)$.

In the following sections, we show that the virtual table of a subobject may share its information with its primary base class subobjects.

7.3.1.3 Virtual bases

Theorem I.11 (Correctness of Vcm virtual table sharing: virtual base offsets). *The adjustment offset to access a virtual base V from a primary base B of a C subobject is the same as from C.*

Proof. Consider a C inheritance subobject of a most-derived D object, say $D \prec\langle\sigma\rangle\xrightarrow{\mathcal{I}} C$. Let B be a primary base of C, say $C \prec\langle l\rangle\xrightarrow{\mathcal{NV}} B$ where l is primary. If V is a virtual base of B, then V is also a virtual base of C; and the adjustment to access V from B is $\mathsf{vboff}_D(V) - \mathsf{soff}_D(\sigma@(\mathsf{Repeated}, l)) = \mathsf{vboff}_D(V) - \mathsf{soff}_D(\sigma) - \mathsf{nvsoff}(l)$. As l is primary , $\mathsf{nvsoff}(l) = 0$, which concludes. □

7.3.1.4 Dynamic cast

Lemma 7.3.3. *Let C be a class and B be a non-virtual base of C, such that $C \prec\langle l\rangle\xrightarrow{\mathcal{NV}} B$ for some non-virtual path l.*

If X is not a base class of C, then, for any subobject σ of D of static type C, dynamic cast from σ to X succeeds if, and only if, dynamic cast from $\sigma@(\mathsf{Repeated}, l)$ to X succeeds.

Proof. First assume that the dynamic cast from σ to X succeeds. Then, there are two cases:

- either C is a non-virtual base of X along σ (i.e. $\sigma = \sigma'@(\mathsf{Repeated}, l')$ where $D \prec\langle\sigma'\rangle\xrightarrow{\mathcal{I}} X \prec\langle l'\rangle\xrightarrow{\mathcal{NV}} C$). In this case, dynamic cast from B to X also succeeds, as B is also a non-virtual base of X along $\sigma@(\mathsf{Repeated}, l)$.
- or there is a unique X subobject within D, then dynamic cast obviously succeeds.

Now assume that the dynamic cast from $\sigma@(\mathsf{Repeated}, l)$ to X succeeds. Then, there are two cases:

- either B is a non-virtual base of X along $\sigma@(\mathsf{Repeated}, l)$. Then, as X is not a base of C, we necessarily have that C is a non-virtual base of X, so the dynamic cast succeeds.
- or there is a unique X subobject within D, then dynamic cast obviously succeeds. □

This result is no longer true if B is a base of C through virtual inheritance. Indeed, consider the following hierarchy:

```
struct B {};
struct C: virtual B {};
struct X:  C {};
struct Y1: X {};
struct Y2: X {};
struct D: Y1, Y2 {};
```

Within a most-derived D object, dynamic cast from C to X succeeds, as C is a non-virtual base of X; however, dynamic cast from B to X fails, as B is not a non-virtual base of X and D has two distinct inheritance paths to K.

Corollary 7.3.4. *If B is a primary base of C such that the dynamic cast from a B subobject to a class X that is not a base of C succeeds, then the adjustment is performed through the same offset as from the C subobject.*

Proof. Let σ' be the inheritance path from D to the X subobject resulting from the dynamic cast. Then, the adjustment corresponding to the cast from B to X is performed through the offset $\mathsf{soff}_D(\sigma') - \mathsf{soff}_D(\sigma @ (\mathsf{Repeated}, l)) = \mathsf{soff}_D(\sigma) + \mathsf{nvsoff}(l)$ per Lemma 5.4.1 (p. 105). But B is a primary base of C. Thus l is a primary path, which implies that $\mathsf{nvsoff}(l) = 0$ and concludes. □

Then, if $D \mathrel{-\!\langle\sigma\rangle\!\overset{\mathcal{I}}{\rightarrowtail}} C$, we define, for any class X, the following offset $\Delta(D, \sigma, X)$, by well-founded induction on C using the well-founded order $\prec$ (Section 4.1.4.2 p. 79):

$$\Delta(D,\sigma,X) = \begin{cases} \mathsf{soff}_D(\sigma') - \mathsf{soff}_D(\sigma) & \text{if } D \mathrel{-\!\langle\sigma\rangle\!\overset{\mathcal{I}}{\rightarrowtail}} C \land \mathsf{DynCast}(D,\sigma,C,X,\sigma') \\ & \text{and } X \text{ not a base of } C \\ \bot & \text{if } D \mathrel{-\!\langle\sigma\rangle\!\overset{\mathcal{I}}{\rightarrowtail}} C \land \mathsf{DynCast}(D,\sigma,C,X,\bot) \\ & \text{and } X \text{ not a base of } C \\ \Delta(D, \sigma @ (\mathsf{Repeated}, C :: B :: \epsilon), X) & \text{if } \mathsf{pbase}(C) = B \text{ and } X \text{ base of } C \\ \bot & \text{if } \mathsf{pbase}(C) = \bot \text{ and } X \text{ base of } C \end{cases}$$

so that:

Theorem I.12 (Correctness of Vcm virtual table sharing: dynamic cast). *If $D \mathrel{-\!\langle\sigma\rangle\!\overset{\mathcal{I}}{\rightarrowtail}} C \mathrel{-\!\langle l\rangle\!\overset{\mathcal{NV}}{\rightarrowtail}} B$ where l is a non-virtual primary path, then, for any class X that is not a base of B, the dynamic cast from B to X succeeds if, and only if, $\Delta(D, \sigma, X) = \delta \neq \bot$, and, in this case, the adjustment is performed by adding offset δ.*

Proof. By induction on the length of l, with the help of Corollary 7.3.4 (p. 148).

- If $l = C :: \epsilon$, then $B = C$ and the result comes by definition of Δ.
- Otherwise, if X is not a base of C, then Lemma 7.3.3 (p. 147) concludes.
- Otherwise, $l = C :: B' :: l'$ and we use the induction hypothesis with

$$D \mathrel{-\!\langle\sigma @ (\mathsf{Repeated}, C :: B' :: \epsilon)\rangle\!\overset{\mathcal{I}}{\rightarrowtail}} B' \mathrel{-\!\langle B' :: l'\rangle\!\overset{\mathcal{NV}}{\rightarrowtail}} B$$

  where B' is the direct primary non-virtual base of C and $B' :: l'$ is a primary path. □

Here the hypothesis of X not being a base class of B plays an important role. To understand why, consider the following example:

```
struct A {
  virtual void f ();
};
struct B1: A {};
struct B2: A {};
struct D: B1, B2 {
  virtual void f ();
};
```

A dynamic cast from `B1` to `A` succeeds, since `A` is a non-ambiguous base class of `B1`. However, a dynamic cast from `D` to `A` fails, since `A` is an ambiguous base class of `D`. But if `B1` and `D` share their pointers to virtual tables, then different information for dynamic cast to `A` have to be stored in the virtual table common to `B1` and `D` (dynamic cast failure has to be explicitly stored as such in the virtual table). Such a discrepancy does not occur for dynamic casts to non-bases.

7.3.1.5 Virtual function dispatch

Lemma 7.3.5. *Let $C_\circ \prec\!\langle\sigma\rangle\!\overset{\mathcal{I}}{\rightarrowtail} C$ be an inheritance path. If σ'' is a final overrider for σ to dispatch some function f, then σ'' is also a final overrider for any $\sigma @ \sigma'$ base class subobject of σ such that the static dispatch of f succeeds. More formally, if:*

$$\mathsf{finalOverrider}(C_\circ, \sigma, C, f, B'', \sigma'')$$

then, for any inheritance path $C \prec\!\langle\sigma'\rangle\!\overset{\mathcal{I}}{\rightarrowtail} B$ such that $\mathsf{staticDispatch}(B, f, B_f, \sigma_f)$, we have:

$$\mathsf{finalOverrider}(C_\circ, \sigma @ \sigma', B, f, B'', \sigma'')$$

Proof. By definition of $\mathsf{finalOverrider}$ ($\mathsf{final\text{-}overrider}$, p. 90), there is a unique σ'_f such that $\mathsf{staticDispatch}(C, f, B'_f, \sigma'_f)$. But we have already $\mathsf{staticDispatch}(B, f, B_f, \sigma_f)$, so that $(B'_f, \sigma'_f) = (B_f, \sigma' @ \sigma_f)$.

By hypothesis, we then have that σ'' is the closest subobject to $C_\circ$ defining f along $\sigma @ \sigma'_f = \sigma @ \sigma' @ \sigma_f$, so by definition, σ'' is also a final overrider for $\sigma @ \sigma'$. □

Corollary 7.3.6. *If method dispatch succeeds for a subobject and one of its base class subobjects, then they agree on the dispatch candidate:*

$$\left.\begin{array}{l}\mathsf{VFDispatch}(C_\circ, \sigma, f, B_1, \sigma_1)\\ \mathsf{VFDispatch}(C_\circ, \sigma @ \sigma', f, B_2, \sigma_2)\end{array}\right\} \Rightarrow (B_1, \sigma_1) = (B_2, \sigma_2)$$

Proof. By definition, (B_1, σ_1) is a final overrider of f for σ.

By hypothesis, $\sigma @ \sigma'$ has a final overrider, so in particular static dispatch succeeds, so we can use Lemma 7.3.5 (p. 149) to say that (B_1, σ_1) is also a final overrider of f for $\sigma @ \sigma'$. Unicity of virtual function dispatch (by definition of $\mathsf{VFDispatch}$) concludes. □

It is worth noting that, for any subobject σ of a most-derived object $C_\circ$, the set:

$$\{(B', \sigma') : \mathsf{VFDispatch}(C_\circ, \sigma, f, B', \sigma')\}$$

can be computed in a finite amount of time. So, we define, by well-founded induction on $\mathsf{last}(\sigma)$ (using the $\prec$ order on class names, cf. Section 4.1.4.2 p. 79), the following $\Phi(C_\circ, \sigma, f)$ function:

$$\Phi(C_\circ, \sigma, f) = \begin{cases} (\underline{(B, f)}, \mathsf{soff}_{C_\circ}(\sigma') - \mathsf{soff}_{C_\circ}(\sigma)) & \text{if } \mathsf{VFDispatch}(C_\circ, \sigma, f, B, \sigma') \\ \Phi(C_\circ, \sigma @ (\mathsf{Repeated}, C :: B :: \epsilon), f) & \text{if dispatch fails for } \sigma \\ & \text{and } \mathsf{pbase}(C) = B \\ \bot & \text{if dispatch fails for } \sigma \\ & \text{and } \mathsf{pbase}(C) = \bot \end{cases}$$

Theorem I.13 (Correctness of Vcm virtual table sharing: virtual function dispatch). *If virtual function dispatch for f succeeds on a primary subobject $\sigma@(\mathsf{Repeated}, l)$ of the subobject σ of the most-derived object $C_\circ$, then $\Phi(C_\circ, \sigma, f) = (\underline{(B, f)}, \delta)$ where B is the class of the final overrider, and δ the offset for the* `this` *pointer adjustment.*

Indeed, the following hierarchy:

```
struct B1 {
  virtual void f();
};
struct B2 {
  virtual void f();
};
struct D: B1, B2 {};
```

is a valid C++ program. Here, the dispatch of f for a D most-derived object fails, but it succeeds for each of its B_1 and B_2 subobjects, in particular for B_1, which shares its virtual table with D. Thus, the virtual table contains information accurate for B_1 but not for D.

7.3.1.6 Summary

Using the previous results in this section, we are now in a position to compute the virtual table types and the virtual tables, starting from the class hierarchy.

Virtual table types The virtual function types are the names of dynamic classes:

$$\mathit{VTableType} \underset{\text{def.}}{=\!=} \{B : \mathsf{isDynamic}(B)\}$$

A virtual function type A is the parent of the virtual function type B if, and only if, A is the direct non-virtual primary base of B:

$$\mathsf{vtabletypes}(B).\mathsf{parent} \underset{\text{def.}}{=\!=} \mathsf{pbase}(B)$$

Let B be a dynamic class.

***Definition* 7.3.5.** *Since the hierarchy is well-founded, the set $\mathcal{M}'(B)$ of virtual functions inherited by B can be recursively computed as follows:*

$$\begin{aligned} \mathcal{M}'(B) \underset{\text{def.}}{=\!=} & \{f : \mathcal{M}(B)(f) = \mathsf{true}\} \\ \cup & \bigcup_{A \in \mathcal{DV}(B) \cup \mathcal{DNV}(B)} \mathcal{M}'(A) \end{aligned}$$

Finally, the following entries may be defined in virtual tables:
– the virtual functions declared in B or inherited by B
– the virtual bases of B
– dynamic casts to any defined classes

$$\begin{aligned} \mathsf{vtabletypes}(B).\mathsf{entries} \underset{\text{def.}}{=\!=} & \{\texttt{disp}\ f : f \in \mathcal{M}'(B)\} \\ & \cup \{\texttt{dyncast}\ X : X \in \mathcal{C}\} \\ & \cup \{\texttt{vboff}\ V : V \in \mathcal{V}(B)\} \end{aligned}$$

Lemma 7.3.7. *If $A \leq B$, then* $\mathsf{vtabletypes}(A).\mathsf{entries} \subseteq \mathsf{vtabletypes}(B).\mathsf{entries}$.

Proof. We first consider the special case where A is the direct non-virtual primary base of B. Then, the virtual functions inherited by B include the virtual functions declared in A or inherited by A. The virtual bases of A are also virtual bases of B. Finally the result generalizes to any A by transitivity. □

Virtual tables The virtual tables are the inheritance paths to dynamic classes that are not primary subobjects of other objects:

$$\mathit{VTableName} \underset{\text{def.}}{=\!=} \left\{ (D,(h,l)) : \begin{array}{l} D \prec\!(h,l)\!\succ\!\xrightarrow{\mathcal{I}} C \\ \mathsf{isDynamic}(C) \\ l = \mathsf{reducePath}(l) \end{array} \right\}$$

This set can be computed in a finite amount of time (such inheritance paths can be enumerated).

Lemma 5.5.20 (p. 120) shows that any inheritance path (h',l') is of the form:

$$(h', \mathsf{reducePath}(l') @_{\mathsf{Repeated}} l'')$$

where l'' is a primary non-virtual path. It follows that no inheritance path has been forgotten. Then, for any virtual table name $(D,(h,l))$, its type is the destination of the inheritance path (h,l):

$$\mathsf{vtables}(D,(h,l)).\mathsf{type} \underset{\text{def.}}{=\!=} \mathsf{last}(l)$$

Thanks to Theorem I.11 (p. 147), the finite map of virtual base offsets of a virtual table $(D,(h,l))$ is computed as follows:

$$\mathit{VBOffRequest} \underset{\text{def.}}{=\!=} \mathcal{C}$$

$$\mathsf{vtables}(D,(h,l)).\mathsf{vboff} \underset{\text{def.}}{=\!=} \begin{array}{lcl} V \in V_{\mathsf{last}(l)} & \mapsto & \mathsf{vboff}_D(V) - \mathsf{soff}_D(h,l) \\ V \notin V_{\mathsf{last}(l)} & \mapsto & \bot \end{array}$$

Similarly, thanks to Theorem I.12 (p. 148), the finite map of dynamic cast offsets for a virtual table $(D,(h,l))$ is computed as follows:

$$\mathit{DynCastRequest} \underset{\text{def.}}{=\!=} \mathcal{C}$$

$$\mathsf{vtables}(D,(h,l)).\mathsf{dyncast} \underset{\text{def.}}{=\!=} X \mapsto \Delta(D,(h,l),X)$$

Similarly, thanks to Theorem I.13 (p. 150), the finite map of virtual function dispatch for a virtual table $(D,(h,l))$ is computed as follows:

$$\mathit{DispRequest} \underset{\text{def.}}{=\!=} \mathit{MethodSig}$$

$$\mathsf{vtables}(D,(h,l)).\mathsf{disp} \underset{\text{def.}}{=\!=} \begin{array}{lcl} f \in \mathcal{M}'(B) & \mapsto & \Phi(D,(h,l),f) \\ f \notin \mathcal{M}'(B) & \mapsto & \bot \end{array}$$

7.3.2 Operations unrelated to C++ multiple inheritance

Structured control, built-in operations, statement blocks compile trivially, using *Notation* 7.3.2 (p. 145) to match variables:

$$
\begin{aligned}
[\![\texttt{if}(x) st_{\textsf{true}}\ \texttt{else}\ st_{\textsf{false}}]\!] &\overset{}{\underset{\text{def.}}{=\!=}} \texttt{if}(\overline{x})[\![st_{\textsf{true}}]\!]\ \texttt{else}\ [\![st_{\textsf{false}}]\!] \\
[\![st_1; st_2]\!] &\underset{\text{def.}}{=\!=} [\![st_1]\!]; [\![st_2]\!] \\
[\![\texttt{skip}]\!] &\underset{\text{def.}}{=\!=} \texttt{skip} \\
[\![\texttt{loop}\ st]\!] &\underset{\text{def.}}{=\!=} \texttt{loop}\ [\![st]\!] \\
[\![\texttt{return}\ x^?]\!] &\underset{\text{def.}}{=\!=} \texttt{return}\ \overline{x}^? \\
[\![x'^? := op(x^*)]\!] &\underset{\text{def.}}{=\!=} \overline{x'}^? := op(\overline{x}^*) \\
[\![x' := x]\!] &\underset{\text{def.}}{=\!=} \overline{x'} := \overline{x} \\
[\![\{st\}]\!] &\underset{\text{def.}}{=\!=} \{[\![st]\!]\} \\
[\![\texttt{exit}\ n]\!] &\underset{\text{def.}}{=\!=} \texttt{exit}\ n
\end{aligned}
$$

Static and non-virtual calls use *Notation* 7.3.3 (p. 145) to match function names.

A class member function call $x\texttt{->}msig(x_1, \ldots, x_n)$ of a class B is compiled with the x argument transformed into an ordinary argument in first position: $\underline{(B, msig)}(x, x_1, \ldots, x_n)$.

$$
\begin{aligned}
[\![x'^? := f(x^*)]\!] &\underset{\text{def.}}{=\!=} \overline{x'}^? := \overline{f}(\overline{x}^*) \\
[\![x'^? := x\texttt{->}C\texttt{::}msig(x^*)]\!] &\underset{\text{def.}}{=\!=} \overline{x'}^? := \underline{(C, msig)}(\overline{x}, \overline{x}^*)
\end{aligned}
$$

Recall that in s++, fully qualified non-virtual class member function calls make no implicit cast, expecting the x argument to be of the exact static type C. Therefore, no `this` pointer adjustment is necessary for x.

7.3.3 Field and array accesses

Scalar fields Let x be a variable containing a pointer to some subobject of static type C. Then, the value of the scalar field f is stored at offset $\mathsf{foff}_C(f)$ within the C subobject. Thus, reading and writing the scalar field f of type T is compiled into the following memory read or write:

$$
\begin{aligned}
[\![x' := x\texttt{->}_C f]\!] &\underset{\text{def.}}{=\!=} \overline{x'} := *_{[\![T]\!]}(\overline{x} + \mathsf{foff}_C(f)) \\
[\![x\texttt{->}_C f := x']\!] &\underset{\text{def.}}{=\!=} *_{[\![T]\!]}(\overline{x} + \mathsf{foff}_C(f)) := \overline{x'}
\end{aligned}
$$

The success of memory write is ensured by its bounds thanks to Theorem I.2 (p. 112), and its correct alignment thanks to Theorem I.1 (p. 110).

Structure fields In s++, access to structure fields is simply a pointer adjustment. In Vcm, this translates to a pointer shift by the constant $\mathsf{foff}_C(f)$ offset, with no memory read:

$$[\![x' := x\text{->}_C f]\!] \underset{\text{def.}}{=\!=} \overline{x'} := \overline{x} + \mathsf{foff}_C(f)$$

Structure array accesses Similarly, in s++, access to an array cell of type C is simply a pointer adjustment from a most-derived C object. In Vcm, this translates to a pointer shift by a variable offset given by the index, multiplied by a constant factor given by the size of a cell, the size of a most-derived object C.

$$[\![x' := x[x_{\mathsf{index}}]_C]\!] \underset{\text{def.}}{=\!=} \overline{x'} := \overline{x} + \overline{x_{\mathsf{index}}} \cdot \mathsf{size}_C$$

7.3.4 Pointer equality tests

Thanks to the object identity requirement (THEOREM I.7 p. 122; THEOREM I.8 p. 124), pointer equality tests are compiled trivially:

$$[\![x' := x_1 ==_C x_2]\!] \underset{\text{def.}}{=\!=} \overline{x'} := \overline{x_1} == \overline{x_2}$$

7.3.5 Static casts

Consider an inheritance subobject σ of D of static type C:

$$D \prec\!\langle\sigma\rangle\!\overset{\mathcal{I}}{\succ} C$$

Consider a static cast of σ to X. C, X are known at compile time. Then, the compiler proceeds to the following case analysis:

- If X is a non-ambiguous non-virtual base of C, the non-virtual path l from C to X is known at compile time, and so is its offset. The cast necessarily succeeds with the result $\sigma@(\mathsf{Repeated}, l)$. Thus, the static cast translates to a constant offset shift:

$$[\![x' := \mathtt{static\_cast}\langle X\rangle_C(x)]\!] \underset{\text{def.}}{=\!=} \overline{x'} := \overline{x} + \mathsf{nvsoff}(l)$$

- Similarly, if C is a non-ambiguous non-virtual base of X, then the non-virtual path l from X to C is known at compile time, and so is its offset. If the cast succeeds, then actually $\sigma = (h, l'@_{\mathsf{Repeated}}l)$, so that the result of the cast is (h, l'). Thus, the static cast translates to a constant offset shift:

$$[\![x' := \mathtt{static\_cast}\langle X\rangle_C(x)]\!] \underset{\text{def.}}{=\!=} \overline{x'} := \overline{x} + -\mathsf{nvsoff}(l)$$

- If X is a non-ambiguous base of C through virtual inheritance, then the path $(\mathsf{Shared}, V :: l)$ is known at compile time. However, the offset to the virtual base V is not. This offset must first be read from the virtual table of the current subobject. Then the further adjustment to the $V :: l$ non-virtual subobject is a constant offset:

$$\begin{aligned}[\![x' := \mathtt{static\_cast}\langle X\rangle_C(x)]\!] \underset{\text{def.}}{=\!=}\ & \underline{tmp} := *_{\mathtt{Vptr}}(\overline{x})\\ & ; \underline{tmp} := \mathtt{vboff}\langle X\rangle_C(\underline{tmp})\\ & ; \overline{x'} := \overline{x} + \underline{tmp} \cdot 1\\ & ; \overline{x'} := \overline{x'} + \mathsf{nvsoff}(l)\end{aligned}$$

7.3.6 Dynamic casts

s++ defines dynamic casts from a C subobject to some class X only if X is not a base of C. Thus, in Vcm, dynamic casts are compiled using the virtual table. Dynamic casts must not fail: if undefined, their result is the null pointer.

$$\begin{aligned}
[\![x' := \texttt{dynamic\_cast}\langle X\rangle_C(x)]\!] \overset{}{\underset{\text{def.}}{=\!=}} \quad & \underline{tmp_1} := *_{\texttt{Vptr}}(\overline{x}) \\
; \quad & \underline{tmp_2} := \texttt{dyncastdef}\langle X\rangle_C(\underline{tmp_1}) \\
; \quad & \texttt{if}(\underline{tmp_2}) \\
& \qquad \underline{tmp_1} := \texttt{dyncastoff}\langle X\rangle_C(\underline{tmp_1}) \\
& \quad ; \ \overline{x'} := \overline{x} + \underline{tmp_1} \cdot 1 \\
& \texttt{else} \\
& \qquad \overline{x'} := \texttt{NULL}
\end{aligned}$$

7.3.7 Virtual function dispatch

Virtual function call is compiled as a lookup into the virtual table for the virtual function entry. If the call is successful, the virtual table holds a corresponding entry giving the pointer to the actual function to run, and the `this` pointer adjustment.

Using detailed accesses to virtual table

$$\begin{aligned}
[\![x'^? := x\texttt{->}_C msig(x^*)]\!] \underset{\text{def.}}{=\!=} \ & \underline{tmp_1} := *_{\texttt{Vptr}}(\overline{x}) \\
& ; \underline{tmp_2} := \texttt{dispfunc}\langle F\rangle_C(\underline{tmp_1}) \\
& ; \underline{tmp_1} := \texttt{dispoff}\langle F\rangle_C(\underline{tmp_1}) \\
& ; \underline{tmp_1} := \overline{x} + \underline{tmp_1} \cdot 1 \\
& ; \overline{x'}^? := (*\underline{tmp_2})(\underline{tmp_1}, \overline{x}^*)
\end{aligned}$$

Using thunk calls Vcm defines the *thunk call* operation to perform all those operations in once.

$$[\![x'^? := x\texttt{->}_C msig(x^*)]\!] \underset{\text{def.}}{=\!=} \overline{x'}^? := \overline{x}\texttt{->}_C msig(\overline{x}^*)$$

7.4 Correctness of the compiler

We prove the correctness of the compiler by proving the preservation of a run-time compilation invariant $s \triangleright s'$ between an execution state s of s++ and an execution state s' of Vcm, finally obtaining Theorem I.14 (p. 156).

We do not achieve full-fledged forward simulation (Theorem B.1 p. 340), as we did not define the notions of initial states for s++ or Vcm. This is due to the initial objects, which are assumed to already exist in both languages. However, once construction is involved to deal with object creation, forward simulation will be made fully possible, as we will show in Section 11.9 (p. 302).

Hypothesis 7.4.1. *For any location ℓ of an existing s++ object of n cells of type D, there exists a block $b(\ell)$ in the Vcm memory state, of size at least $n \cdot \mathsf{size}_D$.*

The b function is assumed to be injective: two distinct complete objects map to distinct memory blocks.

Let $s = (st, stl, e, \mathcal{K}, \mathcal{G})$ be a s++ state, and $s' = (st', stl', e', \mathcal{K}', \mathfrak{M})$ be a corresponding Vcm state.

Invariant 7.4.1 (s++-to-Vcm Invariant). *The invariant $s \rhd s'$ is split into several parts:*

- *The statement and the statement list of s' are compiled from s*
- *The s++ object store is constant*
- *An invariant $v \rhd_{\mathsf{Val}} v'$ holds between the values of s++ variables and the values of their corresponding Vcm variables.*
- *An invariant $\mathcal{K} \rhd_{\mathsf{Stack}} \mathcal{K}'$ relates the continuation stacks.*
- *An invariant $\mathcal{G} \rhd_{\mathsf{global}} \mathfrak{M}$ relates the global state with the concrete memory state.*

$$\frac{\mathcal{G}.\mathsf{LocType} = \mathsf{LocType} \qquad \begin{array}{c} st' = [\![st]\!] \qquad stl' = \mathsf{map}[[\![\cdot]\!]](stl) \\ \forall x : e(x) \neq \bot \Rightarrow e(x) \rhd_{\mathsf{Val}} e'(\overline{x}) \end{array} \qquad \mathcal{K} \rhd_{\mathsf{Stack}} \mathcal{K}' \qquad \mathcal{G} \rhd_{\mathsf{global}} \mathfrak{M}}{(st, stl, e, \mathcal{K}, \mathcal{G}) \rhd (st', stl', e', \mathcal{K}', \mathfrak{M})}$$

7.4.1 Values

Values are related by the $\rhd_{\mathsf{Val}}$ relation, such that:

- A s++ built-in value is unchanged in Vcm:

$$\frac{}{\mathit{Builtin} \rhd_{\mathsf{Val}} \mathit{Builtin}}$$

- A valid pointer to a s++ subobject is related to a concrete pointer to the memory block corresponding to the complete object, under the offset corresponding to the generalized subobject:

$$\frac{\mathsf{LocType}(\ell) = D[n] \qquad D[n] \prec\!\langle p \rangle\!\mapsto C}{(\ell, p) \rhd_{\mathsf{Val}} (b(\ell), \mathsf{off}_D(p))}$$

7.4.2 Continuation stack

An invariant $K \rhd_{\mathsf{Stackframe}} K'$ holds frame by frame:

$$\frac{}{\epsilon \rhd_{\mathsf{Stack}} \epsilon} \qquad \frac{K \rhd_{\mathsf{Stackframe}} K' \qquad \mathcal{K} \rhd_{\mathsf{Stack}} \mathcal{K}'}{K :: \mathcal{K} \rhd_{\mathsf{Stack}} K' :: \mathcal{K}'}$$

Statement blocks For a stack frame corresponding to an enclosing block, the Vcm statement list is compiled from the s++ statement list:

$$\frac{}{\mathsf{Block}(stl) \rhd_{\mathsf{Stackframe}} \mathsf{Block}(\mathsf{map}[[\![\cdot]\!]](stl))}$$

Call frames For a stack frame corresponding to a function caller, the Vcm list of statements to execute upon function return is compiled from its s++ counterpart. The values of s++ variables in the enclosing environment are matched in Vcm.

$$\frac{\forall x : e(x) \neq \bot \Rightarrow e(x) \rhd_{\mathsf{Val}} e'(\overline{x})}{\mathsf{Callframe}(x^?, stl, e) \rhd_{\mathsf{Stackframe}} \mathsf{Callframe}(\overline{x}^?, \mathsf{map}[\![\cdot]\!](stl), e')}$$

7.4.3 Memory

The invariant $\mathcal{G} \rhd_{\mathsf{global}} \mathfrak{M}$ between the s++ global state and the CVcm memory state is composed of two parts:

Field values For any complete object, the values of all its scalar fields are stored in concrete memory.

$$\frac{\begin{array}{c}\mathsf{LocType}(\ell) = D[n] \\ D[n] \prec\!\langle p \rangle\!\succ\!\to C \qquad f = \texttt{scalar}\ T\ t \in \mathcal{F}(C) \qquad \mathcal{G}.\mathsf{FieldValue}((\ell, p), f) = v \neq \bot\end{array}}{\exists v' : \mathsf{load}(\mathfrak{M}, [\![T]\!], b(\ell), \mathsf{off}_D(p) + \mathsf{foff}_C(f)) = v' \neq \bot \wedge v \rhd_{\mathsf{Val}} v'}$$

Whenever a field is written, this invariant is preserved thanks to Theorem I.3 (p. 115).

Dynamic type data Let ℓ be a complete object of type D. For any generalized subobject p of D of static type C such that class C is dynamic, if p can be written $p = (\alpha, i, \sigma)$ so that $D[n] \prec\!\langle \alpha \rangle\!\overset{\mathcal{A}}{\succ\!\to} D'[n'] \prec\!\langle (i, \sigma) \rangle\!\overset{\mathcal{CI}}{\succ\!\to} C$, such that p is not a primary base subobject of another object (i.e. $\sigma = (h, l) = (h, \mathsf{reducePath}(l))$), then it contains a pointer to the virtual table corresponding to the subobject σ of D'.

$$\frac{\begin{array}{c}\mathsf{LocType}(\ell) = D[n] \qquad p = (\alpha, i, \sigma) \\ \sigma = (h, l) \qquad l = \mathsf{reducePath}(l) \qquad D[n] \prec\!\langle \alpha \rangle\!\overset{\mathcal{A}}{\succ\!\to} D'[n'] \prec\!\langle (i, \sigma) \rangle\!\overset{\mathcal{CI}}{\succ\!\to} C \qquad \mathsf{isDynamic}(C)\end{array}}{\mathsf{load}(\mathfrak{M}, \texttt{Vptr}, b(\ell), \mathsf{off}_D(p)) = \&(D', \sigma)}$$

Whenever a field is written, this invariant is preserved thanks to Theorem I.4 (p. 119).

7.4.4 Invariant preservation

The correctness of the compiler is based on the following theorem ensuring that the invariant is preserved:

Theorem I.14 (Correctness of the s++-to-Vcm compiler). *The run-time compilation invariant $\rhd$ is preserved during execution: each s++ execution step corresponds to a finite number of Vcm execution steps.*

$$\forall s_1, s_2, s_1{}' : s_1 \to s_2 \wedge s_1 \rhd s_1{}'$$
$$\Rightarrow \exists s_2{}' : s_1{}' \overset{+}{\to}{}' s_2{}' \wedge s_2 \rhd s_2{}'$$

$$\begin{tikzcd}
s_1 \arrow[r, "\mathfrak{e}^?"'] \arrow[d, dash, "\rhd"] & s_2 \arrow[d, dotted, dash, "\rhd"] \\
s_1{}' \arrow[r, dotted, "+", "\mathfrak{e}^?"'] & s_2{}'
\end{tikzcd}$$

Proof. We sum up the theorems and lemmata used for the proof of invariant preservation for the most interesting steps:

s++ execution step	**Proof case**	**Theorem used**
Scalar field write (s++-field-scalar-write, p. 86)	Success: alignment	Theorem I.1 (p. 110)
	Success: in bounds	Theorem I.2 (p. 112)
	Good variable property wrt. fields	Theorem I.3 (p. 115)
	Good variable property wrt. dynamic type data	Theorem I.4 (p. 119)
Pointer equality test (s++-ptreq, p. 87)	Pointers of non-empty class type	Theorem I.7 (p. 122)
	Pointers of empty class type	Theorem I.8 (p. 124)
Static cast (s++-statcast, p. 88)	Access to virtual base	Theorem I.11 (p. 147)
Dynamic cast (s++-dyncast, p. 89)		Theorem I.12 (p. 148)
Virtual function call (s++-virtual-funcall, p. 91)		Theorem I.13 (p. 150)

□

Chapter 8

Discussion

In this chapter, we give a technical overview of our Coq development. Then, we position our work among earlier work about C++ formal semantics, and object layout. Then, we compare with other languages featuring multiple inheritance. Finally, we investigate perspectives for extending our formalism.

8.1 The Coq development

To design the semantics of s++ and Vcm, we mostly use inductive types, rather than an executable semantics. However, for well-formed class hierarchies, predicates such as knowing whether a class is dynamic are decidable, and sets such as virtual function dispatch candidates are computable in a finite amount of time, which is actually needed to populate virtual tables during compilation. It should therefore be possible to construct executable semantics equivalent to ours.

Our sufficient conditions are grouped into a per-class record, so that, to prove that a class hierarchy compliant to them, it suffices to construct a finite map associating each class with a proof that its layout parameters (data size, total size, and component offsets) make the layout conditions hold. The Coq development turns out to require an overall 23,000 lines of Coq, detailed as follows:

Theories	**Specs loc**	**Proofs loc**
Class hierarchies	996	1308
Well-formed hierarchies	283	1794
Layout conditions	1550	5726
Common Vendor ABI algo	729	3681
Optimized algorithm	689	3522
Simplified compiler	771	2061
Total	**5018**	**18091**

However, those figures correspond to the Coq development [70] of our POPL 2011 article [72], which featured no functions (the use of dynamic type data and virtual tables was illustrated only by dynamic casts and accesses to virtual bases). Moreover, virtual table typing was not developed: we naively forgot to distinguish accesses to the same virtual base from instances of two classes that do not share their virtual tables. Furthermore, we used a simplified memory model, with only one block containing all values.

In fact, we wrote and proved a more general compiler including C++ object construction and description, described in Section 11.8 (p. 293). The proofs presented in this part are an excerpt of this compiler, describing the features unrelated to C++ object construction and destruction, and readapted to the formalism of [72] for virtual tables.

8.2 Related work

8.2.1 Formalizations of C++ multiple inheritance

Our work is based on the formalism of Rossie et al. [74], an algebraic model for C++ multiple inheritance and subobjects. On top of this model, Wasserrab et al. [85, 84] formalized in Isabelle an operational semantics for C++ casts and virtual function dispatch. Those two works leave aside concrete object layout.

Chen [20] proposed a typed intermediate language for compiling C++ multiple inheritance. This work formally describes which pointer adjustments are necessary for casts and virtual function calls (including the `this` pointer adjustments), and introduces an abstract model of virtual tables. However, this work leaves largely unspecified the concrete object layout; moreover, the thorny issue of object identity is not addressed.

Luo et al. [56] formalized a separation logic to allow high-level reasoning about field accesses under C++ multiple inheritance, based on a field resolution algorithm close to the work by Ramalingam et al. [69]. However, those two works do not address concrete object layout. Moreover, Luo et al. restrict their work to non-virtual inheritance only, with an abstract storage model, and defined the semantics of field access through substitution.

Norrish [64] models the semantics of C++ in a machine-checked formalization in HOL4. This work reuses the formalization of subobjects by Wasserrab et al. [85]. However, it is built on a low-level memory model down to the level of bytes. It supports two kinds of pointers: pointers to scalar data under the form of a memory address, and pointers to inheritance subobjects under the form (a, σ) of the memory address a of a most-derived object along with the inheritance path σ of the subobject. Scalar or structure field selection from such a pointer to a subobject is made through a `subobj_offset` function computing the memory offset of a subobject within its most-derived object (used similarly as soff), and a function `lookup_offset` returning the memory offset of a field within an object (used similarly as foff). As such, the notion of subobject offsets is directly part of the abstract semantics. However, the semantics lacks properties about those `subobj_offset` and `lookup_offset` functions, left “under-specified”, arguing that “there is no specification of how base sub-objects are laid out”. We believe that our work completes this semantics by suggesting to equip those functions with the essential properties of field separation and subobject identity.

8.2.2 Concrete object layout

Layout algorithms have been formally verified in the simpler world of C structures. Tuch [83] axiomatized a field separation property of structure layout and used it in a separation logic able to verify low-level system C code. The CompCert Clight formal semantics of Blazy and Leroy [17] defines a simple structure layout algorithm; the field separation, field alignment and prefix compatibility properties were mechanically verified.

However, to the best of our knowledge, no C++ object layout algorithm so far has been reported as formally verified. Stroustrup [30] extensively discusses object layout algorithms found in earlier C++ compilers such as `cfront`. Sweeney and Burke [81] developed a formalism to characterize when compiler artifacts (such as virtual tables) are needed to support run-time semantics of C++ multiple inheritance. In practice, their work influenced design choices in real-world C++ compilers such as IBM Visual Age C++ 5.

Our layout algorithms do not prescribe any explicit implementation for virtual function dispatch. However, our compiler tackles, in an abstract fashion, the special case of virtual tables, used by most present-day compilers such as GNU GCC. We believe that our compiler can be easily reinterpreted under different viewpoints to model other possible implementations for virtual function dispatch, not necessarily based on virtual tables, but always requiring dynamic type data, as shown by an extensive survey by Driesen et al. [29].

Empty base optimization In [30, p. 164], Stroustrup explicitly stated that "objects of an empty class have a nonzero size". An unwanted consequence of this requirement was that no empty base optimization was possible at the time of `cfront`. This requirement was relaxed and clarified in the C++ Standard [42, § 10.3] to allow such optimizations, so that they are finally required by the Common Vendor ABI for Itanium [22].

Our CCCPP optimized layout algorithm featuring optimized storage for fields of empty class types, is based on observations by Myers [63]. He credited Jason Merrill for the possibility of optimizing empty member subobjects.

8.3 Application to other languages with inheritance

Our data layout algorithms obviously apply to all languages with single inheritance, such as Java or *C♯*: in such cases, the base class of a class can always be chosen as the primary base class as soon as it is dynamic. Therefore, an inheritance subobject holds at most one memory zone for dynamic type data. Consequently, since all inheritance subobjects are primary, the subobject identity requirement becomes trivial.

Additionally, a Java or *C♯* class can inherit from several *interfaces*. However, an interface is a very special class: it has no field, and none of its methods has any body. Thus, implementations do not really consider them as classes: they do not require subobjects, and the mechanisms of interface dispatch (calling a function declared in an interface on an instance of a class implementing an interface) are often different. For instance, Alpern et al. [12] propose an interface dispatch algorithm where each class is associated with a positive integer, the type identifier, such that its divisors actually represent the interfaces implemented by the class.

Simula only features single inheritance, but Krogdahl proposes an implementation of multiple inheritance and outlines a layout algorithm that can easily be expressed in our framework. [48]. However, this work explicitly excludes shared inheritance.

Eiffel [59] features multiple inheritance. But it does not feature the subobject identity principle: the notion of repeated or shared inheritance is not defined at the level of classes, but at the level of each *feature* (equivalent to methods or virtual functions). In practice, feature dispatch is done by a dictionary approach: the dynamic type data actually stores the dispatch key. Such an implementation paves the way to dispatch optimizations without having to use any virtual table [87]. More generally, this validates the approach of our formalization of object layout, where the actual nature of the dynamic type data is left unspecified and can be applied to various virtual function dispatch mechanisms such as those covered by Driesen's survey [29].

8.4 Future work

Our formalization of C++ object layout may be extended in a number of directions, either theoretical in terms of the abstract semantics for C++, or practical in terms of layout optimizations.

8.4.1 Extending the semantics of s++

8.4.1.1 Scalar arrays

Our work features arrays, but only considers structure arrays. Indeed, arrays of scalars can be simulated through arrays of structures having only one scalar field. For instance[1]:

```
struct A {
  int t[18];
};
```

can be simulated with:

```
struct Sint {
  int value;
};
struct A {
  Sint t[18];
};
```

This point of view has no significant impact on performance[2]: accessing the field of a structure from a structure array yields only one memory access, as many as accessing a field from an array of scalars.

8.4.1.2 Towards a more realistic semantics of function dispatch

Implicit argument cast Our formal semantics dispatches virtual functions by expecting an exact type match for function arguments. In C++, however, this is not always the case. Consider the following example:

```
struct B    {};
struct D: B {};
struct X {
  virtual void f(B* b);
};
struct Y: X {
  virtual void f(D* d);
};
```

1. This simulation pattern works for all scalar types: not only for built-in types, but also for pointer types. A generic pattern for scalar types would use a template. However, templates are out of the scope of our work.

2. As long as `Sint` has only a default constructor, which may be ignored at execution. However, constructor inlining is out of the scope of our work.

```
Y y;
D d;
main () {
  X* x = static_cast<X*>(&y);
  x->f(&d);
}
```

Wasserrab et al. [85] correctly address this issue and make the above example call f of X by first implicitly casting its argument from D to B.

On the contrary, we chose not to support such implicit casts in the semantics of s++, thus requiring the programmer to provide the function signature with the exactly right argument types.

Indeed, function dispatch occurs in two steps: first the non-virtual function selection, then the choice of the final overrider. The argument types are determined by the function found upon the non-virtual selection. We argue that this first step can be resolved at compile time. Thus, s++ can be considered as an intermediate language to which a compiler would have first determined the argument types and inserted the appropriate static casts before the call. From this point of view, nothing needs to be changed in the semantics of s++, and the above program would become:

```
main () {
  X* x = static_cast<X*>(&y);
  B* b = static_cast<B*>(&d);
  x->f(&b);
}
```

Covariant return Wasserrab [84] extended his earlier work [85] by adding *covariant return functions*, thus making the return type depend on function dispatch. Consider the following example:

```
struct B    {};
struct D: B {};
struct X {
  virtual B* f();
};
struct Y: X {
  virtual D* f();
};
Y y;
main() {
  X* x = static_cast<X*>(&y);
  B* b = x->f();
}
```

Here, C++ dispatches f to the function of Y returning D instead of B. It follows that the *return value* must be cast again from D to the return type B expected by the caller. However, this adjustment is only known at run time. We can see two ways of performing this adjustment:

– Either explicitly distinguish functions with names depending on the return type, and provide wrapper functions. That is, for the above example, the function f of Y refining the return type to D would be distinguished from a wrapper function returning actually B:
  ```
  struct X {
    virtual B* fB();
  };
  struct Y: X {
    virtual D* fD();
    virtual B* fB() {
      D* d = this->fD();
      B* b = (d == null ? static_cast<B*>(d) : null);
      return b;
    };
  };
  Y y;
  main() {
    X* x = static_cast<X*>(&y);
    B* b = x->fB();
  };
  ```
  Then, virtual function dispatch may be performed as usual. One advantage of this solution is that it may be performed at the abstract level, with no modification required in our semantics. But in practice, this would cause expensive code duplication and increase the sizes of virtual tables. Moreover, this could preclude optimizations where covariant return casts would compile to zero-offset adjustments (especially if B is a primary base of D).
– Or adapt the semantics of s++ following [85]. The dispatch has to be performed independently of return type (i.e. return type must no longer be part of the function signature), and further information to the Callframe stack frame has to be added: the actual return type of the callee, and the return type expected by the caller, so that the cast may be performed upon function return. Then, from an implementation point of view, the return type adjustment offset would be provided by virtual tables. In practice, such an adjustment, if non-trivial, prevents thunks from being tail-called.

8.4.1.3 Bit fields

A structure can contain *bit fields*. A bit field is a field whose size in bits is explicitly indicated by the programmer. Then, the compiler usually transforms a contiguous sequence of bit fields into an integer, accessing each field through shift and mask operations.

In our formalization, we did not specify the size unit, so that sizes and offsets can be expressed in bits instead of bytes. However, this may not be enough to fully implement bit fields, as they require specific alignment constraints. Consider the following example:

```
struct A {
  int i: 29;
  int j: 2;
  int k: 3;
};
```

On a 32-bit platform, `i` and `j` can be packed into a single 32-bit integer, and retrieved by shifting and masking from this integer. However, `k` should not be packed adjacent to `j`, otherwise two correctly aligned memory accesses would be required to recover the value of `k`. Our alignment constraints over field offsets would have to be strengthened accordingly.

8.4.1.4 Unions

C++, similarly to C, features unions. When a union data type is used within another structure, its tail padding may be reused. However, the practical benefits are low, as unions are rarely used within a multiple inheritance hierarchy.

8.4.1.5 Accessibility

In our work, we do not consider the accessibility of fields or bases. Indeed, in C++, a programmer can declare a class field hidden from the user, by declaring it `private`. Our formalization does not take this accessibility aspect into account. In practice, access operations involving `private` components could be seen as a two-step access operation: first locate the object or field assuming that everything is `public`, then check on the obtained path whether `private` fields or bases are traversed.

8.4.2 Covering more layout optimizations

8.4.2.1 PODs

Our work does not consider the specificities of POD (*Plain Old Data*), and does not distinguish them from other C++ structures. Roughly speaking, a POD structure is a structure with no inheritance, and no non-POD fields: as such, it is roughly equivalent to a C structure. The C++ Standard [42] mandates that the layout of a POD be compatible with C. Under this requirement, we would have to prove the correctness of compiling structure assignment using `memcpy` raw bitwise memory copy operation. However, we suspect that $\mathsf{nvdsize}_C = \mathsf{nvsize}_C = \mathsf{dsize}_C = \mathsf{size}_C$ for any POD type C could ensure this property, even if C is inherited by another class.

Dawes [25] proposed to amend the C++03 Standard, in order to specify PODs by ensuring compatibility with C without precluding optimizations. This proposal has been accepted to C++11.

8.4.2.2 Concrete virtual table layout

We mostly left abstract the representation of virtual tables. Actually, studying their concrete representation would depend on the concrete implementation of polymorphic operations: virtual function calls on the one hand, dynamic cast on the other hand. Such a formalization would be a further step towards integrating our verified compiler to the existing CompCert.

Virtual function calls In our abstract representation, each virtual function entry features, along with the function pointer, a further field holding its corresponding `this` pointer adjustment offset. This requires the caller to perform the adjustment by reading this field before accessing the actual function pointer and performing the call.

Many C++ compilers such as GNU GCC use a different approach based on *adjusting entry points*: the virtual function entry holds a pointer to a function, called *thunk*, which adjusts the `this` pointer before transferring control to the actual function body. In practice, the latter transfer, however, must be performed with extreme care: a full-fledged function call would worsen performances [54]. So we would have to support functions with multiple entry points. But we would face the following practical issues:

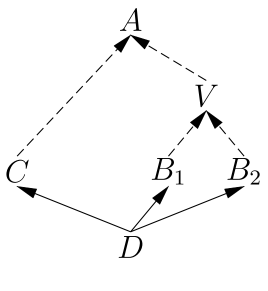
- not all platforms support multiple-entrypoint functions
- in particular, CompCert, meant to be platform-independent (at least for front-end), does not. One solution could be to use tail calls, but CompCert imposes limitations on their use (e.g. number of arguments)
- Intricacy between front-end and back-end: the order of entry points heavily relies on hardware properties to avoid cache misses (prediction of the right entry point). We are not aware of the heuristics to ensure that.

Dynamic cast and run-time type identification Our formalization left mostly abstract the implementation of dynamic cast: we only rely on a single virtual table entry for each base. However, in practice, most compilers resolve dynamic casts thanks to data related to the dynamic type of the object, namely *run-time type identification* (RTTI), which is part of the virtual table. RTTI involves the C++ `typeinfo` class; such objects have to be automatically generated and populated with sound information at the beginning of the program. So we would have to formalize this step not only at the level of the compiled Vcm code, but also as early as in the formal semantics of s++.

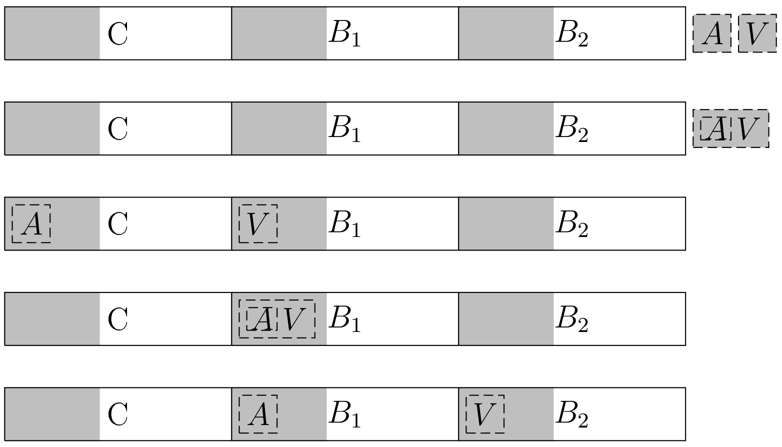
8.4.2.3 Virtual primary bases

Our implementation allows a class to share its dynamic type data only with a non-virtual primary base. However, the Common Vendor ABI [22] extends this sharing to virtual bases, as long as they are *nearly empty*: they have no fields, and they have no base except their primary base and empty bases. In practice, such a virtual base is similar to Java interfaces, without fields, but with method declarations. However, this optimization breaks the scheme of laying out the non-virtual parts of an object separately from the virtual parts. Moreover, nearly empty virtual base may appear several times in an object. Thus it is necessary to choose which will be the corresponding subobject within the most-derived object. We do not know how to solve this layout ambiguity. Consider the following example:

```
struct A { virtual void f(); };
struct V:  virtual A       {};
struct C:  virtual A       {};
struct B1: virtual V       {};
struct B2: virtual V       {};
struct D:  C, B1, B2       {};
```



Then, the following figure depicts several layouts where virtual primary bases may be laid out within the dynamic type data of derived classes. In practice, such issues have brought the development consortium of the Common Vendor ABI to finally consider this optimization as an “error in the design” of the ABI.



Moreover, virtual primary base optimization raises an issue with dynamic cast:

```
#include <cassert>

struct V {
  virtual void f();
};
struct A: virtual V {};
struct B: A {};
struct C1: B {};
struct C2: B {};
struct D: C1, C2 {
  virtual void f();
};

D d;
main () {
  C1* c1 = static_cast<C1*>(&d);
  A* a1 = static_cast<A*>(c1);
  V* v = static_cast<V*>(a1);
  assert(dynamic_cast<B*>(v) == NULL);
  assert(dynamic_cast<B*>(a1) != NULL);
}
```

Both assertions succeed, even though `v` and `a1` share the same pointer to virtual table. This shows that the dynamic cast operator also depends on the source class.

However, in C++ such program design is progressively discouraged in favor of templates.

8.4.2.4 Bidirectional object layout

In our formalism, an object contains its bases and fields at positive offsets from its starting point. However, it might be also possible to store data at negative offsets. Moreover, the dynamic type data could then be shared with two primary bases. Such data layout is called *bidirectional object layout* and has been proposed by Gil et al. [35]. Such an optimization could be arguably space and time-efficient. This optimization was refined by Gil et al. [36] to a “language-independent” object layout algorithm. However, the latter work explicitly excludes C++-style non-virtual multiple inheritance, which is not rare in practice.

Moreover, bidirectional object layout heavily assumes that the compiler knows in advance the complete hierarchy: it is impossible to perform separate compilation, or even to use dynamically linked libraries. Thus, it is not suitable to most realistic C++ compilers.

Part II

Verification of C++ object construction and destruction

Chapter 9

The semantics of C++ construction and destruction

In this chapter, we define a language, called κ++, to formally specify how C++ objects are constructed and destructed.

9.1 Overview of the construction and destruction process

In C++, the construction of an object allows it to initialize its fields and bases. But, conjointly with destruction, construction offers a way to manage resources related to the lifetimes of objects, as explained in the tutorial of Section 2.5 (p. 55).

9.1.1 Construction

Construction of an object starts when the *constructor* is called with its arguments. An object is constructed when the body of the constructor exits: at this moment starts its *lifetime*.

Construction must follow these two basic principles:
– An object requires prior construction of all of its subobjects
– An object must not be constructed more than once

9.1.1.1 Non-virtual inheritance only

If a class has no virtual base, then the construction of an instance is straightforward:

1. First construct the direct non-virtual bases, in declaration order
2. Then construct the fields, in declaration order
3. Finally run the constructor body

To construct a direct non-virtual base, the constructor of the object first has to compute the arguments to pass to the constructor of the direct non-virtual base. This step corresponds to running the *initializer* for the direct non-virtual base. The construction only starts when the arguments are passed to the constructor.

However, even though arguments are passed, the body of the constructor is not immediately run: the construction of non-virtual bases, and of fields, must first occur.

For instance, consider the following class hierarchy:

```
struct B1 { B1 (int i) {} };
struct B2 { B2 (int j) {} };
struct C: B1, B2 { C (int i1, int i2) : B1 (i1 - i2), B2(i2 - i1) {} }
```

Suppose that an instance of C is requested to be constructed using $C(18, 42)$. Then:

1. Arguments 18 and 42 are passed as i_1 and i_2 to the constructor `C(int, int)`
2. The construction of the non-virtual base B_1 is requested using $B_1(18 - 42)$
3. The value of the argument $18 - 42$ is computed (*initializer* phase)
4. The computed value -24 is passed as i to the constructor `B1(int)`
5. B_1 has no bases or fields, so the body of its constructor is run
6. Then, back to C, the construction of the non-virtual base B_2 is requested using $B_2(42-18)$
7. The value of the argument $42 - 18$ is computed (*initializer* phase)
8. The computed value 24 is passed as j to the constructor `B2(int)`
9. B_2 has no bases or fields, so the body of its constructor is run
10. Then, back to C, there are no more bases or fields to construct, so the body of the cosnstructor C is run

To construct a structure array field requires the construction of the object of each array cell, in increasing index order.

9.1.1.2 Virtual inheritance

Consider the following class hierarchy:

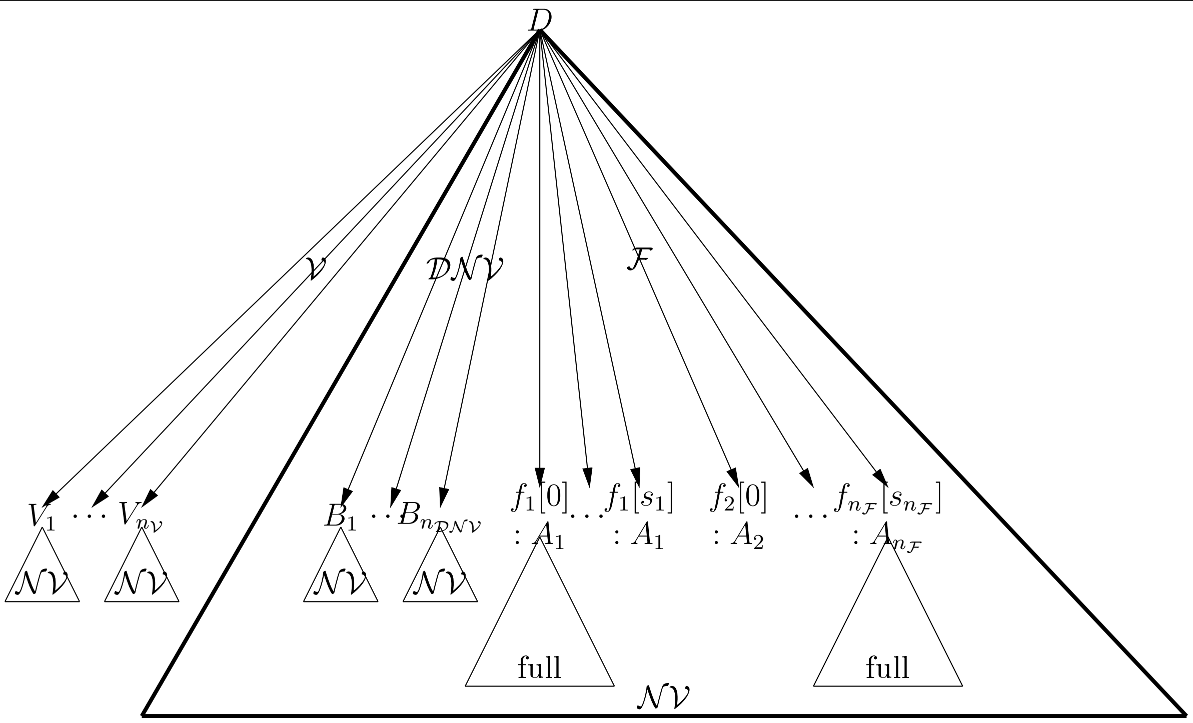
```
struct A {};
struct B0 {};
struct B1: virtual A {};
struct B2: virtual A {};
struct C: B0, B1, B2 {}
```

If A were to be constructed as if it were a non-virtual base, following the above protocol, then A would have been constructed twice. This is prevented by the C++ standard.

In fact, virtual bases are constructed independently before any non-virtual part of the most-derived object. For instance, in the hierarchy above, the A subobject within C is constructed even before B_0, even though B_0 has no virtual base.

However, this is not precise enough. Indeed, consider the following hierarchy:

```
struct A {};
struct B0 {};
struct B1: virtual A {};
struct B2: virtual A {};
struct C: B0, virtual B1, virtual B2 {}
```


D
$\mathcal{V}$
$\mathcal{DNV}$
$\mathcal{F}$
$V_1 \cdots V_{n_\mathcal{V}}$
$B_1 \cdot\cdot B_{n_{\mathcal{DNV}}}$
$f_1[0] : A_1$
$f_1[s_1] : A_1$
$f_2[0] : A_2$
$f_{n_\mathcal{F}}[s_{n_\mathcal{F}}] : A_{n_\mathcal{F}}$
$\mathcal{NV}$
full
full
$\mathcal{NV}$

Class D is assumed to have (direct or indirect) virtual bases $V_1, \ldots, V_{n_\mathcal{V}}$, direct non-virtual bases $B_1, \ldots, B_{n_{\mathcal{DNV}}}$ and structure array fields $f_1 : A_1[s_1], \ldots f_{n_\mathcal{F}} : A_{n_\mathcal{F}}[s_{n_\mathcal{F}}]$. The tree represented here is the "full" tree of all the subobjects of a most-derived object of type D. The subtree inside the thick $\mathcal{NV}$ triangle represents the non-virtual part of D, the only part considered for a D object that is a base-class subobject of another object, thus excluding the virtual bases of D.

Figure 9.1: A tree representation of the subobjects of a class, such that a depth-first left-to-right traversal exactly yields the subobject construction order.

Then, A still has to be prevented from being constructed twice. However, A must be constructed before B_1 and B_2 because A is a base of B_1 and B_2.

In fact, when constructing a most-derived object, all its virtual bases are listed in a certain order $\prec^\mathcal{V}_C$ guaranteeing that if a virtual base A of C is actually a virtual base of B, then $A \prec^\mathcal{V}_C B$ and A is constructed before B. The Standard prescribes such an order, called *inheritance graph order*. We shall see further down that we have modeled this order (*Definition* 10.3.4 p. 214) and we have proved that it meets this requirement (LEMMA 10.3.7 p. 214).

Then, to construct a most-derived object of type C, the standard mandates the following process:

1. Construct the non-virtual parts of the virtual bases of C, following the order $\prec^\mathcal{V}_C$
2. Construct the non-virtual part of the most-derived object

Then, constructing the *non-virtual* part of a subobject follows the protocol mentioned before as if there were no virtual bases.

The order of construction of subobjects is summarized on Figure 9.1 (p. 173).

9.1.2 Destruction

Destruction of an object follows the only one principle: if two subobjects constructed in order, then they are destructed in the exact reverse order. In more detail:

- the cells of an array of size n are destructed from cell $n-1$ down to cell 0
- a most-derived object has its non-virtual part destructed first, before its virtual bases following the order $\succ^{\mathcal{V}}_{C}$ (reverse of $\prec^{\mathcal{V}}_{C}$)
- to destruct the non-virtual part of an inheritance subobject: the destructor is run first, then the fields are destructed in the reverse declaration order, then the direct non-virtual bases in the reverse declaration order.

9.2 Syntax of κ++

The formal model presented in this chapter aims at capturing the essence of object construction and destruction semantics. Like s++ (Section 4.2 p. 80), the language we consider in this chapter[1] features multiple inheritance (both shared and repeated), virtual functions (a.k.a "methods"), static and dynamic casts, and scalar or structure nonstatic data members (a.k.a "fields"). Additionally, this chapter introduces constructors and destructors, and block-scoped objects: an object is created upon entering a statement block, and destructed upon block exit. For this reason, we call our language κ++, κ standing for "constructors".

n	$\in \mathbb{N}$	
$op, \ldots$	$\in Op$	Built-in operations
$x, \ldots$	$\in Var$	Variables
$B, C, \ldots$	$\in ClassName$	Classes
$fsig$	$: FieldSig$	Field names
$msig$	$: Method$	Method names
st	$::= x' := op(x^*)$	Built-in operation
	$\mid x' := x$	Assignment between variables
	$\mid$ `if` (x) st_{true} `else` st_{false}	Conditional
	$\mid st_1; st_2$	Statement sequence
	$\mid$ `skip`	Do nothing
	$\mid$ `loop` st	Loop
	$\mid \{st\}$	Statement block
	$\mid$ `exit` n	Exit from n blocks
	$\mid$ `return` $x^?$	Return from virtual function
	$\mid x'^? := sfname(x^*)$	Static function call
	$\mid x'^? := x$`->`C`::`$msig(x^*)$	Non-virtual function call
	$\mid x' := x$`->`$_C fsig$	Field read
	$\mid x$`->`$_C fsig := x'$	Scalar field write
	$\mid x' :=$ `&`$x[x_{\mathsf{index}}]_C$	Array cell access
	$\mid x' := x_1 ==_C x_2$	Pointer equality test
	$\mid x' :=$ `dynamic_cast`$\langle B\rangle_C(x)$	Dynamic cast
	$\mid x' := x$`->`$_C mname(x^*)$	Virtual function call

1. Coq development: theory `Cppsem`.

$\mid x' :=$ `static_cast`$\langle B\rangle_C(x)$	Static cast
$\mid \{C\ x[\mathit{size}] = \{\mathit{ObjInit}^*\}; \mathit{st}\}$	Complete object lifetime
$\mid C(x^*)$	Constructor call (class initializer only)
$\mid \mathsf{initScalar}(x)$	Scalar field initialization

Each *constructor* comes with:

- the variable names of its arguments
- the initializers for direct non-virtual bases
- the initializers for fields
- the initializers for all (direct or indirect) virtual bases, used only for the construction of a most-derived object

For any class B, an *initializer* of a subobject of type B is a statement containing a call to a constructor of B with variables given as arguments. Such a statement allows for initializing variables for constructor arguments, before handing over to the constructor. A scalar field of a class may also have an initializer, then this initializer exits by giving, through the initScalar statement, the variable used to initialize the field.

In our semantics, an initializer ends by handing over to a constructor. In this last step, it cannot pass a reference to a temporary object. Indeed, such a temporary would have to be destructed after returning from the constructor. Our semantics does not allow initializers to perform any additional steps after calling the constructor.

Contrary to C++03 [42], where structure array fields with several cells were necessarily constructed using the default constructor (with no argument), the latest C++11 Standard [43] allows different constructors to be called for each cell. Our κ++ language correctly models this new feature.

$\mathit{ObjInit}$	$::= C\{\mathit{st}\}$	Class object initializer
$\mathit{FieldInit}$	$::= \mathit{fsig}\{\mathit{st}\}$	Scalar field initializer
	$\mid \mathit{fsig}\{\mathit{ObjInit}^*\}$	Structure field initializers (one for each array cell)
Init	$::= \mathit{ObjInit} \mid \mathit{FieldInit}$	
Constr	$::= C(x^*) : \mathit{Init}^*\{\mathit{st}\}$	Constructor
Destr	$::= \sim C()\{\mathit{st}\}$	Destructor

Finally, along with a class hierarchy, a program defines constructors and a *destructor* for each class, each provided with some piece of code. Moreover, a program provides pieces of code for class member functions.

A class member function comes with its argument types and its statement body. Function arguments and return values can be either values of built-in types (integers, floats) or pointers to objects: temporary objects for function arguments must be made explicit in our language, and only pointers to structures may be passed. By the way, this allows for fixing the order of construction of temporaries. However, functions returning structures are not allowed.

StaticFunDef	::=	$(x^*)\{st\}$	Static function
MethodDef	::=	*this*->$(x^*)\{st\}$	Class member function (method) definition

Program =
{

hierarchy	:	*Hierarchy*	; Class hierarchy
staticfuns	:	*StaticFunName* ⤀ *StaticFunDef*	; Static functions
methods	:	*ClassName* × *MethodSig* ⤀ *MethodDef*	; Class method codes
constructors	:	*ClassName* ⤀ *Constr**	; Constructors
destructors	:	*ClassName* ⤀ *Destr*	; Destructors
main	:	*st*	; Entry point

}

9.3 Semantic elements

We formalized a small-step style semantics for C++ object construction and destruction, with a continuation stack to precisely model each step of computation.

9.3.1 Construction states

Subobjects A constructor body (resp. a destructor) may use **virtual functions** of its class or one of its bases, under the following principle: As long as the body of the constructor (resp. destructor) is still running, the overriding of its virtual functions behaves as if the object being constructed were the most-derived object.

In fact, initializers for fields may also use the virtual functions of the object being constructed, as well as their constructors (indirectly, when a pointer or a reference to the object being constructed is used within the field constructor). Then, the virtual function overriding mechanism happens "as if" fields were constructed within the body of the constructor. However, virtual functions are prevented from use as long as bases have not been constructed.

To formalize this ability, we introduce the notion of the *construction state* of a subobject, to mark the precise steps of construction or destruction undergone by the subobject. However, this notion has to be interpreted in two different ways, depending on whether the considered object is a most-derived object or an inheritance subobject.

For a most-derived object:

- Unconstructed: Construction has not started yet
- StartedConstructing: The construction of inheritance bases has started, but not the fields
- BasesConstructed: The bases are wholly constructed. Now starting the construction of fields, and the constructor body.
- Constructed: The constructor body has left, and the destruction body has not yet entered
- StartedDestructing: The destructor body has entered, and the fields are being destructed
- DestructingBases: The fields have been wholly destructed. Bases are being destructed
- Destructed: All bases and fields have been destructed

For other inheritance subobjects:

- Unconstructed: Construction of the *non-virtual part* has not started yet (virtual bases may have been already constructed)

- StartedConstructing: The construction of *non-virtual* inheritance bases has started, but not the fields
- BasesConstructed: The bases are wholly constructed. Now starting the construction of fields, and the constructor body.
- Constructed: The constructor body has left, and the destruction body has not yet entered
- StartedDestructing: The destructor body has entered, and the fields are being destructed
- DestructingBases: The fields have been wholly destructed. *Non-virtual bases* are being destructed
- Destructed: All fields and *non-virtual bases* have been destructed

In other words, for an inheritance subobject different from the most-derived object, the construction state is only relative to its *non-virtual part*. This is due to the fact that virtual bases are constructed separately from non-virtual bases: a (virtual or non-virtual) base may have been destructed before its virtual bases.

Then, the *lifetime* of an object can be defined as the time interval when its construction state is exactly Constructed. However, virtual functions may already be used as soon as the construction state is at least BasesConstructed, and strictly before DestructingBases. In that case, for the purpose of function overriding, the object is considered as the most-derived object as long as the construction state is not Constructed.

Consider the following example:

```
struct A             {virtual void f ();};
struct B1: virtual A {};
struct B2: virtual A {virtual void f ();};
struct C: B1, B2 {}
```

Consider an instance of C. Then, during the execution of the constructor body of its base B_2, the corresponding B_2 subobject is BasesConstructed, and the virtual function f can be executed from within the constructor body of B_2.

***Definition* 9.3.1.** *We define a successor function*[2] S *on construction states, such that:*

$$\begin{array}{lllll} & & & & \mathsf{Unconstructed} \\ \overset{\mathsf{S}}{\mapsto} & \mathsf{StartedConstructing} & \overset{\mathsf{S}}{\mapsto} & \mathsf{BasesConstructed} \;\overset{\mathsf{S}}{\mapsto} & \mathsf{Constructed} \\ \overset{\mathsf{S}}{\mapsto} & \mathsf{StartedDestructing} & \overset{\mathsf{S}}{\mapsto} & \mathsf{DestructingBases} \;\overset{\mathsf{S}}{\mapsto} & \mathsf{Destructed} \end{array}$$

Then, we define an order $<$ *on construction states, namely the smallest transitive relation such that* $c < \mathsf{S}(c)$.

Fields Similarly, a field can be accessed only if it has been constructed.

We also define the construction state of a scalar field to be one of the following:
- Unconstructed: the field has not yet been constructed, it cannot be accessed yet
- Constructed: the field has been constructed, its value initialized
- Destructed: the field has been destructed, it can no longer be accessed

Similarly, for a structure field:
- Unconstructed: the field has not yet been constructed, it cannot be accessed yet

2. This function is partial, as it is undefined for Destructed

- StartedConstructing: the cells of the array are under construction
- Constructed: all cells of the array are constructed
- StartedDestructing: the cells of the array are under destruction
- Destructed: all cells of the array are destructed

The differences are that:

- when executing the initializer for a scalar field, it cannot be accessed until given its final value, so it is still considered Unconstructed; however, for a structure array field, as different initializers are used for the different cells of the array, the first cell may be accessed from within the initializer of subsequent cells, so there is an observational difference with Unconstructed which really intends that no subobject of the field has started its construction.
- when destructing a scalar field, the semantics allows no specific code to run (a scalar type has no destructor).

9.3.2 Values

κ++ values are exactly the same as in s++ (Section 4.3.1 p. 82): a value can be either a built-in value, a pointer to a subobject (which is a pair of a complete object location and a generalized subobject within this complete object), or a null pointer.

$\ell, \ldots$	$\in \Lambda$	Complete object location
Ptr	$::= (\ell, (\alpha, i, \sigma))$	Pointer to subobject
Val	$::= Builtin$	Value of built-in type
	$\mid Ptr$	Non-null pointer
	$\mid \mathsf{NULL}_C$	Null pointer of C class type

9.3.3 Execution state

An *execution state* of the small-step semantics is the combination of the following parts:

- the *execution point* of the state : code point, or list of objects about to be constructed or destructed
- the *continuation stack* modeling the resumption points on the return from a function, or on the completion of the construction or destruction of a subobject
- the *class types and array sizes* of complete objects
- the *scalar field values.* A field is uniquely identified by its complete object location, the generalized subobject within this complete object defining it, and the field signature.
- the *construction states* of subobjects and fields
- the list of *deallocated objects*

For presentation convenience, a common *global state* groups the complete object types and the scalar field values as in s++, but additionally features the construction states and the list of deallocated objects, so that a κ++ execution state is written as a triple $(S, \mathcal{K}, \mathcal{G})$ where S is the execution point, $\mathcal{K}$ the continuation stack, and $\mathcal{G}$ the global state.

$$
\begin{array}{lcll}
\mathit{Env} & = & x \rightharpoonup \mathit{Val}^{?} & \text{Environment} \\
e & ::= & \mathit{Env} & \\
\mathcal{G} & = & & \\
\{ & & & \\
\qquad \mathsf{LocType} & : & \Lambda \rightharpoonup (\mathit{ClassName} \times \mathbb{N}^{>0})^{?} & ;\ \text{Complete object types} \\
\qquad \mathsf{FieldValue} & : & \mathit{Ptr} \times \mathit{FieldSig} \rightharpoonup \mathit{Val}^{?} & ;\ \text{Scalar field values} \\
\qquad \mathsf{ConstrState} & : & \mathit{Ptr} \rightharpoonup \mathit{ConstrState} & ;\ \text{Construction states of objects} \\
\qquad \mathsf{ConstrState}^{\mathcal{F}} & : & \mathit{Ptr} \times \mathit{FieldSig} \rightharpoonup \mathit{ConstrState} & ;\ \text{Construction states of fields} \\
\qquad \mathsf{dealloc} & : & \Lambda^{\star} & ;\ \text{Deallocated objects} \\
\} & & & \\
\mathit{State} & ::= & (S, \mathcal{K}, \mathcal{G}) & \text{Execution state}
\end{array}
$$

***Notation* 9.3.2.** *Throughout this thesis, we may also use an alternate notation to read a component of a state $s = (S, \mathcal{K}, \mathcal{G})$: we write $\mathsf{ConstrState}_s(\pi) = \mathcal{G}.\mathsf{ConstrState}(\pi)$, and similarly for other components of $\mathcal{G}$.*

Once a complete object is given its class type and array size, such data remains in the store forever, to allow reasoning about the construction states of the subobjects even when they are destructed. Thus, the store alone does not say anything about *deallocated* objects. This is the purpose of providing the state with the list of deallocated objects, so that an object ℓ is allocated if, and only if, its location ℓ is defined in the store, and is not in the list of deallocated objects.

***Definition* 9.3.3 (Object lifetime).** *The* lifetime *of an object (ℓ, p) is the set of all states $(S, \mathcal{K}, \mathcal{G})$ such that $\mathcal{G}.\mathsf{ConstrState}(\ell, p) = \mathsf{Constructed}$.*

This notion of lifetime is consistent with the Standard, except for arrays: the C++03 Standard [42] considers the lifetime of an array to start at allocation and end at deallocation, regardless of the construction and destruction process. Actually, our notion of construction state only covers subobjects with a certain static type, not whole arrays. However, this notion may be fixed in an upcoming C++ Standard to match the array lifetime with the lifetime of its last cell.

9.3.3.1 Execution point

Contrary to s++, the κ++ *execution point* (or *kind*) of a state is not always a statement. It can be of one of the following:

- Executing a statement. This execution point also indicates whether the statement is included in a statement block.
- About to construct a list of (virtual or direct non-virtual) bases, or fields
- About to construct an array cell and all its next neighbors
- About to destruct a list of (virtual or direct non-virtual) bases, or fields
- About to destruct an array cell and all its previous neighbors

S	$::=$	$\mathsf{Codepoint}(st_1, st^*, e, \mathit{Block}^*)$ Executing statement st_1 followed by the list of statements st^*, under variable environment e. Block^* is the list of all blocks enclosing the current statement
	$\mid$	$\mathsf{Constr}(\pi, \mathit{ItemKind}, \kappa, L, e)$ About to construct the list L of the bases or fields of the subobject π. Initializers are to be looked for using constructor κ, and they operate on the variable environment e to pass arguments to their constructors.
	$\mid$	$\mathsf{ConstrArray}(\ell, \alpha, n, i, C, \mathit{ObjInit}^*, e)$ About to construct cells i to $n-1$ of type C, of the array α from the complete object ℓ, using the initializers $\mathit{ObjInit}^*$ to initialize the cells, and e as variable environment to execute the initializers.
	$\mid$	$\mathsf{Destr}(\pi, \mathit{ItemKind}, L)$ About to destruct the list L of bases or fields of the subobject π.
	$\mid$	$\mathsf{DestrArray}(\ell, \alpha, i, C)$ About to destruct cells i down to 0 of type C, of the array α from the complete object ℓ .

$\mathit{ItemKind}$	$::=$	$\mathsf{Bases}(\mathit{BaseKind})$	Construct (or destruct) bases
	$\mid$	Fields	Construct (or destruct) fields
$\mathit{BaseKind}$	$::=$	$\mathsf{DirectNonVirtual}$	Construct (or destruct) direct non-virtual bases
	$\mid$	$\mathsf{Virtual}$	Construct (or destruct) virtual bases
Block	$::=$	$(\ell^?, st^*)$	A block: the automatic object to destruct at block exit, if any, and the remaining statements to execute *after* exiting from the block

9.3.3.2 Continuation stack

A state features a continuation stack to model the pending operations that are to be resumed on the return from a function as in s++, or additionally on the completion of the construction or destruction of a subobject. Each element of this stack, or *stack frame*, represents a resumption point. Such a frame can be:

– remaining statements to execute after returning from a function call
– remaining subobjects to construct/destruct
– pending constructor call after returning from the initializer

However, contrary to s++, enclosing statement blocks are not represented by individual stack frames: they are included in the corresponding stack frames and execution point depending on the context.

$\mathcal{K}$ ::= *Kcode*
| *Kconstruction*

| *Kdestruction*

Kcode ::= $\mathsf{Kcontinue}(\ell, st_1, e, st_2^*, Block^*)$

After the construction of a complete object ℓ, enter the block and execute statement st_1, then, after exiting the block, execute statements st_2^* enclosed by other blocks $Blocks^*$, under variable environment e. Also used when destructing ℓ on block exit, with st_1 the corresponding exit statement, and st_2^* the pending statements of the enclosing block, once the block is exited.

| $\mathsf{Kretcall}(res^?, e, st^*, Block^*)$

On returning from a virtual function call, update the environment e by storing the result (if any) in variable $res^?$, then continue the caller execution with the further statements st^* enclosed by other blocks $Blocks^*$.

Kconstruction ::= $\mathsf{Kconstr}(\pi, \mathit{ItemKind}, B, L, \kappa)$

Used during the execution of a base or scalar field initializer. When it returns, it will give the constructor to construct the base B of subobject π, then the other items L of π will have to be constructed, using constructor κ to retrieve their initializers.

| $\mathsf{Kconstrother}(\pi, \mathit{ItemKind}, B, L, \kappa, e)$

Used during the construction of the base or field B of subobject π. When the constructor body returns, then the other bases or fields L of π will have to be constructed, using constructor κ to retrieve their initializers.

| $\mathsf{Kconstrarray}(\ell, \alpha, n, i, C, \mathit{ObjInit}^*)$

Used during the execution of an initializer for a most-derived object of class type C. When it returns, it will give the constructor to construct the most-derived object at the cell i of the array α of the complete object ℓ. Then, the other cells from $i+1$ to $n-1$ will remain to be constructed, $\mathit{ObjInit}^*$ giving the corresponding initializers.

| $\mathsf{Kconstrothercells}(\ell, \alpha, n, i, C, \mathit{ObjInit}^*, e)$

Used during the construction of the most-derived object of class type C at the cell i of the array α of the complete object ℓ. When the constructor body returns, the other cells from $i+1$ to $n-1$ will remain to be constructed, $\mathit{ObjInit}^*$ giving the corresponding initializers to run under the variable environment e.

Kdestruction ::= $\mathsf{Kdestr}(\pi)$

The body of the destructor for subobject π is running. When it returns, the destruction of the fields and direct non-virtual bases of π will start.

| $\mathsf{Kdestrother}(\pi, \mathit{ItemKind}, f, L)$

Used during the destruction of a base or field B of π. When destruction of B is complete, then the other bases or fields L will have to be destructed

| $\mathsf{Kdestrcell}(\ell, \alpha, i, C)$

Used during the destruction of the *non-virtual part* of the most-derived object of class type C at the cell i of the array α of the complete object ℓ. When the destruction is complete, the virtual bases of this most-derived object will have to be destructed, before the other cells from $i-1$ down to 0.

9.3.4 Initial and final states

As in s++, the program starts by executing the main statement with no allocated object at all. Moreover, the construction states of all subobjects are $\mathsf{Unconstructed}$:

***Definition* 9.3.4.** *The initial state is:*

$$(\mathit{Codepoint}(\mathit{main}, \epsilon, \varnothing, \epsilon), \epsilon, \mathcal{G}_\circ)$$

where:

$$\forall \ell : \mathcal{G}_\circ.\mathsf{LocType}(\ell) = \bot \qquad \forall \pi : \mathcal{G}_\circ.\mathit{ConstrState}(\pi) = \mathsf{Unconstructed}$$

$$\forall \pi, f : \mathcal{G}_\circ.\mathit{ConstrState}^{\mathcal{F}}(\pi, f) = \mathsf{Unconstructed} \qquad \forall \pi, f : \mathcal{G}_\circ.\mathit{FieldValues}(\pi, f) = \bot$$

$$\mathcal{G}_\circ.\mathsf{dealloc} = \epsilon$$

The program ends when the main statement returns after having left all statement blocks:

***Definition* 9.3.5.** *A state $(S, \mathcal{K}, \mathcal{G})$ is* final with return value *i if, and only if, all the following conditions hold:*

$$S = \mathit{Codepoint}(\texttt{return } x, L, e, \epsilon) \qquad e(x) = i \in \mathbb{Z} \qquad \mathcal{K} = \epsilon$$

Note that this definition does not a priori prevent from having some undestructed objects in the global state $\mathcal{G}$ of a final state. However, we shall prove that this is not possible (if there is no free store): such a state would not be reachable from an initial state.

9.4 Semantic rules

The small-step semantics of κ++ is given by the transition relation $\to$ between two transition states, defined in this section.

9.4.1 Structured control and built-in operations

Structured control Most structured control (conditionals, sequences, infinite loops) behaves similarly as in s++. However, since the structure of κ++ states differs, we shall present the corresponding κ++ rules separately below. Indeed, in an execution state of kind $\mathsf{Codepoint}(st, L, e, \mathcal{B})$, st is the statement to run, and L is a pipeline of pending statements *within the same block*, while $\mathcal{B}$ represents the list of pending enclosing blocks. The pipeline is not guaranteed to be executed, in particular if the statement is `exit` or `return`.

$$\frac{e(x) = b \ \in \{\mathsf{true}, \mathsf{false}\}}{\begin{array}{ll} & (\mathsf{Codepoint}(\texttt{if}(x)\ st_{\mathsf{true}}\ \texttt{else}\ st_{\mathsf{false}}, L, e, \mathcal{B}), \ \mathcal{K}, \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(st_b, L, e, \mathcal{B}), \ \mathcal{K}, \ \mathcal{G}) \end{array}} \qquad (\kappa\text{++-if})$$

$$\frac{}{\begin{array}{ll} & (\mathsf{Codepoint}(st_1; st_2, L, e, \mathcal{B}), \ \mathcal{K}, \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(st_1, st_2 :: L, e, \mathcal{B}), \ \mathcal{K}, \ \mathcal{G}) \end{array}} \qquad (\kappa\text{++-seq})$$

$$\frac{}{\begin{array}{rl} & (\mathsf{Codepoint}(\texttt{skip}\ , st :: L, e, \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(st\ , L, e, \mathcal{B})\ \ \ \ \ ,\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}} \quad (\kappa\text{++-skip-cons})$$

$$\frac{}{\begin{array}{rl} & (\mathsf{Codepoint}(\texttt{loop}\ st\ , L, e, \mathcal{B})\ \ \ \ ,\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(st\ , \texttt{loop}\ st :: L, e, \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}} \quad (\kappa\text{++-loop})$$

Built-in operations They are unchanged since s++.

Hypothesis 9.4.1. *The set of values of built-in types Builtin is assumed to contain:*
- *a subset of $\mathbb{Z}$ to model array cell indexes*
- *the Boolean values* true *and* false

$$\frac{\forall i, e(x_i) = v_i \qquad [op](v_1 :: \ldots :: v_n :: \epsilon) \ni (\mathfrak{e}^?, res) \qquad e' = e[x' \leftarrow res]}{\begin{array}{rl} & (\mathsf{Codepoint}(x' := op(x_1, \ldots, x_n)\ , L, e, \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\texttt{skip}\ , L, e', \mathcal{B})\ \ \ \ \ \ \ \ \ ,\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}} \quad (\kappa\text{++-atom})$$

Blocks with no stack objects In this section, we first define the semantics of the blocks that do not define stack objects. Entering such a block embeds the pipeline into a new enclosing block with no stack object. However, contrary to s++, this new block is not added to the continuation stack as a standalone frame, but added to $\mathcal{B}$ within the execution point.

$$\frac{}{\begin{array}{rl} & (\mathsf{Codepoint}(\{st\}\ , L, e, \mathcal{B})\ \ \ \ \ ,\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(st\ , \epsilon, e, (\bot, L) :: \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}} \quad (\kappa\text{++-block-no-obj})$$

`exit` n exits from n blocks. We first define the semantics of exiting from blocks with no stack objects.

$$\frac{}{\begin{array}{rl} & (\mathsf{Codepoint}(\texttt{exit}\ 0\ , L, e, \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\texttt{skip}\ , L, e, \mathcal{B})\ \ ,\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}} \quad (\kappa\text{++-exit-0})$$

$$\frac{}{\begin{array}{rl} & (\mathsf{Codepoint}(\texttt{exit}\ (\mathsf{S}\ n)\ , L, e, (\bot, L') :: \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\texttt{exit}\ n, L', e, \mathcal{B})\ \ \ \ \ \ \ \ \ ,\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}} \quad (\kappa\text{++-exit-S})$$

If there are no more instructions to execute in the block, then the following rule requests automatic exit from the block:

$$\frac{}{\begin{array}{rl} & (\mathsf{Codepoint}(\texttt{skip}\ , \epsilon, e, \mathcal{B})\ \ ,\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\texttt{exit}\ 1\ , \epsilon, e, \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}} \quad (\kappa\text{++-skip-nil})$$

This rule implies that if there are no more instructions to execute at the highest level of the function, i.e. $\mathcal{B} = \epsilon$, then the semantics gets stuck. So, in such cases, the user is mandated to explicitly provide a `return` statement.

9.4.2 Static and non-virtual function call

Similarly to s++, κ++ allows calling static (non-class-member) functions, and calling class member functions in a non-virtual fashion, bypassing virtual function dispatch: in the latter case, there is no implicit cast, and the type of the pointer on which to perform the call must be the class actually defining the function.

$$\frac{\begin{array}{c}\mathsf{staticfuns} f = (\mathit{varg}_1, \dots, \mathit{varg}_n)\{\mathit{body}\} \\ \forall j, e(x_j) = v_j \qquad e' = \varnothing[\mathit{varg}_1 \leftarrow v_1] \dots [\mathit{varg}_n \leftarrow v_n]\end{array}}{\begin{array}{lll} & (\mathsf{Codepoint}(x\text{->}C\text{::}f(x_1 \dots x_n), L, e, \mathcal{B}), & \mathcal{K}, \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathit{body}, \epsilon, e', \epsilon) \quad , & \mathsf{Kretcall}(x, L, e, \mathcal{B}) :: \mathcal{K}, \ \mathcal{G})\end{array}}$$

(κ++-static-funcall)

$$\frac{\begin{array}{c}e(x) = \pi \qquad \mathcal{G} \vdash \pi : C \qquad \mathsf{methods}(C, f) = \mathit{this}\text{->}(\mathit{varg}_1, \dots, \mathit{varg}_n)\{\mathit{body}\} \\ \forall j, e(x_j) = v_j \qquad e' = \varnothing[\mathit{varg}_1 \leftarrow v_1] \dots [\mathit{varg}_n \leftarrow v_n][\mathsf{this} \leftarrow \pi]\end{array}}{\begin{array}{lll} & (\mathsf{Codepoint}(x\text{->}C\text{::}f(x_1 \dots x_n), L, e, \mathcal{B}), & \mathcal{K}, \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathit{body}, \epsilon, e', \epsilon) \quad , & \mathsf{Kretcall}(x, L, e, \mathcal{B}) :: \mathcal{K}, \ \mathcal{G})\end{array}}$$

(κ++-non-virtual-funcall)

Then, **once all blocks have been exited**, returning from a virtual function call is modelled as follows:

$$\frac{e(x) = v \qquad e'' = e'[\mathit{res} \leftarrow v]}{\begin{array}{lll} & (\mathsf{Codepoint}(\mathtt{return}\ x, L, e, \epsilon), & \mathsf{Kretcall}(\mathit{res}, e', L', \mathcal{B}) :: \mathcal{K}, \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathtt{skip}, L', e'', \mathcal{B}) \quad , & \mathcal{K}, \ \mathcal{G})\end{array}}$$

(κ++-return-arg)

$$\frac{}{\begin{array}{lll} & (\mathsf{Codepoint}(\mathtt{return}, L, e, \epsilon), & \mathsf{Kretcall}(\bot, e', L', \mathcal{B}) :: \mathcal{K}, \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathtt{skip}, L', e', \mathcal{B}) \ , & \mathcal{K}, \ \mathcal{G})\end{array}}$$

(κ++-return-no-arg)

`return`ing from within a block with no stack objects first dismisses this block:

$$\frac{}{\begin{array}{lll} & (\mathsf{Codepoint}(\mathtt{return}\ x^?, L, e, (\bot, L') :: \mathcal{B}), & \mathcal{K}, \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathtt{return}\ x^?, L', e, \mathcal{B}) \quad , & \mathcal{K}, \ \mathcal{G})\end{array}}$$

(κ++-return-block-no-obj)

9.4.3 Object-oriented features

9.4.3.1 Field and array accesses

Scalar fields Like s++, reading a scalar field is modelled by retrieving its value from the global state; the static type of the object must be exactly the class where to find the field: there is no implicit cast.

$$\frac{\begin{array}{c}e(x) = \pi \qquad \mathcal{G} \vdash \pi : C \\ f = (\mathtt{scalar}\ t\ \mathit{fid}) \in \mathcal{F}(C) \qquad \mathcal{G}.\mathsf{FieldValues}(\pi, f) = \mathit{res} \qquad e' = e[x' \leftarrow \mathit{res}]\end{array}}{\begin{array}{lll} & (\mathsf{Codepoint}(x' := x\text{->}_C f \ , L, e, \mathcal{B}), & \mathcal{K}, \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathtt{skip} \ , L, e', \mathcal{B}) \quad , & \mathcal{K}, \ \mathcal{G})\end{array}}$$

(κ++-field-scalar-read)

Likewise, writing to a scalar field is modelled by updating its value in the global state as in s++. However, we add a further requirement in κ++: conformingly to the C++ Standard, our formalization forbids writing data to a scalar field that is not $\mathsf{Constructed}$. Such restrictions are not needed when reading: we can prove (THEOREM II.11 p. 217) that if a field has a value, then it is necessarily $\mathsf{Constructed}$.

$$\frac{\begin{array}{c} e(x) = \pi \qquad \mathcal{G} \vdash \pi : C \qquad f = (\mathtt{scalar}\ t\ \mathit{fid}) \in \mathcal{F}(C) \\ \mathcal{G}.\mathsf{ConstrState}^{\mathcal{F}}(\pi, f) = \mathsf{Constructed} \qquad e(x') = \mathit{res} \qquad \mathcal{G}' = \mathcal{G}[\mathsf{FieldValues}(\pi, f) \leftarrow \mathit{res}] \end{array}}{\begin{array}{rl} & (\mathsf{Codepoint}(x\text{->}_C f := x'\ , L, e, \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathtt{skip}\ , L, e, \mathcal{B})\qquad\quad ,\ \ \mathcal{K},\ \ \mathcal{G}') \end{array}}$$

(κ++-field-scalar-write)

Structure fields Similarly to s++, accessing a structure field actually makes a pointer to its first cell, so it is only "pointer adjustment" without actually reading any value. So, as there is no “dereferencing”, no constraint on construction states is needed. However, an additional κ++ requirement allows structure field access only if the complete object is not deallocated.

$$\frac{\begin{array}{c} e(x) = \pi = (\ell, (\alpha, i, \sigma)) \qquad \ell \notin \mathcal{G}.\mathsf{dealloc} \qquad \mathcal{G} \vdash \pi : C \\ f = (\mathtt{struct}\ B[n]\ \mathit{fid}) \in \mathcal{F}(C) \qquad e' = e[x' \leftarrow (\ell, (\alpha +\!\!+ (i, \sigma, f) :: \epsilon, 0, (\mathsf{Repeated}, B :: \epsilon)))] \end{array}}{\begin{array}{rl} & (\mathsf{Codepoint}(x' := x\text{->}_C f\ , L, e, \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathtt{skip}\ , L, e', \mathcal{B})\qquad\quad ,\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}}$$

(κ++-field-struct-point)

Array subscripting Similarly to s++, accessing an array cell is only valid on a reference to a most-derived object. Then it is a mere “pointer adjustment” without actually reading any value, so no constraint on construction states is needed. However, an additional κ++ requirement allows array subscripting only if the corresponding complete object is not deallocated.

$$\frac{\begin{array}{c} e(x) = \pi = (\ell, (\alpha, j, (\mathsf{Repeated}, C :: \epsilon))) \\ \ell \notin G.\mathsf{dealloc} \qquad \mathcal{G}.\mathsf{LocType}(\ell) = (C', n') \qquad C'[n'] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\succ\!\!\to} C[n] \qquad 0 \leq j < n \\ e(x_{\mathsf{i}}) = i \in \mathbb{Z} \qquad 0 \leq j + i < n \qquad e' = e[x' \leftarrow (\ell, (\alpha, j + i, (\mathsf{Repeated}, C :: \epsilon)))] \end{array}}{\begin{array}{rl} & (\mathsf{Codepoint}(x' := x[x_{\mathsf{i}}]_C\ , L, e, \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathtt{skip}\ , L, e', \mathcal{B})\qquad\quad ,\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}}$$

(κ++-array-point)

9.4.3.2 Pointer equality test

Similarly to s++, κ++ allows comparing two pointers of the same type C; either may be null. However, pointers must not refer to deallocated objects.

$$\frac{\begin{array}{c} \forall i \in \{1, 2\} : e(x_i) = \tilde{\pi}_i \\ \forall i \in \{1, 2\} : \mathcal{G} \vdash \tilde{\pi}_i \div C \qquad \forall i \in \{1, 2\} : \tilde{\pi}_i = (\lambda, p) \Rightarrow \lambda \notin \mathcal{G}.\mathsf{dealloc} \\ b \in \{\mathsf{true}, \mathsf{false}\} \qquad b = \mathsf{true} \Leftrightarrow \tilde{\pi}_1 = \tilde{\pi}_2 \qquad e' = e[x' \leftarrow b] \end{array}}{\begin{array}{rl} & (\mathsf{Codepoint}(x' := x[x_{\mathsf{i}}]_C\ , L, e, \mathcal{B}),\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathtt{skip}\ , L, e', \mathcal{B})\qquad\quad ,\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}}$$

(κ++-ptreq)

9.4.3.3 Static cast

To reach a base through static cast, the class must have all its bases constructed (construction state between BasesConstructed, and StartedDestructing). From the implementation point of view, it is necessary for the class to know where its bases, in particular its virtual bases, are located, which justifies the requirement on the bases being constructed. Then, apart from this additional requirement in κ++, static cast behaves as in s++ using the two static cast flavours formally described in Section 4.4.4.1 (p. 87).

$$\frac{\begin{array}{c} e(x) = \pi = (\ell, (\alpha, i, \sigma)) \\ \mathsf{BasesConstructed} \leq \mathcal{G}.\mathsf{ConstrState}(\pi) \leq \mathsf{StartedDestructing} \qquad \mathcal{G}.\mathsf{LocType}(\ell) = (D, n) \\ D[n] \prec\!\langle(\ell, i, \sigma)\rangle\!\!\rightarrowtail B \qquad \mathsf{StatCast}(\sigma, B, B', \sigma') \qquad e' = e[x' \leftarrow \sigma'] \end{array}}{\begin{array}{rll} & (\mathsf{Codepoint}(x' := \mathtt{static\_cast}\langle B'\rangle_B(x)\ , L, e, \mathcal{B}), & \mathcal{K}, \ \ \mathcal{G}) \\ \rightarrow & (\mathsf{Codepoint}(\mathtt{skip}\ , L, e', \mathcal{B}) & , \ \ \mathcal{K}, \ \ \mathcal{G}) \end{array}}$$

(κ++-statcast)

9.4.3.4 Virtual function call: the generalized dynamic type of a subobject

Generalized dynamic type A program is allowed to use virtual functions on a subobject π and all of its bases, in two cases:

- not only during the lifetime of its most-derived object,
- but also during the execution of the constructor body (or field initializers), or the destructor of π.

But the behaviour of virtual function resolution is not the same in the two cases. Indeed, during construction, the subobject for which the constructor body is running is considered as if it were the most-derived object for the purpose of virtual function resolution.

This leads us to define the notion of *generalized dynamic type*, to designate such a subobject, as an extension to the Standard notion of *dynamic type* (which designates the most-derived object for any subobject, during the lifetime of the most-derived object): informally, the generalized dynamic type of an object is the type of the object that is considered as the most-derived object for the purpose of dynamic operations such as virtual function call or dynamic cast.

***Definition* 9.4.1 (Generalized dynamic type).** *Formally, we introduce the predicate* $\mathcal{G} \vdash \mathsf{gDynType}(\ell, \alpha, i, \sigma, C_\circ, \sigma_\circ, \sigma')$ *to denote that the generalized dynamic type of* $(\ell, (\alpha, i, \sigma))$ *is* $\sigma_\circ$*, which is of static type* $C_\circ$*, and* σ' *is an inheritance subobject of* $C_\circ$ *such that* $\sigma = \sigma_\circ @ \sigma'$*.*

There are two cases:

- *Either the most-derived object is constructed. Then, it is the generalized dynamic type for all its subobjects, following the Standard notion of dynamic type:*

$$\frac{\begin{array}{c} \mathcal{G}.\mathsf{LocType}(\ell) = (D, n) \qquad D[n] \prec\!\langle\alpha\rangle\!\!\overset{\mathcal{A}}{\rightarrowtail} C[m] \prec\!\langle(i, \sigma)\rangle\!\!\overset{\mathcal{CI}}{\rightarrowtail} B \\ \mathcal{G}.\mathsf{ConstrState}(\ell, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon))) = \mathsf{Constructed} \end{array}}{\mathcal{G} \vdash \mathsf{gDynType}(\ell, \alpha, i, \sigma, C, (\mathsf{Repeated}, C :: \epsilon), \sigma)}$$

(κ++-dyntype-constructed)

- *Otherwise, the generalized dynamic type is an object of construction state* BasesConstructed *(during the construction or fields, or within the constructor body) or* StartedDestructing *(during the destruction of fields, or within the destructor body), and it is defined only on* *inheritance subobjects of such objects:*

$$\frac{\begin{array}{c}\mathcal{G}.\mathsf{LocType}(\ell) = (D, n)\\ D[n] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\rightarrowtail} C[m] \prec\!\langle(i, \sigma_\circ)\rangle\!\overset{\mathcal{CI}}{\rightarrowtail} C_\circ \qquad \mathcal{G}.\mathsf{ConstrState}(\ell, (\alpha, i, \sigma_\circ)) = c\\ c = \mathsf{BasesConstructed} \vee c = \mathsf{StartedDestructing} \qquad C_\circ \prec\!\langle\sigma'\rangle\!\overset{\mathcal{I}}{\rightarrowtail} B \qquad \sigma = \sigma_\circ @ \sigma'\end{array}}{\mathcal{G} \vdash \mathsf{gDynType}(\ell, \alpha, i, \sigma, C_\circ, \sigma_\circ, \sigma')}$$

(κ++-dyntype-pending)

The properties of the generalized dynamic type are studied in more detail in Section 10.5 (p. 218).

Virtual function call So, we combine our notion of generalized dynamic type with the s++ virtual function dispatch rules defined in Section 4.4.6.1 (p. 90), to obtain the κ++ rule for virtual function call:

1. determine the generalized dynamic type $\sigma_\circ$ of $(\ell, (\alpha, i, \sigma))$; let $C_\circ$ be the type of $\sigma_\circ$, and let σ' such that $\sigma = \sigma_\circ @ \sigma'$
2. dispatch the virtual function assuming that the most-derived object is of type $C_\circ$; let σ'' be the resulting inheritance subobject of $C_\circ$
3. finally the selected subobject is $(\ell, (\alpha, i, \sigma_\circ @ \sigma''))$, adjust the this pointer to this subobject and call f on it.

$$\frac{\begin{array}{c}e(x) = (\ell, (\alpha, i, \sigma)) \qquad \mathcal{G} \vdash \mathsf{gDynType}(\ell, \alpha, i, \sigma, C_\circ, \sigma_\circ, \sigma')\\ \mathsf{VFDispatch}(C_\circ, \sigma', f, B'', \sigma'') \qquad B''.f = f(\mathit{varg}_1, \ldots, \mathit{varg}_n)\{\mathit{body}\}\\ \forall j, e(x_j) = v_j \qquad e' = \varnothing[\mathit{varg}_1 \leftarrow v_1] \ldots [\mathit{varg}_n \leftarrow v_n][\mathsf{this} \leftarrow (\ell, (\alpha, i, \sigma_\circ @ \sigma''))]\end{array}}{\begin{array}{lll} & (\mathsf{Codepoint}(x\texttt{->}_B f(x_1 \ldots x_n), L, e, \mathcal{B}), & \mathcal{K}, \ \mathcal{G})\\ \to & (\mathsf{Codepoint}(\mathit{body}, \epsilon, e', \epsilon) \quad , & \mathsf{Kretcall}(x, L, e, \mathcal{B}) :: \mathcal{K}, \ \mathcal{G})\end{array}}$$

(κ++-virtual-funcall)

We currently do not handle pure virtual functions, or unimplemented virtual functions.

9.4.3.5 Dynamic cast

The behaviour of *dynamic cast* also makes such a distinction on the object that is considered as the most-derived object, "origin" of dynamic cast. So its semantics also makes use of the generalized dynamic type of the subobject.

Then, the κ++ dynamic cast operation first obtains the generalized dynamic type of the subobject following Section 9.4.3.4 (p. 187), then performs the cast under this object considered as a most-derived object, following the s++ rules of Section 4.4.5.1 (p. 88).

$$\frac{\begin{array}{c}e(x) = (\ell, (\alpha, i, \sigma_1)) \qquad \mathcal{G}.\mathsf{LocType}(\ell) = (D, n)\\ D[n] \prec\!\langle(\ell, i, \sigma_1)\rangle\!\rightarrowtail B \qquad \mathcal{G} \vdash \mathsf{gDynType}(\ell, \alpha, i, \sigma_1, C, \sigma_\circ, \sigma) \qquad \mathsf{DynCast}(C, \sigma, B, B') = s\\ s' = \mathsf{match}\ s\ \mathsf{with}\ \sigma' \mapsto (\ell, (\alpha, i, \sigma_\circ @ \sigma')) \ \mid \mathsf{NULL}_{B'} \mapsto \mathsf{NULL}_{B'}\ \mathsf{end} \qquad e' = e[x' \leftarrow s']\end{array}}{\begin{array}{lll} & (\mathsf{Codepoint}(x' := \texttt{dynamic\_cast}\langle B'\rangle_B(x)\ , L, e, \mathcal{B}), & \mathcal{K}, \ \mathcal{G})\\ \to & (\mathsf{Codepoint}(\texttt{skip}\ , L, e', \mathcal{B}) \quad , & \mathcal{K}, \ \mathcal{G})\end{array}}$$

(κ++-dyncast)

9.4.4 Construction

***Definition* 9.4.2.** *Let C be a class, and $n \in \mathbb{N}^*$. A* complete *object of type C is an array of structures explicitly declared in a program block, by a C++ language construct of the form:*

$$\{C\ c[n] = \{\mathit{ObjInit}^*\}; st_1\}$$

The notion of *complete* object must not be confused with that of *most-derived* object. Indeed, a *most-derived* object is an object that cannot be cast to a derived class, but it can be a cell of an array field contained in another object. Actually, any most-derived object is a cell of an array field corresponding either to a complete object, or to an object field.

We shall see that, even though st may `return`, the destruction of the created block-scoped object is still ensured.

Consider entering a block defining a complete object:

$$\{C\ c[n] = \{\iota\}; st\}$$

Then, a new object is allocated in the store, and the construction of the array path ϵ starts. The current code point is not yet saved into the list of enclosing blocks, but is still pending in a continuation frame specifying that a block is to be entered.

$$\frac{\begin{array}{c}\ell \notin \mathsf{dom}(\mathcal{G}.\mathsf{LocType}) \\ \mathcal{G}' = \mathcal{G}[\mathsf{LocType}(\ell) \leftarrow (C, n)] \qquad e' = e[c \leftarrow \mathsf{Ptr}(\ell, \epsilon, 0, (\mathsf{Repeated}, C :: \epsilon))]\end{array}}{\begin{array}{lll}& (\mathsf{Codepoint}(\{C\ c[n] = \{\iota\}; st\}, St, e, Bl), & \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{ConstrArray}(\ell, \epsilon, n, 0, C, \iota, e') \quad , & \mathsf{Kcontinue}(st, e', St, Bl) :: \mathcal{K}, \quad \mathcal{G}')\end{array}}$$

(κ++-block-obj)

So, when the last cell has been constructed, then the execution resumes, entering a new block (with the detail that the variable environment during array construction supersedes the environment in continuation, as cell initializers may have modified some variables):

$$\frac{}{\begin{array}{lll}& (\mathsf{ConstrArray}(\ell, \epsilon, n, n, C, \iota, e') \quad , & \mathsf{Kcontinue}(st, e, St, Bl) :: \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Codepoint}(st, \epsilon, e', (\ell, St) :: Bl), & \mathcal{K}, \quad \mathcal{G})\end{array}}$$

(κ++-constr-array-nil-kcontinue)

Most-derived object Consider a cell $i < n$ of class type C of array α of size n from a complete object ℓ. Then, the corresponding initializer is being run, to choose the right constructor for the cell, which is put in a pending state:

$$\frac{i < n \qquad st = \iota(i)}{\begin{array}{lll}& (\mathsf{ConstrArray}(\ell, \alpha, n, i, C, \iota, e), & \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Codepoint}(st, \epsilon, e, \epsilon) \quad , & \mathsf{Kconstrarray}(\ell, \alpha, n, i, C, \iota) :: \mathcal{K}, \quad \mathcal{G})\end{array}}$$

(κ++-constr-array-cons)

Then, the initializer hands over to a constructor when **there are no pending blocks while running the initializer**. In particular, any object created within the initializer has to be destructed before handing over to the constructor. Thus, no reference to a temporary object can be passed to the constructor. Indeed, such a temporary would have to be destructed after returning from the constructor. Our language does not allow initializers to perform any additional steps after calling the constructor.

So, when the initializer hands over to a constructor to construct an array cell, the arguments are passed to the constructor, forming a new variable environment. Then the construction of a most-derived object starts with the (direct or indirect) virtual bases, assuming the following hypothesis:

Hypothesis 9.4.2. *For any class C, there exists a list $\mathcal{VO}(C)$ of virtual base classes of C such that each virtual base of C appears exactly once in $\mathcal{VO}(C)$, and, if A and B are virtual bases of C such that A is a virtual base of B, then A appears before B in $\mathcal{VO}(C)$.*

The Standard prescribes a way to compute the list of the virtual bases of a class, inducing an order called *inheritance graph order*. We shall see further down that we have modeled this computation (*Definition* 10.3.4 p. 214) and we have proved that it meets this requirement (Lemma 10.3.6 p. 214 and Lemma 10.3.7 p. 214).

$$\frac{\begin{array}{c}\pi = (\ell, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon))) \\ L = \mathcal{VO}(C) \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow \mathsf{StartedConstructing}] \\ vars = x_0, \dots, x_j \qquad \kappa = C(arg_0, \dots, arg_j) : \dots \{\dots\} \\ \forall i, e(x_i) = v_i \qquad e' = \varnothing[arg_0 \leftarrow v_0] \dots [arg_j \leftarrow v_j][\mathsf{this} \leftarrow \mathsf{Ptr}(\pi)]\end{array}}{\begin{array}{lll} & (\mathsf{Codepoint}(C_\kappa(vars), l, e, \epsilon) &, \mathsf{Kconstrarray}(\ell, \alpha, n, i, C, \iota) :: \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Constr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), L, \kappa, e'), & \mathsf{Kconstrothercells}(\ell, \alpha, n, i, C, \iota, e) :: \mathcal{K}, \quad \mathcal{G}')\end{array}}$$

(κ++-constructor-kconstrarray)

Then, when all virtual bases are done constructing, the construction of the non-virtual part of the object starts, beginning with direct non-virtual bases.

$$\frac{\pi = (\ell, (\alpha, i, (h, l))) \qquad \mathsf{last}(l) = C \qquad L = \mathcal{DNV}(C)}{\begin{array}{ll} & (\mathsf{Constr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \epsilon, \kappa, e) \qquad , \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Constr}(\pi, \mathsf{Bases}(\mathsf{DirectNonVirtual}), L, \kappa, e), \quad \mathcal{K}, \quad \mathcal{G})\end{array}}$$

(κ++-constr-bases-virtual-nil)

For the above rule, the semantics does not *a priori* require that π be a most-derived object when constructing the virtual bases. But in fact, we prove it as a run-time invariant.

Non-virtual part of a subobject For any subobject (that is, not necessarily a most-derived object), after constructing all its direct non-virtual bases, then its fields are being constructed, marking the subobject as BasesConstructed to allow using virtual functions:

$$\frac{\begin{array}{c}\pi = (\ell, (\alpha, i, (h, l))) \\ \mathsf{last}(l) = C \qquad L = \mathcal{F}(C) \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow \mathsf{BasesConstructed}]\end{array}}{\begin{array}{ll} & (\mathsf{Constr}(\pi, \mathsf{Bases}(\mathsf{DirectNonVirtual}), \epsilon, \kappa, e), \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Constr}(\pi, \mathsf{Fields}, L, \kappa, e) \qquad , \quad \mathcal{K}, \quad \mathcal{G}')\end{array}}$$

(κ++-constr-bases-direct-non-virtual-nil)

Finally, when all fields have been constructed, the body of the constructor is entered:

$$\frac{\kappa = C(\dots)\{body\}}{\begin{array}{ll} & (\mathsf{Constr}(\pi, \mathsf{Fields}, \epsilon, \kappa, e), \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Codepoint}(body, \epsilon, e, \epsilon), \quad \mathcal{K}, \quad \mathcal{G})\end{array}}$$

(κ++-constr-fields-nil)

What happens on constructor exit (at a `return`) depends on the first continuation stack frame, and shall be discussed later.

Now let us see in more detail what happens when constructing a virtual base, a direct (or indirect) non-virtual base, or a field.

Base or scalar field For all cases except structure fields, starting the construction of such a component c first runs the corresponding initializer:

$$\frac{\beta = \mathsf{Fields} \Rightarrow \mathsf{scalar}\ c \qquad \kappa = C(\dots) : \dots, c\{\mathit{init}\}, \dots \{\dots\}}{\begin{array}{ll} & (\mathsf{Constr}(\pi, \beta, c :: L, \kappa, e), \quad \mathcal{K}, \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathit{init}, \epsilon, e, \epsilon)\ ,\ \ \mathsf{Kconstr}(\pi, \beta, c, L, \kappa) :: \mathcal{K}, \ \mathcal{G}) \end{array}} \quad (\kappa\text{++-constr-cons})$$

Then, there are two cases.

First, if c is a scalar field, then the initializer returns by giving, through $\mathsf{init}(x)$, the variable x to be used as the initial value of the field. But again, **all blocks must have been exited before**, no temporaries are allowed to survive to the handover. The given value constructs the field, then other fields are to be constructed.

$$\frac{\mathsf{scalar}\ f \qquad e(x) = v \qquad \mathcal{G}' = \mathcal{G}[\mathsf{FieldValue}(\pi, f) \leftarrow v][\mathsf{ConstrState}^{\mathcal{F}}(\pi, f) \leftarrow \mathsf{Constructed}]}{\begin{array}{ll} & (\mathsf{Codepoint}(\mathsf{initScalar}(x), sl, e, \epsilon), \ \ \mathsf{Kconstr}(\pi, \mathsf{Fields}, f, L, \kappa) :: \mathcal{K}, \ \ \mathcal{G}\,) \\ \to & (\mathsf{Constr}(\pi', \mathsf{Fields}, L', e) \qquad , \qquad \mathcal{K}, \ \ \mathcal{G}') \end{array}}$$

(κ++-initscalar)

The second case is if c is a base, that is a (direct or indirect) virtual base, or a direct non-virtual base, then the initializer shall exit through handing over to a constructor, again only with **no pending blocks** (any temporary object has to be destructed before handing over to the constructor). On such an exit, arguments are passed to the constructor (marking the actual start of the construction of the base, thus its construction state changes), then the construction of the **non-virtual part** of the base starts, beginning with its direct non-virtual bases.

The rule below applies for both virtual and direct non-virtual bases, the only difference between the two is the computation of the path of the base.

$$\begin{array}{rl} \mathsf{AddBase}((\ell, (\alpha, i, (h, l))), \beta, B) = & \mathsf{match}\ \beta\ \mathsf{with} \\ & \mid\ \mathsf{DirectNonVirtual} \ \mapsto \ (\ell, (\alpha, i, (h, l +\!\!\!+ B :: \epsilon))) \\ & \mid\ \mathsf{Virtual} \qquad\quad\ \mapsto \ (\ell, (\alpha, i, (\mathsf{Shared}, B :: \epsilon))) \\ & \mathsf{end} \end{array}$$

(κ++-addbase)

$$\frac{\begin{array}{c} \pi' = \mathsf{AddBase}(\pi, \beta, B) \qquad \kappa' = B(\mathit{arg}_0, \dots, \mathit{arg}_j) : \dots \{\dots\} \\ \mathit{vars} = x_0, \dots, x_j \qquad \forall i, e(x_i) = v_i \qquad e' = \varnothing[\mathit{arg}_0 \leftarrow v_0] \dots [\mathit{arg}_j \leftarrow v_j][\mathsf{this} \leftarrow \mathsf{Ptr}(\pi')] \\ \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi') \leftarrow \mathsf{StartedConstructing}] \end{array}}{\begin{array}{ll} & (\mathsf{Codepoint}(B_{\kappa'}(\mathit{vars}), sl, e, \epsilon) \qquad , \qquad \mathsf{Kconstr}(\pi, \mathsf{Bases}(\beta), B, L, \kappa) :: \mathcal{K}, \ \ \mathcal{G}\,) \\ \to & (\mathsf{Constr}(\pi', \mathsf{Bases}(\mathsf{DirectNonVirtual}), L', e'), \ \ \mathsf{Kconstrother}(\pi, \mathsf{Bases}(\beta), B, L, \kappa, e) :: \mathcal{K}, \ \ \mathcal{G}') \end{array}}$$

(κ++-constructor-kconstr-base)

Again, for a virtual base, the rule does not explicitly require that the object π be most-derived: we prove it as a run-time invariant.

Then, the construction of the bases goes on, until the constructor body exits through a `return`(): in that case, the base becomes wholly Constructed, and the construction of other sibling bases goes on.

$$\frac{\pi' = \mathsf{AddBase}(\pi,\beta,B) \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi') \leftarrow \mathsf{Constructed}]}{\begin{array}{ll} & (\mathsf{Codepoint}(\mathtt{return}()),\epsilon,e,\epsilon), \quad \mathsf{Kconstrother}(\pi,\mathsf{Bases}(\beta),B,L,\kappa,e) :: \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Constr}(\pi,\mathsf{Bases}(\beta),L,\kappa,e)\ , \quad \mathcal{K}, \quad \mathcal{G}')\end{array}}$$

(κ++-return-kconstrother-bases)

A particular case: scalar fields with no initializer In real-world C++, a scalar field may be left uninitialized though declared constructed, if no initializer is specified in the constructor. The following rule allows such a behaviour.

$$\frac{\kappa \text{ has no initializer for } c \qquad c \text{ is scalar} \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrStates}^{\mathcal{F}}(\pi,f) \leftarrow \mathsf{Constructed}]}{\begin{array}{ll} & (\mathsf{Constr}(\pi,\mathsf{Fields},c :: L,\kappa,e), \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Constr}(\pi,\mathsf{Fields},L,\kappa,e) \quad , \quad \mathcal{K}, \quad \mathcal{G}')\end{array}}$$

(κ++-constr-cons-field-scalar-no-init)

Structure fields The construction of a structure field is equivalent to the construction of its array. Contrary to bases, running initializers is part of the construction of the field, so that the construction state of the field changes before the first initializer starts running.

$$\frac{\begin{array}{c}\pi = (\ell,(\alpha,i,\sigma)) \qquad \mathsf{struct}\ B[n]\ f \\ \kappa = C(\dots) : \dots, f\{\mathit{inits}\}, \dots \{\dots\} \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}^{\mathcal{F}}(\pi,f) \leftarrow \mathsf{StartedConstructing}]\end{array}}{\begin{array}{ll} & (\mathsf{Constr}(\pi,\mathsf{Fields},f :: L,\kappa,e) \quad , \quad \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{ConstrArray}(\ell,\alpha +\!\!+ (i,\sigma,f) :: \epsilon,n,0,B,\mathit{inits},e), \quad \mathsf{Kconstrother}(\pi,\mathsf{Fields},f,L,\kappa,e) :: \mathcal{K}, \quad \mathcal{G}')\end{array}}$$

(κ++-constr-cons-field-struct)

Then, when the last cell of the array is constructed, then the field is considered wholly Constructed, and the construction of other remaining fields goes on, with the variable environment at the end of structure array construction superseding the old one.

$$\frac{\mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}^{\mathcal{F}}(\pi,f) \leftarrow \mathsf{Constructed}]}{\begin{array}{ll} & (\mathsf{ConstrArray}(\ell',\alpha',n,n,B,\mathit{inits},e'), \quad \mathsf{Kconstrother}(\pi,\mathsf{Fields},f,L,\kappa,e) :: \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{Constr}(\pi,\mathsf{Fields},L,\kappa,e') \quad , \quad \mathcal{K}, \quad \mathcal{G}')\end{array}}$$

(κ++-constr-array-nil-kconstrother)

End of the construction of a most-derived object When the body of a most-derived object returns, the first continuation stack frame requests the construction of the further remaining sibling cells of this most-derived object. At that point, the most-derived object becomes Constructed.

$$\frac{\mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\ell,(\alpha,i,(\mathsf{Repeated},C :: \epsilon))) \leftarrow \mathsf{Constructed}]}{\begin{array}{ll} & (\mathsf{Codepoint}(\mathtt{return}()),l,E',\epsilon) \quad , \quad \mathsf{Kconstrothercells}(\ell,\alpha,n,i,C,\iota,e) :: \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathsf{ConstrArray}(\ell,\alpha,n,i+1,C,\iota,e), \quad \mathcal{K}, \quad \mathcal{G}')\end{array}}$$

(κ++-return-kconstrothercells)

9.4.5 Destruction

Complete object We have seen the semantics of `exit` and `return` statements when run from inside a block with no stack object.

To exit from a block with a stack object, this object must first be destructed, starting from the destruction of its last cell.

$$\mathit{ExitStmt} ::= \texttt{exit}\ (\mathsf{S}\ n)\ \mid\ \texttt{return}\ x^{?} \qquad (\kappa\text{++-exitstmt})$$

$$\frac{\mathcal{G}.\mathsf{LocType}(\ell) = (C, n)}{\begin{array}{rll} & (\mathsf{Codepoint}(\mathit{ExitStmt}, L, e, (\ell, L') :: \mathcal{B}), & \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{DestrArray}(\ell, \epsilon, n-1, C)\ , & \mathsf{Kcontinue}(\ell, \mathit{ExitStmt}, e, L', \mathcal{B}) :: \mathcal{K},\ \ \mathcal{G}) \end{array}}$$

(κ++-exit-block-obj)

Then, once the cell -1 is requested to be destructed (i.e. once all cells have been destructed), the object is deallocated from the stack (i.e. it disappears from block definitions, and appears in the list of deallocated objects) and the execution resumes after block exit.

$$\mathsf{ExitSucc}(\mathit{ExitStmt}) = \begin{array}{lll} \multicolumn{3}{l}{\mathsf{match}\ \mathit{ExitStmt}\ \mathsf{with}} \\ \mid\ \texttt{exit}\ (\mathsf{S}\ n) & \mapsto & \texttt{exit}\ n \\ \mid\ \texttt{return}\ x^{?} & \mapsto & \texttt{return}\ x^{?} \\ \multicolumn{3}{l}{\mathsf{end}} \end{array} \qquad (\kappa\text{++-exitsucc})$$

$$\frac{\mathcal{G}' = \mathcal{G}[\mathsf{dealloc} \leftarrow \ell' :: \mathcal{G}.\mathsf{dealloc}]}{\begin{array}{rll} & (\mathsf{DestrArray}(\ell, \alpha, -1, C)\ , & \mathsf{Kcontinue}(\ell', \mathit{ExitStmt}, e, L', \mathcal{B}) :: \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathsf{ExitSucc}(\mathit{ExitStmt}), L', e, \mathcal{B}), & \mathcal{K},\ \ \mathcal{G}') \end{array}}$$

(κ++-destr-array-nil-kcontinue)

Here, nothing enforces $\ell = \ell'$, nor the array path $\alpha = \epsilon$, this is to be shown as an invariant, as we shall see further down.

However, note that $\mathcal{G}.\mathsf{LocType}(\ell)$ is still defined, so as to allow reasoning on construction states even after ℓ is deallocated. Another purpose of the list of deallocated objects is to prevent from reusing ℓ for a later allocated object.

Most-derived object When the destruction of a most-derived object (i.e. a structure array cell) is requested, then the destruction of the non-virtual part of this object starts: the destructor body is entered, then the destruction of fields and bases is requested through Kdestr. Kdestrcell reminds, not only that other array cells have to be destructed, but also that a most-derived object is being destructed, so as not to forget virtual bases once the non-virtual part is destructed.

$$\frac{\begin{array}{c} 0 \le i \qquad \sim C()\{st\} \qquad \pi = (\ell, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon))) \\ e = \varnothing[\mathsf{this} \leftarrow \pi] \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow \mathsf{StartedDestructing}] \end{array}}{\begin{array}{rll} & (\mathsf{DestrArray}(\ell, \alpha, i, C), & \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(st, \epsilon, e, \epsilon)\ , & \mathsf{Kdestr}(\pi) :: \mathsf{Kdestrcell}(\ell, \alpha, i, C) :: \mathcal{K},\ \ \mathcal{G}') \end{array}}$$

(κ++-destr-array-cons)

Non-virtual part of a subobject When a destructor returns, then the fields of the corresponding subobject have to be destructed, in the reverse declaration order.

$$\frac{\pi = (\ell, (\alpha, i, (h, l))) \qquad \mathsf{last}(l) = C \qquad L = \mathsf{rev}(\mathcal{F}(C))}{\begin{array}{rll} & (\mathsf{Codepoint}(\mathtt{return}, st^*, e, \epsilon), & \mathsf{Kdestr}(\pi, C) :: \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Destr}(\pi, \mathsf{Fields}, L) & ,\quad \mathcal{K},\ \ \mathcal{G}) \end{array}} \quad (\kappa\text{++-return-kdestr})$$

Destructing a scalar field erases its value and changes its construction state. Then the destruction of other fields is requested.

$$\frac{f = (\mathit{fid}, (\mathsf{Sc}, t)) \qquad \mathcal{G}' = \mathcal{G}[\mathsf{FieldValues}(\pi, f) \leftarrow \bot][\mathsf{ConstrStates}^{\mathcal{F}}(\pi, f) \leftarrow \mathsf{Destructed}]}{\begin{array}{rll} & (\mathsf{Destr}(\pi, \mathsf{Fields}, f :: L), & \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Destr}(\pi, \mathsf{Fields}, L) & ,\ \ \mathcal{K},\ \ \mathcal{G}') \end{array}}$$
$(\kappa\text{++-destr-fields-cons-scalar})$

Destructing a structure field changes its construction state to StartedDestructing, then requests the destruction of the corresponding array, starting from its last cell, and remembering about other fields through Kdestrother.

$$\frac{\begin{array}{c}\pi = (\ell, (\alpha, i, \sigma)) \qquad f = (\mathit{fid}, (\mathsf{St}, (C, n))) \\ \alpha' = \alpha +\!\!+ (i, \sigma, f) :: \epsilon \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrStates}^{\mathcal{F}}(\pi, f) \leftarrow \mathsf{StartedDestructing}]\end{array}}{\begin{array}{rll} & (\mathsf{Destr}(\pi, \mathsf{Fields}, f :: L) & ,\quad \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{DestrArray}(\ell, \alpha', n-1, C), & \mathsf{Kdestrother}(\pi, \mathsf{Fields}, f, L) :: \mathcal{K},\ \ \mathcal{G}') \end{array}}$$
$(\kappa\text{++-destr-fields-cons-struct})$

Then, once all cells have been destructed, the field is Destructed, and the destruction of further fields can be proceeded.

$$\frac{\mathcal{G}' = \mathcal{G}[\mathsf{ConstrStates}^{\mathcal{F}}(\pi, f) \leftarrow \mathsf{Destructed}]}{\begin{array}{rll} & (\mathsf{DestrArray}(\ell', \alpha', -1, C), & \mathsf{Kdestrother}(\pi, \mathsf{Fields}, f, L) :: \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Destr}(\pi, \mathsf{Fields}, L) & ,\quad \mathcal{K},\ \ \mathcal{G}') \end{array}}$$
$(\kappa\text{++-destr-array-nil-kdestrother})$

Then, once all fields have been destructed, the subobject changes its construction state to DestructingBases (at this point, no virtual function call may be used from this subobject) and the destruction of the direct non-virtual bases can be proceeded, in their reverse declaration order.

$$\frac{\begin{array}{c}\pi = (\ell, (\alpha, i, \sigma)) \\ \mathsf{last}(l) = C \qquad L = \mathsf{rev}(\mathcal{DNV}(C)) \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow \mathsf{DestructingBases}]\end{array}}{\begin{array}{rll} & (\mathsf{Destr}(\pi, \mathsf{Fields}, \epsilon) & ,\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{DirectNonVirtual}), L), & \mathcal{K},\ \ \mathcal{G}') \end{array}}$$
$(\kappa\text{++-destr-fields-nil})$

Destructing a (virtual or direct non-virtual) base B of π enters its destructor, remembering other bases through Kdestrother.

$$\frac{\begin{array}{c}\sim B()\{st\} \\ \pi' = \mathsf{AddBase}(\pi, \beta, B) \qquad e = \varnothing[\mathsf{this} \leftarrow \pi'] \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi') \leftarrow \mathsf{StartedDestructing}]\end{array}}{\begin{array}{rll} & (\mathsf{Destr}(\pi, \mathsf{Bases}(\beta), B :: L), & \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathsf{Codepoint}(st, \epsilon, e, \epsilon) & ,\ \ \mathsf{Kdestr}(\pi') :: \mathsf{Kdestrother}(\pi, \mathsf{Bases}(\beta), B, L) :: \mathcal{K},\ \ \mathcal{G}') \end{array}}$$
$(\kappa\text{++-destr-bases-cons})$

Then, once all direct non-virtual bases of π have been destructed, there are two cases, depending on the top of the continuation stack.

Either the continuation stack starts with a $\mathsf{Kdestrother}(\pi', \mathsf{Bases}(\beta))$, then π' is not a most-derived object. So, only the non-virtual part of π had to be destructed, so it may become Destructed, and the destruction of those other bases of π' is requested.

$$\frac{\mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow \mathsf{Destructed}]}{\begin{array}{rllll} & (\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{DirectNonVirtual}), \epsilon), & \mathsf{Kdestrother}(\pi', \mathsf{Bases}(\beta), B, L) :: \mathcal{K}, & \mathcal{G}) \\ \to & (\mathsf{Destr}(\pi', \mathsf{Bases}(\beta), L) & , \quad \mathcal{K}, & \mathcal{G}') \end{array}}$$
$(\kappa$++-destr-bases-direct-non-virtual-nil-kdestrother$)$

End of the destruction of a most-derived object The second case is when the continuation stack starts with a Kdestrcell. Then, π is a most-derived object (this is not constrained by the rules, but must be proved as an invariant), and its virtual bases need to be destructed, in the reverse order of their construction (given by the list $\mathcal{VO}(C)$).

$$\frac{L = \mathsf{rev}(\mathcal{VO}(C))}{\begin{array}{rllll} & (\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{DirectNonVirtual}), \epsilon), & \mathsf{Kdestrcell}(\ell, \alpha, i, C) :: \mathcal{K}, & \mathcal{G}) \\ \to & (\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), L) & , \quad \mathcal{K}, & \mathcal{G}) \end{array}}$$
$(\kappa$++-destr-bases-direct-non-virtual-nil-kdestrcell$)$

Finally, when all virtual bases have been destructed, then the object (which is actually the most-derived object) is Destructed, and the destruction of further cells may be proceeded.

$$\frac{\pi = (\ell, (\alpha, i, (h, l))) \qquad \mathsf{last}(l) = C \qquad \mathcal{G}' = \mathcal{G}[\mathsf{ConstrState}(\pi) \leftarrow \mathsf{Destructed}]}{\begin{array}{rlll} & (\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \epsilon), & \mathcal{K}, & \mathcal{G}) \\ \to & (\mathsf{DestrArray}(\ell, \alpha, i-1, C) \;, & \mathcal{K}, & \mathcal{G}') \end{array}}$$
$(\kappa$++-destr-bases-virtual-nil$)$

Contrary to the construction semantics, one can see that Kdestrcell is not a strict counterpart to Kconstrothercells, as it is not present in the continuation stack when destructing the virtual bases (then, the object π in $\mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \dots)$ or $\mathsf{Kdestrother}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \dots)$ actually refers to the most-derived object from which the next array cell may be deduced. It would have been redundant to keep Kdestrcell under such circumstances; conversely, Kconstrothercells is required during construction because of the variable environment, which changes between each cell due to their initializers.

9.5 Impact on the C++ language specification

Following our work, Gabriel Dos Reis and Bjarne Stroustrup submitted proposals for modifying the C++03 Standard [42]. Some of them have been adopted in time for C++11 [3] as voted in March 2011 [43]. This proves the interest of machine-checked formalizations to reason about the consistency of prose standards. The question of whether such textual standards should be replaced by mathematized formalizations is, however, left open.

3. formerly dubbed "C++0x"

9.5.1 Virtual function calls during field destruction

The C++03 Standard [42] prescribes that virtual functions can be called directly or indirectly from within the constructor body, including the mem-initializers for data members. That is, during the construction of data member subobjects, virtual functions of the object may be used, as in the following C++ code:

```
struct B;
struct X {
  X(B* b) {
    b->f();
  }
};
struct B {
  X x;
  B(): x(this) {} /* initializer for x calls this->f(), well-defined */
  virtual void f();
};
```

However, according to [42], the semantics of function call was not defined in the symmetric case of destructor body:

```
struct B;
struct X {
  X(B* b) {}
  ~X() {
    b->f();
  }
};
struct B {
  X x;
  B(): x(this) {}
  ~B() {}  /* destructing a calls this->f(), undefined */
  virtual void f();
};
```

Our work led to a proposal [45] to amend the Standard. This amendment has been accepted to the newest C++11 Standard [43]. Our κ++ language correctly models the modified semantics: the generalized dynamic type is always defined for an object during the construction or destruction of its data members.

9.5.2 Object lifetime

9.5.2.1 Conflicting descriptions of end of object lifetime

Although our formalism does not allow explicit management of object lifetime (apart from objects attached to statement blocks by scoping rules), the development of our formalism led us to discover several conflicts and unintended semantics in both C++03 and C++11 standard documents.

For instance, C++11 introduces a conflict in the effects of calling a destructor. On the one hand, §3.8p1 and 3.8p4 claim that the lifetime of an object ends when its non-trivial destructor is called or its storage is reused or releasted. On the other hand, §14.2p4 claims that once a (trivial or non-trivial) destructor is invoked, “the object no longer exists”. As an object can be manipulated during its construction or destruction, it must necessarily already exist. This discrepancy pointed out a confusion between object lifetime (= already constructed, destruction not yet started) and object existence (= allocated, not yet deallocated). This issue will be corrected after C++11.

9.5.2.2 Lifetime of array objects

Again, both C++03 and C++11 inconsistently define the lifetime of array objects: they make it start right upon allocation, and end right upon deallocation, independently of the lifetimes of array cells. This contradicts the general principle that the lifetime of an object starts once the lifetimes of all its subobjects have started. The lifetime of an array could be more accurately defined as the lifetime of its last cell. This issue will be corrected after C++11.

Chapter 10

Formalization of object lifetime

In this section, we first present a run-time invariant holding on κ++ states during the execution of a program (Section 10.1 p. 199). Then, thanks to this invariant, we show a wide range of interesting high-level properties on the construction states of objects during program execution. We summarize them below:

10.2 Construction and destruction rules are structurally sound: they may fail only because of wrong user code in initializers or constructor/destructor bodies.

10.3 Construction states are monotonic; each object goes through each construction state following S from $\mathsf{Unconstructed}$ to $\mathsf{Destructed}$, and changes its construction state exactly once each. This monotonicity property allows to formally state and prove the principle of RAII (Resource Acquisition is Initialization). Each object becomes constructed and destructed exactly once. Subobjects of a complete object are destructed in the reverse order of their construction. The lifetime of an object is included in the lifetime of all of its subobjects. This allows to prove that, when an object is deallocated, then all its subobjects have been destructed before. Moreover, as κ++ only features stack-allocated objects, lifetimes of different complete objects actually follow a stack discipline.

10.4 If a scalar field has a value, then it is constructed.

10.5 The generalized dynamic types of each object and all of its bases change at well-defined execution points.

10.1 κ++ Run-Time invariant

On top of the operational semantics of κ++ defined in Chapter 9 (p. 171), we built a run-time invariant [1] and we proved that this invariant always holds during program execution [2], and also holds for the initial state with no initially allocated objects. This invariant is built of several layers:

- *Contextual invariants*: states some properties about subobjects involved in the state kind and the stack frames.
  - *Kind-level invariant*: states that the subobject involved in the state kind during construction and destruction are valid (i.e. it is consistent with the hierarchy and the object heap), and sets their construction state with respect to the kind

1. Coq development: theory `Invariant`.
2. Coq development: theory `Preservation`.

  - *Stack-level invariant*: states that the subobjects involved in the continuation stack frames are during construction and destruction are valid (i.e. they are consistent with the hierarchy and the object heap), and sets their construction state with respect to the current construction step
  - *Stackframe chaining*: states that the kind requires the presence of a specific frame on top of the stack, and precises similar conditions allowing or not two stack frames to immediately follow each other
  - *Stack well-foundedness*: states that the stack is "sorted" following an order on the subobjects involved in the construction and destruction stack frames
- *Stack objects and constructed stack objects*: shows how to compute the sets of allocated objects and relates their construction states
- *General relations between construction states*:
  - *Vertical invariant*: relates the construction states between an object and its direct (inheritance or field) subobjects (*vertical* relations between construction states)
  - *Horizontal invariant*: relates the construction states between two "sister" subobjects, that is, two subobjects that have the same direct parent object (*horizontal* relations between construction states).

All those invariants trivially hold on the initial state. But, proving their preservation along semantic rules needs around 15000 lines of Coq and around two hours to compile on a Pentium Core Duo 2GHz, consuming around 2Gb of RAM.

Virtually all invariants in this section need each other to hold.

In this whole section, we consider a fixed execution state $(S, \mathcal{K}, \mathcal{G})$.

10.1.1 ($\star$) Contextual invariants

This section is very technical and allows one to understand how the more "high-level" properties can be proved. On first reading, it can be skipped to Section 10.1.2 (p. 206)

The semantic rules are rather lax: they do not appear to constrain the subobject involved in different state kinds or stack frames. However, such conditions are essential to reason about the construction and destruction process.

10.1.1.1 Invariant on execution point

To safely apply the semantic rules, some conditions must hold on the subobjects involved in state kinds, and on their construction states.

- Whenever a list of items is requested to be constructed:

$$S = \mathsf{Constr}(\pi, \mathit{ItemKind}, \kappa, L, \mathit{Env})$$

  - π is a valid pointer to a subobject of some static type B (regardless of its construction state):

$$\mathcal{G} \vdash \pi : B$$

  - there is a list L' such that the complete list of *ItemKind* of B can be written as $L' +\!\!+ L$, with any item $A \in L'$ being $\mathsf{Constructed}$, and any item $A \in L$ is $\mathsf{Unconstructed}$.
  - if $\mathit{ItemKind} = \mathsf{Bases}(\beta)$:
    - the construction of π has started, but only concerning its bases:

$$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{StartedConstructing}$$

  - if $\beta = \mathsf{Virtual}$, then π is a most-derived object of static type B
- otherwise ($\mathit{ItemKind} = \mathsf{Fields}$), the bases of π are already constructed:

$$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{BasesConstructed}$$

- Whenever an array cell is requested to be constructed:

$$S = \mathsf{Constrarray}(\ell, \alpha, n, i, C, \kappa)$$

  - ℓ is a valid object location:

$$\mathcal{G}.\mathsf{LocType}(\ell) = (C', n')$$

  - α is an array path from (C', n') to (C, n) where n is maximal
  - $0 \leq i \leq n$ (we may have $i = n$, in this case rules (κ++-constr-array-nil-kcontinue, p. 189) and (κ++-constr-array-nil-kconstrother, p. 192) may apply instead of (κ++-constr-array-cons, p. 189))
  - all cells j with $0 \leq j < i$ are Constructed
  - all cells j with $i \leq j < n$ are Unconstructed
- Whenever a list of items is requested to be destructed:

$$S = \mathsf{Destr}(\pi, \mathit{ItemKind}, L)$$

  - π is a valid pointer to a subobject of some static type B (regardless of its construction state):

$$\mathcal{G} \vdash \pi : B$$

  - there is a list L' such that the complete list of $\mathit{ItemKind}$ of B can be written as $\mathsf{rev}(L' +\!\!+ L)$, with any item $A \in L'$ being Destructed, and any item $A \in L$ is Constructed.
  - if $\mathit{ItemKind} = \mathsf{Bases}(\beta)$:
    - the destruction of π has started, already concerning its bases:

$$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{DestructingBases}$$

    - if $\beta = \mathsf{Virtual}$, then π is a most-derived object of static type B
  - otherwise ($\mathit{ItemKind} = \mathsf{Fields}$), the bases of π are still constructed:

$$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{StartedDestructing}$$

- Whenever an array cell is requested to be destructed:

$$S = \mathsf{Destrarray}(\ell, \alpha, i, C, \kappa)$$

  - ℓ is a valid object location:

$$\mathcal{G}.\mathsf{LocType}(\ell) = (C', n')$$

  - α is an array path from (C', n') to (C, n) where n is maximal
  - $-1 \leq i < n$ (we may have $i = -1$, in this case rules (κ++-destr-array-nil-kcontinue, p. 193) and (κ++-destr-array-nil-kdestrother, p. 194) may apply instead of (κ++-destr-array-cons, p. 193))
  - all cells j with $0 \leq j \leq i$ are Constructed
  - all cells j with $i < j < n$ are Destructed

The invariant for Codepoint depends on the first item on top of the stack, as we shall see in the next sections.

10.1.1.2 Invariant for stack frames

To safely apply the semantic rules, some conditions must hold on the subobjects involved in each stack frame, and on their construction states.

- Whenever a list of items is pending to be constructed, during the construction of B:

$$\mathcal{K} \ni \mathsf{Kconstrother}(\pi, \mathit{ItemKind}, \kappa, B, L, \mathit{Env})$$

  - π is a valid pointer to a subobject of some static type B (regardless of its construction state):
$$\mathcal{G} \vdash \pi : B$$
  - there is a list L' such that the complete list of *ItemKind* of B can be written as $L' +\!\!+ B :: L$
  - B is StartedConstructing (or maybe BasesConstructed, allowed only if $\mathit{ItemKind} = \mathsf{Bases}(\beta)$)
  - if $\mathit{ItemKind} = \mathsf{Bases}(\beta)$:
    - the construction of π has started, but only concerning its bases:

$$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{StartedConstructing}$$

    - if $\beta = \mathsf{Virtual}$, then π is a most-derived object of static type B
  - otherwise ($\mathit{ItemKind} = \mathsf{Fields}$), the bases of π are already constructed:

$$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{BasesConstructed}$$

- Whenever an array cell is requested to be constructed:

$$\mathcal{K} \ni \mathsf{Kconstrothercells}(\ell, \alpha, n, i, C, \kappa)$$

  - ℓ is a valid object location:

$$\mathcal{G}.\mathsf{LocType}(\ell) = (C', n')$$

  - α is an array path from (C', n') to (C, n) where n is maximal
  - $0 \leq i < n$
  - cell i is StartedConstructing, or BasesConstructed
- Whenever a list of items is requested to be destructed:

$$\mathcal{K} \ni \mathsf{Kdestrother}(\pi, \mathit{ItemKind}, B, L)$$

  - π is a valid pointer to a subobject of some static type B (regardless of its construction state):
$$\mathcal{G} \vdash \pi : B$$
  - there is a list L' such that the complete list of *ItemKind* of B can be written as $\mathsf{rev}(L' +\!\!+ B :: L)$
  - B is StartedDestructing (or maybe DestructingBases, allowed only if $\mathit{ItemKind} = \mathsf{Bases}(\beta)$)
  - if $\mathit{ItemKind} = \mathsf{Bases}(\beta)$:
    - the destruction of π has started, already concerning its bases:

$$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{DestructingBases}$$

  - if $\beta = \mathsf{Virtual}$, then π is a most-derived object of static type B
- otherwise ($\mathit{ItemKind} = \mathsf{Fields}$), the bases of π are still constructed:

$$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{StartedDestructing}$$

- Whenever an array cell is requested to be destructed:

$$\mathcal{K} \ni \mathsf{Kdestrcell}(\ell, \alpha, i, C, \kappa)$$

  - ℓ is a valid object location:

$$\mathcal{G}.\mathsf{LocType}(\ell) = (C', n')$$

  - α is an array path from (C', n') to (C, n) where n is maximal
  - $0 \leq i < n$
  - cell i is $\mathsf{StartedDestructing}$, or $\mathsf{DestructingBases}$

Those invariants are very close to the kind invariants, but they differ in the construction state of the object being constructed or destructed. The construction states of sibling objects are not specified here, but they may be deduced thanks to the horizontal invariants described further down.

Other stack frames only appear in specific contexts: $\mathsf{Kconstr}$ (during the initializer for a base or scalar field), $\mathsf{Kconstrarray}$ (during the initializer for a structure array cell), and Kdestr (during the destructor). For this reason, they are not treated as "stand-alone" stack frames, but separately, in the following section (chaining).

10.1.1.3 Stackframe chaining

Some stack frames necessarily require to be immediately followed (towards the bottom of the stack) by specific stack frames. The same holds with some kinds, which require specific stack frames on top of the stack. Moreover, depending on those stack frames, their involved subobjects are related in some way.

In this section, we consider that $\mathcal{K} = \mathcal{K}' +\!\!+ K_1 :: K_2 :: \mathcal{K}''$, and we describe, depending on K_1, which frame K_2 may follow. (In parallel, we mention those invariants that may also hold for some kinds requiring that $\mathcal{K} = K_2 :: \mathcal{K}''$).

Construction/destruction kinds and stack frames

- During the construction of the virtual bases of an object π, i.e. any of the following cases:
  - $K_1 = \mathsf{Kconstr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \dots)$
  - $K_1 = \mathsf{Kconstrother}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \dots)$
  - or $S = \mathsf{constr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \dots)$ and $\mathcal{K} = K_2 :: \mathcal{K}''$

  Then, necessarily, the frame (or kind) is immediately followed by pending cells:

$$K_2 = \mathsf{Kconstrothercells}(\ell, \alpha, n, i, C, \dots)$$

  and their subobjects are related:

$$\pi = (\ell, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon)))$$

- During the construction of an array cell, i.e. any of the following cases:

  - $K_1 = \mathsf{Kconstrarray}(\ell, \alpha, i, C, \dots)$
  - $K_1 = \mathsf{Kconstrothercells}(\ell, \alpha, i, C, \dots)$
  - or $S = \mathsf{constrarray}(\ell, \alpha, i, C, \dots)$ and $\mathcal{K} = K_2 :: \mathcal{K}''$

  Then, necessarily, the frame (or kind) is immediately followed by one of the two cases, depending on α:
  - either $\alpha = \epsilon$: then, the array being constructed is the whole complete object ℓ itself, so the frame (or kind) must be followed by a code frame $K_2 = \mathsf{Kcontinue}(\ell, \dots)$ specifying that the array in construction or destruction is actually the allocated stack object.
  - otherwise, $\alpha = \alpha' +\!\!+ (i', \sigma', f') :: \epsilon$ is a structure array field, so the frame (or kind) must be followed by the field construction frame $K_2 = \mathsf{Kconstrother}(\ell, (\alpha', i', \sigma'), \mathsf{Fields}, f', \dots)$.
- During the construction of the non-virtual bases, or the fields, of an object π, i.e. any of the following cases with $\beta \neq \mathsf{Bases}(\mathsf{Virtual})$:
  - $K_1 = \mathsf{Kconstrother}(\pi, \beta, \dots)$
  - $K_1 = \mathsf{Kconstr}(\pi, \beta, \dots)$
  - or $S = \mathsf{constr}(\pi, \beta, \dots)$ and $\mathcal{K} = K_2 :: \mathcal{K}''$

  Then, one of the following cases holds, depending on π:
  - construction of the bases of some object π':

$$K_2 = \mathsf{Kconstrother}(\pi', \mathsf{Bases}(\beta'), B, \dots)$$

    Then, π is the base B being precisely constructed by K_2:

$$\pi = \mathsf{AddBase}(\pi', \beta', B)$$

  - construction of array cells:

$$K_2 = \mathsf{Kconstrothercells}(\ell, \alpha, i, C)$$

    Then, π is the cell being constructed:

$$\pi = (\ell, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon)))$$

Similarly, for destruction:
- During the destruction of the virtual bases of an object π, i.e. any of the following cases:
  - $K_1 = \mathsf{Kdestrother}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \dots)$
  - or $S = \mathsf{Destr}(\pi, \mathsf{Bases}(\mathsf{Virtual}), \dots)$ and $\mathcal{K} = K_2 :: \mathcal{K}''$

  or, during the destruction of an array cell, i.e. any of the following cases:
  - $K_1 = \mathsf{Kdestrcell}(\ell, \alpha, i, C, \dots)$
  - or $S = \mathsf{DestrArray}(\ell, \alpha, i, C, \dots)$ and $\mathcal{K} = K_2 :: \mathcal{K}''$

  Then, necessarily, in the first two cases, $\pi = (\ell, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon)))$ is a most-derived object; and, in all cases, the frame (or kind) is immediately followed by one of the two cases, depending on α:
  - either $\alpha = \epsilon$: then, the array being destructed is the whole complete object ℓ itself, so the frame (or kind) must be followed by a code frame $K_2 = \mathsf{Kcontinue}(\ell, \dots)$ specifying that the array is the stack object.
  - otherwise, $\alpha = \alpha' +\!\!+ (i', \sigma', f') :: \epsilon$ is a structure array field, so the frame (or kind) must be followed by the field destruction frame $K_2 = \mathsf{Kdestrother}(\ell, (\alpha', i', \sigma'), \mathsf{Fields}, f', \dots)$.
- During the destruction of the non-virtual part of an object π, i.e. any of the following cases with $\beta \neq \mathsf{Bases}(\mathsf{Virtual})$:

– $K_1 = \mathsf{Kdestrother}(\pi, \beta, \dots)$
– $K_1 = \mathsf{Kdestr}(\pi)$ (running the destructor)
– or $S = \mathsf{destr}(\pi, \beta, \dots)$ and $\mathcal{K} = K_2 :: \mathcal{K}''$

Then, one of the following cases holds, depending on π:

– destruction of the bases of some object π':

$$K_2 = \mathsf{Kdestrother}(\pi', \mathsf{Bases}(\beta'), B, \dots)$$

Then, π is the base B being precisely destructed by K_2:

$$\pi = \mathsf{AddBase}(\pi', \beta', B)$$

– destruction of array cells:

$$K_2 = \mathsf{Kdestrcell}(\ell, \alpha, i, C)$$

Then, π is the cell being destructed:

$$\pi = (\ell, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon)))$$

Code points If K_1 is a code frame, that is Kretcall or Kcontinue (or if $S = \mathsf{Codepoint}(\dots)$ and $\mathcal{K} = K_2 :: \mathcal{K}''$), then K_2 may be one of the following cases:

– another code point, $\mathsf{Kcontinue}(\dots)$ or $\mathsf{Kretcall}(\dots)$
– while executing an initializer for a base or scalar field: $\mathsf{Kconstr}(\pi, \mathit{ItemKind}, B, L, \dots)$. Then, the kind invariant of $\mathsf{constr}(\pi, \mathit{ItemKind}, B :: L)$ holds (the construction states do not change when entering the initializer)
– while executing an initializer for an array cell: $\mathsf{Kconstrarray}(\ell, \alpha, i, C, \dots)$. Then, the kind invariant of $\mathsf{ConstrArray}(\ell, \alpha, i, C, \dots)$ holds (the construction states do not change when entering the initializer)
– while executing the constructor for a base B of some object π: $\mathsf{Kconstrother}(\pi, \mathsf{Bases}(\beta), B, L, \dots)$. Then, the base B is BasesConstructed, but not yet Constructed (not before the constructor has exited), so it must be made explicit that all its fields are already Constructed.
– while executing the constructor for a most-derived object (i.e. a structure array cell): $\mathsf{Kconstrothercells}(\ell, \alpha, n, i, C)$. Then, the cell i is BasesConstructed, but not yet Constructed (not before the constructor has exited), so it must be made explicit that all its fields are already Constructed.
– symmetrically, while executing the destructor for an object: $\mathsf{Kdestr}(\pi)$. Then, π is StartedDestructing, but no longer Constructed, as the destructor entered, so must be made explicit that all fields of π are still Constructed.

10.1.1.4 Stack well-foundedness

To make reasoning easier, we show an invariant on the stack frames, relating the subobjects of two different stack frames, but regarding the same complete object. Roughly speaking, if (ℓ, σ) is the object being constructed/destructed by some stack frame, then it is a strict subobject of any object (ℓ, σ') being constructed/destructed by a stack frame deeper down in the stack.

More precisely:

***Definition* 10.1.1.** *The subobject being constructed or destructed by a stack frame K is:*
– *π, if $\mathcal{K}$ is any of* $\mathsf{Kconstr}(\pi, \dots)$, $\mathsf{Kconstrother}(\pi, \dots)$, $\mathsf{Kdestr}(\pi)$ *or* $\mathsf{Kdestrother}(\pi, \dots)$,

– *undefined otherwise*
Similarly, the subobject being constructed or destructed by a state kind S is:
– *π, if S is any of* $\mathsf{Constr}(\pi, \dots)$*,* $\mathsf{Destr}(\pi, \dots)$
– *undefined otherwise*

***Definition* 10.1.2.** *The array being constructed or destructed by a stack frame K is:*
– *(ℓ, α), if $\mathcal{K}$ is any of* $\mathsf{Kconstrarray}(\ell, \alpha \dots)$*,* $\mathsf{Kconstrothercells}(\ell, \alpha, \dots)$ *or* $\mathsf{Kdestrcell}(\ell, \alpha, \dots)$
– *undefined otherwise*
Similarly, the array being constructed or destructed by a state kind S is:
– *(ℓ, α), if S is any of* $\mathsf{ConstrArray}(\ell, \alpha, \dots)$*,* $\mathsf{DestrArray}(\ell, \alpha, \dots)$
– *undefined otherwise*

Then, if $\mathcal{K} = \mathcal{K}' +\!\!+ K_1 :: \mathcal{K}''$, and if $K_2 \in \mathcal{K}''$ (or similarly, if S is a state kind and $K_2 \in \mathcal{K}$), then:
- if $\pi_1 = (\ell, (\alpha_1, i_1, \sigma_1))$ is the subobject being constructed or destructed by K_1 or S, then:
  - if $\pi_2 = (\ell, (\alpha_2, i_2, \sigma_2))$ is the subobject being constructed or destructed by K_2, then:
    - either $\alpha_1 = \alpha_2 +\!\!+ \alpha$ for some $\alpha \neq \epsilon$
    - or $\alpha_1 = \alpha_2$, $i_1 = i_2$, and σ_1 is an inheritance subobject of σ_2 distinct from σ_2 itself
  - otherwise, if (ℓ, α_2) is the array being constructed or destructed by K_2, then $\alpha_1 = \alpha_2 +\!\!+ \alpha$ for some α (maybe ϵ)
- otherwise, if (ℓ, α_1) is the array being constructed or destructed by K_1 or S, then, if $(\ell, (\alpha_2, i, \sigma))$ is the subobject, or if (ℓ, α_2) is the array, being constructed or destructed by K_2, then $\alpha_1 = \alpha_2 +\!\!+ \alpha$ for some $\alpha \neq \epsilon$

10.1.2 Stack objects and constructed stack objects

In this section, we investigate how to compute the list of stack objects, and stackframe objects, looking at the code points.

When executing a statement block, the block may or may not define a complete object. In this case, this object is called the *stack object* associated to the block. However, when defining such an object, the block receives this object only when it is wholly constructed, and it loses this object once the object starts destruction.

More formally:

***Definition* 10.1.3 (Stack object of a block).** *A block b : Block has at most one stack object, written $C\Omega(b)$:*
– *If the block is $(\bot, \mathit{Stmt})$, then it has no stack object*
– *If the block is (ℓ, Stmt), then ℓ is its stack object*

The list of constructed stack objects is computed by gathering all stack objects of all blocks of all code points (code stack frames $\mathsf{Kcontinue}$ or $\mathsf{Kretcall}$, and also the kind if it is $\mathsf{Codepoint}$).

***Definition* 10.1.4.** *The list of the* constructed stack objects *$C\Omega(K)$ of a stack frame K is:*
– *if $K = \mathsf{Kretcall}(\mathit{res}^?, \mathit{Env}, \mathit{Stmt}^*, \mathcal{B})$ or $K = \mathsf{Kcontinue}(\ell^?, \mathit{Stmt}_1, \mathit{Env}, \mathit{Stmt}_2^*, \mathcal{B})$, then $C\Omega(K) = \bigcup_{b \in \mathcal{B}} C\Omega(b)$*
– *otherwise, $C\Omega(K) = \epsilon$.*

(The notation $\bigcup$ is meant here to keep the order of collected objects following the order of blocks in the block list).

***Definition* 10.1.5.** *Similarly, the list of the constructed stack objects $C\Omega(S)$ of a state kind S is:*
- *if $S = \mathsf{Codepoint}(Stmt_1, Stmt^*, Env, \mathcal{B})$, then $C\Omega(S) = \bigcup_{b\in\mathcal{B}} C\Omega(b)$*
- *otherwise, $C\Omega(S) = \epsilon$.*

Putting all together:

***Definition* 10.1.6.** *The list of the constructed stack objects $C\Omega(S, \mathcal{K})$ of an execution state $(S, \mathcal{K}, \mathcal{G})$ is computed as follows:*

$$C\Omega(S, \mathcal{K}) = C\Omega(S) \cup \bigcup_{K\in\mathcal{K}} C\Omega(K)$$

To collect all stack objects, we must also take into account the objects in construction or destruction, carried by corresponding Kcontinue frames:

***Definition* 10.1.7.** *The list of the* stack objects *$\Omega(K)$ of a stack frame K are:*
- *if $K = \mathsf{Kcontinue}(\varnothing, Stmt_1, Env, Stmt_2^*, \mathcal{B})$ or $K = \mathsf{Kretcall}(res^?, Env, Stmt^*, \mathcal{B})$, then $\Omega(K) = \bigcup_{b\in\mathcal{B}} C\Omega(b)$*
- *if $K = \mathsf{Kcontinue}(\ell, Stmt_1, Env, Stmt_2^*, \mathcal{B})$, then $\Omega(K) = \ell :: \bigcup_{b\in\mathcal{B}} C\Omega(b)$*
- *otherwise, $\Omega(K) = \epsilon$.*

***Definition* 10.1.8.** *The list of the stack objects $\Omega(S, \mathcal{K})$ of an execution state $(S, \mathcal{K}, \mathcal{G})$ is computed as follows:*

$$\Omega(S, \mathcal{K}) = C\Omega(S) \cup \bigcup_{K\in\mathcal{K}} \Omega(K)$$

Lemma 10.1.1.

$$C\Omega(S, \mathcal{K}) \subseteq \Omega(S, \mathcal{K})$$

Invariants
- If $\mathcal{G}.\mathsf{LocType}(\ell) = \bot$ is undefined, then any construction state on ℓ is Unconstructed.
- $\Omega(S, \mathcal{K})$, and $\mathcal{G}.\mathsf{dealloc}$, have no duplicates.
- $\Omega(S, \mathcal{K})$ and $\mathcal{G}.\mathsf{dealloc}$ are disjoint.
- $\mathcal{G}.\mathsf{LocType}(\ell)$ is defined if, and only if, $\ell \in \Omega(S, \mathcal{K}) \cup \mathcal{G}.\mathsf{dealloc}$
- If $\ell \in C\Omega(S, \mathcal{K})$ and if $\mathcal{G}.\mathsf{LocType}(\ell) = (C, n)$, then all n cells of ℓ are Constructed.
- Similarly, if $\mathcal{G}.\mathsf{LocType}(\ell) = (C, n)$ and $\ell \in \mathcal{G}.\mathsf{dealloc}$, then all n cells of ℓ are Destructed.

The latter invariant is needed to show the kind invariant of DestrArray for rule (κ++-exit-block-obj, p. 193).

10.1.3 General relations between construction states

This section is more high-level: it relates the construction states of objects depending on their relative position in the "subobject ordering" tree[3] of Figure 9.1 (p. 173).

3. Coq development: theory `SubobjectOrdering`.

10.1.3.1 Vertical relations

***Definition* 10.1.9.** *Let p, p' be two subobjects of the same complete object. We say that p is a* direct subobject *of p' if either of the following is true:*
– *p is a direct non-virtual base of p'*
– *p' is a most-derived object and p is a virtual base of p'*
– *p is a cell of a structure array field of p' (then, we say that p is a* field subobject *of p').*
In the first two cases, we say that p is a base subobject *of p'.*

By language abuse, we say that p is a *virtual base* (resp. direct non-virtual base) of p' when p designates the subobject corresponding to a virtual base (resp. direct non-virtual base) of the class that is the static type of p'.

Invariant 10.1.1 (Middle-level invariant: vertical relations on construction states). *If p is a direct subobject of p', then the following table relates their construction states:*

If p' is...	*Then p is...*
Unconstructed	Unconstructed
StartedConstructing	Unconstructed *if p is a field subobject of p'* *between* Unconstructed *and* Constructed *otherwise*
BasesConstructed	Constructed *if p is a base subobject of p'* *between* Unconstructed *and* Constructed *otherwise*
Constructed	Constructed
StartedDestructing	Constructed *if p is a base subobject of p'* *between* Constructed *and* Destructed *otherwise*
DestructingBases	Destructed *if p is a field subobject of p'* *between* Constructed *and* Destructed *otherwise*
Destructed	Destructed

10.1.3.2 Horizontal relations

***Definition* 10.1.10.** *Let p_1, p_2 be two subobjects of a complete object of type C. We say that* p_1 occurs before p_2 *($p_1 \prec_{C[n]} p_2$) if, and only if, either of the following is true:*
– *there is a most-derived object p' subobject of the complete object C such that p_1, p_2 are two virtual bases of p' in inheritance graph order*
– *there is a subobject p' of the complete object C such that p_1 and p_2 are two direct non-virtual bases of p' in declaration order*
– *p_1 and p_2 are two cells of the same array field, in the order of their indexes within the array*
– *p_1 and p_2 are two cells of two different fields in declaration order*

– *there is a most-derived object p' subobject of the complete object C such that p_1 is a virtual base, and p_2 is a direct non-virtual base of p' or a cell of an array field of p'*
– *there is a subobject p' of the complete object C such that p_1 is a direct non-virtual base of p' and p_2 is a cell of an array field of p'*

Virtual base (if p most-derived)	*Direct non-virtual base*	*Structure array field*
p_1, p_2 *inheritance graph order*		
p_1	p_2	
	p_1, p_2 *declaration order*	
	p_1	p_2
		p_1, p_2 *field declaration order or cell index order of the same field*

This definition is consistent insofar as it only depends on the type of the complete object.

Invariant 10.1.2 (High-level invariant: horizontal relations on construction states). *Let p_1, p_2 two subobjects such that $p_1 \prec_{C[n]} p_2$. Then, the following table relates their construction states:*

If p_1 is...	*Then p_2 is...*
Unconstructed	
StartedConstructing	Unconstructed
BasesConstructed	
Constructed	*in an arbitrary state*
StartedDestructing	
DestructingBases	Destructed
Destructed	

More concisely, for any state s:

$$\mathsf{ConstrState}_s(\ell, p_1) < \mathsf{Constructed} \Rightarrow \mathsf{ConstrState}_s(\ell, p_2) = \mathsf{Unconstructed}$$

$$\mathsf{ConstrState}_s(\ell, p_1) > \mathsf{Constructed} \Rightarrow \mathsf{ConstrState}_s(\ell, p_2) = \mathsf{Destructed}$$

Corollary 10.1.2. *In particular, by contraposition, if two subobjects $p_1 \prec_{C[n]} p_2$, then the lifetime of p_2 is included in the lifetime of p_1:*

$$\mathsf{ConstrState}_s(\ell, p_2) = \mathsf{Constructed} \Rightarrow \mathsf{ConstrState}_s(\ell, p_1) = \mathsf{Constructed}$$

10.2 Progress

To show that our rules make sense, i.e. that they do not forget any bases to construct, we showed a sanity-check *progress* theorem [4] [5]: if a class C and all its children have no user-defined constructors, then the construction and destruction of an instance of C always succeeds.

***Definition* 10.2.1 (Nearly trivial constructor).** *We say that a class C has a* nearly trivial constructor *if, and only if, all the following conditions hold:*

- *C has a* default *constructor (with no arguments), having only the following initializers:*
  - *bases and structure array fields are initialized through a call to their default constructor (without arguments), without prior statement*
  - *scalar fields are initialized only with built-in operations without arguments*
- *all virtual bases and direct non-virtual bases of C have nearly trivial constructors*
- *for each structure array field f of C, if f has type B, then B has a nearly trivial constructor*

There is no exactly corresponding notion of the Standard: the latter defines a notion of *trivial constructor* implying that the class C has no virtual bases. In practice, the Standard puts this restriction to allow compilers to produce no code for such classes (e.g. PODs), which is not possible in the presence of virtual bases, as they require additional dynamic data (e.g. pointer to virtual tables) to be initialized, as we pointed out in Section 5.2.2 (p. 96) and Section 5.5.5 (p. 117).

However, the Standard notion of trivial constructor is a particular case of nearly trivial constructors: the conditions are exactly the same, with the further requirements that C have neither virtual bases, nor virtual functions. In other words, a class has a trivial constructor if, and only if, it is a non-dynamic class (in the sense of *Definition* 4.4.2 p. 89, it "has no polymorphic behaviour" following the Standard) with a nearly trivial constructor.

Theorem II.1 (Construction progress). *If C is a class having a nearly trivial constructor, then the allocation of a new array of C calling the default constructor for each cell always succeeds, with all cells becoming* Constructed*.*

The same theorem also holds for destruction. But here, we may directly take the Standard notion of *trivial destructor*, as the destructor has no arguments:

***Definition* 10.2.2 (Trivial destructor).** *A class C has a* trivial destructor *if, and only if, all the following conditions hold:*

- *its destructor immediately* `return`*s*
- *all virtual bases and direct non-virtual bases of C have trivial destructors*
- *for each structure array field f of C, if f has type B, then B has a trivial destructor*

Theorem II.2 (Destruction progress). *If C is a class having a trivial destructor, then the deallocation of a constructed array of C always succeeds, with all cells becoming* Destructed*.*

4. Coq development: theory `Progress`.
5. Coq development: theory `ProgressInv`.

10.3 RAII: Resource Acquisition is Initialization

In this section, we investigate how the lifetime of an object is related to the scope of its most-derived object, and how the lifetime of an object is related to the lifetime of its subobjects. In particular, we aim at showing that destruction of any two subobjects is performed in the reverse order of their construction.

10.3.1 Increase

We first investigate, for a given object, in which order it goes through its construction states[6]. The following theorem gives us a fine-grained viewpoint on the evolution of the construction state of a subobject:

Theorem II.3 (Construction order increment). *If $s \to s'$ is a transition step of the κ++ small-step semantics, and if the construction state of (ℓ, p) goes from c to $c' \neq c$, then $c' = \mathsf{S}(c)$ and any other subobject $p' \neq p$ keeps its construction state unchanged.*

Proof. By case analysis on the small-step semantic rules. □

By transitivity, we easily obtain the following corollary:

Theorem II.4 (Objects are not constructed more than once). *Any subobject is never constructed more than once: given a construction state c, no subobject ever changes its construction state twice to c.*

In particular, any *virtual base* subobject is constructed at most once.

Corollary 10.3.1. *By contraposition, if $s \to^* s'$ and if $\mathsf{ConstrState}_s(\ell, p) = \mathsf{ConstrState}_{s'}(\ell, p)$, then the construction state of (ℓ, p) remains unchanged between s and s': for any state s'' such that $s \to^* s'' \to^* s'$, we have $\mathsf{ConstrState}_s(\ell, p) = \mathsf{ConstrState}_{s''}(\ell, p)$*

In particular, in a given execution sequence, the lifetime of any object is such a state interval. More precisely, we exactly know that an object skips no construction states. In other words, if an object goes from one construction state to another, then it must go through all construction states in between:

Corollary 10.3.2 (Intermediate values theorem for construction states). *If $s \to^* s'$, then, for any subobject (ℓ, p), and for any construction state c'' such that:*

$$\mathsf{ConstrState}_s(\ell, p) \leq c'' < \mathsf{S}(c'') \leq \mathsf{ConstrState}_{s'}(\ell, p)$$

there exist "changing states" s''_1, s''_2 such that:

$$s \to^* s''_1 \to s''_2 \to^* s'$$

and $\mathsf{ConstrState}_{s''_1}(\ell, p) = c''$ and $\mathsf{ConstrState}_{s''_2}(\ell, p) = \mathsf{S}(c'')$.

6. Coq development: theory `Constrorder`.

10.3.2 Construction and destruction order of two subobjects of the same complete object

In this section, we shall prove that two subobjects of the same complete object are destructed in the reverse order of their construction[7].

10.3.2.1 General theorem

Assume that, for any class C and any $n \in \mathbb{N}^*$, there exists a *static* relation $\lhd_{C[n]}$ (i.e. independent on the execution state) on generalized subobjects of a full object of type $C[n]$, such that for any generalized subobjects p_1, p_2 of $C[n]$, the following conditions hold:

- $p_1 \lhd_{C[n]} p_2 \vee p_2 \lhd_{C[n]} p_1$ (i.e. $\lhd_{C[n]}$ is total)
- for any execution state s, and for any complete object ℓ of type $C[n]$, if $p_1 \lhd_{C[n]} p_2$ and $\mathsf{ConstrState}_s(\ell, p_2) = \mathsf{Constructed}$, then $\mathsf{ConstrState}_s(\ell, p_1) = \mathsf{Constructed}$

Then, we may show the following:

Theorem II.5 (Subobjects are destructed in the reverse order of their construction). *Let ℓ be a complete object of type $C[n]$. If p_1 and p_2 are two generalized subobjects of complete object ℓ, such that p_1 was constructed before p_2, then, if p_1 is being destructed, then p_2 was destructed before p_1. In other words, if s_1 is constructed before s_2, then the lifetime of s_2 is included in the lifetime of s_1 .*

$$\begin{tikzcd}
s_0 \arrow[r] & s_1 \arrow[r, "*"] & s_2 \arrow[r] & s_3 \arrow[rr, "*"] & & s_4 \arrow[r] & s_5 \\
\neg c_1 & c_1 & \neg c_2 & c_2 & & c_1 & \neg c_1
\end{tikzcd}$$

$$\Downarrow$$

$$\exists s'_3, s'_4$$

$$\begin{tikzcd}
s_3 \arrow[r, "*"] & s'_3 \arrow[r] & s'_4 \arrow[r, "*"] & s_4 \\
 & c_2 & \neg c_2 &
\end{tikzcd}$$

(c_i means "p_i is $\mathsf{Constructed}$ *at this state")*

Proof. First we show that $p_1 \lhd_{C[n]} p_2$. As $\lhd_{C[n]}$ is total, we may reason by case analysis. Assume $p_2 \lhd_{C[n]} p_1$. Then, at state s_1, p_1 is $\mathsf{Constructed}$, so p_2 is also $\mathsf{Constructed}$ at s_1. But construction states are increasing, so p_2 is at least $\mathsf{Constructed}$ at s_2. As it is $\mathsf{Constructed}$ at s_3, it is necessarily $\mathsf{Constructed}$ at s_2, which is absurd. So, necessarily, as $\lhd_{C[n]}$ is total, we have $p_1 \lhd_{C[n]} p_2$.

We immediately see that $p_1 \neq p_2$. Indeed, if $p_1 = p_2$, then, as $\mathsf{Constructed}$ in s_1 and also in s_3, p_2 would also be $\mathsf{Constructed}$ in s_2, which is absurd.

Now we show that p_2 is no longer $\mathsf{Constructed}$ at s_4. As $\leq$ is total on construction states, we may reason by case analysis. Assume p_2 is at most $\mathsf{Constructed}$ at s_4. Then, p_2 is $\mathsf{Constructed}$ at s_4 (increase from s_3). But $p_1 \neq p_2$ and $s_4 \to s_5$ changes the construction state of p_1. Then p_2 is $\mathsf{Constructed}$ also at s_5. But $p_1 \lhd_{C[n]} p_2$, so p_1 is also $\mathsf{Constructed}$ at s_5, which is absurd.

To sum up, p_2 is no longer $\mathsf{Constructed}$ at s_4, but it is $\mathsf{Constructed}$ at s_3. So, by the intermediate values theorem, there exists $s_3 \to^* s'_3 \to s'_4 \to^* s_4$ such that step $s'_3 \to s'_4$ makes p_2 from $\mathsf{Constructed}$ to $\mathsf{StartedDestructing}$, which concludes. □

7. Coq development: theory `ConstrSubobjectOrdering`.

10.3.2.2 Application: subobject ordering

It only remains to find such a relation $\lhd_{C[n]}$. Here we show that the *subobject lifetime relation*, the depth-first left-to-right traversal of the subobject tree of Figure 9.1 (p. 173), is suitable.

***Definition* 10.3.1.** *Let p_1, p_2 two generalized subobjects of $C[n]$. We say that p_1 is* included *in p_2 (written $p_1 \subseteq_{C[n]} p_2$) if, and only if, either $p_1 = p_2$, or there exists a generalized subobject p of $C[n]$ such that p_1 is a direct subobject of p and p is included in p_2.*

Roughly speaking, this inclusion relation $\subseteq_{C[n]}$ is the "reflexive and transitive closure" of the "direct subobject" relation. It expresses the notion of path in the subobject tree of Figure 9.1 (p. 173).

However, this notion is distinct from the notion of inheritance and array paths: if p is not a most-derived object, then it does not include the subobjects corresponding to its virtual bases (only paths within the *non-virtual part* of the subobject tree are to be considered). Nevertheless, if p is a most-derived object, it does include its virtual bases, so all of its subobjects.

By transitivity, it follows that:

Lemma 10.3.3. *f $p_1 \subseteq_{C[n]} p_2$, then the following table relates their construction states:*

If p_2 is...	*Then p_1 is...*
Unconstructed	Unconstructed
Constructed	Constructed
Destructed	Destructed

Finally, we define the *subobject lifetime relation* $\lhd_{C[n]}$ as follows.

***Definition* 10.3.2.** *Let p_1, p_2 be two subobjects of $C[n]$. We say that p_1* lays before *p_2, written $p_1 \lhd_{C[n]} p_2$, if, and only if, either condition holds:*
- *$p_1 \subseteq_{C[n]} p_2$*
- *there exist two sibling subobjects $p'_1 \prec^{\mathcal{D}}_{C[n]} p'_2$ such that $p_1 \subseteq_{C[n]} p'_1$ and $p_2 \subseteq_{C[n]} p'_2$*

In fact, this definition says that p_1 lays before p_2 if, and only if, p_1 appears before p_2 in a depth-first left-to-right traversal of the subobject tree. However, we do not need to prove that it is an order.

Lemma 10.3.4 ($\lhd_{C[n]}$ is total).

$$\forall B_1, B_2, p_1, p_2 : \left.\begin{array}{l} C[n] \prec\!\langle p_1 \rangle\!\succ\!\to B_1 \\ C[n] \prec\!\langle p_2 \rangle\!\succ\!\to B_2 \end{array}\right\} \Rightarrow \left(\begin{array}{l} \quad p_1 \lhd_{C[n]} p_2 \\ \vee\ p_2 \lhd_{C[n]} p_1 \end{array} \right.$$

Proof. Long and tedious case analysis. □

Lemma 10.3.5. *The subobject lifetime relation is compatible with subobject lifetimes: if $p_1 \lhd_{C[n]} p_2$, then, for any execution state, and for any complete object ℓ of type $C[n]$, whenever (ℓ, p_2) is* Constructed*, then (ℓ, p_1) is* Constructed.

Proof.
- If $p_1 \subseteq_{C[n]} p_2$, then the previous lemma directly applies.
- Otherwise, let $p'_1 \prec^{\mathcal{D}}_{C[n]} p'_2$ be two subobjects such that $p_1 \subseteq_{C[n]} p'_1$ and $p_2 \subseteq_{C[n]} p'_2$. We first show that p'_1 is Constructed. Let c_1 the construction state of p'_1. As $\leq$ is total on construction states, we have two cases:

- if $c_1 < \mathsf{Constructed}$, then p'_2 is $\mathsf{Unconstructed}$, so p_2 is also $\mathsf{Unconstructed}$, which is absurd.
- if $c_1 > \mathsf{Constructed}$, then p'_2 is $\mathsf{Destructed}$, so p_2 is also $\mathsf{Destructed}$, which is absurd.

So p'_1 is $\mathsf{Constructed}$. Thus, p_1 is also $\mathsf{Constructed}$, which concludes. □

The immediate corollary follows from those two lemmata:

Theorem II.6 (Lifetimes of subobjects of the same complete object). *The lifetimes of two subobjects of the same complete object are either included in one another, or disjoint.*

10.3.2.3 Subobject ordering, inheritance and aggregation

Now it remains to relate the order of construction of subobjects with the notions of inheritance and aggregation.

When constructing the virtual bases of a most-derived object, the Standard prescribes an order called *inheritance graph order*, modelled as follows:

***Definition* 10.3.3.** *If the class hierarchy is well-founded, then the following recursive function VO:*

$$\begin{array}{rcl} (\{\mathsf{Repeated}, \mathsf{Shared}\} \times \mathcal{C})^* & \overset{VO}{\to} & (\mathcal{C})^* \\ VO(\epsilon) & = & \epsilon \\ VO((\mathsf{Repeated}, B) :: q) & \underset{\text{def.}}{=\!=} & VO(\mathcal{D}(B)) +\!\!+' VO(q) \\ VO((\mathsf{Shared}, B) :: q) & \underset{\text{def.}}{=\!=} & VO(\mathcal{D}(B)) +\!\!+' (B :: VO(q)) \end{array}$$

is well-defined.

VO actually performs a depth-first search of all virtual bases *induced* by l, including the classes that are elements of l declared as “virtual bases”, but quoting each virtual base only once.

***Definition* 10.3.4.** *For any class C, we pose $\mathcal{VO}(C) \underset{\text{def.}}{=\!=} VO(\mathcal{D}(C))$. $\mathcal{VO}$ is called the* virtual base ordering *function.*

***Definition* 10.3.5.** *Two virtual bases A and B of C are in* inheritance graph order *(written $A \prec_C^{\mathcal{V}} B$) if, and only if, A occurs before B in $\mathcal{VO}(C)$.*

We show that this way of computing the list of the virtual bases of a class satisfies the requirement of Hypothesis 9.4.2 (p. 190):

Lemma 10.3.6. *$\mathcal{VO}(C)$ contains all the virtual bases of C exactly once each, and only them.*

Lemma 10.3.7. *If B is a virtual base of C, then for any virtual base A of B, $A \prec_C^{\mathcal{V}} B$.*

Proof. It suffices to show that, for any list l such that $B \in VO(l)$, A occurs before B in $VO(l)$. Reason by induction on the definition of VO. There are two cases:

- Let $l = (\mathsf{Repeated}, B') :: q$, or $l = (\mathsf{Shared}, B') :: q$ with $B' \neq B$. If $B \in VO(\mathcal{D}(B'))$, then, by induction hypothesis, A occurs before B in $VO(\mathcal{D}(B'))$. Otherwise, either $A \in VO(\mathcal{D}(B'))$, or A occurs before B in $VO(q)$ by induction hypothesis.

- Let $l = (\mathsf{Shared}, B) :: q$. Then, by LEMMA 10.3.6 (p. 214), $A \in VO(\mathcal{D}(B))$ but $B \notin VO(\mathcal{D}(B))$ (otherwise B would be a virtual base of itself, which would contradict the well-formedness of the hierarchy), which concludes. □

Consequently:

LEMMA 10.3.8. *Let p be a subobject of a complete object $C[n]$. If p' is an inheritance subobject of p, then the lifetime of p is included in the lifetime of p'.*

Proof. There are two cases:

- If p' is a non-virtual base-class subobject of p, then $p' \subseteq_{C[n]} p$, so a previous lemma applies.
- Otherwise, if p' is a virtual base-class subobject of p, then there are two cases:
  - If p is a most-derived object, then $p' \subseteq_{C[n]} p$.
  - Otherwise, we can show that $p' \lhd_{C[n]} p$. Let A be the static type of p. Then, p' is a non-virtual base-class subobject of some virtual base V of A. There are two cases:
    - If p is a non-virtual base-class subobject of its most-derived object, then it is a non-virtual base-class subobject of some base B; so let p_V and p_B represent the direct subobjects of the (common) most-derived object of p and p' for B and V, so that $p' \subseteq_{C[n]} p_V$ and $p \subseteq_{C[n]} p_B$. As B is a non-virtual base and V is a virtual base, then we have $p_V \prec_{C[n]} p_B$, which concludes.
    - Otherwise, p is a virtual base-class subobject of its most-derived object, then it is a non-virtual base-class subobject of some virtual base B of the most-derived object. By transitivity, V is a virtual base of B, so $p_V \prec_{C[n]} p_B$ by the above lemma, and we can conclude similarly as the previous case. □

LEMMA 10.3.9. *If $p_1 \subseteq_{C[n]} p_2 \lhd_{C[n]} p_3$, then $p_1 \lhd_{C[n]} p_3$.*

Proof. By definition of $\lhd_{C[n]}$ and by transitivity of $\subseteq_{C[n]}$. □

THEOREM II.7 (The lifetime of an object is included in the lifetimes of its subobjects). *Let p be a subobject of a complete object $C[n]$, and p' be a subobject of p. Then, the lifetime of p is included in the lifetime of p'.*

Proof. There are two cases:

- If p' is an inheritance subobject of p, then LEMMA 10.3.8 (p. 215) applies.
- Otherwise, $p' \subseteq_{C[n]} p_f \subseteq_{C[n]} p_B$ where p_f is an array cell of some field f of some inheritance subobject p_B of p, so we may conclude by LEMMA 10.3.9 (p. 215). □

10.3.3 Formal account of RAII

We can now prove general properties about the RAII paradigm.

THEOREM II.8 (Subobjects are constructed and destructed before deallocation). *If a subobject has its complete object deallocated, then the subobject has been constructed and destructed before, in this order.*

Proof. Let $s \to s'$ be the deallocation step of an object ℓ. Then, by the low-level invariant, we know that the construction state of ℓ is $\mathsf{Destructed}$ at s, so is it for any subobject p of ℓ. As it is $\mathsf{Unconstructed}$ at the initial state of the program, then, by the intermediate values theorem, (ℓ, p) passes through a step $\mathsf{Constructed} \to \mathsf{StartedDestructing}$, which corresponds to entering the destructor. Again, before this step, (ℓ, p) passes through a step $\mathsf{BasesConstructed} \to \mathsf{Constructed}$, which corresponds to leaving the constructor body. □

THEOREM II.9 (Objects destructed when program exits). *At the end of the program, all objects were destructed.*

Proof. It suffices to show that at the final step, there are no allocated objects. This can be shown thanks to the structure of the final step. The result then follows from the above lemma. □

This theorem is not true in the presence of a free store (nothing guarantees that `delete` has been called for each dynamically allocated object).

10.3.4 Subobjects of different complete objects

In general, in a real-world C++ program (except for embedded systems, where dynamic memory allocation is not necessarily permitted), there is no information about whether two complete objects created by `new` have their lifetimes included, disjoint or overlapping.

However, in our model where all objects are in stack, RAII can be extended to a *stack discipline* for object lifetimes [8].

LEMMA 10.3.10. *Consider an object location ℓ. If it is valid:*

$$\mathcal{G}.\mathsf{LocType}(\ell) = (C, n)$$

but outside the list $\Omega(S, \mathcal{K})$ of stack objects, then all n cells of ℓ are $\mathsf{Destructed}$.

Proof. This can be proved as an additional run-time invariant. It needs, however, the run-time invariant about the precise construction states of objects (kind invariants): for the particular step (κ++-destr-array-nil-kcontinue, p. 193) when an object is about to be deallocated, this object must be $\mathsf{Destructed}$. □

Hypothesis 10.3.1. *We assume a total order over object locations, such that the operation "retrieve a new fresh location in the object store" be* strictly increasing.

In practice, object locations may range over $\mathbb{Z}$, for instance. This is the case in our Coq development.

LEMMA 10.3.11. *If $s \to s'$ is a step changing the construction state of a subobject (ℓ, p), then there can be no object $\ell' > \ell$ in the set of allocated objects.*

Proof. It suffices to show that the set of allocated objects forms an *ordered stack* w.r.t. $<$. This can be proved as an invariant along with the run-time invariant.

Then, operations over construction states only modify the top-most object of this stack, which is maximal w.r.t. $<$. □

8. Coq development: theory `ConstrorderOther`.

Lemma 10.3.12. *If $s \to^* s'$ and if ℓ is a complete object belonging to the set of allocated objects for states s and s', then, for any subobject (ℓ', p') such that $\ell' < \ell$, the construction state of (ℓ', p') does not change between s and s'.*

Proof. Follows from the above lemma, by transitivity. □

Lemma 10.3.13. *If, between s and s', an object in the allocation set of s is no longer in the allocation set of s', then it is deallocated between s and s'.*

Proof. Trivial induction on the length of the execution path $s \to^* s'$. □

Lemma 10.3.14. *Let ℓ be an allocated object, and $s_0 \to s$ be an allocation step of some object $\ell' \neq \ell$. Then, if $s \to^* s'$ and if (ℓ, p) changes its construction state between s and s', then ℓ' is deallocated between s and s'.*

Proof. The small-step semantic rule for object allocation $s_0 \to s$ only allocates ℓ', with ℓ already allocated, so $\ell' > \ell$.

Let c be the construction state of (ℓ, p) at s. Then, by the intermediate values theorem, there exists $s \to^* s_1 \to s_2 \to^* s'$ such that $s_1 \to s_2$ makes (ℓ, p) from c to $\mathsf{S}(c)$. At this step, ℓ is necessarily the top-most object on the allocation stack, so in particular, at s_1, ℓ' is no longer allocated. So, by the previous lemma, there exists a deallocation step for ℓ' between s and s_1. □

Lemma 10.3.12 (p. 217) and Lemma 10.3.14 (p. 217) give us the following corollary:

Theorem II.10 (Lifetimes of subobjects of two different complete objects). *The lifetimes of two subobjects of different complete objects are either disjoint, or included in one another.*

10.4 Safety of scalar field accesses

In κ++, reading the contents of a scalar field puts no precondition on the construction state of the field. We can, however, show[9] that (κ++-field-scalar-read, p. 185) gets stuck whenever the field is not in the Constructed state:

Theorem II.11 (Safety of scalar field accesses). *If a scalar field has a value, then it is* Constructed*.*

Proof. There are only three rules modifying the value of a field:
- (κ++-field-scalar-write, p. 186) explicitly requires the field being Constructed
- (κ++-initscalar, p. 191), giving the field its initial value, switches the field construction state to Constructed
- (κ++-destr-fields-cons-scalar, p. 194) erases the value of the field, so the hypothesis no longer holds

The run-time invariant is needed to discriminate between a scalar and a structure field when its construction state changes. □

9. Coq development: theory `ScalarFields`.

In particular, if a scalar field has a value, then the corresponding complete object is not deallocated. This means that, once an object is deallocated, it has actually been turned back into raw memory. However, it is possible to go further in showing the safety of field accesses, once (κ++-constr-cons-field-scalar-no-init, p. 192) is disabled, enforcing each scalar field to be explicitly initialized during object construction.

THEOREM II.12 (Strong safety of scalar field accesses). *If rule (κ++-constr-cons-field-scalar-no-init, p. 192) is disabled, then a field has a value if, and only if, it is* Constructed*.*

10.5 The generalized dynamic type of a subobject

In this section, we study the properties of the *generalized dynamic type* (defined in Section 9.4.3.4 p. 187) of a subobject, i.e. the type of the class considered as most-derived object for the purpose of polymorphic operations.

10.5.1 Safety of virtual function calls

Our definition of the generalized dynamic type is a ground to the safety of virtual function calls:

THEOREM II.13 (Safety of virtual function calls). *Whenever a virtual function is called, the subobject bound to its* `this` *parameter is in state* BasesConstructed*,* Constructed *or* StartedDestructing*, so that all its base class subobjects are* Constructed*.*

Proof. Rule (κ++-virtual-funcall, p. 188) governs κ++ virtual function calls of the form x->$_C f(\dots)$. It requires the generalized dynamic type of the subobject σ referred to by x to be well-defined; then, it adjusts the `this` pointer within the function to an inheritance subobject $\sigma@\sigma'$ of σ. There are two cases:

– If the most-derived object of which σ is an inheritance subobject is Constructed, then by LEMMA 10.3.8 (p. 215), σ and $\sigma@\sigma'$ as well.
– Otherwise, by (κ++-dyntype-pending, p. 188), there exists a subobject $\sigma_\circ$ such that $\sigma = \sigma_\circ@\sigma_1$, and $\sigma_\circ$ is in state BasesConstructed or StartedConstructing. If $\sigma@\sigma' = \sigma_\circ$ (i.e. the subobject referred to by x is exactly the generalized dynamic type, and at the same time requires no `this` pointer adjustment), then the result is trivial. Otherwise, $\sigma@\sigma'$ is an inheritance subobject of $\sigma_\circ$, so we can conclude using INVARIANT 10.1.1 (p. 208) and LEMMA 10.3.8 (p. 215). □

10.5.2 Unicity

We aim at showing that, for any most-derived object, there is at most one inheritance subobject that can play the role of generalized dynamic type for a given execution state[10].

So, tailoring rules (κ++-dyntype-constructed, p. 187) and (κ++-dyntype-pending, p. 188), we can redefine the notion of generalized dynamic type only depending on the most-derived object.

10. Coq development: theory `Dyntype`.

By language abuse, we say that σ is the *generalized dynamic type of the structure array cell* (ℓ, α, i) and we write $\mathsf{getgDynType}(\ell, \alpha, i, \sigma)$:

$$\frac{\begin{array}{c}\mathcal{G}.\mathsf{LocType}(\ell) = (D, n) \\ D[n] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\mapsto} C[m] \qquad \mathcal{G}.\mathsf{ConstrState}(\ell, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon))) = \mathsf{Constructed}\end{array}}{\mathcal{G} \vdash \mathsf{getgDynType}(\ell, \alpha, i, (\mathsf{Repeated}, C :: \epsilon))}$$

(κ++-getgdyntype-constructed)

$$\frac{\begin{array}{c}\mathcal{G}.\mathsf{LocType}(\ell) = (D, n) \qquad D[n] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\mapsto} C[m] \prec\!\langle(i, \sigma')\rangle\!\overset{\mathcal{CI}}{\mapsto} B' \\ \mathcal{G}.\mathsf{ConstrState}(\ell, (\alpha, i, \sigma')) = c \qquad c = \mathsf{BasesConstructed} \vee c = \mathsf{StartedDestructing}\end{array}}{\mathcal{G} \vdash \mathsf{getgDynType}(\ell, \alpha, i, \sigma')}$$

(κ++-getgdyntype-pending)

Immediately, we then have that:

Lemma 10.5.1. *Let ℓ be a complete object of type $D[n]$, and α such that $D[n] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\mapsto} C[m]$ and $0 \leq i < m$. Let σ' be an inheritance subobject of C of static type B. Then, if σ' is the generalized dynamic type of the structure array cell (ℓ, α, i), then, for any inheritance subobject σ'' of B, σ' is the generalized dynamic type of the subobject $(\ell, (\alpha, i, \sigma'@\sigma''))$:*

$$\mathsf{getgDynType}(\ell, \alpha, i, \sigma') \Rightarrow \mathsf{gDynType}(\ell, \alpha, i, \sigma'@\sigma'', B, \sigma', \sigma'')$$

Lemma 10.5.2. *Conversely, for any inheritance subobject σ of C, if σ' is the generalized dynamic type of $(\ell, (\alpha, i, \sigma))$ such that* $\mathsf{gDynType}(\ell, \alpha, i, \sigma, B, \sigma', \sigma'')$ *for some B and σ'', then σ' is the generalized dynamic type of the array cell (ℓ, α, i) and there is an inheritance subobject σ'' of σ' such that $\sigma = \sigma'@\sigma''$.*

$$\mathsf{gDynType}(\ell, \alpha, i, \sigma, B, \sigma', \sigma'') \Rightarrow \mathsf{getgDynType}(\ell, \alpha, i, \sigma') \wedge \sigma = \sigma'@\sigma''$$

In other words, the generalized dynamic type can be obtained using the getgDynType predicate, whereas gDynType can be used to determine whether a subobject has its generalized dynamic type defined, and how the corresponding inheritance subobject of the generalized dynamic type can be deduced.

Moreover, an inheritance subobject has its generalized dynamic type defined only if it is a base of the generalized dynamic type of the array cell. Indeed, consider the following example:

```
struct A             {virtual void f ();};
struct B1: virtual A {};
struct B2: virtual A {virtual void f ();};
struct C: B1, B2 {}
```

Consider an instance of C. Then, during the execution of the constructor body of its base B_2, the corresponding B_2 subobject is BasesConstructed, so it is the generalized dynamic type of the array cell. But, even though the subobject B_1 is already Constructed, its generalized dynamic type is undefined, as B_1 is not a base of B_2. So, calling f on B_1 has undefined behaviour.

Now, we can reason about the generalized dynamic type of an array cell instead of considering the generalized dynamic type of a subobject.

Lemma 10.5.3. *Considering a most-derived object, there can be at most one* inheritance *subobject in construction state* BasesConstructed *or* StartedDestructing.

Proof. If there are two of them, say p_1 and p_2, then there are two cases:
– say p_2 is a base of p_1. Then, as p_1 is BasesConstructed or StartedDestructing, all its bases are Constructed, in particular p_2, which is absurd.
– otherwise, there is a subobject p and two direct bases p'_1, p'_2, say in this order, such that each p_i is a base of p'_i. Then, p'_2 is Unconstructed or Destructed, so p_2 as well, which is absurd. □

Lemma 10.5.4. *The generalized dynamic type of an array cell, if any, is unique.*

Proof. If the most-derived object is Constructed, then the result is trivial. Otherwise, it follows from the lemma above. □

Those lemmata entail the following immediate corollary:

Theorem II.14 (Unicity of generalized dynamic type). *The generalized dynamic type of a subobject, if any, is unique.*

10.5.3 Evolution

However, the generalized dynamic type of an object does not *continuously* exist: during the lifetime of the subobject, while the most-derived object is not yet Constructed, the generalized dynamic type of the array cell does exist only if there is an inheritance subobject that is BasesConstructed or StartedDestructing.

Indeed, in the above example, after exiting from the body of the constructor for B_1, but before entering the body of the constructor for B_2, there is no constructor body in progress for the instance of C, so there is no generalized dynamic type for the instance C.

Lemma 10.5.5. *The following table summarizes the evolution of the dynamic type of an array cell (ℓ, α, i) of type C (writing $\sigma_\circ =$ (Repeated, $C :: \epsilon$) the corresponding most-derived object)*

When the subobject	*goes from*	*to*	*then the dynamic type of (ℓ,α,i) goes*	
			from	*to*
$(\ell, (\alpha, i, \sigma))$	Unconstructed	StartedConstructing	*Undefined*	*Undefined*
$(\ell, (\alpha, i, \sigma))$	StartedConstructing	BasesConstructed	*Undefined*	σ
$(\ell, (\alpha, i, \sigma))$ *with* $\sigma \neq \sigma_\circ$	BasesConstructed	Constructed	σ	*Undefined*
$(\ell, \alpha, i, \sigma_\circ)$	BasesConstructed	Constructed	$\sigma_\circ$	$\sigma_\circ$
$(\ell, \alpha, i, \sigma_\circ)$	Constructed	StartedDestructing	$\sigma_\circ$	$\sigma_\circ$
$(\ell, (\alpha, i, \sigma))$ *with* $\sigma \neq \sigma_\circ$	Constructed	StartedDestructing	*Undefined*	σ
$(\ell, \alpha, i, \sigma)$	StartedDestructing	DestructingBases	σ	*Undefined*
$(\ell, \alpha, i, \sigma)$	DestructingBases	Destructed	*Undefined*	*Undefined*
$(\ell', (\alpha', i', \sigma'))$ *with* $(\ell, \alpha, i) \neq (\ell', \alpha', i')$	*Any*	*Any*	*Does not change*	

Theorem II.15 (Evolution of the generalized dynamic type). *The following table summarizes the evolution of the dynamic types of a subobject depending on the evolution of construction states.*

When the subobject	*goes from*	*to*	*then the dynamic type of*	*goes from*	*to*
$(\ell, (\alpha, i, \sigma))$	Unconstructed	StartedConstructing	$(\ell, (\alpha, i, \sigma'))$	*Undef.*	*Undef.*
$(\ell, (\alpha, i, \sigma))$	StartedConstructing	BasesConstructed	$(\ell, (\alpha, i, \sigma @ \sigma''))$	*Undef.*	σ
			$(\ell, (\alpha, i, \sigma'))$ *not a base of* σ	*Undef.*	*Undef.*
$(\ell, (\alpha, i, \sigma))$ *with* $\sigma \neq \sigma_\circ$	BasesConstructed	Constructed	$(\ell, (\alpha, i, \sigma @ \sigma''))$	σ	*Undef.*
			$(\ell, (\alpha, i, \sigma'))$ *not a base of* σ	*Undef.*	*Undef.*
$(\ell, (\alpha, i, \sigma_\circ))$	BasesConstructed	Constructed	$(\ell, (\alpha, i, \sigma'))$	$\sigma_\circ$	$\sigma_\circ$
$(\ell, (\alpha, i, \sigma_\circ))$	Constructed	StartedDestructing	$(\ell, (\alpha, i, \sigma'))$	$\sigma_\circ$	$\sigma_\circ$
$(\ell, (\alpha, i, \sigma))$ *with* $\sigma \neq \sigma_\circ$	Constructed	StartedDestructing	$(\ell, (\alpha, i, (\sigma @ \sigma'')))$	*Undef.*	σ
			$(\ell, (\alpha, i, \sigma'))$ *not a base of* σ	*Undef.*	*Undef.*
$(\ell, \alpha, i, \sigma)$	StartedDestructing	DestructingBases	$(\ell, ((\alpha, i, (\sigma @ \sigma'')))$	σ	*Undef.*
			$(\ell, (\alpha, i, \sigma'))$ *not a base of* σ	*Undef.*	*Undef.*
$(\ell, (\alpha, i, \sigma))$	DestructingBases	Destructed	$(\ell, (\alpha, i, \sigma'))$	*Undef.*	*Undef.*
$(\ell, (\alpha, i, \sigma))$	*Any*	*Any*	$(\ell', (\alpha', i', \sigma'))$ *with* $(\ell, \alpha, i) \neq (\ell', \alpha', i')$	*Does not change*	

In more detail:

Lemma 10.5.6. *When a subobject p becomes* BasesConstructed *or* StartedDestructing*:*

- *its dynamic type changes and becomes defined, as well as the dynamic type of all of its bases.*
- *the dynamic type of all other subobjects (which are not bases of p) cannot change to a defined value.*

Proof.
- The first case is obvious, as the dynamic type cannot be p before it becomes BasesConstructed.
- In the second case, consider a subobject p'' which is not a base of p. If its dynamic type is, say, p', then, necessarily, p' is in state BasesConstructed or StartedDestructing (as the most-derived object cannot be Constructed). By unicity, $p' = p$, which is absurd. □

Conversely:

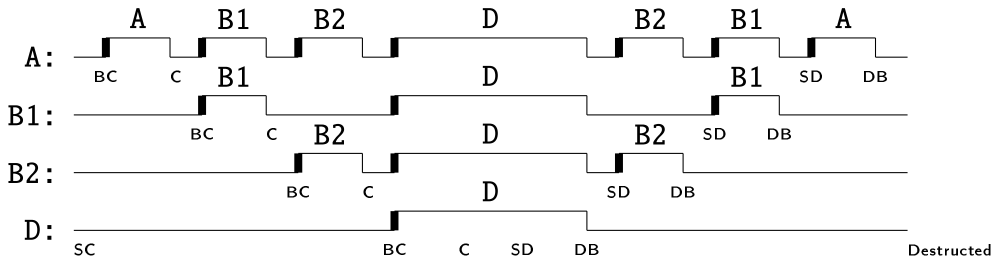
Lemma 10.5.7.
- *If the most-derived object becomes* Constructed*, then nothing happens on the dynamic types.*
- *Otherwise, if a subobject p becomes other than* BasesConstructed *or* StartedDestructing*, then the dynamic type of an object cannot change to a defined value.*

– *Otherwise, if no subobject changes its construction state, then no dynamic type changes.*

Proof. – In the first case, the construction state of the most-derived object passes from BasesConstructed to Constructed. So, in both cases, the dynamic type of all bases has already switched to the most-derived object.
– In the second case, consider a subobject p'' whose dynamic type becomes defined as a subobject p' in state BasesConstructed or StartedDestructing. Then, $p' \neq p$ (as p is no longer in such a construction state). So, p' was not affected by the construction state change, so it was already BasesConstructed or StartedDestructing before the construction state change. So, the dynamic type of p'' was already p', so it has not changed.
– The third case is trivial. □

The diagram below[11] depicts the evolution of the generalized dynamic type on the given class hierarchy:

```
struct A            {};
struct B1: virtual A {};
struct B2: virtual A {};
struct D:  B1, B2    {};
```



This diagram, grounded on the results of this section, points out the precise times when the generalized dynamic type changes, shown by the thick transitions in the diagram above: when all bases are constructed, and just before the construction of fields, the pointers to virtual tables change for the subobject and all of its bases, as well as entering the destructor. Those execution points correspond to precise times when a compiler implementation has to actually change the dynamic type data to reflect the dynamic type at the implementation level. Indeed, we shall define in Section 11.1.2.2 (p. 238) a "set dynamic type" operation in an intermediate language to which we will compile κ++. Following the diagram above, this operation shall be used when compiling a constructor, before constructing the fields of an object (Section 11.5.9.2 p. 266), and, when compiling a destructor, upon entering it (Section 11.5.10 p. 268).

11. where
SC = StartedConstructing
BC = BasesConstructed
C = Constructed
SD = StartedDestructing
DB = DestructingBases

Chapter 11

Verified compilation of object construction and destruction

In this chapter, we aim at compiling κ++ to a language featuring low-level memory accesses. We propose a compilation strategy, which we describe in Section 11.1 (p. 223), separating the compilation of construction-specific features from the other features related to multiple inheritance (virtual function dispatch, casts).

To this purpose, we define an intermediate language called Ds++, which is a superset of the s++ intermediate language defined in Chapter 4 (p. 71). In addition to C++ multiple inheritance, Ds++ features specific operations to manage the changes of generalized dynamic types during construction, but no longer construction and destruction themselves. We describe the syntax of Ds++ in Section 11.2 (p. 245), its semantic elements in Section 11.3 (p. 246), and its semantic rules in Section 11.4 (p. 248). Then, we build a compiler from κ++ to Ds++ in Section 11.5 (p. 254), which we prove to be correct by exhibiting a compilation invariant in Section 11.6 (p. 271).

Once construction-specific features have been compiled away, we adapt the compiler from s++ to the Vcm target language with low-level memory accesses and virtual tables, which we defined in Chapter 7 (p. 135). To take the additional features of Ds++ into account, we have to enrich Vcm with a new read-only data structure, *virtual table tables*, leading to the CVcm language, which we describe in Section 11.7 (p. 286) Then, in Section 11.8 (p. 293), we compile those additional features, essentially corresponding to navigating through virtual table tables and updating the pointers to virtual tables in objects. Finally, we prove the correctness of this compiler by exhibiting a compilation invariant in Section 11.9 (p. 302).

Our work is machine-checked with the Coq proof assistant. The proofs are available at [71].

11.1 Strategy

In this section, we describe in more detail, but from a higher level, our compilation strategy, to investigate which features are to be added to s++ to define the Ds++ language.

11.1.1 Constructors and destructors

In κ++ constructors and destructors are special member functions, with their own calling conventions: constructors can be called only in initializers during construction, and destructors only during destruction. By contrast, we aim at compiling them to ordinary static (not class member) functions, and call them ordinarily as appropriate, as there is no notion of object construction and destruction in Ds++.

In C++, a class can have several constructors, distinguished by the types of their arguments. By contrast, a class has exactly one destructor, with no argument.

11.1.1.1 Scope

In C++, an object declared within a scope block has to be constructed when declared, and destructed anywhere the scope block is being left. Consider the following C++ code:

```
struct A;
void foo(int, A*);
int bar();
struct A {
  A(int i) {
    foo(i, this);
    return;
  }
  ~A() {
    foo(3, this);
    return;
  }
};
main() {
  while(true) {
    A a(1);
    foo(2, &a);
    if(bar()) {
      break;
    }
  }
  return 0;
}
```

The compilation of this example introduces the following design issues:

– The constructor and destructor for `A` should be compiled to ordinary static (not class-member) functions with a further `this` argument
– Compiling the declaration of `a` must call the constructor chosen following the types of the arguments given at the level of the declaration
– The constructor must be called at each entry of the loop body block, as each loop turn creates a new block scope
– The destructor must be called just before leaving the block scope of `a`: at the normal end of each loop turn, and when `break`ing the loop.

Those compilation design choices roughly yield the following compiled “pseudo-C” code:

```
struct A {};
void _constr_A(A* this, int i) {
  foo(i, this);
  return;
};
void _destr_A(A* this) {
  foo(3, this);
  return;
}
main () {
  while(true) {
    A a;
    _constr_A(&a, 1);
    foo(2, &a);
    if(bar()) {
      _destr_A(&a);
      break;
    }
    _destr_A(&a);
  }
  return 0;
}
```

11.1.1.2 Initializers

If a structure has a scalar field, then the constructor is meant to give the member its initial value. There are two ways to do this. The first naive version is to give values to fields directly in the body of the constructor, mimicking Java's initialization model:

```
struct A {
  int i;
  A(int i0) {
    this->i = i0;
    return;
  }
};
```

However, C++ has its own mechanism, which is to use an *initializer* for `i` at the level of the constructors of `A`:

```
struct A {
  int i;
  A(int i0): i(i0) {
    return;
  }
};
```

Conceptually, using an initializer for `i` shows that the notion of construction also applies for the fields of a structure. The presence of initializers in C++ guarantees that the corresponding field will be constructed, whereas a heavy program analysis is required in the naive case where fields are given their values in the body of the constructor.

But in practice, those two pieces of code have very similar behaviours, so they will be compiled to Ds++ in similar ways:

```
struct A {
  int i;
}
void _constr_A(A* this, int i0) {
  this->i0 = i;
  return;
}
```

11.1.1.3 Structure data members (aggregation)

Like C, C++ features embedded structures through data members, as in the following example:

```
struct A {};
struct B {
  A a;
};
```

In C++, a structure is required to take care of the construction and destruction of its embedded structure data members. This means that the constructor (resp. destructor) of `B` is responsible for calling the constructor (resp. destructor) of `A` for its data member `a`. To this purpose, the user should also provide, at the level of the constructors of `B`, an initializer for its structure field `a`, specifying which constructor of `A` to use and which arguments. (Nothing similar is needed for the destructor, as a class has exactly one destructor, with no arguments). For instance:

```
struct A {
  A(int) {}
  ~A() {}
};
struct B {
  A a;
  B(): a(18) {}
  ~B() {}
};
```

So, compiling the constructor (resp. destructor) of `B` makes it call the constructor (resp. destructor) for `A` with a `this` pointer argument adjusted to the data member `a`.

```
struct A {};
struct B {
  A a;
};
void _constr_A(A* this, int) {
  return;
}
void _destr_A(A* this) {
  return;
}
void _constr_B(B* this) {
  _constr_A(&(this->a), 18);
  return;
}
void _destr_B(B* this) {
  _destr_A(&(this->a));
  return;
}
```

11.1.1.4 Resource acquisition is initialization (RAII)

In C++, data members of a class must be fully available in the bodies of its constructors and destructor. This allows the following example, where a structure `A` opens a file, which is used by `B` embedding `a` in a field, in such a way that `B` reads from the file during its construction, and writes to the file during its destruction:

```
#define file_descr int
file_descr open();
char read(file_descr);
void write(file_descr, char);
void close(file_descr);
struct A {
  file_descr fd;
  A() : fd(open()) {
  }
  ~A() {
    close(fd);
  }
};
struct B {
  A a;
  B(): a() {
    read(a.fd);
  }
  ~B() {
    write(a.fd, 42);
  }
};
```

Then, this requires that the file be opened before the beginning of the body of `B()`, and not be closed before the end of the execution of the body of `~B()`. C++ solves this problem by enforcing destruction to behave *symmetrically* of construction:

- `A()` constructor must be called before the body of `B()` executes
- and symmetrically `~A()` destructor must be called after the body of `~B()` has executed.

This paradigm is called *resource acquisition is initialization (RAII)*. In terms of compilation, this would yield:

```
struct A {
  file_descr fd;
};
struct B {
  A a;
};
void _constr_A(A* this) {
  this->fd = open();
  return;
};
void _destr_A(A* this) {
  close(this->fd);
  return;
};
void _constr_B(B* this) {
  _constr_A(&(this->a));
  read(&(this->a)->fd);
  return;
};
void _destr_B(B* this) {
  write(&(this->a)->fd, 42);
  _destr_A(&(this->a));
  return;
}
```

Following the same principle:

- if a structure has several fields, then they are destructed in the reverse order of their construction
- if a structure has an array field, then its cells are destructed in the reverse order of their construction

11.1.1.5 Non-virtual inheritance

Besides structure aggregation, C++ also proposes *inheritance*. The RAII paradigm also applies to base class subobjects, so that in the above example `B` can be written using inheritance rather than aggregation. Then, the user should provide an initializer for the `A` base class subobject of `B`:

```
struct B: A {
  B(): A() {
    read(((A*) this)->fd);
  }
  ~B() {
    write(((A*) this)->fd, 42);
  }
};
```

Similarly to aggregation, the `A` base class subobject must be constructed before the execution of the constructor body of `B`, and destructed after the execution of the constructor body of

A. Similarly to the field access for constructing/destructing a structure field, calling the constructor/destructor for A needs to adjust the this pointer argument to access the A base class subobject:

```
struct B: A {};
void _constr_B(B* this) {
  _constr_A((A*) this);
  read(((A*) this)->fd);
  return;
};
void _destr_B(B* this) {
  write(((A*) this)->fd, 42);
  _destr_A((A*) this);
  return;
}
```

C++ proposes *multiple inheritance*: a class may inherit from several base classes. Then, following the RAII principe, they are to be destructed in the reverse order of their construction.

11.1.1.6 Virtual inheritance

Thanks to multiple inheritance, a same class X may be inherited from another class C through several paths. Ordinarily, then, each path corresponds to its own "copy" of X. But C++ also proposes *shared*, or *virtual*, inheritance, where a base class V may be declared virtual. In this case, all base classes declaring a virtual base class V will share their "copy" of V (independently of the other "copies" of V declared as non-virtual base class subobjects), as in the following hierarchy:

```
struct V {
  V() {}
  ~V() {}
};
struct B1 : virtual V {
  B1(): V() {}
  ~B1() {}
};
struct B2 : virtual V {
  B2(): V() {}
  ~B2() {}
};
struct D : B1, B2 {
  D(): B1(), B2() {}
  ~D() {}
};
```

Consider the construction of an instance of D.

Naively, if we followed a tree-based approach, construction of D would start by constructing B1, which in turn would first initiate the construction of V. Then, after the completion of B1, the construction of `B2` would start. Would it again initiate the construction of `V`, which is a *shared* subobject between `B1` and `B2` No: in C++ each subobject has to be constructed only once.

To enforce this, C++ distinguishes the notion of *most-derived object*: an object that is not a base class subobject of any other object. Then, each object only constructs and destructs its own non-virtual base class subobjects, except for a most-derived object, which, beforehand, must first construct (resp. last destruct) its direct or indirect virtual base class subobjects in such an order that, if V_1 and V_2 are (direct or indirect) virtual bases of some class B such that V_1 is a virtual base of B_2, then V_1 is constructed before (resp. destructed after) V_2. This guarantees that, during the construction (resp. destruction) of any object, all its base class subobjects, including its virtual base class subobjects, are already (resp. still) constructed.

So, when constructing the `B1` and `B2` subobjects, the called `B1()` and `B2()` constructors must not try to construct the virtual base subobject `V`, which has already been constructed by the most-derived object `D`: the initializer for `V` in `B1()` and `B2()` are ignored.

To this purpose, constructors and destructors need to be tailored accordingly to behave differently for a most-derived object than for a base class subobject. One approach can be to compile them with a further argument, a boolean indicating whether the object being constructed is a most-derived object requiring its virtual base subobjects to be constructed:

```
struct V {};
struct B1 : virtual V {};
struct B2 : virtual V {};
struct D : B1, B2 {};
void _constr_V(bool isMostDerived, V* this) {}
void _destr_V(bool isMostDerived, V* this) {}
void _constr_B1(bool isMostDerived, B1* this) {
  if(isMostDerived) {
    _constr_V(false, (V*) this);
  }
}
void _destr_B1(bool isMostDerived, B1* this) {
  if(isMostDerived) {
    _destr_V(false, (V*) this);
  }
}
void _constr_B2(bool isMostDerived, B2* this) {
  if(isMostDerived) {
    _constr_V(false, (V*) this);
  }
}
void _destr_B2(bool isMostDerived, B2* this) {
  if(isMostDerived) {
    _destr_V(false, (V*) this);
  }
}
void _constr_D(bool isMostDerived, D* this) {
  if(isMostDerived) {
    _constr_V(false, (V*) this);
  }
  _constr_B1(false, (B1*) this)
  _constr_B2(false, (B2*) this);
}
void _destr_D(bool isMostDerived, D* this) {
  _destr_B2(false, (B2*) this)
  _destr_B1(false, (B1*) this);
  if(isMostDerived) {
    _destr_V(false, (V*) this);
  }
}
```

Like non-virtual base class construction, the constructor or destructor for the virtual base class has to be called with a `this` pointer argument adjusted to the virtual base class subobject.

Alternately, a constructor or destructor may be duplicated into two versions, one as a most-derived object, another as a base-class subobject (for brevity, only constructors are shown below):

```
void _constr_V_mostDerived(V* this) {}
void _constr_V_subobject(V* this) {}
void _constr_B1_mostDerived(B1* this) {
  _constr_V(false, (V*) this);
}
void _constr_B1_subobject(B1* this) {}
void _constr_B2(B2* this) {
  _constr_V(false, (V*) this);
}
void _constr_D_most_derived(D* this) {
  _constr_V(false, (V*) this);
  _constr_B1(false, (B1*) this)
  _constr_B2(false, (B2*) this);
}
void _constr_D_subobject(D* this) {
  _constr_B1(false, (B1*) this)
  _constr_B2(false, (B2*) this);
}
```

The `_mostDerived` constructor could also call the `_subobject` counterpart (and vice-versa for the destructor), but we shall see further down why that “optimization” on code size is not performed.

11.1.1.7 Inheritance vs. aggregation

Inheritance and aggregation are distinct notions: the base class subobjects of a structure field must be constructed separately from the base class subobjects of its containing class. In particular, the constructor for a structure field must not forget its virtual base class subobjects: a structure field is a most-derived object, as it is not a base class subobject of any other object, in particular of the object containing the structure field:

```
struct V {
  V() {}
};
struct A : virtual V {
  A(): V() {}
};
struct B : virtual V {
  A a;
  B(): V(), a() {}
};
struct D : B {
  D(): V(), B() {}
};
```

In the above hierarchy, `V` is a virtual base class of `B` (and `D`), but there is another unrelated copy of `V` in the `A` data member of `B`, which is a most-derived object, so the constructor called for `a` must not forget this copy of `V`.

To harmonize, C++ mandates that the base class subobjects of an object, but including their data member subobjects, be constructed before (resp. destructed after) its data member subobjects. That is, in `B`, `a` (including all its base class subobjects, in particular its copy of `V`) is constructed after the `V` virtual base class subobject of `B`.

```
struct V {};
struct A : virtual V {};
struct B : virtual V {
  A a;
};
struct D : B {};
void _constr_V(bool isMostDerived, V* this) {}
void _constr_A(bool isMostDerived, A* this) {
  if(isMostDerived) {
    _constr_V(false, (V*) this);
  }
}
void _constr_B(bool isMostDerived, B* this) {
  if(isMostDerived) {
    _constr_V(false, (V*) this);
  }
  _constr_A(true, &(this->a)); /* a is most-derived regardless of B */
}
void _constr_D(bool isMostDerived, D* this) {
  if(isMostDerived) {
    _constr_V(false, (V*) this);
  }
  _constr_B(false, (B*) this);
}
```

We shall see further down another reason why own data member subobjects are constructed after base class subobjects.

11.1.2 Virtual functions during construction and destruction

11.1.2.1 Motivation

One of the main characteristics of inheritance is that a virtual function of a base class can be *overridden* in a derived class, even if the derived class is known only at run time.

Once entering the body of a constructor or destructor, the virtual functions of the class and its base classes are available, not only to direct calls from the constructor/destructor body, but also indirectly through other functions themselves directly or indirectly called from the constructor/destructor body. Consider the following example:

```
struct A {
  virtual void f();
};
void g(A* a) {
  a->f();
}
struct B: A {
  virtual void f();
  B(): A() {
    g((A*) this);
  }
};
struct D: B {
  virtual void f();
  D(): B() {
    g((A*) this);
  }
};
main () {
  D d;
}
```

Consider the instance `d` of `D`, C++ constructs the base class subobjects in the following order: `A`, then `B`, and finally `D`.

Now, during the construction of an instance of `D`, we focus on the constructor body for `B`. Contrary to Java, where the corresponding function of the most-derived class `D` would have been called, C++ avoids a potential use of uninitialized parts of `D`, by choosing the function declared in class `B` instead of `D`.

As the call occurs only indirectly, through another function `g` whose code may not be known to `B`, this particular choice cannot be resolved by a static program transformation, but only at run time. This corrects a bug in Wasserrab's Ph.D. thesis [84], where constructors only transform virtual function calls in the constructor body, where indeed some of them are known at compile-time to resolve to `B` instead of `D`, but erroneously ignoring indirect calls.

This example illustrates that the polymorphic behaviour of an object changes during its construction. Our κ++ language features *construction states* to model such behaviour changes, and from these construction states can be inferred the notion of *generalized dynamic type*, that is the subobject considered as the most-derived object for polymorphic operations (calling a virtual function or performing a dynamic cast).

In Ds++, we would like to explicitly determine the precise times when the generalized dynamic type has to change, by transposing the results of Section 10.5 (p. 218) without referring to construction states any longer. To this purpose, we define the "generalized dynamic type" in Ds++ as a property of any subobject that can be explicitly modified by a new operation, "set dynamic type", to indicate that the current being constructed should be considered as the most-derived object for polymorphic behaviour. (This is by contrast to κ++, where the "generalized dynamic type" was defined as a consequence of construction states.)

In practice, the "set dynamic type" operation must occur once all base class subobjects are constructed. So, the constructors for this class hierarchy would be compiled as:

```
struct A {
  virtual void f();
};
struct B: A {
  virtual void f();
};
struct D: B {
  virtual void f();
};
void g(A* a) {
  a->f();
}
void _constr_A(bool isMostDerived, A* this) {
  set dynamic type of this to A;
}
void _constr_B(bool isMostDerived, B* this) {
  _constr_A(false, (A*) this);
  set dynamic type of this to B;
  g((A*) this);
}
void _constr_D(bool isMostDerived, D* this) {
  _constr_B(false, (A*) this);
  set dynamic type of this to D;
  g((A*) this);
}
```

Aggregation and polymorphic behaviour The main difference between inheritance and aggregation is that the polymorphic behaviour of a class is related to the polymorphic behaviour of its base class subobjects, but not related to the polymorphic behaviour of its data members, thus pointing out the fact that the two notions of aggregation and inheritance are semantically distinct, commonly dubbed as "has-a vs. is-a".

To illustrate this independence, consider the following example:

```
struct X {
  X(int) {}
};
struct A {
  virtual int f();
};
int g(A* a) {
  return a->f();
}
struct B: A {
  X x;
  virtual int f();
  B(): x(g((A*) this)) {}
};
struct D: B {
  virtual int f();
  D(): B() {}
};
```

The virtual functions of a class are available for use in the arguments of its data member initializers, as well as during the construction and destruction of the latter. This means that the "set dynamic type" operation has to be performed before constructing the data member subobjects.

This justifies that data members have to be constructed after (resp. destructed before) the non-virtual and virtual base class subobjects of an object.

So, the above example yields:

```
struct X {};
struct A {
  virtual int f();
};
int g(A* a) {
  return a->f();
}
struct B: A {
  X x;
  virtual int f();
};
struct D: B {
  virtual int f();
};
void _constr_X(bool isMostDerived, X* this, int) {
  set dynamic type of this to X;
}
void _constr_A(bool isMostDerived, A* this) {
  set dynamic type of this to A;
}
```

```
void _constr_B(bool isMostDerived, B* this) {
  _constr_A(false, (A*) this);
  set dynamic type of this to B;
  _constr_X(true, &(this->x), g((A*) this));
}
void _constr_D(bool isMostDerived, D* this) {
  _constr_B(false, (B*) this);
  set dynamic type of this to D;
}
```

To be more precise, the "set dynamic type" operation must act on an object and all its base class subobjects (to reflect the right polymorphic behaviour on `A`), but not their data member subobjects (in particular the polymorphic behaviour of `x`). We shall see in Section 11.1.2.2 (p. 238) the inner workings of the "set dynamic type" operation.

Summary To sum up, a κ++ constructor of a class having direct or indirect virtual or non-virtual bases, or structure array or scalar fields, should be compiled into a Ds++ static function as roughly follows (we formalize this compilation step more precisely in Section 11.5.9 p. 264):

```
void _constr_C(bool isMostDerived, C* this, ...) {
  if(isMostDerived) {
    for each V direct or indirect virtual base of C {
      execute the initializer for V, ending with
      _constr_V(false, (V*) this, ... );
    }
  }
  for each B direct non-virtual base of C {
    execute the initializer for B, ending with
    _constr_B(false, (B*) this, ... );
  }
  set dynamic type to C;
  for each m data member of C {
    if m is a scalar {
      execute the initializer for m, ending with
      this->m = value;
    } else, m is a structure A[n] {
      for(i = 0, i < n, ++i) {
        execute the initializer for m[i], ending with
        _constr_A(true, &(this->m[i]), ...);
      }
    }
  };
  execute the constructor body;
  return;
}
```

Destructors behave symmetrically, with the simplification that a destructor has no user-defined arguments, so that no counterpart to "initializers" exist. The body of the destructor is executed first, before destructing the fields in reverse declaration order, then the direct non-virtual bases, and finally, if the destructed object is a most-derived object, the direct or indirect virtual bases.

The destruction of a base-class subobject needs to call the corresponding class destructor, as well as the destruction of each cell of a structure array data member, with the difference that structure array cells are most-derived objects, so their virtual base subobjects also have to be destructed.

By contrast, nothing has to be done for a scalar field. (In κ++, their value were erased, but we know that no κ++ program would have a defined behaviour when accessing a destructed scalar field. As far as our compiler is concerned, we wish to prove the preservation of the semantics of programs that actually have a well-defined behaviour, thus following the principle of "garbage in, garbage out".)

A virtual function can be called in the body of the destructor (or indirectly during the destruction of a data member, which was not explicitly allowed in the C++ Standard until we submitted a modification request integrated into the latest C++11, cf. Section 9.5.1 p. 196), the destructed object being considered as the most-derived object. Thus, the destructor call has to first set the dynamic type. However, such an operation is useless if the destructor is called for a most-derived object: in such a case, the dynamic type would not change through this operation.

Finally, the compilation scheme for a destructor is as roughly follows (we formalize this compilation step more precisely in Section 11.5.10 p. 268):

```
void _destr_C(bool isMostDerived, C* this) {
  if(!isMostDerived) {
    set dynamic type to C;
  }
  execute the destructor body;
  for each f data member of C in reverse order {
    if f is a scalar {
    } else, f is a structure A[n] {
      for(i = n-1, i >= 0, --i) {
        _destr_A(true, &(this->f[i]));
      }
    }
  };
  for each B direct non-virtual base of C in reverse order {
    _destr_B(false, (B*) this);
  }
  if(isMostDerived) {
    for each V direct or indirect virtual base of C in reverse order {
      execute the initializer for V, ending with
      _destr_V(false, (V*) this);
    }
  }
  return;
}
```

11.1.2.2 The “set dynamic type” operation

In κ++, each base class subobject $(\ell, (\alpha, i, \sigma)))$ of a most-derived object $(\ell, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon)))$ has a “generalized dynamic type” whose value is computed depending on the construction states of the most-derived object $(\ell, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon)))$ and all of its base class subobjects:

- if the most-derived object $(\ell, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon)))$ is $\mathsf{Constructed}$, then the generalized dynamic type of $(\ell, (\alpha, i, \sigma))$ is $((\mathsf{Repeated}, C :: \epsilon), \sigma)$
- otherwise, if $(\ell, (\alpha, i, \sigma))$ is a base class subobject of some $(\ell, (\alpha, i, \sigma_\circ))$ such that $(\ell, (\alpha, i, \sigma_\circ))$ is $\mathsf{BasesConstructed}$, or $\mathsf{StartedDestructing}$, then the generalized dynamic type of $(\ell, (\alpha, i, \sigma))$ is $(\sigma_\circ, \sigma')$ with σ' such that $\sigma = \sigma_\circ @ \sigma'$.
- otherwise, the generalized dynamic type of $(\ell, (\alpha, i, \sigma))$ is undefined.

In Ds++, subobjects no longer have construction states, so the notion of generalized dynamic type has to be redefined manually. To this purpose, we introduce a new operation, $\mathsf{setDynType}(\pi)$, to explicitly modify the generalized dynamic type of a subobject π *and all of its base class subobjects* at the same time. That is, considering the following hierarchy:

```
struct X {
  virtual void u();
};
void t(X* x) {
  x->u();
}
struct Y : X {
  virtual void u();
  Y(): X() {
    t((X*) this);
  }
};
struct Z : Y {
  virtual void u();
};
```

the constructor for `Y` is roughly turned into:

```
void _constr_Y(bool isMostDerived, Y* this, ...) {
  _constr_X((X*) this);
  set dynamic type to Y;
  t((X*) this);
  return;
}
```

so that, when `t` is called with the `X` subobject of `Y`, then `u` of `Y` is called: that is, the “set dynamic type” operation turns the generalized dynamic type of not only the `Y`, but also the `X` subobject, to the `Y` subobject.

Real-world C++ compilers such as GNU GCC implement the polymorphic behaviour of C++ classes with multiple inheritance using *virtual tables*: each object of a class having at least one virtual base, or one virtual function defined in itself or in one of its direct or indirect bases, has a pointer to a read-only memory zone, called a *virtual table* containing pointers to virtual functions and offsets to virtual bases, so that virtual function dispatch and access to a virtual base be realized in constant time and memory access.

So, in practice, when compiling Ds++ into a more low-level CVcm language, the “set dynamic type” operation corresponds to the time when the pointers to virtual tables of subobjects are to be changed. We formalize this compilation step in Section 11.8.6 (p. 297).

Naively, the “set dynamic type” operation on a subobject of some static type A could write pointers to the virtual tables of A considered as a most-derived object. However, considering the following hierarchy with virtual inheritance:

```
struct V {
  int vi;
};
struct B: virtual V {};
struct D: B {
  int di;
};
```

Concretely, a `B` subobject of `D` sees its `V` subobject at an offset different than if `B` were a true most-derived object, because of the further data member of `D`. As accessing `V` from a `B` subobject requires reading the virtual table, this shows that different virtual tables have to be used. But, consequently, this also requires to know the path from the most-derived object to the subobject being constructed (or destructed). In real world, this cannot be done by a computation on the pointer to the subobject, so a further argument is required to indicate this path, which we call the *construction path*.

In practice, this further argument corresponds to a pointer to the *virtual table table* (VTT) of $\sigma_\circ$, that is a table where to find the pointers to virtual tables for base class subobjects of $(\ell, (\alpha, i, \sigma_\circ))$ when setting the generalized dynamic type of $(\ell, (\alpha, i, \sigma_\circ))$. Those virtual tables are called *construction virtual tables* (although also used similarly during destruction). Such implementations are common in several compilers, and standardized for the Itanium platform by the Common Vendor ABI.

So, the “set dynamic type” operation has two arguments: the pointer to the subobject for which the generalized dynamic type is to be modified, as well as the dynamic type of all of its bases, and the corresponding construction path. The operation has a defined semantics only if the pointer to the subobject actually corresponds to the construction path: that is, when setting the generalized dynamic type of $(\ell, (\alpha, i, \sigma_\circ))$ with the construction path $\sigma_\circ$.

This in turn entails the need for further operations in Ds++ to use those construction paths:

- retrieve the path to a most-derived object of a class C
- given a construction path σ, retrieve the construction path of a direct non-virtual base, or of a direct or indirect virtual base (only for most-derived objects).

We also add a third operation on construction paths to determine whether a most-derived is being constructed, thus replacing the boolean argument of the constructor with the construction path:

```
void _constr_C(void* _cpath, C* this, ...) {
  if(_cpath represents a most-derived C object) {
    for each V direct or indirect virtual base of C {
      execute the initializer for V, ending with
      _cpath' = construction path from _cpath to the virtual base V;
      _constr_V(_cpath', (V*) this, ...);
    }
  }
  for each B direct non-virtual base of C {
    execute the initializer for B, ending with
    _cpath' = construction path from _cpath to the direct non-virtual base B;
    _constr_B(_cpath', (B*) this, ...);
  }
  set dynamic type to C;
  for each m data member of C {
    if m is a scalar {
      execute the initializer for m, ending with
      this->m = value;
    } else, m is a structure A[n] {
      for(i = 0, i < n, ++i) {
        execute the initializer for m[i], ending with
        _cpath' = construction path of a most-derived A;
        _constr_A(_cpath', &(this->m[i]), ...);
      }
    }
  };
  execute the constructor body;
  return;
}
```

Then, it is the responsibility of the caller to pass the right construction path to the constructor: it shall be the path representing a most-derived object, except in initializers for base-class subobjects as seen above.

In more detail, if δ is a partial function retrieving the generalized dynamic type $\delta(\ell, (\alpha, i, \sigma)) = (\sigma_\circ, \sigma')$ of a subobject $(\ell, (\alpha, i, \sigma))$ in Ds++, then the "set dynamic type" operation should update δ. We write it as $\delta[\mathsf{setDynType}(\ell, (\alpha, i, \sigma_\circ))]$.

One of the invariants to prove within the κ++-to-Ds++ compiler is that, if a subobject $(\ell, (\alpha, i, \sigma))$ has a well-defined generalized dynamic type $\sigma_\circ$ in κ++, then it has the same generalized dynamic type in Ds++: if $\mathsf{gDynType}(\ell, \alpha, i, \sigma, B, \sigma_\circ, \sigma')$, then $\delta(\sigma) = (\sigma_\circ, \sigma')$.

Moreover, one of the invariants to prove within the Ds++-to-CVcm compiler is that, if $\delta(\sigma) = (\sigma_\circ, \sigma')$ in Ds++, then $\sigma = \sigma_\circ @ \sigma'$ and the corresponding CVcm concrete object holds a pointer to the virtual table of σ' considering $\sigma_\circ$ as the "most-derived object" for the purpose of polymorphic behaviour.

However, we saw in Section 5.5.5 (p. 117) that an object may share its pointer to virtual table with one of its non-virtual bases, called the *primary base*. The actual choice of this non-virtual primary base is left to the layout algorithm. Thus, the Ds++-to-CVcm invariant has to be rephrased accordingly. In fact, we also saw that for any inheritance path σ, there exists a unique inheritance path $\mathsf{reducePath}(\sigma)$ and a path σ'' such that $\sigma = \mathsf{reducePath}(\sigma) @ \sigma''$ with σ'' being a *primary path* of maximal length. So, if a subobject is a primary base class subobject of another, then they have the same reduced path. But a class can share its virtual table with its primary bases. So, the Ds++-to-CVcm invariant is: if $\delta(\sigma) = (\sigma_\circ, \sigma')$ in Ds++, then $\sigma = \sigma_\circ @ \sigma'$ and the corresponding CVcm concrete object holds a pointer to the virtual table of $\mathsf{reducePath}(\sigma')$ considering $\sigma_\circ$ as the "most-derived object" for the purpose of polymorphic behaviour.

Our question is now: which semantics should be given to the "set dynamic type" operator that would make both invariants hold?

A naive attempt: leave others alone One solution could be to say that only the generalized dynamic types of $(\ell, (\alpha, i, \sigma_\circ))$ and all of its base class subobjects would change:

$$\begin{array}{rccl} \delta[\mathsf{setDynType}(\ell, (\alpha, i, \sigma_\circ))] & : & (\ell, (\alpha, i, \sigma_\circ @ \sigma')) & \mapsto (\sigma_\circ, \sigma') \\ & & \begin{array}{c}(\ell, (\alpha, i, \sigma)) \\ \sigma \text{ not a base of } \sigma_\circ\end{array} & \mapsto \delta(\ell, (\alpha, i, \sigma)) \\ & & \begin{array}{c}(\ell'', (\alpha'', i'', \sigma'')) \\ (\ell'', \alpha'', i'') \neq (\ell, \alpha, i)\end{array} & \mapsto \delta(\ell'', (\alpha'', i'', \sigma'')) \end{array}$$

However, this would break the Ds++-to-CVcm invariant. Consider the following simple example:

```
struct B {
  virtual void f();
};
struct D : B {
  virtual void f();
};
```

When destructing an instance of D, we first call the destructor of D, then the destructor of B, which performs a setDynType operation on the B subobject. In CVcm, this operation would be compiled into writing a pointer to a B virtual table into the B subobject. But, as we saw in Section 5.3.1 (p. 98) and Section 5.5.5 (p. 117) a layout algorithm may (and actually does, in the case of GNU GCC, or the Common Vendor ABI for Itanium) choose B as the *primary base* of D, thus sharing B's and D's pointer to virtual table. In Ds++, the generalized dynamic type of D remains defined to $((\mathsf{Repeated}, D :: \epsilon), (\mathsf{Repeated}, D :: \epsilon))$ through the setDynType operation on the B subobject, but the generalized dynamic type of B becomes $((\mathsf{Repeated}, D :: B :: \epsilon), (\mathsf{Repeated}, B :: \epsilon))$. So, the pointer to virtual table of D contains information valid only for B but not for D.

This illustrates the discrepancy of the generalized dynamic types of a class and its primary base.

A further naive attempt: erase everyone To dissipate the problem, we could argue that, in κ++, when setting the generalized dynamic type of an object $(\ell, (\alpha, i, \sigma_\circ))$, all base class subobjects of the same most-derived object but that are not base class subobjects of $\sigma_\circ$ have their generalized dynamic type undefined. We could mimic this behaviour into Ds++:

$$\begin{array}{rccl} \delta[\mathsf{setDynType}(\ell, (\alpha, i, \sigma_\circ))] & : & (\ell, (\alpha, i, \sigma_\circ @ \sigma')) & \mapsto (\sigma_\circ, \sigma') \\ & & \begin{array}{c}(\ell, (\alpha, i, \sigma)) \\ \sigma \text{ not a base of } \sigma_\circ\end{array} & \mapsto \text{Undefined} \\ & & \begin{array}{c}(\ell'', (\alpha'', i'', \sigma'')) \\ (\ell'', \alpha'', i'') \neq (\ell, \alpha, i)\end{array} & \mapsto \delta(\ell'', (\alpha'', i'', \sigma'')) \end{array}$$

Then, in CVcm, no information on pointers to virtual tables is available for objects whose dynamic type is undefined in κ++. However, pointers to virtual tables are used not only for polymorphic operations (virtual function calls, dynamic casts), but also to access the virtual base class subobjects of an object. And this must be possible when all base class subobjects have been constructed, but also during the lifetime of the subobject, even once leaving the constructor body. Consider the following example:

```
struct B1;
B1* b1;
struct V {
  int i;
  V() : i(18) {}
};
struct B1 : virtual V {
  B1() : V() {
    b1 = this;
  }
};
struct B2 {
  B2 () {
    ((V*) b1)->i++;
  };
};
struct D : B1, B2 {};
```

Consider an instance of D. Then, we construct in order: V, B_1, B_2, then D. But, once leaving the constructor body of B_1, the lifetime of B_1 subobject begins, so the lifetime of B_1 starts, and this before entering the constructor body of B_2 which triggers a “set dynamic type” operation. If generalized dynamic type information for B_1 (which is not a base of B_2) were erased in Ds++, then we would have no further information on the pointer to the virtual table held by B_1, so the access to V from B_1 inside the constructor of B_2, which should be valid as B_1 has entered its lifetime, cannot be proved feasible in CVcm.

This problem also illustrates the need for a condition in Ds++ to allow accessing the virtual base subobjects of an object. In κ++, this can be explained thanks to the construction states of the object, which must be BasesConstructed, Constructed, or StartedDestructing. But Ds++ no longer has construction states for objects, and in CVcm we know that accessing a virtual base needs the pointer to virtual table even though the generalized dynamic type is no longer defined in κ++. It then turns out that the generalized dynamic type must still be defined in Ds++.

Our solution When setting the dynamic type of some subobject $(\ell, (\alpha, i, \sigma_\circ))$, we know that in CVcm we overwrite the pointers to virtual tables of those objects $(\ell, (\alpha, i, \sigma))$ such that $\sigma_\circ$ is a primary non-virtual base subobject of σ. Then, we could mimic this behaviour into Ds++. However, it would not be clean to have layout-dependent elements in Ds++. But in fact, Ds++ can be *parameterized* by a notion of primary non-virtual path $(\mathsf{Repeated}, l)$, say $\Pi(l) \in \{\mathsf{true}, \mathsf{false}\}$; no specific property about Π is required in Ds++ except its decidability. Using this parameter, we can now give our solution to model the “set dynamic type” operation. We adopt this solution in our Ds++ language, more precisely in Section 11.4.10 (p. 252).

$$
\begin{array}{lccc}
\delta[\mathsf{setDynType}(\ell,(\alpha,i,\sigma_\circ))] & : & (\ell,(\alpha,i,\sigma_\circ@\sigma')) & \mapsto (\sigma_\circ,\sigma') \\
 & & \begin{array}{c}(\ell,(\alpha,i,\sigma)) \\ \sigma \text{ not a base of } \sigma_\circ \\ \sigma_\circ = \sigma@(\mathsf{Repeated}, l) \text{ with } \Pi(l) = \mathsf{true}\end{array} & \mapsto \text{Undefined} \\
 & & \begin{array}{c}(\ell,(\alpha,i,\sigma)) \\ \sigma \text{ not a base of } \sigma_\circ \\ \sigma_\circ \text{ not a primary base of } \sigma\end{array} & \mapsto \delta(\ell,(\alpha,i,\sigma)) \\
 & & \begin{array}{c}(\ell'',(\alpha'',i'',\sigma'')) \\ (\ell'',\alpha'',i'') \neq (\ell,\alpha,i)\end{array} & \mapsto \delta(\ell'',(\alpha'',i'',\sigma''))
\end{array}
$$

Then, in Ds++, accessing a virtual base of $(\ell,(\alpha,i,\sigma))$ would be allowed if and only if its generalized dynamic type is defined, regardless of its value. This leads to the further κ++-to-Ds++ invariant: any Constructed subobject in κ++ has its generalized dynamic type defined in Ds++.

Recall that part of the Ds++-to-CVcm invariant is to show that if $\delta(\ell,(\alpha,i,\sigma)) = (\sigma_\circ,\sigma')$, then $\sigma = \sigma_\circ@\sigma'$. Thanks to this invariant, we can show that the access to virtual bases only depends on $(\sigma_\circ@\sigma')$, that is on σ but not individually on $\sigma_\circ$ and σ'. This illustrates the behaviour of some early Microsoft Visual C++ compilers, which stored for each object *two* pointers: the one being a pointer to the table of virtual base offsets, while the other being a pointer to the table of virtual functions and dynamic casts, modeling true C++ polymorphic behaviour. But such layout has finally been abandoned to reduce the size of objects (the sizes of virtual tables being deemed neglected before the sizes of all objects potentially constructed during program lifetime).

11.1.3 Optimizations

Compile-time vs. run-time offset resolution When constructing an object, adjustments of "this" have to be performed to construct the subobjects of this offset. The offsets of direct non-virtual subobjects, and data members, are known at compile time and do not change whether the object being constructed is a most-derived object or an inheritance subobject. But this is not the case for virtual bases, only known for most-derived objects. However, virtual base subobjects are being constructed only within the constructor for a most-derived object. So, we have to provide, in Ds++, an operation allowing to access the virtual base subobjects of a most-derived object without referring to its generalized dynamic type (which is undefined at this time). Then, those adjustments can be compiled into CVcm through constant offset shifts.

But the question also arises when compiling the $\mathsf{setDynType}(\ell,(\alpha,i,\sigma))$ operation from Ds++ to CVcm: for each inheritance subobject σ' of σ, update the pointer to its virtual table. Such a traversal of paths could require reading virtual tables if σ has virtual base subobjects. But again, those accesses to virtual tables can be avoided if we know that σ is a most-derived object: then, the offsets to all of its base class subobjects are known at compile time. So, we choose to parameterize the setDynType operation in Ds++ with a *compile-time* boolean flag, true only if a most-derived object, or an object with no virtual base subobjects, is constructed.

Those two elements lead us to finally choose to *duplicate* constructors and destructors, instead of adding a run-time test to discriminate between a most-derived object and a subobject.

Restriction to dynamic classes All classes do not necessarily require a pointer to virtual table: only those having virtual functions, or virtual bases. Such classes are called *dynamic*, or *polymorphic* classes. So, while it is not necessary to introduce this distinction as early as in the semantics of Ds++, however, an optimization can occur in the Ds++-to-CVcm compilation, by stating Ds++-to-CVcm invariants only for subobject of dynamic class type, so that the operations on construction paths, as well as setDynType, can be compiled into no-ops for non-dynamic classes.

It is worth noting that this notion is part of the Standard, in such a way that dynamic casts from an object of non-dynamic class type are not allowed.

11.2 Syntax of the Ds++ intermediate language

The Ds++ language[1] is a superset of s++ (Chapter 4 p. 71) featuring static and non-virtual function calls and C++ object-oriented operations such as data member and array accesses, static and dynamic casts, and virtual function calls.

Ds++ differs from κ++ by having no constructors or destructors. Instead, Ds++ is based on s++ to which it adds construction-specific features: the "set dynamic type" operation, and construction path operations (most-derived path, direct non-virtual base path, or virtual base path).

n	$\in \mathbb{N}$	
$op, \ldots$	$: Op$	Built-in operation
$this, x, \ldots$	$: x$	Variables
$B, C, \ldots$	$: ClassName$	Classes
$fname$	$: FieldName$	Field names
$mname$	$: MethodName$	Method names
$sfname$	$: StaticFunName$	Static function names
$BaseKind$	$::= \mathsf{DirectNonVirtual} \mid \mathsf{Virtual}$	Base kind for construction paths

$st ::= \mathtt{if}\ (x)\ st_\top\ \mathtt{else}\ st_\bot$	Conditional
$\mid st_1; st_2$	Statement sequence
$\mid \mathtt{skip}$	Do nothing
$\mid \mathtt{loop}\ st$	Loop
$\mid \{st\}$	Statement block
$\mid \mathtt{exit}\ n$	Leaving n blocks
$\mid x' := x$	Variable value duplication
$\mid x' := op(x^*)$	Built-in operation
$\mid x' := x\texttt{->}C\texttt{::}mname(x^*)$	Non-virtual function call
$\mid x' := sfname(x^*)$	Static function call
$\mid \mathtt{return}\ x^?$	Return from function
$\mid x' := x\texttt{->}_C fname$	Field read
$\mid x\texttt{->}_C fname := x'$	Scalar field write
$\mid x' := \&x[x_{\mathsf{index}}]_C$	Array cell access
$\mid x' := x_1 ==_C x_2$	Pointer equality test
$\mid x' := \mathtt{dynamic\_cast}\langle B\rangle_C(x)$	Dynamic cast
$\mid x' := x\texttt{->}_C mname(x^*)$	Virtual function call
$\mid x' := \mathtt{static\_cast}\langle B\rangle_C(x)$	Static cast
$\mid \{C\ x[size]; st\}$	Complete object allocation
$\mid x' := \mathtt{base\_cast}\langle BaseKind, B\rangle_C(x)$	Special cast to base
$\mid x' := \mathtt{setDynType}(x, x_{\mathsf{cpath}})_C^{Bool}$	Set dynamic type
$\mid x' := \mathtt{rootPath}(C)$	Most-derived construction path
$\mid x' := \mathtt{basePath}(x, C, BaseKind, B)$	Base construction path

1. Coq development: theory `Interm`.

StaticFunDef	::=	$(x^*)\{body\}$	Static function
MethodDef	::=	*this*->$(x^*)\{st\}$	Class member function (method) definition

ProgramCode =
{

hierarchy	:	*Hierarchy*	; Class hierarchy
staticfuns	:	$\mathit{StaticFunName} \nrightarrow \mathit{StaticFunDef}$	; Static functions
methods	:	$\mathit{Class} \times \mathit{MethodName} \nrightarrow \mathit{MethodDef}$	; Class method codes
main	:	*st*	; Entry point

}

Like κ++, our Ds++ language only features stack objects declared in scope blocks. It allows no manual memory management: no kind of `new` (neither for dynamic memory allocation, nor for construction at an explicit memory location) or `delete` is provided.

11.3 Ds++ semantic elements

We formalized a small-step style semantics for the Ds++ language. Contrary to κ++, but similarly to s++ and many Compcert-like intermediate languages, the execution stack is provided with a simple "block or callframe" continuation stack to precisely model each step of computation.

11.3.1 Values

A Ds++ value is either a value of built-in type (integer, floating-point number, etc.), a pointer to a subobject, a null pointer, as in s++, or also a construction paths (pairing the class of the most-derived object with an inheritance subobject from this class).

$\ell, \ldots$	$\in \Lambda$	Complete object location
Ptr	$::= (\ell, (\alpha, i, \sigma))$	Pointer to subobject
Val	$::=$ *Builtin*	Value of built-in type
	$\mid$ *Ptr*	Non-null pointer
	$\mid \mathsf{NULL}_C$	Null pointer of C class type
	$\mid (C, \sigma)$	Construction path (inheritance path from C)

11.3.2 Execution state

Similarly to s++, a Ds++ *execution state* of the small-step semantics is composed of the current statement to execute, the list of further statements to execute in the same block, the environment (mapping of values to variables), the class types and array sizes of complete objects, and the values of scalar fields. Additionally, a Ds++ execution state also features:

– a continuation stack, which is a list of frames, each frame being either of:
  – leaving a block, with the stack object location to deallocate and the further statements to execute after leaving the block

  - returning from a function, with the caller variable to store the result (if any), the caller environment, and the further statements to execute on resumption
- the location of the next object to be allocated
- the *generalized dynamic types* of subobjects (which are the last remnants of the κ++ construction states)

For presentation convenience, the types and sizes of complete objects, the scalar field values, and the generalized dynamic types of objects are grouped into a common *global state*, so that a state is written as a tuple $(st, st^*, e, \mathcal{K}, \mathcal{G})$ where st is the current statement, st^* is the list of further statements, $\mathcal{K}$ the continuation stack, and $\mathcal{G}$ the global state grouping the store, the scalar field values, and the generalized dynamic types of subobjects.

Env	$=$	$x \to Val^?$	Environment
e	$::=$	Env	
$Frame$	$::=$	$\mathsf{Block}(\ell^?, st^*)$	Further statements after leaving a block
	$\mid$	$\mathsf{Callframe}(x^?, st^*, e)$	Return from function
$\mathcal{K}$	$::=$	$Frame^*$	Continuation stack
$Path$	$::=$	σ	
$\mathcal{G}$	$=$		
$\{$			
$\mathsf{LocType}$	$:$	$\Lambda \to (ClassName \times \mathbb{N}^{>0})^?$;	Complete object types
$\mathsf{FieldValue}$	$:$	$Ptr \times FieldSig \to Val^?$;	Scalar field values
$\mathsf{gDynType}$	$:$	$Ptr \to (Path \times Path)^?$;	Generalized dynamic type of objects
$\}$			
$State$	$::=$	$(st, st^*, e, \mathcal{K}, \mathcal{G})$	Execution state

Contrary to κ++, a complete object location is given its class type and array size only within its defining block: once the block exits, those data are removed from the store. However, the store is guaranteed to never reuse the same location for two objects allocated at different times, thanks to the following hypotheses to determine the location of the next object to be allocated:

Hypothesis 11.3.1. *The set ℓ of object locations is assumed to be equipped with*[2]*:*
- *an element $\ell_\circ$ which will be the location of the first object allocated by the program*
- *a strict order $<$*
- *a function* $\mathsf{next} : \Lambda \to \Lambda$ *such that* $\ell < \mathsf{next}(\ell)$ *for any location* ℓ

11.3.3 Initial and final states

Consider a program of the form

$$ClassDecl^* \, StaticFunDef^* \, MethodDef^* \{st\}$$

Then, the initial state is the state featuring the entry point statement with no allocated object at all, and an empty continuation stack:

2. In our Coq development, this is the case as $\ell = $ `positive`. Moreover, those hypotheses are analogous to Hypothesis 10.3.1 (p. 216) otherwise used for reasoning about the construction states of different subobjects.

***Definition* 11.3.1.** *The initial state is:*

$$(st, \epsilon, \varnothing, \epsilon, \mathcal{G}_\circ)$$

where $\mathcal{G}_\circ$ is the initial global state:

$$\forall \ell : \mathcal{G}_\circ.\mathsf{LocType}(\ell) = \bot \qquad \forall \pi, f : \mathcal{G}_\circ.\mathsf{FieldValue}(\pi, f) = \bot \qquad \forall \pi : \mathcal{G}_\circ.\mathsf{gDynType}(\pi) = \bot$$

$$\mathcal{G}_\circ.\ell_{\mathsf{next}} = \ell_\circ$$

In a final state, the entry point statement returns with an integer, after having exited from all blocks and functions:

***Definition* 11.3.2.** *A state $(st, st^*, e, \mathcal{K}, \mathcal{G})$ is* final with return value *i if, and only if, all the following conditions hold:*

$$st = \texttt{return}\ x \qquad e(x) = i \in \mathit{Builtin} \qquad \mathcal{K} = \epsilon$$

Note that this definition does not a priori prevent from having some undestructed objects in the global state $\mathcal{G}$ of a final state. However, as we shall see later, the semantics preservation of the compiler from κ++ to Ds++ shall prove that it is not possible to produce such a program by compiling a κ++ program.

11.4 Ds++ Semantic rules

The small-step semantics of Ds++ is given by the transition relation $\rightarrow$ between two transition states, defined in this section.

11.4.1 Built-in operations and structured control

In an execution state $(st, stl, e, \mathcal{K}, \mathcal{G})$, st is the statement to run, and L is a pipeline of pending statements *within the same block*, each pending enclosing block being represented by a frame in the continuation stack $\mathcal{K}$. However, the pipeline stl is not guaranteed to be executed, in particular if the statement is `exit` or `return`.

Most structured control behaves similarly as in other Compcert-like languages: conditionals, sequences, infinite loops, and return from call (once all statements blocks within the current function have been left), as well as variable value duplication, and built-in operations (Hypothesis 3.2.2 p. 68). Ds++ reuses the corresponding rules of s++ defined in Section 4.4.1.1 (p. 83)

11.4.2 Blocks with no stack objects

In this section, we first define the semantics of the blocks that do not define objects. Entering such a block embeds the pipeline into a new enclosing block without attached object, as a new stack frame added on top of the continuation stack:

$$\frac{}{\begin{array}{rllll} & (\{st\}, & stl, & e, & \mathcal{K}, \quad \mathcal{G}) \\ \rightarrow & (st\ \ \ , & \epsilon, & e, & \mathsf{Block}(\bot, stl) :: \mathcal{K}, \quad \mathcal{G}) \end{array}} \quad \text{(Ds++-block-no-obj)}$$

`exit` n leaves n blocks.

$$\frac{}{\begin{array}{rlllll} & (\texttt{exit}\ 0, & \mathit{stl}, & e, & \mathcal{K}, & \mathcal{G}) \\ \rightarrow & (\texttt{skip}\ \ , & \mathit{stl}, & e, & \mathcal{K}, & \mathcal{G}) \end{array}} \quad \text{(Ds++-exit-0)}$$

$$\frac{}{\begin{array}{rlllll} & (\texttt{exit}\ (\mathsf{S}\ n), & \mathit{stl}, & e, & \mathsf{Block}(\bot, \mathit{stl}') :: K, & \mathcal{G}) \\ \rightarrow & (\texttt{exit}\ n \quad\ , & \mathit{stl}', & e, & \mathcal{K}, & \mathcal{G}) \end{array}} \quad \text{(Ds++-exit-S)}$$

`return`ing from within a block with no stack objects first dismisses this block.

$$\frac{}{\begin{array}{rlllll} & (\texttt{return}\ x^?, & \mathit{stl}, & e, & (\bot, \mathit{stl}') :: \mathcal{K}, & \mathcal{G}) \\ \rightarrow & (\texttt{return}\ x^?, & \mathit{stl}', & e, & \mathcal{K}, & \mathcal{G}) \end{array}} \quad \text{(Ds++-return-block-no-obj)}$$

11.4.3 Static and non-virtual functions

Ds++ allows to call static (non-class-member) functions, as well as class member functions in a non-virtual fashion (as in C++ with explicit qualification), except that the class where to explicitly find the function must be exactly the static type of the object on which to perform the call: there is no implicit cast. The rules are exactly the same as for s++ (Section 4.4.2 p. 85).

11.4.4 Field and array accesses; pointer comparison

The semantic rules of Ds++ for data member and array accesses are unchanged since s++ (Section 4.4.3 p. 86).

They differ from κ++ by the fact that there are no condition on object construction states (as such a notion no longer appears in Ds++). Moreover, contrary to κ++ where an object location has to be checked not to be in the set of deallocated objects, it is enough to have in Ds++ that the object location is associated with a class and a number of cells in the heap (as object deallocation actually removes heap data in Ds++, by contrast to κ++).

To access an array cell, we need to express array cell indices:

Hypothesis 11.4.1. *The set of values of built-in types Builtin is assumed to contain:*
– *$\mathbb{Z}$ to model array cell indexes*[3]
– *the Boolean values* true *and* false

By contrast to κ++, Ds++ poses no condition about when to forbid writing data to a scalar field, as Ds++ no longer features construction states. Likewise, contrary to κ++, Ds++ does not require fields to be unassigned.

3. Requiring built-in values to include the whole $\mathbb{Z}$ may not seem realistic; however, we could argue that the problem would arise only after CVcm, as all object sizes are known statically at compile time (there are only stack-allocated objects), so that a static check on those requested sizes could tell whether memory accesses fit in realistic bounds (offsets less than 2^{31} for instance).

11.4.5 Blocks with stack objects

Ds++ can allocate and deallocate stack objects attached with a block, but has no notion of construction or destruction (as constructors and destructors are to be compiled into ordinary static functions).

The following rule allocates a complete object of n instances of C:

$$\frac{\begin{array}{c} n > 0 \qquad \ell = \mathcal{G}.\ell_{\mathsf{next}} \\ \mathcal{G}' = \mathcal{G}[\mathsf{LocType}(\ell) \leftarrow (C, n)][\ell_{\mathsf{next}} \leftarrow \mathsf{next}(\ell)] \qquad e' = e[c \leftarrow (\ell, \epsilon, 0, (\mathsf{Repeated}, C :: \epsilon))] \end{array}}{\begin{array}{rlllll} & (\{C\ c[n]; st\}, & stl, & e, & \mathcal{K}, & \mathcal{G}) \\ \to & (st\ , & \epsilon, & e', & \mathsf{Block}(\ell, stl) :: \mathcal{K}, & \mathcal{G}') \end{array}}$$

(Ds++-block-some)

When leaving a block attached to a complete object, this object is deallocated, i.e. removed from the heap, by contrast to κ++. However, the location of the next object to be allocated does not change.

$$\frac{\mathcal{G}' = \mathcal{G}[\mathsf{LocType}(\ell) \leftarrow \bot]}{\begin{array}{rlllll} & (\mathtt{exit}\ (\mathsf{S}\ n), & stl', & e, & \mathsf{Block}(\ell, stl) :: \mathcal{K}, & \mathcal{G}) \\ \to & (\mathtt{exit}\ n\ , & stl, & e, & \mathcal{K}, & \mathcal{G}') \end{array}}$$

(Ds++-exit-some)

$$\frac{\mathcal{G}' = \mathcal{G}[\mathsf{LocType}(\ell) \leftarrow \bot]}{\begin{array}{rlllll} & (\mathtt{return}\ x^?, & stl', & e, & \mathsf{Block}(\ell, stl) :: \mathcal{K}, & \mathcal{G}) \\ \to & (\mathtt{return}\ x^?, & stl, & e, & \mathcal{K}, & \mathcal{G}') \end{array}}$$

(Ds++-return-some)

11.4.6 Virtual function call

Operations such as virtual function call or dynamic cast require to know the generalized dynamic type of the object on which to operate. Contrary to κ++, where the generalized dynamic was computed from the construction states of the objects, in Ds++ the generalized dynamic type is explicitly immediately available in the global state. Thus, it suffices to retrieve the generalized dynamic type from this part of the global state, before dispatching the virtual function or performing the dynamic cast using the generalized dynamic type as the most-derived type.

The Ds++ virtual function call first retrieves the generalized dynamic type from the global state. Once the generalized dynamic type has been determined, it is used as the most-derived type to perform the virtual function call dispatch through the VFDispatch predicate defined in Section 4.4.6.1 (p. 90):

$$\frac{\begin{array}{c} e(x) = \pi = (\ell, (\alpha, i, \sigma)) \qquad \mathcal{G} \vdash \pi : B \qquad \mathcal{G}.\mathsf{gDynType}(\ell, \alpha, i, \sigma) = (\sigma_\circ, \sigma') \\ \mathsf{last}(\sigma_\circ) = C_\circ \qquad \mathsf{VFDispatch}(C_\circ, \sigma', f, B'', \sigma'') \qquad B''.f = f(this, varg_1, \ldots, varg_n)\{body\} \\ \forall j, e(x_j) = v_j \qquad e' = \varnothing[varg_1 \leftarrow v_1] \ldots [varg_n \leftarrow v_n][this \leftarrow (\ell, (\alpha, i, \sigma_\circ @ \sigma''))] \end{array}}{\begin{array}{rlllll} & (x^? := x\text{->}_B f(x_1 \ldots x_n), & stl, & e, & \mathcal{K}, & \mathcal{G}) \\ \to & (body\ , & \epsilon, & e', & \mathsf{Callframe}(x^?, stl, e) :: \mathcal{K}, & \mathcal{G}) \end{array}}$$

(Ds++-virtual-funcall)

11.4.7 Dynamic cast

The dynamic cast language operation first obtains the generalized dynamic type of the subobject, then performs the cast under this object considered as a most-derived object, using the DynCast predicate defined in Section 4.4.5.1 (p. 88). As in s++ and κ++, dynamic casts can be performed only from a *dynamic* class type (*Definition* 4.4.2 p. 89), as required by the C++ Standard [42, 43], corresponding in practice to classes requiring dynamic type data.

However, to prepare for the compilation to CVcm, we add a restriction to the dynamic cast from a class B to B': B' must not be a base class of B, similarly to s++ (s++-dyncast, p. 89); we shall also see in the compiler from κ++ to Ds++ (Section 11.5.2 p. 257) cast to a base class may be safely replaced with a static cast following Lemma 4.4.3 (p. 89).

$$\frac{\begin{array}{c} e(x) = \pi = (\ell, (\alpha, i, \sigma_1)) \\ \mathcal{G} \vdash \pi : B \qquad \mathcal{G}.\mathsf{gDynType}(\ell, \alpha, i, \sigma_1) = (\sigma_\circ, \sigma) \qquad \mathsf{DynCast}(C, \sigma, B, B') = s \\ s' = \mathsf{match}\ s\ \mathsf{with}\ \sigma' \mapsto (\ell, (\alpha, i, \sigma_\circ @ \sigma'))\ \mid \mathsf{NULL} \mapsto \mathsf{NULL}\ \mathsf{end} \qquad e' = e[x' \leftarrow s'] \end{array}}{\begin{array}{rl} & (x' := \mathtt{dynamic\_cast}\langle B' \rangle_B(x),\ \ stl,\ \ e\ ,\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathtt{skip}\ ,\ \ stl,\ \ e',\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}}$$

(Ds++-dyncast)

11.4.8 Static cast

In κ++, base class subobjects may be accessed once they are all constructed. So, a static cast is allowed if the object from which to cast is BasesConstructed, StartedDestructing, but also Constructed: in the latter case, the generalized dynamic type may not be defined in κ++, although a virtual table would be necessary in CVcm when accessing a virtual base class subobject.

We model this necessity by the need for the generalized dynamic type to be defined in Ds++, although its value is not relevant. The existence of the generalized dynamic type is ensured if the object is BasesConstructed, or StartedDestructing, in κ++. So, it will be necessary to show, as part of the κ++-to-Ds++ invariant, that if the object is Constructed in κ++, then its generalized dynamic type exists in Ds++.

Finally, static cast, based on the usual rules by Wasserrab et al. (cf. Section 4.4.4.1 p. 87) follows:

$$\frac{\begin{array}{c} e(x) = \pi = (\ell, (\alpha, i, \sigma)) \\ \mathcal{G}.\mathsf{gDynType}(\pi) \neq \bot \qquad \mathcal{G} \vdash \pi : B \qquad \mathsf{StatCast}(\sigma, B, B', \sigma') \qquad e' = e[x' \leftarrow (\ell, (\alpha, i, \sigma'))] \end{array}}{\begin{array}{rl} & (x' := \mathtt{static\_cast}\langle B' \rangle_B(x),\ \ stl,\ \ e\ ,\ \ \mathcal{K},\ \ \mathcal{G}) \\ \to & (\mathtt{skip}\ ,\ \ stl,\ \ e',\ \ \mathcal{K},\ \ \mathcal{G}) \end{array}}$$

(Ds++-statcast)

11.4.9 Special casts to bases

When a constructor is about to construct one of its direct non-virtual bases (or one of its virtual bases if the object is the most-derived object), it has to provide the constructor with an adjusted `this` pointer.

We could naively compile this operation using static cast. However, this would require the generalized dynamic type of the object to be defined in Ds++, which is not the case at that stage of object construction, so static type cannot be used there.

Another reason is that, actually, the offsets used to perform those special casts to bases in CVcm are always statically known at compile time. Indeed, such a cast to a virtual base is only used if the class is a most-derived object (as base class subobjects must not construct or destruct their virtual base subobjects).

For this reason, there are two cases. The first case is a cast to a direct non-virtual base.

$$\frac{\begin{array}{c} e(x) = \pi = (\ell, (\alpha, i, \sigma)) \\ \mathcal{G} \vdash \pi : D \qquad B \in \mathcal{DNV}_D \qquad e' = e[x' \leftarrow (\ell, (\alpha, i, \sigma@(\mathsf{Repeated}, D :: B :: \epsilon)))] \end{array}}{\begin{array}{rllllll} & (x' := \mathtt{base\_cast}\langle \mathsf{DirectNonVirtual}, B\rangle_D(x), & stl, & e, & \mathcal{K}, & \mathcal{G}) \\ \to & (\mathtt{skip} & , stl, & e', & \mathcal{K}, & \mathcal{G}) \end{array}}$$

(Ds++-casttobase-direct-non-virtual)

The second case casts to a virtual base only from a most-derived object:

$$\frac{\begin{array}{c} e(x) = \pi = (\ell, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon))) \\ \mathcal{G} \vdash \pi : C \qquad B \in \mathcal{V}_C \qquad e' = e[x' \leftarrow (\ell, (\alpha, i, (\mathsf{Shared}, B :: \epsilon)))] \end{array}}{\begin{array}{rllllll} & (x' := \mathtt{base\_cast}\langle \mathsf{Virtual}, B\rangle_C(x), & stl, & e, & \mathcal{K}, & \mathcal{G}) \\ \to & (\mathtt{skip} & , stl, & e', & \mathcal{K}, & \mathcal{G}) \end{array}} \quad \text{(Ds++-casttobase-virtual)}$$

11.4.10 Set dynamic type

As we motivated in Section 11.1.2.2 (p. 238), before constructing the data members of an object π, or before executing the destructor body of the object, its generalized dynamic type has to be modified, as well as the generalized dynamic type of all of its base class subobjects, so that they reflect π as considered a "most-derived object" for the purpose of polymorphic operations.

But also, the generalized dynamic type of the ancestors of which π is a non-virtual *primary* base subobject, have to be erased, to provide for sharing the pointer to virtual table when compiling Ds++ to CVcm. However, this particular notion of *primary* base subobject by virtual pointer sharing is related to object layout, so it should not appear in the semantics of Ds++.

Actually, it suffices to parameterize the semantics of Ds++ with an arbitrary notion of non-virtual *primary* inheritance path:

Hypothesis 11.4.2. *We assume given a boolean function* $\Pi(l) \in \{\mathsf{true}, \mathsf{false}\}$ *over non-virtual inheritance paths. Then, any non-virtual path* l *is said to be* Π-primary *(or* primary *if the context is clear enough) if, and only if,* $\Pi(l) = \mathsf{true}$.

***Definition* 11.4.1.** *Let* D *be a class, and* σ', σ *two inheritance subobjects of* D. σ' *is said to be a* non-virtual Π-primary base *of* σ *(written* $\sigma' \subseteq^{\Pi}_{D} \sigma$*) if, and only if, there exists a* Π*-primary non-virtual path* l *such that* $\sigma' = \sigma@(\mathsf{Repeated}, l)$.

Then, the following definition shows how $\mathcal{G}.\mathsf{gDynType}$ is modified by the "set dynamic type" operation[4]:

4. Coq development: theory `IntermSetDynType`.

***Definition* 11.4.2 (Set dynamic type).** *If $\delta : Ptr \nrightarrow (Path \times Path)$ is a finite map retrieving generalized dynamic types to some subobjects, then, given a class D and a pointer to a subobject $\pi = (\ell, (\alpha, i, \sigma_\circ))$ such that $\sigma_\circ$ is an inheritance subobject of D, the following function written $\delta[\mathsf{setDynType}^{\Pi}_{D}(\pi)]$ retrieves the generalized dynamic types of subobjects after the operation $\mathsf{setDynType}^{\Pi}_{D}(\pi)$:*

$$
\begin{array}{rcccl}
\delta[\mathsf{setDynType}^{\Pi}_{D}(\ell, (\alpha, i, \sigma_\circ))] & : & (\ell, (\alpha, i, \sigma_\circ @ \sigma')) & \mapsto & (\sigma_\circ, \sigma') \\[1em]
 & & \begin{array}{c}(\ell, (\alpha, i, \sigma)) \\ \sigma \textit{ not a base of } \sigma_\circ \\ \sigma_\circ \subseteq^{\Pi}_{D} \sigma\end{array} & \mapsto & \bot \\[2em]
 & & \begin{array}{c}(\ell, (\alpha, i, \sigma)) \\ \sigma \textit{ not a base of } \sigma_\circ \\ \sigma_\circ \not\subseteq^{\Pi}_{D} \sigma\end{array} & \mapsto & \delta(\ell, (\alpha, i, \sigma)) \\[2em]
 & & \begin{array}{c}(\ell'', (\alpha'', i'', \sigma'')) \\ (\ell'', \alpha'', i'') \neq (\ell, \alpha, i)\end{array} & \mapsto & \delta(\ell'', (\alpha'', i'', \sigma''))
\end{array}
$$

This operation is well-defined if the class hierarchy is well-formed[5].

Finally, the "set dynamic type" operation in itself is defined as follows: it takes two arguments known at run-time, the one being the object on which to operate, and the other being the *construction path*. The operation has a defined semantics only if the construction path is actually the inheritance path from the true most-derived object to the subobject on which to operate.

Moreover, the "set dynamic type" operation is flagged with a boolean, true only if the object on which to operate is a most-derived object, or has no virtual bases. This flag only introduces a condition under which the compilation to CVcm can be optimized (in particular, offsets to virtual bases are known at compile time in the case of a most-derived object); this flag has no influence on the actual semantics of the operator (it is always correct, albeit naive, for a κ++-to-Ds++ compiler to produce `setDynType`s invariably flagged false).

$$
\frac{\begin{array}{c}
e(x) = \pi = (\ell, (\alpha, i, \sigma_\circ)) \qquad e(x_{\mathsf{path}}) = (D, \sigma_\circ) \qquad \mathcal{G} \vdash \langle \ell \rangle\ D'[n'] \prec\!\langle \alpha \rangle\!\overset{\mathcal{A}}{\mapsto} D[n] \prec\!\langle (i, \sigma) \rangle\!\overset{\mathcal{CI}}{\mapsto} C \\
\mathcal{G}' = \mathcal{G}[\mathsf{gDynType} \leftarrow \mathcal{G}.\mathsf{gDynType}[\mathsf{setDynType}^{\Pi}_{D}(\pi)]] \\
b = \mathsf{true} \Rightarrow (\sigma_\circ = (\mathsf{Repeated}, D :: \epsilon) \vee \forall V : C \not\overset{\mathcal{V}}{\mapsto} V
\end{array}}{\begin{array}{rlcccc}
 & (\mathtt{setDynType}(x, x_{\mathsf{path}})^{b}_{C}, & stl, & e, & \mathcal{K}, & \mathcal{G}) \\
\to & (\mathtt{skip} \hphantom{xxxxxxxxxxxxx}, & stl, & e, & \mathcal{K}, & \mathcal{G}')
\end{array}}
$$

(Ds++-setdyntype)

11.4.11 Construction paths

The `setDynType` statement shows the actual purpose of construction paths. However, there remains to describe operators allowing to produce construction paths. There are two of them.

The `rootPath` operator produces the construction path for a most-derived class D:

$$
\frac{e' = e[x' \leftarrow (D, (\mathsf{Repeated}, D :: \epsilon))]}{\begin{array}{rlcccc}
 & (x' := \mathtt{rootPath}(D), & stl, & e, & \mathcal{K}, & \mathcal{G}) \\
\to & (\mathtt{skip} \hphantom{xxxxxxxxx}, & stl, & e', & \mathcal{K}, & \mathcal{G})
\end{array}}
$$

(Ds++-rootpath)

5. Coq development: theory `IntermSetDynTypeWf`.

Then, given a construction path, the `basePath` operator produces the construction path for a direct non-virtual base class, or for a (direct or indirect) virtual path, the latter case only if the given construction path corresponds to a most-derived object:

$$\frac{e(x) = (D,\sigma) \qquad D \mathrel{-\!\langle\sigma\rangle\!\overset{\mathcal{I}}{\rightarrow}} C \qquad B \in \mathcal{DNV}_C \qquad e' = e[x' \leftarrow (D, \sigma @ (\mathsf{Repeated}, C :: B :: \epsilon))]}{\begin{array}{lllll} & (x' := \mathtt{basePath}(x, C, \mathsf{Repeated}, B), & \mathit{stl}, & e\,, & \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathtt{skip} \hfill, & \mathit{stl}, & e', & \mathcal{K}, \quad \mathcal{G}) \end{array}}$$

(Ds++-basepath-direct-non-virtual)

$$\frac{e(x) = (D, (\mathsf{Repeated}, D :: \epsilon)) \qquad D \overset{\mathcal{V}}{\rightarrow} V \qquad e' = e[x' \leftarrow (D, (\mathsf{Shared}, V :: \epsilon))]}{\begin{array}{lllll} & (x' := \mathtt{basePath}(x, D, \mathsf{Shared}, V), & \mathit{stl}, & e\,, & \mathcal{K}, \quad \mathcal{G}) \\ \to & (\mathtt{skip} \hfill, & \mathit{stl}, & e', & \mathcal{K}, \quad \mathcal{G}) \end{array}}$$

(Ds++-basepath-virtual)

11.5 A compiler from κ++ to Ds++

Now we present and prove our compilation scheme[6] [7] from κ++ to Ds++ by compiling constructors and destructors into ordinary functions. We show that other object-oriented features of C++ irrelevant to construction are left unchanged.

11.5.1 Compilation contexts

In this section, we investigate the parameters on which the compilation of a statement must depend. We shall group them into a single structure called *compilation context.*

Variables If v is a κ++ variable name, then it is injectively translated into a Ds++ variable name written $\overline{v}$. This allows us to introduce, for the purpose of compilation, new variables written $\underline{w}$ such that, for any v, w: $\overline{v} \neq \underline{w}$.[8] Then, if $\underline{w}$ is a Ds++ variable, then w is called a *compiler-purpose variable.*

However, we shall see that the number of different needed compiler-purpose variables actually depends on the size of the program.

Blocks Consider a κ++ block defining a block-scoped object. Then, the semantics of κ++ makes this object available to the programmer who provides a variable name, say v, to this purpose:

```
{
  C v[2];
  ...
}
```

6. Coq development: theory `Cppsem2IntermAux`.
7. Coq development: theory `Cppsem2Interm`.
8. In our Coq development, this is easy because variables are `positive` integers in binary representation, so that, for instance, $\overline{v} \underset{\text{def.}}{=\!=} 2 \cdot v$ whereas $\underline{w} \underset{\text{def.}}{=\!=} 2 \cdot w + 1$.

κ++ stores in v a pointer to the first cell of the array declared by the programmer. But v is not a particular variable: nothing prevents overwriting this value in v. But when compiling to Ds++, the compiler has to destruct the object when leaving the block. So, a pointer to the block-scoped object has to be stored into a compiler-purpose variable whose value never changes during the execution of the block. This compiler-purpose variable will be said to be *attached* to the block.

Then, to enforce this, it is necessary that a further embedded block have its attached variable different from the attached variables of its embedding blocks. This shows that the number of different needed compiler-purpose variables depends on the size of the program – actually, on the level of block nesting.

For those two reasons, the compilation of a statement depends on the domain of available compiler-purpose variables. But it also depends on the variables attached to blocks: when compiling `exit` or `return`, destruction of all exited blocks has to be initiated.

Implicit variables for initializers An initializer terminates with a call to a constructor, or with a initScalar to give a scalar field its initial value. Those statements operate on an implicit object. In Ds++, this object must be made explicit. So, the compilation of a statement depends on a compiler-purpose variable introduced to make this object explicit.

Choice of constructors and destructors We chose to duplicate constructors and destructors at compile time: each class features both a constructor for a most-derived object (called from within the initializer for a block-scoped object or for a structure data member) on the one hand, and a constructor for a base class subobject (called from within a base class initializer) on the other hand. So, it is necessary to statically know which to choose when compiling a constructor call.

Inlining destructor bodies In κ++, destructor bodies end with `return`. So, one naive implementation would be to isolate the destructor body in a separate function, which would be called by the actual compiled destructor. But it is possible to *inline* the destructor body by including it into a block, and replacing each `return` with an appropriate `exit`. So, the compilation of statements also depends on whether the statement being compiled is within the body of a destructor.

Summary To sum up the parameters on which the compilation of a statement must depend, we introduce the notion of *compilation context*:

***Definition* 11.5.1.** *A* compilation context Γ *is a tuple whose constituents are:*

- Γ.Used *is the set of already used compiler-purpose variables*[9]
- Γ.curobj *is the compiler-purpose variable corresponding to the object on which to call the constructor (irrelevant if not in an initializer)*
- Γ.curpath *is the compiler-purpose variable corresponding to the construction path with which to call the constructor (irrelevant if not in an initializer)*
- Γ.isMostDerived *is a boolean,* true *if and only if the constructor or destructor for a most-derived object is to be called (irrelevant if not in an initializer)*

9. In the Coq development, compiler-purpose variables are `positive`s, and this set is a finite interval $[1 \ldots maxvar)$, so it is represented by its upper bound $maxvar$.

- $\Gamma.\mathsf{Blocks}$ *is the list of the compiler-purpose variables attached to each block: its elements are of the form* $\perp$ *for blocks defining no stack object, or* $(newobj, Used, (C, n))$ *for a block defining a scope object, which is an array of* n *cells of class* C*, where newobj is the compiler-purpose variable corresponding to the object, and Used is the set of already used compiler-purpose variables at this block*
- $\Gamma.\mathsf{curfield}$ *is a tuple* (C, f) *where* C *is the class of the current object to construct, and* f *is the scalar field to initialize (irrelevant if not in the initializer of some scalar field)*
- $\Gamma.\mathsf{isDestructorBody}$ *is a boolean,* true *if and only if the statement being compiled is part of the body of a destructor*

We fix an initial compilation context *written* $\Gamma_\circ$*, such that:*

$$\Gamma_\circ.\mathsf{Used} = \varnothing$$

$$\Gamma_\circ.\mathsf{Blocks} = \epsilon$$

$$\Gamma_\circ.\mathsf{isDestructorBody} = \mathsf{false}$$

the other values being irrelevant.[10] *This initial compilation context serves in the compilation of the main program statement and function bodies.*

Then, we introduce the compiler notations:

***Definition* 11.5.2.** *Given a compilation context* Γ*, a* κ++ *statement st is compiled into a Ds++ statement written* $[\![st]\!]^\Gamma$*, which is computed by structural induction on st.*

By contrast, the compilation of functions, constructors, destructors does not depend on the compilation context and will be simply written $[\![\cdot]\!]$*.*

11.5.2 Statements unrelated to construction and destruction

Operations that compile trivially Tests, loops, sequences and built-in operations are trivially transformed:

$$\begin{aligned}
[\![\texttt{if}(b)\ s_{\mathsf{true}}\ \texttt{else}\ s_{\mathsf{false}}]\!]^\Gamma &\underset{\text{def.}}{=\!=} \texttt{if}(\overline{b})\ [\![s_{\mathsf{true}}]\!]^\Gamma\ \texttt{else}\ [\![s_{\mathsf{false}}]\!]^\Gamma \\
[\![\texttt{loop}\ s]\!]^\Gamma &\underset{\text{def.}}{=\!=} \texttt{loop}\ [\![s]\!]^\Gamma \\
[\![s_1; s_2]\!]^\Gamma &\underset{\text{def.}}{=\!=} [\![s_1]\!]^\Gamma; [\![s_2]\!]^\Gamma \\
[\![x'^? := op(x_1, \ldots, x_n)]\!] &\underset{\text{def.}}{=\!=} \overline{x'}^? := op(\overline{x_1}, \ldots, \overline{x_n})
\end{aligned}$$

The semantics of field and array accesses, as well as non-virtual function calls, are the same in Ds++ as in κ++, the only difference being fewer conditions, so their compilation is trivial:

$$\begin{aligned}
[\![x' := x\texttt{->}_C f]\!]^\Gamma &\underset{\text{def.}}{=\!=} \overline{x'} := \overline{x}\texttt{->}_C f \\
[\![x\texttt{->}_C f := x']\!]^\Gamma &\underset{\text{def.}}{=\!=} \overline{x}\texttt{->}_C f := \overline{x'} \\
[\![x' := x[x_{\mathsf{index}}]]\!] &\underset{\text{def.}}{=\!=} \overline{x'} := \overline{x}[\overline{x_{\mathsf{index}}}] \\
[\![x'^? := x\texttt{->}C\texttt{::}f(x_1, \ldots, x_n)]\!]^\Gamma &\underset{\text{def.}}{=\!=} \overline{x'}^? := \overline{x}\texttt{->}C\texttt{::}f(\overline{x_1}, \ldots, \overline{x_n})
\end{aligned}$$

10. In the Coq development, such irrelevant values are actually of option type.

Static casts also compile trivially, the access condition does not modify the semantics:

$$[\![x' := \texttt{static\_cast}\langle B\rangle_C(x)]\!]^\Gamma \underset{\text{def.}}{=\!=} \overline{x'} := \texttt{static\_cast}\langle B\rangle_C(\overline{x})$$

Virtual function calls also compile trivially:

$$[\![x'^? := x\texttt{->}_C f(x_1, \ldots, x_n)]\!]^\Gamma \underset{\text{def.}}{=\!=} \overline{x'}^? := \overline{x}\texttt{->}_C f(\overline{x_1}, \ldots, \overline{x_n})$$

Static function call The static (non-class-member) functions of the Ds++ compiled program are not exactly the same as the static functions of the κ++ source program. Indeed, the compiler adds constructors and destructors along with them. So, we have to follow the same convention as for variables:

***Definition* 11.5.3.** *Let f be a static function of the κ++ program, then the name of the corresponding static function of the Ds++ program will be written $\overline{f}$. This name translation is injective.*

Then, static function calls are compiled in a straightforward way:

$$[\![x'^? := f(x_1, \ldots, x_n)]\!]^\Gamma \underset{\text{def.}}{=\!=} \overline{x'}^? := \overline{f}(\overline{x_1}, \ldots, \overline{x_n})$$

Dynamic cast The case of dynamic cast from a class B to a class B' is slightly different. If B' is a base of B, then the dynamic cast is turned into a static cast. Otherwise, it is left unchanged:

$$[\![x' := \texttt{dynamic\_cast}\langle B\rangle_C(x)]\!]^\Gamma \underset{\text{def.}}{=\!=} \begin{cases} \overline{x'} := \texttt{static\_cast}\langle B\rangle_C(\overline{x}) & \text{if } B \text{ is a base class of } C \\ \overline{x'} := \texttt{dynamic\_cast}\langle B\rangle_C(\overline{x}) & \text{otherwise} \end{cases}$$

This transformation is not due to construction or destruction [11], but it simplifies the compilation from Ds++ to CVcm.

Blocks with no stack object A block with no object compiles almost trivially, except that a block can be automatically exited in κ++. To solve this issue, we systematically append an `exit` 1 at the end of the block body *before* compiling. However, we shall see later how to compile this `exit` 1 more precisely.

$$[\![\{st\}]\!]^\Gamma \underset{\text{def.}}{=\!=} \{[\![st; \texttt{exit } 1]\!]^{\Gamma'}\}$$

where Γ' is the compilation context updated as follows:

$$\Gamma' = \Gamma[\mathsf{Blocks} \leftarrow \bot :: \Gamma.\mathsf{Blocks}]$$

11. C++ allows the use of the dynamic cast operator for casts to bases, which are not truly dynamic casts. Likewise, C++ allows the use of unqualified calls for non-virtual function calls, so they have to be differentiated from virtual function calls in a similar way (by statically determining whether there is a base class defining a virtual function with the name and argument types prescribed by the call). We believe that the correctness proof of such a transformation differentiating class member function calls is as simple as differentiating static casts from dynamic casts. However, we argue that such a transformation would have to happen when generating the κ++ program rather than the Ds++ program, as it is much easier to have them already differentiated when studying the semantics of κ++.

to reflect that a new block with no stack object has been entered.

`exit` 1 will be similarly appended when compiling the body of a block with a stack object. This allows for trivially compiling `skip`:

$$[\![\texttt{skip}]\!]^{\Gamma} \underset{\text{def.}}{=\!=} \texttt{skip}$$

The transformation trick has to be used also for compiling the body of a function, constructor, destructor, or an initializer: indeed, the semantics of a function body without **`return`** gets stuck only after successfully applying the "automatic block exit" rule in κ++, so this step must be matched in Ds++.

As $[\![\cdot]\!]^{\cdot}$ is computed by structural induction on the statement, $[\![st; \texttt{exit}\ 1]\!]^{\Gamma'}$ is actually rendered as $[\![st']\!]^{\Gamma'}; \chi_{\Gamma'}$ where $\chi_{\Gamma'}$ is an auxiliary to compile `exit` 1. So we have to ensure that $\chi_{\Gamma'}$ does not recursively call the compilation function $[\![\cdot]\!]^{\cdot}$. We shall see later how this is ensured.

11.5.3 Blocks with stack objects

Consider a κ++ block defining a stack object of type C, which is a structure array with n cells:

$$\{C\ c[n] = \mathit{inits}; st\}$$

Then, we must take care of the following issues:
- Construct the cells of c from 0 up to $n-1$
- Compile its body st remembering its attached object for destruction
- Destruct the cells of c from $n-1$ down to 0

As we said before, c is not a particular variable in κ++, nothing prevents it from being overwritten by the programmer, including in the initializers *inits* themselves. So, we have to provide a compiler-purpose variable to remember the object for its destruction, and even for the construction of its further cells. To make proofs easier, we choose different compiler-purpose variables for those two purposes.

Structure array cell initializer When constructing an array cell, the construction of all the other cells is remembered by a unique κ++ continuation stack frame Kconstrarray or Kconstrothercells. We match this stack frame with a Ds++ Block continuation stack frame. So, each cell initializer has to be embedded into a Ds++ block with no stack object, so that the further statements of the enclosing block comprises the construction of the further cells.

So, a compiled initializer performs the following operations:
- access the corresponding array cell as the implicit object to be constructed.
- execute the initializer, with the construction path corresponding to a most-derived object
- leave its block

To this purpose, several compiler-purpose variables are needed[12]:
- *index* for the cell index
- *newobj* for the pointer to the cell to construct
- *newpath* for the construction path

12. In our Coq development, the proof is made simpler by reusing those compiler-purpose variables throughout all the cell initializers for a given array

So, if *obj* is a compiler-purpose variable supposed to hold the pointer to the first cell of the whole array, then we define the compilation $\mathsf{compileConstrArrayInit}(C, i, \mathit{init}, \mathit{obj}, \Gamma)$ of the initializer *init* for the i-th cell as follows:

$$\begin{array}{rcl} \mathsf{compileConstrArrayInit}(C, i, \mathit{init}, \mathit{obj}, \Gamma) & \underset{\text{def.}}{=\!=} & \underline{\mathit{index}} := i; \\ (\{\mathit{index} \notin \Gamma.\mathsf{Used}) & & \underline{\mathit{newobj}} := \underline{\mathit{obj}}[\underline{\mathit{index}}]_C; \\ & & \underline{\mathit{newpath}} := \mathtt{rootPath}(C); \\ & & \{[\![\mathit{init}; \mathtt{exit}\ 1]\!]^{\Gamma'}\} \end{array}$$

where Γ' is the compilation context updated as follows:

$$\begin{array}{rcll} \Gamma' & = & \Gamma & [\mathsf{Used} \leftarrow \Gamma.\mathsf{Used} \uplus \{\mathit{newobj}\} \uplus \{\mathit{newpath}\}] \\ & & & [\mathsf{curobj} \leftarrow \mathit{newobj}] \\ & & & [\mathsf{curpath} \leftarrow \mathit{newpath}] \\ & & & [\mathsf{Blocks} \leftarrow \epsilon] \end{array}$$

to reflect that the implicit object on which to call the constructor must be the cell being constructed. The stack of blocks, however, is emptied, because the blocks to be opened within the initializer are irrelevant to the blocks enclosing the initializer.

Array initializer When entering a block with a stack object, the execution of the body is postponed onto the κ++ continuation stack by a Kcontinue frame. We match this stack frame with a Ds++ Block continuation stack frame. So, the whole construction of the array has to be embedded in a Ds++ block with no stack object, and this block has to be exited after constructing the last array cell.

For each cell, its index i is known at compile time, so finally, the initializers for an array are compiled by unfolding a "for" loop at compile time[13].

Assuming that the compiler-purpose variable *obj* holds a pointer to the first cell of an array of C, we define $\mathsf{compileConstrArray}(C, n, i, \mathit{inits}, \mathit{obj}, \Gamma)$ to compile the inializers for that array as follows:

$$\begin{array}{rcl} \mathsf{compileConstrArray}(C, n, i, \mathit{inits}, \mathit{obj}, \Gamma) & \underset{\text{def.}}{=\!=} & \mathsf{compileConstrArrayInit}(C, i, \mathit{inits}_i, \mathit{obj}, \Gamma); \\ (i < n) & & \mathsf{compileConstrArray}(C, n, i+1, \mathit{inits}, \mathit{obj}, \Gamma) \\ \mathsf{compileConstrArray}(C, n, n, \mathit{inits}, \mathit{obj}, \Gamma) & \underset{\text{def.}}{=\!=} & \mathtt{exit}\ 1 \end{array}$$

Block body Finally, once the construction ends, the body of the block is executed, so that the block is compiled as:

$$\begin{array}{rcl} [\![\{C\ c[n] = \mathit{inits}; \mathit{st}\}]\!]^{\Gamma} & \underset{\text{def.}}{=\!=} & \{ \\ & & \quad C\ \underline{\mathit{obj}'}[n]; \\ & & \quad \overline{c} := \underline{\mathit{obj}'}; \\ & & \quad \underline{\mathit{obj}} := \underline{\mathit{obj}'}; \\ & & \quad \{\mathsf{compileConstrArray}(C, n, 0, \mathit{inits}, \mathit{obj}, \Gamma_1)\}; \\ & & \quad [\![\mathit{st}; \mathtt{exit}\ 1]\!]^{\Gamma_2} \\ & & \} \end{array}$$

13. Coq development: theory `ForLoop`.

where Γ_1 is the compilation context used during construction, updated as follows:

$$\Gamma_1 = \Gamma[\mathsf{Used} \leftarrow \Gamma.\mathsf{Used} \uplus \{obj\} \uplus \{obj'\}]$$

and Γ_2 is the compilation context used for the block body, updated as follows:

$$\begin{array}{rcll} \Gamma_2 & = & \Gamma & [\mathsf{Used} \leftarrow \Gamma.\mathsf{Used} \uplus \{obj'\}] \\ & & & [\mathsf{Blocks} \leftarrow (obj', \Gamma.\mathsf{Used}, (C, n)) :: \Gamma.\mathsf{Used}] \end{array}$$

to reflect the new block with a stack object attached to the compiler-purpose variable obj'. To make the proof simpler, a different variable obj is used for the construction process. This explains why Γ_1 must be used instead of Γ during the construction process: to prevent obj' from reuse.

11.5.4 Constructor names

Let κ be a constructor for class C. Then, κ is compiled into two Ds++ static functions, whose names are written $\underline{\kappa_{\mathsf{true}}}$ corresponding to the constructor for a most-derived object, and $\underline{\kappa_{\mathsf{false}}}$ corresponding to the constructor for a base class subobject. Such names are guaranteed to be distinct from the names of static functions ($\overline{f}$ for each κ++ static function f). They are computed through a function taking as argument the types of the arguments of κ: this is a form of *name mangling* [14].

This process is injective, except for a slight optimization: if C has no virtual bases, then $\underline{\kappa_{\mathsf{true}}} = \underline{\kappa_{\mathsf{false}}}$. This allows the same function to serve as constructor for both a most-derived object and a base class subobject. We shall see later how this peculiarity is exploited when compiling a constructor, and why it is correct in such cases.

11.5.5 Constructor calls

Now, suppose we are compiling an initializer calling κ. Then, determining whether the constructor for the most-derived object or for a base class subobject must be called, is done through the $\Gamma.\mathsf{isMostDerived}$ parameter of the compilation context.

Then, the constructor is called using the implicit pointer to the object to construct, assumed to be stored in the $\Gamma.\mathsf{curobj}$ compiler-purpose variable, and the construction path assumed to be stored in the $\Gamma.\mathsf{curpath}$ compiler-purpose variable.

Finally, the semantics of κ++ imposes to have left all blocks within the initializer before calling the constructor; and so the call terminates the execution of the initializer. In Ds++ this translates to leaving the block in which the initializer was embedded:

$$\begin{array}{rcl} [\![\kappa(x_1, \dots, x_n)]\!] & \underset{\text{def.}}{=\!=} & \underline{\kappa_{\Gamma.\mathsf{isMostDerived}}}(\underline{\Gamma.\mathsf{curobj}}, \underline{\Gamma.\mathsf{curpath}}, \overline{x_1}, \dots, \overline{x_n}); \\ & & \mathtt{exit}\ 1 \end{array}$$

11.5.6 Destructor names

The destructor for a class C is compiled into two Ds++ static functions, whose names are written $\underline{\sim C_{\mathsf{true}}}$ corresponding to the destructor for a most-derived object, and $\underline{\sim C_{\mathsf{false}}}$ corresponding to the destructor for a base class subobject. Such names are guaranteed to be distinct from the names of static functions, as well as from the names of constructors.

14. Coq development: theory `Mangle`.

This process is injective. Contrary to constructors, we did not optimize destructor sharing. Indeed, such sharing would impose the use of the destructor for a base-class subobject, even for a most-derived object. This would make the destructor for a most-derived object perform a useless setDynType operation. However, such an optimization could be done – and would make sense – for non-dynamic classes (with no polymorphic behaviour: no virtual bases, no virtual functions): then, in such cases, even though the setDynType operation were useless in Ds++, it would have been compiled to a no-op in CVcm, as we shall see later in this chapter.

11.5.7 Leaving blocks

Leaving a block requires to destruct its attached stack object, if any. However, κ++ provides the feature of leaving several blocks at the same time. In this case, all stack objects attached to all blocks from which to exit are to be destructed. The issue is that it is necessary to destruct those objects before leaving any block. Indeed, a block can have several exit points; but if we destructed objects one block at a time, then any block exit would branch to the same point in the immediately enclosing block, so that it would be impossible to implement multiple points of multiple block exits.

So, a `exit` n statement shall be compiled into "destruct n objects, then exit n blocks". The destruction phase looks at Γ.Blocks to retrieve the variables holding the corresponding stack objects to destruct, and pick n off the list, destruct them, then discard them from the list.

The destruction of a structure array is simply performed by a destructor call on each cell.

Structure array cell destruction Similarly to cell construction, for each cell of an array pointed to by the compiler-purpose variable *obj*, the cell is accessed, but there is no counterpart to the initializer: instead, the destructor for a most-derived object is directly called, provided with the construction path corresponding to a most-derived object.

A Kdestrcell frame is present on top of the κ++ continuation stack frame during cell destruction (at least during the destruction of its non-virtual part). But, contrary to construction, we match this frame with a Callframe frame in Ds++, corresponding to the return from destructor call, so it is not necessary to embed the call in a block.

So, we define compileDestrArrayFin(C, i, obj, Γ) to compile the destruction of the i-th cell is as follows:

$$\begin{array}{rcl} \mathsf{compileDestrArrayFin}(C, i, obj, \Gamma) & \overset{=}{\underset{\text{def.}}{=}} & \underline{index} := i; \\ (\{index\} \uplus \{newobj\} \uplus \{newpath\} \# \Gamma) & & \underline{newobj} := \underline{obj}[\underline{index}]_C; \\ & & \underline{newpath} := \mathtt{rootPath}(C); \\ & & \underline{\sim C_{\mathsf{true}}}(\underline{newobj}, \underline{newpath}) \end{array}$$

Structure array destruction When destructing a stack object, a Kcontinue frame is present on top of the κ++ continuation stack frame to indicate the pending exit statement (and the further block objects to destruct). We match this frame with a Block frame in Ds++, so that array destruction has to be embedded in a block with no stack objects, and leave that block once the cell 0 has been destructed.

Then, similarly to construction, we define compileDestrArray(C, i, obj, Γ) to compile the destruction of cells i down to 0 by unfolding a "for" loop at compile time:

$$\begin{array}{rcl}\mathsf{compileDestrArray}(C, i, \mathit{obj}, \Gamma) & \overset{}{\underset{\text{def.}}{=\!=}} & \mathsf{compileDestrArrayFin}(C, i, \mathit{obj}, \Gamma); \\ (0 \leq i) & & \mathsf{compileDestrArray}(C, i-1, \mathit{obj}, \Gamma) \\ \mathsf{compileDestrArray}(C, -1, \mathit{obj}, \Gamma) & \underset{\text{def.}}{=\!=} & \mathtt{exit}\ 1\end{array}$$

Leaving n blocks We define $\mathsf{compileDiscard}(n, \Gamma, st)$ to discard n κ++ blocks from $\Gamma.\mathsf{Blocks}$, destructing their block objects on the way, before executing st. However, blocks are discarded from κ++ without leaving their Ds++ counterparts. There are three cases:

- if $n = 0$, then st is executed;
- otherwise, if the current block is attached with no object, then the block is simply discarded and the destruction of the further $n-1$ block objects resumes, but a `skip` statement has to be inserted to match the execution steps;
- otherwise, the current block object is destructed, then the further $n - 1$ block objects resumes with the new set of already used compiler-purpose variables provided with the block.

$$\begin{array}{rcl}\mathsf{compileDiscard}(0, \Gamma, st) & \underset{\text{def.}}{=\!=} & st \\ \mathsf{compileDiscard}(\mathsf{S}n, \Gamma, st) & \underset{\text{def.}}{=\!=} & \mathtt{skip}; \\ (\Gamma.\mathsf{Blocks} = \bot :: \mathit{Blocks}') & & \mathsf{compileDiscard}(n, \Gamma', st) \\ (\Gamma' = \Gamma[\mathsf{Blocks} \leftarrow \mathit{Blocks}']) & & \\ \mathsf{compileDiscard}(\mathsf{S}n, \Gamma, st) & \underset{\text{def.}}{=\!=} & \mathsf{compileDestrArray}(C, n-1, \mathit{obj}, \Gamma); \\ (\Gamma.\mathsf{Blocks} = (\mathit{obj}, \mathit{Used}', (C, n)) :: \mathit{Blocks}') & & \mathsf{compileDiscard}(n, \Gamma', st) \\ (\Gamma' = \Gamma[\mathsf{Blocks} \leftarrow \mathit{Blocks}'][\mathsf{Used} \leftarrow \mathit{Used}']) & & \end{array}$$

Then, leaving n blocks is compiled as follows:

$$[\![\mathtt{exit}\ n]\!]^{\Gamma} \underset{\text{def.}}{=\!=} \mathsf{compileDiscard}(n, \Gamma, \mathtt{exit}\ n)$$

However, such an equation is very difficult to handle within proofs. To make them easier, we add a further parameter to the compilation context:

***Definition* 11.5.4.** *In addition to* Used, curobj, curpath, isMostDerived, Blocks, curfield, isDestructorBody*, a compilation context* Γ *also holds a further parameter,* furtherBlocks*, which is a natural number indicating how many blocks have been exited in* κ++ *but not yet in Ds++.*

This number is 0 *in the* $\Gamma_\circ$ *initial compilation context.*

and then, we redefine the compilation of `exit` as follows:

$$[\![\mathtt{exit}\ n]\!]^{\Gamma} \underset{\text{def.}}{=\!=} \mathsf{compileDiscard}(n, \Gamma, \mathtt{exit}\ (n + \Gamma.\mathsf{furtherBlocks}))$$

We note that, during the compilation process itself, furtherBlocks is invariably set to 0; however, this parameter will change during the proof. Indeed, after $\Gamma.\mathsf{furtherBlocks}$ block exits, the actual statement in the Codepoint κ++ execution state is `exit` $(n - \Gamma.\mathsf{furtherBlocks})$, which is the actual number of blocks *remaining* to be exited in κ++, whereas the final exit statement in Ds++ (after destructing all block objects) is still `exit` n.

The interest of adding a parameter lies in the following lemma, relating $[\![\mathtt{exit}\ (\mathsf{S}\ n)]\!]^{\Gamma}$ with $[\![\mathtt{exit}\ n]\!]^{\Gamma'}$ for some Γ' (i.e. uniquely in terms of $[\![\cdot]\!]^{\cdot}$ and not compileDiscard):

Lemma 11.5.1. *The following equations hold:*

$$\begin{array}{rcl}
[\![\texttt{exit } 0]\!]^{\Gamma} & = & \texttt{exit } \Gamma.\mathsf{furtherBlocks} \\[1ex]
[\![\texttt{exit } (\mathsf{S}n)]\!]^{\Gamma} & = & \texttt{skip}; \\
(\Gamma.\mathsf{Blocks} = \bot :: \mathit{Blocks}') & & [\![\texttt{exit } n]\!]^{\Gamma'} \\
(\Gamma_1 = \Gamma[\mathsf{Blocks} \leftarrow \mathit{Blocks}']) & & \\
(\Gamma' = \Gamma_1[\mathsf{furtherBlocks} \leftarrow \mathsf{S}(\Gamma.\mathsf{furtherBlocks})]) & & \\[1ex]
[\![\texttt{exit } (\mathsf{S}n)]\!]^{\Gamma} & = & \mathsf{compileDestrArray}(C, n-1, \mathit{obj}, \Gamma); \\
(\Gamma.\mathsf{Blocks} = (\mathit{obj}, \mathit{Used}', (C, n)) :: \mathit{Blocks}') & & [\![\texttt{exit } n]\!]^{\Gamma'} \\
(\Gamma_1 = \Gamma[\mathsf{Blocks} \leftarrow \mathit{Blocks}']) & & \\
(\Gamma_2 = \Gamma_1[\mathsf{Used} \leftarrow \mathit{Used}']) & & \\
(\Gamma' = \Gamma_2[\mathsf{furtherBlocks} \leftarrow \mathsf{S}(\Gamma.\mathsf{furtherBlocks})]) & &
\end{array}$$

Compiling an `exit` does not recursively call the compilation function $[\![\cdot]\!]^{\cdot}$. This ensures that, when a statement st requires the compilation of another statement of the form st'; `exit` 1 (where st' is a structural subterm of st), it is actually rendered as

$$[\![st']\!]^{\Gamma}; \mathsf{compileDiscard}(n, \Gamma, \texttt{exit } (n + \Gamma.\mathsf{furtherBlocks}))$$

so that $[\![\cdot]\!]^{\cdot}$ can be actually defined by structural induction on the κ++ statement.

Return from function `return`ing from a function first requires all blocks to exit. The number of all those blocks is actually $\mathsf{length}(\Gamma.\mathsf{Blocks})$, the length of the list of blocks in the compilation context. However, care must be taken when compiling a `return` from within a destructor body: in that case, as the body is inlined in the destructor under the form of a block, `return` has to be changed to an appropriate `exit`.

$$[\![\texttt{return } x^?]\!]^{\Gamma} \underset{\text{def.}}{=\!=\!=} \mathsf{compileDiscard}(\mathsf{length}(\Gamma.\mathsf{Blocks}), \Gamma, \mathit{stm})$$

with stm being given as:

$$\mathit{stm} = \begin{cases} \texttt{exit } (\mathsf{S}(\mathsf{length}(\Gamma.\mathsf{Blocks})) + \Gamma.\mathsf{furtherBlocks}) & \text{if } \Gamma.\mathsf{isDestructorBody} = \mathsf{true} \\ \texttt{return } \overline{x}^? & \text{otherwise} \end{cases}$$

Indeed, to leave a destructor body, the Ds++ code has to leave:
- the block embedding the initializer (hence S)
- the remaining κ++ blocks (hence $\mathsf{length}(\Gamma.\mathsf{Blocks})$)
- but also the blocks already exited in κ++ but not yet in Ds++ (hence $\Gamma.\mathsf{furtherBlocks}$)

Similarly, the following lemma eliminates the $\mathsf{compileDiscard}$ terms:

Lemma 11.5.2. *The following equations hold:*

$$\begin{array}{rcl}
[\![\texttt{return } x^?]\!]^\Gamma & = & \texttt{return } \overline{x}^? \\
(\Gamma.\mathsf{Blocks} = \epsilon) & & \\
(\Gamma.\mathsf{isDestructorBody} = \mathsf{false}) & & \\
[\![\texttt{return } x^?]\!]^\Gamma & = & \texttt{exit}(\mathsf{S}(\Gamma.\mathsf{furtherBlocks})) \\
(\Gamma.\mathsf{Blocks} = \epsilon) & & \\
(\Gamma.\mathsf{isDestructorBody} = \mathsf{true}) & & \\
[\![\texttt{return } x^?]\!]^\Gamma & = & \texttt{skip}; \\
(\Gamma.\mathsf{Blocks} = \bot :: \mathit{Blocks}') & & [\![\texttt{return } x^?]\!]^{\Gamma'} \\
(\Gamma_1 = \Gamma[\mathsf{Blocks} \leftarrow \mathit{Blocks}']) & & \\
(\Gamma' = \Gamma_1[\mathsf{furtherBlocks} \leftarrow \mathsf{S}(\Gamma.\mathsf{furtherBlocks})]) & & \\
[\![\texttt{return } x^?]\!]^\Gamma & = & \mathsf{compileDestrArray}(C, n-1, \mathit{obj}, \Gamma); \\
(\Gamma.\mathsf{Blocks} = (\mathit{obj}, \mathit{Used}', (C, n)) :: \mathit{Blocks}') & & [\![\texttt{return } x^?]\!]^{\Gamma'} \\
(\Gamma_1 = \Gamma[\mathsf{Blocks} \leftarrow \mathit{Blocks}']) & & \\
(\Gamma_2 = \Gamma_1[\mathsf{Used} \leftarrow \mathit{Used}']) & & \\
(\Gamma' = \Gamma_2[\mathsf{furtherBlocks} \leftarrow \mathsf{S}(\Gamma.\mathsf{furtherBlocks})]) & &
\end{array}$$

11.5.8 Functions

In the previous sections, we described how to compile the statements whose semantics are defined in function bodies. Now we can pack all together and give the compilation scheme for function bodies. Compiling a function does not depend on the compilation context: the compilation of bodies use the $\Gamma_\circ$ initial compilation context.

Static functions Recall that the name f of a static function has to be translated into $\overline{f}$ because static functions are added to the program for compiling constructors and destructors, so it is necessary to avoid name clashes. Then we have:

$$[\![f(x_1, \dots, x_n)\{st\}]\!] \underset{\text{def.}}{=\!=} \overline{f}(\overline{x_1}, \dots, \overline{x_n})\{[\![st; \texttt{exit } 1]\!]^{\Gamma_\circ}\}$$

Class member function The compilation of a (virtual or non-virtual) class member function is straightforward:

$$[\![\texttt{virtual}^? \mathit{this}\texttt{->}f(x_1, \dots, x_n)\{st\}]\!] \underset{\text{def.}}{=\!=} \texttt{virtual}^? \overline{\mathit{this}}\texttt{->}f(\overline{x_1}, \dots, \overline{x_n})\{[\![st; \texttt{exit } 1]\!]^{\Gamma_\circ}\}$$

11.5.9 Constructors

In the previous sections, we showed how to compile static and class member functions. Now it remains to compile constructors and destructors.

Let D be a class, and κ be a constructor of D. Then, it will be compiled into two static functions, $\underline{\kappa_{\mathsf{true}}}$ corresponding to the constructor for a most-derived object, and $\underline{\kappa_{\mathsf{false}}}$ for a base class subobject.

Suppose that D has $V_1, \dots, V_v$ direct or indirect virtual bases, $B_1, \dots, B_b$ direct non-virtual bases, and $M_1, \dots, M_m$ data members; and that the constructor κ is written:

$$D(\kappa.\mathit{this}, \kappa.\mathit{arg}_1, \ldots, \kappa.\mathit{arg}_a) : \kappa.\mathit{inits}\{\kappa.\mathit{body}\}$$

where $\kappa.\mathit{this}, \kappa.\mathit{arg}_1, \ldots, \kappa.\mathit{arg}_a$ are the arguments of the constructor, $\kappa.\mathit{inits}$ are the initializers (such that $\kappa.\mathit{inits}(\mathsf{Virtual}, V_i)$ is the initializer for the virtual base V_i, $\kappa.\mathit{inits}(\mathsf{DirectNonVirtual}, B_i)$ for the direct non-virtual base B_i, and, for each data member M_i, $\kappa.\mathit{inits}(M_i)$ is the initializer for M_i if M_i is scalar, or a collection of n initializers if M_i is a structure array field of n cells).

We briefly recall the execution scheme for a constructor:

1. if the constructor corresponds to a most-derived object, construct the virtual bases
2. construct the direct non-virtual bases
3. set the dynamic type
4. construct the fields
5. execute the constructor body

We describe the compilation of the constructor starting from the fields up to the construction of virtual bases.

11.5.9.1 Fields

We define $\mathsf{compileConstrFields}(D, \kappa, L, \Gamma)$ to compile the construction of the list of fields L of a class D, assuming that a pointer to the current object being constructed is stored in the compiler-purpose variable $\Gamma.\mathsf{curobj}$. Once all fields are constructed, then the body of the constructor is run. It is compiled with the initial compilation context, $\Gamma_\circ$, as compiler-purpose variables are no longer useful in or after the constructor body.

$$\begin{array}{rcl} \mathsf{compileConstrFields}(D, \kappa, \epsilon, \Gamma) & \overset{=}{\underset{\text{def.}}{=}} & [\![\kappa.\mathit{body}; \mathtt{exit}\ 1]\!]^{\Gamma_\circ} \\ \mathsf{compileConstrFields}(D, \kappa, M :: L, \Gamma) & \overset{=}{\underset{\text{def.}}{=}} & \mathsf{compileConstrField}(D, \kappa, M, \Gamma); \\ & & \mathsf{compileConstrFields}(D, \kappa, L, \Gamma) \end{array}$$

where, for any data member M, $\mathsf{compileConstrField}(D, \kappa, M, \Gamma)$ compiles the construction of a field M defined in class D, using the initializers of the constructor κ, and assuming that $\Gamma.\mathsf{curobj}$ holds a pointer to the object being constructed.

Thanks to this definition, a $\mathsf{Kconstrother}(\ldots, \mathsf{Field}, L, \ldots)$ continuation stack frame in κ++ will be matched with a Block frame where the next statement to execute is $\mathsf{compileConstrFields}(\ldots, L, \ldots)$.

Scalar fields Assume M is a scalar data member. Then, its initializer is compiled as follows:

$$\mathsf{compileConstrField}(D, \kappa, M, \Gamma) \overset{=}{\underset{\text{def.}}{=}} \{[\![\kappa.\mathit{inits}(M); \mathtt{exit}\ 1]\!]^{\Gamma'}\}$$

with Γ' being the compilation context updated as follows:

$$\Gamma' = \Gamma[\mathsf{curfield} \leftarrow (D, M)]$$

to reflect that the implicit field to initialize through $\mathsf{initScalar}$ is actually the field M defined in class D.

Indeed, the initializer has to terminate its execution with a $\mathsf{initScalar}$ statement to give the field its initial value. Thus, this statement is compiled into a mere field write followed by leaving the block in which the initializer is to be embedded:

$$\begin{array}{rcl} [\![\mathsf{initScalar}(x)]\!]^{\Gamma} & \overset{=}{\underset{\text{def.}}{=}} & \underline{\Gamma.\mathsf{curobj}}\text{->}_D M := \overline{x}; \\ (\Gamma.\mathsf{curfield} = (D, M)) & & \mathtt{exit}\ 1 \end{array}$$

Structure fields Assume M is a structure data member of n cells of type C. Then, we need to construct a structure array, so we need further compiler-purpose variables to store a pointer to the first cell of the array. Let $\mathit{newobj} \notin \Gamma.\mathsf{Used}$ be such a compiler-purpose variable. Then, we have:

$$\begin{array}{rcl}\mathsf{compileConstrField}(D, \kappa, M, \Gamma) & \underset{\text{def.}}{=\!=} & \underline{\mathit{newobj}} := \underline{\Gamma.\mathsf{curobj}}\text{->}_D M; \\ & & \{\mathsf{compileConstrArray}(C, n, 0, \kappa.\mathit{inits}(M), \mathit{newobj}, \Gamma')\}\end{array}$$

with Γ' being the compilation context updated as follows:

$$\Gamma' = \Gamma[\mathsf{Used} \leftarrow \Gamma.\mathsf{Used} \uplus \{\mathit{newobj}\}]$$

to prevent newobj from reuse. However, $\Gamma.\mathsf{curfield}$ needs no update: indeed, $\Gamma.\mathsf{curfield}$ is used only for compiling $\mathsf{initScalar}$, which does not apply to structure fields.

11.5.9.2 Bases: where the "set dynamic type" operation comes into play

Consider a β base C of D, that is a direct non-virtual base of D ($\beta = \mathsf{DirectNonVirtual}$), or a direct or indirect virtual base of D ($\beta = \mathsf{Virtual}$) if constructing a most-derived object. Then, its initializer has to call the constructor for a base class subobject of C with a *this* argument adjusted to the base class subobject. Thus, we need a further compiler-purpose variable, say $\mathit{newobj} \notin \Gamma.\mathsf{Used}$.

To perform this adjustments, we cannot use a static cast: indeed, the generalized dynamic type of the object being constructed is not defined yet, so it would invalidate the guard condition for Ds++ static cast, see (Ds++-statcast, p. 251). So, we instead use the $\mathtt{base\_cast}\langle\beta, C\rangle_D$ operator, which we specifically designed to this purpose.

We also need to pass a further argument to the constructor, to indicate the construction path. So we also need a further compiler-purpose variable, say $\mathit{newpath} \notin \Gamma.\mathsf{Used}$. The construction path will be obtained by an adjustment of the path contained in $\Gamma.\mathsf{curpath}$, using the specific `basePath` operator.

As usual, the initializer for a base class subobject is to be embedded in a block with no stack object, so as to match Kconstrother continuation stack frames in κ++ with Block frames in Ds++.

Then we define $\mathsf{compileConstrBase}(D, \kappa, \beta, C, \Gamma)$ to compile the construction of a base class subobject B of D:

$$\begin{array}{rcl}\mathsf{compileConstrBase}(D, \kappa, \beta, C, \Gamma) & \underset{\text{def.}}{=\!=} & \underline{\mathit{newobj}} := \mathtt{base\_cast}\langle\beta, C\rangle_D(\underline{\Gamma.\mathsf{curobj}}); \\ & & \underline{\mathit{newpath}} := \mathtt{basePath}(\underline{\Gamma.\mathsf{curpath}}, D, \beta, C); \\ & & \{[\![\kappa.\mathit{inits}(\beta, C); \mathtt{exit}\ 1]\!]^{\Gamma'}\}\end{array}$$

The initializer is compiled using a compilation context Γ' updated from Γ as follows:

$$\begin{array}{rcll}\Gamma' & = & \Gamma & [\mathsf{Used} \leftarrow \Gamma.\mathsf{Used} \uplus \{\mathit{newobj}\} \uplus \{\mathit{newpath}\}] \\ & & & [\mathsf{curobj} \leftarrow \mathit{newobj}] \\ & & & [\mathsf{curpath} \leftarrow \mathit{newpath}] \\ & & & [\mathsf{Blocks} \leftarrow \epsilon]\end{array}$$

to reflect that the implicit object on which to call the constructor must be the base class subobject being constructed. The stack of blocks, however, is emptied, because the blocks to be opened within the initializer are irrelevant to the blocks enclosing the initializer.

Direct non-virtual bases Then, we use it to compile a list L of direct non-virtual bases of D.

When all direct non-virtual bases are ($L = \epsilon$), then, **before constructing the fields of D, the "set dynamic type" operation must be performed** on the current object being constructed ($\Gamma.\mathsf{curobj}$), with the current provided construction path ($\Gamma.\mathsf{curpath}$).

Recall that this operation has to be flagged with a boolean b, true only if we are constructing a most-derived D object or if D has no virtual bases. This flag is intended to optimize the Ds++-to-CVcm compilation of "set dynamic type". To exploit this opportunity, we need to know whether we are in the constructor for a most-derived object, which is given by $\Gamma.\mathsf{isMostDerived}$. So, the natural choice for the flag b follows:

$$b = \Gamma.\mathsf{isMostDerived} \vee (\mathcal{V}(D) = \epsilon)$$

Then we define $\mathsf{compileConstrDNVBases}(D, \kappa, L, \Gamma)$ to compile the construction of the list L of direct non-virtual base subobjects of D:

$$\begin{aligned}
\mathsf{compileConstrDNVBases}(D, \kappa, \epsilon, \Gamma) \;&\underset{\text{def.}}{=\!=}\; \texttt{setDynType}(\overline{\Gamma.\mathsf{curobj}}, \overline{\Gamma.\mathsf{curpath}})^{b}_{D};\\
&\qquad \mathsf{compileConstrFields}(D, \kappa, \mathcal{F}_D, \Gamma)\\
\mathsf{compileConstrDNVBases}(D, \kappa, B :: L, \Gamma) \;&\underset{\text{def.}}{=\!=}\; \mathsf{compileConstrBase}(D, \kappa, \mathsf{DirectNonVirtual}, B, \Gamma);\\
&\qquad \mathsf{compileConstrDNVBases}(D, \kappa, L, \Gamma)
\end{aligned}$$

Virtual bases We define $\mathsf{compileConstrVBases}(D, \kappa, L, \Gamma)$ to compile the construction of the list L of direct or indirect virtual base subobjects of D. These operations only occurs in the constructor for a most-derived object.

When all virtual bases are constructed, then the construction of the direct non-virtual bases of D starts. However, there is a κ++ execution step in between, doing nothing else than changing the execution point. To reflect this no-operation step, we have to insert a `skip`.

$$\begin{aligned}
\mathsf{compileConstrVBases}(D, \kappa, \epsilon, \Gamma) \;&\underset{\text{def.}}{=\!=}\; \texttt{skip};\\
&\qquad \mathsf{compileConstrDNVBases}(D, \kappa, \mathcal{DNV}_D, \Gamma)\\
\mathsf{compileConstrVBases}(D, \kappa, V :: L, \Gamma) \;&\underset{\text{def.}}{=\!=}\; \mathsf{compileConstrBase}(D, \kappa, \mathsf{Virtual}, V, \Gamma);\\
&\qquad \mathsf{compileConstrVBases}(D, \kappa, L, \Gamma)
\end{aligned}$$

11.5.9.3 Summary

Finally, the constructor is compiled into two static functions.

For a most-derived object, the construction of virtual bases is requested, using the *inheritance graph order* for virtual bases (Hypothesis 9.4.2 p. 190, *Definition* 10.3.4 p. 214):

$$\begin{aligned}
\underline{\kappa_{\mathsf{true}}}(\underline{\mathit{curobj}}, \underline{\mathit{curpath}}, \overline{\kappa.\mathit{arg}_1}, \ldots, \overline{\kappa.\mathit{arg}_a}) \;\; \{ \;\;& \\
& \overline{\kappa.\mathit{this}} = \underline{\mathit{curobj}};\\
& \mathsf{compileConstrVBases}(D, \kappa, \mathcal{VO}(D), \Gamma_{\mathsf{true}})\\
\}\;\;&
\end{aligned}$$

For a base class subobject, the construction of direct non-virtual bases is requested:

$$\underline{\kappa_{\mathsf{false}}}(\underline{curobj}, \underline{curpath}, \overline{\kappa.arg_1}, \ldots, \overline{\kappa.arg_a}) \quad \{$$
$$\overline{\kappa.this} = \underline{curobj};$$
$$\texttt{skip};$$
$$\mathsf{compileConstrDNVBases}(D, \kappa, \mathcal{DNV}_D, \Gamma_{\mathsf{false}})$$
$$\}$$

The compilation contexts Γ_{true} and Γ_{false} used for compiling construction operations is obtained from the initial compilation context $\Gamma_\circ$ as follows:

$$\begin{array}{rcl} \Gamma_b & = & \Gamma_\circ \quad [\mathsf{Used} \leftarrow \{curobj, curpath\}] \\ & & [\mathsf{curobj} \leftarrow curobj] \\ & & [\mathsf{curpath} \leftarrow curpath] \\ & & [\mathsf{isMostDerived} \leftarrow b] \end{array}$$

The additional `skip` prepended before the construction of direct non-virtual bases allows enforcing the:

Lemma 11.5.3. *If D has no virtual bases, then, for any constructor κ of D, the static functions $\underline{\kappa_{\mathsf{true}}}$ and $\underline{\kappa_{\mathsf{false}}}$ generated as the compilation of κ have the same code. Thus, they can be shared by identifying their names:*

$$\kappa_{\mathsf{true}} = \kappa_{\mathsf{false}}$$

Proof. We know that, if D has no virtual bases, $\mathcal{VO}(D) = \epsilon$ and:

$$\mathsf{compileConstrVBases}(D, \kappa, \mathcal{VO}(D), \Gamma_{\mathsf{false}}) = \texttt{skip}; \mathsf{compileConstrDNVBases}(D, \kappa, \mathcal{DNV}_D, \Gamma_{\mathsf{false}})$$

because of the no-operation step in κ++. Hence the additional `skip` in κ_{false}.

Moreover, compiling construction operations with Γ_{true} instead of Γ_{false} only changes the isMostDerived parameter, which is only used for determining the flag on `setDynType` (indeed, the choice of constructors for the bases and data members does not depend on this parameter). Then, the chosen flags on `setDynType` actually give

$$\forall b : \ (\Gamma_b.\mathsf{isMostDerived} \vee (\mathcal{V}(D) = \epsilon)) \ = \ \mathsf{true}$$

which concludes. □

11.5.10 Destructors

Let D be a class, then D has exactly one destructor ~D. Then, it will be compiled into two static functions, $\underline{\sim D_{\mathsf{true}}}$ corresponding to the destructor for a most-derived object, and $\underline{\sim D_{\mathsf{false}}}$ for a base class subobject. Let ~$D.this$ be its *this* argument, and ~$D.body$ be its body.

We briefly recall the compilation scheme for a destructor, which is the exact reverse of the constructor *except* for the position of the "set dynamic type" operation:

1. set the dynamic type, except for a most-derived object
2. execute the body
3. destruct the fields in reverse order

4. destruct the direct non-virtual bases in reverse order
5. if the constructor corresponds to a most-derived object, destruct the virtual bases in reverse order

The compilation of destructors is much simpler, as there is no counterpart to the initializers.

The body of the destructor is not compiled with the initial compilation context, but with ad-hoc compilation contexts:

$$\begin{array}{rcll} \Gamma & = & \Gamma_\circ & [\mathsf{Used} \leftarrow \{\mathit{curobj}, \mathit{curpath}\}] \\ & & & [\mathsf{curobj} \leftarrow \mathit{curobj}] \\ & & & [\mathsf{curpath} \leftarrow \mathit{curpath}] \\ & & & [\mathsf{isDestructorBody} \leftarrow \mathsf{true}] \\ \Gamma_b & = & \Gamma & [\mathsf{isMostDerived} \leftarrow b] \end{array}$$

to allow inlining the destructor body, and to prevent the compiler-purpose variables from reuse by the destructor body, as they are needed to destruct the bases and members.

While the destructor body is inlined, it is also embedded in a block with no stack object, so that the `return` statement is compiled into an appropriate `exit` statement, and the Kdestr continuation stack frame, indicating that the fields must be destructed first, corresponds to a Block frame.

Assume compileDestrFields(D, L, Γ) destructs the list L of fields of D as well as all base class subobjects of D. Then, for a base class subobject, the “set dynamic type” operation is requested, with the flag set to false:

$$\begin{array}{ll} \underline{\sim D_{\mathsf{false}}}(\underline{\mathit{curobj}}, \underline{\mathit{curpath}}) & \{ \\ & \quad \overline{\sim D.\mathit{this}} = \underline{\mathit{curobj}}; \\ & \quad \mathtt{setDynType}(\underline{\mathit{curobj}}, \underline{\mathit{curpath}})^{\mathsf{false}}_{D}; \\ & \quad \{[\![\sim D.\mathit{body}; \mathtt{exit}\ 1]\!]^{\Gamma}\}; \\ & \quad \mathsf{compileDestrFields}(D, \mathsf{rev}(\mathcal{F}_D), \Gamma_{\mathsf{false}}) \\ & \} \end{array}$$

For a most-derived object, however, the “set dynamic type” operation is omitted:

$$\begin{array}{ll} \underline{\sim D_{\mathsf{true}}}(\underline{\mathit{curobj}}, \underline{\mathit{curpath}}) & \{ \\ & \quad \overline{\sim D.\mathit{this}} = \underline{\mathit{curobj}}; \\ & \quad \{[\![\sim D.\mathit{body}; \mathtt{exit}\ 1]\!]^{\Gamma}\}; \\ & \quad \mathsf{compileDestrFields}(D, \mathsf{rev}(\mathcal{F}_D), \Gamma_{\mathsf{true}}) \\ & \} \end{array}$$

Γ_b.isMostDerived is not used in the compilation of the destructor body. As we shall see later, it will be used only to determine whether to destruct virtual bases.

Contrary to constructors, we perform here no optimization on code size and let the destructor code be duplicated. However, $\underline{\sim D_{\mathsf{true}}}$ could be optimized away by only using $\underline{\sim D_{\mathsf{false}}}$ in case D were a non-dynamic class: not only would D have no virtual bases (which would ensure that the behaviour of the compiled destructor would not depend on Γ_b.isMostDerived), but no virtual functions either. In this case, the `setDynType` operator added in Ds++ wouild compile to a no-op in CVcm.

Fields We define $\mathsf{compileDestrFields}(D, L, \Gamma)$ to compile the destruction of the list L of fields of D as follows:

$$\begin{array}{rcl} \mathsf{compileDestrFields}(D, \epsilon, \Gamma) & \underset{\text{def.}}{=\!=} & \texttt{skip}; \\ & & \mathsf{compileDestrDNVBases}(D, \mathsf{rev}(\mathcal{DNV}_D), \Gamma) \\ \mathsf{compileDestrFields}(D, M :: L, \Gamma) & \underset{\text{def.}}{=\!=} & \mathsf{compileDestrField}(D, M, \Gamma); \\ & & \mathsf{compileDestrFields}(D, L, \Gamma) \end{array}$$

where compileDestrDNVBases compiles the destruction of direct non-virtual base class subobjects, in reverse declaration order, once there are no fields left to destruct (a `skip` is necessary to match the execution steps); and $\mathsf{compileDestrField}(D, M, \Gamma)$ destructs field M.

Destructing a scalar field erases its value in κ++, but this is not necessary in Ds++. So, nothing has to be done, except a `skip` to make the execution step match.

$$\mathsf{compileDestrField}(D, M, \Gamma) = \texttt{skip}$$

Destructing a structure array field of n cells of type C destructs the array starting from the last cell $n - 1$. The first cell of the array has to be accessed, yielding a pointer, which must be stored in a compiler-purpose variable, say $newobj \notin \Gamma.\mathsf{Used}$.

$$\begin{array}{rcl} \mathsf{compileDestrField}(D, M, \Gamma) & \underset{\text{def.}}{=\!=} & \underline{newobj} := \underline{\Gamma.\mathsf{curobj}}\text{->}_D M; \\ & & \{\mathsf{compileDestrArray}(C, n-1, newobj, \Gamma')\} \end{array}$$

with Γ' being the compilation context updated as follows:

$$\Gamma' = \Gamma[\mathsf{Used} \leftarrow \Gamma.\mathsf{Used} \uplus \{newobj\}]$$

to prevent $newobj$ from reuse.

Non-virtual bases We define $\mathsf{compileDestrDNVBases}(D, L, \Gamma)$ to compile the destruction of the list L of direct non-virtual bases of D.

Once all direct non-virtual bases are destructed, it must be decided whether virtual bases are to be destructed. The answer lies in the $\Gamma.\mathsf{isMostDerived}$ parameter. If it is necessary, then this further stage is performed through compileDestrVBases, after a `skip` to match the execution steps. Otherwise, there is nothing left to destruct, so the destructor may be exited.

$$\begin{array}{rcl} \mathsf{compileDestrDNVBases}(D, \epsilon, \Gamma) & \underset{\text{def.}}{=\!=} & \texttt{skip}; \\ (\Gamma.\mathsf{isMostDerived} = \mathsf{true}) & & \mathsf{compileDestrVBases}(D, \mathsf{rev}(\mathcal{V}_D), \Gamma) \\ \mathsf{compileDestrDNVBases}(D, \epsilon, \Gamma) & \underset{\text{def.}}{=\!=} & \texttt{return} \\ (\Gamma.\mathsf{isMostDerived} = \mathsf{false}) & & \\ \mathsf{compileDestrDNVBases}(D, B :: L, \Gamma) & \underset{\text{def.}}{=\!=} & \mathsf{compileDestrBase}(D, \mathsf{DirectNonVirtual}, B, \Gamma); \\ & & \mathsf{compileDestrDNVBases}(D, L, \Gamma) \end{array}$$

where $\mathsf{compileDestrBase}(D, \beta, B, \Gamma)$ compiles the destruction of a $\beta \in \{\mathsf{DirectNonVirtual}, \mathsf{Virtual}\}$ base class subobject, as described below: the destructor of B corresponding to a base class subobject is called with a *this* argument adjusted to the base class subobject (through $\texttt{base\_cast}\langle\beta, B\rangle_D$ operator instead of static cast, for similar reasons as for construction) and with the constructor path adjusted to the path to the base.

$$\begin{array}{rcl} \mathsf{compileDestrBase}(D, \beta, B, \Gamma) & \underset{\text{def.}}{=\!=} & \underline{newobj} := \texttt{base\_cast}\langle\beta, B\rangle_D(\underline{\Gamma.\mathsf{curobj}}); \\ (\{newobj\} \uplus \{newpath\} \ \# \ \Gamma.\mathsf{Used}) & & \underline{newpath} := \texttt{basePath}(\underline{\Gamma.\mathsf{curpath}}, D, \beta, B); \\ & & \underline{\texttt{\~{}}C_{\mathsf{false}}}(\underline{newobj}, \underline{newpath}) \end{array}$$

Virtual bases We define $\mathsf{compileDestrVBases}(D, L, \Gamma)$ to compile the destruction of the list L of direct or indirect virtual bases of D, for a most-derived object.

Once all virtual bases are destructed, there is nothing left to destruct, so the destructor may be exited.

$$\begin{aligned}
\mathsf{compileDestrVBases}(D, \epsilon, \Gamma) &\underset{\text{def.}}{=\!=} \texttt{return} \\
\mathsf{compileDestrVBases}(D, B :: L, \Gamma) &\underset{\text{def.}}{=\!=} \mathsf{compileDestrBase}(D, \mathsf{Virtual}, B, \Gamma); \\
&\qquad \mathsf{compileDestrVBases}(D, L, \Gamma)
\end{aligned}$$

11.6 Correctness of the κ++-to-Ds++ compiler

We prove the correctness of the compiler by forward simulation (THEOREM B.1 p. 340): we show that an invariant $s \rhd s'$ is preserved between an execution state s of κ++ and an execution state s' of Ds++, finally obtaining THEOREM II.16 (p. 284).

Let $s = (S, \mathcal{K}, \mathcal{G})$ be a κ++ execution state, and $s' = (st', stl', e', \mathcal{K}', \mathcal{G}')$ be a Ds++ state.

INVARIANT 11.6.1 (Invariant). *The invariant is split into three parts:*

- *The execution invariant of κ++ stated in Section 10.1 (p. 199) holds on s.*
- *An invariant $\mathcal{G} \rhd_{\mathsf{global}} \mathcal{G}'$ relates the global states.*
- *An invariant $s \rhd_{\mathsf{exec}} s'$ relates the execution points and the continuation stacks.*

11.6.1 Global states

In this section, we describe the components of the invariant $\mathcal{G} \rhd_{\mathsf{global}} \mathcal{G}'$ relating the global states.

11.6.1.1 Object locations

INVARIANT 11.6.2. *Let ℓ be an object location. Then, either ℓ corresponds to an object deallocated in κ++, or the κ++ and Ds++ object stores agree on ℓ:*

$$\ell \in \mathcal{G}.\mathsf{dealloc} \vee \mathcal{G}.\mathsf{LocType}(\ell) = \mathcal{G}'.\mathsf{LocType}(\ell)$$

This lemma allows to really remove ℓ from $\mathcal{G}'.\mathsf{LocType}$ once ℓ becomes deallocated in κ++. Indeed, we know that ℓ is never reused in κ++ more than once; moreover, whenever ℓ is deallocated in κ++, it will stay deallocated forever.

An important consequence of this invariant is the:

LEMMA 11.6.1. *If $(\ell, (\alpha, i, \sigma))$ is a subobject of ℓ that is not* $\mathsf{Destructed}$*, then ℓ is a valid Ds++ object, and we have $\mathcal{G}.\mathsf{LocType}(\ell) = \mathcal{G}'.\mathsf{LocType}(\ell)$.*

Proof. If ℓ were deallocated in κ++, then, the κ++ run-time invariant would require all subobjects of ℓ to be $\mathsf{Destructed}$. □

Conversely, what actually allows to make non-deallocated objects exactly match in spite of removing deallocated objects from the Ds++ store, is the following:

INVARIANT 11.6.3. *The location of the next object to be allocated is the same in the two languages:*

$$\mathcal{G}.\ell_{\mathsf{next}} = \mathcal{G}'.\ell_{\mathsf{next}}$$

11.6.1.2 Generalized dynamic type

The following invariant allows proving the correctness of the compilation of polymorphic operations (virtual function call, dynamic cast):

Polymorphic operations

INVARIANT **11.6.4.** *Let $\pi = (\ell, (\alpha, i, \sigma))$ be a pointer to a subobject. If the generalized dynamic type of π in κ++ is well-defined, then it is also defined in Ds++ and they are the same:*

$$(\mathcal{G} \vdash \mathsf{gDynType}(\ell, \alpha, i, \sigma, \sigma_\circ, \sigma_1)) \Rightarrow \mathcal{G}'.\mathsf{gDynType}(\ell, (\alpha, i, \sigma)) = (\sigma_\circ, \sigma_1)$$

The preservation of this invariant is done thanks to the corresponding theorem (cf. Construction).

Static cast However, a further invariant about the dynamic types is required to prove the preservation of the semantics of static cast:

INVARIANT **11.6.5.** *If an object is* Constructed*, then its generalized dynamic type exists in Ds++:*

$$\mathcal{G}.\mathsf{ConstrState}(\pi) = \mathsf{Constructed} \Rightarrow \mathcal{G}'.\mathsf{gDynType}(\pi) \neq \bot$$

This invariant allows showing that, during the whole lifetime of a subobject (even during the construction or destruction of one of its siblings), it is possible to access its virtual bases: the guard condition of the `static_cast`$\langle\cdot\rangle$. operator holds.

Proof. The preservation of this invariant is done thanks to the κ++ run-time invariant on the relations between object construction states. Let $s_1 \to s_2$ be a κ++ transition step. Then, there are several cases:

- If no construction state changes, or if an object changes its construction state to StartedConstructing, DestructingBases or Destructed, then neither the κ++ nor the Ds++ generalized dynamic types change. Moreover, the Constructed objects are the same before (at s) as after (at s_2) the execution step $s_1 \to s_2$, so the invariant trivially keeps holding.
- Assume an object $\tilde{\pi}$ changes its construction state to Constructed. Then, the Ds++ generalized dynamic types do not change. Let π a Constructed object in s_2. Then, there are two cases:
  - If $\pi = \tilde{\pi}$, then π was BasesConstructed in s_1, so its generalized dynamic type was defined in κ++, thus per INVARIANT 11.6.4 (p. 272) in Ds++ as well. The latter does not change.
  - Otherwise, INVARIANT 11.6.5 (p. 272) ensures the Ds++ generalized dynamic type of π is defined before the step. The latter does not change.
- If an object $\tilde{\pi}$ changes its construction state to BasesConstructed or StartedDestructing, then the generalized dynamic types change: they become defined in Ds++ for every base class subobject of $\tilde{\pi}$. Let π be a Constructed object in s_2. Then, $\pi \neq \tilde{\pi}$ and there are three cases:
  - If neither one is a base class subobject of the other, then neither the construction state nor the Ds++ generalized dynamic type of π change. Thus, INVARIANT 11.6.5 (p. 272) keeps holding.

- Otherwise, if $\tilde{\pi}$ is a base class subobject of π, then, as π is $\mathsf{Constructed}$, $\tilde{\pi}$ is necessarily $\mathsf{Constructed}$, which is absurd. In particular, this allows to eliminate the case where $\tilde{\pi}$ is a primary base of π and the Ds++ generalized dynamic type of π is erased.
- Otherwise, π is a base class subobject of $\tilde{\pi}$. As the Ds++ generalized dynamic types of every base class subobject of $\tilde{\pi}$ become defined, this concludes. $\square$

11.6.1.3 Scalar data members

A consequence of the κ++ invariant is that, if a scalar data member has a value in κ++, then this member is $\mathsf{Constructed}$. Thus, the corresponding object is not deallocated in κ++, so it also exists in Ds++. Then, it is possible to reason about the value of the data member in this corresponding Ds++ object:

Invariant 11.6.6. *If a scalar data member of any object has a value in κ++, then in the corresponding Ds++ object, it has the same value:*

$$\mathcal{G}.\mathsf{FieldValue}(\pi, f) = v \neq \bot \Rightarrow \mathcal{G}'.\mathsf{FieldValue}(\pi, f) = v$$

11.6.2 ($\star$) Execution points and continuation stack frames unrelated to construction or destruction

In the following sections, we describe the remaining part of the compilation invariant, namely the components of the invariant $s \rhd_{\mathsf{exec}} s'$ relating the execution points. These sections are rather technical; on first reading, they may be skipped to the proof of forward simulation in Section 11.6.5 (p. 284).

11.6.2.1 Non-exiting statements

When a κ++ statement st is being executed, it corresponds to a Ds++ statement $[\![st]\!]^\Gamma$ for some compilation context Γ.

Then, the pipeline of further κ++ statements to be executed also correspond to their Ds++ counterparts compiled with the same compilation context Γ. However, due to the "automatic block exit", a further `exit` 1 has to be present at the end of the Ds++ statement sequence, so that it is peeked from the Ds++ pipeline when applying the κ++ automatic block exit. This explains why function, constructor and destructor bodies are compiled with a further `exit` 1.

In κ++, the continuation stack $\mathcal{K}$ and the enclosing blocks $\mathcal{B}$ are separated, whereas in Ds++ information about enclosing blocks is included in the continuation stack. So, the latter is divided into two parts: the top part related to the blocks of the current κ++ execution point, and the bottom part related to the κ++ continuation stack.

So, assume that st is not an exiting statement (that is, neither `exit` n nor `return` $x^?$). Then, no block is exited in κ++ and not in Ds++, and the invariant relating the execution points and the continuation stacks can be written as follows:

$$\frac{\begin{array}{c}\forall i, st \neq \mathtt{exit}\ i \qquad \forall x^?, S \neq \mathtt{return}\ x^? \\ \Gamma.\mathsf{furtherBlocks} = 0 \qquad e \rhd_{\mathsf{Var}} e' \qquad \mathcal{G}, e';\ \Gamma \downarrow \Gamma_1 \vdash \mathcal{B} \rhd_{\mathsf{Block}} \mathcal{K_B}' \qquad \mathcal{G}; \Gamma_1 \vdash (e, \mathcal{K}) \rhd_{\mathsf{Stack}} (e', \mathcal{K}')\end{array}}{(\mathsf{Codepoint}(st, stl, e, \mathcal{B}), \mathcal{K}, \mathcal{G}) \rhd_{\mathsf{exec}} ([\![st]\!]^\Gamma, \mathsf{map}[[\![\cdot]\!]^\Gamma](stl +\!\!+ \mathtt{exit}\ 1 :: \epsilon), e', \mathcal{K_B}' +\!\!+ \mathcal{K}', \mathcal{G}')}$$

where:

- $\triangleright_{\mathsf{Var}}$ relates the environments;
- $\mathcal{G}, e';\ \Gamma \downarrow \Gamma_1 \vdash \cdot \triangleright_{\mathsf{Block}} \cdot$ relates the list of κ++ enclosing blocks with the corresponding top part of the Ds++ continuation stack;
- $\mathcal{G}; \Gamma_1^? \vdash (e^?, \mathcal{K}) \triangleright_{\mathsf{Stack}} (e'^?, \mathcal{K}')$ relates the κ++ continuation stack with the bottom part of the Ds++ continuation stack

11.6.2.2 Environments

If a variable x has a value in κ++, then its Ds++ counterpart $\overline{x}$ has the same value.

$$\frac{\forall x, v : e(x) = v \neq \bot \Rightarrow e'(\overline{x}) = v}{e \triangleright_{\mathsf{Var}} e'}$$

11.6.2.3 Blocks

$\mathcal{G}, e';\ \Gamma \downarrow \Gamma_1 \vdash \cdot \triangleright_{\mathsf{Block}} \cdot$ relates the list of κ++ enclosing blocks to the top-part of the Ds++ continuation stack, such that Γ_1 is the compilation context once the most enclosing block has been reached. There are three cases:

- If there is no enclosing block, then the corresponding part of the Ds++ continuation frame is empty, and the compilation context does not change.

$$\frac{}{\mathcal{G}, e';\ \Gamma \downarrow \Gamma \vdash \epsilon \triangleright_{\mathsf{Block}} \epsilon}$$

- Otherwise, if there is an enclosing block without stack object, then it must correspond to a Ds++ block without stack object as well, and the statements must match:

$$\frac{\Gamma = \Gamma_1[\mathsf{Blocks} \leftarrow \bot :: \Gamma_1.\mathsf{Blocks}] \qquad \mathcal{G}, e';\ \Gamma_1 \downarrow \Gamma_2 \vdash \mathcal{B} \triangleright_{\mathsf{Block}} \mathcal{K_B}'}{\mathcal{G}, e';\ \Gamma \downarrow \Gamma_2 \vdash (\bot, \mathit{stl}) :: \mathcal{B} \triangleright_{\mathsf{Block}} \mathsf{Block}(\bot, \mathsf{map}[\![\cdot]\!]^{\Gamma_1}](\mathit{stl} +\!\!+ \mathtt{exit}\ 1 :: \epsilon)) :: \mathcal{K_B}'}$$

- Otherwise, there must be a Ds++ compiler-purpose variable corresponding to the object attached to the enclosing block. This variable must contain a pointer to the first cell of the object, and the type and number of cells of the object must match the ones expected during compilation. This variable will be actually used for the destruction of this block object.

$$\frac{\begin{array}{c}\Gamma = \Gamma_1[\mathsf{Blocks} \leftarrow (x, \Gamma_1.\mathsf{Used}, (C, n)) :: \Gamma_1.\mathsf{Blocks}][\mathsf{Used} \leftarrow \Gamma_1.\mathsf{Used} \uplus \{x\}] \\ e'(\underline{x}) = (\ell, (\epsilon, 0, (\mathsf{Repeated}, C :: \epsilon))) \qquad \mathcal{G} \vdash \langle \ell \rangle\ C[n] \qquad \mathcal{G}, e';\ \Gamma_1 \downarrow \Gamma_2 \vdash \mathcal{B} \triangleright_{\mathsf{Block}} \mathcal{K_B}'\end{array}}{\mathcal{G}, e';\ \Gamma \downarrow \Gamma_2 \vdash (\ell, L) :: \mathcal{B} \triangleright_{\mathsf{Block}} (\mathsf{Block}(\ell, \mathsf{map}[\![\cdot]\!]^{\Gamma_1}](L +\!\!+ \mathtt{exit}\ 1 :: \epsilon)) :: \mathcal{K_B})'}$$

11.6.2.4 Continuation stack

Once blocks have been treated, the κ++ continuation stack can be confronted with the remaining bottom part of the Ds++ continuation stack. We shall see later why the environments may be actually needed to help this invariant hold.

Empty stacks match; the environments are irrelevant, as they are assumed to be already constrained within $\triangleright_{\mathsf{exec}}$.

$$\frac{}{\mathcal{G}; \Gamma \vdash (e^?, \epsilon) \triangleright_{\mathsf{Stack}} (e'^?, \epsilon)}$$

When returning from a function, the provided environments are the environments of the callee, so they are irrelevant, as they are expected to be discarded once the function exits. The invariant is a simplified version of $\rhd_{\mathsf{exec}}$ on $\mathsf{Codepoint}$: the environments of the caller must match, the return value variables must match, further caller statements and statements of enclosing blocks must match the top part of the stack, but there are no blocks exited in κ++ and not in Ds++. Moreover, the callee must not be a destructor body (which is not a true function, as it is inlined into a block, so that `return` is compiled to `exit`):

$$\frac{\begin{array}{c}\Gamma.\mathsf{isDestructorBody} = \mathsf{false}\\ e_1 \rhd_{\mathsf{Var}} {e_1}' \qquad \Gamma_1.\mathsf{furtherBlocks} = 0 \qquad \mathit{stl}' = \mathsf{map}[\llbracket\cdot\rrbracket^{\Gamma_1}](\mathit{stl} \mathbin{+\!\!+} \texttt{exit}\ 1 :: \epsilon)\\ \mathcal{G}, {e_1}';\ \Gamma_1 \downarrow \Gamma_2 \vdash \mathcal{B} \rhd_{\mathsf{Block}} {\mathcal{K}_{\mathcal{B}}}' \qquad \mathcal{G}; \Gamma_2 \vdash (e_1, \mathcal{K}) \rhd_{\mathsf{Stack}} ({e_1}', \mathcal{K}')\end{array}}{\mathcal{G}; \Gamma \vdash (e^?, \mathsf{Kretcall}(x^?, \mathit{stl}, \mathcal{B}, e_1) :: \mathcal{K}) \rhd_{\mathsf{Stack}} (e'^?, \mathsf{Callframe}(\overline{x}^?, \mathit{stl}', {e_1}') :: {\mathcal{K}_{\mathcal{B}}}' \mathbin{+\!\!+} \mathcal{K}')}$$

Other possible continuation frames are related to construction and destruction. We shall see them later.

11.6.2.5 Leaving blocks

Recall that, leaving a block in κ++ does not immediately trigger leaving the corresponding Ds++ block. Following how `exit`s are compiled, objects associated to blocks requested to be left are first destructed, but without actually leaving any block in Ds++. Only once all requested blocks are left in κ++, i.e. when reaching `exit` 0, the block exits start in Ds++.

Remember that the number of blocks already exited in κ++ but not in Ds++ is given by $\Gamma.\mathsf{furtherBlocks}$. However, this data is not enough for our invariant: we must also prove that the objects attached to those exited blocks, which still exist in Ds++, have been actually destructed.

To this purpose, the top part of the Ds++ continuation stack, related to blocks, has to be itself split into two parts, a bottom part ${\mathcal{K}_{\mathcal{B}}}'$ related to the blocks not yet exited in κ++, and a top part of size $\Gamma.\mathsf{furtherBlocks}$ related to the blocks already exited. This part is written $\mathsf{map}[\mathsf{Block}](\mathit{Bll})$ to ensure that it contains only Block continuation frames, so that Bll is a list whose elements are of the form $(\ell^?, \mathit{st})$ where any defined ℓ is deallocated in κ++.

First consider a non-trivial exit statement (that is, `exit` $(\mathsf{S}\ n)$ or `return`). Then, the statement pipeline of the current execution point is pointless. The compilation of the statement pipelines of the enclosing blocks, however, will be executed only once all exited blocks in κ++ are also exited in Ds++, so they must be compiled with $\mathsf{furtherBlocks} = 0$:

$$\frac{\begin{array}{c}\mathit{st} = \texttt{exit}\ (\mathsf{S}\ n) \lor \mathit{st} = \texttt{return}\ x^?\\ e \rhd_{\mathsf{Var}} e' \qquad \forall \ell, {\mathit{stl}_\ell}' : (\ell, {\mathit{stl}_\ell}') \in \mathit{Bll} \Rightarrow \ell \in \mathcal{G}.\mathsf{dealloc} \qquad \mathsf{length}(\mathit{Bll}) = \Gamma.\mathsf{furtherBlocks}\\ \Gamma_1 = \Gamma[\mathsf{furtherBlocks} \leftarrow 0] \qquad \mathcal{G}, e';\ \Gamma_1 \downarrow \Gamma_2 \vdash \mathcal{B} \rhd_{\mathsf{Block}} {\mathcal{K}_{\mathcal{B}}}' \qquad \mathcal{G}; \Gamma_2 \vdash (e, \mathcal{K}) \rhd_{\mathsf{Stack}} (e', \mathcal{K}')\end{array}}{(\mathsf{Codepoint}(\mathit{st}, \mathit{stl}, e, \mathcal{B}), \mathcal{K}, \mathcal{G}) \rhd_{\mathsf{exec}} (\llbracket \mathit{st} \rrbracket^{\Gamma}, \mathit{stl}', e', \mathsf{map}[\mathsf{Block}](\mathit{Bll}) \mathbin{+\!\!+} {\mathcal{K}_{\mathcal{B}}}' \mathbin{+\!\!+} \mathcal{K}', \mathcal{G}')}$$

By contrast, in the case of `exit` 0, the κ++ statement pipeline has to be matched in Ds++. If no κ++ block has been exited yet, then the corresponding Ds++ pipeline is the current statement pipeline; otherwise, it is the statement pipeline of the deepest block in Bll, which is the most-enclosing exited block. So it is retrieved from the list Bll of blocks not yet exited from Ds++

by the following retrieveStmts function:

$$\begin{aligned}\mathsf{retrieveStmts}(stl', \epsilon) &\overset{}{\underset{\text{def.}}{=\!=}} stl' \\ \mathsf{retrieveStmts}(stl', (\ell^?, {stl_1}') :: Bll_1) &\underset{\text{def.}}{=\!=} \mathsf{retrieveStmts}({stl_1}', Bll_1)\end{aligned}$$

so that:

$$\frac{\begin{array}{c} e \rhd_{\mathsf{Var}} e' \qquad \forall \ell, stl_\ell : (\ell, stl_\ell) \in Bll \Rightarrow \ell \in \mathcal{G}.\mathsf{dealloc} \\ \mathsf{length}(Bll) = \Gamma.\mathsf{furtherBlocks} \qquad st = \mathtt{exit}\ 0 \qquad st' = [\![st]\!]^{\Gamma} \\ \Gamma_1 = \Gamma[\mathsf{furtherBlocks} \leftarrow 0] \qquad \mathsf{retrieveStmts}(stl', Bll) = \mathsf{map}[[\![\cdot]\!]^{\tilde{\Gamma}}](stl +\!\!+ \mathtt{exit}\ 1 :: \epsilon) \\ \mathcal{G}, e';\ \Gamma_1 \downarrow \Gamma_2 \vdash \mathcal{B} \rhd_{\mathsf{Block}} \mathcal{K_B}' \qquad \mathcal{G}; \Gamma_2 \vdash (e, \mathcal{K}) \rhd_{\mathsf{Stack}} (e', \mathcal{K}') \end{array}}{(\mathsf{Codepoint}(stmt, stl, e, \mathcal{B}), \mathcal{K}, \mathcal{G}) \rhd_{\mathsf{exec}} (st', stl', e', \mathcal{K_B}' +\!\!+ \mathcal{K}', \mathcal{G}')}$$

Actually, the latter rule also covers the case of other non-exit statements, for which there is no block exited in κ++ and not in Ds++: $Bll = \epsilon$. So, finally, the following unique rule summarizes the three cases:

$$\frac{\begin{array}{c} st = \mathtt{exit}\ n_1 \vee st = \mathtt{return}\ x^? \vee Bll = \epsilon \\ e \rhd_{\mathsf{Var}} e' \qquad \forall \ell, {stl_\ell}' : (\ell, {stl_\ell}') \in Bll \Rightarrow \ell \in \mathcal{G}.\mathsf{dealloc} \\ \mathsf{length}(p) = \Gamma.\mathsf{furtherBlocks} \qquad \Gamma_1 = \Gamma[\mathsf{furtherBlocks} \leftarrow 0] \\ st = \mathtt{exit}\ (\mathsf{S}\ n_2) \vee st = \mathtt{return}\ x^? \vee \mathsf{retrieveStmts}(stl', Bll) = \mathsf{map}[[\![\cdot]\!]^{\Gamma_1}](stl +\!\!+ \mathtt{exit}\ 1 :: \epsilon) \\ st' = [\![st]\!]^{\Gamma} \qquad \mathcal{G}, e';\ \Gamma_1 \downarrow \Gamma_2 \vdash \mathcal{B} \rhd_{\mathsf{Block}} \mathcal{K_B}' \qquad \mathcal{G}; \Gamma_2 \vdash (e, \mathcal{K}) \rhd_{\mathsf{Stack}} (e', \mathcal{K}') \end{array}}{(\mathsf{Codepoint}(st, stl, e, \mathcal{B}), \mathcal{K}, \mathcal{G}) \rhd_{\mathsf{exec}} (st', stl', e', \mathsf{map}[\mathsf{Block}](Bll) +\!\!+ \mathcal{K_B}' +\!\!+ \mathcal{K}', \mathcal{G}')}$$

Application: blocks with no object Now we sketch the proof of preservation of $\rhd_{\mathsf{exec}}$ for a block exit with no object. Consider the following $s_1 \to s_2$ step:

$$\begin{array}{rlll} & (\mathsf{Codepoint}(\mathtt{exit}\ (\mathsf{S}\ n), stl_1, e, (\bot, stl_2) :: \mathcal{B}), & \mathcal{K}, & \mathcal{G}) \\ \to & (\mathsf{Codepoint}(\mathtt{exit}\ n, stl_2, e, \mathcal{B}) \qquad\qquad\qquad\quad , & \mathcal{K}, & \mathcal{G}) \end{array}$$

Assume $s_1 \rhd_{\mathsf{exec}} {s_1}'$. Then, this invariant gives:

$${s_1}' = ([\![\mathtt{exit}\ (\mathsf{S}\ n)]\!]^{\Gamma_1}, {stl_1}', e', \mathsf{map}[\mathsf{Block}](Bll_1) +\!\!+ \mathsf{Block}(\bot, \mathsf{map}[[\![\cdot]\!]^{\tilde{\Gamma}}](stl_2 +\!\!+ \mathtt{exit}\ 1 :: \epsilon)) ::$$

$$\mathcal{K_B}' +\!\!+ \mathcal{K}', \mathcal{G}')$$

$$\Gamma_1.\mathsf{Blocks} = (blockvar, \Gamma_2.\mathsf{Used}, (C, n)) :: \Gamma_2.\mathsf{Blocks}$$

$$\tilde{\Gamma} = \Gamma_2[\mathsf{furtherBlocks} \leftarrow 0]$$

where Bll_1 is the list of blocks exited in κ++ but not in Ds++, so that $\Gamma.\mathsf{furtherBlocks} = \mathsf{length}(Bll_1)$. But then, we know that $[\![\mathtt{exit}\ (\mathsf{S}\ n)]\!]^{\Gamma} = \mathtt{skip}; [\![\mathtt{exit}\ n]\!]^{\Gamma_2}$ with $\Gamma_2.\mathsf{furtherBlocks} = \mathsf{S}(\Gamma_1.\mathsf{furtherBlocks})$. Actually, to show the preservation of $\rhd_{\mathsf{exec}}$, we perform:

$$Bll \leftarrow Bll_2 +\!\!+ (\bot, \mathsf{map}[[\![\cdot]\!]^{\tilde{\Gamma}}](stl_2 +\!\!+ \mathtt{exit}\ 1 :: \epsilon)) :: \epsilon$$

to record the κ++ block exit, so that $\Gamma_2.\mathsf{furtherBlocks} = \mathsf{length}(Bll_2)$. Then, we notice:

$$\begin{array}{l} \mathsf{map}[\mathsf{Block}](Bll_1) +\!\!+ \mathsf{Block}(\bot, \mathsf{map}[[\![\cdot]\!]^{\Gamma_1}](stl_2 +\!\!+ \mathtt{exit}\ 1 :: \epsilon)) :: \mathcal{K_B}' \\ = \mathsf{map}[\mathsf{Block}](Bll_1 +\!\!+ (\bot, \mathsf{map}[[\![\cdot]\!]^{\Gamma_1}](stl_2 +\!\!+ \mathtt{exit}\ 1 :: \epsilon)) :: \epsilon) +\!\!+ \mathcal{K_B}' \end{array}$$

meaning that the Ds++ continuation stack remains unchanged: the corresponding Ds++ steps actually trigger no Ds++ block exit. Finally, we have:

$$\begin{aligned}&(\llbracket \texttt{exit}\ (\mathsf{S}\ n)\rrbracket^{\Gamma_1}, stl_1', e', \mathsf{map}[\mathsf{Block}](Bll_1) + \mathsf{Block}(\bot, \mathsf{map}[\llbracket\cdot\rrbracket^{\Gamma_1}](stl_1 + \texttt{exit}\ 1 :: \epsilon)) :: \mathcal{K}_{\mathcal{B}}' + \mathcal{K}', \mathcal{G}')\\ &\xrightarrow{+}(\llbracket \texttt{exit}\ n\rrbracket^{\Gamma_2}, stl_1', e', \mathsf{map}[\mathsf{Block}](Bll_2) + \mathcal{K}_{\mathcal{B}}' + \mathcal{K}', \mathcal{G}')\end{aligned}$$

However, if all requested κ++ blocks have been exited (i.e. if $n = 0$), then the statement pipeline becomes significant again, so it remains to show that, if $n = 0$, then $\mathsf{retrieveStmts}(stl_1', Bll_2) = \mathsf{map}[\llbracket\cdot\rrbracket^{\tilde{\Gamma}}](stl_2 + \texttt{exit}\ 1 :: \epsilon)$. This is actually the case as $(\bot, \mathsf{map}[\llbracket\cdot\rrbracket^{\tilde{\Gamma}}](stl_2 + \texttt{exit}\ 1 :: \epsilon))$ is the last element of Bll_2.

Application: actual Ds++ block exit We shall see now when blocks are actually exited in Ds++. Consider the following $s_1 \rightarrow s_2$ step:

$$\begin{array}{rlll} & (\mathsf{Codepoint}(\texttt{exit}\ 0, stl, e, \mathcal{B}), & \mathcal{K}, & \mathcal{G}) \\ \rightarrow & (\mathsf{Codepoint}(\texttt{skip}, stl, e, \mathcal{B})\ \ , & \mathcal{K}, & \mathcal{G}) \end{array}$$

Assume $s \rhd_{\mathsf{exec}} s'$. Then, this invariant gives:

$$s_1' = (\llbracket \texttt{exit}\ 0\rrbracket^{\Gamma}, stl_1', e', \mathsf{map}[\mathsf{Block}](Bll) + \mathcal{K}_{\mathcal{B}}' + \mathcal{K}', \mathcal{G}')$$

where Bll is the list of blocks exited in κ++ but not in Ds++, so that $\Gamma.\mathsf{furtherBlocks} = \mathsf{length}(Bll)$ and $\mathsf{retrieveStmts}(stl_1', p) = \mathsf{map}[\llbracket\cdot\rrbracket^{\tilde{\Gamma}}](stl + \texttt{exit}\ 1 :: \epsilon)$ where $\tilde{\Gamma} = \Gamma_2[\mathsf{furtherBlocks} \leftarrow 0]$ (this is true as $\texttt{exit}\ 0$ is not a non-trivial exit statement). But then, we know that $\llbracket \texttt{exit}\ 0\rrbracket^{\Gamma} = \texttt{exit}\ \Gamma.\mathsf{furtherBlocks} = \texttt{exit}\ (\mathsf{length}(Bll))$. So we conclude by easily showing the:

Lemma 11.6.2 (Ds++ exit progress). *Let Bll be a list of Ds++ execution blocks. If* $\mathsf{map}[\mathsf{Block}](Bll)$ *describes the top of the continuation stack, then* $\texttt{exit}\ (\mathsf{length}(Bll))$ *succeeds:*

$$\exists \mathcal{G}_2 : \begin{array}{rllll} & (\texttt{exit}\ (\mathsf{length}(Bll)), & stl', & e', & \mathsf{map}[\mathsf{Block}](Bll) + \mathcal{K}', \ \mathcal{G}_1) \\ \rightarrow^+ & (\texttt{skip}\ \ , & \mathsf{retrieveStmts}(stl', Bll), & e', & \mathcal{K}', \ \mathcal{G}_2) \end{array}$$

The global state $\mathcal{G}_1$ changes to $\mathcal{G}_2$ where all objects attached to any block in Bll are deallocated:

$$\forall \ell : \mathcal{G}_1.\mathsf{LocType}(\ell) = \mathcal{G}_2.\mathsf{LocType}(\ell) \vee \exists stl_\ell', (\ell, stl_\ell') \in Bll$$

with other components of $\mathcal{G}_1$ being unchanged.

A similar reasoning scheme can be applied to show the preservation of $\rhd_{\mathsf{exec}}$ for `return`ing from a function.

11.6.3 (⋆) Construction

11.6.3.1 Continue after construction

When a block-scoped object is requested to be constructed, a Kcontinue stack frame is present in the κ++ stack. This frame records the body of the block, the statements of the immediately enclosing block, as well as the statements of the further enclosing blocks. However, the environment has to be explicitly provided.

The initializer for the array is compiled in an additional block, so that there are two successive corresponding Ds++ continuation frames, from top towards bottom:

- the first frame is a Block with no stack object, representing the block *enclosing* the initializer block; so the further statements are actually the body of the κ++ block
- the second frame is also a Block representing the block enclosing the κ++ block itself.

Then, a compiler-purpose variable, say *blockvar*, holds a pointer to the first cell of the array being constructed. This variable shall be used later to destruct the array. The type and number of cells of the array must match the ones expected by the compilation.

This rule involves four different compilation contexts:

- the provided Γ compilation context is the context of the initializer, where *blockvar* is defined
- the compilation context $\tilde{\Gamma}$ is used for compiling the block body, so it has to record a new enclosing block
- the compilation context Γ_1 is used for compiling the further enclosing blocks, so *blockvar* is no longer relevant
- the compilation context Γ_2 is obtained when reaching the most-enclosing block

$$\frac{\begin{array}{c}\Gamma = \Gamma_1[\mathsf{Used} \leftarrow \Gamma_1.\mathsf{Used} \uplus \{\mathit{blockvar}\}] \\ \mathcal{G} \vdash \langle \ell \rangle\ C[n] \qquad e'(\underline{\mathit{blockvar}}) = (\ell, \epsilon, 0, (\mathsf{Repeated}, C :: \epsilon)) \\ \tilde{\Gamma} = \Gamma[\mathsf{Blocks} \leftarrow (\mathit{blockvar}, \Gamma.\mathsf{Used}, (C, n)) :: \Gamma.\mathsf{Blocks}] \\ st' = [\![st]\!]^{\tilde{\Gamma}} \qquad stl' = \mathsf{map}[[\![\cdot]\!]^{\Gamma_1}](stl + \mathtt{exit}\ 1 :: \epsilon) \\ \mathcal{G}, e';\ \Gamma_1 \downarrow \Gamma_2 \vdash \mathcal{B} \rhd_{\mathsf{Block}} \mathcal{K_B}' \qquad \mathcal{G}; \Gamma_2 \vdash (e, \mathcal{K}) \rhd_{\mathsf{Stack}} (e', \mathcal{K}')\end{array}}{\mathcal{G}; \Gamma \vdash (e, \mathsf{Kcontinue}(\ell, st, stl, \mathcal{B}, \mathsf{Constr}) :: \mathcal{K}) \rhd_{\mathsf{Stack}} (e', \mathsf{Block}(\bot, st' :: \epsilon) :: \mathsf{Block}(\ell, stl') :: \mathcal{K_B}' + \mathcal{K}')}$$

11.6.3.2 Array of structures

When an array of n cells of type C is requested to be constructed, the execution point is a ConstrArray: it corresponds to the initializer for the array, which is compiled to a Ds++ statement thanks to the compileConstrArray compilation function. The compiler-purpose variable *obj* is expected to hold a pointer to the first cell of the array being constructed. There are no further statements within the same block of the initializer (further construction operations are recorded by continuation stacks). So, the corresponding $\rhd_{\mathsf{exec}}$ rule follows:

$$\frac{\begin{array}{c}\Gamma = \Gamma_2[\mathsf{Used} \leftarrow \Gamma_2.\mathsf{Used} \uplus \{\mathit{obj}\}] \qquad e'(\underline{\mathit{obj}}) = (\ell, (\alpha, 0, (\mathsf{Repeated}, C :: \epsilon))) \\ e \rhd_{\mathsf{Var}} e' \qquad \mathcal{G}; \Gamma_2 \vdash (e, \mathcal{K}) \rhd_{\mathsf{Stack}} (e', \mathcal{K}') \qquad st' = \mathsf{compileConstrArray}(C, n, i, \mathit{inits}, \mathit{obj}, \Gamma)\end{array}}{(\mathsf{ConstrArray}(\ell, \alpha, n, i, C, \mathit{inits}, e), \mathcal{K}, \mathcal{G}) \rhd_{\mathsf{exec}} (st', \epsilon, e', \mathcal{K}', \mathcal{G}')}$$

Then, when executing the initializer for a cell of some array, a Kconstrarray frame is present in the κ++ continuation stack, corresponding to the construction of the further remaining cells. But the cell initializer is compiled within a block, so this stack frame matches a Ds++ Block continuation frame. This rule involves three different compilation contexts:

- The provided Γ compilation context is the context of the initializer, with Γ.curobj and Γ.curpath defined and supposed to respectively hold a pointer to the cell being constructed, and the construction path for a most-derived object (those data will be used when calling the constructor) [15]

15. Note the interesting construct: $\Gamma = \Gamma_1[\ldots][\mathsf{curobj} \leftarrow \Gamma.\mathsf{curobj}][\ldots]$, which is not a *definition* of Γ but an *equality* between Γ and Γ_1, simply expressing that the value of Γ_1.curobj is overwritten by Γ. In particular, this construct allows to express that $\Gamma_1.\mathsf{curobj} = \Gamma.\mathsf{curobj}$ is **not** enforced.

- Γ_1 is the compilation context for compiling the construction of the further cells of the array, thus requiring a compiler-purpose variable *obj* to hold a pointer to the first cell of the array
- Γ_2 is the compilation context for the remainder of the stack

$$\frac{\begin{array}{c}\Gamma = \Gamma_1[\mathsf{Used} \leftarrow \Gamma_1.\mathsf{Used} \uplus \{\Gamma.\mathsf{curobj}\} \uplus \{\Gamma.\mathsf{curpath}\}][\mathsf{curobj} \leftarrow \Gamma.\mathsf{curobj}][\mathsf{curpath} \leftarrow \Gamma.\mathsf{curpath}] \\ e'(\underline{\Gamma.\mathsf{curobj}}) = (\ell, (\alpha, i, (\mathsf{Repeated}, C :: \epsilon))) \qquad e'(\underline{\Gamma.\mathsf{curpath}}) = (C, (\mathsf{Repeated}, C :: \epsilon)) \\ \Gamma_1 = \Gamma_2[\mathsf{Used} \leftarrow \Gamma_2.\mathsf{Used} \uplus \{obj\}] \qquad e'(\underline{obj}) = (\ell, (\alpha, 0, (\mathsf{Repeated}, C :: \epsilon))) \\ e \rhd_{\mathsf{Var}} e' \qquad K' = \mathsf{Block}(\bot, \mathsf{compileConstrArray}(C, n, i+1, inits, obj, \Gamma_1) :: \epsilon) \\ \mathcal{G}; \Gamma_2 \vdash (e, \mathcal{K}) \rhd_{\mathsf{Stack}} (e', \mathcal{K}')\end{array}}{\mathcal{G}; \Gamma \vdash (e, \mathsf{Kconstrarray}(\ell, \alpha, n, i, C, inits) :: \mathcal{K}) \rhd_{\mathsf{Stack}} (e', K' :: \mathcal{K}')}$$

Then, once the constructor for this cell has been called, then Kconstrothercells is present in the κ++ continuation stack. The difference between the two stack frames is the fact that in Kconstrarray, the constructor has not been called yet. So, for Kconstrothercells, a further Callframe continuation frame, expecting the return from constructor (thus $\Gamma.\mathsf{isDestructorBody} = \mathsf{false}$ enforces `return` to be compiled to `return` in the constructor body), has to be prepended before the frame corresponding to the construction of further cells. Thus, the provided compilation context and the environments, related to the body of the called constructor, are irrelevant to the construction of the other cells of the array. Upon return from the constructor, this Callframe continuation frame immediately requests exit from the block enclosing the initializer:

$$\frac{\begin{array}{c}\Gamma.\mathsf{isDestructorBody} = \mathsf{false} \qquad \Gamma_2 = \Gamma_3[\mathsf{Used} \leftarrow \Gamma_3.\mathsf{Used} \uplus \{obj\}] \\ {e_1}'(\underline{obj}) = (\ell, (\alpha, 0, (\mathsf{Repeated}, C :: \epsilon))) \qquad e_1 \rhd_{\mathsf{Var}} {e_1}' \qquad {K_1}' = \mathsf{Callframe}(\bot, \mathtt{exit}\ 1 :: stl', {e_1}') \\ {K_2}' = \mathsf{Block}(\bot, \mathsf{compileConstrArray}(C, n, i+1, inits, obj, \Gamma_2) :: \epsilon) \\ \mathcal{G}; \Gamma_3 \vdash (e_1, \mathcal{K}) \rhd_{\mathsf{Stack}} ({e_1}', \mathcal{K}')\end{array}}{\mathcal{G}; \Gamma \vdash (e^?, \mathsf{Kconstrothercells}(\ell, \alpha, n, i, C, inits, e_1) :: \mathcal{K}) \rhd_{\mathsf{Stack}} (e'^?, {K_1}' :: {K_2}' :: \mathcal{K}')}$$

11.6.3.3 Bases and fields

The construction of a list L of base class subobjects or data members, of an object π within its constructor κ, is represented in κ++ by the Constr execution point, directly compiled to a Ds++ statement thanks to the following compileConstr compilation function:

$$\begin{aligned}\mathsf{compileConstr}(C, \kappa, \mathsf{Bases}(\mathsf{DirectNonVirtual}), L, \Gamma) &\underset{\text{def.}}{=\!=} \mathsf{compileConstrDNVBases}(C, \kappa, L, \Gamma) \\ \mathsf{compileConstr}(C, \kappa, \mathsf{Bases}(\mathsf{Virtual}), L, \Gamma) &\underset{\text{def.}}{=\!=} \mathsf{compileConstrVBases}(C, \kappa, L, \Gamma) \\ \mathsf{compileConstr}(C, \kappa, \mathsf{Fields}, L, \Gamma) &\underset{\text{def.}}{=\!=} \mathsf{compileConstrFields}(C, \kappa, L, \Gamma)\end{aligned}$$

Actually, those compilation functions already include the construction of all subsequent subobjects of the object (i.e. for instance compileConstrDNVBases also includes the construction of the fields of C), which is consistent with the semantics of Constr in κ++.

The compiler-purpose variable $\Gamma.\mathsf{curobj}$ is expected to hold a pointer to the object π, and $\Gamma.\mathsf{curpath}$ is expected to hold the construction path corresponding to the inheritance path to π

from its most-derived object. so that we have the following corresponding $\rhd_{\mathsf{exec}}$ rule:

$$\dfrac{\begin{array}{c} e'(\underline{\Gamma.\mathsf{curobj}}) = \pi = (\ell, (\alpha, i, \sigma)) \\ \mathcal{G} \vdash \langle\ell\rangle\ D_\ell[n_\ell] \ -\!\!\langle\alpha\rangle\!\!\overset{\mathcal{A}}{\rightarrow}\ D[n] \qquad e'(\underline{\Gamma.\mathsf{curpath}}) = (D, \sigma) \qquad e \rhd_{\mathsf{Var}} e' \\ \Gamma = \Gamma_1[\mathsf{Used} \leftarrow \Gamma_1.\mathsf{Used} \uplus \{\Gamma.\mathsf{curobj}\} \uplus \{\Gamma.\mathsf{curpath}\}][\mathsf{curobj} \leftarrow \Gamma.\mathsf{curobj}][\mathsf{curpath} \leftarrow \Gamma.\mathsf{curpath}] \\ \mathcal{G}; \Gamma_1 \vdash (e, \mathcal{K}) \rhd_{\mathsf{Stack}} (e', \mathcal{K}') \end{array}}{(\mathsf{Constr}(\pi, C, \kappa, \beta, L, e), \mathcal{K}, \mathcal{G}) \rhd_{\mathsf{exec}} (\mathsf{compileConstr}(C, \kappa, \beta, L, \Gamma), \epsilon, e', \mathcal{K}', \mathcal{G}')}$$

Then, when actually executing the initializer for such a component, there are three cases: bases, scalar fields, and structure fields.

11.6.3.4 Bases

Consider the construction of a base B of π through the corresponding initializer in κ.

When executing its initializer, it is actually compiled under the Γ compilation context, so that $\Gamma.\mathsf{curobj}$ is assumed to hold a pointer to the base to construct, and $\Gamma.\mathsf{curpath}$ the corresponding construction path, in order to pass those data to the constructor. However, the pointer to the object π itself, as well as its corresponding construction path, have to be kept in the compilation context Γ' used for the construction of siblings of B and further subobjects of π. As usual, the initializer is embedded in a Ds++ block, so the corresponding Ds++ stack frame is a Block:

$$\begin{array}{r} \mathsf{AddBase}(\sigma, C, \mathsf{Virtual}, V) \underset{\text{def.}}{=\!=} (\mathsf{Shared}, V :: \epsilon) \\ \mathsf{AddBase}(\sigma, C, \mathsf{DirectNonVirtual}, B) \underset{\text{def.}}{=\!=} (\sigma @ (\mathsf{Repeated}, C :: B :: \epsilon)) \end{array}$$

$$\dfrac{\begin{array}{c} \tilde{\sigma} = \mathsf{AddBase}(\sigma, C, \beta, B) \\ e'(\underline{\Gamma.\mathsf{curobj}}) = (\ell, (\alpha, i, \tilde{\sigma})) \qquad \mathcal{G} \vdash \langle\ell\rangle\ D_\ell[n_\ell] \ -\!\!\langle\alpha\rangle\!\!\overset{\mathcal{A}}{\rightarrow}\ D[n] \qquad e'(\underline{\Gamma.\mathsf{curpath}}) = (D, \tilde{\sigma}) \\ \Gamma = \Gamma_1[\mathsf{Used} \leftarrow \Gamma_1.\mathsf{Used} \uplus \{\Gamma.\mathsf{curobj}\} \uplus \{\Gamma.\mathsf{curpath}\}][\mathsf{curobj} \leftarrow \Gamma.\mathsf{curobj}][\mathsf{curpath} \leftarrow \Gamma.\mathsf{curpath}] \\ e'(\underline{\Gamma_1.\mathsf{curobj}}) = \pi = (\ell, (\alpha, i, \sigma)) \qquad e'(\underline{\Gamma_1.\mathsf{curpath}}) = (D, \sigma) \\ \Gamma_1 = \Gamma_2[\mathsf{Used} \leftarrow \Gamma_2.\mathsf{Used} \uplus \{\Gamma_1.\mathsf{curobj}\} \uplus \{\Gamma_1.\mathsf{curpath}\}][\mathsf{curobj} \leftarrow \Gamma_1.\mathsf{curobj}][\mathsf{curpath} \leftarrow \Gamma_1.\mathsf{curpath}] \\ K' = \mathsf{Block}(\bot, \mathsf{compileConstr}(C, \kappa, \mathsf{Bases}(\beta), L, \Gamma_1) :: \epsilon) \qquad \mathcal{G}; \Gamma_2 \vdash (e, \mathcal{K}) \rhd_{\mathsf{Stack}} (e', \mathcal{K}') \end{array}}{\mathcal{G}; \Gamma \vdash (e, \mathsf{Kconstr}(\pi, C, \kappa, \mathsf{Bases}(\beta), B, L) :: \mathcal{K}) \rhd_{\mathsf{Stack}} (e', K' :: \mathcal{K}')}$$

Then, once the constructor is called, the corresponding κ++ stack frame is $\mathsf{Kconstrother}(\mathsf{Bases}(\beta))$. Thus, returning from the constructor needs a further $\mathsf{Callframe}$ prepended to the Block continuation frame. Then, the provided environments and compilation context Γ, corresponding to the body of the called constructor, are irrelevant to thhe construction of the further siblings of B. However, $\Gamma.\mathsf{isDestructorBody} = \mathsf{false}$ is necessary to enforce `return` to be compiled to `return` within the constructor body.

$$\dfrac{\begin{array}{c} \Gamma.\mathsf{isDestructorBody} = \mathsf{false} \\ {e_1}'(\underline{\Gamma_2.\mathsf{curobj}}) = \pi = (\ell, (\alpha, i, \sigma)) \qquad \mathcal{G} \vdash \langle\ell\rangle\ D_\ell[n_\ell] \ -\!\!\langle\alpha\rangle\!\!\overset{\mathcal{A}}{\rightarrow}\ D[n] \qquad {e_1}'(\underline{\Gamma_2.\mathsf{curpath}}) = (D, \sigma) \\ \Gamma_2 = \Gamma_3[\mathsf{Used} \leftarrow \Gamma_3.\mathsf{Used} \uplus \{\Gamma_2.\mathsf{curobj}\} \uplus \{\Gamma_2.\mathsf{curpath}\}][\mathsf{curobj} \leftarrow \Gamma_2.\mathsf{curobj}][\mathsf{curpath} \leftarrow \Gamma_2.\mathsf{curpath}] \\ e_1 \rhd_{\mathsf{Var}} {e_1}' \qquad {K_1}' = \mathsf{Callframe}(\bot, \mathtt{exit}\ 1 :: S, {e_1}') \\ {K_2}' = \mathsf{Block}(\bot, \mathsf{compileConstr}(C, \kappa, \mathsf{Bases}(\beta), L, \Gamma_2) :: \epsilon) \qquad \mathcal{G}; \Gamma_3 \vdash (e_1, \mathcal{K}) \rhd_{\mathsf{Stack}} ({e_1}', \mathcal{K}') \end{array}}{\mathcal{G}; \Gamma \vdash (e^?, \mathsf{Kconstrother}(\pi, C, \kappa, \mathsf{Bases}(\beta), B, L, e_1) :: \mathcal{K}) \rhd_{\mathsf{Stack}} (e'^?, {K_1}' :: {K_2}' :: \mathcal{K}')}$$

11.6.3.5 Scalar fields

For a scalar field f, no constructor is expected to be called by the initializer, but a initScalar instead, thus requiring Γ.curfield to be set to f. Moreover, Γ.curobj and Γ.curpath must refer to the current object being constructed, holding the field.

$$\frac{\begin{array}{c} e'(\underline{\Gamma.\mathsf{curobj}}) = \pi = (\ell, (\alpha, i, \sigma)) \\ \mathcal{G} \vdash \langle \ell \rangle\ D_\ell[n_\ell] \prec\!\!\langle \alpha \rangle\!\!\overset{\mathcal{A}}{\rightarrow} D[n] \qquad e'(\underline{\Gamma.\mathsf{curpath}}) = (D, \sigma) \qquad \Gamma = \Gamma_1[\mathsf{curfield} \leftarrow f] \\ \Gamma_1 = \Gamma_2[\mathsf{Used} \leftarrow \Gamma_2.\mathsf{Used} \uplus \{\Gamma.\mathsf{curobj}\} \uplus \{\Gamma.\mathsf{curpath}\}][\mathsf{curobj} \leftarrow \Gamma.\mathsf{curobj}][\mathsf{curpath} \leftarrow \Gamma.\mathsf{curpath}] \\ K' = \mathsf{Block}(\bot, \mathsf{compileConstr}(C, \kappa, \mathsf{Fields}, L, \Gamma_1) :: \epsilon) \qquad \mathcal{G}; \Gamma_2 \vdash (e, \mathcal{K}) \rhd_{\mathsf{Stack}} (e', \mathcal{K}') \end{array}}{\mathcal{G}; \Gamma \vdash (e, \mathsf{Kconstr}(\pi, C, \kappa, \mathsf{Fields}, f, L) :: \mathcal{K}) \rhd_{\mathsf{Stack}} (e', K' :: \mathcal{K}')}$$

11.6.3.6 Structure fields

For a structure field f, actually there is one initializer for each cell, so the κ++ continuation stack frame is Kconstrother(Fields). The invariant is very similar as the one for Kconstr(Fields) for a scalar field, except Γ.curfield is irrelevant:

$$\frac{\begin{array}{c} e'(\underline{\Gamma.\mathsf{curobj}}) = \pi = (\ell, (\alpha, i, \sigma)) \qquad \mathcal{G} \vdash \langle \ell \rangle\ D_\ell[n_\ell] \prec\!\!\langle \alpha \rangle\!\!\overset{\mathcal{A}}{\rightarrow} D[n] \qquad e'(\underline{\Gamma.\mathsf{curpath}}) = (D, \sigma) \\ \Gamma = \Gamma_1[\mathsf{Used} \leftarrow \Gamma_1.\mathsf{Used} \uplus \{\Gamma.\mathsf{curobj}\} \uplus \{\Gamma.\mathsf{curpath}\}][\mathsf{curobj} \leftarrow \Gamma.\mathsf{curobj}][\mathsf{curpath} \leftarrow \Gamma.\mathsf{curpath}] \\ K' = \mathsf{Block}(\bot, \mathsf{compileConstr}(C, \kappa, \mathsf{Fields}, L, \Gamma) :: \epsilon) \qquad \mathcal{G}; \Gamma_1 \vdash (e, \mathcal{K}) \rhd_{\mathsf{Stack}} (e', \mathcal{K}') \end{array}}{\mathcal{G}; \Gamma \vdash (e, \mathsf{Kconstrother}(\pi, C, \kappa, \mathsf{Fields}, f, L, e_1) :: \mathcal{K}) \rhd_{\mathsf{Stack}} (e', K' :: \mathcal{K}')}$$

Note that the κ++ e_1 environment is irrelevant, as it is superseded by e which is used for the initializers of the cells of f.

11.6.4 (⋆) Destruction

11.6.4.1 Array of structures

When an array of n cells of type C is requested to be destructed, the execution point is a DestrArray: it corresponds to the destruction of the cells of the array, which is compiled to a Ds++ statement thanks to the compileDestrArray compilation function using some compilation context Γ such that the compiler-purpose variable *obj* is expected to hold a pointer to the first cell of the array being destructed. There are no further statements within the same block of the initializer (further destruction operations are recorded by continuation stacks). Then, the embedding block, used once the array has been entirely destructed, is compiled using the compilation context Γ_1. So, the corresponding $\rhd_{\mathsf{exec}}$ rule follows:

$$\frac{\begin{array}{c} \Gamma = \Gamma_1[\mathsf{Used} \leftarrow \Gamma_1.\mathsf{Used} \uplus \{obj\}] \qquad e'(\underline{obj}) = (\ell, (\alpha, 0, (\mathsf{Repeated}, C :: \epsilon))) \\ \mathcal{G}; \Gamma_1 \vdash (\bot, \mathcal{K}) \rhd_{\mathsf{Stack}} (e', \mathcal{K}') \qquad st' = \mathsf{compileDestrArray}(C, i, inits, obj, \Gamma) \end{array}}{(\mathsf{DestrArray}(\ell, \alpha, i, C), \mathcal{K}, \mathcal{G}) \rhd_{\mathsf{exec}} (st', \epsilon, e', \mathcal{K}', \mathcal{G}')}$$

$\rhd_{\mathsf{Stack}}$ carries no κ++ environment (⊥) as there is actually none available.

Then, when destructing an array cell, a Kdestrcell frame appears in the κ++ continuation stack frame, actually corresponding to a Callframe expecting return from destructor. It requires no compilation context of its own. By contrast, Γ_1 is the environment used for compiling the destruction of further array cells, where *obj* holds a pointer to the first cell of the array. Finally,

Γ_2 is the compilation context to use once the array has been entirely destructed, either for the destruction of further fields if the array corresponds to a field ($\mathsf{Kdestrother}(\mathsf{Fields})$), or for the execution of further statements if the array corresponds to a complete object to destruct ($\mathsf{Kcontinue}$).

$$\frac{\begin{array}{c}\Gamma_1 = \Gamma_2[\mathsf{Used} \leftarrow \Gamma_2.\mathsf{Used} \uplus \{obj\}] \qquad {e_1}'(\underline{obj}) = (\ell, (\alpha, 0, (\mathsf{Repeated}, C :: \epsilon)))\\ K' = \mathsf{Callframe}(\bot, \mathsf{compileDestrArray}(C, i-1, obj, \Gamma_1) :: \epsilon, {e_1}')\\ \mathcal{G}; \Gamma_2 \vdash (\bot, \mathcal{K}) \triangleright_{\mathsf{Stack}} ({e_1}', \mathcal{K}')\end{array}}{\mathcal{G}; \bot \vdash (\bot, \mathsf{Kdestrcell}(\ell, \alpha, i, C) :: \mathcal{K}) \triangleright_{\mathsf{Stack}} (\bot, K' :: \mathcal{K}')}$$

11.6.4.2 Destructor body

While executing the destructor body for an object π, a Kdestr frame is present in the κ++ continuation stack, to mean that once the destructor body exits, the destruction of fields and bases of π may start, in the reverse order of their construction. In Ds++, it actually corresponds to a Block frame, expecting exit from the block within which the destructor body has been compiled. The destructor body is compiled using the compilation context Γ, for which curobj must hold a pointer to π being destructed, and curpath the corresponding construction path. The isDestructorBody flag enforces `return` statements to be turned into appropriate `exit`s, which allows inlining the destructor body. Upon exit, the fields of π will start destructing in reverse order.

$$\frac{\begin{array}{c}\{\Gamma.\mathsf{curobj}\} \uplus \{\Gamma.\mathsf{curpath}\} \subseteq \Gamma.\mathsf{Used} \qquad \Gamma.\mathsf{isDestructorBody} = \mathsf{true}\\ e'(\underline{\Gamma.\mathsf{curobj}}) = \pi = (\ell, (\alpha, i, \sigma)) \qquad \mathcal{G} \vdash \langle \ell \rangle\ D_\ell[n_\ell] \prec\!\!\langle \alpha \rangle\!\!\overset{\mathcal{A}}{\rightarrow} D[n] \qquad e'(\underline{\Gamma.\mathsf{curpath}}) = (D, \sigma)\\ K' = \mathsf{Block}(\bot, \mathsf{compileDestrFields}(C, \mathsf{rev}(\mathcal{F}(C)), \Gamma) :: \epsilon) \qquad \mathcal{G}; \bot \vdash (\bot, \mathcal{K}) \triangleright_{\mathsf{Stack}} (\bot, \mathcal{K}')\end{array}}{\mathcal{G}; \Gamma \vdash (e, \mathsf{Kdestr}(\pi, C) :: \mathcal{K}) \triangleright_{\mathsf{Stack}} (e', K :: \mathcal{K}')}$$

The e environment and the Γ compilation context provided during the execution of the destructor body are not "transmitted" to the remaining part of the stack (which is relevant only upon return from the compiled destructor).

11.6.4.3 Fields

During the destruction of a list of fields L of an object of type C, the κ++ execution point is Destr(Fields), which is directly compiled to a Ds++ statement thanks to the compileDestrFields compilation function. The environment e' and the compilation context Γ are no longer relevant when the destruction of π ends; so they are not "transmitted" to the stack.

$$\frac{\begin{array}{c}\{\Gamma.\mathsf{curobj}\} \uplus \{\Gamma.\mathsf{curpath}\} \subseteq \Gamma.\mathsf{Used} \qquad e'(\underline{\Gamma.\mathsf{curobj}}) = \pi = (\ell, (\alpha, i, \sigma))\\ \mathcal{G} \vdash \langle \ell \rangle\ D_\ell[n_\ell] \prec\!\!\langle \alpha \rangle\!\!\overset{\mathcal{A}}{\rightarrow} D[n] \qquad e'(\underline{\Gamma.\mathsf{curpath}}) = (D, \sigma) \qquad \mathcal{G}; \bot \vdash (\bot, \mathcal{K}) \triangleright_{\mathsf{Stack}} (\bot, \mathcal{K}')\end{array}}{(\mathsf{Destr}(C, \mathsf{Fields}, L), \mathcal{K}, \mathcal{G}) \triangleright_{\mathsf{exec}} (\mathsf{compileDestrFields}(C, L, \Gamma), \epsilon, e', \mathcal{K}', \mathcal{G}')}$$

During the destruction of a structure field f, a Kdestrother(Fields) is present in the stack to remind the further fields to destruct. As we saw before, the destruction of the array is included in a block, so that the embedding block corresponds to the destruction of further fields. So, this κ++ stack frame matches a Block in Ds++. The construction context Γ and the environment e' come from the compilation of the destruction of the array cells of f, thus they must be explicitly provided. But as before, they are no longer relevant once the destruction of π ends.

$$\frac{\begin{array}{c}\{\Gamma.\mathsf{curobj}\} \uplus \{\Gamma.\mathsf{curpath}\} \subseteq \Gamma.\mathsf{Used} \qquad e'(\Gamma.\mathsf{curobj}) = \pi = (\ell, (\alpha, i, \sigma)) \\ \mathcal{G} \vdash \langle \ell \rangle\ D_\ell[n_\ell] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\mapsto} D[n] \qquad e'(\Gamma.\mathsf{curpath}) = (D, \sigma) \qquad \mathcal{G}; \bot \vdash (\bot, \mathcal{K}) \rhd_{\mathsf{Stack}} (\bot, \mathcal{K}')\end{array}}{\mathcal{G}; \Gamma \vdash (\bot, \mathsf{Kdestrother}(\pi, C, \mathsf{Fields}, L) :: \mathcal{K}) \rhd_{\mathsf{Stack}} (e', \mathsf{Block}(\bot, \mathsf{compileDestrFields}(C, L, \Gamma) :: \epsilon) :: \mathcal{K}')}$$

11.6.4.4 Non-virtual bases

During the destruction of a list of direct non-virtual bases L of an object of type C, the κ++ execution point is $\mathsf{Destr}(\mathsf{Bases}(\mathsf{DirectNonVirtual}))$, which is directly compiled to a Ds++ statement thanks to the $\mathsf{compileDestrDNVBases}$ compilation function. The environment e' and the compilation context Γ are no longer relevant when the destruction of π ends; so they are not "transmitted" to the stack.

$$\frac{\begin{array}{c}\{\Gamma.\mathsf{curobj}\} \uplus \{\Gamma.\mathsf{curpath}\} \subseteq \Gamma.\mathsf{Used} \qquad e'(\underline{\Gamma.\mathsf{curobj}}) = \pi = (\ell, (\alpha, i, \sigma)) \\ \mathcal{G} \vdash \langle \ell \rangle\ D_\ell[n_\ell] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\mapsto} D[n] \qquad e'(\underline{\Gamma.\mathsf{curpath}}) = (D, \sigma) \qquad \mathcal{G}; \bot \vdash (\bot, \mathcal{K}) \rhd_{\mathsf{Stack}} (\bot, \mathcal{K}')\end{array}}{(\mathsf{Destr}(C, \mathsf{Bases}(\mathsf{DirectNonVirtual}), L), \mathcal{K}, \mathcal{G}) \rhd_{\mathsf{exec}} (\mathsf{compileDestrDNVBases}(C, L, \Gamma), \epsilon, e', \mathcal{K}', \mathcal{G}')}$$

During the destruction of a direct non-virtual base B, a $\mathsf{Kdestrother}(\mathsf{Bases}(\mathsf{DirectNonVirtual}))$ is present in the stack to remind the further direct non-virtual bases to destruct. The destruction of B corresponds to running the destructor, which is compiled to a Ds++ static function, so this κ++ stack frame matches a $\mathsf{Callframe}$ in Ds++. For this reason, the construction context Γ and the environment e' need not be provided. Similarly, they need not be transmitted to the remaining part of the stack.

$$\frac{\begin{array}{c}\{\Gamma.\mathsf{curobj}\} \uplus \{\Gamma.\mathsf{curpath}\} \subseteq \Gamma.\mathsf{Used} \\ e'(\Gamma.\mathsf{curobj}) = \pi = (\ell, (\alpha, i, \sigma)) \qquad \mathcal{G} \vdash \langle \ell \rangle\ D_\ell[n_\ell] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\mapsto} D[n] \qquad e'(\Gamma.\mathsf{curpath}) = (D, \sigma) \\ K' = \mathsf{Callframe}(\bot, \mathsf{compileDestrDNVBases}(C, L, \Gamma) :: \epsilon, e') \qquad \mathcal{G}; \bot \vdash (\bot, \mathcal{K}) \rhd_{\mathsf{Stack}} (\bot, \mathcal{K}')\end{array}}{\mathcal{G}; \bot \vdash (\bot, \mathsf{Kdestrother}(\pi, C, \mathsf{Bases}(\mathsf{DirectNonVirtual}), L) :: \mathcal{K}) \rhd_{\mathsf{Stack}} (\bot, K' :: \mathcal{K}')}$$

11.6.4.5 Virtual bases

During the destruction of a list of virtual bases L of an object of type C, the κ++ execution point is $\mathsf{Destr}(\mathsf{Bases}(\mathsf{Virtual}))$, which is directly compiled to a Ds++ statement thanks to the $\mathsf{compileDestrVBases}$ compilation function. Destructing the virtual bases of an object only occurs for a most-derived object, i.e. for an array cell; however, the corresponding κ++ **Kdestrcell frame has disappeared from the stack**, as its information is already contained in the $\mathsf{Destr}(\mathsf{Bases}(\mathsf{Virtual}))$ execution point: destructing the virtual bases of cell i also reminds of destructing the further cells $i-1$ down to 0 of the same array. This does not mean that the corresponding Ds++ frame has disappeared, on the contrary. The trick is then to pretend that this κ++ frame still exists for the purpose of the proof. Thus, the following rule is slightly different from its non-virtual counterpart:

$$\frac{\begin{array}{c}\{\Gamma.\mathsf{curobj}\} \uplus \{\Gamma.\mathsf{curpath}\} \subseteq \Gamma.\mathsf{Used} \\ e'(\underline{\Gamma.\mathsf{curobj}}) = \pi = (\ell, (\alpha, i, \sigma)) \qquad \mathcal{G} \vdash \langle \ell \rangle\ D_\ell[n_\ell] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\mapsto} D[n] \\ e'(\underline{\Gamma.\mathsf{curpath}}) = (D, \sigma) \qquad \mathcal{G}; \bot \vdash (\bot, \mathsf{Kdestrcell}(\ell, \alpha, i, C) :: \mathcal{K}) \rhd_{\mathsf{Stack}} (\bot, \mathcal{K}')\end{array}}{(\mathsf{Destr}(C, \mathsf{Bases}(\mathsf{Virtual}), L), \mathcal{K}, \mathcal{G}) \rhd_{\mathsf{exec}} (\mathsf{compileDestrVBases}(C, L, \Gamma), \epsilon, e', \mathcal{K}', \mathcal{G}')}$$

The same reasoning is necessary during the destruction of a virtual base V of C, when $\mathsf{Kdestrother}(\mathsf{Bases}(\mathsf{Virtual}))$ is present in the κ++ continuation stack:

$$\frac{\begin{array}{c}\{\Gamma.\mathsf{curobj}\} \uplus \{\Gamma.\mathsf{curpath}\} \subseteq \Gamma.\mathsf{Used} \\ e'(\Gamma.\mathsf{curobj}) = \pi = (\ell, (\alpha, i, \sigma)) \qquad \mathcal{G} \vdash \langle \ell \rangle\ D_\ell[n_\ell] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\succ}\ D[n] \\ e'(\Gamma.\mathsf{curpath}) = (D, \sigma) \qquad K' = \mathsf{Callframe}(\bot, \mathsf{compileDestrDNVBases}(C, L, \Gamma) :: \epsilon, e') \\ \mathcal{G}; \bot \vdash (\bot, \mathsf{Kdestrcell}(\ell, \alpha, i, C) :: \mathcal{K}) \rhd_{\mathsf{Stack}} (\bot, \mathcal{K}')\end{array}}{\mathcal{G}; \bot \vdash (\bot, \mathsf{Kdestrother}(\pi, C, \mathsf{Bases}(\mathsf{Virtual}), L) :: \mathcal{K}) \rhd_{\mathsf{Stack}} (\bot, K' :: \mathcal{K}')}$$

11.6.4.6 Continue after destruction

When a block-scoped object is requested to be constructed, a $\mathsf{Kcontinue}$ stack frame is present in the κ++ stack. This frame records the "successor exit" statement (i.e. the transformed `exit` statement once the block is actually left – or the `return` statement), the statements of the immediately enclosing block, as well as the statements of the further enclosing blocks, and the environment. Indeed, during destruction, the κ++ environment is not used.

The instruction to destruct an array is compiled in an additional block. However, there are also blocks previously exited in κ++ and not yet in Ds++. Let Bll be this list of blocks, then this list has $\Gamma.\mathsf{furtherBlocks}$ elements, and any object attached to one of those blocks is deallocated in κ++; and the κ++ statement pipeline corresponds to the statement pipeline once all those blocks are actually exited in Ds++.

A compiler-purpose variable, say *blockvar*, holds a pointer to the first cell of the array being destructed. The type and number of cells of the array must match the ones expected by the compilation.

This rule involves three different compilation contexts:

– the provided Γ compilation context is the context of the initializer, where *blockvar* is defined

– the compilation context Γ_1 is used for compiling the further enclosing blocks, so *blockvar* is no longer relevant

– the compilation context Γ_2 is obtained when reaching the most-enclosing block

$$\frac{\begin{array}{c}\Gamma.\mathsf{Blocks} = (\mathit{blockvar}, \Gamma.\mathsf{Used}, (C, n)) :: \mathit{Blocks}_1 \qquad \Gamma.\mathsf{Used} = \mathit{Used}_1 \uplus \{\mathit{blockvar}\} \\ \Gamma_1 = \Gamma[\mathsf{Blocks} \leftarrow \mathit{Blocks}_1][\mathsf{Used} \leftarrow \mathit{Used}_1][\mathsf{furtherBlocks} \leftarrow \Gamma.\mathsf{furtherBlocks} + 1] \\ \mathcal{G} \vdash \langle \ell \rangle\ C[n] \qquad e'(\underline{\mathit{blockvar}}) = (\ell, \epsilon, 0, (\mathsf{Repeated}, C :: \epsilon)) \\ \mathsf{retrieveStmts}(\mathit{stl}', \mathit{Bll}) = \mathsf{map}[\llbracket\cdot\rrbracket^{\Gamma_1}](\mathit{stl} +\!\!+ \texttt{exit}\ 1 :: \epsilon) \\ \mathsf{length}(\mathit{Bll}) = \Gamma.\mathsf{furtherBlocks} \qquad \forall \ell', \mathit{stl}_{\ell'} : (\ell', \mathit{stl}_{\ell'}) \in \mathit{Bll} \Rightarrow \ell' \in \mathcal{G}.\mathsf{dealloc} \\ K' = \mathsf{Block}(\bot, \llbracket \mathit{st} \rrbracket^{\Gamma_1} :: \mathit{stl}') \qquad \mathcal{G}, e';\ \Gamma_1 \downarrow \Gamma_2 \vdash \mathcal{B} \rhd_{\mathsf{Block}} \mathcal{K}_{\mathcal{B}} \qquad \mathcal{G}; \Gamma_2 \vdash (e, \mathcal{K}) \rhd_{\mathsf{Stack}} (e', \mathcal{K}')\end{array}}{\mathcal{G}; \Gamma \vdash (\bot, \mathsf{Kcontinue}(\ell, \mathit{st}, \mathit{stl}, \mathcal{B}, \mathsf{Destr}(e)) :: \mathcal{K}) \rhd_{\mathsf{Stack}} (e', K' :: \mathsf{map}[\mathsf{Block}](\mathit{Bll}) +\!\!+ \mathcal{K}_{\mathcal{B}} +\!\!+ \mathcal{K}')}$$

11.6.5 Forward simulation

The correctness of the compiler is based on forward simulation, using THEOREM B.1 (p. 340). The following theorem shows that forward simulation holds:

THEOREM II.16 (κ++ to Ds++ forward simulation). $\rhd$ *is a* forward simulation *from κ++ to Ds++:*

– *Initial states match:*

$$\forall s_\circ \in \mathfrak{I} : \exists s'_\circ \in \mathfrak{I}', \exists s'_1 \in \mathfrak{S}' : s'_\circ \overset{\star}{\to}' s'_1 \wedge s_\circ \triangleright s'_1$$

– *Forward invariant preservation:*

$$\begin{aligned} \forall s_1, s_2, {s_1}' : &\quad s_1 \to s_2 \wedge s_1 \triangleright {s_1}' \\ \Rightarrow \exists {s_2}' : &\quad {s_1}' \overset{+}{\to}' {s_2}' \wedge s_2 \triangleright {s_2}' \end{aligned}$$

– *Final states match:*

$$\begin{aligned} \forall (s_f, z) \in \mathfrak{F}, \forall s'_9 \in \mathfrak{S}' : &\quad s_f \triangleright s'_9 \\ \Rightarrow \exists s'_f \in \mathfrak{F}'_z : &\quad s'_9 \overset{\star}{\to}' s'_f \end{aligned}$$

Graphically:

$$\begin{tikzcd}
\mathbb{S} & \mathfrak{I} \arrow[rr, dash, "\ni"] & & s_\circ \arrow[d, dotted, dash, "\triangleright"] & s_1 \arrow[r, "\mathfrak{e}^?"'] \arrow[d, dash, "\triangleright"] & s_2 \arrow[d, dotted, dash, "\triangleright"] & s_f \arrow[rr, dash, "\in"] \arrow[d, dash, "\triangleright"] & & \mathfrak{F}_z \\
\mathbb{S}' & \mathfrak{I}' \arrow[r, dotted, dash, "\ni"] & s'_\circ \arrow[r, dotted, "'\star"] & s'_1 & s'_1 \arrow[r, dotted, "'+", "\mathfrak{e}^?"'] & s'_2 & s'_{99} \arrow[r, dotted, "'\star"] & s'_f \arrow[r, dotted, dash, "\in"] & \mathfrak{F}'_z
\end{tikzcd}$$

Proof. We sum up the theorems and lemmataused for the proof of invariant preservation for the high-level steps:

s++ execution step	**Proof case**	**Theorem used**
Scalar field read (κ++-field-scalar-read, p. 185)	Object not deallocated	THEOREM II.11 (p. 217)
Starting the construction of the fields of an object (κ++-constr-bases-direct-non-virtual-nil, p. 190)	Generalized dynamic type change	THEOREM II.15 (p. 221)
Entering destructor body for a most-derived object (κ++-destr-array-cons, p. 193)		
Entering destructor body for a base class subobject (κ++-destr-bases-cons, p. 194)		

The remaining part of the proof mostly corresponds to matching the structures of execution states, by fitting the κ++ complex structures for execution points and continuation stack frames (due to construction and destruction) to the Ds++ "current statement and block-or-callframe" simple structure. □

11.7 The CVcm target language

Now that we have compiled constructors and destructors into Ds++, we aim at compiling the obtained intermediate program into a language featuring low-level accesses, which we present in this section.

11.7.1 Virtual table tables

As we saw, modern compilers such as the Common Vendor ABI for Itanium use different virtual tables for construction. To retrieve those virtual tables, additional read-only data structures are required at run time. Those structures are called *virtual table tables* or VTT.

Basically, for each dynamic class C and for each possible inheritance path σ to C, there is a virtual table table containing:

- (a pointer to) the virtual table of each dynamic direct or indirect base class subobject σ' of C, assuming that σ is considered a most-derived object for the purpose of polymorphic operations
- (a pointer to) the VTT of each dynamic direct non-virtual base B of C
- for the VTT of a most-derived object ($\sigma = (\mathsf{Repeated}, C :: \epsilon)$), (a pointer to) the VTT of each dynamic direct or indirect virtual base V of C

A VTT for a most-derived object is called a *main VTT*. For any path σ, the VTT of any direct non-virtual base class subobject of σ (as well as the VTT of any direct or indirect virtual base subobject of σ, if σ is a most-derived object) is said to be a *sub-VTT* of the VTT of σ.

11.7.2 Syntax

The CVcm language[16] features low-level memory accesses. The class hierarchy is no longer required; polymorphic operations are modelled through read-only memory accesses to virtual tables and virtual table tables.

CVcm is based on Vcm (Section 7.2 p. 137) with which it shares the memory model; however, CVcm additionally features virtual table tables.

In more detail, we recall the following operations on low-level memory accesses are common to Cminor, Vcm and CVcm:

- memory read and write
- constant and variable pointer shift
- pointer equality test

Additionally, Vcm and CVcm model C++-style polymorphic operations by the following operations on virtual tables:

- retrieve offset of virtual base
- retrieve offset for dynamic cast
- retrieve pointer to function and *this* pointer adjustment offset for virtual function call

However, contrary to Vcm, CVcm models construction-specific features through virtual table tables (VTTs). Those operations are split into two parts:

- on the one hand, VTT handling operations are close to Ds++ operations handling construction paths, as the representation of VTT is left abstract in CVcm:
  - retrieve pointer to a main VTT

16. Coq development: theory `Target`.

  - retrieve pointer to a sub-VTT of a VTT
- on the other hand, an operation is specific to CVcm and help compile the "set dynamic type" operation: retrieve pointer to virtual table from a VTT and a constant inheritance path.

However, to emphasize that a CVcm program no longer requires a C++-like class hierarchy, we develop an abstract representation of virtual tables independent of class hierarchy.

The CVcm operations for low-level memory access (load, store) are parameterized with the *memory chunk*, which expresses the expected kind of the value to read from memory or to write to memory:

T	$: \mathit{Chunk}$	Type of data writable to memory (chunk)
$\mathit{Chunk} ::=$	t	Built-in type
	$\mid$ `Ptr`	Pointer to memory
	$\mid$ `Fptr`	Pointer to function
	$\mid$ `Vptr`	Pointer to virtual table
	$\mid$ `VTTptr`	Pointer to VTT

In program syntax, we assume a type system for accessing virtual tables and virtual table tables. Section 7.2.1.2 (p. 138) describes the "type system" for virtual tables of Vcm, reused by CVcm without changes. A virtual table type declares all entries that can be defined in virtual tables of this type. From the C++ point of view, a virtual table type corresponds to a class definition, but limited to the virtual bases, the virtual functions and the results of dynamic cast operations.

CVcm additionally introduces VTT types to describe a "type system" for virtual table tables. Similarly, a VTT type declares all entries that can be defined in virtual table tables of this type. From the C++ point of view, a VTT type corresponds to the inheritance hierarchy. However, this type system will be ignored for the "main VTTs", which are the virtual table tables corresponding to most-derived objects.

$$\mathit{VTTTypeDefs} = \mathit{VTTType} \nrightarrow \mathit{VTTTypeDef}$$

$$\mathit{VTTTypeDef} =$$

$\{$

    vtable : $\mathit{VTableRequest} \nrightarrow \mathit{VTableType}$; Types of expected virtual tables

    subvtt : $\mathit{SubVTTRequest} \nrightarrow \mathit{VTTType}$; Types of expected sub-VTTs

$\}$

Then, the syntax of CVcm statements follows:

B	$: \mathit{VBOffRequest}$	Virtual base whose offset is requested
X	$: \mathit{DynCastRequest}$	Target of a requested dynamic cast
F	$: \mathit{DispRequest}$	Virtual function whose dispatch is requested
V	$: \mathit{VTableRequest}$	Request a virtual table from a VTT
τ	$: \mathit{VTableType}$	Type of virtual table
θ	$: \mathit{VTTType}$	Type of VTT

$st ::=$ `if` (x) $st_\top$ `else` $st_\bot$	Conditional
$\mid st_1; st_2$	Statement sequence
$\mid$ `skip`	Do nothing

$\mid$ `loop` st	Loop
$\mid \{st\}$	Statement block
$\mid$ `exit` n	Leaving n blocks
$\mid x' := x$	Variable value duplication
$\mid x' := op(x^*)$	Built-in operation
$\mid x' := sfname(x^*)$	Function call
$\mid x' := (*x)(x^*)$	Function call through function pointer
$\mid$ `return` $x^?$	Return from function
$\mid x' := *_T x$	Memory read
$\mid *_T x := x'$	Memory write
$\mid x' := x_1 == x_2$	Pointer comparison
$\mid \{x : size; st\}$	Memory block allocation
$\mid x' :=$ `vboff`$\langle B\rangle_\tau(x)$	Offset to virtual base
$\mid x' :=$ `dyncastdef`$\langle X\rangle_\tau(x)$	Is dynamic cast defined?
$\mid x' :=$ `dyncastoff`$\langle X\rangle_\tau(x)$	Dynamic cast offset
$\mid x' :=$ `dispfunc`$\langle F\rangle_\tau(x)$	Virtual function dispatch
$\mid x' :=$ `dispoff`$\langle F\rangle_\tau(x)$	`this` pointer adjustment for virtual function dispatch
$\mid x' :=$ `MainVTT`(M)	Main VTT
$\mid x' :=$ `SubVTT`$\langle S\rangle_\theta(x)$	Sub-VTT of a VTT
$\mid x' :=$ `vtable`$\langle V\rangle_\theta(x)$	

Virtual table tables contain the corresponding construction virtual tables, and sub-VTTs for the construction of base class subobjects.

$$VTTs \quad = \quad VTTName \rightharpoonup VTT$$

$VTT \quad =$

$\{$

type	:	$VTTType$	; Type of VTT (for abstract VTT layout)
vtable	:	$VTableRequest \rightharpoonup VTableName$	; Virtual tables
subvtt	:	$SubVTTRequest \rightharpoonup VTTName$	; Sub-VTTs

$\}$

Finally, the components of a CVcm program are:

- the entry point statement
- functions (as in Vcm, there are no more conceptual distinctions between two functions, be they compiled from a Ds++ static function or class member function)
- virtual tables (Section 7.2.1.4 p. 139) and virtual table type declarations (Section 7.2.1.2 p. 138), similarly to Vcm
- virtual table tables and VTT type declarations
- the main VTTs (which contain the virtual table tables used during the normal lifetime of objects)

$$
\begin{array}{lcll}
\mathit{Func} & ::= & (x^*)\{st\} & \text{Function definition} \\
\mathit{Program} & = & & \\
\{ & & & \\
\mathsf{main} & : & st & ;\ \text{Entry point statement} \\
\mathsf{funcs} & : & \mathit{FuncName} \twoheadrightarrow\!\!\!\!\!\!| \ \mathit{Func} & ;\ \text{Functions} \\
\mathsf{vtables} & : & \mathit{VTables} & ;\ \text{Virtual tables (Section 7.2.1.4 p. 139)} \\
\mathsf{vtts} & : & \mathit{VTTs} & ;\ \text{Virtual table tables} \\
\mathsf{vtabletypes} & : & \mathit{VTableTypes} & ;\ \text{Virtual table types} \\
 & & & \ \ \text{(for abstract vtable layout)} \\
 & & & \ \ \text{(Section 7.2.1.2 p. 138)} \\
\mathsf{vtttypes} & : & \mathit{VTTTypes} & ;\ \text{VTT types} \\
 & & & \ \ \text{(for abstract VTT layout)} \\
\mathsf{mainvtts} & : & \mathit{MainVTTRequest} \twoheadrightarrow\!\!\!\!\!\!| \ \mathit{VTTName} & ;\ \text{Main VTTs} \\
\} & & &
\end{array}
$$

11.7.3 Memory model

The low-level memory model[17] of CVcm extends the memory model of Vcm (Section 7.2.2 p. 139), inspired from the CompCert memory model [16]. Memory is organized in several *memory blocks*. Each block is a finite array of bytes. Values are stored within one block, spanning one or several bytes.

Contrary to Vcm, memory blocks may be allocated and deallocated in CVcm. To simplify, similarly to early versions of CompCert, we assume that allocation of a memory block never fails. So, memory operations are summarized as follows:

$$
\begin{array}{lcrl}
\mathit{MemSpec} & = & & \\
\{ & & & \\
\mathsf{chunksize} & : & \mathit{Chunk} \to \mathbb{N}^{>0} & ;\ \text{Size of data chunk} \\
\mathsf{chunkalign} & : & \mathit{Chunk} \to \mathbb{N}^{>0} & ;\ \text{Alignment of data chunk} \\
\mathsf{load} & : & \mathit{Mem} \times \mathit{Chunk} \times \mathit{MemBlock} \times \mathbb{Z} \to \mathit{Value}^? & ;\ \text{Memory load} \\
\mathsf{store} & : & \mathit{Mem} \times \mathit{Chunk} \times \mathit{MemBlock} \times \mathbb{Z} \times \mathit{Value} \to \mathit{Mem}^? & ;\ \text{Memory store} \\
\mathsf{blocksize} & : & \mathit{Mem} \times \mathit{MemBlock} \to \mathbb{N}^? & ;\ \text{Size of a memory block} \\
\mathsf{alloc} & : & \mathit{Mem} \times \mathbb{Z} \to \mathit{MemBlock} \times \mathit{Mem} & ;\ \text{Allocate a new block} \\
\mathsf{free} & : & \mathit{Mem} \times \mathit{Block} \to \mathit{Mem} & ;\ \text{Deallocate a block} \\
\} & & &
\end{array}
$$

Pointer arithmetics (constant offset shift, and variable offset shift by a constant factor) allow to operate on pointers within a given block. It is impossible to retrieve a pointer within a block from a pointer within another block.

Section 7.2.2.4 (p. 140) axiomatizes the behaviour of memory load and store operations, which do not change since Vcm. Additionally, the following rules axiomatize the behaviour of memory block allocation and free:

– allocation has no impact on reading from already existing blocks:

$$
\frac{\mathsf{alloc}(\mathfrak{M}, sz) = (\mathfrak{M}', b') \qquad b \neq b'}{\mathsf{load}(\mathfrak{M}', T, b, o) = \mathsf{load}(\mathfrak{M}, T, b, o)} \quad \text{(CVcm-mem-load-alloc-other)}
$$

17. Coq development: theory `Memory`.

– the size of an allocated block is well-defined after allocation, and not before allocation:

$$\frac{\mathsf{alloc}(\mathfrak{M}, sz) = (\mathfrak{M}', b')}{\mathsf{blocksize}(\mathfrak{M}', b') = sz} \qquad (\text{CVcm-mem-blocksize-alloc-same-after})$$

$$\frac{\mathsf{alloc}(\mathfrak{M}, sz) = (\mathfrak{M}', b')}{\mathsf{blocksize}(\mathfrak{M}, b') = \bot} \qquad (\text{CVcm-mem-blocksize-alloc-same-before})$$

– allocation does not change the block sizes of already allocated blocks:

$$\frac{\mathsf{alloc}(\mathfrak{M}, sz) = (\mathfrak{M}', b') \qquad b \neq b'}{\mathsf{blocksize}(\mathfrak{M}', b) = \mathsf{blocksize}(\mathfrak{M}, b)} \qquad (\text{CVcm-mem-blocksize-alloc-other})$$

– deallocation has no impact on reading from other blocks, and does not change the sizes of other blocks:

$$\frac{\mathsf{free}(\mathfrak{M}, b) = \mathfrak{M}' \qquad b \neq b'}{\mathsf{load}(\mathfrak{M}', T, b', o') = \mathsf{load}(\mathfrak{M}, T, b', o')} \qquad (\text{CVcm-mem-load-free-other})$$

$$\frac{\mathsf{free}(\mathfrak{M}, b) = \mathfrak{M}' \qquad b \neq b'}{\mathsf{blocksize}(\mathfrak{M}', b') = \mathsf{blocksize}(\mathfrak{M}, b')} \qquad (\text{CVcm-mem-blocksize-free-other})$$

11.7.4 Semantic elements

Values A value is either a *value of built-in type* (integer, floating-point number, etc.), a pointer to a memory location (that is, a pair of a memory block and an integer offset within this memory block), a null pointer, a pointer to a function, a pointer to a virtual table, as in Vcm, or additionally a pointer to a virtual table table:

$Val ::= Builtin$	Value of built-in type
$\mid @(b, o)$	Pointer to memory location at offset o within block b
$\mid \mathsf{NULL}$	Null pointer
$\mid \&FuncName$	Pointer to function
$\mid \&VTableName$	Pointer to virtual table
$\mid \&VTTName$	Pointer to virtual table table

Then, "typing" a value — determining whether a value corresponds to some memory chunk — is given by the rules defined as in Vcm (Section 7.2.2.3 p. 140), to which pointers to virtual table tables have to be additionally taken into account:

$$\frac{}{\&VTTName : \mathtt{VTTptr}}$$

Execution state A CVcm *execution state* of the small-step semantics is composed of the same parts as in Vcm, except that a stack frame corresponding to a statement block can be attached to an allocated memory block. To sum up, a CVcm execution state is composed of:

– the current statement to execute,
– the list of further statements to execute in the same block,

- the environment (mapping of values to variables),
- the continuation stack, which is a list of frames, each frame being either of:
  - leaving a block, with the memory block to deallocate on block exit (if any, as an addition to Vcm) and the further statements to execute after leaving the block
  - returning from a function, with the caller variable to store the result (if any), the caller environment, and the further statements to execute on resumption
- the memory state

$$\begin{array}{llll}
e & = & x \to \mathit{Val}^? & \text{Environment} \\
\mathit{Frame} & ::= & \mathsf{Block}(b^?, st^*) & \text{Further statements} \\
 & & & \text{after leaving a block} \\
 & | & \mathsf{Callframe}(x^?, st^*, e) & \text{Return from function} \\
\mathcal{K} & ::= & \mathit{Frame}^* & \text{Continuation stack} \\
\mathit{State} & ::= & (st, st^*, e, \mathcal{K}, \mathit{Mem}) & \text{Execution state}
\end{array}$$

11.7.5 Semantic rules

The small-step semantics of CVcm is given by the transition relation $\to$ between two transition states, defined in this section.

11.7.5.1 Structured control, variable value duplication, built-in operations

Most structured control behaves similarly as in other Compcert-like languages: conditionals, sequences, infinite loops, and return from call (once all statements blocks within the current function have been left), as well as variable value duplication, and built-in operations (Hypothesis 3.2.2 p. 68). CVcm reuses the corresponding rules of s++ defined in Section 4.4.1.1 (p. 83).

11.7.5.2 Function call, memory accesses, pointer arithmetics and virtual tables

The semantics of CVcm exactly agrees with Vcm concerning function calls (Section 7.2.4.2 p. 142), memory accesses and pointer arithmetics (Section 7.2.4.3 p. 142), and virtual tables (Section 7.2.4.4 p. 143), including thunk calls.

11.7.5.3 Statement blocks

For statement blocks with no attached memory blocks, CVcm takes the corresponding Ds++ rules defined in Section 11.4.2 (p. 248).

However, contrary to Vcm, CVcm allows creating and destroying memory blocks. When entering a statement block requesting a memory block of size sz, such a memory block is allocated and attached to the statement block. The programmer then obtains a pointer to the offset 0 within this block:

$$\frac{sz > 0 \qquad \mathsf{alloc}(\mathfrak{M}, sz) = (b, \mathfrak{M}') \qquad e' = e[c \leftarrow @(b,0)]}{\begin{array}{lllll} & (\{c : sz; st\}, & stl, & e, & \mathcal{K}, \quad \mathfrak{M}) \\ \to & (st, & \epsilon, & e', & \mathsf{Block}(b, stl) :: \mathcal{K}, \quad \mathfrak{M}') \end{array}} \quad (\text{CVcm-block-some})$$

When leaving a statement block attached to a memory block, this object is deallocated:

$$\frac{\mathsf{free}(\mathfrak{M}, b) = \mathfrak{M}'}{\begin{array}{llllll} & (\texttt{exit}\ (\mathsf{S}\ n), & stl', & e, & \mathsf{Block}(b, stl) :: \mathcal{K}, & \mathfrak{M}) \\ \to & (\texttt{exit}\ n \quad , & stl, & e, & \mathcal{K}, & \mathfrak{M}') \end{array}} \quad (\textsf{CVcm-exit-some})$$

$$\frac{\mathsf{free}(\mathfrak{M}, b) = \mathfrak{M}'}{\begin{array}{llllll} & (\texttt{return}\ x^?, & stl', & e, & \mathsf{Block}(b, stl) :: \mathcal{K}, & \mathfrak{M}) \\ \to & (\texttt{return}\ x^?, & stl, & e, & \mathcal{K}, & \mathfrak{M}') \end{array}} \quad (\textsf{CVcm-return-some})$$

11.7.5.4 Virtual table tables

Finally, it remains to define the operations on virtual table tables: retrieving a pointer to a sub-VTT, or a pointer to a construction virtual table.

Sub-VTTs The relation $vttname : \theta \vdash S \rightsquigarrow vttname'$ denotes the fact that accessing the sub-VTT S of the VTT $vttname$ of type θ succeeds by returning the VTT $vttname'$. Access to sub-VTT actually relies on the underlying type system for virtual table tables. This operation is parameterized with a virtual table table type θ such that the type declaration of θ explicitly allows access to the sub-VTT S:

$$\frac{\begin{array}{c} \mathsf{mainvtts}(\theta) = vttname \\ \mathsf{vtts}(vttname) = V \qquad V.\mathsf{type} = \theta \qquad \mathsf{vtttypes}(\theta).\mathsf{subvtt}(S) = \bot \qquad V.\mathsf{subvtt}(S) = vttname' \end{array}}{vttname : \theta \vdash S \rightsquigarrow vttname'}$$

(CVcm-subvtt-access-main)

However, VTT type information is not used when reading the *main VTT* of this type: indeed, from the C++ point of view, a main VTT corresponds to a most-derived object, which may have sub-VTTs for virtual bases, contrary to ordinary VTTs.

$$\frac{\begin{array}{c} \mathsf{vtts}(vttname) = V \qquad V.\mathsf{type} = \theta \\ \mathsf{vtttypes}(\theta).\mathsf{subvtt}(S) = \theta' \qquad V.\mathsf{subvtt}(S) = vttname' \qquad \mathsf{vtts}(vttname').\mathsf{type} = \theta' \end{array}}{vttname : \theta \vdash S \rightsquigarrow vttname'}$$

(CVcm-subvtt-access)

Those two access rules are packed together to form the actual transition rule for accessing a sub-VTT:

$$\frac{e(x) = \&vttname \qquad vttname : \theta \vdash S \rightsquigarrow vttname' \qquad e' = e[x' \leftarrow \&vttname']}{\begin{array}{llllll} & (x' := \mathsf{SubVTT}\langle S\rangle_\theta(x), & stl, & e\,, & \mathcal{K}, & \mathfrak{M}) \\ \to & (\texttt{skip} \quad , & stl, & e', & \mathcal{K}, & \mathfrak{M}) \end{array}} \quad (\textsf{CVcm-subvtt})$$

Virtual tables Access to a construction virtual table contained in a VTT relies on the underlying type system for virtual table tables. This operation is also parameterized with a virtual table table type θ such that the type definition of θ must explicitly allow accessing the requested virtual table:

$$\frac{\begin{array}{cccc} e(x) = \&\mathit{vttname} & \mathsf{vtts}(\mathit{vttname}) = v & v.\mathsf{type} = \theta & \mathsf{vtttypes}(\theta).\mathsf{type}(V) = \tau \\ v.\mathsf{vtable}(V) = \mathit{vname}' & \mathsf{vtables}(\mathit{vname}').\mathsf{type} = \tau & e' = e[x' \leftarrow \&\mathit{vname}'] & \end{array}}{\begin{array}{rl} & (x' := \mathtt{vtable}\langle V\rangle_\theta(x), \ \mathit{stl}, \ e, \ \mathcal{K}, \ \mathfrak{M}) \\ \to & (\mathtt{skip}, \ \mathit{stl}, \ e', \ \mathcal{K}, \ \mathfrak{M}) \end{array}}$$

(CVcm-vtt-vtable)

11.8 A compiler from Ds++ to CVcm

Now that the semantics of the CVcm target has been defined, we can compile Ds++ programs into CVcm. Our compiler, described in this section[18], is mostly based on the compilation of s++ to Vcm described in Section 7.3 (p. 145). The main differences are that virtual tables have to be compiled slightly differently to include construction virtual tables; and virtual table tables have to be added, as well as the “set dynamic type” operation, which is actually intended to update the low-level dynamic type data (pointers to virtual tables).

We reuse the notations and definitions of the s++-to-Vcm compiler: compiled statements (*Notation* 7.3.1 p. 145), variables (*Notation* 7.3.2 p. 145), function names (*Notation* 7.3.3 p. 145), memory chunks corresponding to scalar data types (*Definition* 7.3.4 p. 146. Moreover, we also match the CVcm sizes of memory chunks and pointers to virtual tables with the corresponding sizes of scalar data and dynamic type data given by the object layout algorithm, in the same way as in Vcm (Hypothesis 7.3.1 p. 146, Hypothesis 7.3.2 p. 146). However, no hypothesis is necessary about the size or alignment of pointers to virtual table tables, since such pointers are never stored or read from memory.

11.8.1 Construction of virtual tables

The virtual tables[19] of an object change during their construction or destruction, due to the “set dynamic type” operation. Let $\pi = (\ell, \alpha, i, \sigma)$ a pointer to an inheritance subobject of a most-derived D object: such that $\mathcal{G} \vdash D_\ell[n_\ell] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\succ} D[\ell]$. Then, σ may be written $\sigma_\circ @ \sigma'$ where $D \prec\!\langle\sigma_\circ\rangle\!\overset{\mathcal{I}}{\succ} C_\circ \prec\!\langle\sigma_1\rangle\!\overset{\mathcal{I}}{\succ} C$ such that $\sigma_\circ$ is the generalized dynamic type of π. But, once this generalized dynamic type exists, the polymorphic behaviour only depends on σ_1, pretending that the most-derived object is of type $C_\circ$, although offset computations must also take $\sigma_\circ$ into account.

Thus, the program needs the virtual tables for all inheritance objects σ from D to some dynamic class C such that $\sigma = \sigma_\circ @ \sigma'$ for some inheritance paths $\sigma_\circ$ from D to some class $C_\circ$ and σ' from $C_\circ$ to C that are not primary subobjects of other subobjects of $\sigma_\circ$, i.e. such that $\sigma' = \mathsf{reducePath}(\sigma')$. As such, we pose:

$$\mathit{VtableType} \underset{\text{def.}}{=\!=} \{C : \mathsf{isDynamic}(C)\}$$

$$\mathit{VtableName} \underset{\text{def.}}{=\!=} \left\{ (D, \sigma_\circ, \sigma') : \begin{array}{l} D \prec\!\langle\sigma_\circ\rangle\!\overset{\mathcal{I}}{\succ} C_\circ \prec\!\langle\sigma'\rangle\!\overset{\mathcal{I}}{\succ} C \\ \mathsf{isDynamic}(C) \\ \sigma' = \mathsf{reducePath}(\sigma') \end{array} \right\}$$

18. Coq development: theory `Interm2Target`.
19. Coq development: theory `Vtables`.

For any virtual table name $(D, \sigma_\circ, (h, l))$, its type is the destination of the inheritance path (h, l):

$$\mathsf{vtables}(D, \sigma_\circ, (h, l)).\mathsf{type} \underset{\text{def.}}{=\!=} \mathsf{last}(l)$$

The declarations of entries allowed in virtual tables, at the level of virtual table types, are computed in a way similar to Vcm (Section 7.3.1.6 p. 150).

The actual contents of virtual tables are also computed in a way similar to Vcm, but with the main difference that in CVcm the most-derived object is given by the generalized dynamic type. The consequent changes are explained in more detail in the following sections.

11.8.1.1 Dynamic casts

Lemma 11.8.1. *Consider a most-derived $D_\circ$ object. Let D be a base of $D_\circ$, C be a base of D, B be a dynamic primary non-virtual (direct or indirect) base of C, and X a class that is not a base of C. Then, the dynamic cast from B to X pretending D is the most-derived object succeeds if, and only if, the dynamic cast from C to X succeeds, and then both cast to the same subobject of $D_\circ$ and with the same offset adjustment.*

Proof. This is a direct consequence of Lemma 7.3.3 (p. 147). As regards the offset adjustment, let $\sigma_\circ$ be the path from $D_\circ$ to D, and let σ' be the inheritance subobject from D to the X subobject resulting from the dynamic cast. Then, the adjustment corresponding to the cast from B to X is performed through the offset $\mathsf{soff}_{D_\circ}(\sigma_\circ @ \sigma') - \mathsf{soff}_{D_\circ}(\sigma_\circ @ \sigma @ (\mathsf{Repeated}, l)) = \mathsf{soff}_{D_\circ}(\sigma_\circ @ \sigma) + \mathsf{nvsoff}(l)$ per Lemma 5.4.1 (p. 105). But B is a primary base of C. Thus l is a primary path, which implies that $\mathsf{nvsoff}(l) = 0$ and concludes. □

Let $D_\circ \prec\!\langle \sigma_\circ \rangle\!\overset{\mathcal{I}}{\rightarrowtail} D \prec\!\langle \sigma \rangle\!\overset{\mathcal{I}}{\rightarrowtail} C$ be an inheritance path from $D_\circ$ to a dynamic class C. Thanks to this lemma, we define, for any class X, the following offset $\Delta(D_\circ, \sigma_\circ, \sigma, X)$, by well-founded induction on C using the well-founded order $\prec$ (Section 4.1.4.2 p. 79):

$$\Delta(D_\circ, \sigma_\circ, \sigma, X) = \begin{cases} \mathsf{soff}_{D_\circ}(\sigma_\circ @ \sigma') - \mathsf{soff}_{D_\circ}(\sigma_\circ @ \sigma) & \text{if } D \prec\!\langle \sigma \rangle\!\overset{\mathcal{I}}{\rightarrowtail} C \wedge \mathsf{DynCast}(D, \sigma, C, X, \sigma') \\ & \text{and } X \text{ not a base of } C \\ \bot & \text{if } D \prec\!\langle \sigma \rangle\!\overset{\mathcal{I}}{\rightarrowtail} C \wedge \mathsf{DynCast}(D, \sigma, C, X, \bot) \\ & \text{and } X \text{ not a base of } C \\ \Delta(D_\circ, \sigma_\circ @ \sigma @ (\mathsf{Repeated}, C :: B :: \epsilon), X) & \text{if } \mathsf{pbase}(C) = B \text{ and } X \text{ base of } C \\ \bot & \text{if } \mathsf{pbase}(C) = \bot \text{ and } X \text{ base of } C \end{cases}$$

Theorem II.17 (Correctness of CVcm virtual tables: dynamic cast). *Let $D_\circ \prec\!\langle \sigma_\circ \rangle\!\overset{\mathcal{I}}{\rightarrowtail} D \prec\!\langle \sigma \rangle\!\overset{\mathcal{I}}{\rightarrowtail} C \prec\!\langle l \rangle\!\overset{\mathcal{NV}}{\rightarrowtail} B$ be an inheritance path from $D_\circ$ to B such that the generalized dynamic type of $\sigma_\circ @ \sigma$ is $\sigma_\circ$, and l is a non-virtual primary path.*

Then, for any class X that is not a base of B, the dynamic cast from B to X pretending D to be the most-derived object succeeds if, and only if, $\Delta(D_\circ, \sigma_\circ, \sigma, X) = \delta \neq \bot$, and, in this case, the adjustment is performed by adding offset δ.

Proof. The proof is similar to that of Theorem I.12 (p. 148), using Lemma 11.8.1 (p. 294). □

11.8.1.2 Virtual function dispatch

Let $D_\circ \prec\!\langle\sigma_\circ\rangle\!\overset{\mathcal{I}}{\rightarrowtail} C_\circ \prec\!\langle\sigma\rangle\!\overset{\mathcal{I}}{\rightarrowtail} C$ be an inheritance subobject of $D_\circ$ of static type C, such that the generalized dynamic type of $\sigma_\circ @ \sigma$ is $\sigma_\circ$ of type $C_\circ$. Corollary 7.3.6 (p. 149) allows to define, by well-founded induction on $\mathsf{last}(\sigma)$ (using the $\prec$ order on class names, cf. Section 4.1.4.2 p. 79), the following $\Phi(D_\circ, \sigma_\circ, \sigma, f)$ function:

$$\Phi(DC_\circ, \sigma_\circ, \sigma, f) = \begin{cases} (\underline{(B, f)}, \mathsf{soff}_{D_\circ}(\sigma_\circ @ \sigma') - \mathsf{soff}_{D_\circ}(\sigma_\circ @ \sigma)) & \text{if } \mathsf{VFDispatch}(C_\circ, \sigma, f, B, \sigma') \\ \Phi(D_\circ, \sigma_\circ, \sigma @ (\mathsf{Repeated}, A :: B :: \epsilon), f) & \text{if dispatch fails for } \sigma \\ & \text{and } \mathsf{pbase}(A) = B \\ \bot & \text{if dispatch fails for } \sigma \\ & \text{and } \mathsf{pbase}(A) = \bot \end{cases}$$

Theorem II.18 (Correctness of CVcm virtual tables: virtual function dispatch). *Let $D_\circ \prec\!\langle\sigma_\circ\rangle\!\overset{\mathcal{I}}{\rightarrowtail} C_\circ \prec\!\langle\sigma\rangle\!\overset{\mathcal{I}}{\rightarrowtail} C$ be an inheritance subobject of $C_\circ$ of static type C, such that the generalized dynamic type of $\sigma_\circ @ \sigma$ is $\sigma_\circ$.*

If virtual function dispatch for f succeeds on a primary subobject $\sigma @ (\mathsf{Repeated}, l)$ of σ, then $\Phi(C_\circ, \sigma_\circ, \sigma, f) = (\underline{(B, f)}, \delta)$ where B is the class of the final overrider, and δ the offset for the `this` *pointer adjustment.*

Proof. The proof is similar to the proof of Theorem I.13 (p. 150). □

11.8.1.3 Summary

Now we can compute the contents of virtual tables, based on the class hierarchy. Thanks to Theorem I.11 (p. 147), the finite map of virtual base offsets of a virtual table $(D, \sigma_\circ, (h, l))$ may be computed as follows:

$$\mathsf{vtables}(D, \sigma_\circ, (h, l)).\mathsf{vboff} \overset{}{\underset{\text{def.}}{=\!=}} \begin{array}{rcl} \mathcal{V}(\mathsf{last}(l)) & \twoheadrightarrow\!\!\!\!\!\not\;\; & \mathbb{Z} \\ V & \mapsto & \mathsf{vboff}_D(V) - \mathsf{soff}_D(\sigma_\circ @ (h, l)) \end{array}$$

Similarly, thanks to Theorem II.17 (p. 294), the finite map of dynamic cast offsets for a virtual table $(D, \sigma_\circ, (h, l))$ may be computed as follows:

$$\mathsf{vtables}(D, \sigma_\circ, (h, l)).\mathsf{dyncast} \underset{\text{def.}}{=\!=} \begin{array}{rcl} \mathcal{C} & \twoheadrightarrow\!\!\!\!\!\not\;\; & \mathbb{Z} \\ X & \mapsto & \Delta(D, \sigma_\circ, (h, l), X) \end{array}$$

Similarly, thanks to Theorem II.18 (p. 295), the finite map of virtual function dispatch for a virtual table $(D, \sigma_\circ, (h, l))$ may be computed as follows:

$$\mathsf{vtables}(D, \sigma_\circ, (h, l)).\mathsf{disp} \underset{\text{def.}}{=\!=} \begin{array}{rcl} \mathcal{M}'(\mathsf{last}(l)) & \twoheadrightarrow\!\!\!\!\!\not\;\; & \mathit{FuncName} \times \mathbb{Z} \\ f & \mapsto & \Phi(D, \sigma_\circ, (h, l), f) \end{array}$$

11.8.2 Construction of virtual table tables

VTT types Virtual table tables are used during the construction of an object of dynamic class type:

$$\mathit{VTTTypes} \underset{\text{def.}}{=\!=} \{C : \mathsf{isDynamic}(C)\}$$

Then, a virtual table table of type C may contain the sub-VTTs for each direct non-virtual base B of C. Then, each such sub-VTT is expected to be of the corresponding type B:

$$\begin{array}{rrcl} \mathsf{vtttypes}(C).\mathsf{subvtt} : & \{\mathsf{DirectNonVirtual}\} \times \mathcal{DNV}(C) & \twoheadrightarrow\!\!\!\!\!\!\!\!+ & \mathcal{DNV}(C) \\ & (\mathsf{DirectNonVirtual}, B) & \mapsto & B \end{array}$$

Indeed, the entries for virtual bases are expected only at the level of the VTT for the most-derived object, i.e. the main VTT. Thus, it is not necessary to include them in the VTT type declaration.

Then, a virtual table table of type C may also contain the virtual tables for all inheritance paths from C to dynamic classes, which are not primary subobjects of other subobjects of C. The set of all those inheritance paths is written P_C and defined as follows:

$$P_C \underset{\text{def.}}{=\!=} \left\{ (h,l) : \begin{array}{l} C \prec\!(h,l)\!\overset{\mathcal{I}}{\succ} B \\ \mathsf{isDynamic}(B) \\ l = \mathsf{reducePath}(l) \end{array} \right\}$$

Thanks to this definition, the virtual tables that may be defined in a virtual table table of type C are:

$$\begin{array}{rrcl} \mathsf{vtttypes}(C).\mathsf{vtable} : & P_C & \twoheadrightarrow\!\!\!\!\!\!\!\!+ & \mathcal{C} \\ & (h,l) & \mapsto & \mathsf{last}(l) \end{array}$$

Virtual table tables A virtual table table is defined for each inheritance path to a dynamic class. It is worth noting that VTT names actually correspond to *construction paths*:

$$\mathit{VTTNames} \underset{\text{def.}}{=\!=} \left\{ (D,\sigma) : \begin{array}{l} D \prec\!\sigma\!\overset{\mathcal{I}}{\succ} C \\ \mathsf{isDynamic}(C) \end{array} \right\}$$

The type of a virtual table table (D,σ) is given by the destination of the inheritance path $\sigma = (h,l)$:

$$\mathsf{vtt}(D,(h,l)).\mathsf{type} \underset{\text{def.}}{=\!=} \mathsf{last}(l)$$

The virtual tables of a VTT (D,σ) of type C are the virtual tables $((D,\sigma,\sigma'))$ of all inheritance subobjects $\sigma' \in P_C$ of C of type B where B is a dynamic class, and which are not primary base class subobjects of other inheritance subobjects of C:

$$\begin{array}{rrcl} \mathsf{vtt}(D,\sigma).\mathsf{vtables} : & P_C & \twoheadrightarrow\!\!\!\!\!\!\!\!+ & \mathit{VTableNames} \\ & \sigma' & \mapsto & (D,\sigma,\sigma') \end{array}$$

The sub-VTTs of a VTT (D,σ) of type C are the sub-VTT corresponding to the direct non-virtual bases of C. Moreover, if (D,σ) is the VTT of a most-derived D object $(D,(\mathsf{Repeated}, D :: \epsilon))$ (i.e. the main VTT of type D), then the sub-VTTs for virtual bases of D have to be added:

$$\begin{array}{lrcl} \mathsf{vtt}(D,\sigma).\mathsf{subvtt} : & \mathit{BaseKind} \times \mathcal{C} & \twoheadrightarrow\!\!\!\!\!\!\!\!+ & \mathit{VTTNames} \\ & (\mathsf{DirectNonVirtual}, B)(B \in \mathcal{DNV}(C)) & \mapsto & (D, \sigma@(\mathsf{Repeated}, C :: B :: \epsilon)) \\ & (\mathsf{Virtual}, V)(V \in \mathcal{V}(C), \sigma = (\mathsf{Repeated}, D :: \epsilon)) & \mapsto & (D, (\mathsf{Shared}, V :: \epsilon)) \end{array}$$

11.8.3 Operations unrelated to C++ construction or destruction

The compilation of Ds++ operations to CVcm mostly follow the same compilation scheme as for the compilation from s++ to Vcm:

- structured control, built-in operations, statement blocks with no objects, block exits, static and non-virtual function calls (Section 7.3.2 p. 152)
- field and array accesses (Section 7.3.3 p. 152)
- pointer equality tests (Section 7.3.4 p. 153)
- static casts (Section 7.3.5 p. 153)
- dynamic casts (Section 7.3.6 p. 154)
- virtual function dispatch (Section 7.3.7 p. 154)

11.8.4 Blocks with stack objects

When a Ds++ statement block allocates an object of n cells of type D, this object must correspond to a new memory block large enough to contain the array. That is, of size $n \times \mathsf{size}_D$:

$$[\![\{D\ d[n]; st\}]\!] \underset{\text{def.}}{=\!=} \{\overline{d} : n \times \mathsf{size}_D; [\![st]\!]\}$$

The compilation of statement block exits is also correct, because, contrary to κ++, no destructor is called upon exiting a Ds++ statement block, so that the object is directly deallocated.

11.8.5 Special casts to bases

Ds++ allows special casts from an object to one of its direct non-virtual base subobjects, or virtual base subobjects in the case of a most-derived object. Those casts are used when the generalized dynamic type of the object is not well-defined, especially during construction.

Such casts are translated by constant offset shifts, with no need to use any virtual table:

$$[\![x' := \mathsf{base\_cast}\langle \mathsf{DirectNonVirtual}, B\rangle_C(x)]\!] \underset{\text{def.}}{=\!=} \overline{x'} := \overline{x} + \mathsf{dnvboff}_C(B)$$
$$[\![x' := \mathsf{base\_cast}\langle \mathsf{Virtual}, V\rangle_C(x)]\!] \underset{\text{def.}}{=\!=} \overline{x'} := \overline{x} + \mathsf{vboff}_C(V)$$

11.8.6 Set dynamic type

The *set dynamic type* operation $\mathtt{setDynType}(x, x_{\mathsf{cpath}})^b_C$ consists in declaring a subobject of type C, referred to by variable x, to be the most-derived object for the purpose of polymorphic operations. Then, the construction path held by variable x_{cpath} allows to know the inheritance path from the most-derived object to the subobject operated on.

In practice, this step corresponds to the times when pointers to virtual tables within subobjects are changed[20]. Actually, the construction path held in x_{cpath} corresponds in CVcm to a pointer to the relevant virtual table table. This pointer helps retrieve those pointers to virtual tables.

20. Coq development: theory `CompileSetDynType`.

11.8.6.1 Compilation

There are three cases:

Not a dynamic class type If the subobject, on which the set dynamic type operation is performed, is not of dynamic class type, then there are no subobjects to update: there is nothing to do.

$$[\![\mathtt{setDynType}(x, x_{\mathsf{cpath}})^b_C]\!] \overset{}{\underset{\text{def.}}{=\!=}} \mathtt{skip} \qquad\qquad (C \text{ not dynamic})$$

Most-derived object, or object with no virtual bases In this case, we statically know the offsets of subobjects of the objects. We first recursively define the $\mathsf{seqlist}$ operator, which turns a list of statements into a sequence of statements:

$$\mathsf{seqlist}(\epsilon) \underset{\text{def.}}{=\!=} \mathtt{skip} \qquad\qquad \mathsf{seqlist}(st :: stl) \underset{\text{def.}}{=\!=} st; \mathsf{seqlist}(stl)$$

Then, we remark that the set P_C of inheritance subobjects from C to dynamic classes and that are not primary subobjects of other subobjects of C, can be computed in a finite amount of time: the paths may be enumerated into a list.

Consider such a path $\sigma' \in P_C$. We statically know its offset within C (either because σ' is non-virtual, or because the operated subobject is a most-derived object), so that the following $S(\sigma')$ statement writes the corresponding pointer to the virtual table of σ':

$$\begin{aligned} S(\sigma') \underset{\text{def.}}{=\!=} {} & \underline{tmp} := \mathtt{vtable}\langle\sigma'\rangle_C(\overline{x_{\mathsf{cpath}}}) \\ & ; *_{\mathtt{Vptr}}(\overline{x} + \mathsf{soff}_C(\sigma')) := \underline{tmp} \end{aligned}$$

Finally, to implement "set dynamic type" operation, it suffices to update the pointers to virtual tables for all inheritance subobjects of σ which are not primary bases of other subobjects of σ:

$$[\![\mathtt{setDynType}(x, x_{\mathsf{cpath}})^{\mathsf{true}}_C]\!] \underset{\text{def.}}{=\!=} \mathsf{seqlist}(\mathsf{map}[S](P_C))$$

General case Assume that the subobject operated on is the σ inheritance subobject of D of static type C. Then, x_{cpath} actually holds the construction path (D, σ).

In the general case, the offsets of subobjects to dynamic class types are known at compile time with the notable exception of virtual inheritance. To this purpose, offsets to virtual bases must be read from the virtual table $(D, \sigma, (\mathsf{Repeated}, C :: \epsilon))$ of the C subobject σ considered as the most-derived object.

Assume that the pointer to this virtual table is already stored in the temporary variable $\underline{tmp_1}$ introduced by the compiler. Then, for each path $\sigma' \in P_C$, the following $S(\sigma')$ statement updates its dynamic type data to the corresponding (D, σ, σ') pointer to virtual table:

$$\begin{aligned} S(\mathsf{Repeated}, l) &\underset{\text{def.}}{=\!=} \underline{tmp_2} := \mathtt{vtable}\langle\sigma'\rangle_C(\underline{\overline{x}_{\mathsf{cpath}}}) \\ &\quad ; *_{\mathtt{Vptr}}(\overline{x} + \mathsf{nvsoff}(l)) := \underline{tmp_2} \\ S(\mathsf{Shared}, V :: l) &\underset{\text{def.}}{=\!=} \underline{tmp_2} := \mathtt{vtable}\langle\sigma'\rangle_C(\underline{\overline{x}_{\mathsf{cpath}}}) \\ &\quad ; \underline{tmp_3} := \mathtt{vboff}\langle V\rangle_C(\underline{tmp_1}) \\ &\quad ; \underline{tmp_3} := \overline{x} + \underline{tmp_3} \times 1 \\ &\quad ; *_{\mathtt{Vptr}}(\underline{tmp_3} + \mathsf{nvsoff}(V :: l)) := \underline{tmp_2} \end{aligned}$$

Finally, to implement "set dynamic type" operation, it suffices to first retrieve the virtual table of σ pretended as the "most-derived object", then update the pointers to virtual tables for all inheritance subobjects of σ which are not primary bases of other subobjects of σ:

$$\begin{aligned} [\![\mathtt{setDynType}(x, x_{\mathsf{cpath}})_C^{\mathsf{false}}]\!] &\underset{\text{def.}}{=\!=} \underline{tmp_1} := \mathtt{vtable}\langle(\mathsf{Repeated}, C :: \epsilon)\rangle_C(\underline{\overline{x}_{\mathsf{cpath}}}) \\ &\quad ; \mathsf{seqlist}(\mathsf{map}[S](P_C)) \end{aligned}$$

11.8.6.2 Correctness

Using P_C has two advantages. First, it allows to minimize the number of memory writes (in particular, if P_C is represented without duplicates, then we never write twice at the same place). Moreover, it makes easier to prove the:

Theorem II.19 (Dynamic type data update of an object and its subobjects). *Assume that x holds a pointer to an inheritance subobject σ of a most-derived D object, and that x_{cpath} holds the corresponding (D, σ) construction path.*

Then, $[\![\mathtt{setDynType}(x, x_{\mathsf{cpath}})_C^b]\!]$ updates the dynamic type data of all inheritance subobjects σ' of σ that are not primary base subobjects of other subobjects of σ to pointers to their corresponding virtual tables (D, σ, σ').

Indeed, it suffices to show that two such distinct subobjects σ'_1, σ'_2 have their dynamic type data disjoint, by using Theorem I.5 (p. 121), which requires to show that the reduced paths of $\sigma @ \sigma'_1$ and $\sigma @ \sigma'_2$ are distinct. The proof of this statement boils down to the:

Lemma 11.8.2. *Let (α, i, σ) be a generalized subobject of some type C:*

$$D[n] \prec\!\langle\alpha\rangle\!\overset{\mathcal{A}}{\succ\!\!\to} D'[n'] \prec\!\langle(i,\sigma)\rangle\!\overset{\mathcal{CI}}{\succ\!\!\to} C$$

Let $C \prec\!\langle(h_1, l_1)\rangle\!\overset{\mathcal{I}}{\succ\!\!\to} B_1$ and $C \prec\!\langle(h_2, l_2)\rangle\!\overset{\mathcal{I}}{\succ\!\!\to} B_2$ be two inheritance paths from C. Pose $(h'_i, l'_i) = \sigma @ (h_i, l_i)$ for $i \in \{1, 2\}$.

Then, $\mathsf{reducePath}(l'_1) = \mathsf{reducePath}(l'_2)$ *implies* $\mathsf{reducePath}(l_1) = \mathsf{reducePath}(l_2)$.

Proof. Let B be the class of the reduced path $\mathsf{reducePath}(l'_1)$:

$$B = \mathsf{last}(\mathsf{reducePath}(l'_1)) = \mathsf{last}(\mathsf{reducePath}(l'_2))$$

Then, Lemma 5.5.20 (p. 120) gives:

$$l'_1 = \mathsf{reducePath}(l'_1) @_{\mathsf{Repeated}} (B :: l''_1) \qquad l'_2 = \mathsf{reducePath}(l'_2) @_{\mathsf{Repeated}} (B :: l''_2)$$

with $B :: l''_1$ and $B :: l''_2$ primary inheritance paths from the same class B to some classes B_1 and B_2. Then, necessarily, one of them is a prefix of the other. By symmetry, assume $l''_2 = l''_1 @_{\mathsf{Repeated}} (B_1 :: l'')$ for some l'' such that $B_1 :: l''$ is a primary path from B_1 to B_2. Then, we actually have $(h'_2, l'_2) = (h'_1, l'_1) @ (\mathsf{Repeated}, B_1 :: l'')$, i.e.:

$$\sigma @ (h_2, l_2) = \sigma @ (h_1, l_1) @ (\mathsf{Repeated}, B_1 :: l'')$$

where $B_1 :: l''$ is primary. Then, Lemma 4.1.11 (p. 80) allows getting rid of the σ on the left-hand-side of @. This leads to (h_2, l_2) being a primary base class subobject of (h_1, l_1), so $h_1 = h_2$ by definition of path concatenation (Lemma 4.1.4 p. 76), and $\mathsf{reducePath}(l_1) = \mathsf{reducePath}(l_2)$ thanks to Lemma 5.5.21 (p. 120). □

However, it is also necessary to show that the dynamic types of irrelevant subobjects are not changed. The semantics of Ds++ erases those generalized data types of subobjects σ' such that σ is a Π-primary inheritance subobject of σ' for some notion of Π, which we choose here to match with the “primary path” notion of object layout:

Hypothesis 11.8.1. *A non-virtual inheritance path is* Π*-primary in the sense of Ds++ (Hypothesis 11.4.2 p. 252) if, and only if, it is a primary path in the sense of the object layout algorithm (Definition 5.5.10 p. 120):*

$$\Pi(l) = \mathsf{true} \Leftrightarrow \mathsf{isPrimaryPath}(l)$$

Thanks to our choice, we can now complete the proof of the correctness of the compilation of “set dynamic type”:

Theorem II.20 (Dynamic type data update for other objects). *Assume that x holds a pointer to an inheritance subobject σ of a most-derived D object, and that x_{cpath} holds the corresponding (D, σ) construction path.*

Then, $[\![\mathtt{setDynType}(x, x_{\mathit{cpath}})^b_C]\!]$ does not change the dynamic type data of inheritance subobjects σ' of D such that σ is not a primary base class subobject of σ'.

Indeed, it suffices to show that two such distinct subobjects σ', σ have their dynamic type data disjoint, by using Theorem I.5 (p. 121), which requires to show that the reduced paths of σ and $\sigma'2$ are distinct. The proof of this statement boils down to the:

Lemma 11.8.3. *Let (α, i, σ) be a generalized subobject of some type C:*

$$D[n] \prec\!\langle \alpha \rangle\!\overset{\mathcal{A}}{\mapsto} D'[n'] \prec\!\langle (i, \sigma) \rangle\!\overset{\mathcal{CI}}{\mapsto} C$$

Let $C \prec\!\langle \sigma_0 \rangle\!\overset{\mathcal{I}}{\mapsto} B_0$ and $C \prec\!\langle \sigma_1 \rangle\!\overset{\mathcal{I}}{\mapsto} B_1$ be two inheritance paths from B_1 such that $\sigma_1 = (h_1, l_1)$ is not an inheritance subobject of σ_0, and $\sigma_0 = (h_0, l_0)$ is not a primary inheritance subobject of σ_1.

Then, $\mathsf{reducePath}(l_0) \neq \mathsf{reducePath}(l_1)$.

Proof. Assume $\mathsf{reducePath}(l_0) = \mathsf{reducePath}(l_1)$. Then, l_0 and l_1 are two primary base class subobjects of the same subobject. In particular, they are non-virtual inheritance subobjects from the same class, so $h_0 = h_1$. Moreover, one is a primary base class subobject of the other, which contradicts the hypothesis. □

Here again, the hypothesis "a primary base class is non-virtual" plays an important role. Indeed, if we allowed sharing the virtual pointer of a class with one of its *virtual* base classes, then consider the following C++ example:

```
struct V1        {
   virtual void f();
};
struct V2        {
   int i;
};
struct B1: V1, V2 {};
struct B2: V1     {};
struct D: B1, B2  {};
```

If for instance `D`, `B1` and `V1` share their dynamic type data, then, when constructing `B2`, the dynamic type data of `V1` is updated, which overwrites the already constructed `B1`. The problem is that such data overwrite must preserve the access to `V2` from `B1`, which requires data related to `V2` in the construction virtual table of `V1` in `B2`, even though `V2` is totally unknown to the inheritance tree from `B2`.

11.8.7 Construction paths

The Ds++ operations on construction paths actually correspond in CVcm to the selection of a sub-VTT. However, sub-VTTs only exist for dynamic classes.

Root path The "root path" for some class D actually corresponds to retrieving the main VTT for D. So, there are two cases:
– If D is not dynamic, then simply retrieve a null pointer:

$$[\![x' := \texttt{rootPath}(D)]\!] \underset{\text{def.}}{=\!=} \overline{x'} := \texttt{NULL} \qquad (D \text{ not dynamic})$$

– Otherwise, retrieve a pointer to the main VTT of D:

$$[\![x' := \texttt{rootPath}(D)]\!] \underset{\text{def.}}{=\!=} \overline{x'} := \texttt{MainVTT}(D) \qquad (D \text{ dynamic})$$

Base path The "base path" for some class C to some class (β, B) actually corresponds to retrieving a sub-VTT from the VTT of C. Assume that, if $\beta = \mathsf{DirectNonVirtual}$, then $B \in \mathcal{DNV}(C)$, and if $\beta = \mathsf{Virtual}$, then $B \in \mathcal{V}(C)$. Then, there are two cases:
– If B is not dynamic, then simply retrieve a null pointer:

$$[\![x' := \texttt{basePath}(x, C, \beta, B)]\!] \underset{\text{def.}}{=\!=} \overline{x'} := \texttt{NULL} \qquad (B \text{ not dynamic})$$

– Otherwise, retrieve a pointer to the corresponding sub-VTT:

$$[\![x' := \texttt{basePath}(x, C, \beta, B)]\!] \underset{\text{def.}}{=\!=} \overline{x'} := \texttt{SubVTT}\langle(\beta, B)\rangle_C(\overline{x}) \qquad (B \text{ dynamic})$$

11.9 Correctness of the Ds++-to-CVcm compiler

We prove the correctness of the compiler by forward simulation (Theorem B.1 p. 340): we show that an invariant $s \rhd s'$ is preserved between an execution state s of Ds++ and an execution state s' of CVcm, finally obtaining Theorem II.16 (p. 284).

Let $s = (st, stl, e, \mathcal{K}, \mathcal{G})$ be a Ds++ state, and $s' = (st', stl', e', \mathcal{K}', \mathfrak{M})$ be a corresponding CVcm state.

Invariant 11.9.1 (Invariant). *The invariant $s \rhd s'$ is split into several parts:*

– *The statement and the statement list of s' are compiled from s*
– *An invariant $\mathcal{G}, b \vdash v \rhd_{\mathsf{Val}} v'$ holds between the values of s++ variables and the values of their corresponding Vcm variables.*
– *An invariant $\mathcal{G}, b \vdash \mathcal{K} \rhd_{\mathsf{Stack}} \mathcal{K}'$ relates the continuation stacks.*
– *An invariant $b \vdash \mathcal{G} \rhd_{\mathsf{global}} \mathfrak{M}$ relates the global state with the concrete memory state.*

where $b : \Lambda \to MemBlock^?$ is a partial function associating a CVcm memory block to every allocated Ds++ object.

$$\frac{\begin{array}{c} st' = [\![st]\!] \qquad stl' = \mathsf{map}[[\![\cdot]\!]](stl) \\ \forall x : e(x) \neq \bot \Rightarrow \mathcal{G}, b \vdash e(x) \rhd_{\mathsf{Val}} e'(\overline{x}) \qquad \mathcal{G}, b \vdash \mathcal{K} \rhd_{\mathsf{Stack}} \mathcal{K}' \qquad b \vdash \mathcal{G} \rhd_{\mathsf{global}} \mathfrak{M} \end{array}}{(st, stl, e, \mathcal{K}, \mathcal{G}) \rhd (st', stl', e', \mathcal{K}', \mathfrak{M})}$$

Contrary to s++, the b function may vary during execution. Its properties are stated more precisely in $\rhd_{\mathsf{global}}$.

11.9.1 Values

Values are related by the $\rhd_{\mathsf{Val}}$ relation, such that:

– A s++ built-in value is unchanged in Vcm:

$$\frac{}{\mathcal{G}, b \vdash Builtin \rhd_{\mathsf{Val}} Builtin}$$

– A valid pointer to a s++ subobject is related to a concrete pointer to the memory block corresponding to the complete object, under the offset corresponding to the generalized subobject:

$$\frac{\mathcal{G}.\mathsf{LocType}(\ell) = D[n] \qquad D[n] \prec\!\langle p\rangle\!\succ\!\to C}{\mathcal{G}, b \vdash (\ell, p) \rhd_{\mathsf{Val}} (b(\ell), \mathsf{off}_D(p))}$$

– A valid construction path to a dynamic class is related to the corresponding pointer to VTT:

$$\frac{D \prec\!\langle\sigma\rangle\!\succ\!\overset{\mathcal{I}}{\to} C \qquad \mathsf{isDynamic}(C) \ \Rightarrow v = (D, \sigma)}{\mathcal{G}, b \vdash (D, \sigma) \rhd_{\mathsf{Val}} v}$$

11.9.2 Continuation stack

An invariant $\mathcal{G}, b \vdash K \rhd_{\mathsf{Stackframe}} K'$ holds frame by frame:

$$\frac{}{\mathcal{G}, b \vdash \epsilon \rhd_{\mathsf{Stack}} \epsilon} \qquad \frac{\mathcal{G}, b \vdash K \rhd_{\mathsf{Stackframe}} K' \qquad \mathcal{G}, b \vdash \mathcal{K} \rhd_{\mathsf{Stack}} \mathcal{K}'}{\mathcal{G}, b \vdash K :: \mathcal{K} \rhd_{\mathsf{Stack}} K' :: \mathcal{K}'}$$

Statement blocks For a stack frame corresponding to an enclosing block, the CVcm statement list is compiled from the Ds++ statement list, and, if any, :

$$\frac{}{\mathsf{Block}(\bot, \mathit{stl}) \rhd_{\mathsf{Stackframe}} \mathsf{Block}(\bot\mathsf{map}[\llbracket\cdot\rrbracket](\mathit{stl}))}$$

$$\frac{\ell \neq \bot \qquad b(\ell) \neq \bot}{\mathcal{G}, b \vdash \mathsf{Block}(\ell, \mathit{stl}) \rhd_{\mathsf{Stackframe}} \mathsf{Block}(b(\ell), \mathsf{map}[\llbracket\cdot\rrbracket](\mathit{stl}))}$$

Call frames For a stack frame corresponding to a function caller, the CVcm list of statements to execute upon function return is compiled from its Ds++ counterpart. The values of Ds++ variables in the enclosing environment are matched in CVcm.

$$\frac{\forall x : e(x) \neq \bot \Rightarrow \mathcal{G}, b \vdash e(x) \rhd_{\mathsf{Val}} e'(\overline{x})}{\mathcal{G}, b \vdash \mathsf{Callframe}(x^?, \mathit{stl}, e) \rhd_{\mathsf{Stackframe}} \mathsf{Callframe}(\overline{x}^?, \mathsf{map}[\llbracket\cdot\rrbracket](\mathit{stl}), e')}$$

11.9.3 Memory

The invariant $b \vdash \mathcal{G} \rhd_{\mathsf{global}} \mathfrak{M}$ between the Ds++ global state and the CVcm memory state is composed of several parts:

11.9.3.1 Objects and memory blocks

- $b : \Lambda \to \mathit{MemBlock}^?$ is injective
- for any object ℓ defined in $\mathcal{G}$, $b(\ell)$ is well-defined
- if ℓ is a complete object of n cells of type D, then the corresponding $b(\ell)$ block must be large enough:

$$\mathcal{G}(\ell) = D[n] \Rightarrow \mathsf{blocksize}(b(\ell)) = n \times \mathsf{size}_D$$

11.9.3.2 Field values

For any complete object, the values of all its scalar fields are stored in concrete memory.

$$\frac{\begin{array}{c}\mathsf{LocType}(\ell) = D[n] \\ D[n] \prec\!\langle p\rangle\!\!\rightarrowtail C \qquad f = \texttt{scalar}\ T\ t \in \mathcal{F}(C) \qquad \mathcal{G}.\mathsf{FieldValue}((\ell, p), f) = v \neq \bot\end{array}}{\exists v' : \mathsf{load}(\mathfrak{M}, \llbracket T\rrbracket, b(\ell), \mathsf{off}_D(p) + \mathsf{foff}_C(f)) = v' \neq \bot \land v \rhd_{\mathsf{Val}} v'}$$

Whenever a field is written, Theorem I.3 (p. 115) ensures that the written field does not impact the values of other fields stored in concrete memory. Similarly, whenever dynamic data is written (because of the compilation of “set dynamic type”), Theorem I.4 (p. 119) ensures that the values of scalar fields are not overwritten in concrete memory.

11.9.3.3 Dynamic type data

Let ℓ be a complete object of type D. Let $p = (\alpha, i, \sigma_\circ @ \sigma)$ be a generalized subobject p of D of static type C such that class C is dynamic. Assume that the generalized dynamic type of p is $\sigma_\circ$, such that:

$$D[n] \prec\!\langle\alpha\rangle\!\!\rightarrowtail^{\mathcal{A}} D'[n'] \prec\!\langle(i, \sigma_\circ)\rangle\!\!\rightarrowtail^{\mathcal{CI}} C_\circ \prec\!\langle\sigma\rangle\!\!\rightarrowtail^{\mathcal{I}} C$$

Assume that σ is not a primary base subobject of another object (i.e. $\sigma = (h, l) = (h, \mathsf{reducePath}(l))$). Then the subobject contains a pointer to the virtual table $(D', \sigma_\circ, \sigma)$ corresponding to the subobject σ of $C_\circ$ where $C_\circ$ is considered as the most-derived object, but within an object of type D'.

$$\frac{\begin{array}{c} \mathcal{G} \vdash \langle \ell \rangle\ D[n] \prec\!\langle \alpha \rangle\!\overset{\mathcal{A}}{\rightarrowtail} D'[n'] \prec\!\langle (i, \sigma_\circ) \rangle\!\overset{\mathcal{CI}}{\rightarrowtail} C_\circ \prec\!\langle \sigma \rangle\!\overset{\mathcal{I}}{\rightarrowtail} C \\ p = (\alpha, i, \sigma_\circ @ \sigma) \qquad \mathcal{G}.\mathsf{gDynType}(\ell, (\alpha, i, \sigma_\circ @ \sigma)) = (\sigma_\circ, \sigma) \\ \mathsf{isDynamic}(C) \qquad \sigma = (h, l) \qquad l = \mathsf{reducePath}(l) \end{array}}{\mathsf{load}(\mathfrak{M}, \mathtt{Vptr}, b(\ell), \mathsf{off}_D(p)) = \&(D', \sigma_\circ, \sigma)}$$

Whenever a field is written, Theorem I.4 (p. 119) ensures that the dynamic type data are not overwritten in concrete memory. By contrast, whenever a “set dynamic type” operation is performed on some generalized subobject $\pi = (\ell, (\alpha, i, \sigma))$, the generalized dynamic types of several subobjects change. Theorem II.19 (p. 299) proves that the dynamic type data of σ and all its subobjects are correctly changed, whereas Theorem II.20 (p. 300) proves that the dynamic type data of other inheritance subobjects σ' such that σ is not a primary base class subobject of σ', are correctly kept unchanged.

Ds++ invariant for static cast To prove the correctness of the compilation of static cast, it is necessary to keep an invariant on the Ds++ source program:

Invariant 11.9.2. *Let $\mathcal{G} \vdash \langle \ell \rangle\ D[n] \prec\!\langle \alpha \rangle\!\overset{\mathcal{A}}{\rightarrowtail} D'[n'] \prec\!\langle (i, \sigma) \rangle\!\overset{\mathcal{CI}}{\rightarrowtail} C$ be a subobject of dynamic type $\sigma_\circ$:*

$$\mathcal{G}.\mathsf{gDynType}(\ell, (\alpha, i, \sigma)) = (\sigma_\circ, \sigma')$$

Then, $\sigma = \sigma_\circ @ \sigma'$ and $D' \prec\!\langle \sigma_\circ \rangle\!\overset{\mathcal{I}}{\rightarrowtail} C_\circ \prec\!\langle \sigma' \rangle\!\overset{\mathcal{I}}{\rightarrowtail} C$ for some class $C_\circ$.

11.9.4 Forward simulation

The correctness of the compiler is based on forward simulation, using Theorem B.1 (p. 340). The following theorem shows that forward simulation holds:

Theorem II.21 (Ds++ to CVcm forward simulation). $\rhd$ *is a* forward simulation *from Ds++ to CVcm:*

– *Initial states match:*

$$\forall s_\circ \in \mathfrak{I} : \exists s'_\circ \in \mathfrak{I}', \exists s'_1 \in \mathfrak{S}' : s'_\circ \overset{\star}{\to}{}' s'_1 \wedge s_\circ \rhd s'_1$$

– *Forward invariant preservation:*

$$\begin{aligned} \forall s_1, s_2, {s_1}' : &\quad s_1 \to s_2 \wedge s_1 \rhd {s_1}' \\ \Rightarrow \exists {s_2}' : &\quad {s_1}' \overset{+}{\to}{}' {s_2}' \wedge s_2 \rhd {s_2}' \end{aligned}$$

– *Final states match:*

$$\begin{aligned} \forall (s_f, z) \in \mathfrak{F}, \forall s'_9 \in \mathfrak{S}' : &\quad s_f \rhd s'_9 \\ \Rightarrow \exists s'_f \in \mathfrak{F}'_z : &\quad s'_9 \overset{\star}{\to}{}' s'_f \end{aligned}$$

Graphically:

$$\begin{tikzcd}
\mathbb{S} & \mathfrak{I} \arrow[rr, dash, "\ni"] & & s_\circ \arrow[d, dotted, dash, "\rhd"] & s_1 \arrow[r, "\mathfrak{e}^?"'] \arrow[d, dash, "\rhd"] & s_2 \arrow[d, dotted, dash, "\rhd"] & s_f \arrow[rr, dash, "\in"] \arrow[d, dash, "\rhd"] & & \mathfrak{F}_z \\
\mathbb{S}' & \mathfrak{I}' \arrow[r, dotted, dash, "\ni"] & s'_\circ \arrow[r, dotted, "'\star"] & s'_1 & s'_1 \arrow[r, dotted, "'+", "\mathfrak{e}^?"'] & s'_2 & s'_{99} \arrow[r, dotted, "'\star"] & s'_f \arrow[r, dotted, dash, "\in"] & \mathfrak{F}'_z
\end{tikzcd}$$

Proof. We sum up the theorems and lemmata, as well as the relevant parts of the invariant, used for the proof of invariant preservation for the most interesting steps:

s++ execution step	**Proof case**	**Theorem used**
Scalar field write (s++-field-scalar-write, p. 86) (unchanged since s++-to-Vcm)	Success: alignment	Theorem I.1 (p. 110)
	Success: in bounds	Theorem I.2 (p. 112)
	Good variable property wrt. fields	Theorem I.3 (p. 115)
	Good variable property wrt. dynamic type data	Theorem I.4 (p. 119)
Pointer equality test (s++-ptreq, p. 87) (unchanged since s++-to-Vcm)	Pointers of non-empty class type	Theorem I.7 (p. 122)
	Pointers of empty class type	Theorem I.8 (p. 124)
Static cast (Ds++-statcast, p. 251)	Dynamic type data	Invariant 11.9.2 (p. 304) (Ds++ only, independent of the CVcm compiled program)
	Access to virtual base (unchanged since s++-to-Vcm)	Theorem I.11 (p. 147)
Dynamic cast (Ds++-dyncast, p. 251)		Theorem II.17 (p. 294)
Virtual function call (s++-virtual-funcall, p. 91)		Theorem II.18 (p. 295)
Set dynamic type (Ds++-setdyntype, p. 253)	Same object and its subobjects	Theorem II.19 (p. 299)
	Other objects	Theorem II.20 (p. 300)
	Scalar fields unchanged	Theorem I.4 (p. 119)

□

Chapter 12

Discussion

In this chapter, we give a technical overview of our Coq development. Then, we position our work among earlier work about object construction and destruction. Then, we compare with other object-oriented languages. Finally, we investigate perspectives for extending our formalism.

12.1 The Coq development

To express the semantics of κ++, we mostly use inductive types, rather than an executable semantics. However, for well-formed class hierarchies, predicates such as knowing whether a class is dynamic are decidable, and sets such as virtual function dispatch candidates are computable in a finite amount of time, which is actually needed to populate virtual tables and virtual table tables during compilation.

The main drawback of our development is the lengthy proof of the κ++ run-time invariant: it accounts for half of the entire development, and needs an estimated $2\frac{1}{4}$ hours of proof checking on a 2 GHz Intel Pentium IV consuming about half of the 4 Go of RAM. After some tests, it turns out that this lengthy time for proof checking might be due to the definition unfolding system of Coq during type unification.

On the contrary, once this invariant is proved, theorems related to object lifetime and resource management are advantageously clear to state and easy to prove. This shows that the invariant is strong enough, and that the burden of the proof is only related to the complexity of the underlying semantics, and not to the high-level notions of object lifetime or resource management expected by programmers. This is illustrated by the following detailed proof sizes:

Theories	**Specs loc**	**Proofs loc**
κ++ language	895	144
κ++ invariant	693	81
Invariant preservation	324	13154
Object lifetime	2296	6306
Ds++ language	814	651
κ++ to Ds++	1577	6781
CVcm language	445	0
Ds++ to CVcm [1]	1822	5780
Total	**8866**	**32897**

12.2 Related work

12.2.1 C++ object construction and destruction

There have been very few works on the formalization of C++ object construction and destruction. In his Ph.D. thesis, Wasserrab [84] models object construction and destruction not in a formal way, but through an unproven algorithm (which Wasserrab credits to Frank Tip) transforming constructors and destructors to special functions that construct and destruct an object and its subobjects by explicitly calling their corresponding “constructor/destructor” functions, similarly to our κ++-to-Ds++ compiler pass of Section 11.5 (p. 254). However, they differ on their treatment of virtual function calls during construction and destruction. Indeed, whereas our compiler does not change such calls, Wasserrab's unproven algorithm additionally transforms, at compile time, direct calls to virtual functions from within constructor bodies, into the appropriate non-virtual calls, thus statically solving function dispatch, such as in the following example, where this original C++ code:

```
struct A {
  virtual void f();
  A();
};

/* constructor for A */
A::A() {
  this->f();       /* must call A::f, not B::f */
}

struct B: A {
  void f();
  B() {}
};
```

is correctly transformed by Wasserrab's unproven algorithm to:

```
struct A {
  virtual void f();
```

1. Except object layout

```
};

/* "Constructor" for A */
void constr_A(A* this) {
  this->A::f();   /* non-virtual function call, bypasses inheritance:
                     calls A::f instead of B::f */
}

struct B {
  void f();
};

/* "Constructor" for B */
void constr_B(B* this) {
  constr_A(static_cast<A*>(this));
}
```

However, this transformation fails to capture indirect calls to virtual functions, i.e. calls from functions previously called from a constructor body, as in the following example, where this original C++ code:

```
struct A {
  virtual void f();
  A();
};

void g(A* a) {
  a->f();
}

/* Constructor for A */
A::A() {
  g(this);  /* indirect call to f: must call A::f, not B::f */
}

struct B: A {
  void f();
  B() {}
};
```

is erroneously transformed by Wasserrab's unproven transformation to:

```
struct A {
  virtual void f();
  A();
};

void g(A* a) {
  a->f();   /* full dispatch */
}

/* Constructor for A */
void constr_A(A* this) {
  g(this);  /* indirect call to f with full dispatch:
               for a B instance, erroneously calls B::f
               instead of expected A::f */
}

struct B: A {
  void f();
};

/* Constructor for B */
void constr_B(B* this) {
  constr_A(static_cast<A*>(this));
}
```

The most complete formalization of C++ object construction and destruction so far is the work by Norrish [64], presenting a full-fledged semantics of C++ in HOL4. This semantics covers C++ object construction and destruction, including temporary objects (an issue we do not tackle, as discussed below): as such, Norrish's semantics is very close to the C++ Standard. However, this semantics is based on "on-the-fly" program transformations: a constructor call with its sequence of initializers is transformed into an ordinary statement sequence, losing the logical relations between the components (bases and fields) to construct. Thus, it would be difficult to reason about the construction order of subobjects.

Moreover, we found an inaccuracy in the formalization of indirect virtual function calls during and after object construction. In Norrish's work, pointers to subobjects during construction are not the same as pointers to subobjects of a constructed object: pointers carry the path to the object considered as "most-derived object" for the purpose of function call. However, consider the following C++ code:

```
struct A {
  virtual void f();
};
A* amem;
void g(A* a) {
  if(amem == nullptr) {
    amem = a;
  }
  amem->f();
}
struct B: A {
  B(): A() {
    g((A*) this);
  }
};
struct D: B {
  virtual void f();
  D(): B() {
    amem->f();
  }
};
main () {
  D d;
}
```

When constructing an instance of `D`, the `B` constructor calls `g` with a special pointer to the `A` subobject considering that the most-derived object is `B`. `g` memorizes this argument to `amem`. When entering the constructor for `D`, the `D` subobject will be considered the most-derived object, but `amem` will not change, so that `amem->f()` incorrectly calls `A::f()` instead of the expected `D::f()`. Our semantics of κ++ shows that the notion of which object should be considered the "most-derived object" for polymorphic operations, the notion of "generalized dynamic type", is a notion related to the object itself (rather than to pointers).

12.2.2 Safety of object initialization

Object initialization has been mechanically formalized in order to ensure that all fields and bases of an object are correctly initialized, i.e. that programs never access uninitialized parts of objects, or that initialization leaves no uninitialized fields behind. Such safety properties have been mostly studied for object-oriented languages featuring single inheritance only, such as Java or C♯.

Programmers often make use of a naive technique to ensure safe field initialization: they preinitialize all fields of an object to `null` before determining the actual values to use for initialization. Fähndrich et al. [32] developed a type system to determine which of those `null` initializations are useless, and to remove them.

Qi et al. [68] developed a type system for Java to statically determine at compile time which fields may be read or not, at precise program points. For the same purpose, Hubert et al. [39] formalized in Coq a type system for Java, which, similarly to our work, makes use of construction states for objects, but lifted to the level of types and maintained at compile time. Although those type systems are restricted to Java single-inheritance object model, we suspect that the semantics of our κ++ language could be equipped with a similar type system to statically reason about the safety of C++ object initialization.

12.3 Comparison with other languages

Most object-oriented languages extensively deal with object initialization through various models of object construction. However, those models are often inadequate for resource management, as those languages mostly do not feature equally precise destruction mechanisms.

Until recently, Java only featured *object finalization*. The `Object` class (from which every class directly or indirectly inherits) is equipped with a `finalize()` method, which the Java specification [37] requires to be called once by the runtime system, before the object is deallocated from memory by the garbage collector. However, the specification leaves undetermined when precisely this method is called, so that, in particular, there is no guarantee about the order in which two objects are finalized. Thus, finalization is inadequate to model the disposal of two objects a and b if the lifetime of a depends on the lifetime of b (e.g. a file and a lock on its device).

In July 2011, Java 7 introduced the *try with resource* construct. This language feature, similar to `using` in C♯ (already present in C♯ 1.0 since 2001), features a mechanism of *object disposal*. The following example creates a file named `toto` and writes `Hello world` to this file. In Java 7:

```
try (FileOutputStream file = new FileOutputStream("toto")) {
  file.writeln("Hello world");
}
```

In C♯:

```
using (TextWriter file = File.CreateText("toto")) {
    file.WriteLine("Hello world");
}
```

Internally, those mechanisms attach a resource to a statement block. This resource must be of a class type implementing a specific interface for resources (`Closeable` in Java, `IDisposable` in C♯), featuring a disposal method (`close()` in Java, `Dispose()` in C♯), which is actually called once the statement block exits. This also guarantees that an object declared in an enclosed statement block is disposed before the object of the enclosing block, thus implementing a form of RAII, in particular if an uncaught exception is thrown.

However, those disposal mechanisms do not take inheritance into account. If, for instance in Java, the programmer replaces `FileOutputStream` with the following class:

```
public class File extends FileOutputStream {
  public void close() {}
}
```

Then, the `close()` method of `File` is called. This method actually does nothing, and in particular nothing constraints it to call the `close()` method of the parent class: in practice, the physical file is not closed. In other words, the Java and C♯ languages do not enforce the destruction of base class subobjects.

12.4 Future work

12.4.1 Extending the semantics of κ++

The semantics of κ++ can be extended in a number of directions towards a more realistic C++ language.

12.4.1.1 Manual memory management

Free store Our κ++ language only features objects with a lifetime controlled by statement blocks. However, C++ also features a *free store*, which allows to create objects on the fly through the `new` language construct, without attaching their lifetime to statement blocks. When an object is created in such a way, its constructor is called.

Most object-oriented languages feature a free store, more often called a *heap*, which is mostly garbage-collected, contrary to C++[2], where only the programmer controls the lifetime on such objects, by explicitly requesting their destruction and deallocation through `delete`.

```
int* i = NULL;
{
  int* j = new int[10]; /* creates an array of 10 integers */
  i = &j[2];
};                      /* j is not destructed here */
...
delete &i[-2];          /* &i[-2] == &j[0] == j,
                           thus destructs the entire array */
```

The programmer can use `delete` on a pointer to any object, which may be an inheritance subobject of another object. In such case, the most-derived object must be destructed. To ensure that the destructor of the most-derived object and all its subobjects will be called, C++ requires that the destructor be *virtual*.

```
struct B {
  virtual ~B();
};

struct D: B {};

D* d = new D();
B* b = d;
...
delete b;        /* actually destructs d */
```

Adding a free store would not significantly change the semantics of object construction and destruction. It would only require additional rules for object allocation and deallocation from the free store. However, from the point of view of proofs, it is not possible to reason on the lifetimes of free store objects. Thus, the free store should be disjoint from the stack.

Moreover, the compilation of array construction and destruction would have to change, as the number of cells is known only at run time in such cases. In practice, the Common Vendor ABI [22] provides a function to iterate construction and destruction over all cells of an array.

2. The new C++11 Standard [43] allows garbage-collected implementations, but the programmer still has control on free store object lifetimes through `delete`.

Explicit destructor call and object placing In addition to the free store, C++ provides tools for fine-grained memory management: the programmer can explicitly provide the memory location where to create an object. This can be useful on devices with a limited amount of memory. However, very few guarantees help the programmer find out whether the memory location is available and not occupied by another object. To offer such a guarantee, C++ allows the programmer to directly call the destructor on an object. Such direct destructor calls indicate that the memory occupied by the object of type T can be reused to create an object of type T' as long as $\texttt{sizeof}(T') \leq \texttt{sizeof}(T)$.

```
struct A {
  int   i;
};

struct B {
  float f;
};

A* a = new A();
...
a->~A();           /* destructs A without deallocating */
B* b = new(a) B(); /* constructs B at the location of the old A
                      without allocating new space
                      possible because sizeof(B) <= sizeof A() */
```

To add such features to κ++ would require a more precise memory model, in particular taking object layout into account.

12.4.1.2 Temporary objects

Our languages specified and formalized in this thesis are 3-address languages. On the contrary, C++ features embedded complex expressions, which often leads to the creation and destruction of unnamed objects, not attached to any variable. Those objects are called *temporary objects*.

Passing temporary objects as constructor arguments In C++, a temporary object can be passed to a function, as in the following example creating an unnamed instance of `A` passed to `f`:

```
struct A    {...};
void f(const A& a) {...};

f(A());
```

This can be modelled in κ++ using the ordinary mechanism of block-scoped objects:

```
void f(A* a) {...}

{
    A a[1];          // construct the temporary object
    pa = &a[0];
    f(pa);
}                    // destruct the temporary object
```

However, such reasoning is not possible for constructor calls: any object created within an initializer must have been destructed before calling the constructor. This prevents passing pointer to such temporary objects to the constructor, as in the following C++ example:

```
struct A {
  A(int i) {...}
};

struct C {
  C(const A& a) {...}
};

main() {
  C c(A(1));
  ...
};
```

In this example, the following objects are constructed and destructed:

1. An unnamed instance of `A`, said to be a *temporary object*, is created and constructed.
2. An instance `c` of `C` is created, and constructed with the constructor of `C` given a reference to the temporary `A` instance.
3. Then, the C++ Standard [42, 43] mandates that, when the *full expression* `C c(A());` has finished execution, all created temporaries must be destructed. This requires the temporary `A` to destruct at this point.
4. Then, once the temporary is destructed, the remaining part of the body of `main()` is executed.
5. Finally, upon exit, the `c` instance of `C` is destructed.

This example illustrates that the temporary `A` object is destructed right upon the end of the constructor of `c`. This is not covered by our semantics: once a constructor exits, no more operation is allowed within the initializer from which the constructor has been called. In practice, if the `main()` function were naively expressed in κ++, then κ++ would have required the temporary

`A` to be destructed before the `C` constructor call:

```
{
  $C\ c[1]$
  {                          // Initializer for $c[0]$
     $A\ tmp[1]$;            // Temporary object
     $ptmp = \&tmp[0]$;
     $C(ptmp)$;              // in $\kappa$++, $tmp$ must have been destructed
                             // before this constructor call
  };
  ...                        // body of main()
}
```

In κ++, when an initializer calls the constructor, a Kconstrother or Kconstrothercells frame is added on top of the continuation stack. This frame instructs subsequent sibling subobjects to start their construction once the constructor exits, thus disallowing any further action at the level of the initializer for the object being constructed. One solution to allow the destruction of temporaries could be to add a further argument to Kconstrother and Kconstrothercells, namely the list of temporary objects not yet destructed when the constructor is called. Then, upon constructor exit, the destruction of all those temporary objects is performed before starting the construction of subsequent sibling subobjects.

This example also illustrates that the lifetime of the temporary is not included in the lifetime of `c`. As a consequence, adding temporaries would invalidate theorems about the construction order of subobjects of different complete objects. Actually, the C++ Standard does not specify the relations between the lifetime of an object explicitly named by the programmer and the lifetime of a temporary object. However, the Standard mandates that two temporary objects constructed in the same context — actually, in the same *full expression*, until the next `;` — are destructed in the reverse order of their construction:

```
struct C {
  C(const A& a1, const A& a2) {...}
};

main() {
  C c(A(1), A(2));
  ...
};
```

The Standard does not mandate the order in which the two temporaries `A(1)` and `A(2)` are constructed, but it imposes `A(2)` to destruct before `A(1)` if, and only if, `A(1)` was constructed before `A(2)`. It could be interesting to reason about the construction order of two temporary objects used for the purpose of the same constructor call. However, this would require to precisely define the notion of temporary context, as the construction and destruction order of two “unrelated” temporary objects is unspecified. One idea to define such a context is to state that two temporary objects belong to the same context if, and only if, they belong to the same list of objects to deallocate upon constructor return, i.e. within a Kconstrother or Kconstrothercells construction stack frame.

Call by value: implicit argument copy Another context where temporary objects are needed is when a function expects its arguments to be given *by value*. Consider the following C++ example:

```
struct A {};
void f(A a) {
  A(const A&); /* copy constructor */
};

A a0;
f(a0);          /* A copy of a0 is passed to f.  This copy is
                   implicitly created using the copy constructor,
                   and implicitly destructed once f returns. */
```

Actually `f` does not receive a reference to `a0`, but to a *copy* of `a0`, obtained by creating a new instance of `A` using the *copy constructor* `A(const A&)`. This can be modeled by our κ++ language; however, call by value becomes a property of the function call, but no longer of the function itself:

```
struct A {
  A(A*);                                /* copy constructor */
};
void f(A* a) {...};

A a0[1];
{
  A copya0[1] = { pa = &a0[0]; A(pa) }; /* explicit argument copy */
  pcopya0 = &copya0[0];
  f(pcopya0);
}                                       /* copy destructed upon block exit */
```

Consequently, if κ++ is extended to allow temporary objects as constructor arguments, then such an extension will also model constructors expecting arguments passed by value following this pattern.

Functions returning structures Once temporary objects are added to the semantics of κ++, it would then be possible to further extend κ++ by allowing functions returning structures. Consider the following example:

```
#include <cstdio>

struct C {
  int i;
  C(int i0) : i(i0) {}
};
C f(int i) {
  C c1 = C(i);
  C c2 = C(-i);
  if(i >= 0) {
    return c1;
  } else {
    return c2;
  }
};
main () {
 printf("%d\n", f(-42).i);
};
```

Roughly speaking, in C++, when such a function is called, the following steps are taken:

1. The caller allocates temporary space to hold the return value of the function.
2. Then, the caller calls the function (here `f`), passing an additional "hidden" argument to indicate to the function where to construct the return value.
3. The callee executes normally (here, two instances of `C` are created and constructed: `c1` then `c2`).
4. When the callee encounters a `return` statement, it first evaluates the expression to return, then this result is *copied* into the temporary space for return value: actually, the *copy constructor* `C(const C&)` is *implicitly* called to construct an object in the temporary space, and the argument to the constructor is the computed value. (Here, `C(c2)` is called, to create a copy of `c2`.)
5. Once the constructor exits, all objects allocated and created within the callee are destructed (here `c2` then `c1`), then the callee exits.
6. Once the callee has exited, the caller uses the temporary object (in this example, its field `i` is read).
7. Once the full expression involving the temporary has finished its execution (here, the call to `printf`), the temporary is destructed and finally deallocated.

The following pseudo-C code illustrates how such a program would be compiled:

```
struct C {
  int i;
};
void constr_C(C* target, int i0) {
  target->i = i0;
}
void copy_C(C* target, C* source) {...} /* corresponds to copy constructor
                                           C(const C&)
                                           the compiler has to generate one
                                           if the programmer has provided none */
void f(C* target, int i) {
  C c1;
  constr_C(&c1, i);
  C c2;
  constr_C(&c2, -i);
  if(i >= 0) {
    copy_C(target, c2);
    return;
  } else {
    copy_C(target, c1);
    return;
  }
}
```

To extend the semantics of κ++, we could follow this scheme and consider that a function "returning" a structure is actually a function initializing an object. As such, a call to a function returning a structure could occur in lieu of a constructor call, within an initializer; conversely, such function call could only occur in an initializer.

More precisely, consider the initializer for a most-derived object of class C. Then, during this initializer, a Kconstrarray frame is present on top of the continuation stack. This stack frame indicates the array cell being constructed, so that if the initializer calls a function to construct the array cell, then in practice, the location of the object to construct is given by the Kconstrarray stack frame. Upon such a call, the function body is then itself an initializer, expected to call the constructor, or again another function returning a structure. However, the temporary objects used for the function call must be destructed once the construction of the cell ends. To ensure this, it is necessary to equip Kconstrarray with the list of objects to destruct, so that this list will be transferred to Kconstrothercells once the constructor is called. This extension could be also applied to base class subobjects. Thus, our example could be modelled in an extension of κ++ as follows (roughly speaking):

```
struct C {
  int i;
  C(int i0) : i(i0) {}
  C(C* c)   : i(..) {...}
                    /* copy constructor */
};
C f(int i) {
  C c1[1] = { C(i); };
  pc1 = &c1[0];
  C c2[1] = { j = -i; C(j); };
  pc2 = &c2[0];
  if(i >= 0) {
    C(pc1);         /* "return" by constructor call */
  } else {
    C(pc2);         /* "return" by constructor call */
  }
};
main () {
  {
    C tmp[1] = { i0 = -42; f(i0); };
                   /* temporary is created,
                      initialized by f */
    ptmp = &tmp[0];
    s = "%d\n";
    j = ptmp->i;
    printf(s, j);
  }                /* temporary is destructed */
};
```

This is an example in an "extended κ++", where all constructor calls are explicit. If this code were executed in real C++, copy constructor calls would be inserted according to the Standard. The following section discusses this issue.

Copy elision: return value optimizations When a function returns a structure, the computed result is copied into the temporary space allocated by the caller, through a call to the "copy constructor". This is the reason why the semantics of C++ temporaries is called the *copy semantics*. One question raised by programmers and compiler developers is the following: in which cases can calls to copy constructors be *elided*? In other words, is it possible to construct the result directly in the designated memory space, thus avoiding temporaries?

Return value optimization Consider the following C++ code:

```
C c = f(18);
```

The Standard prescribes that a temporary object be created to hold the value of the *full expression* `f(18)`, actually the return value of `f`. Then this temporary has to be again copied to the final memory space of `c`. However, is it possible to not allocate temporary memory space, and to directly tell `f` to construct its return value into the final space allocated for `c`? In fact, the C++03 Standard [42] also allows this behaviour, called *return value optimization* (RVO). However, this actually makes the semantics of C++ non-deterministic, as those two behaviours are not always equivalent (in particular, if for instance the copy constructor changes the value of an object field). Thus, such program transformation is not really an "optimization", in the sense that it is not possible to prove that a compiler performing such "optimization" would preserve the semantics of programs.

However, an extended version of κ++ as modified for functions returning structures, could deal with RVO. The following "extended-κ++" code models the example without RVO:

```
C c[1] = {
  C tmp[1] = { i=18; f(i); }; /* temporary introduced by function call */
  ptmp = &tmp[0];
  C(ptmp);                    /* temporary copied into final memory space */
}
```

The following "extended-κ++" code models the same example with RVO:

```
C c[1] = { i=18; f(i); }; /* f directly constructs into final memory space */
```

Thus, RVO can be seen as a program transformation independent of "extended-κ++".

Named return value optimization Consider the following C++ code:

```
C g() {
  C c;
  ...
  return c;
}
printf("%d", g().i);
```

Then, the following objects are constructed and destructed:

1. Temporary memory space is created to hold the result of `g()`.
2. `g()` is called.
3. Within `g`, memory space for `c` is allocated.
4. The constructor `C()` is called to construct `c`
5. Upon `return`, `c` is copied into the temporary return space.
6. `c` is destructed and deallocated.
7. `g()` is exited.

However, if all `return` statements of `g` return the same object reference (here`c`), then C++ allows to construct `c` directly in the temporary return space, yielding the following sequence of object constructions and destructions:

1. Temporary memory space is created to hold the result of `g()`.
2. `g()` is called.
3. Within `g()`, `c` directly refers to the temporary return space, with no further memory allocation.
4. The constructor `C()` is called to directly construct the return value `c`.
5. `g()` is exited.

An extended version of κ++ would have to allow further operations on `c` within `g` after it has been actually constructed, but before it is returned. So, we would have to show that those further operations would not change the construction states of `c` or its subobjects.

12.4.1.3 Unifying built-in types with structure types

Our semantics of construction and destruction treats scalar fields separately from structure fields. However, it could be possible to unify the semantics of construction and destruction of scalar and structure types, by considering that a scalar type is actually a type whose all values already exist: as such, the set of object locations would be separated between a finite set of user-allocated objects, and an infinite set of “scalar” values.

12.4.2 Compiler optimizations

12.4.2.1 Concrete VTT layout

Our CVcm language leaves abstract the representation of virtual table tables. In particular, it is independent of whether a VTT contains its sub-VTTs or only pointers to sub-VTTs. The same question can be asked for construction virtual tables managed by VTTs.

VTT sharing The Common Vendor ABI for Itanium [22] prescribes sharing virtual table tables: in the case of a non-virtual base that has no virtual bases, its main VTT may be used instead of the corresponding construction VTT.

In practice, such sharing of virtual table tables could be done at the level of CVcm, by recognizing that the contents of a virtual table table is included in the contents of another, independently of the class hierarchy, which no longer exists in CVcm. However, the Common Vendor ABI prescribes such sharing early depending on the class hierarchy, which still exists. To systematize this sharing would require to show, at the level of class hierarchies, that some construction path shall always produce virtual table table contents included in the virtual table table of some other construction path.

However, this actually depends on the actual implementation of dynamic casts. Instead of our abstract formalization, dynamic cast could be implemented by a two-step process: first look for a dynamic cast offset in the virtual table of the current subobject, then, if none found, fallback to reading the virtual table of the “most-derived” object (for the purpose of polymorphic operations, i.e. the generalized dynamic type). Casting back to the most-derived object could be done thanks to a further offset in the virtual table.

12.4.2.2 Elision of trivial constructor/destructor

Our compiler systematically compiles all constructors and destructors. However, this incurs a time overhead for PODs (Plain Old Data, roughly C structures with no inheritance and no non-POD fields) and other structures which have no user-defined constructors or destructors. Actually, the construction and destruction of such structures produce no side effects. Thus, an optimized compiler could eliminate the corresponding constructor/destructor calls.

For non-dynamic classes such as PODs, this issue is related to the more general question of function inlining, so that such an optimization could occur at the level of CVcm, once all object-oriented features have been compiled. However, for dynamic classes requiring to update dynamic type data, such an optimization would have to consider aliasing, showing that intermediate trivial constructor calls for subobjects are useless, as they produce no side effect other than updating dynamic type data which will be anyway overwritten by constructors for derived class objects.

Part ∞

In closing

Chapter 13

Conclusion and perspectives

In this chapter, we give an overall assessment of our work. We comment on our experiment with the Coq proof assistant. We outline the practical and potential impacts of our work. Then, we investigate more general directions to extend our work towards a full-fledged C++. Finally, we generalize the conclusions of our work to the wider topic of formal verification.

13.1 Assessment

13.1.1 The Coq experiment

At first sight, formalizing the C++ language with a proof assistant such as Coq [4] might seem frightening and highly discouraging, due to the alleged complexity of the C++ object model and the large amount of details to deal with during formalization: nothing can be left as a triviality to the reader.

In fact, this is not the case, as the development proper to object construction and destruction, mainly the set of κ++ semantic rules, is no more than 900 Coq lines long. We even believe that our semantics could be made shorter if we adopted a more general point of view than objects with inheritance (e.g. by unifying the construction of scalars with the construction of objects, which would allow scalar arrays). This shows the tractability of the Coq specifications, thanks to the Gallina specification language, which is clear and relies on precise mathematical backgrounds such as the Calculus of Inductive Constructions.

By contrast, at the level of the proofs, the scripts amount an estimated 80000 lines of code altogether, often look quite repetitive (we believe that they could be revamped by recognizing and factorizing into tactics some proof patterns such as symmetry reasonings), and take more than 2.5 hours to recheck using `coqc` on a 2 GHz Pentium Core Duo consuming half of the 4 GB RAM. After some informal tests, we claim that this issue might be due to Coq-specific shortcomings about definition unfoldings during type unification. However, as an explanation to those apparently huge figures, we have deliberately chosen to limit the use of proof automation, actually restricted to propositional or first-order logic (using the `tauto` tactic), or integer arithmetics (`omega`): none of our lemmata are integrated into any `Hint` automation databases. The purpose of our choice is to understand which and when our lemmata are reused in proofs, enabling us to explain them at a high level, which led to the redaction of this thesis.

13.1.2 Practical impact of our work

Our work allowed us to find and report errors and inconsistencies in the C++03 standard [42]. Virtual function calls are allowed during the construction of the data members of an object, but, surprisingly, the Standard lacked symmetry by leaving unspecified virtual function calls during their destruction. This issue has been corrected [45] in the C++11 standard [43]. The following other issues have been submitted to the C++ Standards Committee and are planned to integrate a future version of C++: the lifetime of an array is not considered to be included in the lifetime of its cells, violating the high-level principle stating that the lifetime of an object is included in the lifetime of all its subobjects; and the lifetime of objects of built-in types is not ended by explicitly calling their destructor, contrary to objects of compound (e.g. structure) types.

From the practical point of view, our work also allowed us to explain some known bugs in real-world C++ compilers such as Microsoft Visual C++ 7.0 or Borland C++ Builder 5.x [38]. Those bugs are violations of the subobject identity requirement in the presence of empty bases and members.

Finally, we have proved a layout algorithm covering almost all of the widely used Common Vendor ABI. The only omission is the controversial virtual primary base optimization, being dubbed as “an error in the design” of the Common Vendor ABI, as stated a posteriori by its development consortium.

13.1.3 Potential impacts of our work

Besides the immediate effects on the C++ Standard, our work is valuable as an alternative description of C++ to help understand the C++ object model. Our formal semantics validates a posteriori the object-oriented design principles of C++ and the requirements of the Standard such as the subobject identity principle (to formally account C++ non-virtual inheritance), or virtual function dispatch during construction (to correctly model the RAII resource management principle). Moreover, our approach based on verified compilation gives sound foundations for implementation techniques commonly used in most modern-day C++ compilers (including GNU GCC) such as empty base optimization, virtual tables and virtual table tables.

As a formal description of an object-oriented subset of C++, we believe that our work can serve as a basis for applying formal methods to C++ programs. Thanks to our formal semantics, a promising approach could be static analysis of programs by abstract interpretation [23]. Such a method would rely on a straightforward abstract interpreter directly implementing the κ++ semantic rules. By contrast, applying our formal semantics to deductive program verification a la Frama-C [5] or Why [8] could need more work than via abstract interpretation. Research directions include developing a Hoare-like logic for multiple inheritance. Such a logic could be based on the separation logic by Luo et al. [56] to reason about field accesses in the presence of multiple inheritance. Based on such logical systems, we hope that our work will help find a way of specifying preconditions and postconditions of constructors and destructors and of virtual functions during construction or destruction, maybe expressed in terms of our subobject construction states.

13.2 Future work

There is only so much that can be done during a Ph.D.; a full formalization of C++ cannot. However, our work can be extended in a number of directions towards the whole C++ language, by including references, constness, accessibility (public/private), exceptions, templates, concurrency. The work by Norrish [64] could be a solid base for extending our work. A first realistic step would be to cover the subset of C++ described by the guidelines edicted by Lockheed Martin [55].

13.2.1 Exceptions

Exceptions are one of the hallmarks of C++, which has been one of the first languages to feature and to efficiently implement them, so that they have become an idiom in contradiction with their name: exceptions are not exceptional in a real-world C++ program.

In C++, a program can throw any value to break the execution of a statement. For instance, the following program computes the greatest common divisor of two integers, using the formula:

$$\gcd(a, b) = \gcd(b, a \mod b)$$

The program executes until $a \mod b$ fails (when $b = 0$, in which case a is thrown).

```
int modulo(int a, int b) {
  if (b == 0) {
    throw a; /* give the dividend back */
  }
  return (a % b);
};

int gcd(int a, int b) {
  try {
    while (true) {
      int result = modulo(a, b);
      a = b;
      b = result;
    }
  } catch (int result) {
    return result;
  }
};
```

Exceptions interact with the C++ object model insofar as they are the only way for object construction to fail. For instance, back to the tutorial (Section 2.5.3 (p. 56)), the `OutputFile` class opens a file for data output, but such operation must fail on a read-only device. In such case, the file object must not be created. The programmer implements such case using an exception:

```
struct OutputFile {
  FILE* fileHandler;
  OutputFile(char* name) {
    if (! fileHandler = fopen(name, "w")) { /* perform system call
                                               to open file */
       /* fileHandler is NULL, open failed:
          abort object creation by throwing an exception */
       throw 42;
    }
  }
};

main () {
  try {
    OutputFile example("toto");
    example.write("18");
  } catch (int i) {
    printf("Open failed\n");
  }
}
```

Then, in the `main` function, if file opening fails, then the exception `42` is thrown, so that the constructor call fails in the caller, thus preventing file write.

Now recall the definition of the `LockedDeviceFile`, which models a file on a non-cooperative device requiring a lock at the level of the device:

```
struct LockedDeviceFile {
  /* WARNING: this order is important */
  DeviceLock deviceLock;
  File       file;

  LockedDeviceFile(char* device, char* fileName):
    /* order is irrelevant here */
    File(fileName),
    DeviceLock(device)
  {}

  ~LockedDeviceFile() {}
};
```

The constructor takes a lock on the device, then tries to open the file. What happens if file opening fails? The lock has to be released! Thus, C++ ensures that, if the construction of a field or base class subobject of an object fails, then the parts of the object that have been successfully constructed so far must be destructed, and the construction of the whole object fails.

In our formalism of construction states (Section 9.3.1 p. 176), this means that in the presence of exceptions, objects do not necessarily visit through all construction states: in particular, if the construction of a subobject of an object π fails, then π never goes through the Constructed construction state. As a consequence, the κ++ run-time invariant would have to be tailored to take this issue into account.

Now, what would happen if an uncaught exception broke a destructor, for instance if file closing failed? Actually, in this case, C++ does not define the semantics of such programs.

Moreover, exceptions interact with the semantics of C++ temporary objects (discussed in Section 12.4.1.2 p. 314): when an exception is caught, it might have to be copied before being used by the exception handler. Presumably for all those reasons, Lockheed Martin guidelines [55] prescribe their project engineers not to use C++ exceptions.

13.2.2 Templates

C++ offers a way to parameterize structure and function definitions with types or data at compile time, through *templates*. Siek and Taha [76] formalized templates in Isabelle-Isar. For instance, the following function template represents the identity function:

```
template <typename T> T id(T t) {
  return t;
};
```

This is actually not a function, but a function template. To be used, a template must be *instantiated* by giving a value to its template parameters:

```
cout << f<int>(18) << f<float>(4.2) << endl;
```

This asks the compiler to actually produce two functions, namely `f<int>` and `f<float>`, by replacing the template parameter `T` with `int` or `float`. However, the programmer is also allowed to omit the type name, such as `cout << f(18) << f(4.2) << endl`, thus asking the compiler to perform type inference to determine the type to use for `T`. This is called *implicit instantiation*.

Moreover, the programmer is also allowed to "override" particular instantiations of templates. This is called *specialization*. For instance, if the programmer specializes `f<int>` as follows:

```
int f(int i) {
  return -i;
};
```

Then, implicit template instantiations should select this specialized template. Template specialization interacts with the C++ object model in the presence of overloading and *partial specialization* (a template with several parameters, but not all of them receive a value).

13.2.3 Concurrency

Before C++11, concurrency was not a core feature of C++: it was only implemented through libraries not covered by the C++03 Standard [42]. Batty et al. [14] formalized in Isabelle the concurrency model candidate for the C++0x draft standard, which has since become official under C++11 [43]. This concurrency model is based on several kinds of *atomic* operations: *sequentially consistent atomics* for which sequential consistency is ensured but expensive, and *low-level atomics* for which the memory order may be fine-tuned by the programmer depending on the expected level of guarantee.

In practice, concurrency could interact with C++ object construction in compiler implementations: when the generalized dynamic type of an object changes, compilers update its dynamic type data as well as the data of all its subobjects, which often represent several memory accesses. Is this sequence of memory accesses free of data races?

13.3 Final thoughts

When I started my Ph.D. thesis on formal verification of compilers for object-oriented languages, I first focused on Java, fearing that the C++ object model would be too difficult and too technical to tackle, in the same way as Cargill [19] (in particular, delegate to sister class, or virtual functions during construction). When I decided to switch to C++, my colleagues warned me against the alleged complexity of what would later become this thesis: I would study the trickiest language, namely object-oriented C++ with multiple inheritance and object construction and destruction, using the trickiest formal methods, namely machine-checked formal verification with the Coq proof assistant.

Now that I have formalized the C++ object model on machine, on the contrary, my work has helped me understand important parts of the semantics of C++ multiple inheritance and object construction and destruction, and even more: I am convinced that they are relevant and useful in practice.

On the one hand, my Coq formalism shows that C++ object-oriented features are not obscure tools reserved for hackers. On the other hand, my verified compilation approach also shows that those features are not obscure high-level notions reserved for academic scholars. On the contrary, through my work, I am convinced that my work can serve as a **bridge of trust** between two distant worlds: the academic world and industrial engineering. Moreover, this bridge links the two worlds in both directions: scholars help engineers understand the meaning of their programs, while conversely engineers keep scholars aware of the real world.

From the general point of view of formalization, my work has strengthened more than ever my thoughts about the relevance of machine-checked verification. I am now convinced that any system, however complex it might be, deserves machine-checked formal verification. Complexity is not an obstacle (not even practical), but on the contrary a motivation for formal verification: formalizing a complex system in a proof assistant shows that the system is well understood. Conversely, a formal system description is more understandable than a prose textual specification, as formalization clarifies and eliminates most ambiguities of textual descriptions. Mechanized formalization in particular, and formal verification in general, prove to be the ultimate way of understanding and making understand complex systems to discuss of their correctness. I strongly hope that standardization organisms will one day call on formal verification to build a solid foundation to their standards and to reduce the risk of interpretative ambiguities or missing unspecified cases, following the final manifesto of my thesis:

Formal verification leads to understanding. Understanding leads to trust.

In Coq we trust.

Appendix A

Architecture of the Coq development

All proofs (except in the Discussion chapters) have been carried out with the Coq proof assistant [4]. They are available at [71]. The Coq theories do not include separate theories for s++ and Vcm [1]: the specification proofs presented in this thesis for those two languages are parts of the specifications and proofs for Ds++ and CVcm.

This chapter describes the architecture of the Coq development, by listing the theories in dependency order (so that no theory depends on any other theory listed below it). The dependencies between theories are depicted p. 334.

A.1 Small-step semantics

Events Section 3.2.1 (p. 66): Observational semantics of traces.

Smallstep Section 3.2 (p. 66): Small-step operational semantics.

A.2 Semantics of C++ Multiple Inheritance

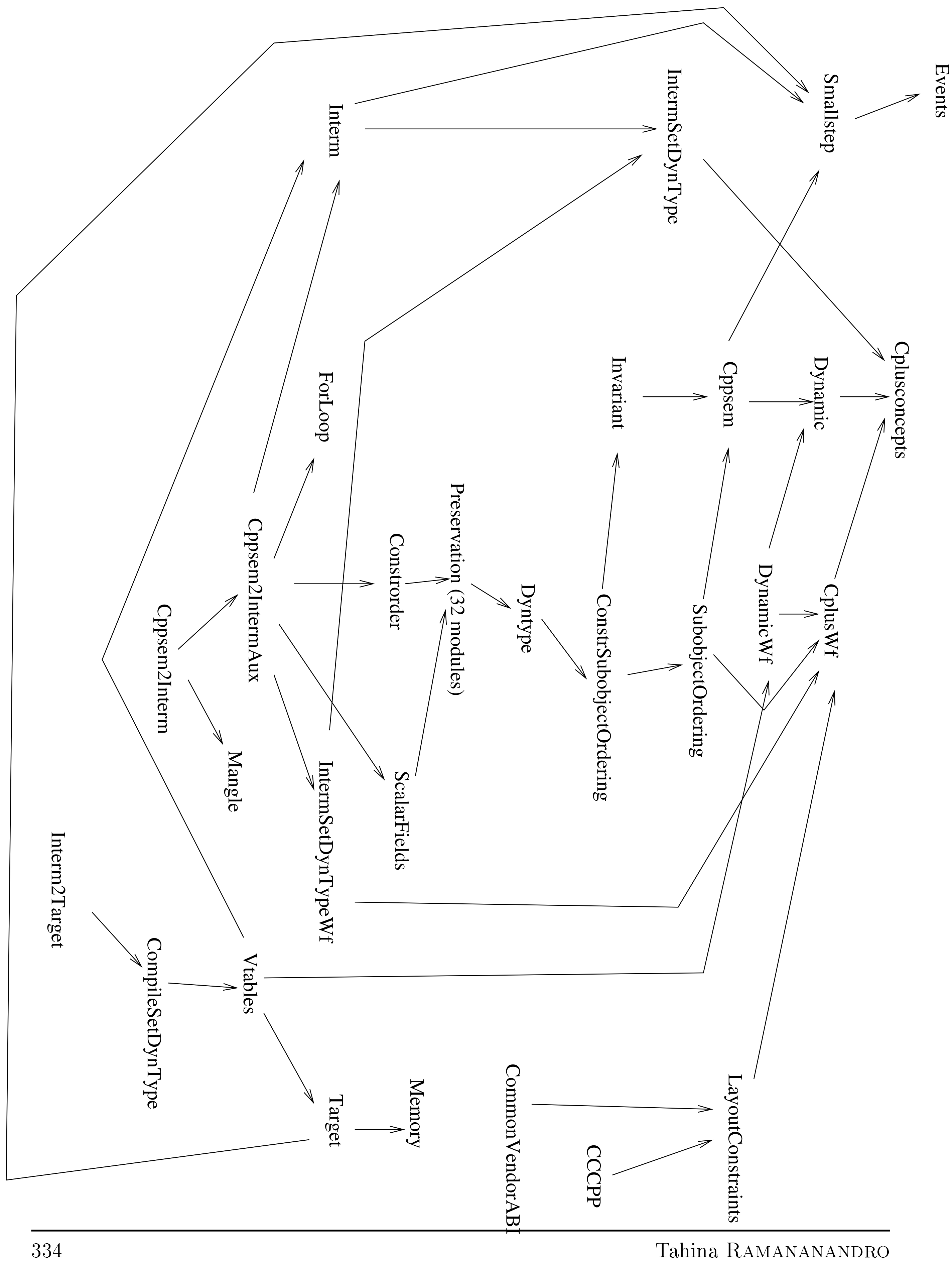
Cplusconcepts Section 4 (p. 71): Class hierarchies, subobjects; static and dynamic cast, virtual function dispatch.

Dynamic Section 4.4.5.2 (p. 89): Dynamic classes.

CplusWf Section 4.1.4 (p. 78): Well-formed hierarchies. Casts, function dispatch are decidable. The list of inheritance paths, the list of virtual bases, are computable in a finite amount of time.

DynamicWf Section 4.4.5.2 (p. 89): If the hierarchy is well-formed, the notion of "dynamic class" is decidable.

1. However, an older proof about object layout can be seen at [70] (at the time of our POPL 2011 [72] article). It models a simplified version of s++ and Vcm without functions.


Events
Smallstep
IntermSetDynType
Interm
Cplusconcepts
Dynamic
Cppsem
Invariant
ForLoop
Preservation (32 modules)
Constrorder
Cppsem2IntermAux
Dyntype
ConstrSubobjectOrdering
SubobjectOrdering
DynamicWf
CplusWf
Cppsem2Interm
Mangle
IntermSetDynTypeWf
ScalarFields
Interm2Target
CompileSetDynType
Vtables
Target
Memory
CommonVendorABI
CCCPP
LayoutConstraints

A.3 Object layout

LayoutConstraints Section 5.5 (p. 106): Sufficient conditions for object layout, and their proof of soundness.

CommonVendorABI Section 6.1 (p. 128): The Common Vendor ABI object layout algorithm, and its proof that it satisfies the soundness conditions.

CCCPP Section 6.2 (p. 132): Our CCCPP object layout algorithm, optimized for empty data member optimization, and its proof that it satisfies the soundness conditions.

A.4 Semantics of object construction and destruction

Cppsem Section 9.2 (p. 174): Syntax and semantics of κ++.

SubobjectOrdering Section 10.1.3 (p. 207): “Direct subobject” and “appears before” subobject orderings.

Progress Section 10.2 (p. 210): Construction and destruction rules progress for objects with nearly trivial constructors and trivial destructors.

Invariant Section 10.1 (p. 199): κ++ Run-time invariant.

ConstrSubobjectOrdering Section 10.3.2 (p. 212): A subobject goes through all construction states in order.

Dyntype Section 10.5.2 (p. 218): Unicity of the generalized dynamic type.

Preservation Section 10.1 (p. 199): (a total 32 theories) Proof of preservation of the κ++ run-time invariant.

ProgressInv Section 10.2 (p. 210): A more precise statement of progress theorems, including the construction states of constructed or destructed objects.

Constrorder Section 10.3.1 (p. 211): Resource Acquisition is Initialization, subobject construction order, evolution of the generalized dynamic type.

ConstrorderOther Section 10.3.4 (p. 216): Construction order of two subobjects of different complete objects.

ScalarFields Section 10.4 (p. 217): Safety of scalar field accesses.

A.5 Verified compilation

A.5.1 κ++ to Ds++

IntermSetDynType Section 11.4.10 (p. 252): Specification of the “set dynamic type” Ds++ operation.

Interm Section 11.2 (p. 245): Syntax and semantics of the Ds++ intermediate language.

IntermSetDynTypeWf Section 11.4.10 (p. 253): If the hierarchy is well-formed, then the “set dynamic type” operation is well-defined, decidable, and its result is unique.

ForLoop Section 11.5.3 (p. 259): Compile-time “for” loops, to compile construction and destruction of arrays.

Cppsem2IntermAux Section 11.5 (p. 254): A compiler for κ++ statements, constructors and destructors to Ds++, and its correctness proof.

Mangle Section 11.5.4 (p. 260): Name mangling: encoding the constructor argument types in a function name.

Cppsem2Interm Section 11.5 (p. 254): A compiler from κ++ to Ds++, and its proof of correctness: optimization of constructors for non-dynamic classes.

A.5.2 Ds++ to CVcm

Memory Section 11.7.3 (p. 289): The CVcm memory model, inspired from CompCert.

Target Section 11.7.2 (p. 286): Syntax and semantics of the CVcm target language.

Vtables Section 11.8.1 (p. 293): Construction of the contents and types of virtual tables and virtual table tables from a well-formed hierarchy.

CompileSetDynType Section 11.8.6 (p. 297): Compilation of the “set dynamic type” operation: updating the pointers to virtual tables.

Interm2Target Section 11.8 (p. 293): A compiler from Ds++ to CVcm, and its proof of correctness.

Appendix B

Formal verification of compilers

Verifying a compiler implies proving that the semantics of the source is preserved by the target. Thus, it requires to relate somehow all the semantics of source, intermediate and target languages. To this purpose, this chapter describes the formal foundations for proving compiler correctness. In particular, this chapter justifies our use of forward simulation for our verified compilers of Section 7.3 (p. 145) and Section 11.8 (p. 293). All facts presented in this chapter are taken from CompCert [2].

B.1 Program behaviours

***Definition* B.1.1.** *A* behaviour *of a transition system* $(\mathfrak{S}, (\rightarrow, \mathfrak{I}, \mathfrak{F}))$ *can be one of the following:*

- *A* terminating *behaviour with return code z is a finite trace* $\mathfrak{t} \in \mathfrak{E}^\star$ *produced by a finite number of transitions from an initial state s_0 to a final state s_f:*

$$\exists s_0 \in \mathfrak{I} : \exists s_f \in \mathfrak{F}_z : s_0 \overset{\star}{\underset{\mathfrak{t}}{\rightarrow}} s_f$$

- *A* diverging *behaviour is a finite trace* $\mathfrak{t} \in \mathfrak{E}^\star$ *produced by a finite number of transitions from an initial state s_0 to a state s followed by an infinite number of silent transitions:*

$$\exists s_0 \in \mathfrak{I} : s_0 \overset{+}{\underset{\mathfrak{t}}{\rightarrow}} s_1 \underset{\epsilon}{\rightarrow} s_2 \underset{\epsilon}{\rightarrow} \ldots \underset{\epsilon}{\rightarrow} s_n \underset{\epsilon}{\rightarrow} \ldots$$

- *A* reacting *behaviour is an infinite trace $\mathfrak{T}$ produced by an infinite number of transitions from an initial state s_0, but with no infinite sequence of silent transitions. In other words, it is an infinite sequence of finite sequences with non-empty finite traces:*

$$\exists s_0 \in \mathfrak{I} : \begin{cases} s_0 \overset{+}{\underset{\mathfrak{t}_0}{\rightarrow}} s_1 \overset{+}{\underset{\mathfrak{t}_1}{\rightarrow}} s_2 \overset{+}{\underset{\mathfrak{t}_2}{\rightarrow}} \ldots \overset{+}{\underset{\mathfrak{t}_{n-1}}{\rightarrow}} s_n \overset{+}{\underset{\mathfrak{t}_n}{\rightarrow}} \ldots \\ \forall i : \mathfrak{t}_i \neq \epsilon \\ \forall i : \mathfrak{T}_i = \mathfrak{t}_i +\!\!\!+ \mathfrak{T}_{i+1} \\ \mathfrak{T} = \mathfrak{T}_0 \end{cases}$$

- *A* wrong *behaviour is a finite number of transitions from an initial state s_0 to a non-final* stuck *state s_1 (from which there can be no step to any state s_2):*

$$\mathfrak{t} \in \mathfrak{E}^\star \ \wedge \ \exists s_0 \in \mathfrak{I} : \exists s_1 : s_0 \overset{\star}{\underset{\mathfrak{t}}{\rightarrow}} s_1 \ \wedge \ \forall \mathfrak{t}_2, s_2 : s_1 \underset{\mathfrak{t}_2}{\not\rightarrow} s_2 \ \wedge \ \forall z : (s_1, z) \notin \mathfrak{F}$$

Lemma B.1.1. *Any transition system has at least one behaviour*[1].

B.2 Semantics preservation

Program behaviors allow to formalize the notion of *semantics preservation* following [50]: if a transition system $\mathbb{S}$ is transformed into a transition system $\mathbb{S}'$, then every specification satisfied by a transition system $\mathbb{S}'$ has to be *preserved*, i.e. satisfied by the produced transition system $\mathbb{S}'$.

***Definition* B.2.1 (Specification).** *A* specification *is a logical predicate over non-wrong behaviours.*

A transition system $\mathbb{S}$ is said to satisfy *a specification S, denoted $\mathbb{S} \vDash S$, if and only if, $\mathbb{S}$ cannot go wrong and, for any behaviour b of $\mathbb{S}$, $S(b)$ holds.*

This formalism allows to specify *safety* properties (specifications over non-wrong behaviours only). However, CompCert [2] also allows reasoning about *liveness* properties (involving finite traces that are prefixes of possibly wrong behaviours). For the sake of brevity, we will not detail those proofs here.

***Definition* B.2.2 (Semantics preservation).** *Let $\mathbb{S}$ and $\mathbb{S}'$ be two transition systems. $\mathbb{S}'$ is said to* preserve the semantics of $\mathbb{S}$ *if, and only if, any specification satisfied by $\mathbb{S}$ is satisfied by $\mathbb{S}'$:*

$$\forall S :\ (\mathbb{S} \vDash S) \Rightarrow (\mathbb{S}' \vDash S)$$

***Definition* B.2.3 (Compiler correctness).** *Let $(\mathbb{P}, \mathbb{S})$ and $(\mathbb{P}', \mathbb{S}')$ be two programming languages. A compiler C, partial function from $\mathbb{P}$ to $\mathbb{P}'$, is said to be* correct *if, and only if, for each source program $P \in \mathbb{P}$ such that its compiled program $C(P)$ exists, the semantics of the transition system $\mathbb{S}'(C(P))$ preserves the semantics of $\mathbb{S}(P)$.*

Proving semantics preservation directly using this definition can be tedious. Fortunately, CompCert provides several proof techniques to this purpose, which are described in the following sections.

B.2.1 Backward simulation

The following lemma provides a way of showing semantics preservation at the level of program behaviours:

Lemma B.2.1. *Let $\mathbb{S}, \mathbb{S}'$ be two transition systems such that the following two conditions hold:*

(a) if $\mathbb{S}$ cannot go wrong, then neither can $\mathbb{S}'$

(b) any non-wrong behaviour of $\mathbb{S}'$ is a behaviour of $\mathbb{S}$

Then, $\mathbb{S}'$ preserves the semantics of $\mathbb{S}$.

Proof. Let S be a specification satisfied by $\mathbb{S}$. Then, by hypothesis, $\mathbb{S}$ cannot go wrong, so neither can $\mathbb{S}'$.

Let B be a non-wrong behaviour of $\mathbb{S}'$. Then, as $\mathbb{S}$ cannot go wrong, B is also a non-wrong behaviour of S, thus $S(B)$ holds, which concludes. □

1. This result requires the excluded middle.

The most accurate way to show semantics preservation between two transition systems $\mathbb{S}$ and $\mathbb{S}'$ is to show a *backward simulation.* Any transition step of $\mathbb{S}'$ match a finite sequence of $\mathbb{S}$ transition steps from *safe* $\mathbb{S}$ states:

Definition **B.2.4.** *Let $\mathbb{S}$ be a transition system. A state s is said to be* safe *if, and only if, it is final or not stuck.*

Definition **B.2.5 (Backward simulation).** *Let $\mathbb{S} = (\mathfrak{S}, (\rightarrow, \mathfrak{I}, \mathfrak{F}))$ and $\mathbb{S}' = (\mathfrak{S}', (\rightarrow', \mathfrak{I}', \mathfrak{F}'))$ be two transition systems, and $\triangleright \subseteq \mathfrak{S} \times \mathfrak{S}'$ a relation between $\mathbb{S}$ states and $\mathbb{S}'$ states. $\triangleright$ is said to be a* backward simulation *between $\mathbb{S}$ and $\mathbb{S}'$ if, and only if, there is a well-founded order $<$ on $\mathfrak{S}$ such that the following conditions hold:*

(i) Final states match:

$$\begin{aligned} \forall (s'_f, z) \in \mathfrak{F}', \forall s_9 \in \mathfrak{S} : &\quad s_9 \triangleright s'_f \\ \Rightarrow \exists s_f \in \mathfrak{F}_z : &\quad s_9 \overset{\star}{\rightarrow} s_f \end{aligned}$$

(ii) Backward invariant preservation:

$$\begin{aligned} \forall s'_1, s'_2, s_1 : &\quad s'_1 \rightarrow s'_2 \land s_1 \triangleright {s_1}' \land s_1 \textit{ safe} \\ \Rightarrow \exists s_2 : &\quad s_1 \overset{\star}{\rightarrow} s_2 \land s_2 \triangleright {s_2}' \\ &\quad \land (s_1 = s_2 \Rightarrow s'_2 < s'_1) \end{aligned}$$

(iii) Initial states match:

$$\mathfrak{I} \neq \varnothing \Rightarrow \forall s'_\circ \in \mathfrak{I}' : \exists s_\circ \in \mathfrak{I}, \exists s_1 \in \mathfrak{S} : s_\circ \overset{\star}{\rightarrow} s_1 \land s_1 \triangleright s'_\circ$$

(iv) For any states $s \triangleright s'$ such that s is safe, then s' is safe.

(v) If there is a $\mathbb{S}$ initial state, then there is a $\mathbb{S}'$ initial state.

The correctness of backward simulation is ensured by the following lemma:

Lemma B.2.2 (Semantics preservation by backward simulation). *If there is a backward simulation between $\mathbb{S}$ and $\mathbb{S}'$, then $\mathbb{S}'$ preserves the semantics of $\mathbb{S}$.*

Proof. Using Lemma B.2.1 (p. 338):

(a) is proved thanks to $(iv), (v)$

(b) is proved thanks to $(i), (ii), (iii)$ following CompCert [2] by case analysis on the behaviours, then by induction on terminating behaviours, or coinduction on diverging or reacting behaviours. □

B.2.2 Forward simulation

Backward simulation is often difficult to prove, even not mechanically. Indeed, usually, semantics preservation is proved in a more natural way following the execution of the source program rather than the target program, using *forward simulation*:

Definition **B.2.6 (Forward simulation).** *Let $\mathbb{S} = (\mathfrak{S}, (\rightarrow, \mathfrak{I}, \mathfrak{F}))$ and $\mathbb{S}' = (\mathfrak{S}', (\rightarrow', \mathfrak{I}', \mathfrak{F}'))$ be two transition systems, and $\triangleright \subseteq \mathfrak{S} \times \mathfrak{S}'$ a relation between $\mathbb{S}$ states and $\mathbb{S}'$ states. $\triangleright$ is said to be a* forward simulation *from $\mathbb{S}$ to $\mathbb{S}'$ if, and only if, the three following conditions hold:*

– *Initial states match:*

$$\forall s_\circ \in \mathfrak{I} : \exists s'_\circ \in \mathfrak{I}', \exists s'_1 \in \mathfrak{S}' : s'_\circ \overset{\star}{\to}' s'_1 \land s_\circ \rhd s'_1$$

– *Forward invariant preservation:*

$$\begin{aligned} \forall s_1, s_2, {s_1}' : &\quad s_1 \to s_2 \land s_1 \rhd {s_1}' \\ \Rightarrow \exists {s_2}' : &\quad {s_1}' \overset{+}{\to}' {s_2}' \land s_2 \rhd {s_2}' \end{aligned}$$

– *Final states match:*

$$\begin{aligned} \forall (s_f, z) \in \mathfrak{F}, \forall s'_9 \in \mathfrak{S}' : &\quad s_f \rhd s'_9 \\ \Rightarrow \exists s'_f \in \mathfrak{F}'_z : &\quad s'_9 \overset{\star}{\to}' s'_f \end{aligned}$$

Graphically:

$$\begin{tikzcd}
\mathbb{S} & \mathfrak{I} \arrow[rr, dash, "\ni"] & & s_\circ \arrow[d, dotted, dash, "\rhd"] & s_1 \arrow[r, "\mathfrak{e}^?"'] \arrow[d, dash, "\rhd"] & s_2 \arrow[d, dotted, dash, "\rhd"] & s_f \arrow[rr, dash, "\in"] \arrow[d, dash, "\rhd"] & & \mathfrak{F}_z \\
\mathbb{S}' & \mathfrak{I}' \arrow[r, dotted, dash, "\ni"] & s'_\circ \arrow[r, dotted, "'\star"] & s'_1 & s'_1 \arrow[r, dotted, "'+", "\mathfrak{e}^?"'] & s'_2 & s'_{99} \arrow[r, dotted, "'\star"] & s'_f \arrow[r, dotted, dash, "\in"] & \mathfrak{F}'_z
\end{tikzcd}$$

Fortunately, if the target language is deterministic enough, then the following theorem allows to prove semantics preservation through forwards simulation:

Theorem B.1 (Semantics preservation by forward simulation). *Let* $\mathbb{S} = (\mathfrak{S}, (\to, \mathfrak{I}, \mathfrak{F}))$ *and* $\mathbb{S}' = (\mathfrak{S}', (\to', \mathfrak{I}', \mathfrak{F}'))$ *be two transition systems such that there is a forward simulation between* $\mathbb{S}$ *and* $\mathbb{S}'$. *Then:*

(i) any non-wrong behaviour of $\mathbb{S}_1$ *is a behaviour of* $\mathbb{S}_2$*;*

(ii) moreover, if $(\mathbb{S}, \mathbb{S}')$ *form a* receptive-determinate *system, then,* $\mathbb{S}'$ *preserves the semantics of* $\mathbb{S}$.

Proof. (i) Following CompCert [2], by case analysis on the behaviours, then by induction on terminating behaviours, or coinduction on diverging or reacting behaviours.

(ii) Following Sevcik et al. [3, 75] introducing the:

***Definition* B.2.7.** $(\mathbb{S}, \mathbb{S}')$ *form a* receptive-determinate *system if, and only if, there exists some equivalence relation* $\sim$ *between events, such that:*

– $\mathbb{S}$ *is* receptive*, i.e. for any step from a state s producing an event* $\mathfrak{e}_1$*, there is a step from s producing* $\mathfrak{e}_2$ *for each* $\mathfrak{e}_2 \sim \mathfrak{e}_1$

– $\mathbb{S}'$ *is* determinate*, i.e. for any two steps from a common state, either none produces any event, or both produce events matching through* $\sim$

– $\mathbb{S}'$ *is* internally deterministic*, i.e. for any state* s'_1 *and any* $\mathfrak{e}^? \in \mathfrak{E}^?$*, there is at most one state* s'_2 *such that* $s'_1 \underset{\mathfrak{e}^?}{\to}' s'_2$

Then, roughly speaking, the proof constructs a backward simulation by making the $\mathbb{S}$ transition system *stutter*: it is already known that a $\mathbb{S}'$ transition is part of a sequence of $\mathbb{S}'$ transitions matching one $\mathbb{S}$ transition through the forward simulation diagram. Then, determinacy hypotheses allow defining the backward simulation. □

Internal determinism says that the only possible source of non-determinism is the trace semantics, e.g. the sequence of user (or random) inputs.

In practice, the receptive-determinate hypothesis shall be verified by all the languages considered in our work, as regards transition steps other than built-in operations: any step related to C++ multiple inheritance shall be deterministic. Then, it suffices to assume them on built-in operations, as other operations are assumed to produce no event.

Obviously, if there are a forward simulation commutative diagram from $\mathbb{S}$ to $\mathbb{S}'$ on the one hand, and from $\mathbb{S}'$ to $\mathbb{S}''$ on the other hand, then it is possible to construct a commutative forward simulation diagram from $\mathbb{S}$ to $\mathbb{S}''$.

So, finally, to prove the correctness of a compiler, it suffices to:

1. show each compilation pass [2] by a commutative forward simulation diagram
2. then show that the last (lowest-level) target language has no internal non-determinism [3] and forms a receptive-determinate system with the first (highest-level) source language [4]

This justifies why, throughout this thesis, all results about verified compilation shall be proved through commutative forward simulation diagram: our target language is a superset of Cminor, which is then intended (in the future) to be compiled to Cminor; then, the CompCert backend (which is proved only using such forward simulation patterns) shall be reused, so that the semantics shall be preserved by transitivity of forward simulation.

2. Unless there are compilation passes that preserve only some selected behaviours from a non-determinist transition system. However, this is not the case with the languages that we shall present throughout this thesis.

3. which is the case for the assembly language as formalized by CompCert

4. which can be assumed at the level of built-in operations (Section 3.2.4 p. 67)

Bibliography

[1] The Astrée static analyzer. `http://astree.ens.fr`.

[2] The CompCert verified compiler. `http://compcert.inria.fr`.

[3] The CompCertTSO compiler – commented Coq development. `http://www.cl.cam.ac.uk/~pes20/CompCertTSO/doc/`.

[4] The Coq proof assistant. `http://coq.inria.fr`.

[5] Frama-C Software Analyzers. `http://frama-c.com`.

[6] The Alloy Analyzer. `http://alloy.mit.edu`.

[7] The Isabelle proof assistant. `http://www.cl.cam.ac.uk/research/hvg/isabelle/`.

[8] Why. `http://why.lri.fr`.

[9] Jean-Raymond Abrial. A formal approach to large software construction. In *Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University*, pages 1–20, London, UK, 1989. Springer-Verlag.

[10] Jean-Raymond Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, 1996.

[11] Jean-Raymond Abrial, Stephen A. Schuman, and Bertrand Meyer. A specification language. In A. M. Macnaghten and R. M. McKeag, editors, *On the Construction of Programs*. Cambridge University Press, 1980.

[12] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David Grove. Efficient implementation of Java interfaces: `invokeinterface` considered harmless. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 108–124, New York, NY, USA, 2001. ACM.

[13] Brian Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Foster, Benjamin Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized Metatheory for the Masses: The PoplMark Challenge. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer Berlin / Heidelberg, 2005.

[14] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL'11*, pages 55–66, 2011.

[15] Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Towards Formally Verified Optimizing Compilation in Flight Control Software. In Philipp Lucas, Lothar Thiele, Benoit Triquet, Theo Ungerer, and Reinhard Wilhelm, editors, *Predictability and Performance in Embedded Systems : PPES 2011*, volume 18 of *OpenAccess Series in Informatics (OASIcs)*, pages 59–68, Grenoble, France, March 2011. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[16] Sandrine Blazy and Xavier Leroy. Formal verification of a memory model for C-like imperative languages. In *International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2005.

[17] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[18] Cari Borrás. Overexposure of radiation therapy patients in Panama: problem recognition and follow-up measures. *Pan-American J. of public health*, 20(2/3):173–187, 2006.

[19] Thomas A. Cargill. Controversy: The Case Against Multiple Inheritance in C++. *Computing Systems*, pages 69–82, 1991.

[20] Juan Chen. A typed intermediate language for compiling multiple inheritance. In *34th symp. Principles of Programming Languages*, pages 25–30. ACM, 2007.

[21] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8:244–263, April 1986.

[22] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. Itanium C++ ABI. `http://www.codesourcery.com/public/cxx-abi`, 2001.

[23] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[24] Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28:2–2, November 2003.

[25] Beman Dawes. POD's Revisited; Resolving Core Issue 568 (Revision 2). Technical report, ISO/IEC SC22/JTC1/WG21, March 2007.

[26] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.

[27] Nachum Dershowitz. Software Horror Stories. `http://www.cs.tau.ac.il/~nachumd/horror.html`.

[28] Mark Dowson. The Ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22:84–, March 1997.

[29] Karel Driesen, Urs Hölzle, and Jan Vitek. Message dispatch on pipelined processors. In Mario Tokoro and Remo Pareschi, editors, *ECOOP'95 – Object-Oriented Programming, 9th European Conference, Aarhus, Denmark, August 7-11, 1995*, volume 952 of *Lecture Notes in Computer Science*, pages 253–282. Springer Berlin / Heidelberg, 1995.

[30] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[31] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182 –211, 1976.

[32] Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 337–350. ACM, 2007.

[33] The Apache Foundation. Apache Lucene. `http://lucene.apache.org`.

[34] Pierre Froment. L'architecture avionique de l'A380. In *Les Annales des Mines*, Réalités Industrielles. November 2005.

[35] Joseph Gil and Peter F. Sweeney. Space and Time-Efficient Memory Layout for Multiple Inheritance. In *14th conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1999)*, pages 256–275. ACM, 1999.

[36] Joseph (Yossi) Gil, William Pugh, Grant E. Weddell, and Yoav Zibin. Two-dimensional bidirectional object layout. *ACM Trans. Program. Lang. Syst.*, 30(5):1–38, 2008.

[37] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison Wesley, 3rd edition edition, 2005.

[38] Nicolas Hognon, Jeff Schwab, and Rob Williscroft. Multiple Inheritance size problem. `http://bytes.com/topic/c/answers/129536-multiple-inheritance-size-problem`, 2005.

[39] Laurent Hubert, Thomas Jensen, Vincent Monfort, and David Pichardie. Enforcing secure object initialization in Java. In *Computer Security – ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 101–115. Springer, 2010.

[40] International Organization for Standards. *International Standard ISO/IEC 14882:1998. Programming Languages — C++*, 1998.

[41] International Organization for Standards. *International Standard ISO/IEC 13568:2002. Information Technology — Z Formal Specification Notation — Syntax, Type System and Semantics*, 2002.

[42] International Organization for Standards. *International Standard ISO/IEC 14882:2003. Programming Languages — C++*, 2003.

[43] International Organization for Standards. *International Standard ISO/IEC 14882:2011. Programming Languages — C++*, 2011.

[44] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11:256–290, April 2002.

[45] C++ Standards Committee (ISO/IEC JTC1/SC22/WG21). CWG 1202: Calling virtual functions during destruction. In *C++ Standard Core Language Defect Reports*. International Organization for Standards, `http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#1202`, March 2011.

[46] Cem Kaner, Jack L. Falk, and Hung Quoc Nguyen. *Testing computer software.* Wiley, 1993.

[47] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[48] Stein Krogdahl. Multiple inheritance in Simula-like languages. *BIT Numerical Mathematics*, 25:318–326, 1985.

[49] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[50] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[51] Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *Lecture Notes in Computer Science*, pages 617–617. Springer Berlin / Heidelberg, 2003.

[52] Pierre Letouzey. Extraction in Coq, an Overview. In *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, volume 5028 of *Lecture Notes in Computer Science*, Athens, Grèce, 2008. Springer-Verlag. ANR CompCert.

[53] N.G. Leveson and C.S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26:18–41, 1993.

[54] Stanley B. Lippman. *Inside the C++ object model.* Addison-Wesley, 1996.

[55] Lockheed Martin. Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program. `http://www.research.att.com/~bs/JSF-AV-rules.pdf`, 2005.

[56] Chenguang Luo and Shengchao Qin. Separation Logic for Multiple Inheritance. *Electron. Notes Theor. Comput. Sci.*, 212:27–40, 2008.

[57] John McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

[58] John Mccarthy and James Painter. Correctness of a compiler for arithmetic expressions. pages 33–41. American Mathematical Society, 1967.

[59] Bertrand Meyer. *Eiffel, le langage.* InterEditions, 1992.

[60] R Milner and R Weyhrauch. Proving compiler correctness in a mechanized logic. In *In Proc. 7th Annual Machine Intelligence Workshop, volume 7 of Machine Intelligence*, pages 51–72. Edinburgh University Press, 1972.

[61] Robin Milner, Mads Tofte, Robert Harper, and David McQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[62] J Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5:461–492, 1989.

[63] Nathan Myers. The Empty Member C++ Optimization. *Dr Dobb's Journal*, August 1997.

[64] Michael Norrish. A Formal Semantics for C++. Technical report, NICTA, 2008.

[65] Radio Technical Commission on Aviation. DO-178B – Software Considerations in Airborne Systems and Equipment Certification, 1992.

[66] Oracle Sun Developer Network. Hotspot creashed with sigsegv from PorterStemmer (Bug ID 7070134). `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=7070134`, July 2011.

[67] Marie-Laure Potet and Didier Bert. La méthode B. `http://www-verimag.imag.fr/~potet/ejcp-expose.pdf`, May 2010. Course given at École des Jeunes Chercheurs en Programmation, Dinard.

[68] Xin Qi and Andrew C. Myers. Masked Types for Sound Object Initialization. In *POPL '09*, pages 53–65, New York, NY, USA, 2009. ACM.

[69] G. Ramalingam and Harini Srinivasan. A member lookup algorithm for C++. In *Programming Language Design and Implementation (PLDI'97)*, pages 18–30. ACM, 1997.

[70] Tahina Ramananandro. Machine-checked object layout for C++ multiple inheritance with empty-base optimization – Technical report, Coq development and supplementary material. `http://gallium.inria.fr/~tramanan/cxx/object-layout`, 2010.

[71] Tahina Ramananandro. Verified compilation of C++ Multiple Inheritance – Coq proofs. `http://gallium.inria.fr/~tramanan/cxx/compiler`, 2011.

[72] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. Formal verification of object layout for C++ multiple inheritance. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 67–80. ACM, 2011.

[73] Tahina Ramananandro, Gabriel Dos Reis, and Xavier Leroy. A Mechanized Semantics for C++ Object Construction and Destruction, with Applications to Resource Management. In *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL)*, 2012.

[74] Jonathan G. Rossie and Daniel P. Friedman. An Algebraic Semantics of Subobjects. In *10th conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1995)*, pages 187–199. ACM, 1995.

[75] Jaroslav Sevcik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11. ACM, 2011.

[76] Jeremy Siek and Walid Taha. A Semantic Analysis of C++ Templates. In Dave Thomas, editor, *ECOOP 2006 Object-Oriented Programming*, volume 4067 of *Lecture Notes in Computer Science*, pages 304–327. Springer Berlin / Heidelberg, 2006.

[77] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995.

[78] J. Michael Spivey. *The Z Notation: A reference manual*. International Series in Computer Science. Prentice Hall, 2nd edition, 1992.

[79] Bjarne Stroustrup. *The design and evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[80] Bjarne Stroustrup. *A history of C++: 1979–1991*, pages 699–769. ACM, New York, NY, USA, 1996.

[81] Peter F. Sweeney and Michael G. Burke. Quantifying and evaluating the space overhead for alternative C++ memory layouts. *Software: Practice and Experience*, 33(7):595–636, 2003.

[82] Jean-Baptiste Tristan. *Vérification formelle de validateurs de traduction*. PhD thesis, Université Paris. Diderot (Paris 7), 2009.

[83] Harvey Tuch. Formal verification of C systems code: Structured types, separation logic and theorem proving. *Journal of Automated Reasoning*, 42(2):125–187, 2009.

[84] Daniel Wasserrab. *From Formal Semantics to Verified Slicing – A Modular Framework with Applications in Language Based Security*. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, October 2010.

[85] Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *21st conf. on Object-Oriented*

Programming, Systems, Languages, and Applications (OOPSLA 2006), pages 345–362. ACM, 2006.

[86] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *2011*, 2011.

[87] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient dynamic dispatch without virtual function tables: the SmallEiffel compiler. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '97, pages 125–141, New York, NY, USA, 1997. ACM.

Index of theorems

Index of concepts

Index of equations

Index of notations