HAL open science

Semantic foundations of intermediate program representations

Delphine Demange

▶ To cite this version:

Delphine Demange. Semantic foundations of intermediate program representations. Other [cs.OH]. École normale supérieure de Cachan - ENS Cachan, 2012. English. NNT : 2012DENS0053 . tel-00905442

HAL Id: tel-00905442

https://theses.hal.science/tel-00905442v1

Submitted on 18 Nov 2013

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

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

THÈSE / ENS CACHAN - BRETAGNE

sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de

DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

Mention : Informatique

École doctorale MATISSE

présentée par

Delphine Demange

Préparée à l'Unité Mixte de Recherche 6074
Institut de recherche en informatique
et systèmes aléatoires

Semantic Foundations of Intermediate Program Representations

Thèse soutenue le 19 octobre 2012
devant le jury composé de :

Xavier Leroy
Directeur de recherche - INRIA / *rapporteur*

John Gregory Morrisett
Professeur des universités - Harvard University / *rapporteur*

Luc BOUGÉ
Professeur des universités - ENS Cachan - Bretagne / *examinateur*

Albert COHEN
Directeur de recherche - INRIA / *examinateur*

Marie-Laure POTET
Professeur des universités - ENSIMAG / *examinatrice*

Thomas JENSEN
Directeur de recherche - INRIA / *directeur de thèse*

David PICHARDIE
Chargé de recherche - INRIA / *co-directeur de thèse*

Remerciements

Je tiens en premier lieu à remercier Xavier Leroy et Greg Morrisett d'avoir accepté de rapporter ma thèse, et de l'intérêt qu'ils ont porté à mon travail. C'est un grand honneur, et je les remercie très sincèrement pour les retours de grande qualité qu'ils m'ont donnés à propos du manuscrit. Je remercie également Albert Cohen et Marie-Laure Potet d'avoir accepté d'être examinateurs à ma soutenance et de l'intérêt qu'ils ont porté à ma présentation. Je remercie aussi Luc Bougé d'avoir bien voulu présider ce jury, 6 années après m'avoir ouvert les portes de l'ENS Cachan Antenne Bretagne et du Magistère Informatique et Télécommunications.

Cette thèse ne serait pas sans mes deux encadrants, Thomas Jensen et David Pichardie. Ces trois ans de travail à leur côté ont été très intenses, car passionnants. Je les remercie de m'avoir montré les directions à suivre, mais aussi pour leur disponibilité, leur enthousiasme, les conseils précieux qu'ils m'ont donnés, leurs encouragements, et toutes les opportunités de de collaborations, en France ou à l'étranger, qu'ils m'ont offertes durant ces trois ans. Leurs retours sur les versions préliminaires du manuscrit en ont considérablement amélioré la qualité, et je les en remercie.

Je remercie également Gilles Barthe, avec qui collaborer a été très enrichissant et un réel plaisir. Durant presque une année de collaboration, ses visites à Rennes et les nôtres à Madrid, Gilles m'a, par ailleurs, beaucoup appris par son expérience et ses conseils d'ordre extra scientifiques, et je lui en suis reconnaissante. Un grand merci aussi à Jan Vitek et Suresh Jagannathan pour cette aventure très riche passée à Purdue avec Vincent Laporte, David Pichardie et Lei Zhao. Je les remercie pour avoir réussi à me transmettre, grâce à leur expertise et nos nombreux échanges, des intuitions sur un sujet déroutant de prime abord. Je les remercie aussi pour leur accueil très chaleureux, leurs conseils, et le soutien académique qu'ils m'ont apporté.

Merci aussi à toute l'équipe Celtique, au sein de laquelle j'ai fait ma thèse. Merci à Frédéric Besson, Laurent Hubert, et aux ingénieurs de l'équipe avec qui j'ai pu travailler sur Sawja: Nicolas Barré, Tiphaine Turpin, Vincent Monfort et Pierre Vittet. Une mention spéciale à Arnaud Jobin, mon co-bureau hors-pair de presque 4 ans. Je remercie aussi Lydie Mabil pour sa patience, sa disponibilité et son aide efficace pour mes démarches administratives.

Un énorme merci à Philippe Aoustin, de l'atelier, pour avoir réussi à réssuciter ma machine de travail à 15 jours de la soutenance.

Finalement, je remercie mes parents et ma soeur d'avoir fait le déplacement jusqu'à Rennes pour assister à ma soutenance, mais aussi toute ma proche famille, qui aurait tant voulu y assister. Je les remercie de m'avoir soutenue depuis le début, et ce quelles que soient les circonstances. Je leur en suis infiniment reconnaissante, et leur dédie ce manuscrit.

Contents

Remerciements **3**

Résumé étendu en français **9**

1 Introduction **15**
1.1 Formal verification of software . . . 15
1.1.1 Compilers and program verifiers . . . 15
1.1.2 Verified compilers and verifiers . . . 18
1.2 Intermediate representations to the rescue . . . 19
1.3 Contributions and structure of the document . . . 20

2 Intermediate representations **23**
2.1 A first informal definition . . . 23
2.2 Some leading IRs . . . 24
2.2.1 Three-address code: TAC . . . 25
2.2.2 Stack-based code: STACK . . . 26
2.2.3 Static Single Assignment: SSA . . . 28
2.2.4 Continuation-passing style: CPS . . . 29
2.2.5 Cost analysis: Costa . . . 32
2.2.6 Program verification: Boogie . . . 33
2.3 Discussion . . . 35
2.3.1 The semantic impact of syntax and structure . . . 35
2.3.2 A perfect IR? . . . 36
2.4 Conclusions . . . 37

3 Proving transformations correct **39**
3.1 Semantics preservation . . . 39
3.1.1 Formal semantics . . . 40
3.1.2 Observational semantics . . . 41
3.1.3 Choosing the right preservation criteria . . . 43
3.2 Simulation relations . . . 45
3.2.1 Simulations for semantics preservation . . . 45
3.2.2 Simulations as semantic transformations . . . 48
3.3 Proof techniques for transformation . . . 49
3.3.1 Provably correct transformations . . . 49
3.3.2 Translation validation . . . 49
3.4 Related work and conclusion . . . 51
3.4.1 Relational approaches to transformation correctness . . . 51
3.4.2 Summary . . . 51

4 A stackless IR for Java bytecode **53**
4.1 Introduction . . . 53
4.1.1 Key problems to address . . . 54
4.1.2 Contribution and content . . . 56
4.2 The source and target languages . . . 57
4.2.1 Languages syntax . . . 57
4.2.2 Semantics . . . 59
4.2.2.1 Semantic domains . . . 59
4.2.2.2 Transition relations . . . 60
4.2.2.3 Semantics of BC . . . 61
4.2.2.4 Semantics of BIR . . . 63
4.3 The BC2BIR algorithm . . . 66
4.3.1 Transforming instructions . . . 66
4.3.2 Transforming method code . . . 68
4.4 Semantic correctness of BC2BIR . . . 70
4.4.1 Semantic relations . . . 71
4.4.2 Soundness result . . . 71
4.4.3 Application examples . . . 75
4.5 The Sawja tool bench . . . 76
4.5.1 Overview of the library . . . 76
4.5.2 From BC to JBC . . . 77
4.5.3 Experiments . . . 77
4.6 Related work . . . 80
4.6.1 Transformation and analysis frameworks . . . 80
4.6.2 Transformation techniques and proofs . . . 81
4.7 Conclusions . . . 82

5 Static Single Assignment form **83**
5.1 Introduction . . . 83
5.1.1 Powerful properties require care . . . 83
5.1.2 Verified compilers need semantic properties . . . 84
5.1.3 Contributions . . . 84
5.1.4 Contents . . . 85
5.2 Background on SSA . . . 86
5.3 The CompCert C compiler . . . 88
5.3.1 Observational semantics . . . 88
5.3.2 Behavior preservation . . . 89
5.4 The RTL language . . . 90
5.4.1 Syntax and semantics . . . 90
5.4.2 Normalizing RTL syntax . . . 91
5.5 The SSA language . . . 92
5.5.1 SSA programs . . . 92
5.5.1.1 Syntax . . . 92
5.5.1.2 Strict SSA . . . 93
5.5.1.3 Well-formed SSA programs . . . 93
5.5.2 Semantics . . . 94
5.5.2.1 Exploiting normalization for an intuitive semantics . . . 94

5.5.2.2 Parallel execution of ϕ-blocks . . . 95
5.6 Translation validation of the SSA generation . . . 95
5.6.1 Type system . . . 96
5.6.1.1 Liveness . . . 96
5.6.1.2 Typing rules for instructions . . . 97
5.6.1.3 Typing rules for edges and functions . . . 99
5.6.2 The type system ensures strict SSA form . . . 99
5.6.3 Soundness of the type system . . . 100
5.6.3.1 Simulation relation . . . 100
5.6.3.2 Proof sketch . . . 101
5.6.4 Completeness of the type system . . . 102
5.6.5 Implementation . . . 103
5.7 SSA-based optimizations and the equational lemma . . . 104
5.7.1 Equational lemma . . . 104
5.7.2 Application to Copy Propagation . . . 104
5.7.3 Validation of Global Value Numbering . . . 105
5.7.4 Discussion . . . 107
5.8 Conversion out of SSA . . . 108
5.8.1 Critical edges . . . 108
5.8.2 The swap problem . . . 108
5.8.3 Correctness proof . . . 109
5.9 Implementation and experimental results . . . 109
5.9.1 Efficiency of the SSA validator . . . 109
5.9.2 Effectiveness of the GVN optimizer . . . 110
5.9.3 Efficiency of the generated code . . . 110
5.10 Related work . . . 111
5.11 Conclusions and future work . . . 112

6 Memory model for concurrent Java IRs 113
6.1 Introduction to weak memory models . . . 115
6.1.1 Hardware memory models . . . 115
6.1.1.1 Relaxing SC . . . 115
6.1.1.2 Total Store Order . . . 116
6.1.1.3 DRF guarantee . . . 116
6.1.2 Software memory models . . . 117
6.1.2.1 SC-enforcing compilers . . . 118
6.1.2.2 SC for correctly synchronised programs . . . 118
6.1.2.3 Weak DRF memory models . . . 121
6.1.2.4 The limits of the Java Memory Model . . . 121
6.2 An alternative contract: BMM . . . 123
6.2.1 BMM from a larger perspective . . . 124
6.2.2 Summary . . . 124
6.2.3 Contributions and content . . . 125
6.3 Background on Java Memory Model . . . 126
6.3.1 Inter-thread actions . . . 126
6.3.2 Intra-thread semantics . . . 129
6.4 Axiomatic memory model: BMM . . . 131

6.4.1 BMM is a least post-fixpoint . . . 133
6.4.2 BMM is a subset of JMM . . . 133
6.4.3 DRF guarantee . . . 133
6.5 Operational memory model: $\mathrm{BMM}_{\mathsf{o}}$ . . . 134
6.6 BMM and $\mathrm{BMM}_{\mathsf{o}}$ are equivalent . . . 136
6.6.1 $\rho(\mathrm{BMM}_{\mathsf{o}}) \subseteq \mathrm{BMM}$ . . . 137
6.6.2 $\mathrm{BMM} \subseteq \rho(\mathrm{BMM}_{\mathsf{o}})$ . . . 139
6.7 Validity of transformations . . . 139
6.7.1 Validity of WR and WR⋆R . . . 140
6.7.2 Proving transformations invalid . . . 140
6.8 Empirical evaluation of BMM . . . 141
6.9 Related work . . . 142
6.10 Conclusion . . . 143

7 Conclusions and perspectives **145**
7.1 Summary . . . 145
7.2 Interactions between IRs and analyses . . . 146
7.2.1 Semantics preservation and program proof . . . 146
7.2.2 IR as an analysis . . . 148
7.3 Extensions . . . 149
7.3.1 A verified front-end for Sawja . . . 149
7.3.2 Concurrent BIR . . . 149
7.3.3 SSA deconstruction . . . 150
7.3.4 SSA-based optimizations . . . 150
7.4 Perspective: towards more abstract IRs . . . 151

8 Appendix **153**
8.1 Correctness of BC2BIR . . . 153
8.2 Completeness of the SSA validator . . . 157
8.2.1 Specification of Cytron et al.'s algorithm . . . 157
8.2.2 Building a witness global typing . . . 158
8.2.3 The witness global typing is a correct typing . . . 159
8.3 BMM and $\mathrm{BMM}_{\mathsf{o}}$ are equivalent . . . 161

Bibliography **164**

Résumé étendu en français

Motivations

Contexte général

Nos vies quotidiennes dépendent de plus en plus, sans même parfois que nous nous en rendions compte, de l'utilisation de programmes informatiques. Ces programmes n'ont toutefois pas tous le même niveau de criticité. Par exemple, les programmes embarqués dans les sytèmes bancaires, dans les systèmes de contrôle de vol des avions, ou même dans la chirurgie assistée par ordinateur ou les centrales nucléaires sont appelés *systèmes critiques*: la présence d'erreur durant leur exécution pourrait avoir des conséquences désastreuses, que ce soit en termes de vies humaines, de dégâts écologiques, ou de coût financier. Ce type de programme requiert donc de fortes garanties: leur exécution ne devrait pas échouer, et leur correction fonctionelle devrait être garantie.

De manière générale, nous nous intéressons dans ces travaux à la *vérification formelle de logiciel*, c'est à dire à l'ensemble des techniques et d'outils scientifiques qui permettent d'assurer qu'un logiciel remplit ces exigences. Cela consiste en l'utilisation d'outils mathématiques dans le but de prouver l'absence d'erreur dans les programmes, sans même avoir à les exécuter. Ces techniques, dites *statiques*, comprennent par exemple la vérification de modèle, l'analyse statique, ou la preuve formelle de programme.

La vérification formelle de programme connaît un succès grandissant dans le milieu industriel. Astrée [BCC+03] est un des analyseurs statiques en pointe. En 2003, il a été utilisé pour vérifier l'absence d'erreur dans le code embarqué dans le système de contrôle de vol de l'Airbus A340, un programme C de 132 000 lignes. Un autre exemple est Caveat [BPR+02], qui été appliqué également chez Airbus pour la vérification de programmes C critiques.

Vérifier formellement les compilateurs et les analyseurs

Pour que la garantie apportée par la vérification formelle de programmes soit complète, il convient de considérer les deux problèmes suivants. Tout d'abord, l'outil de vérification est lui-même un programme, il pourrait donc contenir des bugs. Ensuite, les programmes sont généralement vérifiés au niveau source, avant d'être compilés en code machine exécutables. Le compilateur utilisé pourrait également introduire des bugs lors de la phase de compilation.

Ces problèmes sont d'autant plus importants que les compilateurs et vérifieurs modernes sont des programmes très complexes, de part les langages très haut niveau qu'ils doivent traiter, la difficulté des propriétés à analyser, et les besoin en performance qu'ils doivent en plus assurer (le temps d'analyse ou de compilation devrait rester raisonable).

Parce que les compilateurs et les vérifieurs interviennent dans la génération ou la vérification de programmes critiques, ils demandent le même niveau de garantie que les programmes critiques eux-mêmes. Le principe est donc d'appliquer la vérification formelle à ces outils. La vérification formelle des compilateurs et vérifieurs est le sujet général auquel nous nous intéressons dans cette thèse.

Les représentation intermediaires de programmes

Les compilateurs et vérifieurs de programmes sont des programmes complexes. Pour simplifier l'analyse et la transformation de code, ils font appel à des *représentations intermédiaires* (IR) de programmes. Par exemple, la chaîne de compilation est décomposée en plusieurs étapes clef, chacune traitant un aspect particulier du langage. Ainsi, le programme source n'est compilé en un code machine que progressivement. Typiquement, les constructions redondantes du langage sont d'abord unifiées; puis, les expressions riches sont décomposées en expressions de base, auxquelles on peut faire correspondre des instructions du processeur cible; plus tard a lieu l'allocation de registres, où la contrainte du nombre de registres physiques du processeur est prise en compte. A chaque phase de la chaîne de compilation correspond une représentation, ou un langage, intermédiaire. Certaines IRs sont particulièrement bien adaptées à l'optimisation et la transformation de code. Les IRs sont aussi beaucoup utilisées dans les vérifieurs et analyseurs de programmes, qui se sont inspirés des choix effectués dans les compilateurs. Parfois, les vérifieurs définissent des IRs spécifiques à leur besoin, en les adaptant par exemple au type de propriétés à analyser. Les IRs sont alors conçues pour rendre moins coûteux les calculs à effectuer sur les programmes.

Le Chapitre 2 de ce document est dédié à la notion générale de représentation intermédiaires. Nous y donnons un aperçu, nécessairement partiel, des IRs utilisées dans les compilateurs et vérifieurs modernes. Nous présentons leur avantages, et aussi leur inconvénients ou les difficultés posées par leur utilisation. Il n'existe en effet pas d'IR qui soit adaptée à tout type d'analyse, d'optimisation, ou de transformation. Ce chapitre tente, par le biais de cet aperçu, de caractériser, ou définir ce qu'est une IR. Essentiellement, une IR est un langage orienté analyse (car la transformation de programme requiert la plupart du temps une forme d'analyse, aussi simple soit elle), qui possède des propriétés structurelles et sémantiques.

Fondements sémantiques des IRs

Le point de vue que nous défendons dans ces travaux est que la preuve formelle des compilateurs et des vérifieurs ou analyseurs réalistes, c'est à dire tels que ceux qui sont utilisés en pratique (avec une distance idéallement nulle entre la formalisation et le code propre de ces programmes), ne peut être envisagée sans considérer les IRs comme levier. En effet, si les IRs sont utilisées en pratique, ce sont pour leur propriétés sémantiques. Ainsi, les IRs ne simplifient pas uniquement l'implantation des algorithmes de transformation ou d'analyse, mais elles devraient aussi en simplifier la preuve de correction.

Pour soutenir ce point de vue, nous étudions d'un point de vue sémantique et formel les IRs. Nous sommes évidemment contraints d'effectuer un choix dans les études de cas que nous menons. Aussi, nous choissisons d'étudier trois aspects ou problématiques récurrentes, que nous détaillons dans la section suivante.

Pour chacun des cas d'étude, notre approche est la suivante. Nous voulons d'abord formaliser la sémantique de l'IR. Cette formalisation doit être fidèle à la réalité, ou capturer l'intuition généralement admise dans la littérature. Puis, nous formalisons les algorithmes de génération de ces IRs, et les prouvons corrects vis à vis d'un critère sémantique adapté (il arrive qu'une IR modifie en un certain sens la sémantique du programme initial). Les algorithmes formalisés doivent être à l'état de l'art, pour s'inscrire dans notre démarche. Un autre objectif pour ces formalisations est de pouvoir les utiliser pour prouver des analyses ou des optimisations correctes. Cela demande que la sémantique soit suffisamment simple à utiliser.

Egalement, cela requiert d'identifier les propriétés sémantiques des IRs, et de les isoler dans des lemmes sémantiques, facilement applicables. Finalement, nous validerons nos formalisations de manière expérimentale, vis à vis de critères de succès tels que le temps de génération, ou l'efficacité du code généré.

Le Chapitre 3 rappelle les notions de base de sémantique formelle. Nous y discutons également différents critères de préservation sémantique pour les transformations de programmes. Finalement, nous rappelons aussi la technique de preuve principale que nous utilisons dans notre travail, qui repose sur des diagrammes de simulation entre systèmes de transition.

Résumé des contributions

Une IR basée registres pour le bytecode Java

Le Chapitre 4 présente notre travail sur BIR, une IR basée registres pour le bytecode Java. L'algorithme de génération de l'IR utilise une technique d'évaluation symbolique du bytecode, inspirée du travail de Whaley [Wha99] pour le compilateur optimisant de la machine virtuelle Jikes RVM [Jik]. Dans cette IR, les arbres d'expressions sont reconstruits tout en restant sans effet de bord. De plus, le processus de création d'objet, i.e. allocation de l'objet, construction des paramètres d'initialisation, puis appel du constructeur, est décompilé en une seule instruction.

Nous présentons la preuve de correction sémantique de la transformation. Notre théorème sémantique explicite ce que la transformation préserve (comme l'initialisation des objets, et l'ordre de levée des exceptions) mais aussi ce qu'elle modifie et comment (par exemple, l'ordre d'allocation des objets).

Nous avons implémenté l'IR dans Sawja, un outil de développement d'analyses statiques pour Java développé au sein de l'équie projet Celtique, et une évaluation empirique de ses performances, en terme d'efficacité de génération, et de compacité du code obtenu (mesuré en nombre de variables temporaires introduites pour les besoins de la transformation). Nous veillons à ce que la formalisation modélise fidèlement la version implanté de la transformation.

La représentation SSA

Puis, nous nous intéressons à la forme Single Static Assignment (SSA), une IR au coeur des compilateurs et vérifieurs modernes. Dans cette IR, chaque variable du programme ne possède qu'un unique point de définition (statique). Chaque point de définition correspond ainsi à une nouvelle version de la variable initiale. Aux points de jonction du graphe de flôt de contrôle, des instructions virtuelles, les *phi-instructions*, sont utilisées pour assurer la sélection de la bonne version de variable, selon le chemin suivi par le flôt d'exécution. Bien que la forme SSA soit beaucoup utilisée en pratique, les travaux de formalisation de sa sémantique et de ses propriétés restent encore insuffisamment développés. Typiquement, l'intuition selon laquelle la forme SSA serait une forme equationnelle de programme n'a jamais été formalisée, ni même clairement énoncée.

Dans le Chapitre 5, nous présentons les contributions suivantes. Nous implantons et prouvons correct dans l'assistant de preuve Coq un middle-end SSA pour le compilateur C CompCert. Ceci comprend donc la formalisation de la sémantique de SSA, ainsi que la preuve de préservation sémantique des algorithmes de génération et de destruction associés. Notamment, nous utilisons la technique de validation a posteriori pour l'algorithme de construction

de la forme SSA. Nous avons aussi implanté et prouvé correcte une optimisation typique de la forme SSA, l'Elimination des Sous-expressions Communes (CSE), basée sur une technique de Numérotation Globale des Valeurs [AWZ88] (GVN).

Nous identifions et prouvons l'ingrédient clef pour la preuve de ce type d'optimisation, c'est à dire l'invariant global sémantique de SSA permettant de voir les programmes SSA comme définissant un ensemble d'équations. Notamment, nous montrons que les égalités ainsi "générées" par SSA ne sont pas valides à tout endroit du programme, mais uniquement dans les régions strictement dominées par les points de définition des variables impliquées dans les égalités.

Grâce au mécanisme d'extraction fourni par Coq, nous obtenons une version OCaml du middle-end formalisé, complètement exécutable, et qui satisfait les propriétés que nous prouvons à son propos. Ceci nous permet également de conduire quelques expériences pour évaluer les performances du middle-end. Tout d'abord, le validateur SSA a un temps de calcul raisonable par rapport au temps de génération de la forme SSA. Ensuite, nous observons que l'optimisation CSE à base de GVN est capable d'optimiser davantage (en nombre d'instructions) l'ensemble de programmes que nous considérons. Finalement, les temps d'exécution des programmes compilés n'est pas drastiquement amélioré, mais les résultats préliminaires sont encourageants.

Un modèle mémoire pour les IRs de Java concurrent

Enfin, nous étudions un aspect difficile des languages modernes auquel les compilateurs et les analyseurs modernes devront faire face de plus en plus, la concurrence. IdÃľalement, la sémantique d'un language concurrent, dans lequel les fils d'exécution communiquent au travers d'une mémoire partagée, pourrait être modélisée par un simple entrelacement des fils d'exécution; ceux-ci auraient à chaque étape de calcul une vision cohérente de la mémoire. Cependant, pour certains langages, ce modèle dit *séquentiellement consistant* ne suffit pas à décrire tous les comportements des programmes. En effet, les architectures multi-processeurs modernes utilisent des mécanismes de tampons, de caches mémoire, et des mécanismes de spéculation, qui invalident ce modèle, car ils introduisent des comportements que l'on pourrait observer si certaines instructions du programme étaient exécutées dans un ordre différent de celui du code programme. Ces modèles de mémoire sont dits *faiblement consistants*. Nous donnons au début du Chapitre 6 une brève introduction à cette problématique, dans le cadre de la vérification de compilateurs.

Dans ce chapitre, nous nous penchons sur le cas du modèle mémoire Java (JMM), lui aussi faiblement consistant. Il a été formalisé en 2005 par Manson et al. [MPA05]. Les garanties de sécurité que se veut offrir Java demandent de définir aussi une sémantique pour les programmes dont les fils d'exécution peuvent entrer en conflit entre eux, lors d'accès simultanés à la mémoire partagée. Ceci complique la définition du JMM, puisque les relaxations du modèle mémoire apparaissent pour de tels programmes. La définition actuelle du Java Memory Model (JMM) autorise également les optimisations aggressives des compilateurs et des architectures parallèles sous-jacentes, sans faire a priori d'hypothèse sur une architecture cible particulière.

Ainsi, la sémantique des programmes Java concurrent est, strictement parlant, définie. Toutefois, cette sémantique est très complexe, trop pour pouvoir envisager, à ce jour, de manière raisonable, la preuve d'un compilateur vis à vis d'elle; elle fait d'ailleurs encore l'objet de travaux de recherche récents [AŠ07b, Šev08, TVD10]. Ces travaux montrent que cette définition n'est pas celle attendue (elle autorise des optimisations qui ne devraient pas être

permises, et en interdit d'autres qui le devraient), terriblement complexe, et qu'un compilateur actuel de référence, Sun Hotspot JVM, n'est pas conforme vis à vis de cette sémantique.

Devant ce constat, nous choisissons de fixer une famille d'architectures cibles, TSO (Total Store Order), dont le modèle de mémoire, bien que faiblement consistant, reste suffisamment simple pour envisager une preuve formelle de compilateur vis à vis de lui, comme en témoigne le travail de Ševčík et al. [ŠVZN+11]. Nous proposons un modèle de mémoire Java spécialisé pour ces architectures. Nous le caractérisons axiomatiquement par les réordonnancements d'actions mémoire qu'il autorise, et montrons qu'il constitue un sous-ensemble du JMM. Ainsi, la solution proposée est sémantiquement valide, consistant en un raffinement sémantique des programmes. Dans le but de pouvoir conduire des preuves formelles dans la lignée de Ševčík et al., nous montrons également qu'il peut être défini de manière opérationnelle, et prouvons l'équivalence avec le modèle axiomatique. Ces preuves s'appliquent à toutes les couches objet du compilateur, étant donné que le modèle mémoire est paramétrisé par la sémantique intra-thread (les actions locales aux threads) des programmes. Finalement, une validation expérimentale préliminaire a été conduite, montrant que, sur les architectures TSO, en choisissant avec soin les optimisations du compilateur, le coût d'implanter ce modèle est raisonnable, en comparaison avec le JMM.

Notes à propos des chapitres 4, 5, and 6

Matériel disponible en ligne. Du matériel additionnel pour ces chapitres est disponible en ligne. Les liens spécifiques sont précisés dans les chapitres.

Preuves. Ces trois chapitres contiennent des résumés étendus des preuves réalisées pendant cette thèse. Les preuves complètes des résultats apparaissent, soit en Annexe de ce document, soit dans les documents supplémentaires disponibles en ligne.

Publications. La contribution présentée dans le Chapitre 4 a été publiée dans les actes de la conférence internationale 8th Asian Symposium on Programming Languages and Systems (APLAS'10) et présentée à Shanghai en Décembre 2010 [DJP10]. La présentation technique plus complète de l'outil Sawja a été publiée dans les actes de la conférence internationale Conference on Formal Verification of Object-Oriented Software (FoVeOOS'10) [HBB+11]. Ont contribué à l'implémentation de Sawja et Javalib, en ordre alphabétique : Etienne André, Nicolas Barré, Frédéric Besson, Nicolas Cannasse, Delphine Demange, Laurent Hubert, Florent Kirchner, Vincent Monfort, David Pichardie et Tiphaine Turpin.

La contribution présentée dans le Chapitre 5 a été publiée dans les actes de la conférence internationale 21th European Symposium on Programming (ESOP'12) et présentée à Tallin en Mars 2012 [BDP12]. La preuve mécanisée en Coq de l'optimisation CSE basée sur GVN est principalement due à David Pichardie.

La contribution présentée en Chapitre 6 a été acceptée pour publication dans les actes de la conférence 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13) [DLZ+13]. L'évaluation empirique, dont un résumé des résultats est présenté dans ce chapitre dans un souci de complétude, a été conduite par Vincent Laporte et Lei Zhao.

Chapter 1

Introduction

1.1 Formal verification of software

Our daily lives rely more and more on the use of computer programs. Sometimes we do not even realize it. These programs are embedded in systems ranging from alarm clocks, washing machines, elevators and vending machines to smart cards, mobile banking, smart phones, automated driverless subways, computer-assisted surgery, aircraft flight control systems, or nuclear power plants. Obviously, these programs do not all have the same level of criticality. Among them, we distinguish *safety-critical software*, the category of programs whose malfunction can result in considerable losses, either financially or humanly. This category of software demands a high level of confidence in its execution. No error should arise at runtime (the system should not crash) and their functional correctness should be guaranteed (the system should compute the right information).

Formal software verification is a set of scientific techniques and tools that can be used to ensure that a software system fulfills these requirements. It consists in using mathematical tools for *proving the absence of bugs* in programs without actually executing the programs. These are *static* verification techniques that are based on either model-checking, static analysis or program proof. Formal software verification is becoming increasingly popular in the safety-critical industry. Astrée [BCC+03] is one of the leading static analyzers of safety-critical embedded software. In 2003, it was able to verify the absence of certain classes of errors in the primary flight control software of the Airbus A340 fly-by-wire system, a program of 132,000 lines of C code. Another example is Caveat [BPR+02], that has been applied at Airbus to verify safety-critical C programs.

To achieve an end-to-end guarantee, there are two sources of bugs that must be accounted for. First, there might be errors in the verification tool itself. Second, the formal verification is usually applied at the source level of the program, before it is compiled to an executable machine code. In the end, what matters is that the compiled program (the one who is executed) fulfills these requirements. Thus, the compilation should also be trustworthy.

We now discuss these two issues, and then how to tackle them. This will lead us to the topic of our thesis: the *intermediate representations* of programs on which these tools are built.

1.1.1 Compilers and program verifiers

Compilers and program verifiers are equally important potential sources of bugs due to their complexity.

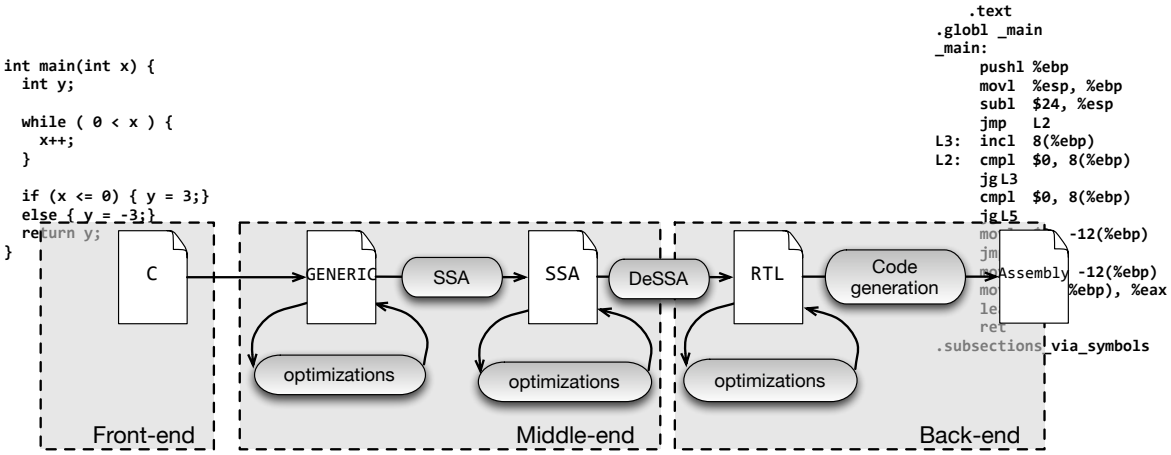


Figure 1.1: Distance between what the programmer writes and what the machine executes. A C source program (left) written by a programmer compiled by GCC into the assembly program (right).

Compilation errors Compilers are complex pieces of software for several reasons. First, they must translate the rich features of high-level languages, e.g. exception handling, object creation, function pointers or automatic memory management, down to the processor instruction set. In order to simplify this process, the compilation chain is split into several phases, each translation step focusing on a particular feature of the language to compile. To each step of the compilation corresponds an intermediate representation (IR) of the program.

Figure 1.1 shows an example of source program written in C (left), and the corresponding x86 assembly code (right) generated by the GCC compiler [FSF]. GCC's IRs mainly comprise GENERIC, SSA (Static Single Assignment) and RTL (Register Transfer Language) formats. We will come back to these IRs in the rest of this document. Second, the generated code should be efficient, so the compiler performs code optimization. In Figure 1.1, optimizations are performed at each stage of the chain. But some IRs are particularly well suited to program optimization. SSA [AWZ88, CFR+91] is one of them. It comes with strong properties (in particular, variables definition points are unique) that analyses and optimizations can exploit for more precision and efficiency. The SSA optimization phase of GCC comprises around 100 passes [FSF]. Finally, the compilation time should be reasonable. The code of the compiler itself is thus usually optimized, in terms of algorithms and data structures.

The inherent complexity of compilers make their implementation error-prone. This can result in the failure of the compiler to process the input program: the compiler crashes. A more problematic outcome is miscompilation: the compiler succeeds to produce an output program, but this program behaves differently from the source program. The recent work of Yang et al. [YCER11] shows, using a randomized test-case generator, that these errors remain frequent in mainstream C compilers. They report:

> *We found and reported hundreds of previously unknown bugs [...]. Many of the bugs we found cause a compiler to emit incorrect code without any warning. 25 of the bugs we reported against GCC were classified as release-blocking.*

Verification errors Formal verification relies on the use of mathematical tools for (i) stating the *specification* of the program, i.e. its expected behavior (ii) modelling the behaviors of the

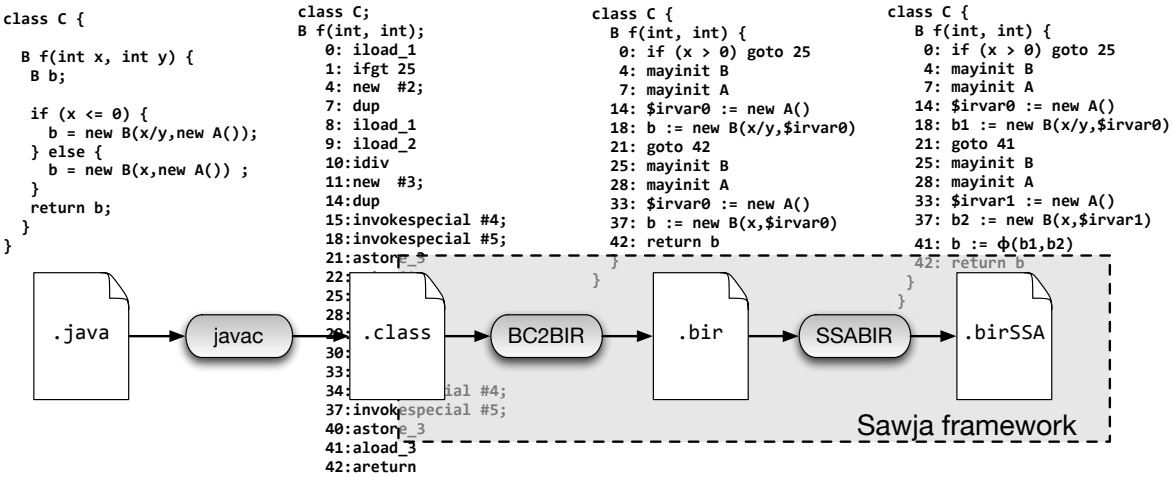


Figure 1.2: Java bytecode analysis and the IRs provided by the Sawja framework. A Java program (`.java`) is compiled into a Java bytecode program (`.class`), executable by the Java Virtual Machine. Some analyses cannot be performed at the source level (e.g. the source code could be unavailable to the client). Sawja provides high-level IRs of the bytecode to simplify the design and implementation of static analyses.

program, or its *semantics* and (iii) proving that the semantics of the program conforms to its specification. Once the program has been formally verified, it is mathematically ensured, relative to the model of the environment, not to crash, and to always – meaning for all the possible input parameters – yield a correct result, with regard to a formal specification. Formal verification techniques are numerous. They include model-checking, static analysis, or program proof. In this work, we consider program static analyzers and program proofs.

Static analyzers, automatically check that the program execution is free of run-time errors. Examples include the absence of division by zero, null pointer dereferencing, out-of-bound array accesses, or arithmetic overflows. The absence of run-time errors in a program is not decidable in general, so static analyzers must over-approximate the set of possible program executions into a domain in which properties become decidable. For instance, a set of values is abstracted into an interval of integers. In return, a coarse approximation can make the analyzer raise false-alarms. Program provers use deductive reasoning techniques, such as Hoare logic, to prove that programs satisfy their specification. Usually, these tools are less automatic than static analysis. The program is first annotated with its specification (that takes the form of pre- and post-conditions, as well as loop invariants). These annotations can express high level properties, such as the functional correctness of the program (e.g. this program sorts a list). Then, some verification conditions are automatically generated, and then solved either automatically or manually.

Program verification tools are programs too, and their implementation is a challenging task, for several reasons. Consider program analyzers. They should first be accurate, i.e. the abstraction should be precise, so that the analyzer does not raise too many false-alarms. Precise abstract domains, such as polyhedra, are algorithmically costly, and thus complex to implement efficiently and correctly. The APRON library [JM09] provides such features; the domains of convex polyhedra and linear equalities represent a total of 12,000 lines of delicate and optimized C code. Also, some analyses, such as the analysis of resource consumption or

worst-case execution time, need to be done on low-level code, where the timing information about the hardware and all its specific features are known. Low-level code can be harder to manipulate and to reason about in analyzers.

Increasingly, verification tools also rely on program transformations and IRs to either simplify their implementation design, or gain precision and efficiency. They either reuse the IRs from the compiler community, e.g. SSA, or define their own ones, specifically tailored to the language or the analyses to perform. We give some examples of such IRs and verifiers in Chapter 2. Figure 1.2 illustrates the use of Sawja, an OCaml library developed in the Celtique research group for developing Java Bytecode static analyzers. It provides high level IRs of the bytecode (BIR and an SSA form of BIR) which simplify the design and implementation of bytecode static analyses. In particular, these IRs are register-based, and avoid the need of reasoning on the operand stack of the bytecode. This kind of IR is very popular in low level code analysis frameworks (e.g. Soot [VRCG+99] or Wala [Fin]).

1.1.2 Verified compilers and verifiers

Compilers and verification tools are thus complex programs. When they are used for compiling and analyzing critical software, they require the same level of confidence as the critical software itself. In order to reach a high-level of guarantee, the idea is then to apply formal verification techniques to the compilers and analyzers themselves.

They both manipulate the same notions: programs and semantics. Proving their correctness hence follows a similar pattern. First, the semantics of programs is formalized with e.g. transition systems. Then, the algorithms used in the compiler or analyzer are rigorously proved with respect to these formal semantics, using results coming from graph and lattice theory, Hoare logic, data-flow analyses [Kil73, NNH99], or abstract interpretation [CC77].

Trusting proofs Again, there can be a gap between the algorithm that is formalized and its actual implementation. To solve this issue, a more radical approach consists in conducting the proofs with the help of proof-assistants, such as Coq [CDT] or Isabelle/HOL [NPW]. These programming environments allows for writing programs, their specification, and the corresponding correctness proof in a unified, logical framework that ensures that the proofs are valid. In this case, the validity of the proofs only relies on the proof checking kernel of the assistant, whose correctness is apparent thanks to its small size and its relative simplicity. Besides, proof-assistants provide an *extraction mechanism* that automatically generates executable code that fulfills the formalized specification. The extraction mechanism is a key feature of proof-assistants, as it amounts to performing the proof directly on the compiler or verifier implementation code.

Compiler verification Compiler verification aims at providing a formal, i.e. mathematically-grounded, proof that the compiler does not insert bugs in the compiled program. More specifically, a compiler should be proved to *preserve the semantics of programs*, i.e. their dynamic behaviors. Compiler verification is not a new problem, and has a 40-years history, starting with the work of McCarthy and Painter [MP67] on the correctness of a compiler for arithmetic expressions. In 1973, Morris proposed in [Mor73] a methodology for proving the correctness of real size compilers using simulation diagrams. Moore [Moo89, Moo96] provided in the 90's the first machine-assisted proof of a compiler for a high-level assembly language. The compilation of Java has also been formally studied. Stark et al. [SBS01] provide an on-paper

formalization for a subset of source and target language, and a simplified compilation scheme. Strecker [Str02] and Klein and Nipkow [KN06] formalized Java-like non-optimizing compilers to a Java-like bytecode in Isabelle. The Verisoft project [LPP05] provides a simple compiler for C0, a type-safe fragment of C. The CompCert C compiler of Leroy et al. [Ler09, Ler12] is the most advanced and realistic formally verified compiler. It handles a large subset of C[1], provides some carefully chosen optimizations – e.g. tail-call detection, constant propagation, common subexpression elimination with memory variables or function inlining – and produces assembly code for realistic processors (PowerPC, x86 and ARM). Ševčík et al. [ŠVZN+11] provide an adaptation of CompCert for C with concurrency primitives for thread managment and synchonization. Chlipala [Chl10] formalized in Coq a compiler from a small, untyped functional language with mutable references and exceptions to an idealized assembly language. Finally, the Vellvm project [ZNMZ12] aims at verifying in Coq components of the LLVM compiler [LLV].

Analyzer verification Analyzers prove the absence of runtime errors only if the abstraction they rely on is an over-approximation of the possible concrete values – they may otherwise miss some bugs. This correctness criteria is called the *semantic soundness* of the verifiers. Proof assistants become increasingly popular for proving realistic analyses. For instance, the Java Bytecode Verifier [LY99], a data-flow analysis for type-checking Java Virtual Machine bytecode programs, has been formalized and proved sound in Isabelle by Klein and Nipkow [KN06] for a subset of the language. Other examples of realistic analyses on Java bytecode include the work of Dabrowski et al. [DP09] on a static data-race analysis implemented and verified in Coq, and Barthe et al. [BPR07] on a type system for detecting unsecure information flows. Appel et al. [App11, SBA12] provide a formally verified analysis of shape properties about programs' heap-data. The analysis is done on Cminor, an IR of C provided by CompCert, where expressions are side-effect free, the evaluation order is fixed, control structures are simplified, and local variables' addresses cannot be taken anymore.

1.2 Intermediate representations to the rescue

Viewed broadly, our work aims at studying the formal proofs of compilers and verifiers. The goal of this work is not to provide new optimizations techniques for compilers, nor to design new static analyzers. Rather, we aim at finding appropriate formalizations and proof methods of existing techniques that have demonstrated their usefulness.

Our work is grounded on the observation that these tools heavily rely on intermediate representations. We believe that formalizing these IRs is a key element for the formal verification of compilers and analyzers. The notion of IR and the benefits it brings will be discussed in more detail in a dedicated chapter, but the previous examples already give an overview: they allow numerous front-ends, aggressive optimizations or fast, simple and precise analyses. The approach followed in CompCert confirms this need for formalized IRs. Its compilation chain comprises 11 differents IRs, that is twice as many IRs as used in GCC. The proof of correctness would not have been possible without splitting the compilation chain in such elementary steps.

We argue that IRs should be exploited more by formal verified compilers and verifiers. Some recent techniques used by mainstream compilers, mostly related to optimization, are still missing in the picture. Compared to GCC or LLVM, CompCert does not include an SSA

[1] The only missing features in the current version of CompCert are unstructured `switch`es, unprototyped, long long arithmetic and variable-argument functions. `longjmp` and `setjmp` statements are partially handled.

form in its middle-end. The gap is even wider for formally verified analyses. For instance, the analysis presented in [BPR07] is performed at the bytecode level. Keeping track of information flow in a bytecode program requires defining a subtle notion of undistinguishability both on the operand stack and local registers, that must be handled differently in the analysis. Handling the operand stack is difficult in the analysis itself, and hence considerably complicates its soundness proof. Relying on an IR such as the one provided by Sawja could help scaling the analysis (and its proof) to the full language. The data-race analysis of [DP09] must keep track of a flow-sensitive alias information on local registers. An interesting property one gets with the SSA form is that flow-insensitive analyses have the same precision as flow-sensitive ones, thus allowing to implement simpler analyses, that are easier to prove.

The thesis we defend in this dissertation is the following:

> *"The intermediate program representations used in modern compilers and analyzers can be faithfully formalized and their formalization can be leveraged to simplify the proof of analyses and optimizations."*

To support our claim, we study from a formal point of view the semantic foundations of IRs, with an emphasis on real-size languages and modern techniques. Obviously, the spectrum of existing IRs that are used in the analysis and compilation communities is too large, so we focus on three representative cases. The first IR we study is BIR, the stackless Java bytecode IR introduced in Figure 1.2, where the use of the operand stack is replaced with local variables. The second IR we study is SSA, whose apparently simple property of unique variable definition points triggers many semantic invariants. Finally, we investigate a feature that modern compilers and analyzers have to face: concurrency on multiprocessor architectures. From a syntactic point of view, concurrent IR languages differ from sequential IR languages only by a few extra instructions. However, their semantics are far more complex, in particular with respect to shared memory accesses. We focus on concurrent Java IRs. Their semantics are dictated by the so-called Java memory model [MPA05], whose definition is so complex that reasoning formally about it is still an active field of reasearch [AŠ07b, Šev08, TVD10].

More specifically, for each of these case studies, our objectives are the following:

IR semantics. Capturing the intuition that compiler writers have about an IR, and to reflect this intuition in the formalization. The semantics should also be amenable to formal proof, i.e. it shoud be easy to reason formally about it.

IR properties. Stating adequate semantics preservation results for the generation algorithms, and understanding what makes the IR valuable. This means identifying and formalizing its properties, and leveraging them in the proof of subsequent optimizations or analyses.

IR generation. Providing realistic versions of the formalized generation algorithms, to confront our formalizations with practical considerations of performance, and making them usable in compilers or analysis frameworks.

1.3 Contributions and structure of the document

The first two chapters are introductory. Chapter 2 is dedicated to the general notion of intermediate representation and gives a (necessarily incomplete) overview of the leading IRs used in modern compilers or program verification tools. In Chapter 3, we recall the basic notions of

formal semantics, and discuss the possible semantics preservation criteria for program transformations. We also recall the proof technique, based on simulation diagrams, that we will use throughout this work.

Chapter 4 presents our work on BIR. Its conversion algorithm is based on a symbolic execution of the bytecode, a technique that we borrow from the work of Whaley [Wha99] on the optimizing compiler of the Jikes RVM [Jik]. This IR aims at reconstructing side-effect free expressions, and simplifying the object creation scheme that is used in the bytecode. We present its formal correctness proof, which is not mechanized, but rigorous. We also present its implementation within Sawja, and an experimental validation of its performance. We have taken care that the implementation follows closely our formalization.

In Chapter 5, we provide a formally verified SSA-based compiler middle-end. It includes the formalization of the semantics of SSA, and its generation and deconstruction algorithms. We also have implemented and proved a typical and challenging SSA-based optimization, Common-Subexpression Elimination (CSE) based on Global Value Numbering [AWZ88] (GVN). The proof relies on key properties of the SSA form, that we identify and prove as global invariants of the IR. All this formalization work has been done within the Coq proof assistant, relying on the CompCert C compiler. Thanks to the extraction mechanism provided by Coq, we obtain an OCaml version of our middle-end. We provide some experimental results about its performance.

In Chapter 6, after an introduction to the field of *weak memory models*, we identify a subset of the Java memory model for which we can capture the intuition that is used in the folklore example-driven presentations of the Java memory model. To do so, we expose the valid reorderings directly in the model. Additionally, we make its definition amenable to formal reasoning in a proof assistant like Coq by relying on an operational characterization. We argue that this model is nonetheless efficiently implementable on TSO multiprocessor architectures.

Chapter 7 concludes this dissertation with a summary of our contributions and implementation results. We also discuss extensions to this work, and future perspectives.

Notes about Chapters 4, 5, and 6

On-line material. Additional material for these chapters can be found on-line, links are indicated in the chapters.

Proofs. These chapters provides some extended proof sketches to give an intuition about the complete proofs, which can be found either in Appendix or in the on-line material.

Publications. The work presented in Chapter 4 is published in the proceedings of the 8th Asian Symposium on Programming Languages and Systems (APLAS'10) [DJP10]. The presentation of the Sawja tool bench is published in the proceedings of the International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'10) [HBB+11]. The contributors to Sawja and Javalib are, in alphabetic order: Etienne André, Nicolas Barré, Frédéric Besson, Nicolas Cannasse, Delphine Demange, Laurent Hubert, Florent Kirchner, Vincent Monfort, David Pichardie and Tiphaine Turpin.

The work in Chapter 5 has been published in the proceedings of the 21th European Symposium on Programming (ESOP'12) [BDP12]. The Coq proof of the GVN-based CSE optimizations is mainly due to David Pichardie.

The work in Chapter 6 has been accepted for publication in the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13) [DLZ+13]. The experimental results are due to Vincent Laporte and Lei Zhao.

Chapter 2

Intermediate representations

As pointed out in Chapter 1, intermediate representations (IRs) are a key component of compilers and analyzers. Their variety is explained by the diversity of source and target languages, of the program properties to analyze, and on the numerous existing verification techniques. Nevertheless, an IR is not just yet another language, some characteristics can be identified. This chapter aims at giving a more precise definition of what IRs are. We then review six leading IRs that are particularly relevant to our purpose, give the main features that make them used in practice, and also point out their main limitations. This necessarily incomplete overview will help us refining the informal definition of intermediate representation, and motivating our work upon the limitations we observed.

2.1 A first informal definition

What is an IR? To answer the question, we first recall the two main use cases of an IR.

Elementary transformation step Historically speaking, compilers were initially not optimizing: the first task that a compiler had to accomplish was the translation of human-readable code into machine code. Grace Hopper in 1952 developed the very first compiler for the A-0 system language [Hop52], a simple language for arithmetic expressions. It was only later in 1957, that the Fortran team led by John W. Backus at IBM introduced the first complete compiler [BBB+57], with the clear intention of producing machine code that could compete, in terms of quality, with hand-written machine code. Thus, IRs were originally introduced to *split the compilation chain* in elementary steps, each phase focusing on one aspect of the language only. A whole compilation process comprises several internal representations of the code (e.g. abstract syntax tree, control-flow graph or instruction array), but more importantly several languages and instruction sets, that allows for progressively transforming source code into low-level code. In this case, the representation is intermediate in the sense that it is part of a complex process, and helps its design and implementation.

This role is even more crucial for verified compilers, where IRs represent not only a basic step in the compilation process, but also a *basic step in its proof*. To simplify the proof process, verified compilers thus tend to use more IRs. As an example, the GCC compiler architecture builds on about 6 IRs [FSF], whereas the formally verified CompCert C compiler chain is made of about ten [Ler12].

Right level of abstraction The second role of an IR is to provide a representation of a program at the right level of abstraction. Typically, the compiler middle-end IRs are independent from both the source and target languages. This is what allows a given middle-end to be used by different front-ends and back-ends. GCC for instance comprises front-ends for

C, C++, Objective-C, Fortran or Java. All of these front-ends generate GENERIC code, the highest level IR of the middle-end (see Figure 1.1). The IR is used here as a way to factorize the common phases of a compiler: programs written in different languages can be *normalized* into the IR language. Beyond, what the IR really brings here is an appropriate representation *abstracting all the information irrelevant* for further transformations. In particular, the number of distinct IR statements can be drastically reduced compared to a source language: an IR provides only the basic constructs required to reach a given expressivity — it is syntactically desugared. This also holds for program analyzers: almost all realistic analyzers or verifiers will work on an IR rather than on the source language directly (see Sections 2.2.5 and 2.2.6).

At the light of this discussion, we note that the first characteristic of an IR is that it is not a *programming language*: it is not designed to write programs in this language. An IR is automatically generated by tools, and well suited to further transformations and/or analyses. In this respect, it is an *analysis-oriented* – all non-trivial program transformations requiring a preliminary analysis – and not a *user-oriented* language.

Definition 2.1 (Intermediate representation)**.** *An intermediate representation is an analysis-oriented language, usable by program-processing tools, and designed according to a verification or transformation purpose.*

In order to refine Definition 2.1, we present in the next section six IRs that are widely used in the compilation and analysis communities. For making clearer their differences, we will use the following common source language: While, a simple structured and imperative programming language. Its syntax is given in Figure 2.1, together with a simple program used as a running example. While provides arithmetic and comparison expressions, built on integers constants and variables. While is not typed, but we assume all values are integers. For the sake of simplicity, we do not consider function definitions or function calls.

$$
\begin{array}{rcl}
expr & ::= & const \mid var \mid expr\ op\ expr \\
op & ::= & + \mid \leq \\
stmt & ::= & \texttt{skip} \mid x := expr \mid stmt\ ;\ stmt \\
 & & \mid \texttt{if}\ expr\ \texttt{then}\ stmt\ \texttt{else}\ stmt \\
 & & \mid \texttt{while}\ expr\ \{\ stmt\ \} \\
 & & \mid \texttt{return}\ e \\
prog & ::= & \texttt{prog}\ name\ (var, \ldots, var)\{\ \ stmt\ \ \}
\end{array}
$$

(a) Syntax

```
prog foo (n, t) {
  r := 0 ;
  i := 0 ;
  while (i ≤ n) {
    r := r+i ;
    i := i+t
  };
  return r
}
```

(b) Example program

Figure 2.1: While programming language

2.2 Some leading IRs

The IRs we present here fall into two categories. TAC, STACK, SSA and CPS are *elementary*: they can be thought of independently, and can be used as basic units of more complex IRs found in compilers or analyzers. The two last IRs we present are *rich* IRs; they are used in the Costa analysis system and the Boogie program verifier respectively.

In the following, we will stay as much as possible at the concrete syntax level, abstracting the internal representation of IRs (e.g. instruction vectors or graphs). The IR internal representation is undoubtely crucial for the ease of code manipulation and traversal (and thus efficiency) but this issue is quite orthogonal to what this chapter aims at presenting, i.e. *what* information is embedded in IRs, rather than *how* it is represented.

2.2.1 Three-address code: TAC

TAC stands for *Three-Address Code*. It is a *virtual register transfer* representation. It is one of the most common IRs, described in all compiler textbooks [ASU86, Muc97, App98a]. Any compiler for an imperative language will use a variant of TAC: it corresponds to RTL in the CompCert C compiler and to low-GIMPLE in GCC [FSF]. The concrete syntax of TAC is given in Figure 2.2a. Figure 2.2b gives the TAC representation of our example program.

$$
\begin{array}{rcl}
\mathit{bexpr} & ::= & \mathit{const} \mid \mathit{var} \\
\mathit{expr} & ::= & \mathit{bexpr} \mid \mathit{bexpr}\ \mathit{op}\ \mathit{bexpr} \\
\mathit{op} & ::= & + \mid > \\
\mathit{stmt} & ::= & \texttt{goto}\ \ell \mid \mathit{var} := \mathit{expr} \\
 & & \mid \texttt{if}\ \mathit{bexpr}\ \texttt{goto}\ \ell \\
 & & \mid \texttt{return}\ \mathit{var} \\
\mathit{prog} & ::= & \texttt{prog}\ \mathit{name}\ (\mathit{var}, \ldots, \mathit{var})\{\ \ (\ell : \mathit{stmt})\ \mathit{list}\ \ \}
\end{array}
$$

(a) Syntax

```
prog foo (n, t) {
  1: r := 0
  2: i := 0
  3: c := i > n
  4: if c goto 8
  5: r := r+i
  6: i := i+t
  7: goto 3
  8: return r
}
```

(b) Example program

Figure 2.2: TAC language

Syntax Each instruction represents exactly one fundamental operation: expression trees have height at most one, so that each statement references at most three virtual registers (e.g. one result and two arguments). Hence the name *three-address code* in the literature. In Figure 2.2b, notice that a new temporary variable c must be used to store the result of the comparison $i > n$, since the condition must be a basic expression. Similarly, complex While assignments like `x:= a+b+c-1` would be decomposed in TAC as e.g. `t1:=a+b;t2:=t1+c;t3:=t2-1`.

TAC is not structured: while-loops and if-statements are encoded using conditional and unconditional jumps only. In Figure 2.2b the while-loop is encoded by the instructions between program points 3 and 7. While could have included many other kinds of loops (such as for-loop, or do-while loops) without TAC needing any other statement.

Expressions and control lowering In TAC, the evaluation order of expressions has been determined thanks to the decomposition of computations into elementary expressions. They also match better the machine instruction format (there is no hidden computation). Control structures have also been lowered[1], and are closer to the assembly representation.

[1] Code lowering refers to the process of compiling code to a language of a lower level, with fewer and more basic constructs.

Optimization is possible TAC is low-level but expressions are still processor-independent, and expression computations can still be modified or reordered by optimizations. In particular, source and target operands may refer to any virtual register, available in an unbounded number. Optimizations of expression computations (common sub-expressions or partial redundancy elimination) can therefore be performed on TAC without dealing with register constraints.

The fact that TAC is unstructured can have a non-negligible impact on analyses and optimizations. Indeed, the iteration strategy for solving data-flow equations can no longer be syntax directed (as in While), as the CFG might become arbitrary, i.e. non-reducible. In this case, another iteration strategy must be adopted to solve the equations in an efficient way (see e.g. [Bou93]). Unstructured programs can still have reducible CFG, and this can be exploited to use cheaper iteration strategies (e.g. depth-first-search traversal, interval techniques, or dominators [ASU86]).

Precise analyses remain difficult As pointed out by Logozzo and Fähndrich in [LF08], basic expressions of TAC may be difficult to handle in static analyses. We illustrate this issue with an interval analysis, that computes, at each program point, the conservative value range (or value interval) for each variable. Consider the conditional statement ℓ`:if x goto` ℓ', where the variable `x` is defined by e.g. `x := a < b`. Suppose that the analysis (soundly) infers at point ℓ the information $\langle a \mapsto [10;15], b \mapsto [8;12], x \mapsto [0;1]\rangle$. Because of the basic expression `x` in the conditional at point ℓ, the analyzer does not have any information about the two variables a and b involved in the value of the test – it only knows the abstract value of x (here the interval $[0;1]$). It is hence unable to refine the intervals of a and b inside the conditional branch at point ℓ', although under the hypothesis that $a \geq b$, one could deduce a strictly more precise interval for a, i.e $a \mapsto [12;15]$. Logozzo and Fähndrich discuss the possibility of recovering this information with the help of an additional symbolic abstract domain to keep track of the symbolic expressions stored in program variables. An alternative, frequently adopted in practice [BCF+99, FL11, Fin], is to make this information explicit in the program text, by enriching the IR language with rich expressions, so that the conditional statement becomes `if a<b goto` ℓ'.

To summarize, the main goal of TAC is to lower the control-flow and expressions, while still allowing some optimizations to be performed. It also simplifies compiler retargeting: it is an intermediate step for optimizing the code before further lowering it with e.g. register allocation.

2.2.2 Stack-based code: STACK

STACK is a stack-based representation of the code: it does not include any expression language, but instructions only, that use and modify an operand stack. In this respect, it is a lower-level IR than TAC. Figure 2.3 gives the syntax of STACK and the STACK representation of our example program (some comments indicate the initial While instructions).

Instruction set STACK instructions use and modify an operand stack at run-time. There are three categories of instructions. The first one consists in expression computations: `push c` pushes the (integer) constant `c` on the operand stack, `load` x pushes the value of the local register x, `dup` duplicates the top element of the stack, and `add` pops the two top elements off

$$
\begin{array}{lcl}
\mathit{instr} & ::= & \texttt{push}\ \mathit{const} \\
 & & \mid \texttt{dup} \\
 & & \mid \texttt{add} \\
 & & \mid \texttt{load}\ \mathit{var} \\
 & & \mid \texttt{store}\ \mathit{var} \\
 & & \mid \texttt{goto}\ \ell \\
 & & \mid \texttt{ifg}\ \ell \\
 & & \mid \texttt{return} \\
\mathit{prog} & ::= & \texttt{prog}\ \mathit{name}\ (\mathit{var}, \ldots, \mathit{var})\{ \\
 & & \quad (\ell : \mathit{instr})\ \mathit{list} \\
 & & \}
\end{array}
$$

(a) Syntax

```
prog foo (n, t) {
   1: push 0
   2: dup
   3: store r   // r := 0
   4: store i   // i := 0
   5: load i    // Loop header
   6: load n
   7: ifg 17    // if i>n goto 17
   8: load r
   9: load i
  10: add
  11: store r   // r:= r+i
  12: load i
  13: load t
  14: add
  15: store i   // i:= t+i
  16: goto 5    // End of loop body
  17: load r
  18: return
}
```

(b) Example program

Figure 2.3: STACK language

the stack, and pushes back the result of their addition. Second are the instructions modifying the memory (here only local registers): `store` x pops the top value of the stack, and assigns this value to the local register x. Finally, the control-flow is handled with `goto`s and `ifg` l which compares (*if greater*) the top two elements of the stack and branches to program point l if the comparison holds, or to the next instruction otherwise.

Light-weight code Stack-based representations are usually employed in the context of virtual machines (VM): the Smalltalk virtual machine [GR83], the Java Virtual Machine [LY99] and the Common Language Runtime of the Microsoft .NET platform [ECM10] are virtual stack machines. The common arguments [SCEG08] in favor for stack-based codes are: i) they are smaller in size, due to the absence of explicit operands, ii) they tend to be more compressible, an important point in the context of mobile code and iii) they permit to write simpler interpreters. However, these claims about the light-weightness of stack-based codes are not so clear, since many other virtual machines are nowadays register-based (Dalvik VM [Bor08] and Parrot [Fag05]).

Optimization is hard Stack-based representations are not suited to code optimization, because of the mixing of instructions and expressions. Expression computations are hidden in the operand stack manipulation, which makes redundant computations hard to detect and the code hard to manipulate. Additionally, the previous remarks about unstructured CFG also holds on STACK. In practice, even industrial-stength compilers producing stack-based code (e.g. `javac`) perform very few optimizations. Optimizations are done on a register-based representations, either by the virtual machine's JIT compiler (see HotSpot Server [PVC01] or JikesRVM [ABC+02]) or an external optimization framework like Soot [VRCG+99].

STACK is mainly used by virtual machines. It is independent of both the source language and the underlying architecture running the virtual machine, but the light-weightness claim

$$
\begin{array}{rcl}
bexpr & ::= & const \mid var \\
expr & ::= & bexpr \mid bexpr\ op\ bexpr \\
op & ::= & + \mid > \\
stmt & ::= & \texttt{goto}\ \ell \mid var := bexpr \\
 & & \mid \texttt{if}\ bexpr\ \texttt{goto}\ \ell \\
 & & \mid \texttt{return}\ var \\
 & & \mid (var := \phi(var, \ldots, var))\ list \\
prog & ::= & \texttt{prog}\ name\ (var, \ldots, var)\{\ \ (\ell : stmt)\ list\ \ \}
\end{array}
$$

(a) Syntax

```
prog foo (n, t) {
   1: r0 := 0
   2: i0 := 0

   3: r2 := φ(r0,r1)
      i2 := φ(i0,i1)

   4: c0 := i2 > n
   5: if c0 goto 9
   6: r1 := r2+i2
   7: i1 := i2+t
   8: goto 3
   9: return r2
}
```

(b) Example program

Figure 2.4: SSA language

about stack-based bytecodes is subject to debate [SCEG08]. STACK is also hard to optimize and analyze, mainly because of the operand stack-manipulation for computing expressions.

2.2.3 Static Single Assignment: SSA

The Static Single Assignement form (SSA) is a popular IR used in many modern optimizing compilers, e.g. GCC or LLVM [LLV]. In SSA, each program variable is assigned only once in the program text. Introduced in the late 80's by Alpern et al. [AWZ88], this IR was designed to improve both code quality and algorithmic efficiency of optimizations. This IR encountered a great success, and the literature is huge. An on-going work is [SSAar] which aims at giving a comprehensive state-of-the-art and practice about SSA. In particular, SSA has numerous variants, and as been extended to many programming languages features and aspects (e.g. global memory or arrays). These are out of the scope of the present work. Here we just give an overview of this IR; SSA will be studied in Chapter 5.

Syntax The syntax of SSA is given in Figure 2.4, along with the SSA form of our example program. Its syntax is very close to the one of TAC: SSA is essentially TAC code that is constrained so that each variable is statically assigned only once. For straight-line code, it is enough to rename properly variables definitions and uses by introducing new versions of them (for instance, the initial TAC variable `r` in Figure 2.2b gives rise to three new versions `r0`, `r1` and `r2` in Figure 2.4b). But to handle junction points, where distinct versions of a variable can reach, SSA needs an extra instruction, the so-called ϕ-function `x :=` ϕ`(x1,...,xn)`, with the following informal semantics. The value of ϕ`(x1,...,xn)` depends on the flow of control. It evaluates to the value of its kth argument whenever the flow of control is reaching that instruction from its kth predecessor. In Figure 2.4b, consider the junction point 3, and suppose its first (resp. second) predecessor is 2 (resp. 8), before the loop has ever been executed, the value of ϕ`(r0,r1)` is the one of `r0`. When reaching point 3 with the `goto` statement at point 8 (i.e. at the end of every execution of the loop body), it equals the value `r1`.

Notice that not all variables need a ϕ-function (for instance, none is required for the instances of the TAC variable `c`). At a given join point, ϕ-functions are gathered into blocks, to be evaluated, in parallel, with respect to the same predecessor. Several ϕ-function placements

policies exist (maximal, minimal, pruned, or semi-pruned), as well as various algorithms for computing the corresponding sets of nodes (using either dominance-frontiers, merge sets, or maximal pruning), that trade-off their number with the cost of minimizing it [BCHS98].

Strong structural properties SSA comes with two strong structural properties, that are clearly identified in the literature. First is the single assignment property, which implies that each variable use can only refer to a single definition of that variable. In Figure 2.2b, two definitions of the variable `r` were reaching its use at point 5, while in Figure 2.4b, they are merged into the definition of `r2`. This property makes explicit the use-definition chains in the program text: the reaching definition of a variable use is just the definition point of that variable. Another interesting property of this naming convention is that, after converting a program into SSA, the live-ranges of the versions of a same initial variable do not overlap.

Second is a strictness property [Bri06]: in SSA, all variables are defined before being used. Strictness seems to be a reasonable constraint on the program code, but some languages such as C or C++ do not impose any such restriction. Combined with the single assignment property, the strictness of SSA can be exploited to compute efficiently the live-ranges of variables, as well as the interferences between variables (i.e. the intersection of their live-ranges) [BCH+02, BHG+08], information that is particularly useful for performing register allocation.

Data-flow analyses SSA is undoubtebly well-suited for data-flow analyses and optimizations. We already discussed the benefits of singleton use-definition chains, compared to multiple reaching definitions, as well as the algorithmic impact, in terms of efficiency, of strictness.

In addition, the single assignment property makes SSA a sparse representation of the program: on SSA, many data-flow analyses (e.g. constant detection) do not need to propagate data-flow facts throughout the program (i.e. to each program point); attaching the information at the variable definition points is enough, because it is propagated implicitely through the definition-use links. This is why SSA is often considered as bringing flow sensitivity for free.

Finally, SSA has been used as a basis for many powerful optimizations, such as Common Sub-expression Elimination based on Global Value Numbering [AWZ88], or Partial Redundancy Elimination [CCK+97]. These optimizations extensively use the so-called equational nature of SSA, that exploits the single assignment property and the lexical congruence or identity of expressions for detecting equality between variables and expressions at run-time. The unique definitions of program variables can be seen as a functional aspect of SSA (we discuss the link between SSA and CPS in the next paragraph), thanks to which, intuitively, persistent information can be associated to variable's definitions.

The SSA form was designed for the purpose of program optimizations. Its strong structural properties (single assignment and strictness) are exploited by SSA optimizations to gain precision and/or efficiency. In fact, the structural properties of SSA guarantee semantic invariants that are global to the program. Still, these invariants remain unclear in the literature, or are at least not formally stated.

2.2.4 Continuation-passing style: CPS

The use of CPS in compilers for functional languages can be traced back to the work of Sussman and Steele, for a Scheme compiler [SS75, Ste78]. Nowadays, many optimizing compilers are based on CPS (Orbit Scheme [AKK+86], SML/NJ [App92], MLton [MLt], SML.NET [Ken07])

and many of the properties of CPS are used outside of the context of functional language compilation – Costa is one of them (see Section 2.2.5). SSA and CPS are similar in many ways, we discuss this at the end of the section.

The key idea behing CPS is that *"in this continuation-passing programming style, a function always returns its result by sending it to another function"*[SS75], indicating what to do next, i.e a *continuation*.

$$
\begin{array}{rcl}
\mathit{bexpr} & ::= & \mathit{const} \mid \mathit{var} \\
\mathit{expr} & ::= & \mathit{bexpr} \mid \mathit{bexpr}\ \mathit{op}\ \mathit{bexpr} \\
\mathit{op} & ::= & + \mid \leq \\
\mathit{val} & ::= & \mathit{expr} \\
 & & \texttt{fun}\ k\ (\mathit{var}, \ldots, \mathit{var}) \rightarrow\ \mathit{term} \\
\mathit{fdecl} & ::= & \texttt{letv}\ f = \mathit{val}\ \texttt{in}\ \mathit{term} \\
 & & \texttt{letc}\ k(\mathit{var}, \ldots, \mathit{var}) = \mathit{term}\ \texttt{in}\ \mathit{term} \\
\mathit{term} & ::= & \mathit{fdecl} \\
 & & \texttt{let}\ \mathit{var} = \mathit{expr}\ \texttt{in}\ \mathit{term} \\
 & & f\ k\ x \mid k\ x \\
 & & \texttt{if}\ \mathit{expr}\ \texttt{then}\ k\ \ \texttt{else}\ \ k \\
\mathit{prog} & ::= & \mathit{term}
\end{array}
$$

(a) Syntax

```
letv foo = fun k (n,t) →
  letc test (k1,k2,i,n) =
    if (i≤n) then k1 else k2
  in
  letv loop = fun k (r,i,n,t) →
    let bodyk = body k in
    let k' = test (bodyk,k,i,n) in
    k' (r,i,n,t)
  in
  letv body = fun k (r,i,n,t) →
    let r = r+i in
    let i = i+t in
    loop k (r,i,n,t)
  in
  let r = 0 in
  let i = 0 in
  loop k (r,i,n,t)
```

(b) Example program

Figure 2.5: CPS language

Syntax The syntax for CPS is given Figure 2.5a. It is basically a lambda-calculus, with local name declarations (`let`), function declaration (`letv` and `letc`), function application, and basic operators[2]. For clarity, we distinguish traditional functions (declared with `letv`), whose generic name is f in Figure 2.5a, from continuations, declared using `letc` (with generic name k). Traditional functions take a continuation as an extra-argument (here, by convention the first one). In the example in Figure 2.5b, these correspond to `foo`, `body` and `loop`, respectively encoding the main initial program, the body of the while-loop, and the loop itself.

Optimizations In CPS, the data-flow information is explicit thanks to the local declaration names. The use of variables is similar to the use of virtual registers in TAC: operator and function arguments are atomic (i.e. constants or variables). Besides, CPS exposes the control-flow in a unified (interprocedural) manner, with functions and continuations calls. CPS is thus well-suited to code optimization, the reasons being essentially the same as for TAC.

Atomic function arguments additionally makes function inlining easier to implement: the substitution of actual for formal parameters cannot modify the termination behavior of the inlined program. It also avoids dealing with the preservation of side-effect order (assuming the language includes side-effects).

Machine code generation The machine code must use a limited number of physical registers, but here again, a CPS program can be easily transformed into an equivalent one where

[2]For the sake of clarity, we choose to treat basic operations as values, as is the case in [Ste78]

functions (i) have no free variables (closure conversion), (ii) are not nested (nested scope elimination) (iii) and in which no subexpressions uses more than n variables, where n is the number of physical registers (register spilling) [App92]. The CPS form can thus be used until late stages of the compilation chain.

The problem of administrative redexes The CPS generation suffers from the production of many administrative redexes, i.e. superfluous computations that were not present in the initial term. For efficiency concerns, the compiler must reduce their number [Ste78, App92]. CPS, as an IR for compilers, was criticized in that respect, because the compiler has to remove much of what the conversion to CPS introduces. In [FSDF93], Flanagan et al. argue that this problem can be overcome, observing that most of the CPS features used by compilers were already provided by the Administrative Normal Form (ANF). ANF is a *direct* programming style in which `let` assign names to all intermediate computations. Many compilers opted in favor of ANF, such as TIL [TMC+96]. The remaining issue of ANF was that continuations were implicit, which complicates tail-call elimination [FSDF04]. Some CPS conversions have been proposed that does not introduce administrative redexes [DN03, DL07]. All the previously cited compilers based on CPS get rid of these redexes during the CPS conversion.

CPS and SSA These two IRs are similar in many ways, and this has been studied extensively. Kelsey [Kel95] establish a correspondence between both, with a conversion algorithm (back and forth) that is guided by annotations of the lambdas. CPS satisfies a variant of the SSA property: each variable is bound only once. Besides, the strictness property of SSA can be mapped to the variable lexical scopes in CPS [Kel95]. The difference between CPS and SSA lies in that a CFG edge in SSA is a function call in CPS: the implicit assignment of the function arguments realized when calling the function in CPS will be somewhat materialized in SSA by ϕ-functions. This is observed by Appel in [App98b] who additionnaly explains that the optimal function nesting relates to the conversion to minimal SSA.

Analyses CPS provides a clear and uniform encoding of the CFG, thus simplifying data-flow analyses. Compared to SSA, CPS seems however to be harder to analyze: any intra-procedural analysis on SSA will have to be inter-proceduralized for CPS. This problem can be partly solved by distinguishing functions and continuations (as we have done here) and inlining CPS functions [Kel95]: interprocedural analyses are easier to perform on CPS. This has to be compared to the analysis of loops in a program in SSA, where global analyses (as opposed to basic-block analyses), although far from being trivial to justify formally, can leverage the properties of SSA to reason across basic blocks.

Like TAC, CPS expressions and control structures have been lowered compared to While. However, compared to TAC, the additional structural constraints of CPS give rise to several global properties that can be used to balance that lowering, allowing for more precise code optimizations and analyses. This is what makes both SSA and CPS widely used in modern compilers. Moreover, the optimization potential of these IRs is compatible with later lowering phases, such as register allocation.

We switch now to the presentation of complex IRs: they combine several basic IRs, and were designed for a special verification purpose. These are used in verification frameworks and as such, include a large amount of features. We present only the IR fragments corresponding to While – in particular, we will not detail all their object-oriented features.

2.2.5 Cost analysis: Costa

The Costa [AAG+12] system is a framework for designing static cost analyses of sequential Java programs. Cost analysis bounds the amount of resource requirements (e.g. time or space complexity) for a given program to run. Costa targets mobile code or commercial software, it thus does not take Java as input, but Java bytecode (JBC) programs, an extension of STACK to Java. Another reason for analyzing JBC is that it is what the virtual machine executes: the cost can be estimated more faithfully.

Cost analysis Costa automatically extracts, by static analysis, a cost relation system (CRS) from the input JBC program. A CRS is basically a recursive linear equation system, capturing the execution cost as a function from the program input arguments. The CRS is then passed to an external solver that computes the upper bound cost of the program. A full description of CRS, out-of-scope of the present work, can be found in [AAGP09]. The analysis does not infer the CRS directly from the JBC program, but relies on its RBR representation, and this is what we present now, focusing on the STACK fragment of JBC[3].

Tackling STACK analysis difficulties RBR expresses STACK programs in a form that is close to a linear equation system. RBR is a *Rule-Based Representation*: a program is a set of rules, each of which will be later abstracted by a linear equation.

The two main difficulties discussed previously about STACK analysis are i) its unstructured control-flow and ii) the extensive use of the operand stack. RBR will rely on a variant of CPS to tackle these two challenges. The program CFG is encoded into function calls (and recursion when the CFG contains cycles), and expression computations will be reconstructed with three-address instructions: each STACK instruction is translated into a sequence of basic operations manipulating local variables. Additionally, the STACK instructions irrelevant to the cost analysis will be abstracted so that they will analyzed as the linear constraint *true*.

Syntax Figure 2.6 gives the syntax of RBR and the (slightly optimized) RBR form of the STACK program given in Figure 2.3b. A rule $h(\vec{args}, res) \leftarrow g, b, \ldots b$ is made of a head $h(\vec{args}, res)$ and a body $b, \ldots b$, guarded by a condition g (the *true* condition is kept implicit). By convention, the last parameter of the function h is the result register. Each basic-block of the STACK CFG is coded by such a rule: to the block at point l corresponds the function $block_l$, whose definition is given by rules with head $block_l(\vec{args}, res)$ (the last argument res is the return variable). The actual STACK code inside the block is encoded by the body $b_1, \ldots, b_n$ of the rule, where each b_i is either a three-address assignment, a cancelled STACK instruction ($\texttt{nop}(instr)$, where $instr$ is any STACK instruction) when it is irrelevant to the cost analysis, or a call to a continuation, i.e. to the next block to execute. When the control flow branches, the continuation depends on the evaluation of a boolean condition guarding the continuation body. In Figure 2.6b, the branching at the end of block 5 is translated into two rules for foo_5^c (the continuation of foo_5).

The RBR representation is what we called a complex IR: it mixes several basic IRs, namely a variant of TAC, and a form of CPS. Both of these techniques will help the cost analysis building the corresponding CRS: (i) each basic RBR statement is translated into a linear cost constraint (ii) a cost is inferred for each of the rules, as a function of parameters.

[3] RBR handles the full JBC. See [AAG+07] for a full description.

$$
\begin{array}{rcl}
bexpr & := & var \mid const \\
expr & := & bexpr \mid bexpr\ op\ bexpr \\
op & ::= & + \mid\ > \mid\ \leq \ldots \\
b & ::= & x := expr \\
 & & \mid \texttt{nop}(instr) \\
 & & \mid id(\vec{var}, var) \\
g & ::= & \texttt{true} \mid bexpr\ op\ bexpr \\
rule & ::= & id(\vec{var}, var) \leftarrow g, b, \ldots b \\
prog & ::= & rule \ldots rule
\end{array}
$$

(a) Syntax

$$
\begin{array}{lcl}
foo(n,t,v) & \leftarrow & foo_1(n,t,v) \\
foo_1(n,t,v) & \leftarrow & r := 0, i := 0, \\
 & & foo_5(n,t,r,i,v) \\
foo_5(n,t,r,i,v) & \leftarrow & \textbf{nop}(\textbf{ifg } 17), \\
 & & foo_5^c(n,t,r,i,v) \\
foo_5^c(n,t,r,i,v) & \leftarrow & i > n, foo_{17}(r,v) \\
foo_5^c(n,t,r,i,v) & \leftarrow & i \leq n, foo_8(n,t,r,i,v) \\
foo_{17}(r,v) & \leftarrow & v := r \\
foo_8(n,t,r,i,v) & \leftarrow & r := r+i, i := i+t, \\
 & & \textbf{nop}(\textbf{goto } 5), \\
 & & foo_5(n,t,r,i,v).
\end{array}
$$

(b) Example program

Figure 2.6: The RBR language

2.2.6 Program verification: Boogie

Boogie [BCD+06] is a fully automatic deductive program verifier for object-oriented programs developed at Microsoft Research. It has been used as a common back-end for several verifiers, such as Spec# [LM10] or Dafny [Lei10].

Program verification Boogie accepts programs written in the Boogie language [DL05], an intermediate verification language specially designed for expressing proof obligations. The program code is annotated with its specification. Boogie optionally infers some invariants, and then generates, using a weakest preconditions calculus [Dij76], some verification conditions (VC), i.e. first-order (and arithmetic) logical formulas whose satisfiability implies the correctness of the program with regard to its specification[4].

Avoiding the exponential blow-up in VCs Barnett and Leino [BL05, Lei05], inspired by the work of Flanagan and Saxe [FS01], have identified a program normalization process on top of the Boogie syntax which makes the VC generation algorithm much more efficient, in the sense that it avoids redundancy in VCs: resulting VCs are both compact and more easily verified by theorem provers. We first focus on the Boogie IR, the normalization will follow.

Syntax The syntax of Boogie is given in Figure 2.7a, along with the Boogie representation of our program example (Figure 2.7b). Boogie provides rich expressions, of arbitrary height. Boogie is unstructured, but CFGs are made reducible. A program is a set of basic blocks, identified by their unique label and each basic block has a (possibly empty) set of successors, to which the control-flow branches explicitely with non-deterministic `goto`s. A block is a sequence of statements, i.e. variable assignment (to a given value) and havoc (the variable is assigned any possible value), unconditional jumps. Additionaly, assertions (`assert`) and assumptions (`assume`) make it possible to express the specification of the program: a program is said to be correct if none of its traces ends erroneously (i.e. no assertion fails).

[4]For keeping this presentation simple, we do not add a specification to the program example and keep the invariants opaque

$$
\begin{array}{rcl}
expr & ::= & const \mid var \\
 & & \mid expr\ op\ expr \\
op & ::= & + \mid \leq \mid > \\
stmt & ::= & \texttt{skip} \\
 & & \mid var := expr \\
 & & \mid \texttt{havoc}\ var \\
 & & \mid \texttt{assert}\ expr \\
 & & \mid \texttt{assume}\ expr \\
 & & \mid stmt; stmt \\
 & & \mid \texttt{goto}\ \ell, \ldots, \ell \\
block & ::= & \ell\ :\ stmt \\
prog & ::= & \texttt{prog}\ name\ (\vec{var}) \\
 & & \quad \texttt{return}(var) \\
 & & \quad \{\ block^{+}\}
\end{array}
$$

(a) Syntax

```
prog foo(n, t) return(r){
  entry: r:= 0;
         i:= 0;
         goto Loop;

  Loop:  goto LoopT,LoopF;

  LoopT: assume i ≤ n;
         r:= r+i;
         i:= i+t;
         goto Loop;

  LoopF: assume i > n;
         ret:= r;
         goto End;

  End:   r:= ret;
}
```

(b) Example program

```
prog foo(n, t) return(r){
  entry:assume r = 0;
        assume i = 0;
        assert loopI(r,i);
        goto Loop;

  Loop: assume loopI(r0,i0);
        goto LoopT,LoopF;

  LoopT:assume i0 ≤ n;
        assume r1 = r0+i0;
        assume i1 = i0+t;
        assume ret = r1 ;
        assert loopI(r1,i1);
        goto ;

  LoopF:assume i0 > n;
        assume ret = r0 ;
        goto End ;

  End:  assume r2 = ret;
}
```

(c) Example program (normalized)

Figure 2.7: The Boogie language

Normalizing Boogie programs The normalization process makes the programs satisfy structural and semantic constraints that avoid expression and/or formula duplication during the VC generation. The normalized program is given in Figure 2.7c. The transformation follows three steps, that we explain in more detail below. First, all loops are removed. Then, the program is put in Dynamic Single Assignment (DSA) form [Fea91]. Finally, it is made passive (programs no longer perform any assignment).

The process starts with eliminating loops in the program by removing back-edges, resulting in an acyclic CFG. In order to ensure that this CFG is correct only if the initial one is, some assumptions and assertions of loop invariants are added, so that the loop body represents any arbitrary loop iteration (in Figure 2.7c, this corresponds to the invariant `loopI`).

The program being loop-free, it can be converted in DSA form. DSA ensures that, along each path of the CFG, each variable is defined only once. This is what makes variables interpretable in their mathematical sense [Fea91] — the variable value is unknown, but constant. At junction points, the need for ϕ-functions (which would select the right version of variables) is replaced by a suitable assignement at all predecessors-points, similar to what would have produced the sequence SSA conversion-SSA destruction. In Figure 2.7c, the variable `ret`, used in block `End` is assigned at the end of the two blocks `LoopF` and `LoopT`.

Finally programs are converted into a passive form, replacing each assignment $x := expr$ with a statement `assume` $(x = expr)$. Given a postcondition Q, the weakest precondition of $x := expr$ is $Q[expr/x]$ while the assumption translates into an implication $(x = expr \Rightarrow Q)$. Passivity avoids the need for substituting the variable x with expression e, that could appear many times in Q (the worst case being reached with the statement $x_1 := x_0 + x_0\ ;\ x_2 := x_1 + x_1\ ;\ \cdots\ ;\ x_n := x_{n-1} + x_{n-1}$ when Q contains x_0). This final step is what makes the VC generation not suffer from an exponential blow up in the size of the formula with respect to the size of the initial statement [FS01].

Modularity If While would have included function calls, they would have been encoded with pre-condition assertion and post-condition assumptions: the VCs generation algorithm works intra-procedurally. When looking closer at loops handling in normalized Boogie, invariant assertions and assumptions are similar. This allows for generating VCs in a more modular way, namely basic-block wise.

Object-oriented features The full Boogie language also handles object-oriented features, modelling the heap as an array indexed by references and field names, and axiomatising all OO-features such as classes or inheritances. Verifications of such programs necessitate adding special formulas for the invariants the objects satisfy in the program, identifying what part of the heap is not modified. This is far from a trivial exercise and out of the scope of this chapter, the so-called Boogie methodology is described in [BCD+06].

The Boogie representation, and in particular its assertion and assumption statements, makes it possible to annotate the program with its specification. In addition, those statements are used to normalize the program so that compact VCs can be generated in a modular way. It is also worth noting that the Boogie syntax is not restricted to express normalized programs only: it still allows performing optimizations such that the ones working on TAC, with the additional benefit of rich expressions.

2.3 Discussion

2.3.1 The semantic impact of syntax and structure

Reduced instruction set Compared to a source language, an IR provides a representative but restricted set of instructions and constructs. For instance, Java high-level constructs such as `try catch finally` will be expanded and Java generics will be instantiated by the compiler, so that the JBC has fewer constructs than Java. It allows for programs to be normalized and source-language independent, at the right level of abstraction. A reduced instruction set is the first step towards an IR with a simple, well-defined semantics. Additionally, this allows for analyses and optimizations considering a restricted number of cases, as the syntax does not contain too many redundancies.

Explicit semantic facts Syntax makes explicit some semantic aspects of an IR. As discussed for STACK, making expressions part of the IR syntax, making explicit the operands of operators makes data-flow analyses more easily defined. They do not need to do additional work for reaching the same precision level as a source-language analysis. Three-address expressions can help, but arbitrary complex expressions can sometimes be preferred (see our discussion about TAC).

Implicit run-time checks and side-effect expressions are two other good examples. Exceptions and side-effects should be isolated into dedicated instructions. This can be respectively done in Java by e.g. demanding to each object creation (`new C()`) to be immediately stored in auxiliary variables (it thus becomes an instruction), and to add dedicated run-time checks to the language, rather than allowing instructions to raise exceptions.

This simplifies the semantics, and analyses become easier to define and to prove. Code transformations also benefit from the fact that each instruction has exactly one effect.

Control structure The representation of control flow is also an important design choice for an IR, as discussed previously about static analyses on TAC. In structured languages, the iteration strategy is somewhat explicit: data-flow equations can be solved based on the language syntax. When control structures disappear from the language, iteration strategies must be recovered by analysing the program CFG. CPS allows to recover some of the control flow, by distinguishing continuation from other function calls. In fact, Costa uses CPS-like constructs to rebuild the dependence relations between basic blocks.

At another extreme, some IRs remove almost all branching control-information from the program. This is the example of Boogie where branchings are non-deterministic and programs are normalized to be loop-free.

Structural constraints for semantic invariants Beyond syntax and control information, some structural constraints of an IR can give rise to semantic meta-properties of programs. For example, if the IR language includes explicit run-time checks, the IR construction can ensure that enough of such instructions are inserted, so that expression evaluation always succeeds.

SSA, CPS satisfy both a static single assignement property and a strictness property. This implies the global semantic property that allows interpreting a program as a set of equations (one for each variable or local name definition). Further, these structural properties can be exploited by SSA. While being register-based with basic expressions only, the invariants of SSA will avoid the need for static analyses to reconstructing full side-effect free expressions.

We thus observe that the syntax and structure of an IR have a non-negligible impact on its semantics. Further, syntactic and structural properties are used to give rise to semantic invariants, and some of them are non-trivial. From a formal point of view however, these invariants are often diffuse. Could we isolate those syntactic and structural properties, and derive some semantic theorems thereof?

Another remark must be done about syntactic and structural properties. We argued that they are part of the IR. Some of them are present in the IR only: they are established during the very generation of the IR. In this work, we thus not only aim at formalizing the properties of IRs, we also take care of proving that IR generation algorithms ensure those invariants.

2.3.2 A perfect IR?

The perfect IR does not exist Since the early work of Steel [Ste61] with UNCOL (*UNiversal Computer Oriented Language*), an attempt for solving the compiler-writing problem, and as pointed out by Macrakis in [Mac92], *"a universal intermediate language has been a dream for many years"*. An IR is originally designed with a given purpose in mind, leaving some other concerns on the side: as a striking example, STACK was designed according to a light-weightness concern, ignoring code optimization. Some purposes can even be contradictory, think e.g. of code lowering versus optimizations.

A good IR is a good trade-off More optimistically, Muchnick [Muc97] states that "intermediate language design is largely an art, not a science". Choosing a compiler IR should be done according to both the source and target languages, and with the kind and degree of optimizations (in terms of efficiency and difficulty) in mind.

Sometimes, the level at which the analysis or optimization must be performed puts some constraints on the IR. A good example is the JBC Verifier, that must be run at class loading. The stack-based nature of JBC was not chosen for perfoming this analysis. In this case, the analysis had to adapt to the code to verify.

Still, an IR should not be too ad-hoc, compared to the savings that code reuse would allow. An IR should not be designed for a single optimization: in this case, the perfect IR does exists. It does not correspond to any program code, but rather to the result of the underlying analysis itself. Costa and Boogie avoids this pitfall by keeping the IR general enough: many instructions are irrelevant to a cost analysis but Costa does not remove any instruction — it tags them as neutral — and Boogie keeps the Boogie syntax separated from the normalization phase. Thus, if the perfect IR does not exist, one can combine IRs, either in sequence – as is done along the whole compiler chain – or by mixing and trading-off their properties and advantages.

2.4 Conclusions

We are now able to refine Definition 2.1.

Definition 2.2 (Intermediate representation)**.** *An intermediate representation is an analysis-oriented language adapted to a verification or transformation purpose, thanks to its syntactical, structural and semantic properties.*

Definition 2.2 can be summarized as follows. An IR is defined by a representation of the program code, and some structural and semantic properties. Converting a program into an IR can be seen as a preprocessing of the analysis. The main challenge being thus to trade-off what properties should be embedded in the IR, and what properties should be computed by the analysis.

In this work, we do not aim at designing new IRs. Our goal is rather to study carefully, from a semantic point of view, some (variants of) of the leading IRs we were presenting here.

As discussed about STACK, the JBC language is not an IR that is analysis or verification-friendly, due to the extensive use of the operand stack. Modern JIT compilers and JBC analyzer rather work on an IR of the JBC similar to TAC. Chapter 4 proposes a generation algorithm for such an IR. It goes a bit further than TAC, by e.g. rebuilding side-effect free expressions trees or adding explicit run-time checks.

As our discussion also suggests, it is also important to give IRs a clear formal semantics, and to identify its global, high-level, meta-properties. In [FS01], Flanagan reports on his work on CPS: *"A large majority of compiler writers reported that our paper confirmed their understanding in a precise and formal way"*. This is what our work aims at with SSA (see Chapter 5): a formal semantics that captures the initial intuitive informal statements of seminal papers, as well as the formal statement and proof of two global properties, namely strictness and equational-form. Once such properties are clearly identified, formalized and proved, they can be used to justify formally SSA-based optimizations and analyses.

When dealing with shared memory accesses, even the semantics of a TAC language can be surprisingly complex. Our work in Chapter 6 defines a formal semantics of multi-threaded Java that is intuitive, in the sense that its definition is characterized by the reorderings compilers and hardware are allowed to perform on program instructions. Here, the valid reorderings hence become part of the definition of the semantics, rather than being meta-properties thereof.

Additionnaly, this work can be seen as the semantic counter part of a multi-level IR [Muc97] (an IR that would fit several layers of the compiler chain): we benefit from the approach taken in [ŠVZN+11], in which the common part of the semantics shared by all the IRs of the Java compiler, i.e. the memory model, is factorized into what we call the BMM machine.

The formalization of the semantics of IRs as well as their properties is undoubtely important. But in the context of verified software, all of these cannot be taken for granted: IR generation algorithms must also be formally studied. What should they ensure, and how to prove such properties? This is the subject of the next chapter.

Chapter 3

Proving transformations correct

In Chapter 2, we reviewed part of the IR landscape, staying mostly at the level of their language and describing informally their semantics and properties. What about the associated program transformations, i.e. the IR generations? First, the languages they have to handle, e.g. Java or C, are large and their semantics can be subtle. Second, they must establish the structural invariants of the IR to generate (e.g. the SSA property, or make the CFG reducible). Finally, because they are part of a larger software (i.e. an analyzer or compiler), such algorithms should be efficient. For these reasons, IR generation algorithms can be complex. Hence, in the context of verified software, not only should the semantics of IRs be formalized, but also their generation algorithm.

In this chapter, we study the relation between an initial program and its IR: how should they relate semantically, or what does it mean for an IR program to be a correct representation of the initial program? We present some of the formal concepts and tools for defining and proving such facts, that we will use in the rest of the document. In Section 3.1, we review the notion of *observational semantics* of programs, defining *what* should be preserved, and some associated preservation criteria describing *how* it should be preserved. The proof technique we use for establishing the preservation of observation between two programs, namely *simulations* is a well established semantic tool that we present in Section 3.2. Semantic correctness must then be lifted to *program transformations*; we present in Section 3.3 the existing approaches for proving them correct.

The correctness of program transformations has been studied since the beginning of compilation with the early works of McCarthy and Painter [MP67] and Morris [Mor73]. It was also considered in the field of program proof, where program transformations are used to produce programs that are easier to prove [Mil71, AL91]. It was the work of Milner [Mil71] that introduced the notion of simulation in an algebraic setting. All of these work have been of great influence in the field of program verification with respect to temporal logics [Sch99, Sch02] or timed trace properties [LV92], as well as program transformation correctness [LJVWF04, GZ99] (see [Dav03] for a comprehensive bibliography), including the work of Leroy et al. [Ler09] on the verified compilation of C. This chapter builds on those work, in particular on Leroy's considerations in [Ler09] about compilers correctness.

3.1 Semantics preservation

In Chapter 2, we gave several representations of the While program in Figure 2.1b. Although those programs are syntactically different, it is easy to convince oneself that they represent the same algorithm, and this is what makes each of them a correct representation of the initial program. This section presents how this intuition can be formalized into what is called *semantics preservation*.

3.1.1 Formal semantics

Formal semantics is a mathematical answer to the question *"What does this program do?"*. It describes formally the meaning of a program, its behavior. Formal semantics comes in different flavors, depending on the language, on the information relevant to describe, or the context of use.

In our work, we give programs an operational semantics, for several reasons. First, this formalism is easy to turn into an interpreter of the language that can be used to increase one's confidence in the formalization by testing it against existing implementations. The formal semantics of a language like C or Java is a large mathematical object, and relying on an interpreter is capital to establish its validity [BL09, ER12]. Second, operational semantics come with the proof technique of simulations (Section 3.2), a tool that is well established and particularly easy to use in proof-assistants [Ler09, ŠVZN+11]. Additionally, static analyses are often defined in an operational way (either with data-flow equations or with transfer functions in a abstract interpretation setting). Finally, safety properties are formulated in terms of sets of reachable states, a concept easily defined on top of an operational semantics.

The operational semantics that we consider are presented as transition systems, intended to describe the step-by-step program execution, akin to a interpreter.

Definition 3.1 (Transition System)**.** *A transition system is a tuple* $(\Sigma, I, F, \rightarrow)$ *where* Σ *is a set of states,* $I \subseteq \Sigma$ *and* $F \subseteq \Sigma$ *are the sets of initial and final states respectively, and* $\rightarrow \subseteq \Sigma \times \Sigma$ *is the transition relation.*

The semantics of a program is then defined as the set of execution traces of the associated transition system.

Definition 3.2 (Operational semantics)**.** *Let* $S = (\Sigma, I, F, \rightarrow)$ *be a transition system. The set of execution traces of* S *is defined*

$$\begin{array}{rcll} \mathit{Traces}(S) & = & \{\sigma_0\sigma_1\sigma_2\ldots\sigma_n \mid \sigma_0 \in I, \forall i, 0 \leq i < n \Rightarrow \sigma_i \rightarrow \sigma_{i+1}\} & \textit{finite traces} \\ & & \cup\ \{\sigma_0\sigma_1\sigma_2\cdots \mid \sigma_0 \in I, \forall i \in \mathbb{N}, \sigma_i \rightarrow \sigma_{i+1}\} & \textit{infinite traces} \end{array}$$

Note that the set of finite traces also includes execution traces that do not halt in a final state. These corresponds to executions that get stuck. We illustrate Definitions 3.1 and 3.2 on the While language defined in Chapter 2 (Figure 2.1). Figure 3.1 defines the small-step operational semantics of While, as given in any semantics textbook, e.g. [Win93]. We assume a While program **prog** $p(\overrightarrow{\mathit{args}})\{s\}$ that is called with integers arguments $\overrightarrow{\mathit{val}}$. Semantic states consist of a statement to execute and a local environment (a mapping from variables to integers values). In the initial state, no instruction has been executed yet and variables have the default value 0 given by ρ_0, except for program arguments *args*, assigned to the input value arguments. A final state is a state in which no instruction remains to execute (denoted by $\bullet$), and the return value is in $\mathbb{Z}$. The transition relation describes the small-step execution of each statement.

Given an initial program and its IR, we want to compare or relate their semantics. Suppose the two programs are written in two distinct languages, e.g While and TAC. It is easy to define an operational semantics for TAC programs as we did for While, but semantics states differ, because e.g. of the introduction of auxiliary variables in TAC. So how to relate their trace

$$
\begin{array}{rcl@{\qquad}rcl}
val & = & \mathbb{Z} & \downarrow & \subseteq & expr \times env \times val \\
stmt_\bullet & = & stmt \cup \{\bullet\} & \rightarrow & \subseteq & \Sigma \times \Sigma \\
env & = & var \rightarrow val & \Sigma_I & = & \{(s, \rho_0[\overrightarrow{args} \leftarrow \overrightarrow{val}])\} \\
\Sigma & = & stmt_\bullet \times (env \cup val) & \Sigma_F & \subseteq & \{\bullet\} \times val
\end{array}
$$

(a) Domains

$$
\frac{c \in \mathbb{Z}}{(c, \rho) \downarrow c} \qquad \frac{x \in var}{(x, \rho) \downarrow \rho(x)} \qquad \frac{(e_1, \rho) \downarrow v_1 \qquad (e_2, \rho) \downarrow v_2}{(e_1 \ op \ e_2, \rho) \downarrow v_1 \ op \ v_2}
$$

$$
\frac{(e, \rho) \downarrow v}{(x := e, \rho) \rightarrow (\bullet, \rho[x \leftarrow v])} \qquad \frac{(e, \rho) \downarrow v}{(\textbf{return } e, \rho) \rightarrow (\bullet, v)} \qquad \frac{(e, \rho) \downarrow v \qquad \begin{array}{l} v \neq 0 \Rightarrow s' = s_1 \\ v = 0 \Rightarrow s' = s_2 \end{array}}{(\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2, \rho) \rightarrow (s', \rho)}
$$

$$
\frac{}{(\textbf{skip}, \rho) \rightarrow (\bullet, \rho)} \qquad \frac{(s_1, \rho) \rightarrow (s_1', \rho') \quad s_1' \neq \bullet}{(s_1; s_2, \rho) \rightarrow (s_1'; s_2, \rho')} \qquad \frac{(s_1, \rho) \rightarrow (\bullet, \rho')}{(s_1; s_2, \rho) \rightarrow (s_2, \rho')} \qquad \frac{(s_1, \rho) \rightarrow (\bullet, v)}{(s_1; s_2, \rho) \rightarrow (\bullet, v)}
$$

$$
\frac{(e, \rho) \downarrow 0}{(\textbf{while } e \ \{s\}, \rho) \rightarrow (\bullet, \rho)} \qquad \frac{(e, \rho) \downarrow v \quad v \neq 0}{(\textbf{while } e \ \{s\}, \rho) \rightarrow (s; \textbf{while } e \ \{s\}, \rho)}
$$

(b) Transition relation

Figure 3.1: Operational semantics of While

semantics? The same problem arises even when both programs are written in the same language (when e.g. the IR is a normalized version of the initial program). For instance, just reordering the initialization statements of local variable i and r in the program example in Figure 2.1b make their execution traces differ, whereas this should make no significant difference for what the user observes.

The problem here is the following: the semantics is defined for programs, but it is in a sense too precise to be used for comparing their meaning. This is where the notion of *observational semantics* comes into play.

3.1.2 Observational semantics

The idea of observational semantics is to define the observable behaviors of programs on top of their semantics. Distinguishing between the semantics and its observation emphasizes that the observation is somewhat biased, compared to program semantics: the semantics is faithful to the program execution while its observation is only partial. The observation that is made at each computation step is put into a label on the transition. When nothing is to be observed, a special label τ is used to indicate a silent step. We also assume a special label $\times$ for observing the executions that get stuck. The observational semantics is then defined as the set of traces of observable (i.e. non-silent) actions.

Definition 3.3 (Labelled transition system)**.** *A labelled transition system (LTS) is a tuple* $(\Sigma, I, F, L_\tau^\times, \rightarrow)$ *where* Σ *is a set of states,* $I \subseteq \Sigma$ *and* $F \subseteq \Sigma$ *are the sets of initial states and final states respectively.* $L_\tau^\times = L \cup \{\tau, \times\}$ *is a set of labels and* $\rightarrow \subseteq \Sigma \times L_\tau \times \Sigma$ *is a labelled transition relation.*

Definition 3.4 (Observational semantics)**.** *Let* $S = (\Sigma, I, F, L_\tau^\times, \rightarrow)$ *be an LTS. The set of*

traces of S is defined

$$\begin{array}{rll} \mathit{Traces}(S) = & \{\ell_0\ell_1\ldots\ell_{n\text{-}1} \mid \exists\sigma_0,\ldots\sigma_n,\ \sigma_0 \in I \wedge \forall i, 0 \leq i < n \Rightarrow \sigma_i \xrightarrow{\ell_i} \sigma_{i+1}\} & \textit{finite traces} \\ \cup & \{\ell_0\ell_1\cdots \mid \exists\sigma_0,\ldots,\ \sigma_0 \in I, \forall i \in \mathbb{N}, \sigma_i \xrightarrow{\ell_i} \sigma_{i+1}\} & \textit{infinite traces} \\ \cup & \left\{\ell_0\ell_1\ldots\ell_{n\text{-}1}\times \mid \begin{array}{ll} \exists\sigma_0,\ldots\sigma_n, & \sigma_0 \in I, \forall i, 0 \leq i < n \Rightarrow \sigma_i \xrightarrow{\ell_i} \sigma_{i+1} \\ & \wedge\sigma_n \notin F \wedge \forall\sigma\forall\ell, \sigma_n \not\xrightarrow{\ell} \sigma \end{array}\right\} & \textit{stuck traces} \end{array}$$

The set of observable behaviors of S is defined as the set

$$\mathit{Obs}(S) = \{t \downarrow_{L^\times} \mid t \in \mathit{Traces}(S)\}$$

It should be the case that the semantics of the program can be recovered from its observational semantics: forgetting about the transition labels should lead back to the initial transition system. In other words, the observational semantics is only an instrumented semantics of the program, and the transition labels never influence computations.

Depending on the application case, labels can include a more or less coarse observation of the execution. For instance, the labels could be the whole execution states, when the execution is completely observed: every transition $s \to s'$ in the transition system would lead to the observational step $s \xrightarrow{s} s'$. As discussed previously, this observation is too fine-grained, and the observation should be made partial. We discuss now the cases of analyzers and compilers.

Observation for analyses Suppose that an analyzer of programs of a language $\mathcal{L}_1$ relies on an IR $\mathcal{L}_2$ to perform the analysis. In this case, the initial $\mathcal{L}_1$ program and its IR must behave similarly, with regard to, *at least*, the property of interest. Usually, a safety property is expressed as a set of observational behaviours (i.e. traces of labels), and the system S is safe when $\mathit{Obs}(S) \subseteq \mathit{Safe}$.

Suppose that the only guarantee we want to have about the program execution is that it does not get stuck. With regard to Definition 3.4, this corresponds to a trace that ends with label $\times$. Here, no observation but $\times$ is absolutely required for expressing the safety property, and all computation steps could be silent. Consider now that the analysis must determine whether a given variable x is constant along the execution. Observing the local environment value for that variable would be enough. All transitions in the system would be silent, except the assignments of that variable, that would be labelled with the value of its right-hand side.

Hence, the observation could be specialized to the analysis, by observing the minimal information required for expressing the soundness of this analysis. But this is not satisfactory, as a new observation would have to be defined for every analysis to be performed. If the observation states exactly what is preserved between both programs, and nothing less, several analyses can reuse the same preservation result. A second issue is that, given an IR, it is not always possible to know *in advance* what analysis will rely on it, and therefore what observation needs to be preserved. We come back to this point in Section 3.2.2.

Observation for compilers As for compilers, the situation differs slightly, and the choice of observation is less free: one wants the source and compiled programs to behave the same *from a functional point of view*. The idea is hence to observe external function calls, which describe precisely how the program interacts with its environment. This is the approach taken

$$\frac{(e,\rho)\downarrow v \quad \begin{array}{c} x \in \mathit{Ext} \Rightarrow \ell = (x,v) \\ x \notin \mathit{Ext} \Rightarrow \ell = \tau \end{array}}{(x := e, \rho) \xrightarrow{\ell} (\bullet, \rho[x \leftarrow v])} \qquad \frac{(e,\rho)\downarrow v}{(\textbf{return}\ e, \rho) \xrightarrow{\tau} (\bullet, v)} \qquad \frac{(e,\rho)\downarrow v \quad \begin{array}{c} v \neq 0 \Rightarrow s' = s_1 \\ v = 0 \Rightarrow s' = s_2 \end{array}}{(\textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2, \rho) \xrightarrow{\tau} (s', \rho)}$$

$$\frac{}{(\textbf{skip}, \rho) \xrightarrow{\tau} (\bullet, \rho)} \qquad \frac{(s_1,\rho) \xrightarrow{\ell} (s_1',\rho') \quad s_1' \neq \bullet}{(s_1; s_2, \rho) \xrightarrow{\ell} (s_1'; s_2, \rho')} \qquad \frac{(s_1,\rho) \xrightarrow{\ell} (\bullet,\rho')}{(s_1; s_2, \rho) \xrightarrow{\ell} (s_2, \rho')} \qquad \frac{(s_1,\rho) \xrightarrow{\ell} (\bullet,v)}{(s_1; s_2, \rho) \xrightarrow{\ell} (\bullet, v)}$$

$$\frac{(e,\rho)\downarrow 0}{(\textbf{while}\ e\ \{s\}, \rho) \xrightarrow{\tau} (\bullet, \rho)} \qquad \frac{(e,\rho)\downarrow v \quad v \neq 0}{(\textbf{while}\ e\ \{s\}, \rho) \xrightarrow{\tau} (s; \textbf{while}\ e\ \{s\}, \rho)}$$

Figure 3.2: Observational semantics for While

in CompCert [Ler09] and its derivative [ŠVZN+11].[1] In Chapter 6, we also follow a similar approach, and will discuss in more detail the influence that compilers can have on the definition of a semantics in the precise context of shared-memory concurrency. A similar observation can be defined on While (as well as on the target language) by observing the updates of a subset $\mathit{Ext} \subseteq \mathit{var}$ of local variables, representing an output channel. This is given in Figure 3.2, where labels are in the set $L_\tau = (\mathit{Ext} \times \mathbb{Z}) \cup \{\tau\}$. The rule of assignment distinguishes two cases, according to whether the defined variable is external or not. All other rules are simply silent, or propagate the emitted label (in the execution rule for the sequence).

The observational semantics defines *what* should be preserved between two programs. The next section discusses *how* this information should be preserved.

3.1.3 Choosing the right preservation criteria

Let $P_1 \in \mathcal{L}_1$ and $P_2 \in \mathcal{L}_2$. In this section, we consider the relation that should be established between $\mathit{Beh}(P_1)$ and $\mathit{Beh}(P_2)$, where $\mathit{Beh}(P_i) = \mathit{Obs}(S_i)$ and S_i is the labelled transition system associated to P_i. The strongest preservation criteria is to demand that both programs have exactly the same observational semantics, i.e.

$$\mathit{Beh}(P_1) = \mathit{Beh}(P_2)$$

Notice that the systems' labels L_1 and L_2 should at least have a non-empty intersection. In Section 3.2.2, we discuss a situation where the equality of L_1 and L_2 does not hold. The above equality requirement is often too much demanding. As discussed in the next two paragraphs, a simple inclusion might be either sufficient or the most we can prove.

Preservation for analyses Suppose program P_1 is analyzed with regard to the safety property *Safe*, and that the analyzer takes as input the IR P_2 of the initial program. What we are interested in at the end, is to know whether P_1 is safe. Thus, the analysis verdict saying whether the program is safe or might be unsafe, should be transfered from P_2 back to P_1. Here, having

$$\mathit{Beh}(P_1) \subseteq \mathit{Beh}(P_2)$$

[1] In CompCert, global variables that are declared as volatile are also observed. We give a more precise definition of CompCert's observations in Chapter 5.

already allows recovering P_1's safety from the safety of P_2, with the following reasoning: $Beh(P_1) \subseteq Beh(P_2) \subseteq Safe$. We cannot deduce anything about P_1 when P_2 might be unsafe.

This inclusion can result in a loss of precision, whenever too many observable behaviors are added in P_2 by the transformation. A typical example is when conditionals are changed for non-deterministic branches in P_2. A sound analysis must consider all branches, while it could infer the infeasibility of some paths in P_1. The syntactic over-approximation performed by the IR is thus translated into the analysis.

Preservation for compilers In a compilation context, the strict preservation of behaviors is sometimes not even possible to achieve. A compiler can be forced to reduce the set of observable behaviors, when the source language semantics is not deterministic. For example, in C, the compiler must choose an expression evaluation order among the several possibilities allowed by the C standard. The compiled program P_2 has thus less behaviors than the source program P_1:

$$Beh(P_2) \subseteq Beh(P_1) \tag{3.1}$$

Note that, compared to the previous criterion, the inclusion is reversed here. By requiring the compiler not to create new behaviors, this inclusion allows concluding that, if the source program P_1 is safe then so is its compiled version P_2. Such a reasoning is necessary when the analysis is done at source level.

The above inclusion does not exactly match what compilers are allowed to do in practice. More precisely, it does not account for *semantics improving*, i.e. when some erroneous executions are removed from the source program. Indeed, compilers are allowed to remove useless computations; whenever the computation led to an error in the initial program, the optimized program can have more observable behaviors than the source. For instance, the statement `x:= a/0` can be removed whenever the variable `x` is not used later in the program. Removing this statement, the program can execute beyond the faulty instruction. This issue can be solved by relaxing the criteria to

$$(Beh(P_1) \cap Wrong = \emptyset) \Longrightarrow Beh(P_2) \subseteq Beh(P_1) \tag{3.2}$$

where *Wrong* denotes the observations that go wrong occuring whenever the program semantics gets stuck. As with inclusion 3.1, this criteria implies that the compiled program P_2 cannot go wrong whenever the source program P_1 does not go wrong – a fact that can be verified by a static analysis.

Criteria 3.2 can be used to prove each compilation phase separately. Suppose that two phases are proved to satisfy the criteria. Program P_1 is first translated to P_2 and then P_2 is translated to P_3. The final criteria for the whole chain, $Beh(P_1) \cap Wrong = \emptyset \Rightarrow Beh(P_3) \subseteq Beh(P_1)$, is easily obtained by composing the auxiliary correctness results.

Observational semantics allows for defining what information should be preserved. On top of this, we reviewed several preservation criteria, that can be chosen according to the application case. In this document, the two IRs we will study respectively instantiate these two criteria. We will discuss in Chapter 7 on how the two can be composed, and comment on this composition. In the next section, we present how such preservation results can be proved formally, relying on simulations between transition systems.

3.2 Simulation relations

3.2.1 Simulations for semantics preservation

To prove that programs P_1 and P_2 meet one of the above requirements, the basic idea is to define a simulation relation $\mathcal{R} \subseteq \Sigma_1 \times \Sigma_2$ on the states of the two corresponding labelled transition systems. The intended meaning of a simulation is that the execution steps of one system can be mimicked by executions steps of the other. Along both system executions, the respective semantic states will stay related through $\mathcal{R}$. The relation $\mathcal{R}$ is carefully defined so that it implies the preservation of observation between the two systems. We consider that the sets of labels include the silent label τ and the failure label $\times$, we thus drop the subscripts to lighten the notations. In this section, we omit the proofs of the theorems. They can be found in [Ler09].

Simulation schemes Depending on the transition systems to be related, there exist various kinds of simulation relations. We give now the simulation schemes that are mostly used in practice. The simplest one is a lock-step simulation relation, in which both systems executions are matching at each single step.

Definition 3.5 (Lock-step simulation)**.** *Consider two LTS $S_1 = (\Sigma_1, I_1, F_1, L_1, \rightarrow_1)$ and $S_2 = (\Sigma_2, I_2, F_2, L_2, \rightarrow_2)$. A binary relation $\mathcal{R} \subseteq \Sigma_1 \times \Sigma_2$ is a lock-step simulation of S_1 by S_2 if:*

- *for all s_1, s_2, ℓ, $s_1 \ \mathcal{R} \ s_2$ and $s_1 \xrightarrow{\ell}_1 s_1'$ implies that there exists $s_2' \in \Sigma_2$ such that $s_2 \xrightarrow{\ell}_2 s_2'$ and $s_1' \ \mathcal{R} \ s_2'$*
- *for all $s_1 \in I_1$, there exists $s_2 \in I_2$ such that $s_1 \ \mathcal{R} \ s_2$*
- *for all $s_1 \in F_1, s_2 \in \Sigma_2$, $s_1 \ \mathcal{R} \ s_2$ implies that $s_2 \in F_2$*

$$\begin{tikzcd} s_1 \arrow[r, "\ell"] \arrow[d, dash, "\mathcal{R}"'] & s_1' \arrow[d, dashed, dash, "\mathcal{R}"] \\ s_2 \arrow[r, dashed, "\ell"] & s_2' \end{tikzcd}$$

The semantic relation can be depicted by the diagram on the right, where solid lines denote hyptoheses and dashed lines represent conclusions.

The correctness of such a simulation diagram follows from the preservation of labels:

Theorem 3.1 (Lock-step simulation correctness)**.** *Let S_1 and S_2 be two LTS. Let $\mathcal{R}$ a lock-step simulation of S_1 by S_2. Then for all $b \in Obs(S_1), b \notin Wrong \Rightarrow b \in Obs(S_2)$.*

As an example, consider a transformation that optimizes a While program and produces another While program, by removing dead-copies: an instruction like `x := y` is replaced by a `skip` whenever `x` is not read later in the program before being redefined and it is not an external variable. Suppose that we use the observational semantics of While as defined in Figure 3.2. The relation $\mathcal{R}$ should first keep track of what allows to prove the observation preservation: $\langle s, \rho \rangle \ \mathcal{R} \ \langle s', \rho' \rangle \triangleq \rho \downarrow_{Ext} = \rho' \downarrow_{Ext}$. But as is, the relation will not make the proof go through. For instance, for one of the cases in the proof, one will have to match the execution step of a conditional statement in the initial program with another conditional. In order to be able to match the computation step, one will thus have to prove that the condition evaluates to the same value in both programs. But the condition might be expressed with non-external variables. Hence, only knowing that environments match on *Ext* will not make one able to conclude. The relation $\mathcal{R}$ must encode a stronger invariant: the environments of the two programs are equal for all the variables (including the external ones) that are live at

the current program point. It is also necessary to reinforce the invariants of statements of $\mathcal{R}$-related semantic states:

$$\langle s, \rho \rangle \ \mathcal{R} \ \langle s', \rho' \rangle \ \triangleq \ s \simeq s' \ \wedge \ \forall x, x \in \mathit{Live}(s) \Rightarrow \rho(x) = \rho'(x)$$

where $\simeq \subseteq \mathit{stmt} \times \mathit{stmt}$ encodes the syntactic correspondance between programs: only a dead copy can be mapped to a `skip`, and other statements are unchanged. In other words, a simulation relation $\mathcal{R}$ often carries some useful invariants about the transformation, that will make the proof possible.

For certain pairs of systems, a lock-step simulation scheme is too strong. Either of the systems might have to wait for the other to reach a state in which both of them will match; this is the case where the second system has been optimized (some computations have been removed), or when an atomic computation in the first system has been decomposed in several computation steps in the second one (as when translating While into TAC). This correspondence is captured by the star simulation scheme (Definition 3.6). This kind of simulation requires extra attention in the infinite case [AL91, Ler09]. Suppose the first system stutters with execution trace $s_1 s_2 s_3 s_4 \ldots$ where all s_i give rise to a silent step, and the second system gets stuck in a state s' related to all s_i through $\mathcal{R}$. In this case, a (silently) diverging behavior can be matched with a finite (stuck) behavior. The first system should hence not be allowed performing an infinite number of stutter steps. A solution is to introduce a measure on the states Σ_1, that must strictly decrease at each stuttering step. Formally, a measure $|.|$ is a function $\Sigma_1 \to M$, where M is a well-founded ordered set.

Definition 3.6 (Star simulation)**.** *Let $S_1 = (\Sigma_1, I_1, F_1, L_1, \rightarrow_1)$ and $S_2 = (\Sigma_2, I_2, F_2, L_2, \rightarrow_2)$ be two LTS. A binary relation $\mathcal{R} \subseteq \Sigma_1 \times \Sigma_2$ is a star simulation of S_1 by S_2 if:*

- *for all s_1, s_2, ℓ, if $s_1 \ \mathcal{R} \ s_2$ and $s_1 \xrightarrow{\ell}_1 s_1'$ then*
  - *either there exists $s_2' \in \Sigma_2$ s.t. $s_2 \xrightarrow{\ell}_2^+ s_2'$ and $s_1' \ \mathcal{R} \ s_2'$*
  - *or $|s_1'| < |s_1|$ and there exists $s_2' \in \Sigma_2$ such that $s_2 \xrightarrow{\ell}_2^* s_2'$ and $s_1' \ \mathcal{R} \ s_2'$*
- *for all $s_1 \in I_1$, there exists $s_2 \in I_2$ such that $s_1 \ \mathcal{R} \ s_2$*
- *for all $s_1 \in F_1, s_2 \in \Sigma_2$, $s_1 \ \mathcal{R} \ s_2$ implies that $s_2 \in F_2$*

$$\begin{array}{ccc} s_1 & \xrightarrow{\ell}_1 & s_1' \\ \mathcal{R} \Big| & & \Big| \mathcal{R} \\ s_2 & \overset{\ell}{\dashrightarrow}_2^* & s_2' \end{array}$$

where $\dot{\rightarrow}^+$ (resp. $\dot{\rightarrow}^*$) denotes the transitive (resp. reflexive-transitive) closure of $\dot{\rightarrow}$, where labels are accumulated into sequences of labels, and the silent label τ is the neutral element for the trace concatenation. In this definition, each time the stutter step is matched by steps in the second system, the value of the measure can be reset to an arbitrary value. Hence, this measure still allows the first system performing infinite executions; it only suppresses infinite stutter steps.

Theorem 3.2 (Star simulation correctness)**.** *Let S_1 and S_2 be two LTS. Let $\mathcal{R}$ be a star simulation of S_1 by S_2. Then for all $b \in \mathit{Obs}(S_1), b \notin \mathit{Wrong} \Rightarrow b \in \mathit{Obs}(S_2)$.*

Another scheme of simulation that we will use in our work is the plus simulation, where the simulating system does not have to wait for the first system, and only needs a non-empty sequence of steps to reach a matching state.

Definition 3.7 (Plus simulation)**.** *Let $S_1 = (\Sigma_1, I_1, F_1, L_1, \rightarrow_1)$ and $S_2 = (\Sigma_2, I_2, F_2, L_2, \rightarrow_2)$ be two LTS. A binary relation $\mathcal{R} \subseteq \Sigma_1 \times \Sigma_2$ is a plus simulation of S_1 by S_2 if:*

- *for all s_1, s_2, ℓ, if $s_1 \ \mathcal{R} \ s_2$ and $s_1 \xrightarrow{\ell}_1 s'_1$,*
  there exists $s'_2 \in \Sigma_2$ s.t. $s_2 \xrightarrow{\ell}_2^+ s'_2$ and $s'_1 \ \mathcal{R} \ s'_2$
- *for all $s_1 \in I_1$, there exists $s_2 \in I_2$ such that $s_1 \ \mathcal{R} \ s_2$*
- *for all $s_1 \in F_1, s_2 \in \Sigma_2$, $s_1 \ \mathcal{R} \ s_2$ implies that $s_2 \in F_2$*

$$\begin{array}{ccc} s_1 & \xrightarrow{\ell}_1 & s'_1 \\ \mathcal{R} \Big| & & \Big| \mathcal{R} \\ s_2 & \overset{\ell}{\dashrightarrow}_2^+ & s'_2 \end{array}$$

This simulation implies the existence of a star simulation. The correctness theorem of such a simulation scheme thus holds as a corollary of Theorem 3.2.

Theorem 3.3 (Plus simulation correctness)**.** *Let S_1 and S_2 be two LTS. Let $\mathcal{R}$ be a plus simulation of S_1 by S_2. Then for all $b \in Obs(S_1), b \notin Wrong \Rightarrow b \in Obs(S_2)$.*

Forward and backward simulation Looking back at the preservation criteria of Section 3.1.3 and the above correctness theorems of simulations, one can notice that the initial program P_1 and its IR P_2 will not always play the same role in the simulation. According to the (sufficient) criteria for analyses, i.e. $Beh(P_1) \subseteq Beh(P_2)$, P_2 must be proved to simulate P_1. This is what we will refer to as a *forward simulation*, following the naming convention of [Ler09][2]. To prove criteria $Beh(P_1) \subseteq Beh(P_2)$, one must include in the simulation relation the preservation of going wrong behaviors.

Conversely, when proving a compiler to be semantics preserving, we have to prove that P_1 simulates P_2 (*backward simulation*). Most of the time, proving a forward simulation is easier than proving a backward simulation. The reason is that the initial program is what is assumed and known, while the IR program depends on it. We can thus easily reason by case analysis or induction on the execution of P_1, and exploit the transformation specification to prove that there is a matching step. In the case where the target language $\mathcal{L}_2$ is deterministic (in the sense of Definition 3.8), and under the hypothesis that $Beh(P_1) \cap Wrong \neq \emptyset$ (which is implied by the notion of non-blocking system of Definition 3.9), proving the existence of a forward simulation allows to show the preservation criteria 3.2. (see Theorem 3.4).

Definition 3.8 (Determinism)**.** *An LTS $(\Sigma, I, F, L, \rightarrow)$ is deterministic if:*

- *I and F are singleton sets*
- *$\rightarrow$ is functional, i.e. $\forall s, s', s'', \ell, \ell', \quad s \xrightarrow{\ell} s' \wedge s \xrightarrow{\ell'} s'' \Longrightarrow s' = s''$ and $\ell = \ell'$*
- *for all final state $s \in F$ all state $s' \in \Sigma$ and all label $\ell \in L$, $s \not\xrightarrow{\ell} s'$*

For instance, the observational semantics we defined for While is deterministic (for fixed entry arguments). On the other hand, the observational semantics of C is not, because the evaluation order of binary operators is not fixed, and the evaluation of operands can have side-effects. For instance, the binary operator + does not correspond to any sequence point, and the expression `f()` + `g()` could be evaluated from left to right, from right to left or could even interleave the evaluation of the two functions.

Definition 3.9 (Non-blocking)**.** *An LTS $(\Sigma, I, F, L, \rightarrow)$ is non-blocking if $I \neq \emptyset$ and for all states $s, s' \in \Sigma$ and label $\ell \in L$, $s \xrightarrow{\ell}* s'$ implies either that $s' \in F$ or there exists s'' and ℓ' such that $s' \xrightarrow{\ell'} s''$.*

[2]These terms should not be confused with the classification of forward or backward simulations used in the automata and temporal logics literature of simulations, e.g. [LV92].

Intuitively, Definition 3.9 says that in a non-blocking system, unless the current state is final, a computation step is always possible, i.e. the corresponding program does not get stuck in this state (hence, the system does not go wrong). For instance, the semantics we defined for While only yields non-blocking systems. But in the case where While would contain integer division, a system performing a division by zero would be blocking: the relation $\downarrow$ for expression evaluation would not include, for any value v, the rule $(x/0) \downarrow v$, so the system would get stuck when executing e.g. `y := x/0`.

Theorem 3.4. *Let S_1 and S_2 be two LTS, such that S_1 is non-blocking and S_2 is deterministic. Let $\mathcal{R}$ be a forward simulation of S_1 by S_2. Then $Obs(S_1) \cap Wrong = \emptyset \Rightarrow Obs(S_2) \subseteq Obs(S_1)$.*

We illustrate with an informal proof how the previous results interact. Let $b \in Obs(S_2)$, we have to show that $b \in Obs(S_1)$. First, because S_1 is non-blocking, it has a non-empty set of observable behaviors. Second, the determinism of S_2 implies that b is the only observable behavior of S_2. Thus, the forward simulation $\mathcal{R}$ implies that S_1 has b as a unique behavior.

In Chapter 5, the source and target languages of the SSA generation are deterministic, and we work under the hypothesis of semantically well-defined source programs. Thus, we will rely on this theorem to prove the preservation of the SSA phase of the compiler using a forward lock-step simulation. Similar theorems can be proved for star and plus simulations, we do not detail them here.

The determinism and non-blocking hypotheses used for proving the above result can also be weakened into *determinacy* and *receptiveness* respectively, by distinguishing in the transition system the program from its external environment (see [ŠVZN+11]).

3.2.2 Simulations as semantic transformations

As presented in the previous section, the simulation relations are carefully chosen so that they imply the preservation of observable behaviors. As pointed out in Chapter 2 an IR is not always defined with a definitive set of analyses in mind, so that the observation is not known in advance. Still, there exists a semantic relation between a given program and its IR, that can be expressed in the framework of simulations by relaxing the constraint of preservation of labels or traces of labels.

Definition 3.10 (Relaxed lock-step simulation)**.** *Consider two LTS $S_1 = (\Sigma_1, I_1, F_1, L_1, \rightarrow_1)$ and $S_2 = (\Sigma_2, I_2, F_2, L_2, \rightarrow_2)$. A binary relation $\mathcal{R} \subseteq \Sigma_1 \times \Sigma_2$ is a relaxed lock-step simulation of S_1 by S_2 if:*

- *for all s_1, s_2, ℓ, $s_1 \ \mathcal{R} \ s_2$ and $s_1 \xrightarrow{\ell}_1 s_1'$ implies that there exists $s_2' \in \Sigma_2, \ell' \in L_2$ such that $s_2 \xrightarrow{\ell'}_2 s_2'$ and $s_1' \ \mathcal{R} \ s_2'$*
- *for all $s_1 \in I_1$, there exists $s_2 \in I_2$ such that $s_1 \ \mathcal{R} \ s_2$*
- *for all $s_1 \in F_1, s_2 \in \Sigma_2$, $s_1 \ \mathcal{R} \ s_2$ implies that $s_2 \in F_2$*

$$\begin{array}{ccc} s_1 & \xrightarrow{\ell}_1 & s_1' \\ \mathcal{R} \Big| & & \Big| \mathcal{R} \\ s_2 & \overset{\ell'}{\dashrightarrow}_2 & s_2' \end{array}$$

By an auxiliary specification of the correspondance between the emitted labels or traces of labels, such a simulation permits to exhibit the similarities and the differences between both semantics. In a way, the simulation becomes a semantic specification of the transformation.

This is our approach in Chapter 4, where we define a simulation relation (between JBC programs and their stackless IR), that not only expresses what is preserved by the transformation, but also captures precisely the differences between the two systems. This correctness statement can then be used later on by several analyses, without having to redo a complete proof of simulation each time a new analysis is performed on the IR.

3.3 Proof techniques for transformation

So far, we only considered the problem of semantics preservation between two programs. We now present two ways of lifting these results to program transformations, namely proof of transformations and translation validation. We consider a transformation T as a partial function from a language $\mathcal{L}_1$ to $\mathcal{L}_2$. This function is partial since it may fail to translate some $\mathcal{L}_1$ programs (because of e.g. syntax or typing errors). We assume that the criterion of semantics preservation is fixed, and write it $P_1 \sim P_2$.

3.3.1 Provably correct transformations

A provably correct transformation is a transformation for which one proves that for all inputs fed to the transformation, the output program is correct:

$$\forall P_1 \in \mathcal{L}_1, P_2 \in \mathcal{L}_2, T(P_1) = P_2 \Longrightarrow P_1 \sim P_2 \tag{3.3}$$

It requires to prove that the transformation T establishes a simulation relation between P_1 and P_2. In this case, the correctness of the transformation is proved once and for all, before the transformation is ever run. Note however that a transformation failing to translate any input program will trivially satisfy (3.3). Such a transformation is semantically correct but useless. In this work, we are primarily interested in transformation correctness. The ability of a transformation to successfully produce some code can be easily assessed empirically, by running it.

This approach is well adapted to the case where the observation is not known in advance. In Chapter 4, we prove the transformation correct in the sense of (3.3), where the relation $\sim$ is not semantics preservation but a relaxed simulation relation.

But proving transformations correct in such a direct way can sometimes be difficult. The proof is done with regard to the transformation algorithm (or code) itself, and the transformations performed by e.g. optimizing compilers are quite smart, and rely on complex data-stuctures for efficiency reasons (in time or space). For instance, the register allocation phase of CompCert relies on a graph coloring algorithm for computing the allocation. Proving an efficient implementation of a coloring algorithm is highly challenging, and another approach, namely translation validation, is preferable in those cases.

3.3.2 Translation validation

Translation validation was introduced by Pnueli et al. [PSS98] and Necula [Nec00] as an alternative to transformation proofs. In this approach, the transformation is not proved. Rather, each pair of input/output programs is checked *a posteriori* by a validator, a function $V : \mathcal{L}_1 \times \mathcal{L}_2 \rightarrow Bool$, to satisfy $P_1 \sim P_2$: the validator is intended to compute or to check the existence of a simulation between P_1 and P_2. It can thus be seen as an automatic way of proving program equivalence.

Among the techniques underlying the validation algorithm, two families can be distinguished. The first consists of the generation of verification conditions that are then checked either by model checking or automatic solvers [PSS98, BFG+05]. The second family relies on symbolic evaluation or static analysis of the pair of programs (e.g. [Nec00, TL08, RL10, TGM11]). Our work on the validation for the SSA generation algorithm lies in this category (see Chapter 5).

Correct validation In order for the validation approach to bring the same formal guarantees than a provably correct transformation, the validator must be proved to be correct:

$$\forall P_1 \in \mathcal{L}_1, P_2 \in \mathcal{L}_2, T(P_1) = P_2 \wedge V(P_1, P_2) = \mathit{true} \Longrightarrow P_1 \sim P_2$$

When the program transformation T is seen as a black box, that is not formalized (see e.g. the work of Tristan [TL08, TL09, TL10]), the hypothesis $T(P_1) = P_2$ is completely opaque (i.e. not exploitable in the proof), and the above requirement becomes:

$$\forall P_1 \in \mathcal{L}_1, P_2 \in \mathcal{L}_2, V(P_1, P_2) = \mathit{true} \Longrightarrow P_1 \sim P_2$$

Once the validator is proved correct, combining the untrusted transformation and the validator leads to a provably correct transformation by making the transformation fail if the validator rejects the pair of programs.

Completeness The semantics preservation $P_1 \sim P_2$ is undecidable in general, validators are thus usually incomplete. The validator will often fail to satisfy

$$\forall P_1 \in \mathcal{L}_1, P_2 \in \mathcal{L}_2, T(P_1) = P_2 \wedge P_1 \sim P_2 \Longrightarrow V(P_1, P_2) = \mathit{true}$$

In other words, such tools raise false alarms, and can fail to validate correct IR of the initial program. This occurs when either the spectrum of transformations targeted by the validator is too wide, or the program transformations are too complex (i.e. global on the code). A possibility for the validator to recover part of its completeness is to rely on some auxiliary information computed by the transformation that is taken as an additional input along with the program pair to validate. As an example, in CompCert, the register allocation phase is validated with the help of a coloring for the program interference graph.

Focused validators As discussed above, correct translation validation is an automatic and powerful approach to prove program transformations. Since a couple of years, it has been successfully employed in the context of verified compilation, and mainly thanks to their *focused* nature: the range of transformations they target are quite specific, compared to e.g. the work of Tristan [TGM11], that defines a single validator for a wide range of optimizations (including global constant and copy propagation and folding, common sub-expression elimination, lazy code-motion and loop invariant motion). For instance, Tristan [TL08] validates instruction schelduling; Rideau [RL10] focuses on register allocation, and Jourdan [JPL12] on a LR(1) automata generator for parsing context-free grammars.

First, it allows validators to accept most correct pairs of programs. One can sometimes show that the validator is complete for a given family or transformations ($\mathit{Spec} \models T$):

$$\forall T, P_1 \in \mathcal{L}_1, P_2 \in \mathcal{L}_2, \mathit{Spec} \models T \wedge T(P_1) = P_2 \wedge P_1 \sim P_2 \Longrightarrow V(P_1, P_2) = \mathit{true}$$

By the same token, dedicated validators are simpler to write, and thus easier to prove correct. It also make them more efficient, as they have fewer properties to infer or to check. We also argue that it is sometimes a more elegant approach than the direct proof of the transformation, as it allows from abstracting many implementation details and complex heuristics used by the transformation. In a sense, the validator can be seen as a specification of the transformation: it captures the essence of the transformation by isolating from the generation algorithm the invariant established by the transformation. We will follows this approach for our SSA validator (see Chapter 5).

3.4 Related work and conclusion

3.4.1 Relational approaches to transformation correctness

So far, we focused on presenting the technique that we use in our work, i.e. the simulation technique. Another interesting line of work is taken by Benton et al. [Ben04, BZ07, BH09]. The basic idea is to lift the semantic correspondence between two programs (to be proved equivalent) at a syntactic level, using types. Types are interpreted as binary relations over the semantics domains of the languages, that encode the semantic correspondence between programs: a type is a set of values together with an appropriate notion of equality, defined according to the observation.

In [Ben04], Benton tackles the problem of proving that an initial program and its optimized version (written in the same language) are observationally equivalent. Types are thus given to pairs of programs, with judgments of the form $\vdash P_1 \sim P_2 : \Phi \Rightarrow \Phi'$ that reads as follows: starting from two states related by Φ, executing programs leads to another pair of states related by Φ' (provided they both terminate). State types Φ and Φ' are similar to Hoare's logic pre- and post-conditions, specifying the context in which P_1 and P_2 are equivalent. Φ and Φ' are formulas on expressions of the language (including the program variables, that are taggued to indicate whether they refer to the state of the first or the second program). Consider the following simple example

$$\vdash (\texttt{if}\ x > 3\ \texttt{then}\ y := x\ \texttt{else}\ y := 7) \sim \texttt{skip} : (y_1 > 2 \wedge y_1 = y_2) \Rightarrow (y_1 > 2 \wedge y_2 > 2) \quad (3.4)$$

Such a judgment is interpreted as follows: suppose that programs start both in a state where y have the same value, which is greater than 2, they will terminate in another state where $y > 2$ still holds (but the value of the variable y may differ between the two final states). Hence, whenever $y > 2$ is the only fact that one cares about at the end of the program, the above conditional is superfluous, provided they are run in a state satisfying the pre-relation.

A provably sound type system is defined in [Ben04] that allows for deriving judgments such as (3.4). The goal of the type system is to directly *axiomatize* what are the correct transformations, and under what hypotheses. Some of the side conditions are purely logical. These can be discharged by an external solver, or a static analysis whose results are used by the transformation. This work already allows to treat non-trivial optimizations, such as loop invariant hoisting or sophisticated dead-code such as the above example. The type system is an elegant and unified way of presenting the proof of validity. But it does not prevent from requiring the addition of new typing rules each time a new optimization will have to be proved. In [BZ07, BH09], this relational approach has been applied to the proof (of type-safety and functional correctness) of a simple compiler from a simply-typed functional language down to a stack-based machine.

The relational approach to the correctness of transformations is an interesting alternative, but its primary goal is above all to tackle the issue of *compositional* compiler verification aiming at deriving the observation preservation between composed source and target programs from the observation preservation of their respective source and target components.

3.4.2 Summary

This chapter reviewed the notion of correct transformation, defined with regard to the formal semantics of programs. We focused on presenting the approach that we take in the rest of this

document, namely giving to programs an operational semantics, on top of which we define an observational semantics. The correctness of the transformation is then expressed through a preservation criterion on observable behaviors, to be carefully defined depending on the situation at hand. We then employ the time-honoured and proven techniques provided by the operational approach, namely simulation relations.

The operational approach (as opposed to the above discussed relational one) has non-negligeable other benefits. First, operational semantics, although formal, can be easily turned into an interpreter. This can be exploited to increase one's confidence in the definition of the semantics, by running the interpreter on programs and comparing program executions against standard implementations. Second, operational semantics and simulations are particularly easy to manipulate in a proof assistant like Coq, thanks again to their simplicity. Another advantage of simulation relations is that this framework allows not only to specify what is preserved by the transformation (with a correctness theorem), but makes it possible to express what information changes, and how it does, during the transformation. In Chapter 4, we rely on such a relaxed simulation diagram to specify the semantic correspondence existing between the input JBC programs and their stackless IR produced by our generation algorithm.

Finally, we review the two main approaches for proving transformations correct. First, proving a transformation correct consists in proving the existence of a simulation relation between any program input to the transformation and the output program. The approach of translation validation relies on a validator that checks, a posteriori, the pair of input and output programs. In order to achieve the same guarantee as a provably correct transformation, the validator must be proved correct: one must prove that the observation preservation holds for all pairs of programs accepted by the validator. We also stressed the fact that the incompleteness problem that can arise with translation validation can be overcome with the use of focused validators. In Chapter 5, the validator we define for the SSA generation is proved correct and complete with regard to a family of SSA generation algorithms. In addition, focused validator can be seen as a specification for the transformation of interest, as it describes sufficient conditions for the observation preservation to hold.

Chapter 4

A stackless IR for Java bytecode

4.1 Introduction

The Java [GJSB05] programming language is a high-level, object-oriented language, and compiled to Java Bytecode (JBC) to be executed by a Java Virtual Machine [LY99] (JVM). Java is convenient for the programmer, thanks to the various constructs and syntax sugaring it offers, but program analyses and optimizations are performed after the code has been compiled, for several reasons. First, the JBC instruction set is reduced, and high level constructs have been lowered by the compiler (e.g. generics, inner classes or try-catch-finally constructs). Second, the JBC is what is actually executed, hence the compiler needs not to be trusted. Third, Java code mobility sometimes implies that the source code is not available. A typical analysis performed at the bytecode level is the JBC Verifier (BCV), which checks type-safety, operand stack underflow and overflow, visibility constraints of attributes and code containment on the program just before it is executed.

The JBC is a stack-based language. Even if it is not as low-level as assembly code could be, the point we made in Chapter 2 is still valid: stack-based IRs are hard for program analysis and code manipulation, mainly because expression computations are not explicit in the program text. Most of the optimization and analysis tools for JBC work on a register-based IR that makes analyses simpler. This includes just-in-time compilers [BCF+99, ABC+02], external optimization frameworks [VRCG+99], and static analysis infrastructures [FL11, Fin].

Using such an IR may simplify the work of the analyzer but the overall correctness of the analysis now becomes dependent on the semantics-preserving properties of the transformation. Semantic correctness is particularly crucial when an analysis forms part of the security defense line. Surprisingly, the semantic foundations of these bytecode transformations have received little attention: the semantics of the IR is described only informally, and the mapping from JBC to the IR is not well specified. Even if those algorithms are initially conceptually simple, implementing them efficiently and scaling them to the whole language make those transformation non trivial.

In this chapter, we study a transformation from Java bytecode to a stackless IR that is similar to the one provided by the work of Whaley for the JikesRVM [BCF+99], which at the same time is efficient and has a formal correctness proof. The correctness criteria is in fact more than a semantics-preservation result, as we have presented in Chapter 3. We establish a fine-grained semantic correspondence between the initial bytecode program and its IR, that not only expresses what is preserved (e.g. function calls, all primitive-type computations, or the object initialization), but also what is modified, and how (e.g. the object allocation). This semantic characterization of the transformation can then be used in the correctness proof of the analyses subsequently performed on the IR.

The goal motivating this work was to build a static analysis framework for Java, that

would rely on such a stackless IR. This collaborative effort has resulted in the Sawja framework (Static Analysis Workbench for Java). Sawja is developed by the Celtique research group since 2009, the main contributor being Laurent Hubert [Hub10]. It provides a set of facilities for manipulating Java class files, and for prototyping JBC static analyses rapidly. For this to scale to the full JBC language, the transformation had to be efficient, and produce compact code. The transformation algorithm that we propose here works in a fixed number of passes over the bytecode program, and the implemented version of the algorithm takes care not to generate too many temporaries.

4.1.1 Key problems to address

In this work, we address three key language features that make a provably correct transformation challenging. We explain them now, and then provide an illustrative program example.

Operand stack The JBC is a stack-based code. The intensive use of the operand stack may make it difficult to adapt standard static analysis techniques that have been first designed for more standard (variable-based) 3-address codes. As noticed by Logozzo and Fähndrich [LF08], a naive translation from stack-based code to 3-address code may result in an explosion of temporary variables, which in turn may dramatically affect the precision of non-relational static analyses (such as intervals) and render some of the most costly analyses (such as polyhedral analysis) infeasible. The current transformation keeps the number of extra temporary variables at a reasonable level without using auxiliary iterative analyses such as copy propagation.

Splitted object creation The object creation scheme of the JVM is another feature which is difficult to track because it is done in two distinct steps: (i) raw object allocation and (ii) constructor call. References to uninitialized objects are systematically pushed and duplicated on the operand stack, which makes it difficult for an analysis to recover this sequence of actions. The BCV not only enforces type safety of bytecode programs but also a complex object initialization property: an object cannot be used before an adequate constructor has been called on it. The BCV verifies this by tracking aliases of uninitialized objects in the operand stack, but this valuable alias information is lost for subsequent static analyses. The present transformation rebuilds the initialization chain of an object with the instruction $x :=$ `new` $C(arg_1, ..., arg_n)$. This specific feature (used e.g in [Hub08] in a null pointer analysis) puts new constraints on the formalization because object allocation order is no longer preserved.

Exception throwing order A last difficulty for such a bytecode transformation is the wealth of dynamic checks used to ensure intrinsic properties of the Java execution model, such as the absence of null-pointer dereferencings, out-of-bound array accesses, *etc.* The consequence is that many instructions may raise different kinds of exceptions and any sound transformation must take care to preserve the exception throwing order. The difficulty is here that the bytecodes that may raise an exception can be moved forward in the code by the transformation. Explicit exception checks are easier to handle by analyses and optimizations [BCF+99, FKR+00]; by the same token, they help preserving the exception throwing order.

```
B f(int x, int y) {
  return (new B(x/y,new A()));
}
```

(a) Source code

```
B f(x,y) {
   0: t1:=new A()
   1: t2:=new B(x/y,t1)
   2: vreturn t2
}
```

(b) BIR code (incorrect)

```
B f(x,y) {
  0: new B
  1: dup
  2: load y
  3: load x
  4: div
  5: new A
  6: dup
  7: constructor A
  8: constructor B
  9: vreturn
}
```

(c) BC code

```
B f(x,y) {
  0: mayinit B
  4: notzero y
  5: mayinit A
  7: t1:=new A()
  8: t2:=new B(x/y,t1)
  9: vreturn t2
}
```

(d) BIR code (correct)

Colors indicate boundaries of related code fragments

Figure 4.1: Example of source code, bytecode and two possible transformations

Illustrating example Figure 4.1 presents an example program illustrating these issues. For more readability, we will also refer to Figure 4.1a that gives the corresponding Java source code. Its corresponding bytecode version (Figure 4.1c) shows the JVM object initialization scheme: an expression `new A()` is compiled to the sequence of lines [5; 6; 7]. A new object of class `A` is first allocated in the heap and its address is pushed on top of the operand stack. The address is then duplicated on the stack by the instruction `dup` and the non-virtual method `A.⟨init⟩()` is called with the bytecode `constructor A`, consuming the top of the stack. The copy is left on the top of the stack and represents from now on an *initialized* object. This initialization by side-effect is particularly challenging for the BCV [FM99] which has to keep track of the alias between uninitialized references on the stack. Using a similar approach, we are able to *fold* the two instructions of object allocation and constructor call into a single IR instruction. Figure 4.1b shows a first attempt of such a fusion. However, in this example, side-effect free expressions are generated in a naive way which *changes the semantics* in several ways. First, the program does not respect the *allocation order*. This is unavoidable if we want to keep side-effect free expressions and still re-build object constructions. The allocation order may have a functional impact because of the static class initializer `A.⟨clinit⟩` that may be called implicitly by the JVM when reaching an instruction `new A` (in which case the class is first loaded and then initialized). In Figure 4.1b this order is not preserved since `A.⟨clinit⟩` may be called before `B.⟨clinit⟩` while the bytecode program follows an inverse order. In Figure 4.1d this problem is solved using a specific instruction `mayinit A` that makes explicit the potential call to a static initializer.

The second major semantic problem of the program in Figure 4.1b is that it does not respect the *exception throwing order* of the bytecode version. In Figure 4.1b the call to `A()` may appear before the `ArithmeticException` exception that can be raised when evaluating `x/y`. The program in Figure 4.1d solves this problem using a specific instruction `notzero y` that explicitly checks if `y` is non-zero and, if not, raises an `ArithmeticException` exception.

Figure 4.2 gives an overview of the transformation algorithm on the example program. Figure 4.2a gives the BC code (identical to Figure 4.1c), Figure 4.2c the BIR code that the algorithm generates, without any simplification. The algorithm is based on a symbolic execution of the BC code, using a stack of symbolic expressions, illustrated in Figure 4.2b. The

```
B f(x,y) {
  0: new B
  1: dup
  2: load y
  3: load x
  4: div
  5: new A
  6: dup
  7: constructor A
  8: constructor B
  9: vreturn
}
```

(a) BC code

```
   []
0: [B0]
1: [B0 :: B0]
2: [y :: B0 :: B0]
3: [x :: y :: B0 :: B0]
4: [x/y :: B0 :: B0]
5: [A5 :: x/y :: B0 :: B0]
6: [A5 :: A5 :: x/y :: B0 :: B0]
7: [t1 :: x/y :: B0 :: B0]
8: [t2]
9: []
```

(b) Abstract stack

```
B f(x,y) {
  0: mayinit B
  1: nop
  2: nop
  3: nop
  4: notzero y
  5: mayinit A
  6: nop
  7: t1:=new A()
  8: t2:=new B(x/y,t1)
  9: vreturn t2
}
```

(c) BIR code

Figure 4.2: **Overview of the transformation** – The BC code (Figure 4.2a) is symbolically executed using a stack of symbolic expressions (Figure 4.2b gives, at each point, the output symbolic stack, the top of the stack is the left-most element). Each BC instruction is transformed into BIR code (Figure 4.2c), and BC instructions dedicated to expressions computation are transformed into a `nop`.

algorithm traverses the BC method instruction array, and for each instruction, given an input symbolic stack, produces some BIR code (potentially a `nop` instruction), and modifies the symbolic stack according to the instruction being transformed. For instance, `load x` at point 3 pushes the symbolic expression `x` on top of the stack, and a BIR `nop` is generated. Instruction `div` pops the top two elements of the abstract stack, and pushes back the expression of their division `x/y` (see point 4 in Figure 4.2b). It generates the explicit checks BIR `notzero y` instruction. Object creation is handled with special symbolic expressions of the form C_{pc}, where C is the name of the class, and pc is the allocation point in the BC code.

The core of the generation algorithm performs a single pass over the BC code. This requires a special handling of join points in the control-flow graph: the input symbolic stack used at a join point can differ from one precedessor of the join point to another (two branches might compute distinct expressions). To solve the problem, we rely on a normalization of the symbolic stack at join point. We will come back to this later.

4.1.2 Contribution and content

The algorithm presented in Section 4.3 and proved correct in Section 4.4 takes care of these pitfalls. The input (BC) and IR (BIR) languages are presented in Section 4.2. The transformation demands that input programs pass the BCV. This is a mild constraint, as BCV invalid programs would not be executed by a JVM anyway. It makes additional assumptions on the use of uninitialized objects, that are checked during the transformation (see Section 4.3), but these patterns are not used by traditional Java compilers, and so far, we never encountered them in practice (see Section 4.5).

Our algorithm uses the state-of-the-art technique of symbolic code execution [BCF+99]. It allows dealing simultaneously with the above challenges. The main alternative techniques, overviewed in Section 4.6, proceed in at least two distinct phases on the code: naive code is first generated, it is then optimized in a second phase, using traditional compiler optimization techniques. We believe the symbolic execution scheme gives rise to a rather elegant correctness proof, compared to the combining of correctness proofs of separate phases.

This transformation has been implemented for the full JBC language (meeting the same

requirements) and is now at the heart of the Sawja static analysis framework. The library is described in Section 4.5, where we provide with an experimental evaluation of the IR generation showing it competes well with Soot, a state-of-the-art bytecode optimization framework.

4.2 The source and target languages

4.2.1 Languages syntax

We assume a common representation of programs for the bytecode and its IR, called BIR. We will describe their instruction sets in the next paragraphs. A program is a record containing a set of classes, a main class, and a lookup operator lookup. This operator is used to determine the method to be executed on a virtual method call instruction. For a given program P, $P.\mathsf{lookup}(C, m)$ returns, if it exists, the first method overriding the method m in the ancestors in P of the class C. In the following, the program P may be omitted, as it is clear from the context. A class is a record containing a name, a super class[1], a set of fields, a set of methods, and the special constructor method init, called for the initialization of an instance after it has been allocated (we assume a unique constructor per class). A method is a record containing a method name, a list of parameters in *var* (including the receiver specific variable `this`), and a method code of type *code*. The definition of variables and code is specific to the language, we define them below.

$$
\begin{array}{lcl}
\mathit{Prog} \ni P & ::= & \{\mathsf{classes} \in \mathcal{P}(\mathit{Class});\quad \mathsf{Main} \in \mathit{Class};\quad \mathsf{lookup} \in \mathit{Class} \to \mathit{Meth} \to \mathit{Meth};\} \\
\mathit{Class} \ni C & ::= & \{\mathsf{name} \in \mathbb{C};\quad \mathsf{super} \in \mathit{Class};\quad \mathsf{fields} \in \mathcal{P}(\mathbb{F});\quad \mathsf{meths} \in \mathcal{P}(\mathit{Meth});\quad \mathsf{init} \in \mathit{Meth};\} \\
\mathit{Meth} \ni m & ::= & \{\mathsf{name} \in \mathbb{M};\quad \mathsf{params} \in \mathit{var}\ \mathrm{list};\quad \mathsf{code} \in \mathit{code};\}
\end{array}
$$

$$\text{with} \quad \text{class names } \mathbb{C}, \quad \text{method names } \mathbb{M}, \quad \text{field names } \mathbb{F}$$

Figure 4.3: Programs, classes, and methods

Source language Our source language BC is an untyped stack-based sequential Java-like bytecode language with object construction, exceptions and virtual calls. Missing features, e.g. 64 bits values, static elements (static fields and static methods) or method overloading would make the current formalization heavier but do not introduce any new difficulties. The set of bytecodes we consider is given in Figure 4.4; it resembles the STACK language we presented in Chapter 2, extended with object-oriented features.

A new object of class `C` is allocated in the heap by the instruction `new` C. Then, it has to be initialized by calling its constructor `constructor` C. The super constructor is called through the same BC instruction, `constructor` B, where B is the direct super class of C[2]. In the JBC language, `constructor` B corresponds to `invokespecial` `B.`$\langle$`init`$\rangle$, but instruction `invokespecial` is used for many other cases (e.g. for calling a private method). Our dedicated bytecode focuses on the role for object constructors. Class fields are read and assigned with `getfield` f and `putfield` f (we suppose the resolution of field class has been done). A virtual method `m` is called on a receiver object with `invokevirtual` $C.m$.

[1] We assume the class `Object` is its own super class

[2] We will see in Section 4.3 that both cases have to be distinguished during the transformation.

$const$	::=	*constant*
	c | `null`	
var_{BC}	::=	BC *variables*
	x | x_1 | x_2 | ... `this`	
$instr_{\mathsf{BC}}$	::=	BC *instructions*
	`nop` | `dup`	
	| `push` c | `pop`	
	| `add` | `div`	
	| `load` var_{BC}	
	| `store` var_{BC}	
	| `new` C	
	| `constructor` C	
	| `getfield` f | `putfield` f	
	| `invokevirtual` $C.m$	
	| `if` pc | `goto` pc	
	| `return` | `vreturn`	
$code_{\mathsf{BC}}$	::=	BC *method code*
	| $instr$ array	

$tvar$	::=	*temporary variables*
	t | t_1 | t_2 | ...	
var_{BIR}	::=	BIR *variables*
	var_{BC} | $tvar$	
exp	::=	*side-effect free expressions*
	$const$ | var_{BIR}	
	| exp+exp | exp/exp | $exp.f$	
$instr_{\mathsf{BIR}}$	::=	BIR *instructions*
	`nop` | `mayinit` C	
	| `notnull` exp | `notzero` exp	
	| var_{BIR}:=exp | $exp.f$:=exp	
	| var_{BIR}:= `new` $C(exp, \ldots, exp)$	
	| exp.`super` $(C, exp, \ldots, exp)$	
	| var_{BIR}:=$exp.m(C, exp, \ldots, exp)$	
	| `if` exp pc | `goto` pc	
	| `return` | `vreturn` exp	
$code_{\mathsf{BIR}}$	::=	BIR *method code*
	| ($instr$ list) array	

Figure 4.4: Instructions of BC and BIR

Target language The BIR language (Figure 4.4) provides expressions and instructions for variable and field assignments. BIR distinguishes two kinds of variables: local variables in var_{BC} are identifiers already used at the BC level, while $tvar$ are fresh identifiers introduced in BIR. Like BC, BIR is unstructured, but BIR conditional jumps now depend on structured expressions.

Object creation and initialization is folded into the single instruction x:= `new` $C(e_1, \ldots e_n)$. BIR disambiguates the bytecode `constructor` C by providing a distinct super constructor call instruction e.`super`$(C', e_1, \ldots, e_n)$, where C' is the super class of C. Subsequent analyses requiring to distinguish the two situations will hence not have to rebuild the information.

The two assertions provided by BIR `notzero` e and `notnull` e respectively check that the expression e does not evaluate to zero or null and raise an exception if the check fails.[3] The transformation will insert all the required assertions prior to generating expressions. Hence, as a by-product, we obtain that the BIR expression evaluation is error-free and non-blocking. Finally, the BIR extra instruction `mayinit` C allows for showing that the class initialization order is preserved. For the sake of simplicity, we consider that class initializers are empty. Thus, `mayinit C` behaves as `nop`. Calling the class initializer $C.\langle$`clinit`$\rangle$ in the semantics could be added to the present work without bringing any new difficulty.

BIR program instructions are organized into an array of instruction lists. This way, the program counter does not index a single instruction, but rather the sequence of instructions that have been generated from a single BC instruction.

[3] In our formalization, heaps are infinite. Dealing with finite heaps would require preserving `OutOfMemory` exceptions. BIR would need to be extended with an instruction `checkheap` C, generated when transforming the BC instruction `new` C and checking if the heap available space is sufficient to allocate a C object.

$$
\begin{array}{rcl}
\mathsf{Val} & = & \mid (\mathsf{N}\ n),\ n \in \mathbb{Z} \\
 & & \mid (\mathsf{R}\ r),\ r \in \mathsf{Ref} \\
 & & \mid \mathsf{Null} \\
\overline{\mathsf{Val}} & = & \mathsf{Val} \cup \{\mathsf{Void}\}
\end{array}
\qquad
\begin{array}{rcl}
\mathit{InitTag} & = & \widetilde{\mathbb{C}}_{\mathbb{N}} \cup \mathbb{C} \\
\mathit{Object} & = & (\mathbb{F} \to \mathsf{Val})_{\mathit{InitTag}} \\
\mathit{Heap} & = & \mathsf{Ref} \rightharpoonup \mathit{Object} \\
\mathit{Error} & = & \{\mathsf{NP}, \mathsf{DZ}\}
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{Stack} & = & \mathsf{Val}^* & \\
\mathit{Env}_{\mathsf{BC}} & = & \mathit{var}_{\mathsf{BC}} \rightharpoonup \mathsf{Val} & \\
\mathit{State}_{\mathsf{BC}} & = & (\mathit{Heap} \times \mathbb{M} \times \mathbb{N} \times \mathit{Env}_{\mathsf{BC}} \times \mathit{Stack}) & \text{Normal state} \\
 & \cup & \big(\mathit{Heap} \times \overline{\mathsf{Val}}\big) & \text{Return state} \\
 & \cup & (\mathit{Heap} \times \mathbb{M} \times \mathbb{N} \times \mathit{Env}_{\mathsf{BC}})_{\mathit{Error}} & \text{Error state}
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{Env}_{\mathsf{BIR}} & = & \mathit{var}_{\mathsf{BIR}} \rightharpoonup \mathsf{Val} & \\
\mathit{State}_{\mathsf{BIR}} & = & \big(\mathit{Heap} \times \mathbb{M} \times (\mathbb{N} \times \mathit{instr}^*_{\mathsf{BIR}}) \times \mathit{Env}_{\mathsf{BIR}}\big) & \text{Normal state} \\
 & \cup & \big(\mathit{Heap} \times \overline{\mathsf{Val}}\big) & \text{Return state} \\
 & \cup & (\mathit{Heap} \times \mathbb{M} \times \mathbb{N} \times \mathit{Env}_{\mathsf{BIR}})_{\mathit{Error}} & \text{Error state}
\end{array}
$$

Figure 4.5: BC and BIR semantic domains

4.2.2 Semantics

The correctness of the BC2BIR transformation amounts not only to the input/output preservation, but also to all what is preserved by BC2BIR. BC and BIR semantics are designed to this end. We first describe semantic domains and the kind of transition relation we will need. We then describe the semantics of each language in a dedicated paragraph.

4.2.2.1 Semantic domains

Semantic domains are given in Figure 4.5. A value is either an integer, a reference or the special value Null. The operand stack is a list of elements of Val. An environment is a partial function from variables to Val.

An object is represented as a total function from its fields names $\mathbb{F}$ to values. One of the subtleties of BC2BIR is that, although the object allocation order is modified, it takes care of preserving a strong relation between objects allocated in the heap, as soon as their initialization has begun. Thus, we attach to objects an initialization tag $\in \mathit{InitTag}$. This was first introduced by Freund and Mitchell in [FM99], but we adapt it to our purpose. Following the Java convention, an object allocated at point pc by `new` C is uninitialized (tagged $\widetilde{C}_{pc}$) as long as no constructor has been called on it; an object is tagged C either if its initialization is ongoing (all along the constructor call chain) or completed when the `Object` constructor is called. Note that, unlike [FM99], *InitTag* does not track intermediate initialization status, but this can be recovered from the observational trace semantics (Section 4.2.2). The heap is a partial function from non-null references to objects. Each time a new object is allocated in the heap, the partial function is extended accordingly.

A *normal execution state* for BC is written $\langle h, m, pc, l, s\rangle$. It consists of a heap h, the current method m, the current program point pc, a local environment l and an operand stack s. A BIR normal execution state is written $\langle h, m, (pc, \ell), l\rangle$ where l is the local envrionment, and $(pc, \ell) \in \mathbb{N} \times \mathit{instr}^*_{\mathsf{BIR}}$ denotes the next instruction to execute (we comment it below). We do not model the usual call stack in execution states, but rely on a so-called mostly-small-step semantics [BGL06] (see Section 4.2.2.2), that is easier to handle in the proof. In the correctness theorem (Section 4.4), one BC step is matched by a sequence of BIR steps. The way we define

BIR program points avoids awkwardness in this matching by defining a program point as a pair $(pc, \ell) \in \mathbb{N} \times instr^*_{\mathsf{BIR}}$ where pc is the program counter and ℓ is the list of instructions being executed. The head element of the list defines the next instruction to execute. More detail is given in the semantic rules about the way the execution flows from one instruction to its successor.

A *return state* $\langle h, v\rangle$ is made of a heap h and a return value $v \in \mathsf{Val} \cup \{\mathsf{Void}\}$.

We also want the semantic preservation to deal with execution errors. We do not model exception catching in this work but it will not bring much difficulty thanks to the way we define *error states*. An error state is written $\langle h, m, pc, l\rangle_k$, where pc is the method program point of the faulty instruction and h and l describe the current context (heap and local environment). We also keep track of the kind of error k, written as a subscript, which is either a division by zero DZ or null pointer dereferencing NP. BC programs passing the BCV only get stuck in an error or return state of the main method. Note that error states of BIR are defined as in BC. Still, the $\mathbb{N}$ parameter uniquely determines the faulty program point. As will be seen in the next section, at most one assertion is generated per instruction list (the first in this list).

4.2.2.2 Transition relations

We give to BC and BIR an observational operational semantics, where transitions are labelled with observable event traces, allowing a fine-grained preservation criterion. Indeed, a correctness criterion only stating the preservation of return values would not bring much information to static analyses dealing with intermediate program points.

Observable events We push further the approach by observing all the program behavior aspects that are preserved by the transformation. We even observe local variable assignments, as this can help transfering a static analysis result from the BIR language to BC. The set *Evt* of events is defined as the union of the following sets:

$$
\begin{array}{lll|lll}
EvtS ::= & x \leftarrow v & \textit{local assignment} & EvtH ::= & r.f \leftarrow v & \textit{field assignment} \\
 & & & \mid & \mathsf{mayinit}(C) & \textit{class initializer} \\
EvtR ::= & \mathsf{ret}(v) & \textit{method return} & \mid & r.C.m(v_1, \ldots, v_n) & \textit{method call} \\
\mid & \mathsf{ret}(\mathsf{Void}) & & \mid & r \leftarrow C.\mathsf{init}(v_1, \ldots, v_n) & \textit{constructor} \\
 & & & \mid & r.C.\mathsf{init}(v_1, \ldots, v_n) & \textit{super constructor}
\end{array}
$$

with $v, v_1, \ldots, v_n \in \mathsf{Val}, r \in \mathsf{Ref}, x \in var, C \in \mathbb{C}, f \in \mathbb{F}$

Actions irrelevant to the correctness of the transformation are silent transitions labelled with τ. These include expression evaluation steps, as expressions are side-effect and error free. Note that, due to the modification of the object allocation order, we do not observe memory effect of the BC instruction `new` C. This is harmless thanks to the strong restrictions imposed by the BCV on the use of uninitialized references [FM99]. In the sequel, we let λ range over *Evt* and Λ range over sequences of labels. We also identify the empty event sequence with τ.

Transition relations The language semantics we describe in the next sections uses different kind of transitions. First, a single step transition gives rise to a sequence of observable events. We will write such a *multi-label* transition $\xrightarrow{\Lambda}$. We write $\dot{\Rightarrow}$ for the transitive closure of $\dot{\rightarrow}$, and label such a *multi-step* transition with the concatenation of event traces of each step.

Second, the operational semantics we give to programs is mostly-small-step, i.e. method calls are executed in a big-step fashion. The execution of the callee is considered in its entirety

from the starting state to its return (or error) state. The above definitions of multi-step and multi-label transitions are mutually recursive. The caller will perform one $\dot{\rightarrow}$ step because the callee performs a $\dot{\Rightarrow}$ step, itself composed of several $\dot{\rightarrow}$ steps. We explain below how the event traces are transfered from the callee to the caller.

For simplifying the inductive reasoning about this mutually recursive definition of transition relations, each transition is parametrized by a natural n representing the *call-depth* of the transition, i.e. the number of method calls that arise within this computation step. Concerning one-step transitions, this index is incremented when calling a method or a constructor. For multi-step transitions, we define the call-depth index by the following two rules:

$$\frac{s_1 \xrightarrow{\Lambda}_n s_2}{s_1 \overset{\Lambda}{\Rightarrow}_n s_2} \qquad \frac{s_1 \overset{\Lambda_1}{\Rightarrow}_{n_1} s_2 \qquad s_2 \xrightarrow{\Lambda_2}_{n_2} s_3}{s_1 \xrightarrow{\Lambda_1.\Lambda_2}_{n_1+n_2} s_3}$$

4.2.2.3 Semantics of BC

BC semantic rules are given in Figure 4.6. Figure 4.7 defines the error cases. We describe here the semantic rules of BC that are related to object-oriented features, other are straightforward.

Basic rules In rule for the bytecode `new` C, $\mathsf{newObject}(C, h)$ allocates a new object of class C in the heap h, pointed to by $(\mathsf{R}\ r)$, and returns this new reference, as well as the new heap h'. Function $\mathsf{zeros}(C)$ returns a new object whose fields are set to their default value (zero for integers and Null for references) and its initialization tag is set to $\widetilde{C}_{pc}$. By definition, $h' = h[r \mapsto \mathsf{zeros}(C)_t]$, with $t = \widetilde{C}_{pc}$. Object fields are accessed with `getfield` f, and modified with `putfield` f. The object must be initialized, and its initialization status does not change.

Method calls Dynamic methods can only be called on initialized objects (see the rule for `invokevirtual` in Figure 4.6). The method resolution $\mathsf{lookup}(m, C')$ returns the right method to execute. The current object, pointed to by $(\mathsf{R}\ r)$, is passed to the callee method m' in the special local variable `this`. Other arguments are passed in variables x_1 to x_n, the identifiers given by $m'.\mathsf{params}$.

The whole method m' is executed and terminates in state $\langle h', v\rangle$, using a multi-step transition. This execution produces an event trace Λ. This trace contains events related to the local variables of the method, its method calls, some heap modifications, and the final return event. While events in *EvtS* and *EvtR* only concern m' (they are irrelevant to m), events related to the heap should be seen outside m' (i.e. from each caller) as well, since the heap is shared across methods. We hence define the filtering Λ_H of an event trace Λ to a category of events *EvtH* as the maximal subtrace of Λ that contains only events in *EvtH*. If the method terminates, then the caller m makes a multi-label step, the trace Λ_H being exported from the callee m'. Constructor calls rules are based on the same idea.[4]

Object construction We now describe the semantic rules of constructor calls (see Figure 4.6). The rule on the left is used when calling the first constructor on an object: the object is not initialized yet. The constructor is called with the reference to the object in its `this` register. At the beginning of the constructor, the object initialization status is updated

[4]Note the semantics does not distinguish between executions that get stuck and executions that do not terminate in method calls. We discuss this point in a dedicated paragraph in Section 4.4.

$$\frac{m.\mathsf{code}[pc] = \texttt{nop}}{\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, s\rangle}$$

$$\frac{m.\mathsf{code}[pc] = \texttt{dup}}{\langle h, m, pc, l, v{::}s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, v{::}v{::}s\rangle}$$

$$\frac{m.\mathsf{code}[pc] = \texttt{pop}}{\langle h, m, pc, l, v{::}s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, s\rangle}$$

$$\frac{\begin{array}{c} m.\mathsf{code}[pc] = \texttt{push}\ c \\ c \neq \texttt{null} \Leftrightarrow v = (\mathsf{N}\ c) \\ c = \texttt{null} \Leftrightarrow v = \mathsf{Null} \end{array}}{\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, v{::}s\rangle}$$

$$\frac{\begin{array}{c} m.\mathsf{code}[pc] = \texttt{add} \\ v_1 = (\mathsf{N}\ n_1) \quad v_2 = (\mathsf{N}\ n_2) \\ v' = (\mathsf{N}\ (n_1 + n_2)) \end{array}}{\langle h, m, pc, l, v_1{::}v_2{::}s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, v'{::}s\rangle}$$

$$\frac{\begin{array}{c} m.\mathsf{code}[pc] = \texttt{div} \\ v_1 = (\mathsf{N}\ n_1) \quad v_2 = (\mathsf{N}\ n_2) \\ n_2 \neq 0 \quad v' = (\mathsf{N}\ (n_1/n_2)) \end{array}}{\langle h, m, pc, l, v_1{::}v_2{::}s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, v'{::}s\rangle}$$

$$\frac{m.\mathsf{code}[pc] = \texttt{load}\ x}{\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, l(x){::}s\rangle}$$

$$\frac{m.\mathsf{code}[pc] = \texttt{store}\ x}{\langle h, m, pc, l, v{::}s\rangle \xrightarrow{[x \leftarrow v]}_0 \langle h, m, pc+1, l[x \mapsto v], s\rangle}$$

$$\frac{m.\mathsf{code}[pc] = \texttt{goto}\ pc'}{\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc', l, s\rangle}$$

$$\frac{\begin{array}{c} m.\mathsf{code}[pc] = \texttt{if}\ pc' \\ \text{if } n \neq 0 \text{ then } pc'' = pc+1 \text{ else } pc'' = pc' \end{array}}{\langle h, m, pc, l, (\mathsf{N}\ n){::}s\rangle \xrightarrow{\tau}_0 \langle h, m, pc'', l, s\rangle}$$

$$\frac{m.\mathsf{code}[pc] = \texttt{vreturn}}{\langle h, m, pc, l, v{::}s\rangle \xrightarrow{[\mathsf{ret}(v)]}_0 \langle h, v\rangle}$$

$$\frac{m.\mathsf{code}[pc] = \texttt{return}}{\langle h, m, pc, l, s\rangle \xrightarrow{[\mathsf{ret}(\mathsf{Void})]}_0 \langle h, \mathsf{Void}\rangle}$$

$$\frac{\begin{array}{c} m.\mathsf{code}[pc] = \texttt{putfield}\ f \\ h(r) = o_C \quad o' = o[f \mapsto v] \quad h[r \mapsto o'] \end{array}}{\langle h, m, pc, l, v{::}(\mathsf{R}\ r){::}s\rangle \xrightarrow{[r.f \leftarrow v]}_0 \langle h', m, pc+1, l, s\rangle}$$

$$\frac{\begin{array}{c} m.\mathsf{code}[pc] = \texttt{getfield}\ f \\ h(r) = o_C \end{array}}{\langle h, m, pc, l, (\mathsf{R}\ r){::}s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, o(f){::}s\rangle}$$

$$\frac{\begin{array}{c} m.\mathsf{code}[pc] = \texttt{new}\ C \\ (h', (\mathsf{R}\ r)) = \mathsf{newObject}(C, h) \\ h'(r) = \mathsf{zeros}(C)_t \quad t = \widetilde{C}_{pc} \end{array}}{\langle h, m, pc, l, s\rangle \xrightarrow{[\mathsf{mayinit}(C)]}_0 \langle h', m, pc+1, l, (\mathsf{R}\ r){::}s\rangle}$$

$$\frac{\begin{array}{c} m.\mathsf{code}[pc] = \texttt{invokevirtual}\ C.m' \\ h(r) = o_{C'} \quad V = v_1{::}\ldots{::}v_n \quad C' \subseteq C \\ \mathsf{lookup}(m', C') = mc \quad \langle h, \mathsf{init_state}(mc)\rangle \overset{\Lambda}{\Rightarrow}_n \langle h', v\rangle \\ \text{if } v = \mathsf{Void} \text{ then } s' = s \text{ else } s' = v{::}s \end{array}}{\langle h, m, pc, l, V{::}(\mathsf{R}\ r){::}s\rangle \xrightarrow{[r.C.mc(V)].\Lambda_H}_{n+1} \langle h', m, pc+1, l, s'\rangle}$$

$$\frac{\begin{array}{c} m.\mathsf{code}[pc] = \texttt{constructor}\ C \\ h(r) = o_t \quad t = \widetilde{C}_j \\ h' = h[r \mapsto o_C] \quad V = v_1{::}\ldots{::}v_n \\ \langle h', \mathsf{init_state}(C.\mathsf{init})\rangle \overset{\Lambda}{\Rightarrow}_n \langle h'', \mathsf{Void}\rangle \end{array}}{\langle h, m, pc, l, V{::}(\mathsf{R}\ r){::}s\rangle \xrightarrow{[r \leftarrow C.\mathsf{init}(V)].\Lambda_H}_{n+1} \langle h'', m, pc+1, l, s\rangle}$$

$$\frac{\begin{array}{c} m.\mathsf{code}[pc] = \texttt{constructor}\ C' \\ h(r) = o_C \quad C \subset C' \\ V = v_1{::}\ldots{::}v_n \\ \langle h, \mathsf{init_state}(C'.\mathsf{init})\rangle \overset{\Lambda}{\Rightarrow}_n \langle h', \mathsf{Void}\rangle \end{array}}{\langle h, m, pc, l, V{::}(\mathsf{R}\ r){::}s\rangle \xrightarrow{[r.C'.\mathsf{init}(V)].\Lambda_H}_{n+1} \langle h', m, pc+1, l, s\rangle}$$

$$\forall m \in \mathbb{M}, \mathsf{init_state}(m) = (m, 0, [\texttt{this} \mapsto (\mathsf{R}\ r), x_1 \mapsto v_1 \ldots x_n \mapsto v_n], \varepsilon), \text{ where } m.\mathsf{params} = x_1{::}\ldots{::}x_n$$

Figure 4.6: BC operational semantics

$$\frac{m.\mathsf{code}[pc] = \texttt{div}}{\langle h, m, pc, l, v{::}(\mathsf{N}\ 0){::}s\rangle \xrightarrow{\tau}_0 \langle h, m, pc, l\rangle_{\mathsf{DZ}}} \qquad \frac{m.\mathsf{code}[pc] = \texttt{putfield}\ f}{\langle h, m, pc, l, v{::}\mathsf{Null}{::}s\rangle \xrightarrow{\tau}_0 \langle h, m, pc, l\rangle_{\mathsf{NP}}}$$

$$\frac{m.\mathsf{code}[pc] = \texttt{getfield}\ f}{\langle h, m, pc, l, \mathsf{Null}{::}s\rangle \xrightarrow{\tau}_0 \langle h, m, pc, l\rangle_{\mathsf{NP}}} \qquad \frac{\begin{array}{c} m.\mathsf{code}[pc] = \texttt{constructor}\ C \\ V = v_1{::}\ldots{::}v_n \end{array}}{\langle h, m, pc, l, V{::}\mathsf{Null}{::}s\rangle \xrightarrow{\tau}_0 \langle h, m, pc, l\rangle_{\mathsf{NP}}}$$

$$\frac{\begin{array}{c} m.\mathsf{code}[pc] = \texttt{invokevirtual}\ C.m' \\ V = v_1{::}\ldots{::}v_n \end{array}}{\langle h, m, pc, l, V{::}\mathsf{Null}{::}s\rangle \xrightarrow{\tau}_0 \langle h, m, pc, l\rangle_{\mathsf{NP}}} \qquad \frac{\begin{array}{c} m.\mathsf{code}[pc] = \texttt{invokevirtual}\ C.m' \\ \mathsf{lookup}(m', C') = mc \quad V = v_1{::}\ldots{::}v_n \quad C' \subseteq C \\ h(r) = o_{C'} \quad \langle h, \mathsf{init_state}(mc)\rangle \overset{\Lambda}{\Rightarrow}_n \langle h_e, mc, pc', l_e\rangle_k \end{array}}{\langle h, m, pc, l, V{::}(\mathsf{R}\ r){::}s\rangle \xrightarrow{[r.C.mc(V)].\Lambda_H}_{n+1} \langle h_e, m, pc, l\rangle_k}$$

$$\frac{\begin{array}{c} m.\mathsf{code}[pc] = \texttt{constructor}\ C \\ h(r) = o_t \quad t = \widetilde{C}_j \\ V = v_1{::}\ldots{::}v_n \quad h' = h[r \mapsto o_C] \\ \langle h', \mathsf{init_state}(C.\mathsf{init})\rangle \overset{\Lambda}{\Rightarrow}_n \langle h_e, \mathsf{C.init}, pc', l_e\rangle_k \end{array}}{\langle h, m, pc, l, V{::}(\mathsf{R}\ r){::}s\rangle \xrightarrow{[r \leftarrow C.\mathsf{init}(V)].\Lambda_H}_{n+1} \langle h_e, m, pc, l\rangle_k} \quad \frac{\begin{array}{c} m.\mathsf{code}[pc] = \texttt{constructor}\ C' \\ h(r) = o_C \quad C \subset C' \\ V = v_1{::}\ldots{::}v_n \\ \langle h, \mathsf{init_state}(C'.\mathsf{init})\rangle \overset{\Lambda}{\Rightarrow}_n \langle h_e, C'.\mathsf{init}, pc', l_e\rangle_k \end{array}}{\langle h, m, pc, l, V{::}(\mathsf{R}\ r){::}s\rangle \xrightarrow{[r \leftarrow C'.\mathsf{init}(V)].\Lambda_H}_{n+1} \langle h_e, m, pc, l\rangle_k}$$

$$\forall m \in \mathbb{M}, \mathsf{init_state}(m) = (m, 0, [\texttt{this} \mapsto (\mathsf{R}\ r), x_1 \mapsto v_1 \ldots x_n \mapsto v_n], \varepsilon), \text{ where } m.\mathsf{params} = x_1{::}\ldots{::}x_n$$

Figure 4.7: BC operational semantics : error cases

to C. The rule on the right is used when calling a constructor on an object whose initialization is ongoing: the initialization tag of the object does not change.

Error cases Error cases are given in Figure 4.7. The instruction `div` might cause a division by zero if the second top element of the stack is (N 0). In this case, the execution goes into the error state $\langle h, m, pc, l\rangle_{\mathsf{DZ}}$. Similarly, reading or writing a field might dereference a null pointer (the kind of error is thus NP). Finally, concerning method and constructor calls, there are two cases: either the error is raised by the call itself (leading to $\langle h, m, pc, l\rangle_{\mathsf{NP}}$), or the error arises during the execution of the callee, at a given program point pc' (other side conditions are equal to the normal case). In this case, the error state is propagated to the caller: it ends in the error state of the same kind (NP or DZ) but parameterized by program point pc (the program point of the faulty instruction, from the point of view of the caller), heap h_e (the heap in which the error arose) and the caller local environment. This mechanism is very similar to JBC exception handlers.

4.2.2.4 Semantics of BIR

Expressions The (big-step) semantics of expressions is defined in a standard way by induction, relative to an environment l and a heap h. As it is clear from the context, we use the same symbols + and / for both syntaxic and semantic versions for addition and division. The semantics of expression is defined by the following relation defined on $Heap \times Env \times exp \times \mathsf{Val}$:

Expressions are side-effect free, as it facilitates their treatment in static analyses. In addition, the order of evaluation of the operands can be left unspecified without the semantics of expression becoming non-deterministic. In addition, we do not need to specify error cases,

$$\frac{}{h, l \vDash \mathtt{c} \Downarrow (\mathsf{N}\ c)} \qquad \frac{}{h, l \vDash \mathtt{null} \Downarrow \mathsf{Null}} \qquad \frac{x \in dom(l)}{h, l \vDash x \Downarrow l(x)}$$

$$\frac{h, l \vDash e_i \Downarrow (\mathsf{N}\ n_i) \text{ for } i = 1, 2}{h, l \vDash e_1 + e_2 \Downarrow (\mathsf{N}\ (n_1 + n_2))} \qquad \frac{h, l \vDash e_i \Downarrow (\mathsf{N}\ n_i) \text{ for } i = 1, 2 \qquad n_2 \neq 0}{h, l \vDash e_1 / e_2 \Downarrow (\mathsf{N}\ (n_1 / n_2))}$$

$$\frac{h, l \vDash e \Downarrow (\mathsf{R}\ r) \quad h(r) = o_{\mathsf{C}} \quad f \in dom(o_{\mathsf{C}})}{h, l \vDash e.f \Downarrow o_{\mathsf{C}}(f)}$$

Figure 4.8: Semantics of BIR expressions

thanks to the explicit exception checks.

Instructions Although they are not relevant in the trace equivalence statement, the modifications of generated temporary variables are made observable (they can help stating the correctness of a static analysis on BIR). We need to distinguish them from the τ event in order to be able to match execution traces. We thus split the set *EvtS* into two parts:

$$\begin{array}{rcl} EvtS & = & EvtSLoc \cup EvtSTmp \\ & = & \{x \leftarrow v \mid x \in var\} \cup \{x \leftarrow v \mid x \in tvar\} \end{array}$$

Transition rules are given in Figures 4.9 and 4.10. The flow of execution goes from one instruction to the other as follows. Suppose the instruction list to execute is $\ell = i; \ell'$. The instruction $i = hd(\ell)$ of ℓ is first executed. Then, if the control flow does not jump, we use the function next defined as:

$$\mathsf{next}(pc, i; \ell') \quad = \quad \begin{cases} (pc + 1, m.\mathsf{code}[pc + 1]) & \text{if } \ell' = nil \\ (pc, \ell') & otherwise \end{cases}$$

As will be seen in the next section, the generated BIR instruction lists are never empty. Hence, the function next is well defined. Moreover, when the control flow jumps, the instruction list to execute is directly identified by the label of the jump target (e.g. rule for `goto`).

In the rule for object creation (Figure 4.9), note that the freshly created object is directly tagged as being initialized, by the constructor folding: as soon as the object has been allocated, its constructor is called; its status is hence updated. No instruction can be executed between the object allocation and its constructor call. In the middle-bottom rule of Figure 4.9, the super constructor is called non-virtually. The virtual method call is handled in the rule at the right-bottom of the figure. Semantic rules for assertions are also rather intuitive: either the assertion passes, and the execution goes on, or it fails and the execution of the program is aborted in the corresponding error state. Concerning error handling, notice that the BIR semantics suggests more blocking states than BC. For instance, no semantic rule can be applied when trying to execute a method call on a null pointer. Here, we do not need to take into account this case: the transformation algorithm generates an assertion when translating method call instruction, which will catch the null pointer dereferencing attempt. Apart from these points, rules of BIR use the same principles as BC rules and we do not further comment them.

$$\frac{hd(\ell) = \texttt{nop}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, m, \mathsf{next}(pc, \ell), l\rangle}$$

$$\frac{hd(\ell) = x{:=}exp \quad h, l \vDash exp \Downarrow v}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[x \leftarrow v]}_0 \langle h, m, \mathsf{next}(pc, \ell), l[x \mapsto v]\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = exp.f{:=}exp' \\ h, l \vDash exp \Downarrow (\mathsf{R}\ r) \quad h, l \vDash exp' \Downarrow v \\ h(r) = o_C \quad o' = o[f \mapsto v] \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[r.f \leftarrow v]}_0 \langle h[r \mapsto o'], m, \mathsf{next}(pc, \ell), l\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{if}\ \ exp\ pc' \\ h, l \vDash exp \Downarrow (\mathsf{N}\ 0) \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, m, (pc', m.\mathsf{code}[pc']), l\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{if}\ \ exp\ pc' \\ h, l \vDash exp \Downarrow (\mathsf{N}\ n) \quad n \neq 0 \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, m, \mathsf{next}(pc, \ell)), l\rangle}$$

$$\frac{hd(\ell) = \texttt{goto}\ \ pc'}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, m, (pc', m.\mathsf{code}[pc']), l\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{vreturn}\ exp \\ h, l \vDash exp \Downarrow v \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[\mathsf{ret}(v)]}_0 \langle h, v\rangle}$$

$$\frac{hd(\ell) = \texttt{return}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[\mathsf{ret}(\mathsf{Void})]}_0 \langle h, \mathsf{Void}\rangle}$$

$$\frac{hd(\ell) = \texttt{mayinit}\ C}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{[\mathsf{mayinit}(C)]}_0 \langle h, \mathsf{next}(pc, \ell), l\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{notnull}\ exp \\ h, l \vDash exp \Downarrow (\mathsf{R}\ r) \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, \mathsf{next}(pc, \ell), l\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{notnull}\ exp \\ h, l \vDash exp \Downarrow \mathsf{Null} \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, m, pc, l\rangle_{\mathsf{NP}}}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{notzero}\ exp \\ h, l \vDash exp \Downarrow (\mathsf{N}\ n) \quad n \neq 0 \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, \mathsf{next}(pc, \ell), l\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = \texttt{notzero}\ exp \\ h, l \vDash exp \Downarrow (\mathsf{N}\ 0) \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\tau}_0 \langle h, m, pc, l\rangle_{\mathsf{DZ}}}$$

$$\frac{\begin{array}{c} hd(\ell) = x{:=}\texttt{new}\ C(e_1, \ldots, e_n) \\ h, l \vDash e_i \Downarrow v_i \quad (h', (\mathsf{R}\ r)) = \mathsf{newObject}(C, h) \\ h'(r) = \mathsf{zeros}(C)_C \\ \langle h', \mathsf{init_state}(C.\mathsf{init})\rangle \overset{\Lambda}{\Rightarrow}_n \langle h'', \mathsf{Void}\rangle \\ \Lambda' = [r \leftarrow C.\mathsf{init}(v_1, \ldots, v_n)].\Lambda_H.[x \leftarrow (\mathsf{R}\ r)] \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\Lambda'}_{n+1} \langle h'', (m, \mathsf{next}(pc, \ell), l[x \mapsto (\mathsf{R}\ r)]\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = e.\texttt{super}(C, e_1, \ldots, e_n) \\ h, l \vDash e \Downarrow (\mathsf{R}\ r) \quad h, l \vDash e_i \Downarrow v_i \\ h(r) = o_{C'} \quad C' \subset C \\ \langle h, \mathsf{init_state}(C.\mathsf{init})\rangle \overset{\Lambda}{\Rightarrow}_n \langle h', \mathsf{Void}\rangle \\ \Lambda' = [r.C.\mathsf{init}(v_1, \ldots, v_n)].\Lambda_H \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\Lambda'}_{n+1} \langle h', m, \mathsf{next}(pc, \ell), l\rangle}$$

$$\frac{\begin{array}{c} hd(\ell) = y{:=}e.m'(C, e_1, \ldots, e_n) \\ h, l \vDash e_i \Downarrow v_i \quad h, l \vDash e \Downarrow (\mathsf{R}\ r) \\ h(r) = o_{C'} \qquad \mathsf{lookup}(m', C') = mc \\ \langle h, \mathsf{init_state}(mc)\rangle \overset{\Lambda}{\Rightarrow}_n \langle h', v\rangle \\ l' = l[y \mapsto v] \\ \Lambda' = [r.C.mc(v_1, \ldots, v_n)].\Lambda_H.[y \leftarrow v] \end{array}}{\langle h, m, (pc, \ell), l\rangle \xrightarrow{\Lambda'}_{n+1} \langle h', m, \mathsf{next}(pc, \ell), l'\rangle}$$

$$\forall m \in \mathbb{M}, \mathsf{init_state}(m) = (m, (0, m.\mathsf{code}[0]), [\texttt{this} \mapsto (\mathsf{R}\ r), x_1 \mapsto v_1 \ldots x_n \mapsto v_n]), \text{ where } m.\mathsf{params} = x_1{::}\ldots{::}x_n$$

Figure 4.9: BIR operational semantics

$$\frac{\begin{array}{c} hd(\ell) = x{:=}\texttt{new}\ C(e_1,\ldots,e_n) \\ h,l \vDash e_i \Downarrow v_i \quad (h',(\mathsf{R}\ r)) = \mathsf{newObject}(C,h) \\ h'(r) = \mathsf{zeros}(C)_C \\ \langle h', \mathsf{init_state}(C.\mathsf{init})\rangle \overset{\Lambda}{\Rightarrow}_n \langle h_e, C.\mathsf{init}, pc', l_e\rangle_k \end{array}}{\langle h,m,(pc,\ell),l\rangle \xrightarrow{[r\leftarrow C.\mathsf{init}(v_1,\ldots,v_n)].\Lambda_H}_{n+1} \langle h_e,m,pc,l\rangle_k}$$

$$\frac{\begin{array}{c} hd(\ell) = e.\texttt{super}(C,e_1,\ldots,e_n) \\ h,l \vDash e \Downarrow (\mathsf{R}\ r) \quad h,l \vDash e_i \Downarrow v_i \\ h(r) = o_{C'} \quad C' \subset C \\ \langle h, \mathsf{init_state}(C.\mathsf{init})\rangle \overset{\Lambda}{\Rightarrow}_n \langle h_e, C.\mathsf{init}, pc', l_e\rangle_k \end{array}}{\langle h,m,(pc,\ell),l\rangle \xrightarrow{[r.C.\mathsf{init}(v_1,\ldots,v_n)].\Lambda_H}_{n+1} \langle h_e,m,pc,l\rangle_k}$$

$$\frac{\begin{array}{c} hd(\ell) = y{:=}e.m'(C,e_1,\ldots,e_n) \\ h,l \vDash e_i \Downarrow v_i \quad h,l \vDash e \Downarrow (\mathsf{R}\ r) \\ h(r) = o_{C'} \qquad \mathsf{lookup}(m',C') = mc \\ \langle h, \mathsf{init_state}(mc)\rangle \overset{\Lambda}{\Rightarrow}_n \langle h_e, mc, pc', l_e\rangle_k \end{array}}{\langle h,m,(pc,\ell),l\rangle \xrightarrow{[r.C.mc(v_1,\ldots,v_n)].\Lambda_H}_{n+1} \langle h_e,m,pc,l\rangle_k}$$

$$\frac{\begin{array}{c} hd(\ell) = e.m'(C,e_1,\ldots,e_n) \\ h,l \vDash e_i \Downarrow v_i \quad h,l \vDash e \Downarrow (\mathsf{R}\ r) \\ h(r) = o_{C'} \qquad \mathsf{lookup}(m',C') = mc \\ \langle h, \mathsf{init_state}(mc)\rangle \overset{\Lambda}{\Rightarrow}_n \langle h_e, mc, pc', l_e\rangle_k \end{array}}{\langle h,m,(pc,\ell),l\rangle \xrightarrow{[r.C.mc(v_1,\ldots,v_n)].\Lambda_H}_{n+1} \langle h_e,m,pc,l\rangle_k}$$

Figure 4.10: BIR operational semantics: error cases

4.3 The BC2BIR algorithm

In this section we describe the BC2BIR transformation algorithm for converting BC code into BIR code. A central feature of our algorithm is the use of a symbolic stack to decompile stack-oriented code into three-address code. We explain how the symbolic stack is used in decompiling BC instructions and how it is managed at control flow join points. Another distinguishing feature of the algorithm is the merging of instructions for object allocation and initialization into one compound BIR instruction which is also performed quite elegantly thanks to the symbolic stack. The transformation is structured in 3 layers. Section 4.3.1 describes the transformation of each BC instruction separately. Section 4.3.2 describes the transformation for a whole BC method body, and a whole BC program.

4.3.1 Transforming instructions

The core of the algorithm is the function $\mathsf{BC2BIR}_i$. It maps a BC instruction into a list of BIR instructions and at the same time symbolically executes BC code using an abstract stack of symbolic expressions:

$$\begin{array}{c} \mathsf{BC2BIR}_i : \mathbb{N} \times \mathit{instr}_{\mathsf{BC}} \times \mathit{AbstrStack} \rightarrow (\mathit{instr}^*_{\mathsf{BIR}} \times \mathit{AbstrStack}) \cup \mathit{Fail} \\ \mathit{AbstrStack} = \mathit{SymbExpr}^* \qquad \mathit{SymbExpr} = \mathit{exp} \cup \{C_{pc} \mid C \in \mathbb{C}, pc \in \mathbb{N}\} \end{array}$$

Expressions in *exp* are BC decompiled expressions and C_{pc} is a placeholder for a reference to an uninitialized object, allocated at point *pc* by the intruction `new` C. Figure 4.11 defines the result of $\mathsf{BC2BIR}_i(pc, \mathit{instr}, \mathit{as})$. All t^i_{pc} denote fresh temporary variables introduced at point *pc*. A paragraph at the end of this section describes the failure cases.

Basic instructions For instruction `load` x , the symbolic expression x is pushed on the abstract stack *as* and the BIR instruction `nop` is generated. We generate `nop` to make the step-matching easier in the proof of the theorem[5]. Other basic bytecodes, such as `pop`, `push`, return statements or jump instructions follows the same principle. These correspond to the

[5]All `nop`s could be removed without changing the semantics of the BIR program.

Inputs		Outputs	
Instr	Stack	Instrs	Stack
`nop`	as	[`nop`]	as
`pop`	$e{::}as$	[`nop`]	as
`push` c	as	[`nop`]	$c{::}as$
`dup`	$e{::}as$	[`nop`]	$e{::}e{::}as$
`load` x	as	[`nop`]	$x{::}as$
`if` pc'	$e{::}as$	[`if` e pc']	as
`goto` pc'	as	[`goto` pc']	as

Inputs		Outputs	
Instr	Stack	Instrs	Stack
`return`	as	[`return`]	as
`vreturn`	$e{::}as$	[`return` e]	as
`add`	$e_1{::}e_2{::}as$	[`nop`]	$e_1 + e_2{::}as$
`div`	$e_1{::}e_2{::}as$	[`notzero` e_2]	$e_1/e_2{::}as$
`new` C	as	[`mayinit` C]	$C_{pc}{::}as$
`getfield` f	$e{::}as$	[`notnull` e]	$e.f{::}as$

Inputs		Outputs		Cond
Instr	Stack	Instrs	Stack	
`store` x	$e{::}as$	$[x := e]$	as	$x \notin as$ [a]
		$[t^0_{pc} := x; x := e]$	$as[t^0_{pc}/x]$	$x \in as$ [a]
`putfield` f	$e'{::}e{::}as$	[`notnull` e; $Fsave(pc, f, as)$; $e.f := e'$]	$as[\mathtt{t}^{\mathtt{i}}_{\mathtt{pc}}/e_i]$	[a] [b]
`invokevirtual` $C.m$	$e'_1 \ldots e'_n{::}e{::}as$	[`notnull` e; $Hsave(pc, as)$; $t^0_{pc} := e.m(e'_1 \ldots e'_n)$]	$t^0_{pc}{::}as[t^j_{pc}/e_j]$	value return [a] [c]
		[`notnull` e; $Hsave(pc, as)$; $e.m(e'_1 \ldots e'_n)$]	$as[t^j_{pc}/e_j]$	Void return[ac]
`constructor` C	$e'_1 \ldots e'_n{::}e_0{::}as$	[$Hsave(pc, as)$; $t^0_{pc} :=$ `new` $C(e'_1 \ldots e'_n)$]	$as[t^j_{pc}/e_j]$	$e_0 = C_{pc'}$ [c]
		[`notnull` e; $Hsave(pc, as)$; $e.$`super`$(C, e'_1 \ldots e'_n)$]	$as[t^j_{pc}/e_j]$	otherwise [a] [c]

Figure 4.11: $\mathsf{BC2BIR}_i(pc, instr, as)$ – Transformation of a BC instruction *instr* at program point pc, given the input abstract stack as. All t^i_{pc} denote fresh temporary variables introduced at this point. For each table, the left column gives the input instruction *instr* and abstract stack as, and the right column gives the list of BIR instructions and output abstract stack. In the bottom table, the last column Cond indicates the conditions where the case applies.

[a] where for all C and pc', $e \neq C_{pc'}$
[b] where $e_i, i = 1 \ldots n$ are all the elements of as such that $f \in e_i$
[c] where $e_j, j = 1 \ldots m$ are all the elements of as that read a field

two top tables in Figure 4.11, except for `new` C and `getfield` instruction, that we describe next.

Example 4.1. *Before going into more technicality, we give a simple example of symbolic execution. Successively symbolically executing* `load` x *and* `load` y *will lead to the abstract stack* $y{::}x{::}\varepsilon$. *Transforming an* `add` *instruction would change the abstract stack for* $(x + y){::}\varepsilon$.

Memory instructions Transforming instructions `store`, `putfield` and `invokevirtual` follows the same principle. However, we must take care of their memory effect, since their execution might modify the value of local variables or object fields appearing in the expressions of the abstract stack, whose value would be erroneously modified by side effect. We tackle this subtlety by storing in temporary variables (of the form t^i_{pc}) each stack element whose value might be modified. In the case of `store` x, it is enough only remembering the old value of x. In the case of `putfield` f, all expressions in as accessing a field f are remembered: $Fsave(pc, f, e_1{::}e_2{::}\ldots{::}e_n)$ generates an assignment $t^i_{pc} := e_i$ for all e_i that reads at least once the field f. In the case of `invokevirtual`, we store the value of each expression accessing the heap, which could be modified by the callee execution: $Hsave(pc, e_1{::}e_2{::}\ldots{::}e_n)$ generates an assignment $t^i_{pc} := e_i$ for all e_i that reads a field.

Object creation Object creation and initialization require special attention as this is done by separate (and possibly distant) instructions. Symbolically executing `new` C at point pc

```
1  function BC2BIR(m) =
2  AS_in[m,0] := nil
3  jmp := get_joinpoint(m)
4
5  for (pc = 0, pc ≤ length(m), pc++) do
6     succs := get_succs(m,pc)
7
8    // Compute entry abstract stack
9    if (pc ∈ jmp) then
10     assert C(pc)
11     AS_in[m,pc] := Normalize(pc, AS_out[m])
12
13   // Decompile instruction
14   IR[m,pc],AS_out[m,pc] := BC2BIR_i(pc, m.code[pc], AS_in[m,pc])
15   IR[m,pc] := TAssign(succs ∩ jmp, AS_out[m,pc])++IR[m,pc]
16
17   // Fail on a non-empty stack backward jump
18   assert(AS_out[m,pc] = nil ∨ ∀pc' ∈ succs. pc < pc')
19
20   // Propagate output abstract stack
21   if (pc+1∈ succs∧ pc+1∉jmp) then AS_in[m,pc+1] := AS_out[m,pc]
22 end
```

Figure 4.12: BC2BIR – BC method code transformation

pushes C_{pc} (representing the freshly allocated reference) on the stack and generates `mayinit` C for class initialization whenever it is required. Instruction `constructor` C will be transformed differently whether it corresponds to a constructor or a super constructor call. Both cases are distinguished thanks to the symbolic expression on which it is called. We generate a BIR folded constructor call at point pc if the symbolic expression is $C_{pc'}$ (and a super constructor call otherwise). C_{pc} are used to keep track of alias information between uninitialized references, when substituting them for the local variable receiving the new object. A similar mechanism is used by the BCV to check for object initialization.

4.3.2 Transforming method code

Transforming the whole code of a BC method is done by BC2BIR. It consists in traversing the whole method body, without iterating. In a nutshell, for each program point, it (i) first computes the entry abstract stack used by $\mathsf{BC2BIR}_i$ to transform the instruction at this point (ii) then performs the BIR generation and (iii) passes on the output abstract stack to the successor points.

BC2BIR is given in Figure 4.12. It computes three arrays: $\mathtt{IR}[m]$ is the BIR version of the method m, $\mathtt{AS_{in}}[m]$ and $\mathtt{AS_{out}}[m]$ respectively contain the input and output symbolic stacks used by $\mathsf{BC2BIR}_i$.[6] The algorithm starts by initializing the input (empty) abstract stack for the entry point of the program, and computing the set of program junction points (Line 3). Then starts at Line 5 the traversal of the method array of instructions, from the entry point to the last point of the method (given by $\mathsf{length}(\mathtt{m})$).

When the control flow is linear (from pc to only $pc+1$), we only perform the $\mathsf{BC2BIR}_i$ generation (Line 14) and the abstract stack resulting from $\mathsf{BC2BIR}_i$ is transmitted as is (Line 21). The case of control flow joins must be handled more carefully. The rest of this section explains how those are managed.

[6] We keep track of the method name mainly for simplifying the notations in Section 4.4.

```
int f(int x) { return (x==0) ? 1 : -1; }
```

(a) Source function

```
int f(x) {
   0: load x
   1: if 4
   2: push -1
   3: goto 5

   4: push 1

   5: vreturn
}
```

(b) BC function

```
0: []
1: [x]
2: []
3: [-1]

4: []

5: [T^1_5]
```

(c) Symbolic input stack

```
0: [x]
1: []
2: [-1]
3: [-1]

4: [1]

5: []
```

(d) Symbolic output stack

```
int f(x) {
   0: nop;
   1: if x 4;
   2: nop;
   3: T^1_5 := -1;
      goto  5;
   4: nop;
      T^1_5 := 1;
   5: vreturn T^1_5;
}
```

(e) BIR function

Figure 4.13: BC transformation – Non-empty stack forward jumps and stack normalization.

Stack normalization In a program passing the BCV, at every join point, the size of the stack is the same regardless of the predecessor point. Still, the content of the abstract stack might change (when e.g. two branches of a conditional compute two different expressions). But stack elements are expressions used in the generated instructions and must not depend on the control flow path.

Example 4.2 (Forward jumps on non-empty stack)**.** *The function in Figure 4.13 returns 1 or -1, depending on whether the argument x is zero or not. We focus on program point* 5*, a join point with predecessors* 3 *and* 4*. The abstract stack after the instruction* `goto` 5 *contains -1 (point* 3 *in Figure 4.13c), while it contains* 1 *after program point* 4*. Two different output abstract stacks must thus be merged into an entry abstract at point* 5*.*

The idea is here to store, before reaching a join point, every stack element in a temporary variable and to use, at the join point, a *normalized* stack made of all these variables. A naming convention ensures that (i) identifiers are independent of the control flow and (ii) an identifier denotes a stack element at a given position. We thus use the identifier T^i_{pc} to store the ith element of the stack for a join point at pc. All T^i_{pc} are initialized when transforming a BC instruction preceeding the join point. In Figure 4.13e, at points 3 and 4, we respectively store -1 and 1 in T^1_5, the top element of the entry stack at point 5.

In the algorithm, this is done at Line 15: we prepend to the code generated by $\mathsf{BC2BIR}_i$ the assignments of all abstract stack elements to the T^i_{jp}, for all join points jp successor of pc. These assignments are generated by $\mathsf{TAssign}(S, as)$, where S is the set of program junction points that are successors of pc. For each of them, we must generate these assignments. The restriction Line 18 ensures these assignments are conflict-free by making the transformation fail on non-empty stack backjumps. The function $\mathsf{Normalize}(jp, as)$ (Line 11) computes a stack of length $n = \mathsf{length}(as)$ normalized with regard to jp (the role of the argument as is explained below):

$$\mathsf{Normalize}(jp, as) = [T^1_{jp} :: \cdots :: T^n_{jp}]$$

Forward jumps and uninitialized references If the stack to normalize contains an uninitialized reference, we cannot store it in a temporary — this element is not a BIR expression,

the assignment would not be a legal BIR. Hence, the function Normalize preserves all C_{pc}. Hence the need of keeping the output abstract stack as parameter. But we need here the following constraint $C(jp)$ on $\mathtt{AS}_{\mathtt{out}}$, that we check before computing the entry abstract stack (Line 10), meaning that before a join point jp, if the stack contains any C_k at position i, then it is the case for all predecessors of jp

$$\forall i. \quad \big(\exists pc' \in pred_m(jp).\, \mathtt{AS}_{\mathtt{out}}[m, pc']_i = C_k\big) \Rightarrow \quad \big(\forall pc' \in pred_m(jp).\, \mathtt{AS}_{\mathtt{out}}[m, pc']_i = C_k\big)$$

Finally, a whole BC program P is translated to BIR by mapping the method transformation BC2BIR to all methods of all classes of P. In the sequel, we write it BC2BIR$(P) = P'$. In the following section, we make further remarks on the algorithm and formalize its semantics preservation property.

Relative BCV-completeness Every case undescribed in Figure 4.11 yields *Fail*. Most of them are ruled out by the BCV (e.g. stack height mismatch, or uninitialised reference field assignment) but few cases remain. First, we fail on backjumps with non-empty stacks. Second, transforming `store` x requires that the expression on top of the stack is not C_{pc} because no valid BIR instruction would match, as constructors are folded. Third, we fail to transform bytecode that does not satisfy C: this constraint prevents us from storing C_{pc} stack elements. Unfortunately these three cases are not ruled out by the JVM specification and we may reject programs that pass the BCV[7]. However this is not a limitation in practice, because such patterns do not seem to be used by Java standard compilers. Since the beginning of the Sawja library, our transformation tool has been run on a set of large benchmarks library (including the one presented in Section 4.5) without encountering such cases.

4.4 Semantic correctness of BC2BIR

The BC2BIR algorithm satisfies a precise semantics preservation property that we formalize in this section: the BIR program BC2BIR(P) simulates the initial BC program P and both have similar execution traces. This similarity cannot be a simple equality, because some variables have been introduced by the transformation and the object allocation order is modified by BC2BIR— both heaps do not keep equal along both program executions. We define in Section 4.4.1 what semantic relations make us able to precisely relate BC and BIR executions. Section 4.4.2 formally states the semantic preservation of BC2BIR. We then provide an extended proof sketch of the theorem, focusing on the interesting and important part of the proof; presenting the complete proof would not bring much to the understanding of the correctness result. The most important lemmas are stated and proved in the present section. We refer the reader to [DJP09] for a complete proof, whose additional lemmas are only mechanical extensions of the results presented in this document. We lighten the notations from now and until the end of this section by considering a BC program P, its BIR version $P' =$ BC2BIR(P).

[7]The specification of the BCV only constrains uninitialized references on backward jumps (see [LY99]), whereas we need also the constraint on forward jumps.

4.4.1 Semantic relations

Heap isomorphism The transformation does not preserve the object allocation order. However, the two heaps stay isomorphic: there exists a partial bijection[8] between them. For example, in P (Figure 4.1c), the `B` object is allocated before the `A` object is passed as an argument to the `B` constructor. In P' (Figure 4.1d), constructors are folded and object creation is not an expression, the `A` object must thus be created (and initialized) before passing $\mathtt{t_1}$ (containing its reference) as an argument to the `B` constructor.

Heaps are not equal along the execution of the two programs: after program point `5` in P, the heap contains two objects that are not yet in the heap of P'. However, after program point `7`, each use in P' of the `A` object corresponds to a use in P of the reference pointing to the `A` object (both objects are initialized, so both references can be used). The same reasoning can be applied just after point 8 about the `B` objects. A bijection thus exists between references of both heaps. It relates references to allocated objects as soon as their initialization has begun. Along the executions of BC and BIR programs, it is extended accordingly on each constructor call starting the initialization of a new object. In Figure 4.1, given an initial partial bijection on the heaps domains, it is first extended at point 7 and then again at point 8.

Semantic relations This heap isomorphism has to be taken into account when relating semantic domains and program executions. Thus, the semantic relations over values, heaps, environments, configurations and observable events (see Table 4.1) are parametrized by a bijection β defined on the heap domains.

When relating values, the interesting case is for references. Only references related by β are in the relation. The semantic relation on heaps is as follows. First, objects related by β are exactly those existing in both heaps and on which a constructor has been called. Secondly, the related objects must have the same initialization status (hence the same class) and their fields must have related values. Here we write $tag_h(r)$ for the tag t such that $h(r) = o_t$. A BIR environment is related to a BC environment if and only if both local variables have related values. Temporary variables are, as expected, not taken into account. Execution states are related through their heaps and environments, the stack is not considered here. Program points are not related to a simple one-to-one relation: the whole block generated from a given BC instruction must be executed before falling back into the relation. Hence, a BC state is matched at the beginning of the BIR block of the same program point, given by $m.\mathsf{code}[pc]$. We only relate error states of the same kind of error. Finally, two observable events are related if they are of the same kind, and the values they involve are related. To relate execution traces, we pointwise extend $\overset{!}{\sim}_\beta$. We now assume that $\mathtt{IR}, \mathtt{AS_{in}}$ and $\mathtt{AS_{out}}$ are the code and abstract stack arrays computed by BC2BIR, and so until the end of the section.

4.4.2 Soundness result

The previously defined observational semantics and semantic relations allows achieving a very fine-grained correctness criterion for the transformation BC2BIR. It says that P' simulates the initial program P: starting from two related initial configurations, if the execution of P terminates in a given (normal or error) state, then P' terminates in a related state, and both

[8] The rigorous definition of a bijection demands that it is total. The term "partial bijection" is however widely used. We consider it as equivalent to "partial injection".

Relation	Definition
$v_1 \overset{\mathrm{V}}{\sim}_\beta v_2$ $v_1, v_2 \in \mathsf{Val}$	$\dfrac{}{\mathsf{Null} \overset{\mathrm{V}}{\sim}_\beta \mathsf{Null}}$ $\dfrac{n \in \mathbb{Z}}{(\mathsf{N}\ n) \overset{\mathrm{V}}{\sim}_\beta (\mathsf{N}\ n)}$ $\dfrac{\beta(r_1) = r_2}{(\mathsf{R}\ r_1) \overset{\mathrm{V}}{\sim}_\beta (\mathsf{R}\ r_2)}$
$h_1 \overset{\mathrm{H}}{\sim}_\beta h_2$ $h_1, h_2 \in \mathit{Heap}$	• $dom(\beta) = \{r \in dom(h_1) \mid \forall C, pc,\ tag_{h_1}(r) \neq \widetilde{C}_{pc}\}$ • $rng(\beta) = dom(h_2)$ • $\forall r \in dom(h_1)$, let $o_t = h_1(r)$ and $o'_{t'} = h_2(\beta(r))$ then $(i)\ t = t' \quad (ii)\ \forall f,\ o_t(f) \overset{\mathrm{V}}{\sim}_\beta o'_t(f)$
$l_1 \overset{\mathrm{E}}{\sim}_\beta l_2$ $(l_1, l_2) \in \mathit{Env}_{\mathsf{BC}} \times \mathit{Env}_{\mathsf{BIR}}$	$dom(l_1) = var_{\mathsf{BC}} \cap dom(l_2)$ and $\forall x \in dom(l_1),\ l_1(x) \overset{\mathrm{V}}{\sim}_\beta l_2(x)$
$c_1 \overset{\mathrm{C}}{\sim}_\beta c_2$ $(c_1, c_2) \in \mathit{State}_{\mathsf{BC}} \times \mathit{State}_{\mathsf{BIR}}$	$\dfrac{h \overset{\mathrm{H}}{\sim}_\beta ht \quad l \overset{\mathrm{E}}{\sim}_\beta lt}{\langle h, m, pc, l, s\rangle \overset{\mathrm{C}}{\sim}_\beta \langle ht, m, (pc, m.\mathsf{code}[pc]), lt\rangle}$ $\dfrac{h \overset{\mathrm{H}}{\sim}_\beta ht \quad rv \overset{\mathrm{V}}{\sim}_\beta rv'}{\langle h, rv\rangle \overset{\mathrm{C}}{\sim}_\beta \langle ht, rv'\rangle}$ $\dfrac{h \overset{\mathrm{H}}{\sim}_\beta ht \quad l \overset{\mathrm{E}}{\sim}_\beta lt}{\langle h, m, pc, l\rangle_k \overset{\mathrm{C}}{\sim}_\beta \langle ht, m, pc, lt\rangle_k}$
$\lambda_1 \overset{!}{\sim}_\beta \lambda_2$ with $\lambda_1, \lambda_2 \in \mathit{Evt}$	$\dfrac{}{\mathsf{ret}(\mathsf{Void}) \overset{!}{\sim}_\beta \mathsf{ret}(\mathsf{Void})}$ $\dfrac{v_1 \overset{\mathrm{V}}{\sim}_\beta v_2}{\mathsf{ret}(v_1) \overset{!}{\sim}_\beta \mathsf{ret}(v_2)}$ $\dfrac{}{\mathsf{mayinit}(C) \overset{!}{\sim}_\beta \mathsf{mayinit}(C)}$ $\dfrac{x \in var_{\mathsf{BC}} \quad v_1 \overset{\mathrm{V}}{\sim}_\beta v_2}{x \leftarrow v_1 \overset{!}{\sim}_\beta x \leftarrow v_2}$ $\dfrac{\beta(r_1) = r_2 \quad v_1 \overset{\mathrm{V}}{\sim}_\beta v_2}{r_1.f \leftarrow v_1 \overset{!}{\sim}_\beta r_2.f \leftarrow v_2}$ $\dfrac{\beta(r_1) = r_2 \quad \forall i = 1 \ldots n, v_i \overset{\mathrm{V}}{\sim}_\beta v'_i}{r_1.C.m(v_1, \ldots, v_n) \overset{!}{\sim}_\beta r_2.C.m(v'_1, \ldots, v'_n)}$ $\dfrac{\beta(r_1) = r_2 \quad \forall i = 1 \ldots n, v_i \overset{\mathrm{V}}{\sim}_\beta v'_i}{r_1 \leftarrow C.\mathsf{init}(v_1, \ldots, v_n) \overset{!}{\sim}_\beta r_2 \leftarrow C.\mathsf{init}(v'_1, \ldots, v'_n)}$ $\dfrac{\beta(r_1) = r_2 \quad \forall i = 1 \ldots n, v_i \overset{\mathrm{V}}{\sim}_\beta v'_i}{r_1.C.\mathsf{init}(v_1, \ldots, v_n) \overset{!}{\sim}_\beta r_2.C.\mathsf{init}(v'_1, \ldots, v'_n)}$

Table 4.1: Semantic relations for values, heaps, environments, configurations and events

execution traces are related, when forgetting temporary variables assignments in the BIR trace (we write Λ_{proj} for such a projection of Λ). More formally:

Theorem 4.1 (Semantic preservation)**.** *Let $m \in \mathbb{M}$ be a method of P (and P') and $n \in \mathbb{N}$. Let $c = \langle h, m, 0, l, \varepsilon\rangle \in \mathit{State}_{\mathsf{BC}}$ and $ct = \langle h, m, (0, m.\mathsf{code}[0]), l\rangle \in \mathit{State}_{\mathsf{BIR}}$. Then:*

Normal return *If $c \overset{\Lambda}{\Rightarrow}_n \langle h', v\rangle$ then there exist unique ht', v', Λ' and β such that $ct \overset{\Lambda'}{\Rightarrow}_n \langle ht', v'\rangle$ with $\langle h', v\rangle \overset{\mathrm{C}}{\sim}_\beta \langle ht', v'\rangle$ and $\Lambda \overset{!}{\sim}_\beta \Lambda'_{proj}$.*

Error *If $c \overset{\Lambda}{\Rightarrow}_n \langle h', m, pc', l'\rangle_k$ then there exist unique ht', lt', Λ' and β such that $ct \overset{\Lambda'}{\Rightarrow}_n \langle ht', m, pc', lt'\rangle_k$ with $\langle h', m, pc', l'\rangle_k \overset{\mathrm{C}}{\sim}_\beta \langle ht', m, pc', lt'\rangle_k$ and $\Lambda \overset{!}{\sim}_\beta \Lambda'_{proj}$.*

Discussion Theorem 4.1 does not deal with executions that get stuck, but this is not required, as the corresponding program would not pass the BCV. It also only partially deals with infinite computations: we e.g. do not show the preservation of executions when they diverge

inside a method call. All reachable states (intra and inter-procedurally) could be matched giving small-step operational semantics to both languages. But this would require parametrizing events by the method from which they arise, and extending the relation on configurations to all frames in the call stack.

We prove this theorem using a strong induction on the call depth n. For the inductive reasoning to be possible, we need to consider intermediate computation states, and provide a simulation scheme similar to what has been presented in Section 3.2.2. The crucial point is that BC intermediate states require dealing with the stack, to which BIR expressions must be related. Semantically, this is captured by a correctness criterion on the abstract stack used by the transformation. It intuitively means that expressions are correctly decompiled:

Definition 4.1 (Symbolic stack correctness $h, ht, lt, \beta \models . \approx .$)**.** *Given $h, ht \in Heap$ such that $h \overset{\mathrm{H}}{\sim}_\beta ht$ and $lt \in Env_{\mathsf{BIR}}$, an abstract stack $as \in AbstrStack$ is said to be correct with regard to a run-time stack $s \in Stack$ if and only if $h, ht, lt, \beta \models as \approx s$:*

$$\frac{}{h, ht, lt, \beta \models \varepsilon \approx \varepsilon} \qquad \frac{\begin{array}{c} ht, lt \vDash e \Downarrow v' \quad v \overset{\mathrm{V}}{\sim}_\beta v' \\ h, ht, lt, \beta \models s \approx as \end{array}}{h, ht, lt, \beta \models v{::}s \approx e{::}as} \qquad \frac{\begin{array}{c} tag_h(r) = \widetilde{C}_{pc} \\ \forall r', (\mathsf{R}\ r') \in s \wedge tag_h(r') = \widetilde{C}_{pc} \Rightarrow r = r' \\ h, ht, lt, \beta \models s \approx as \end{array}}{h, ht, lt, \beta \models (\mathsf{R}\ r){::}s \approx C_{pc}{::}as}$$

The last definition rule says that the symbol C_{pc} correctly approximates a reference r of tag $\widetilde{C}_{pc}$. The alias information tracked by C_{pc} is made consistent if we additionally demand that all references appearing in the stack with the same status tag are equal to r (second condition of this last rule). This strong property is enforced by the restrictions imposed by the BCV on uninitialized references in the operand stack.

We are now able to state the general proposition on intermediate execution states. In order to clarify the induction hypothesis, we parametrize the proposition by the call depth and the name of the executed method:

Lemma 4.2 ($\mathcal{P}(n, m)$ – BC2BIR n call-depth preservation)**.**
Let $m \in \mathbb{M}$ be a method of P (and P') and $n \in \mathbb{N}$. Let β be a partial bijection on Ref*. Let $c = \langle h, m, pc, l, s\rangle \in State_{\mathsf{BC}}$ and $ct = \langle ht, m, (pc, m.\mathsf{code}[pc]), lt\rangle \in State_{\mathsf{BIR}}$ such that*

$$c \overset{\mathrm{C}}{\sim}_\beta ct \text{ and } h, ht, lt, \beta \models s \approx \mathtt{AS_{in}}[m, pc]$$

Then, for all $c' = \langle h', m, pc', l', s'\rangle \in State_{\mathsf{BC}}$, whenever $c \overset{\Lambda}{\Rightarrow}_n c'$, there exist a unique $ct' = \langle ht', m, (pc', m.\mathsf{code}[pc']), lt'\rangle$ and Λ' and a unique β' extending β such that $ct \overset{\Lambda'}{\Rightarrow}_n ct'$ with

$$c' \overset{\mathrm{C}}{\sim}_{\beta'} ct' \ , \ \Lambda \overset{!}{\sim}_{\beta'} \Lambda'_{proj} \text{ and } h', ht', lt', \beta' \models s' \approx \mathtt{AS_{in}}[m, pc']$$

where $\mathtt{AS_{in}}$ denotes the symbolic stack used by the instruction-wise transformation $\mathsf{BC2BIR}_i$ (Line 14 in Figure 4.12). We prove Lemma 4.2 using a strong induction on n.

Base case For proving $\mathcal{P}(0, m)$, we reason by induction on the number of BC steps in the multistep transition $\dot{\Rightarrow}_0$. We show here the base case, the inductive case follows easily. A step

$$\langle h, m, pc, l, s\rangle \xrightarrow{\Lambda}_0 \langle h', m, pc', l', s'\rangle$$

is matched by a BIR computation of the form:

$$\langle ht, m, (pc, m.\mathsf{code}[pc]), lt\rangle \overset{\Lambda_1}{\Longrightarrow}_0 \langle ht, m, (pc, code), lt_0\rangle \overset{\Lambda_2}{\Longrightarrow}_0 \langle ht', m, (pc', m.\mathsf{code}[pc']), lt'\rangle$$

where the intermediate state $\langle ht, m, (pc, code), lt_0\rangle$ is obtained by executing the potential additional assignments $\mathsf{TAssign}(S, as)$, where S is the set of all the successors of pc that are join points, and as is the symbolic stack output by $\mathsf{BC2BIR}_i$ after transforming the BC instruction at pc. Indeed, these are prepended to the instructions *code* generated by $\mathsf{BC2BIR}_i$.

We prove that the abstract stack correctness hypothesis is preserved in the new environments $h, ht, lt_0, \beta \models s \approx \mathtt{AS_{in}}[m, pc]$. We distinguish two cases, depending on whether the input stack $\mathtt{AS_{in}}[m, pc]$ has been normalized (Line 11) or not (Line 21).

- If pc is a join point, then the stack is normalized. The additional assignments generated by TAssign do not alterate the stack correctness: if j is a join point successing pc, all T_j^k are assigned. If $j \neq pc$, we trivially have $lt(T_{pc}^k) = lt_0(T_{pc}^k)$. Otherwise, the instruction at pc can only be a `goto` pc. Thus, the assignment is the identity assignment, and $lt(T_{pc}^k) = lt_0(T_{pc}^k)$.
- Otherwise the stack is not normalized. One could fear that the TAssign assignments could break the correctness of the $T_{j'}^k$ that appear in the stack. However, as we forbid backwards jumps on non-empty stacks, all T_j^k (where j is a joint point successor of pc) assigned by TAssign cannot be used in the stack. This holds for two reasons. First, $T_{j'}^k$ cannot be pushed on the stack by BC instructions, but only by the normalization at point pc'. So, if such a temporary variable appears in the input abstract stack, it means that there is a path going from pc' to pc. But pc' is a successor of pc. It means there is a cycle without the stack having been emptied.

Hence $h, ht, lt_0, \beta \models s \approx \mathtt{AS_{in}}[m, pc]$. We obtain the second part of the matching computation thanks to a correctness lemma about $\mathsf{BC2BIR}_i$:

Lemma 4.3 ($\mathsf{BC2BIR}_i$ 0 call-depth one-step preservation)**.**
Suppose $\langle h, m, pc, l, s\rangle \xrightarrow{\Lambda}_0 \langle h', m, pc', l', s'\rangle$. *Let* ht, lt, as, β *be such that*

$$h \overset{\mathrm{H}}{\sim}_\beta ht \quad l \overset{\mathrm{E}}{\sim}_\beta lt \quad h, ht, lt, \beta \models s \approx as \text{ and } \mathsf{BC2BIR}_i(pc, m.\mathsf{code}[pc], as) = (code, as')$$

There exist unique ht', lt' *and* Λ' *s.t* $\langle ht, m, (pc, code), lt\rangle \overset{\Lambda'}{\Longrightarrow}_0 \langle ht', m, (pc', m.\mathsf{code}[pc']), lt'\rangle$

$$\text{with } h' \overset{\mathrm{H}}{\sim}_\beta ht' \quad l' \overset{\mathrm{E}}{\sim}_\beta lt' \quad \Lambda \overset{!}{\sim}_\beta \Lambda'_{proj} \text{ and } h', ht', lt', \beta \models s' \approx as'$$

It is similar to $\mathcal{P}(n, m)$, but only deals with one-step BC transitions and does not require extending the bijection (instructions at a zero call depth do not initialize any object). Moreover, considering an *arbitrary* correct entry abstract stack allows us to apply the lemma with more modularity. The proof of this lemma is given in Appendix 8.1. It is conducted by a long but rather mechanical case analysis on the current BC instruction.

Applying Lemma 4.3 gives us that $h' \overset{\text{H}}{\sim}_\beta ht'$, $l' \overset{\text{E}}{\sim}_\beta lt'$ and $\Lambda \overset{!}{\sim}_\beta \Lambda_{2proj}$. Furthermore, Λ_1 is only made of temporary variable assignment events, hence Λ_{1proj} is empty, and $\Lambda \overset{!}{\sim}_\beta \left(\Lambda_1.\Lambda_2\right)_{proj}$.

It remains to show the correctness of the transmitted abstract stack, i.e. that $h', ht', lt', \beta \models s' \approx \texttt{AS}_{\texttt{in}}[m, pc']$. There are two cases.

- If pc' is not a join point, then the transmitted abstract stack is either $\texttt{AS}_{\texttt{out}}[m, pc]$ resulting from $\mathsf{BC2BIR}_i$, in which case the conclusion of Lemma 4.3 is enough.
- Now, if pc' is a join point in m, the abstract stack is $\mathsf{Normalize}(pc', as)$. All of the $T^j_{pc'}$ have been assigned, but we must show that they have not been modified by executing the BIR instructions *code*. In Figure 4.11, the only assigned temporary variables are of the form $t^k_{pc'}$, or original bytecode variables.

Thus, $h', ht', lt', \beta \models s' \approx \texttt{AS}_{\texttt{in}}[m, pc']$, which concludes the proof of $\mathcal{P}(0, m)$.

Inductive case For proving $\mathcal{P}(n+1, m)$, the idea is to isolate one of the method calls, and to split the computation into three parts. Indeed, we know that there exist n_1, n_2 and n_3 such that a transition $c \Rightarrow_{n+1} c'$ can be decomposed into $c \Rightarrow_{n_1} c_1 \rightarrow_{n_2} c_2 \Rightarrow_{n_3} c'$, with $n_2 \neq 0$ and $n + 1 = n_1 + n_2 + n_3$. The first and third parts are easily treated applying the induction hypothesis. The method call $c_1 \rightarrow_{n_2} c_2$ is handled in a way similar to the base case. We prove an instruction-wise correctness intermediate lemma (Lemma 8.1 in Appendix 8.1) similar to Lemma 4.3, this time dealing with object initialization and method calls, under the induction hypothesis $\forall k{<}n + 1, \forall m', \mathcal{P}(k, m')$. The induction hypothesis is also applied on the execution of the callee, with a strictly lower call depth.

4.4.3 Application examples

We now illustrate how the verdict of a static analysis on a BIR program can be translated back to the initial BC program. We consider two simple examples of safety properties.

Null-pointer error safety In [Hub08], Hubert *et al.* propose a null-pointer analysis. Their analysis infers , for each field of each class of the program, whether the field is definitely non-null or possibly null after object initialization. The analysis uses expression reconstruction to improve the accuracy of the analysis, needs to reconstruct the link between freshly allocated reference in the heap and the call of the constructor on it. The BIR language provides this information, and this would have eased the analysis.

The BIR program is said to be null-pointer error safe if it does not reach a state of the form $\langle h_e, m, pc, l_e\rangle_{\mathsf{NP}}$. Suppose now the analysis on BIR states that a program is safe. According to Theorem 4.1, if the initial program would have reached an error state $\langle h_e, m, pc, l_e\rangle_{\mathsf{NP}}$, then so would have the BIR program. Thus, the BC program is also null-pointer error safe.

Bounded field value Suppose we want to ensure for a given program, that the integer field f of each object of a class C always has a value within the interval $[0, 10]$. This problem is solved with an interval analysis. It determines at each program point an interval in which variables and object fields take their values. Then, a program is safe if, at every program point where a field f is assigned, the field interval is not too large. Again, the analysis would be easier and more precise on the BIR version of the program. The safety property can be expressed on

the event trace of the BIR program. A program is safe if and only if all its execution traces do not contain any event in the set $\{r.\mathtt{f} \leftarrow (\mathsf{N}\ n) \mid r \in \mathsf{Ref}, n \in [\text{-}\infty; \text{-}1] \cup [11; +\infty]\}$.

Let P be a BC program and $P' = \mathsf{BC2BIR}(P)$ its BIR version. Suppose that P' is safe. We show P is also safe. Let s be a reachable state of P, and Λ the observable trace. Applying Theorems 4.1, we get that there exist a partial bijection β and a trace Λ' emitted by P' such that $\Lambda \overset{!}{\sim}_\beta \Lambda'_{proj}$. We conclude with the definition of $\overset{!}{\sim}_\beta$.

4.5 The Sawja tool bench

BIR and BC2BIR have been implemented for the full JBC language. They are now integrated inside the Sawja library, which provides, together with its sub-component Javalib, all the functionalities for efficiently manipulating Java bytecode programs, and building bytecode static analyzers. The library is developed by the Celtique research group and is freely available at `http://sawja.inria.fr/`. It has already been successfully used in two implementations for the ANSSI (The French Network and Information Security Agency) [JKP11, HJMP10].

Sawja is implemented in OCaml [Ot07], a strongly typed functional language whose automatic memory management (garbage collector), strong typing and pattern-matching facilities make it particularly well suited for implementing program processing tools. In particular, it has been successfully used for programming compilers (e.g., Esterel [PAM+09]) and static analyzers (e.g., Astrée [BCC+03]).

Our contribution to Sawja mainly focused on the IRs. We give below an overview of the library, and refer to [HBB+11] for a full description of the library. Next, we present the adaptations of BIR to the full JBC language. Finally, we provide some experimental results evaluating the IR, and its generation algorithm.

4.5.1 Overview of the library

The main contribution of the Sawja library is to provide, in a unified framework, several features that allow rapid prototyping of efficient static analyses while handling all the subtleties of the JVM specification [LY99]. The main features of Sawja are:

- `.class` files parsing into OCaml structures and the corresponding unparsing
- a decompilation of the bytecode into the high-level stackless IR we described
- a sharing of complex objects both for memory saving and efficiency purpose (structural equality becomes equivalent to pointer equality and indexing allows a fast access to tables indexed by class, field or method signatures, etc.)
- the determination of the set of classes constituting a complete program (using several algorithms, including Rapid Type Analysis (RTA) [BS96])
- a careful translation of many common definitions of the JVM specification, e.g., about the class hierarchy, field and method resolution and look-up, intra and inter-procedural control flow graphs

Sawja is built on top of Javalib, a Java bytecode parser providing basic services for manipulating class files, i.e., an optimised high-level representation of class files, pretty printing and unparsing of class files.[9] Javalib handles all aspects of class files, including stackmaps (J2ME

[9] Javalib is a sub-component of Sawja, which, while being tightly integrated in Sawja, can also be used independently. It was initiated by Nicolas Cannasse before 2004 but, since 2007, it has been largely extended.

and Java 6) and Java 5 annotation attributes. Representing class files constitutes the low-level part of a bytecode manipulation library. The BIR representation is then provided by Sawja for making program analysis and manipulation simpler.

Sawja provides two variants of BIR. JBir is the extension of BIR to the full Java bytecode, while JA3bir is a three-address code variant of JBir (expressions trees are of size at most one). The generation algorithm is factored between the two variants. [10]

4.5.2 From BC to JBC

There are several language features that we did not include in the formalization of the previous sections. First, when dealing with 64 bit values, the behavior of polymorphic bytecodes like `pop2` requires to recover type information in order to predict if the current operand stack will start with two values of 32 bits or one value of 64 bits. This information is easily computed in one pass. We obtain by the same occasion the size of operand stack at each program point and the set of join points. Second, the algorithm of Section 4.3 has been optimised in order to (i) compute the previous information on the fly (except the set of branching points that still require a preliminary scan), (ii) check the constraint C during the algorithm pass by looking only at the successors of the current point, (iii) only keep in memory the symbolic operand stack of the current program point.

Another particularity of the full JBC is the presence of Java subroutines (bytecodes `jsr`/`ret`). Before generating JBir, subroutines are inlined by Sawja. Subroutines have been pointed out by the research community as raising major static analysis difficulties [SA98]. Our restricted inlining algorithm cannot handle nested subroutines but is sufficient to inline all subroutines from Sun's Java 7 JRE.

4.5.3 Experiments

We validate the Sawja IR with respect to three criteria. We first evaluate the time efficiency of the IR generation from JBC. Then, we show that the generated code contains a reasonable number of local variables. We additionally compare our tool with the Soot framework. Finally, we measure the impact of BIR on the precision of an interval analysis.

Our benchmark libraries are real-size Java code available in `.jar` format. This includes Javacc 4.0 (Java Compiler Compiler), JScience 4.3 (a comprehensive Java library for the scientific community), the Java runtime library 1.5.0_12 and Soot 2.2.3. The following experiments have been performed on a Mac-Book Pro with 2.53 GHz Intel Core 2 Duo processor and 4 GB 1067 MHz DDR3 RAM. The Soot framework has been run on the Java HotSpot(TM) Client VM 1.5.0_22-147.

IR generation time We compare the transformation time of our tool with the one of Soot. The results are given in Figure 4.14. For each benchmark library, we compare our running time for transforming all classes with the running time of Soot. For scale reason, the Java runtime library measures are not shown. Here, we choose to generate with Soot the Grimp representation of classes[11], the closest IR to ours that Soot provides. Grimp allows expressions with side-effects, hence expressions are somewhat more aggregated than in our IR. However, this does not change the trend of results. We rely on the time measures provided by Soot, from

[10] Additionally, the last release of Sawja includes SSA variants of JBir and JA3Bir.

[11] The Soot transformation is without any optimisation option.

which we only keep three phases: generation of naive Jimple 3-address code (P1), local def/use analysis used to simplify this naive code (P2), and aggregation of expressions to build Grimp syntax (P3). Other phases, like typing, are not directly relevant. Unlike Java code, OCaml code is usually executed in native form. For the comparison not to be biaised, we compare execution times of both tools in bytecode form and also give the execution time of Sawja in native form. These experiments show that Sawja (both in bytecode and native mode) is very competitive with respect to Soot, in terms of computation efficiency. More specifically, our computation time (in bytecode version) is roughly three times less than Soot. In native mode, we are more than ten times faster. We believe this is mainly due to the fact that, contrary to Soot, our algorithm is non-iterative.

Compactness of the obtained code Intermediate representations rely on temporary variables in order to remove the use of operand stack and generate side-effect free expressions. The major risk here is an explosion in the number of new variables when transforming large programs.

In practice our tool stays below doubling the number of local variables, except for very large methods (> 800 bytecodes). Figure 4.15 presents the percentage of local variable increase induced by our transformation, for each method of our benchmarks, and sorting results according to the method size (indicated by numbers in brackets). The number of new variables stays manageable and we believe it could be further reduced using standard optimization techniques, as those employed by Soot, but this would require to iterate on each method.

We have made a direct comparison with Soot in terms of the local variable increase. Figure 4.16 presents two measures. For each method of our benchmarks we count the number N_{Sawja} of local variables in our IR code and the number N_{Soot} of local variables in the code generated by Soot. A direct comparison of our IR against Grimp code is difficult because it allows expressions with side-effects, thus reducing the amount of required variables. Hence, in this experiment, the comparison is made between Soot's 3-address IR (Jimple) and our 3-address IR. For each method we draw a point of coordinate $(N_{\mathsf{Soot}}, N_{\mathsf{Sawja}})$ and see how the points are spread out around the first bisector. For the left diagram, Soot has been launched with default options. For the right diagram, we added to the Soot transformation the local packer that reallocates local variables using use/def information (and hence increases the transformation time). Our transformation competes well, even when Soot uses this last optimization. We could probably improve this ratio using a similar packing, but this would require to iterate on the code.

Impact on static analysis precision In order to get an indication of the gain in analysis precision that the transformation obtains, we have conducted an experiment in which we compare the precision of an interval analysis before and after transformation. We have developed two intra-procedural interval analyses that track ranges of local variables of type `int`. The first analysis is directly made on `.class` files. At each program point we abstract integer local variables and operand stack elements by an interval. The lack of information in the operand stack and local variables makes it impossible to obtain extra information at conditional jump statement (using an abstract backward test [Cou99]). The second analysis is performed on the intermediate representation that is generated by our tool. This time, we benefit from the `if` statement to gain information at each conditional jump.

We have run these analyses on our benchmark libraries. Figure 4.17 presents two exper-

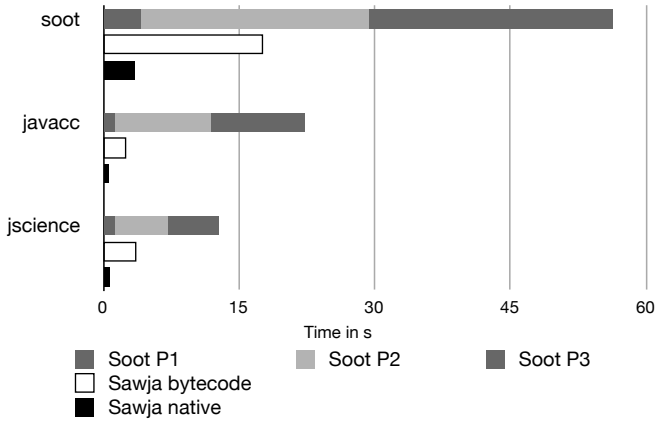


Figure 4.14: Sawja vs. Soot IR generation times

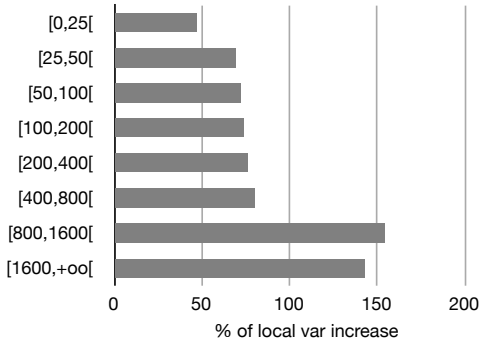


Figure 4.15: Sawja: percentage of local temporaries increase

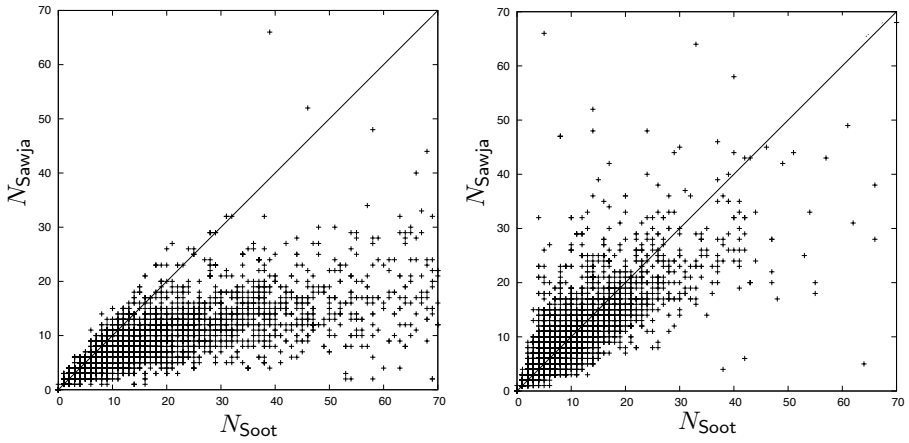


Figure 4.16: Local variable increase ratio between Sawja and Soot

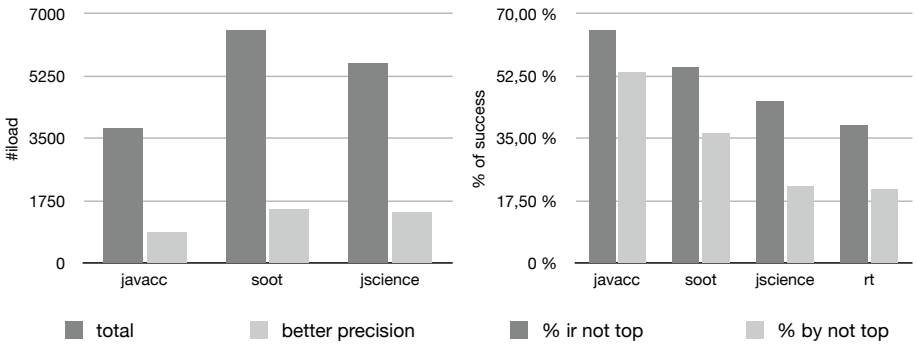


Figure 4.17: Impact on interval analysis precision

imental comparisons. On the left part, we count the total number of `iload` x instructions (left column) and the number of these instructions for which the IR analysis finds a strictly more precise interval for the variable x than the bytecode analysis does. The result of `rt.jar` is similar but not displayed here for scaling considerations (the number of `iload` instruction would be too high). On the right part we take as success criterion the percentage of `iload`s where the interval of x is not a $\top$ ("don't know") information. Again the IR analysis improves significantly the precision. On an average, we gain precision for 25% to 33% of `iload` instructions. For some benchmarks (`jscience.jar` and `rt.jar`), we double the success rate.

4.6 Related work

4.6.1 Transformation and analysis frameworks

Soot [VRCG+99] is a Java bytecode optimization framework providing three IRs: Baf, Jimple and Grimp. Optimizing JBC consists in successively translating bytecode into Baf, Jimple, and Grimp, and then back to bytecode, while performing diverse optimizations on each IR. Baf is a fully typed, stack-based language. Jimple is a typed stackless 3-address code (the type inference has been addressed in [GHM00]). Grimp is a stackless code with tree expressions, obtained by collapsing 3-address Jimple instructions. The stack elimination is performed in two steps, when generating Jimple code from Baf code (see [VRH98] for details). First, naive 3-address code is produced (one variable is associated to each element position of the stack). Then, numerous redundancies of variables are eliminated using a simple aggregation of single def-use pairs. Variables representing stack locations lead to type conflicts when their type is infered, and must hence be desambiguated using additional variables. Our transformation, relying on a symbolic execution, avoids this problem by only merging variables of distinct scopes. Auxiliary analyses (e.g. copy propagation) could further reduce the number of variables, but our algorithm generates very few superfluous variables in practice.

The IR in Sawja and Soot are very similar but are obtained by different transformation techniques. Sawja only targets static analysis tools and does not propose inverse transformations from IR to bytecode. Several state-of-the-art control-flow analyses, based on points-to analyses, are available in Soot through Spark [LH03] and Paddle [LH08]. Such libraries represent a coding effort of several man-years. To this respect, Sawja is less mature and only proposes simple (but efficient) control-flow analyses.

Jalapeño The Jalapeño Optimizing Compiler [BCF+99] which is now part of the Jikes RVM relies on two IR (low and high-level IR) to optimize the bytecode. The high-level IR is a 3-address code providing explicit check operators for common run-time exceptions (`null_check`, `bound_check`...), so that they can be easily moved or eliminated by optimizations. We use similar explicit checks but to another end: static analyses definitely benefit from them as they ensure expressions are error-free, and that the exception throwing order is preserved. We additionally use the `mayinit` instruction to ensure the preservation of the class initialization order, that could otherwise be broken because of folded constructors and side-effect free expressions. Our work pushes the technique further, generating tree expressions in conditional branchings and folding constructors.

Analysis frameworks Wala [Fin] is a Java library dedicated to static analysis of JBC. The framework is very complete and provides several modules like control flow analyses, slicing analyses, an inter-procedural dataflow solver and a IR in SSA form. Wala also includes a front-end for other languages like Java source and JavaScript. Wala and its IBM predecessor DOMO have been widely used in research prototypes. It is the product of the long experience of IBM in the area. Compared to it, Sawja is a more recent library with fewer components, especially in terms of static analyses examples. Nevertheless, the results presented in [HBB+11] show that Sawja loads programs faster and uses less memory than Wala. The last release of Sawja includes an SSA variant of BIR. Julia [Spo05] is a generic static analysis tool for JBC based on the theory of abstract interpretation. It favors a particular style of static analysis specified with respect to a denotational fixpoint semantics of JBC. Initially free software, Julia is not available anymore. The static analyzer Costa [AAG+07] relies on an IR similar to Jimple (although untyped) by removing explicit uses of the operand stack using additional local variables. Finally, Clousot [FL11] is a tool for statically checking code contracts for .NET CLI bytecode, that are specified as pre- and post-conditions and object invariants. It is based on the theory of abstract interpretation, and as such, the authors discuss in [LF08] an alternative solution for reconstructing the expressions, based on the use of symbolic domains exclusively. They have however opted for a code transformation in the implementation of their tool [FL11].

Unlike all works cited above, our transformation does not require iterating on the method code. Still, the number of generated variables stays small in practice. All these previous works have been mainly concerned with the construction of effective and powerful tools but, as far as we know, no attention has been paid to the formal semantic properties that are ensured by these transformations.

4.6.2 Transformation techniques and proofs

The use of a symbolic evaluation of the operand stack to recover some tree expressions in a bytecode program has been employed in several contexts of JBC analysis. The technique was already used in one of the first Sun Just-in-time compilers [CFM+97] for direct translation of bytecode to machine instructions. Whaley's work [Wha99] for the Jalapeño compiler also relies on symbolic interpretation. Our algorithm is largely inspired by this work, but it performs a fixed number of passes on the bytecode while their algorithm is iterative. Xi and Xia propose a dependent type system for array bound check elimination [XX99]. They use symbolic expressions to type operand stacks with *singleton* types in order to recover relations between lengths of arrays and index expressions. Besson *et al.* [BJP06], and independently Wildmoser *et al.* [WCN05], propose an extended interval analysis using symbolic decompilation that verifies that programs are free of out-of-bound array accesses. Besson *et al.* give an example that shows how the precision of the standard interval analysis is enhanced by including syntactic expressions in the abstract domain. Our benchmarks (Section 4.5) confirm this impact empirically. Barthe *et al.* [BKPSF08] also use a symbolic manipulation for the relational analysis of a simple bytecode language and prove it is as precise as a similar analysis at source level.

Our correctness theorem for the transformation is parametrized by a partial bijection between the heaps. A similar notion is used in the work of Banerjee [BN05] and Barthe [BR05] in Java and JBC program non-interference analyses. The indistinguishability relations over the elements of the semantic domains is parametrized by a partial bijection, for defining configurations as indistinguishable, even if their allocation history on high parts of the heap are different. In the CompCert compiler chain, memory states can be modified, but their

structure is somewhat preserved. For instance, to produce Cminor code from Clight, some local variables whose addresses are not taken are moved from the memory to a distinct local environment. Then follows a "packing" phase for the remaining variables into what will be later the stackframe of the function. Leroy and Blazy [LB08] rely on memory injections that, similarly to our β, must be built incrementally during the proof of correctness of the transformation.

Among the numerous works on program transformation correctness proofs, the closest are those dealing with formal verification of the Java compiler algorithms [SBS01, Str02, KN06]. Our work studies a transformation from bytecode to a higher intermediate level and handle difficulties (non preservation of allocation order) that were not present in these works.

4.7 Conclusions

Most of the Java bytecode static analyses and optimizations rely on stackless representation of the code, where the operand stack manipulation is replaced for operations on local registers, which makes analyses and code manipulations easier to implement. However, no semantics is given to the IR that is provided by the tools, and the correspondence between initial and transformed programs is not stated clearly. Moreover, the actual implementation of those transformations are highly optimized. The lack of semantic foundations makes it difficult to figure out what does the conversion ensures, whether the optimizations performed would be correct, or whether an analysis plugged at the level of the IR of such tools would be sound with respect to the initial bytecode program, regarding e.g. object initialization or exceptions.

In this chapter, we have presented a semantically sound, provably correct transformation of Java bytecode into BIR, a stackless IR that (i) removes the use of the operand stack and rebuilds expression trees, (ii) makes explicit the throwing of exceptions and takes care of preserving their order, (iii) rebuilds the initialization chain of an object with a dedicated instruction $x := \texttt{new}\ C(arg_1, \ldots, arg_n)$.

Relying on simulation diagrams and observational semantics with observable event traces, we are able to state precisely whether a given semantic aspect is preserved by the transformation, and if not, how it is mapped to the semantics of the IR. This fined-grained semantic characterization makes it possible, for safety properties expressible on semantic traces, how some BIR static analysis verdicts could be translated back to the initial BC program.

This work has been implemented in Sawja, the first OCaml library providing state-of-the-art components for writing Java static analyzers in OCaml, that accepts full JBC as input. Our benchmarks show the expected efficiency is obtained in practice, with regard to generation time and the number of generated temporaries.

The future directions to give to this work are presented in Chapter 7.

Chapter 5

Static Single Assignment form

5.1 Introduction

Static Single Assignment (SSA) form [CFR+91] is an intermediate representation where variables are statically assigned exactly once. In Chapter 2, we reported on how the considerable strength of this apparently simple property makes the SSA form simplify the definition of many optimizations, the most complex of which being CSE or PRE, and analyses such as, use-definition chains or variable live-ranges. It also improves their efficiency thanks to its sparseness, as well as the quality of their results. Many modern compilers, including GCC [GCC] and LLVMC [LLV], rely heavily on SSA, and there is a vast body of work on SSA (see [SSAar] for a comprehensive state-of-the-art and techniques).

5.1.1 Powerful properties require care

The strengh of the SSA form lies in all the properties this IR embeds. This is why SSA is of particular interest in our work. First, SSA generation algorithms must establish these properties. To perform the translation to SSA efficiently, they rely on powerful and complex data structures (e.g. the dominator tree) whose construction is difficult to justify formally. Those properties are then exploited on a semantic side by SSA-based optimizations, i.e. their correctness depends on them. Further, many SSA-based optimizations are usually performed in sequence. They should hence preserve those properties. Finally, because the SSA form is not executable, it has to be destructed after the SSA code has been optimized. Again, in order for this translation to be correct, the semantics of SSA must be deeply understood.

Hence, the simplicity of the SSA form is deceptive, and should instead be considered with care. Designing a correct SSA-based middle-end compiler has in fact been fraught with difficulties [BCHS98, BDR+09]: it has been a significant challenge to design efficient, semantics-preserving, algorithms for converting programs into SSA form, optimizing SSA programs, or transforming programs out of SSA form. A famous example is the destruction algorithm of SSA presented in [CFR+91]: it was identified to produce incorrect results on optimized SSA code. Briggs et al. clearly identified the problem and proposed a solution published several years later [BCHS98]. This is precisely the kind of situations that could have been avoided with a formal proof of the algorithm. Such a proof can however only be conducted if the semantics of SSA is clearly defined, which was done only rather recently [Gle04, BGLM05, Pop06]. Even more pernicious are bugs that occur in the very implementation of SSA-related algorithms. For instance, computing efficiently the dominator tree can be done with the algorithm proposed by Lengaeur and Tarjan in [LT79]. However, the algorithm is known to be complex, as is attested by a recent paper by Georgiadis and Tarjan [GT05] defining a validating algorithm for this construction, that is hoped to be simpler to understand, and thus to implement. Here,

the formal verification of compilers helps bridge the gap between an algorithm and its concrete implementation, and the extremely demanding nature of verified compilers raises interesting challenges.

5.1.2 Verified compilers need semantic properties

Compiler verification aims at giving a rigorous proof that a compiler preserves the behavior of programs. McCarthy and Painter [MP67] were the first to prove the correctness of a compiler for arithmetic expressions. Later on, in 1973, Morris proposed in [Mor73] a methodology for proving the correctness of real size compilers. Moore [Moo89, Moo96] was the first to provide a machine-assisted proof of a compiler for the custom high-level assembly language Piton. After 40 years of a rich history, the field is entering into a new dimension. Strecker [Str02], and Klein and Nipkow [KN06] formally verified non-optimizing bytecode compilers from a subset of Java to a subset of Java Bytecode with the Isabelle/HOL proof assistant.

In this work, we focus on CompCert [Ler09], a realistic and optimizing compiler that is programmed and verified in the Coq proof assistant and generates compact and efficient assembly code for a large fragment of the C language. Leroy's CompCert has been rightfully acclaimed as a *tour de force*, but it foregoes relying on an SSA-based middle end. In [Ler09], Leroy reports:

> *Since the beginning of CompCert we have been considering using SSA-based intermediate languages, but were held off by two difficulties. First, the dynamic semantics for SSA is not obvious to formalize. Second, the SSA property is global to the code of a whole function and not straightforward to exploit locally within proofs.*

and adds: "A typical SSA-based optimization that interests us is global value numbering". However verifying GVN is a significant challenge, and its formal verification has remained beyond current state-of-the-art in verified compilers.

The structural properties of SSA are well-identified in the literature, and some proofs of SSA-based analyses and transformations can be found [CFR+91, CCK+97, BHG+08]. What is missing in those works is the semantic counterparts of those properties. The proofs are traditionally based on how the SSA-based algorithms work and the information they compute (i.e. properties of the CFG graph). In particular, the semantic properties and invariants established by the SSA generation algorithm are never expressed precisely. This is probably due to the lack of a definition of a semantics for SSA that would be both formal and close to the intuitive definition given in the seminal papers [AWZ88, CFR+91].

5.1.3 Contributions

In this work, we aim at studying from a formal and semantic point of view the SSA form. How to formalize its semantics, and how to prove that the conversions to and out of SSA are correct? What are the key semantic properties used by SSA-based optimizations? Can we isolate those invariants clearly?

To answer these questions, we provide here the first verified *SSA-based middle-end*. Rather than programming and proving a verified compiler from scratch, we have programmed and verified an SSA-based middle-end that can be plugged into CompCert at the level of RTL. As a by-product, this work demonstrates that a compiler can be realistic, verified and still

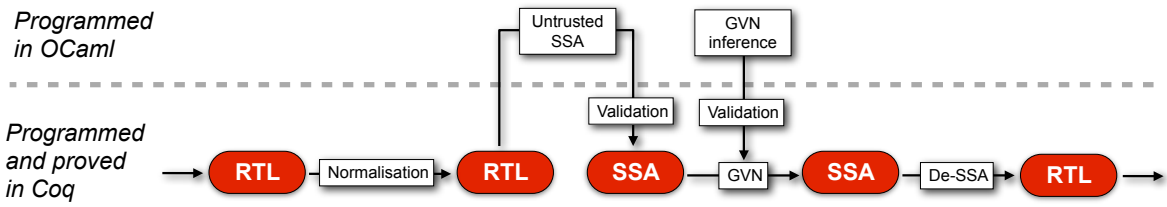


Figure 5.1: Our SSA middle-end

rely on an SSA form. Figure 5.1 describes the overall architecture. Our middle-end performs four phases: (i) normalization of RTL programs (ii) transformation from RTL into SSA form (iii) optimization of programs in SSA form, including Global Value Numbering (GVN) [AWZ88] (iv) transformation of programs from SSA back to RTL. It relies on CompCert for the transformation from C to RTL programs prior to SSA conversion, and from RTL programs to assembly code after conversion out of SSA—our point is to program a realistic and verified SSA-based middle-end, rather than to demonstrate that SSA-based optimizations dramatically improve the efficiency of generated code.

We verify our compiler middle-end with a mix of techniques directly inherited from CompCert. We resort to translation validation —increasingly favored by verified compilers [TL09, TL10, JPL12]—for converting programs into SSA form and for GVN. Specifically, we program in Coq verified checkers that validate *a posteriori* results of untrusted computations, and we implement in OCaml efficient algorithms for these computations; we rely on Cytron et al.'s algorithm [CFR+91] for computing minimal SSA form, and on Alpern et al.'s iteration strategy [AWZ88] for computing a numbering in GVN. In contrast, the normalization of the RTL program, and the conversion out of SSA are directly programmed and proved in Coq.

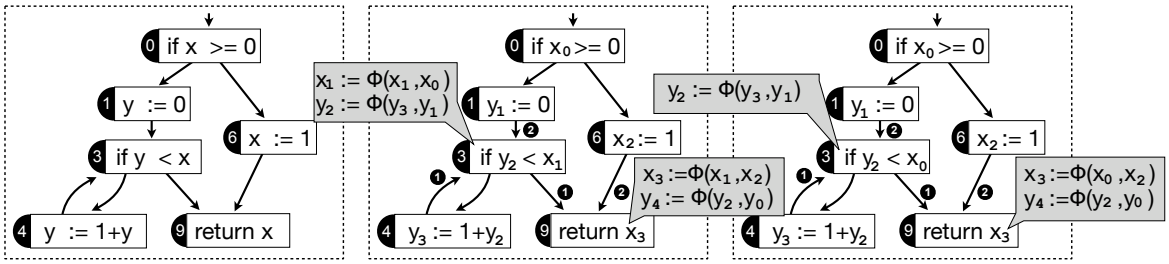
Our work addresses the two issues raised by Leroy [Ler09]. First, we give a simple and intuitive operational semantics for SSA; the semantics follows the informal description given in [CFR+91], and does not require any artificial state instrumentation. Second, we formalize for SSA programs the two global properties of strictness and equational form, allowing to conclude reasonably directly that complex optimizations such as GVN and others are sound.

5.1.4 Contents

Section 5.2 recalls the basic versions of SSA forms, as well as their generation algorithm. The CompCert C compiler is overviewed in Section 5.3. Section 5.4 focuses on the RTL language, at the level of which our SSA middle-end plugs, and defines the code normalization it relies on. In Section 5.5, we define the SSA language used by our middle-end. Conversion to and out of SSA form are presented in Section 5.6 and 5.8 respectively. In Section 5.7, we formalize what we call the equational lemma of SSA, and illustrate its use in SSA-based optimizations, including GVN. We present some experimental results in Section 5.9 and related work in Section 5.10, before concluding in Section 5.11. The full formalization is available online

`http://compcertssa.gforge.inria.fr`

Notations Throughout the chapter, we use Coq syntax for our definitions and results. Statements occasionally involve some notions that are not introduced formally. In such cases, names



(a) Example program (b) Maximal SSA form (c) Minimal SSA form

Figure 5.2: Example program and its SSA forms

are generally chosen to be self-explanatory (for instance, `not_wrong_program`); in other cases, we forego giving precise definitions as they are not needed to understand this work (for instance, the types `chunk` and `addressing` are unspecified in the definition of state). Our formalization extensively uses inductive definitions, introduced in Coq using the keyword **`Inductive`**. Inductive definitions are used for introducing both datatypes, e.g. the type of RTL instructions (Figure 5.4), and inductive relations, e.g. the operational semantics of RTL (Figure 5.4). In the latter case, declarations are written with the following pattern

```
Inductive R : A → B → Prop :=
  | Rule1: ∀ a b, ... → R a b
  | Rule2:... → R a b
```

meaning that the relation `R` is a binary predicate (indicated by `Prop`, the type of propositions in `Coq`) whose arguments are of types `A` and `B` respectively. The relation `R` is defined by two rules `Rules1` and `Rules2`, describing when the proposition (`R a b`) holds for elements `a` and `b` (the hypotheses are indicated by dots).

5.2 Background on SSA

We complete the overview of SSA given in Chapter 2, by recalling the variants of SSA in terms of ϕ-function placement, and the concepts of the underlying generation algorithms.

Straight-line code Converting into SSA form is easy for straight-line code: one simply tags each variable definition with an index, and each variable use with the index corresponding to the last definition of this variable. For example, $[x := 1; y := x + 1; x := y - 1; y := x]$ is transformed into $[x_0 := 1; y_0 := x_0 + 1; x_1 := y_0 - 1; y_1 := x_1]$. The transformation is semantics-preserving, in the sense that the final values of x and y in the first snippet coincide with the final values of x_1 and y_1 in the second snippet.

ϕ-functions One cannot transform arbitrary programs into semantically equivalent programs in SSA form solely by tagging variables: one must insert *ϕ-functions* to handle branching statements. Figure 5.2a shows an example program and an SSA form of it is given in Figure 5.2b. In Program 5.2a, the value of variable x read at node 9 either comes from the definition of x at entry or at node 6. In Program 5.2b, these two definitions of x are renamed into the unique definition of x_0 and x_2 and merged together by the ϕ-function of x_3 at entry of node 9. The precise meaning of a ϕ-block depends on the numbering convention of the

predecessor nodes of each junction point. In Figure 5.2b, we make explicit this numbering by labelling the CFG edges. Node 3 is the first predecessor of point 9 and node 6 is the second one. The semantics of ϕ-functions is given in the seminal paper by Cytron et al. [CFR+91]:

> *If control reaches node j from its kth predecessor, then the run-time support remembers k while executing the ϕ-functions in j. The value of $\phi(x_1, x_2, \ldots)$ is just the value of the kth operand. Each execution of a ϕ-function uses only one of the operands, but which one depends on the flow of control just before entering j.*

Maximal, minimal, pruned SSA There may be several SSA forms for a single program CFG. Figure 5.2 gives alternative SSA forms for a same initial program. In the maximal SSA form (Figure 5.2b), a ϕ-function is inserted for all program variables, at each join point. As the number of ϕ-functions directly impacts the quality of the subsequent optimizations—as well as the size of the SSA form—it is important that SSA generators for real compilers produce an SSA form with a minimal number of ϕ-functions.

The minimal SSA form is informally specified as follows: a ϕ-function is needed for a given variable at join points that can be reached by at least two distinct definitions of that variable in the initial program. This is captured by the notion of convergence point of CFG paths starting at two distinct definition points of a variable (the join operator in [CFR+91]).

Consider the program examples in Figures 5.2a and 5.2c. Two definitions of y (at point 1 and 4) can reach the join point 3: a ϕ-instruction is required at node 3 in Program 5.2c. On the other hand, there is only one definition of x (the initial implicit definition of x) that reaches that point in Program 5.2a and no ϕ-function is inserted for x at point 3 in Program 5.2c.

Algorithmically, it is more efficient to determine the placement of ϕ-functions of minimal SSA using the equivalent notion of *dominance frontier*.

Definition 5.1 (Dominance relation)**.** *A node i in a CFG dominates another node j if every path from the entry node of the CFG to j contains i. The dominance is said to be strict if additionally $i \neq j$.*

Definition 5.2 (Dominance frontier)**.** *For a node i of a CFG, the* dominance frontier *$DF(i)$ of i is defined as the set of nodes j such that i dominates at least one predecessor of j in the CFG but does not strictly dominates j itself. The notion is extended to a set of nodes S with $DF(S) = \bigcup_{i \in S} DF(i)$.*

Definition 5.3 (Iterated dominance frontier)**.** *The* iterated dominance frontier *$DF^+(S)$ of a set of nodes S is $\lim_{i \to \infty} DF^i(S)$, where $DF^1(S) = DF(S)$ and $DF^{i+1}(S) = DF(S \cup DF^i(S))$.*

Efficient algorithms for computing the dominance frontiers rely on an effective representation of the dominance relation, the dominator tree. It relies on the notion of immediate dominator.

Definition 5.4 (Immediate dominator)**.** *The immediate dominator of a node j, written $idom(j)$ is the closest strict dominator of j on every path from the entry node to j. It is uniquely determined.*

Definition 5.5 (Dominator tree)**.** *The dominator tree is defined as follows. The start node is the root of the tree. Each node's children are the nodes it immediately dominates.*

In a *minimal SSA* program generated by Cytron et al.'s algorithm, every ϕ-function of an instance x_i of an original variable x appears in a junction point j if and only if j belongs to the iterated dominance frontier of the set of definition nodes of x in the original program.

However, one can achieve more compact SSA forms by observing that, at any junction point, dead variables need not to be defined by a ϕ-function. The intuition is captured by the notion of *pruned SSA* form: a program is in *pruned SSA* form when the ϕ-functions appear at the iterated dominance frontiers and for each ϕ-function of an instance x_i of an original variable x at a junction point j, x is live at j in the original program (there is a path from j to a use of x that does not redefine x). Compared to minimal SSA (Figure 5.2c), pruned SSA detects that the ϕ-function for y at point 9 can be removed.

5.3 The CompCert C compiler

CompCert is a realistic formally verified compiler that generates PowerPC, ARM or x86 code from source programs written in a large subset of C. Most of the optimizations performed by CompCert are done at the level of RTL (we present this IR in more detail in Section 5.4). The version of CompCert we consider in this work includes the following optimizations: constant propagation, removal of redundant casts, tail-call detection, local value numbering for common-subexpression elimination, and a register allocation that includes copy propagation.

CompCert formalizes the operational semantics of a dozen of intermediate languages, and proves for each compilation phase a semantics preservation theorem. Program behaviors are finite or infinite traces of observable actions performed along the program execution (see Section 5.3.1), and correctness theorems claim that individual compilation phases preserve behaviors.

As discussed in Chapter 3, the global correctness theorem of the CompCert compiler is expressed as follows. For any C program `p` that does not go wrong, and target program `tp` output by the successful compilation of `p` by the compiler `compcert_compiler`, the set of behaviors of `p` contains all behaviors of `tp`. The formal statement of the theorem is:

```
Theorem compcert_compiler_correct: ∀ (p: C.program) (tp: Asm.program),
  (not_wrong_program p ∧ compcert_compiler p = OK tp) →
  (∀ beh, exec_asm_program tp beh → exec_C_program p beh).
```

Thanks to the code extraction mechanism provided by the Coq proof assistant, the (Coq) code of the compiler `compcert_compiler` can then be translated into a piece of Ocaml code that is correct by construction.

5.3.1 Observational semantics

We present the key ingredients of the observational semantics given to the IRs of CompCert. More details can be found in [Ler12].

The observable behaviors of programs is formulated in terms of input/output events, representing the interaction that the program has with the external environment. An event is either a system call, a volatile load (in the sense of C) or a volatile store to a global memory location, with additional arguments (specifying e.g. the name of the system call, its arguments, or the identifier of the global volatile variable). The only requirement imposed on events is that they do not mention memory states and pointer values, which are not preserved during compilation. Along their execution, programs emit traces of observable events. Event traces

```
Inductive program_behavior: Type :=
  | Terminates (t: trace) (ret: int)                  terminating trace and return code
  | Diverges (t: trace)                               diverging after a finite observable trace
  | Reacts (t: traceinf)                              diverging with a infinite trace
  | Goes_wrong: (t: trace).                           getting stuck after a finite observable trace

Variable state: Type.                                 the type of execution states
Variable step: state → trace → state → Prop.          the labelled transition relation
Variable initial_state: state → Prop.                 the initial states
Variable final_state: state → int → Prop.             the final states (and exit codes)


Inductive program_behaves: program_behavior → Prop :=
  | program_terminates: ∀ s t s' r,
      initial_state s →
      star step s t s' →                              finite sequence of steps
      final_state s' r →
      program_behaves (Terminates t r)

  | program_diverges: ∀ s t s',
      initial_state s →
      star step s t s' →                              finite sequence of steps
      forever_silent s' →                             infinitely many silent steps
      program_behaves (Diverges t)

  | program_reacts: ∀ s T,
      initial_state s →
      forever_reactive s T →                          infinitely many non-silent steps
      program_behaves (Reacts T)

  | program_goes_wrong: ∀ s t s',
      initial_state s →
      star step s t s' →                              finite sequence of steps
      nostep s' →                                     no transition can be done
      (∀ r, ¬final_state s' r) →                      the blocking state is non-final
      program_behaves (Goes_wrong t)

  | program_goes_initially_wrong:
      (∀ s, ¬initial_state s) →                       there is no initial state
      program_behaves (Goes_wrong ϵ).
```

Figure 5.3: Observable behaviors of programs in CompCert

can either be finite (of type `trace`) or infinite (of type `inftrace`). A program behavior is then either a terminating, diverging or going wrong observable trace. It is formally defined by predicate `program_behavior` in Figure 5.3.

Assuming a transition system characterized by the type of its states, its initial and final states, and its labelled transition relation (see Figure 5.3), the predicate `program_behaves` then specifies the way execution sequences in the system give rise to a given behavior.

5.3.2 Behavior preservation

Each phase of the compiler is formally proved relying on simulation techniques, as presented in Chapter 3. The formal development of CompCert provides the general correctness theorems of the simulation diagrams presented in Chapter 3.

Some parts of the CompCert compiler are not directly proved in Coq. This is the case of the register allocation phase [Ler09], that is based on a graph coloring algorithm. The graph coloring algorithm is written in Ocaml, and then validated a posteriori by a checker written in Coq. As explained in Chapter 3, the correctness proof of the checker (stating that if a coloring is validated, then this is indeed a valid coloring) ensures this compilation phase has the same guarantees than a transformation that would be written and proved directly in Coq, with the

```
Inductive instr  :=
  | Inop (pc: node)
  | Iop (op: operation) (args: list reg) (res: reg) (pc: node)
  | Iload (chk:chunk) (addr:addressing) (args: list reg) (res: reg) (pc: node)
  | Istore (chk:chunk) (addr:addressing) (args:list reg) (src: reg) (pc: node)
  | Icall (sig: signature) (fn:ident) (args: list reg) (res: reg) (pc: node)
  | Icond (cond: condition) (args: list reg) (ifso ifnot: node)
  | Ireturn (or: option reg).

Definition code := PTree.t instr.   type of code graph

Record function := {
   fn_sig: signature;               function signature
   fn_params: list reg;             parameters
   fn_stacksize: Z;                 activation record size
   fn_code: code;                   code graph
   fn_entrypoint: node              entry node
}.

Inductive state :=
 | State (s: list stackframe)       call stack
         (f: function)              current function
         (sp: val)                  stack pointer
         (pc: node)                 current program point
         (rs: regset)               register state
         (m: mem)                   memory state
 | Callstate (s: list stackframe) (f: fundef) (args: list val) (m: mem)
 | Returnstate (s: list stackframe) (v: val) (m: mem).

Inductive step: genv → state → trace → state → Prop :=
 | ex_Inop: ∀ ge s f sp pc rs m pc',
      fn_code f pc = Some(Inop pc') →
      step ge (State s f sp pc rs m) ϵ (State s f sp pc' rs m)

 | ex_Iop: ∀ ge s f sp pc rs m pc' op args res v,
      fn_code f pc = Some(Iop op args res pc') →
      eval_operation sp op (rs##args) m = Some v →
      step ge (State s f sp pc rs m) ϵ (State s f sp pc' (rs#res←v) m)

 | ex_Iload: ∀ ge s f sp pc rs m pc' chk addr args res a v,
      fn_code f pc = Some(Iload chk addr args res pc') →
      eval_addressing sp addr (rs##args) = Some a →
      Mem.loadv chk m a = Some v →
      step ge (State s f sp pc rs m) ϵ (State s f sp pc' (rs#res←v) m)
```

Figure 5.4: Syntax and semantics of RTL (excerpt)

additional benefit of abstracting from complex implementation details and heuristics.

5.4 The RTL language

Our middle-end is plugged at the level of the RTL language in CompCert. This section presents briefly this language. RTL stands for Register Transfer Language (RTL), a three-address like intermediate representation of the code. It is similar to the TAC language presented in Chapter 2, extended to handle all the features of the CompCert C language.

5.4.1 Syntax and semantics

The syntax and semantics of RTL are given in Figure 5.4. An RTL program is defined as a set of global variables, a set of functions, and an entry node. Functions are modelled as records that include a function signature `fn_sig`, a CFG `fn_code` of instructions over pseudo-registers. The CFG is not a basic-block graph: it partially maps each CFG node to

a single instruction, and we stick to this important design choice of CompCert. As explained by Knoop et al. [KKS98], it allows for simpler implementations of code manipulations and simplifies correctness proofs of analyses or transformations, with low impact on efficiency.

The RTL instruction set includes arithmetic operations (`Iop`), memory loads (`Iload`) and stores (`Istore`), function calls (`Icall`), conditional (`Icond`) and unconditional jumps (`Inop`), and a return statement (`Ireturn`)— we do not discuss here jumptables and other kinds of function calls. The cases of call to a function pointer stored in a register, tail calls, and built-in functions are treated in the full formalization available online. All instructions take as last argument a node `pc` denoting the next instruction to be executed; additionally, all instructions but `Inop` take as arguments pseudo-registers of type `reg`, memory chunks, and addressing modes.

The type of states is defined as the tagged union of regular states, call states and return states (Figure 5.4). We focus on regular states, as we only expose here the intra-procedural part of the language. A regular semantic state (`State s f sp pc rs m`) is a tuple that contains a call stack `s` (representing the current pending function calls), the current function description `f` and stack pointer `sp` (to the stack data block, a part of the global memory where variables dereferenced in the C source program reside), the current program point `pc`, the registers state `rs` (a mapping of local variables to values) and the global memory `m`. The semantics also includes a global environment (of type `genv`) mapping function names and global variables to memory addresses.

The operational behavior of programs is modelled by the relation `step` between two semantic states (see Figure 5.4), and a trace of events; all instructions except function calls do not emit any event, hence the transitions that they induced are tagged by the empty event trace ϵ. We briefly comment on the rules: (`Inop` $\mathtt{pc}'$) branches to the next program point $\mathtt{pc}'$. (`Iop op args res` $\mathtt{pc}'$) performs the arithmetic operation `op` over the values of registers `args` (written `rs##args`), stores the result in `res` (written `rs#res` $\leftarrow$ `v`), and branches to $\mathtt{pc}'$. The instruction (`Iload chk addr args res` $\mathtt{pc}'$) loads a `chk` memory quantity from the address determined by the addressing mode `addr` and the values of the `args` registers, stores the memory quantity just read into `res`, and branches to $\mathtt{pc}'$.

5.4.2 Normalizing RTL syntax

Our SSA middle-end is plugged at the level of RTL in CompCert. But, before generating the SSA form of an RTL code, we rely on a structural normalization phase of the RTL code (see Figure 5.1), that we have added to CompCert, prior to the middle-end proper. This normalization phase consists of transforming an RTL program into another one so that the only instruction that can lead to a junction point is an (`Inop pc`) instruction. The figure below shows an example RTL program and its normalized version. This normalization phase has been programmed and proved in Coq. One could think this normalization phase is quite anecdotal. But this structural constraint will carry over the SSA form of RTL programs, and will allow us lightening our formal development considerably. As will be pointed out in the next sections, this impacts the formal definition of the SSA semantics. This also lightens the definition of and proof of the SSA validator, the GVN-based CSE, and the SSA deconstruction.

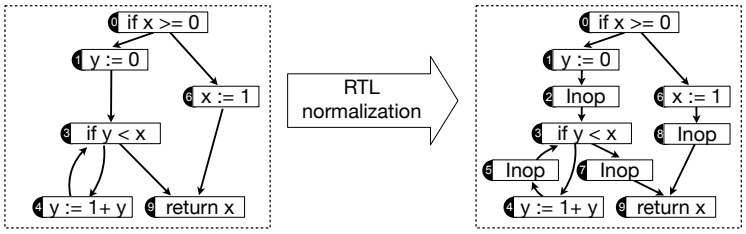


Figure 5.5: An RTL program and its normalized version

```
Definition reg := RTL.reg * idx                    indexed registers

Inductive instr := ...                             RTL-like instructions
                                                   (operating on SSA.regs)
Inductive phiinstr :=
 | Iphi (args: list SSA.reg) (res: SSA.reg).       φ-functions

Definition phiblock:= list phiinstr.               φ-blocks

Record function := {
   fn_sig: signature;            function signature
   fn_params: list SSA.reg;      parameters
   fn_stacksize: Z;              activation record size
   fn_code: code;                code graph
   fn_phicode: phicode;          φ-blocks graph
   fn_entrypoint: node           entry node
}.
```

Figure 5.6: Syntax of SSA: indexed registers, instructions and functions.

5.5 The SSA language

We describe the syntax and operational semantics of the language SSA that provides the SSA form of RTL programs. We equip the notion of SSA program with a *well-formedness* predicate capturing essential properties of SSA forms.

5.5.1 SSA programs

5.5.1.1 Syntax

Our definition of SSA program distinguishes between RTL-like instructions and ϕ-functions; the distinction avoids the need for unwieldy mappings between program points when converting to SSA, and allows for a smooth integration in CompCert. Figure 5.6 introduces the syntax of SSA. Compared to RTL functions, SSA functions operate on indexed registers of type `SSA.reg`, and include an additional field `fn_phicode` mapping junction points to ϕ-blocks. The latter are modelled as lists of ϕ-functions of the form (`Iphi args res`), where `res` is an indexed register, and `args` a list of indexed registers.

We define structural constraints that allow giving an intuitive semantics to SSA programs. First, we require that the domain of the function `fn_phicode` be the set of junction points. Second, we require that all ϕ-functions in a ϕ-block have the same number of arguments as the number of predecessors of that block. Our last requirement is the normalization criterion of the CFG of SSA functions: all predecessors of a junction point must be (`Inop pc`) instructions.

5.5.1.2 Strict SSA

Finally, we consider two essential properties of SSA forms: unique definitions and strictness (see Chapter 2). The unique definitions property states that each register is statically uniquely defined, whereas the strictness property states that each variable use is dominated by the definition of that variable. While the two properties are closely related, none implies the other; the program $[y_0 := x_0; x_0 := 1]$ satisfies the unique definitions property but is not in strict form whereas the program $[x_0 := 1; x_0 := 2; y_0 := x_0]$ is strict but does not satisfy the unique definitions property.

To formalize these properties, one first defines the type of CFG paths, and two predicates `dom` and `sdom` for dominance and strict dominance. We also prove many properties of the dominance relation, such as its reflexivity, transitivity, and anti-symmetry. Then, one must define the two predicates `def` and `use` of type `SSA.function → SSA.reg → node → Prop` such that proposition (`def f x pc`) (respectively (`use f x pc`)) holds iff the register `x` is defined (resp. used) at node `pc` in the (RTL-like or ϕ-) code of the function `f`. Predicate `def` is defined in the obvious way. The definition of `use` is more involved, because of ϕ-functions. A variable is used either by an RTL-like instruction or a ϕ-function:

```
Inductive use : SSA.function → reg → node → Prop :=
  | u_code : ∀ f x pc, use_code f x pc → use f x pc
  | u_phicode : ∀ f x pc, use_phicode f x pc → use f x pc.
```

where predicate `use_code` defines when a variable is used in the RTL-like code. It is defined straightforwardly: a variable is used if it appears in the right hand-side of an assignment, in the condition of an `Icond` instruction, as an argument of a function call etc. We now explain predicate `use_phicode`. The widely adopted convention is to view ϕ-functions as lazily evaluated. The kth argument of a ϕ-function is used at the kth predecessor of the corresponding block.

```
Inductive use_phicode : SSA.function → reg → node → Prop :=
 | upc_intro : ∀ f pc pred k arg args dst phib
    (PHIB  : fn_phicode f pc = Some phib)
    (ASSIG : In (Iphi args dst) phib)
    (KARG  : nth_error args k = Some arg)          arg is the kth elements of args
    (KPRED : index_pred f pred pc = Some k),       pred is the kth predecessor of pc in f
    use_phicode f arg pred.
```

This matches the semantics we formally define in Section 5.5.2: ϕ-functions are executed along the edge leading to the ϕ-block. This definition also allows to reuse the traditional notion of strictness defined on non-SSA programs.

Figure 5.7 illustrates the definition of predicates `def` and `use`. Using those predicates, we are now able to state formally the unique definitions and strictness properties, that defines the strict SSA form. We omit the formal definition of `unique_def` (it is as expected but rather verbose).

```
Definition unique_def (f: SSA.function) := ...

Definition strict (f: SSA.function) := ∀ x u d, use f x u → def f x d → dom f d u.
```

5.5.1.3 Well-formed SSA programs

The well-formedness condition for SSA programs gather the unique definitions and strictness properties, as well as some additional structural constraints on the CFG and ϕ-functions.

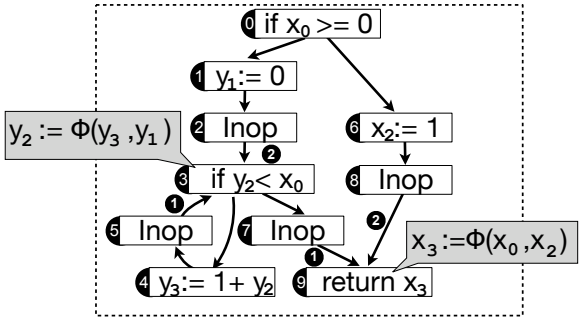


Figure 5.7: Example (normalized) SSA program – The variable y_2 is defined at node 3 (in the ϕ-block attached to that program point), and used at point 3 and 4. The variable x_2 is defined at node 6 and used at node 8, the second predecessor of the junction point 9, where x_2 appears as 2nd argument of the ϕ-function.

```
Record wf_ssa_function (f:SSA.function) : Prop := {
 fn_ssa:        unique_def f;
 fn_wf_block:   block_nb_args f;
 fn_strict:     strict f;
 fn_block_at_jp:∀ jp, join_point jp f ↔ fn_phicode f jp ≠ None;
 fn_normalized: ∀ jp pc, join_point jp f → jp∈(succs f pc) →fn_code f pc =Some(Inop jp);
}.
```

The predicate `block_nb_args` captures that ϕ-function arguments are consistant with the number of predecessors of the CFG node holding the block. In the sequel, we show that conversion to SSA yields well-formed programs. Besides, our SSA-based optimizations will assume that the input SSA programs are well-formed; in turn, we prove for each of them that output programs are well-formed.

5.5.2 Semantics

The notion of SSA state is similar to the notion of RTL state, except that the type of registers and current function are modified into `SSA.reg` and `SSA.function` respectively. We describe now the semantics of SSA programs.

5.5.2.1 Exploiting normalization for an intuitive semantics

The small-step operational semantics is defined on SSA programs that satisfy the structural constraints introduced in the previous paragraph (`wf_ssa_function`).

Formally, we define `SSA.step` as a relation between pairs of SSA states and a trace of events. The definition follows the one of `RTL.step`, except for instructions of the form (`Inop` pc'), where one distinguishes whether the successor pc' is a junction point or not. In the latter case, the semantics coincide with the RTL semantics, i.e. the program point is updated in the semantic state. If on the contrary pc' is a junction point, then one executes the ϕ-block attached to pc' before the control flows to pc'.

Executing ϕ-blocks on the way to pc' avoids the need to instrument the semantics of SSA with the predecessor program point, and crisply captures the intuitive meaning given to ϕ-blocks by Cytron et al. (see Section 5.2). Note in particular that the normalization ensures that the predecessor of a junction point is an `Inop` instruction. This greatly simplifies the

```
Inductive step: SSA.genv → SSA.state → trace → SSA.state → Prop :=
  | ex_Inop_njp: ∀ ge s f sp pc rs m pc',
      fn_code f pc = Some(Inop pc') →
      ¬ join_point pc' f →
      step ge (State s f sp pc rs m) ϵ (State s f sp pc' rs m)

  | ex_Inop_jp: ∀ ge s f sp pc rs m pc' phib k,
      fn_code f pc = Some(Inop pc') →
      join_point pc' f →
      fn_phicode f pc' = Some phib →
      index_pred f pc pc' = Some k →
      step ge (State s f sp pc rs m) ϵ (State s f sp pc' (phistore k rs phib) m)

Fixpoint phistore (k: nat) (rs: SSA.regset) (phib: phiblock) :  SSA.regset :=
  match phib with
    | nil ⇒ rs
    | (Iphi args res)::phib ⇒
          match nth_error args k with
            | None ⇒ rs
            | Some arg ⇒ (phistore k rs phib)#res ← (rs#arg)
          end
  end.
```

Figure 5.8: Semantics of SSA (excerpt)

definition of the semantics (ϕ-block can only be executed after an `Inop`), and subsequently the proofs about SSA programs.

5.5.2.2 Parallel execution of ϕ-blocks

Following conventional practice, ϕ-blocks are given a parallel (big-step) semantics. By construction, the SSA generation algorithm ensures that the ϕ-function arguments are never assigned by a distinct ϕ-function in the same block. So this parallel semantics seems to be of little help. But later optimizations will benefit from this semantics, since it makes explicit the (in)dependences between ϕ-arguments and ϕ-function destinations [HGG06, BDR+09].

The semantics of ϕ-blocks is formally defined with `phistore` (Figure 5.8). When reaching a join point $\mathtt{pc}'$ from its `k`th predecessor, we update the register set `rs` for each register `res` assigned in the ϕ-block `phib` with the value of register `arg` in `rs` (written `rs#arg`), where `arg` is the `k`th operand in the ϕ-function of `res` (written `nth_error args k = Some arg`). The definition of `phistore` satisfies, on well-formed SSA functions, a *parallel assignment* property:

```
  ∀ arg res, In (Iphi args res) phib →
     nth_error args k = Some arg → (phistore k rs phib)#res = rs#arg
```

5.6 Translation validation of the SSA generation

Compilers typically follow the algorithm by Cytron et al. [CFR+91] to generate a minimal SSA form in almost linear time w.r.t. the size of the program. It proceeds in four steps:

- The CFG dominator tree is built using the algorithm of Lengauer and Tarjan [LT79]
- The dominance frontiers are computed with a bottom-up traversal of the dominator tree
- For each initial variable, ϕ-functions are placed using the iterated dominance frontier
- Definitions and uses of RTL variables are renamed with correct indexes, using a top-down traversal of the dominator tree

Programming efficiently the algorithm in Coq and proving formally its correctness is a significant challenge—even verifying formally the construction of the dominator tree requires to formalize a substantial amount of graph theory. Instead, we provide a new validation algorithm that checks in linear time that an SSA program is a correct SSA form of an input RTL program. Below, we show that the algorithm is complete w.r.t. minimal SSA form, and can be enhanced by a liveness analysis to handle pruned and semi-pruned SSA forms. In order to be used in a verified compiler chain, we also show that our validator is correct.

The validation is done in two passes. The first pass performs a structural verification on programs: given an RTL function `f` and an SSA function `tf`, it verifies that `tf` satisfies all clauses of well-formedness except strictness, and that the code of `f` can be recovered from its SSA form `tf` simply by erasing ϕ-blocks and variable indices—the latter property is captured formally by the proposition (`structural_spec f tf`). The second pass relies on a type system to ensure strictness and semantics-preservation. Overall the pseudo-code of the validator is

```
let SSA_validator (f: RTL.function) (tf: SSA.function) (Γ: gtype) : bool :=
  if    (check_blocks_are_wf tf)            (* ensures block_are_wf tf *)
     && (check_blocks_are_at_jp tf)         (* ensures block_at_jp tf *)
     && (check_normalized tf)               (* ensures normalization *)
     && (check_unique_def tf)               (* ensures unique_def tf *)
     && (check_structural_spec f tf)        (* ensures structural_spec f tf *)
  then (is_well_typed f tf Γ)
  else false
```

where `is_well_typed f tf` returns `true` when the function is well-typed with respect to the typing Γ (defined below) in our type system for SSA.

5.6.1 Type system

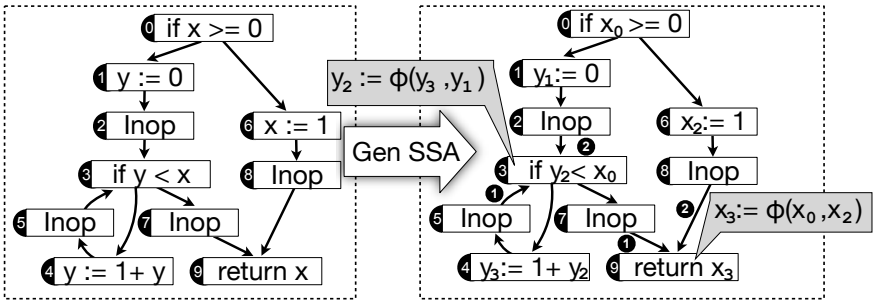
The basic idea of our type system is to track for each variable its *most recent* definition; this is achieved by assigning to all program points a local typing, i.e., an element of `ltype` = `RTL.reg` $\rightarrow$ `idx`; we let γ range over local typings. Then, the global typing of an SSA function `tf` is an element of `gtype` = `node` $\rightarrow$ `ltype`; we let Γ range over global typings. The type system is structured in three layers. The lowest layer checks that RTL-like instructions make a correct use of variables. The middle layer checks that CFG edges are well-typed. Finally, the third layer of the type system defines the notion of well-typed function.

Throughout this section, we use Figure 5.9 as a running example. It provides an RTL example program, and its pruned SSA form. The table on the right gives a typing information, and the result of a liveness analysis.

5.6.1.1 Liveness

As explained in Section 5.2, a liveness information can be used to minimize the number of ϕ-functions in an SSA program. Specifically, ϕ-blocks need to assign only live variables. Hence, our type system is parametrized by a function `live` of type `Liveness` := `node` $\rightarrow$ `Regset.t`, mapping CFG nodes to sets of registers, modelling a liveness analysis result: (`live i`) is the set of registers that are live at the entry of node `i`.

Formally, the type system does not need to know much about the liveness information, and how it is computed. We only demand that the `live` function satisfy two properties: (i) if a variable is used at a program point, then it should be live at this point and (ii) a variable that is live at a given program point is, at the predecessor point, either live or assigned. For a function `f`, the conjunction of these two properties is denoted by the Coq record (`wf_live f live`):



pc	(Γ pc x)	(Γ pc y)	(live pc)
0	0	0	{x}
1	0	0	{x}
2	0	1	{x, y}
3	0	2	{x, y}
4	0	2	{x, y}
5	0	3	{x, y}
6	0	0	{ }
7	0	2	{x}
8	2	0	{x}
9	3	0	{x}

Figure 5.9: An RTL program, its pruned SSA form and a valid typing information

```
Record wf_live (f: RTL.function) (live: Liveness):= {
    wf_live_use: ∀ pc x, use_code f x pc → x ∈ (live pc)

    wf_live_incl: ∀ pc pc' x,
        is_edge f pc pc' →  x ∈ (live pc') →
        x ∈ (live pc) ∨ assigned_code f pc x ;
}.
```

Our type system is able to handle different SSA forms through appropriate instantiations of `live`. Our formalization provides support for minimal SSA and pruned SSA forms, respectively by defining (in Coq) `live` as the trivial over-approximation (at each point, all RTL variables are live), and the result of a standard liveness analysis [App98a]. One could also support semi-pruned forms, by instantiating `live` as the result of the block-local liveness analysis of [BCHS98]. All these three liveness information can be proved to be well-formed.

Example 5.1 (Liveness information)**.** *In the last column of the table in Figure 5.9, we give the liveness information calculated about the variables of the initial RTL function. This information will be used by the validator for validating the pruned SSA form of the program in Figure 5.9. For instance, the variable y is live at node 3, since it is used at node 3. This variable is however dead (i.e. not live) at point 1 because it is defined at this point of the program: it is hence redefined before it is used. At point 6, neither x or y are live. Indeed, the variable x is defined at this point (thus redefined before being used at point 9) and the variable y is not used on any path starting at point 6.*

5.6.1.2 Typing rules for instructions

The type system for instructions checks that RTL-like instructions make a correct use of variables, and that they do not redefine parameters; its formal definition is given in Figure 5.10.

Judgments are of the form $\{\gamma\}$ `ins` $\{\gamma'\}$; intuitively, the judgment is valid if each variable `x` is used in `ins` with the index (γ `x`), and γ' maps each variable to its last definition after execution of `ins`. The typing rules are formalized as an inductive relation `wt_instr`; we briefly comment on some rules.

Several rules correspond to instructions that do not define variables, so the input and output local typings are equal. For these rules, one simply checks that the instruction makes a correct use of the variables (through `use_ok`). The typing rule for (`Inop pc`) states that for every local typing γ, (`Inop pc`) makes a correct use of variables. The typing rule for `Icond` checks that the variables used in the guard are consistent with the local typing input.

Typing instructions

```
Definition use_ok (uses:list SSA.reg)(γ:ltype):= ∀ r i, In (r,i) uses→ γ r = i.

Inductive wt_instr: ltype → SSA.instr → ltype → Prop :=
| wt_Inop: ∀ γ s,
    {γ} Inop s {γ}

| wt_Istore: ∀ γ chk addr args s src,
    use_ok (src::args) γ →
    {γ} Istore chk addr args src s {γ}

| wt_Icond: ∀ γ cond args s1 s2,
    use_ok args γ →
    {γ} Icond cond args s1 s2 {γ}

| wt_Ireturn_some: ∀ γ r,
    use_ok [r] γ →
    {γ} Ireturn (Some r) {γ}

| wt_Ireturn_none: ∀ γ,
    {γ} Ireturn None {γ}

| wt_Iop: ∀ γ op args s r i,
    use_ok args γ  →
    i ≠ dft →
    {γ} Iop op args (r,i) s {γ[r ← i]}

| wt_Iload: ∀ γ chk addr args s r i,
    use_ok args γ →
    i ≠ dft →
    {γ} Iload chk addr args (r,i) s {γ[r ← i]}

| wt_Icall: ∀ γ sig args s id r i,
    use_ok args γ →
    i ≠ dft →
    {γ} Icall sig id args (r,i) s {γ[r ← i]}
```

Typing edges

```
Inductive wt_edge (f:SSA.function)(Γ:gtype)(live:Regset.t):node→ node → Prop:=
| wt_edge_not_jp: ∀ i j ins
    (NOTJP: fn_code f i = Some ins ∧ fn_phicode f j = None)
    (WTI:   {Γ i} ins {Γ j}),
    wt_edge f Γ live i j

| wt_edge_jp: ∀ i j ins block
    (JP:  fn_code f i = Some ins ∧ fn_phicode f j = Some block)
    (USES:∀ args r k, In (Iphi args (r,k)) block → phiuse_ok r args (preds f j) Γ)
    (ASSIG: ∀ r k, assigned (r,k) block → r ∈ live ∧ (Γ j r) = k ∧ k ≠ dft)
    (NASSIG: ∀ r, (∀ k, ¬ (assigned (r,k) block)) → (Γ i r = Γ j r) ∨ r∉live),
    wt_edge f Γ live i j.
```

Typing functions

```
Definition wt_function (f:SSA.function)(Γ:gtype)(live:Liveness): Prop:=
    (∀ i j, is_edge f i j → wt_edge f Γ (live j) i j)
  ∧ (∀ i r, fn_code f i = Some (Ireturn r) → {Γ i} Ireturn r {Γ i})
  ∧ (∀ p, In p (fn_params f) → ∃ r, p = (r, Γ (fn_entrypoint f) r)).
```

Figure 5.10: Type system

In the case of the instruction `Iop`, which defines the variable $(\mathtt{r}, \mathtt{i})$, the output local typing is $\gamma[\mathtt{r} \leftarrow \mathtt{i}]$, i.e. the input local typing updated for the initial variable `r`. From this program node onwards, the new version for `r` is the one indexed with `i`, and this is the one that should be used later on, until another version for `r` is defined.

Note that each time a variable is defined, we demand its index to be different from the index `dft` assigned to parameters at the onset of the program (in the example of Figure 5.9, the default index is 0). This prevents for parameters to be redefined during execution, which would violate the unique definition property.

Example 5.2 (Typing instructions)**.** *We illustrate the typings of instructions with Figure 5.9. Consider the input local typing at point* 3*. The uses of* $\mathtt{x}_0$ *and* $\mathtt{y}_2$ *are consistent with it, since* $(\Gamma\ 3\ \mathtt{x}) = 0$ *and* $(\Gamma\ 3\ \mathtt{y}) = 2$*. The definition of* $\mathtt{x}_2$ *at node* 6 *makes the local typing change for variable* `x` *between nodes* 6 *and* 8*: it changes from* $(\Gamma\ 6\ \mathtt{x}) = 0$ *to* $(\Gamma\ 8\ \mathtt{x}) = 2$*.*

5.6.1.3 Typing rules for edges and functions

The typing rules for edges ensure that ϕ-blocks make a correct use of definitions with regard to a global typing Γ. There are two rules—modelled by the clauses of the inductive relation `wt_edge` in Figure 5.10.

The first rule considers the case where the edge does not end in a junction point; in this case, typing the edge is equivalent to typing the corresponding instruction. The second rule considers the case where the edge ends in a junction point: the typing rule checks the ϕ-block attached to it—structural constraints impose that the instruction is an `Inop`, so we do not need to type-check the instruction. There are three constraints:

- `USES` ensures that the ϕ-arguments `args` passed to ϕ-functions are consistent with all incoming local typings: its kth argument should be the version of the initial variable brought by the kth predecessor of the join point. We omit the formal definition of `phiuse_ok`.
- `ASSIG` ensures the output local typing is consistant with the definitions in the ϕ-block.
- `NASSIG` ensures that, if the variables is not assigned in the ϕ-block, then it means that either it is dead, or the incoming indices for this variable are the same for all predecessors.

Example 5.3 (Typing ϕ-functions)**.** *In Figure 5.9, the ϕ-function for* `x` *at point* 9 *makes correct uses of it because its first argument* $\mathtt{x}_0$ *matches* $(\Gamma\ 7\ \mathtt{x}) = 0$ *and* $\mathtt{x}_2$ *matches* $(\Gamma\ 8\ \mathtt{x}) = 2$*. The local typing at node* 9 *takes into account the definition of* $\mathtt{x}_3$ *in the block by setting* $(\Gamma\ 9\ \mathtt{x})$ *to* 3*. Moreover, no ϕ-function is required for* `y` *at node* 9 *since* $\mathtt{y} \notin (\mathtt{live}\ 9)$*, and no ϕ-function is required for* `x` *at node* 3*, since* $(\Gamma\ 2\ \mathtt{x}) = (\Gamma\ 5\ \mathtt{x})$*.*

Finally, a function is well-typed with regard to global typing Γ if the local typing induced by Γ at the entry node `fn_entrypoint` is consistent with the parameters, and all edges and return instructions are well-typed. Return instructions do not correspond to any edge, we thus need to add this constraint explicitely.

5.6.2 The type system ensures strict SSA form

All SSA programs accepted by the type system are in strict SSA form. It follows that only well-formed SSA functions will be accepted by the validator.

```
Theorem wt_strict: ∀ f tf Γ live,
  wf_live f live →
  wt_function f tf Γ live →
  ∀ (xi: SSA.reg) (u d: node), use tf xi u → def tf xi d → dom tf d u.
```

The proof of `wt_strict` relies on two auxiliary lemmas about local typings for well-typed functions. The first lemma states that if a variable x_i is used at node u, then it must be that (Γ `u x` = `i`). The second lemma states that, whenever (Γ `pc x` = `i`), the definition point of variable x_i dominates pc.

```
Lemma use_gamma : ∀ f tf Γ live,
  wf_live f live →
  wt_function f tf live Γ →
  ∀ x i u, use tf (x,i) u → Γ u x = i.

Lemma def_gamma : ∀ f tf Γ live,
  wf_live f live →
  wt_function f tf live Γ →
  ∀ x pc i d, Γ pc x = i → def tf (x,i) d → dom tf d pc.
```

Under the hypotheses of `wt_strict`, suppose that x_i is used at point u and defined at point d. By `use_gamma`, we get that $(\Gamma\ u\ x) = i$. We conclude by applying `def_gamma` to get that d dominates u.

5.6.3 Soundness of the type system

The SSA generation phase, as any other phase of a formally verified compiler must be proved correct in the sense that all behaviors of the SSA form `tf` are also behaviors of the corresponding initial RTL program `f`. Here, `tf` is generated by the untrusted generator and validated a posteriori, we thus have to prove that if the validator accepts an RTL program `f` and an SSA form `tf`, then all behaviors of `tf` are also behaviors of `f`.

CompCert already provides the general result that a lock-step forward simulation implies preservation of behaviors (see Section 5.3.2). It is thus sufficient to exhibit such a simulation:

```
Theorem validator_correct : ∀ (prog:RTL.program) (tprog:SSA.program),
  SSA_validator prog tprog = true →
    ∀ s1 t s2, RTL.step (genv prog) s1 t s2 →
    ∀ s1', s1 ≃ s1' → ∃s2', SSA.step (genv tprog) s1' t s2' ∧ s2 ≃ s2'.
```

where the binary relation $\simeq$ between semantic states of RTL and SSA carries the invariants needed for proving behavior preservation.

5.6.3.1 Simulation relation

In particular, $\simeq$ should track the correspondence between the registers of semantics states. To do so, we need to capture the semantics of local typings, that specify the correspondence between the variables of `f` and `tf`. This corresponds to the following property:

```
Definition agree (γ:ltype) (rs:RTL.regset) (rs':SSA.regset) (live:Regset.t):=
  ∀ r, r ∈ live → rs#r = rs'#(r, γ r).
```

This intuitively means that the value of an initial RTL register `r` is equal to the value of its current version $(\mathtt{r}, \gamma)$ (determined by the local typing γ) in the SSA function. The idea is then to require that, after each computation step, the register states of the RTL and SSA functions agree, with respect to the local typing at the current program point. Note that we will be

able to prove such a correspondence only for live variables, and that it is actually sufficient for proving behavior preservation.

Now, defining $\simeq$ only in terms of agreement is not enough to make the proof of simulation go through. We have to constrain more the way RTL and SSA states match. For instance, matching states should have the same memory states and stack pointers. Further, their program counters should be equal. Finally, we add locally to the relation $\simeq$ other invariants relative to the function descriptions of semantic states (e.g. the well-formedness of the SSA function and the well-typedness of the pair of functions).

Formally, the $\simeq$ relation is defined with the inductive relation `match_states` below, where we omit the case for relating semantic states of function calls.

```
Inductive match_states : RTL.state → SSA.state → Prop :=
  | match_states_reg: ∀ s f sp pc rs m ts tf rs' Γ live
      (STACKS: match_stackframes s ts)
      (SPEC:   wt_function f tf Γ live)
      (SSA:    wf_ssa_function tf)
      (LIVE:   wf_live f live)
      (AGREE:  agree (Γ pc) rs rs' live),
      (RTL.State s f sp pc rs m) ≃ (SSA.State ts tf sp pc rs' m)

  | match_states_return: ∀ s v m ts
      (STACKS: match_stackframes s ts),
      (RTL.Returnstate s v m) ≃ (SSA.Returnstate ts v m)

   where "s≃t" := (match_states s t).
```

Note that we also define a matching relation for stackframes. It basically lifts the invariants of the current functions to the pair of whole callstacks. This way, at each function call return, the invariants for the caller are available through the matching relation over the stackframes of the callees. This avoids to define (rather clumsily) a global hypothesis on the pair of whole RTL and SSA programs stating the invariants hold for all the functions composing the programs.

5.6.3.2 Proof sketch

The proof proceeds by nested case-analysis on the kind of semantic state of `s1`, the relation $\simeq$, and intruction at the program point under consideration. We treat here the main cases, corresponding to the instructions (i) `Iop` and (ii) `Inop` when a ϕ-block is attached at its successor point. Consider $\texttt{s1} = (\texttt{RTL.State s f sp pc rs m})$ and $\texttt{s1}' = (\texttt{SSA.State ts tf sp pc rs}' \texttt{ m})$, such that $(\texttt{agree } (\Gamma\ \texttt{pc})\ \texttt{rs rs}'\ (\texttt{live pc}))$.

- Suppose $(\texttt{Iop op args res pc}')$ is the instruction at `pc` in `f`. Hence, `f` makes a step towards the state $\texttt{s2} = (\texttt{RTL.State s f sp pc}'\ (\texttt{rs}\#\texttt{res} \leftarrow \texttt{v})\ \texttt{m})$. By the hypothesis $(\texttt{structural_spec f tf})$, we know that there is, at point `pc` in `tf`, an instruction $(\texttt{Iop op args}'\ (\texttt{res}, \texttt{i})\ \texttt{pc}')$, and syntax normalization ensures that $\texttt{pc}'$ is not a junction point. Hence, no ϕ-block is attached to it in `tf`: the matching state is thus $\texttt{s2}' = (\texttt{SSA.State ts tf sp pc}'\ (\texttt{rs}'\#(\texttt{res}, \texttt{i}) \leftarrow \texttt{v})\ \texttt{m})$. In fact both expressions defined by `op` and respectively `args` and $\texttt{args}'$ evaluates to the same value `v`: first, the instruction is well-typed, so that it makes correct uses of its variables, with regard to $(\Gamma\ pc)$. Second, `rs` and $\texttt{rs}'$ agree w.r.t $(\Gamma\ \texttt{pc})$. All uses are live, by hypothesis on `live`. Finally, resulting states are still in the relation $\simeq$, since the update of the local typing specified by the typing rule of the edge $(\texttt{pc}, \texttt{pc}')$ takes into account the actual update of the register states in the semantic step.

- Suppose now (`Inop` $\mathtt{pc}'$) is the instruction at `pc` in `f`, with $\mathtt{pc}'$ a junction point. In this case, $\mathtt{s2} = (\mathtt{RTL.State\ s\ f\ sp\ pc'\ rs\ m})$. We take the following matching state $\mathtt{s2}' = (\mathtt{SSA.State\ ts\ tf\ sp\ pc'\ (phi_store\ k\ p\ rs')\ m})$ where `p` is the ϕ-block at $\mathtt{pc}'$ and `k` is such that $\mathtt{index_pred\ tf\ pc\ pc'} = \mathtt{Some\ k}$. To show the resulting states stay in the relation, we prove that executing a ϕ-block preserves the agreement between register states (as long at the edge $(\mathtt{pc}, \mathtt{pc}')$ is well typed). Let `x` be an RTL variable that is live at $\mathtt{pc}'$. Then, we know that it is live at `pc`, by the definition of `wf_incl` and normalization.
  If no version of `x` is assigned in the block, then we use the agreement between `rs` and $\mathtt{rs}'$ at `pc`. Otherwise, we reason similarly than in the case of `Iop`. We first use hypothesis `ASSIG` in Figure 5.10: we have to show that variable `x` and $(\mathtt{x}, (\Gamma\ \mathtt{pc}'\ \mathtt{x}))$ have the same value in the new register states, and this is the case, thanks to constraints we impose on the format of ϕ-blocks, as well as the hypothesis `USES` in Figure 5.10: if the `k`th argument of the ϕ-function is $(\mathtt{x}, \mathtt{j})$, then it means that $(\Gamma\ \mathtt{pc}\ \mathtt{x}) = \mathtt{j}$, and we can conclude using the agreement of register states at `pc`.

All other cases are treated similarly in the full formalization, except for executing a function call and return. At function call, we have to prove a partial invariant about the caller (that holds just before calling the function), and the invariants for the callee. The former will then be used at the callee's return.

5.6.4 Completeness of the type system

An essential property of our type system is that it accepts all the SSA programs generated by the algorithm by Cytron et al. [CFR+91].

Theorem 5.1 (Type system completeness)**.** *Let* `f` *be a normalized RTL program and let* `tf` *be the SSA program generated from* `f` *by Cytron et al.'s algorithm. Then there exists* Γ *such that* `SSA_validator f tf` Γ `= true`.

Proving this theorem requires to identify some key properties about the algorithm presented in [CFR+91]. Given such a specification, from an SSA function generated by this algorithm, we build a witness global typing Γ_{wit}. We explain the construction below. Finally, we show that the SSA function is typable with Γ_{wit} in our type-system. We consider all edges (i, j) in the CFG of `tf`, and have to prove that the property (`wf_edge f` Γ `live i j`) holds, given a correct and well formed `live` information.

The proof is detailed in Appendix 8.2. It is not formalized in the Coq proof assistant. It would require formalizing the specification in Coq, and proving that the actual running algorithm satisfies this specification. Hence, we would not need to run the validator anymore: by the soundness of our type system, we could deduce a full correctness proof of the SSA generation algorithm a la Cytron.

Building a witness global typing Let `f` be an RTL function, and `tf` the SSA form generated by Cytron et al.'s algorithm. We explain now how to build a global typing Γ by a depth-first-search (DFS) traversal of the CFG of `tf`. Each time we reach a new program point j in the DFS, one of its predecessors i in the CFG has already been treated and $(\Gamma\ i)$ is already defined. To define $(\Gamma\ j)$, we distinguish two cases:

- If j is not a join point, for every RTL variable x, we define $(\Gamma\ j\ x)$ by case analysis:

  - if no instance of x is assigned at i in `tf`, then we set $(\Gamma\ j\ x) = (\Gamma\ i\ x)$
  - if some instance x_k of x is assigned at i in `tf`, then we set $(\Gamma\ j\ x) = k$
- Otherwise, we define $(\Gamma\ j\ x)$ by case analysis on the ϕ-block b at j
  - if no instance of x is assigned in b, then we set $\Gamma\ j\ x = \Gamma\ i\ x$
  - if some instance x_k of x is assigned in b then we set $(\Gamma\ j\ x) = k$

The global typing given in Figure 5.9 can actually be computed using this construction.

5.6.5 Implementation

For the sake of clarity, we have described a non-executable type checker which assumes that structural constraints are satisfied. For efficiency reasons, the Coq implementation of the type system is in fact more complex. In particular, it performs type inference rather than type checking. Additionally, it performs a single, linear scan of the program, and checks the list of arguments of ϕ-functions only once per junction point, rather than once per incoming edge for a given join point. On the benchmarks given in Section 5.9, our implementation is ten times faster than a type checker derived naively from the non-executable type system of Figure5.10. We now give an overview of the implementation.

The untrusted SSA generator does not actually compute the whole code of the SSA form of a function. It provides to the type checker the information that is strictly necessary. We will call this information a hint; it is made of two maps. The first maps indicates for each CFG node, the instance index this node potentially defines. The second map provides the same kind of information but for ϕ-blocks: at a given node, it indicates whenever a block is required, and what index to use for the definition of variables. The signature of the external generator for SSA is thus the following:

```
Definition SSA_hint := (PTree.t index) * (PTree.t (PTree.t index)).
Variable extern_SSA_gen: RTL.function → Liveness → SSA_hint.
```

Given this hint, the type inferencer performs both the type inference and the code generation:

```
Definition type_infer: RTL.function → Liveness → SSA_hint → option SSA.function.
```

Since the hint might be incorrect, the type inferencer may not be able to generate any SSA function, hence the option type of its result. This type inference builds a global typing Γ using the SSA hint, in a way that is similar to the algorithm described in Section 5.6.4.

We then prove that, whenever the inference is successful, the generated function is well typed in the type system described in Figure 5.10:

```
Theorem type_infer_correct: ∀ f tf live hint,
   wf_live f live →
   type_infer f live hint = Some tf →
   ∃ Γ, SSA_validator f tf Γ = true.
```

Finally, our SSA generation algorithm is described by the following snippet.

```
Definition ssa_gen (f: RTL.function) : option SSA.function :=
  let live := (LiveAnalysis f) in
  let hint := extern_gen_ssa f live in
  type_infer f live hint.
```

First, a liveness analysis (implemented in Coq) is performed on the RTL function whose result is shared by the external untrusted SSA generator (written in Ocaml) and the type inferencer.

The external SSA generator computes the hint required for the type inferencer to perform the actual SSA code generation, whilst verifying the validity of the hint.

5.7 SSA-based optimizations and the equational lemma

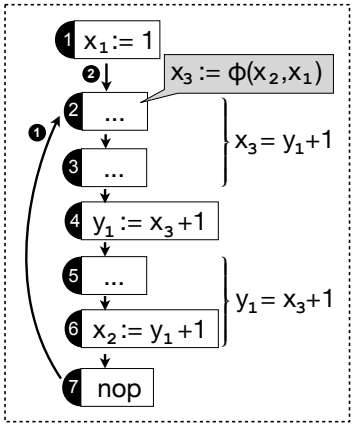
In this section, we introduce the *equational lemma* that supports the view of programs in SSA form as *systems of equations*. We then illustrate how to reason about a simple SSA-based optimization, Copy Propagation. We also formalize and prove correct a GVN optimization.

5.7.1 Equational lemma

The SSA form provides an intuitive reading of programs: one can view the unique definition of a variable as an equation, and by extension SSA programs as systems of equations:

> *Because every assignment creates a new value name it cannot kill (i.e. invalidate) expressions previously computed from other values. In particular, if two expressions are textually the same, they are sure to evaluate the same result. [BM94]*

We illustrate this with an example program. For instance, the definitions of $\mathtt{x_3}$ and $\mathtt{y_1}$ respectively induce the two equations $x_3 = y_1 + 1$ and $y_1 = x_3 + 1$. There is however a pitfall: the two equations entail $x_3 = x_3 + 2$, and thus are inconsistent. In fact, equations are only valid at program nodes dominated by the definition that induce them, as captured formally by the *equational lemma* of SSA:




```
Lemma equation_lemma: ∀ prog d op args x succ f m rs sp pc s,
    wf_ssa_program prog →
      reachable prog (State s f sp pc rs m) →
      fn_code f d  = Some (Iop op args x succ) →
      sdom f d pc →
      eval_operation sp op (rs##args) m = Some (rs#x).
```

where `reachable` is a predicate that defines reachable states. In practice, it is often convenient to rely on a corollary that proves the validity of the defining equation of `x` at program points where `x` is used – thus avoiding reasoning on the dominance relation. The formal statement of the corollary is obtained by replacing the hypothesis (`sdom f d pc`) by the hypothesis (`use f x pc`); the proof of the corollary intensively uses the strictness property of well-formed SSA programs.

5.7.2 Application to Copy Propagation

We conclude with a succinct account of applying the corollary to prove the soundness of copy propagation. This optimization searches for copies `x` := `y` and replaces every use of `x` by a use of `y`. On the SSA form, this can be done by simply walking through the program, identifying statements of the form $x := y$, and replacing every use of x by y.

Indeed, suppose `pc` is a program point where such a replacement has been done. Every time `pc` is reached during the program execution, we are able to derive, using the corollary, that `rs#y` = `rs#x`, where `rs` is the current register state because (i) `y` is the right hand side of the definition of `x` and (ii) `pc` was a use point of `x` in the initial program.

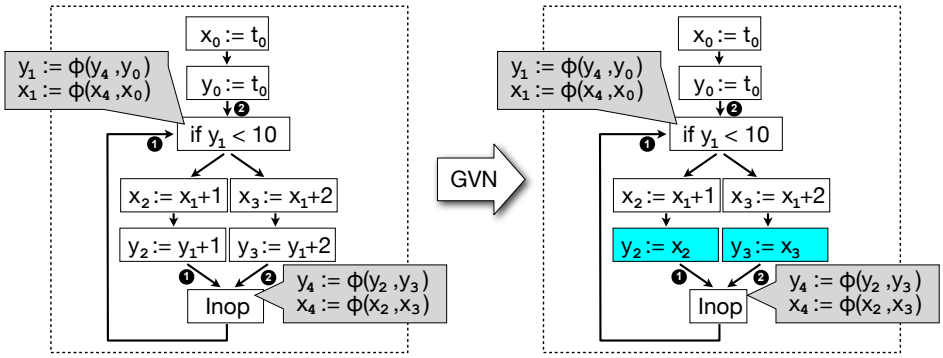


Figure 5.11: Common Sub-expression Elimination (CSE) using GVN

On non-SSA forms, the reasoning is more involved since one has to prove that the reaching definition for `x` is unique at `pc`, and that no redefinition of `y` can occur in between.

5.7.3 Validation of Global Value Numbering

Common sub-expression elimination based on *Global Value Numbering* (GVN) [AWZ88] is a typical SSA-based optimization. GVN assigns to variables an identifying number such that variables with the same number will hold equal values at execution time. The effectiveness of GVN lies in its ability to compute efficiently numberings that identify many variables. Advanced algorithms [AWZ88, BCS97] allow computing efficiently numberings; they are generally presented as highly optimised iterative algorithms.

GVN-based CSE Figure 5.11 illustrates how GVN can be used to eliminate redundant computations. The program on the left is the original code; in this program, for each i, x_i and y_i are assigned the same value number. Hence, the evaluation of $y_1 + 1$ (resp. $y_1 + 2$) is a redundant computation when assigning y_2 (resp. y_3). Thus, one can transform the program into the semantically equivalent one shown on the right of the figure. The strength of the analysis lies in its ability to reason about ϕ-functions, which allows it to infer the equality $x_2 = y_2$. This is only possible because numbering is global to the whole program. In fact, any block-local analysis would fail to discover the equality $x_2 = y_2$.

Validating GVN-based CSE We follow [AWZ88] but clearly separate the optimisation into two phases. First, an untrusted analysis, written in OCaml, computes a numbering for SSA programs. For each program point where the numbering detects a redundant computation $x := e$, it provides a candidate y for replacing the previous operation by $x := y$. In a second phase, a validator checks the numbering and the proposed assignment simplification.

The analysis computes a fixpoint in the domain of congruence partitions, where partitions are modelled as mappings $\mathcal{N} : \texttt{reg} \to \texttt{reg}$ that map a register to the canonical register of its equivalence class (its *number*). This abstract domain is ordered w.r.t. reverse inclusion of equivalence kernels. The equivalence kernel of $\mathcal{N}$ is the relation $\sim_{\mathcal{N}}$ defined by $\texttt{x} \sim \texttt{y}$ if and only if $\mathcal{N}\ x = \mathcal{N}\ y$.

Viewing the result of the analysis as a post-fixpoint is the key to our second component, a validator that checks whether a numbering $\mathcal{N}$ is indeed a post-fixpoint of the analysis on a

program p, and if so returns an optimized SSA program tp. The validator is programmed in Coq, and proved to ensure behavior preservation between the original and optimized programs.

Correctness of the numbering The notion of valid numbering is formally defined by

```
Inductive ≡^N : reg → reg → Prop :=
| GVN_refl : ∀ x, ≡^N x x

| GVN_Iop : ∀ x y pc1 pc2 op args1 args2 pc1' pc2'
    fn_code f pc1 = Some(Iop op args1 x pc1') →
    fn_code f pc2 = Some(Iop op args2 y pc2') →
    same_number N args1 args2 →
    ≡^N x y

| GVN_Phi : ∀ x y pc args_x args_y
    fn_phicode f pc = Some phib →
    (Iphi args_x x) ∈ phib →
    (Iphi args_y y) ∈ phib →
    same_number N args_x args_y →
    ≡^N x y.

Definition GVN_spec (N:reg → reg) : Prop :=
   (∀ x y, N x = N y → param f x → param f y → x=y)
  ∧ (∀ x y, N x = N y → ≡^N x y).
```

First, we define for each numbering $\mathcal{N}$ the relation $\equiv^{\mathcal{N}}$ as the smallest reflexive relation identifying: (i) registers whose assignments share the same operator and corresponding arguments are equivalent w.r.t. $\mathcal{N}$ (predicate same_number checks that each pair of arguments have the same number for $\mathcal{N}$); (ii) registers defined in the same ϕ-block with equivalent arguments. Then, for a numbering $\mathcal{N}$ to be valid (see GVN_spec), its equivalence kernel must not contain a pair of distinct function parameters and it must moreover be included in $\equiv^{\mathcal{N}}$.

The latter ensures the intended post-fixpoint property: if we write $\sqsubseteq$, the reverse inclusion of equivalence kernels ($\mathcal{N}_1 \sqsubseteq \mathcal{N}_2$ iff $\sim_{\mathcal{N}_1} \supseteq \sim_{\mathcal{N}_2}$) and $\mathcal{N}_{\text{param}}$ the numbering that associates each register to itself if it is a function parameter and a default register otherwise, then (GVN_spec $\mathcal{N}$) is equivalent to $F(\mathcal{N}) \sqsubseteq \mathcal{N}$ with F the operator defined by $F(\mathcal{N}) = \mathcal{N}_{\text{param}} \cap \equiv^{\mathcal{N}}$.

The crux of the correctness proof of the GVN validator is the correctness lemma for a valid numbering: if $\mathcal{N}$ is a valid numbering for f, and rs is a register state that can be reached at node pc, and x and y are two registers whose definition strictly dominate pc, then $\mathcal{N}$ x $= \mathcal{N}$ y entails that rs holds equal values for x and y. In Figure 5.11, suppose the analysis infers the same number for registers x_2 and y_2; they are indeed equal just after the assignment of y_2 but not before.

```
Lemma valid_numbering_correct : ∀ prog s sp pc rs m,
    wf_ssa_program prog →
    GVN_spec N →
    reachable prog (State s f sp pc rs m) →
    (∀ x y: reg, def_sdom f x pc → def_sdom f y pc → N x = N y → rs#x = rs#y).
```

Predicate (def_sdom f x pc) states that the definition of x in f strictly dominates pc. The definition of def_sdom given below takes care of the case where x is assigned in a ϕ-block at pc. Indeed, a normal assignement at pc takes effect after leaving pc, but a ϕ-block at pc is actually executed before reaching pc. Thus, the equality of variable number for ϕ-function destination registers can be taken into account at the node holding the ϕ-block. The proof of that lemma makes an extensive use of the equational lemma presented in the previous section.

```
Inductive def_sdom (f:function) (x:reg) (n:node) : Prop :=
| def_sdom_def_sdom : ∀ def_x,
    def f x def_x →
    sdom f def_x n →
    ¬ assigned_phi f n x →
    def_sdom f x n

| def_sdom_def_phi :
    assigned_phi f n x →  def_sdom f x n.
```

The optimization The implementation of the GVN-based CSE takes as input a numbering $\mathcal{N}$, and a partial mapping `Repr` that, given a register `x` and node `pc` returns, if it exists, a representative of the class of `x`, i.e. a register `y` such that (i) `x` and `y` are related by the equivalence kernel of $\mathcal{N}$ and (ii) the definition of `y` strictly dominates `pc`.

For efficiency reasons, we do not check the correctness of `Repr` a priori, but lazily during the construction of the optimized program. The optimizer proceeds as follows: first, it checks whether $\mathcal{N}$ satisfies the predicate `GVN_spec`. Then, for each assignment (`Iop op args x pc`) of the original SSA program, the optimizer checks whether `Repr` provides a canonical representative `y` for `x` at node `pc`. If so, it checks whether the definition of `y` strictly dominates `pc`; this is achieved by means of a dominance analysis, computed directly inside Coq with a standard dataflow framework *a la* Kildall. Provided `y` is validated, we can safely replace the previous instruction by a move from `y` to `x`.

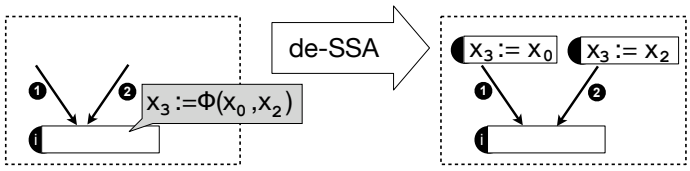
We conclude by commenting briefly on the soundness proof of the transformation. It follows a standard forward simulation proof where the correctness of the numbering is proved at the same time as the simulation itself. Noticeably, the CFG normalization turned out to be extremely valuable for this proof. Indeed, consider a step from node `pc` to node `pc`$'$: we have to prove that (`gamma` $\mathcal{N}$ `pc`$'$ `rs`) holds, asumming (`gamma` $\mathcal{N}$ `pc rs`). We reason by case analysis: if the instruction at `pc` is not an `Inop` instruction, we know by normalization that `pc`$'$ is not a junction point. In this case, (`def_sdom f x pc`$'$) is equivalent to (`def_sdom f x pc`) $\vee$ (`def f x pc`) which is particularly useful to exploit the hypothesis that (`gamma` $\mathcal{N}$ `pc rs`) holds.

5.7.4 Discussion

We have introduced the equational lemma, and demonstrated how it can be used to prove SSA-based optimizations. This lemma is powerful, and greatly simplifies the proofs. Still, proving an SSA-based optimization phase in our middle-end is not immediate. In particular, there are some dominance relations to discharge. Here, we rely on the strictness of the SSA form and on several lemmas about the dominance relation we have prove separately. For Copy Propagation and our GVN-based CSE, the proof follows the same pattern, since a variable assignment `x` := `e` is replaced by a copy `x` := `y` where the definition of `y` necessarily dominates the definition of `x`. Once the transformation has been proved correct, we also have to prove that the well-formedness of functions is preserved. This kind of proof is tedious, but not very hard technically. The general scheme of Copy Progagation and GVN-based CSE does not break any invariant (in particular, we do not modify any ϕ-argument).

5.8 Conversion out of SSA

The final phase of the middle-end converts SSA programs back to RTL programs, so that they can be further processed by the CompCert back-end, starting with register allocation. Several approaches have been proposed [SJGS99, BDR+09]. As a first step, we decided to use the conversion described in [CFR+91]. The basic idea of this conversion is to substitute each ϕ-function with one variable copy at each predecessor of the junction point:



There are several pitfalls to be aware of: performing naively the destruction of SSA by such copy insertions can lead to the non-preservation of behaviors. Two problems were identified by Briggs et al. in [BCHS98]: the presence of critical back-edges (that can lead to the so-called lost-copy problem) and the swap problem. We review both problems and explain how we tackle them in the above sections. We finish this section with an overview of the correctness proofs, where we show how the normalization phase can again be exploited.

5.8.1 Critical edges

In the presence of critical back-edges in the progran CFG, the simple copy insertion described above becomes incorrect. A critical edge is an edge (i, j) whose entry i has several successors and whose exit j has several predecessors. In the case where the sink of the critical edge (i, j) holds a ϕ-block, the copies cannot be inserted at the predecessors, because they would be executed on some paths that initially did not reach the ϕ-function. This can lead to the well-known lost-copy problem in the presence of critical back-edges and optimizations such as copy propagation (see [BCHS98]).

One solution to this problem is to split every critical edge (i, j) into two edges (i, k) and (k, j), so that the copies for replacing the ϕ-function at point j can be safely inserted at node k. Compilers that operate on *basic-block* CFG graphs carefully avoid edge splitting for efficiency concerns in later optimization stages. But this is at the cost of making de-SSA algorithms significantly more complex.

In our case, the normalization we impose on SSA programs pleasingly ensures the absence of critical edges in their CFG. One could fear that the critical edge splitting implied by the normalization could impact later phases of the compiler, but the representation of programs inherited from CompCert deflates this penalty cost. RTL graphs, and thus SSA code graphs, are single-instruction graphs: replacing ϕ-functions with copies automatically splits critical edges by the insertion of code.

5.8.2 The swap problem

One must also take care of the semantics of ϕ-blocks. They are given a parallel semantics, and, because of optimizations, it is not in general equivalent to a sequential interpretation. Indeed, performing copy propagation on SSA can modify the code, so that ϕ-functions argument and destination registers are no longer independent: a variable x_i can appear both as a source

and a target of distinct ϕ-functions in a single ϕ-block. In this case, the copies inserted for converting out of SSA must be sequentialized. This can be done at the reasonable price of inserting at most one temporary variable [RSL08].

In the current state of our development, our conversion out of SSA fails on such ϕ-blocks. This is not a limitation in practice, as the GVN optimization we perform on the code does not cause problems of that kind; from the SSA generation until its destruction, the parallel semantics of ϕ-blocks is ensured to be equivalent to the sequential one. We however plan to reuse the work of Rideau et al. [RSL08] which provides an algorithm for transforming a set of parallel moves into an equivalent sequence of elementary moves (using additional temporaries). This algorithm is already used in CompCert when enforcing calling conventions during the compilation of function calls.

5.8.3 Correctness proof

For proving the transformation correct, we proceed by giving a forward plus simulation (see Chapter 3) between the SSA program and the RTL program after de-SSA. The simulation requires the RTL program to perform several steps to simulate a (big-step) execution of a ϕ-block by the initial SSA program. We also take advantage of the normalization in this proof: the execution of an `Inop` instruction leading to a junction point with a ϕ-block matches the corresponding inserted copies. Without the normalization, all RTL-like instructions would have resulted in a different case in the proof.

5.9 Implementation and experimental results

We have plugged in Compcert 1.8.2 our SSA middle-end made of (i) a Coq normalization (ii) an Ocaml SSA generator and its Coq validator; (iii) an Ocaml GVN inference tool and its Coq validator; (iv) a Coq de-SSA transformation. Our formal development adds 15.000 lines of Coq code and 1.000 lines of Ocaml to the 80.000 lines of Coq and 1.000 lines of Ocaml provided in CompCert. It does not add any axioms to CompCert.

We use the Coq extraction mechanism to obtain an SSA-based verified compiler, that we evaluate experimentally using the CompCert benchmarks. These include around 75.000 lines of C code, and fall into three categories of programs (from 20 to 5.000 LoC): small computation kernels, a raytracer, and the theorem prover Spass. Spass is the largest benchmark with 69.073 LoC. Below we briefly comment on three key points: the efficiency of the SSA validator, the effectiveness of the GVN optimizer and the efficiency of generated code.

5.9.1 Efficiency of the SSA validator

In order to be practical, validators must be more efficient than state-of-the-art implementations of the transformations that they validate. At first sight, this criterion may seem too demanding for SSA, since generation into SSA form is performed in almost linear time. However, experimental results are surprisingly good: overall converting a program into SSA form takes approximately twice longer than type-checking the output program. In more detail, the times for SSA generation—specialized to pruned SSA—distribute as follows: (i) 9% for normalization of RTL; (ii) 37% for liveness analysis of RTL (the liveness analysis is provided in the CompCert distribution); (iii) 35% for conversion to SSA using the untrusted OCaml implementation (based on state-of-the-art algorithms); (iv) 19% for validation using the verified

validator. This distribution appears to be uniform on all benchmarks except on the biggest functions where the liveness analysis exhibits a non-linear complexity.

5.9.2 Effectiveness of the GVN optimizer

We measure the effectiveness of our GVN analyzer by performing a GVN-based CSE right after a (Local Value Numbering) LVN-based CSE implemented in CompCert. We count how many additional `Iop` instructions are optimized by this additional CSE phase. For efficiency concerns about the generated code, we need to keep the LVN phase that optimizes redundant memory loads (currently, this is not done by our GVN optimizer). To keep the comparison fair, we allow CompCert CSE to optimize around function calls—this is disabled in CompCert to keep the register pressure low. The results are given in Table 5.1, for two backends, x86 (left) and PowerPC (right). The overall improvement is significant. Our global CSE optimizes an additional 10% of `Iop` instructions on PowerPC and an additional 25% on x86.

We also measure how the GVN behaves, without the preliminary LVN optimization. Our global CSE manages to optimize all the `Iop` instructions that are optimized by LVN, expect 2 for the small computation kernels, and 1 for the raytracer. For Spass, however, GVN only optimizes half the number of `Iop`. This is due to the fact that in CompCert's LVN, the redundant load elimination and CSE optimizations are interdependent (detecting some redundant loads can in turn help detecting new common sub-expressions, and common sub-expression elimination can lead to new load redundancy detection).

x86	`Iop`	LVN	GVN	GVN only	PPC	`Iop`	LVN	GVN	GVN only
c. kernels	3,494	163	55	216	c. kernels	3,142	422	54	472
raytracer	2,303	131	29	159	raytracer	2,755	303	21	322
spass	51,640	122	19	99	spass	52,451	392	43	306
TOTAL	57,437	416	103	474	TOTAL	58,348	1,117	118	1,100

Table 5.1: **GVN optimizer: results on x86 and PowerPC.** For each set of benchmarks, we count the number of initial `Iop` instructions in the RTL function (column `Iop`), the number of `Iop` optimized away by the LVN-CSE optimization of CompCert (column LVN) and the number of `Iop` optimized away by our GVN-CSE optimization, right after CompCert's LVN-CSE (column GVN). We also measure the number of `Iop` that GVN optimizes away without any prior LVN-CSE (column GVN only).

5.9.3 Efficiency of the generated code

To assess the efficiency of the generated code, we have compiled the benchmarks with three compilers: CompCert, our version of CompCert extended with an SSA middle-end (CompCertSSA), and `gcc -O1`. Figure 5.12 gives the execution times *relative* to Compcert (shorter bars mean faster) on PowerPC. The test suite is too small to draw definite conclusions, but the results are encouraging. Our version of CompCert performs slightly better than CompCert. During our experiments, we observed that the computation time of the allocator is sometimes rather long, and can result in a lot of spill code. The quality of the allocation is essentially impacted by our current SSA deconstruction, that introduces many copies and artificial interferences between variables of a ϕ-block, imposing more constraints on the allocator.

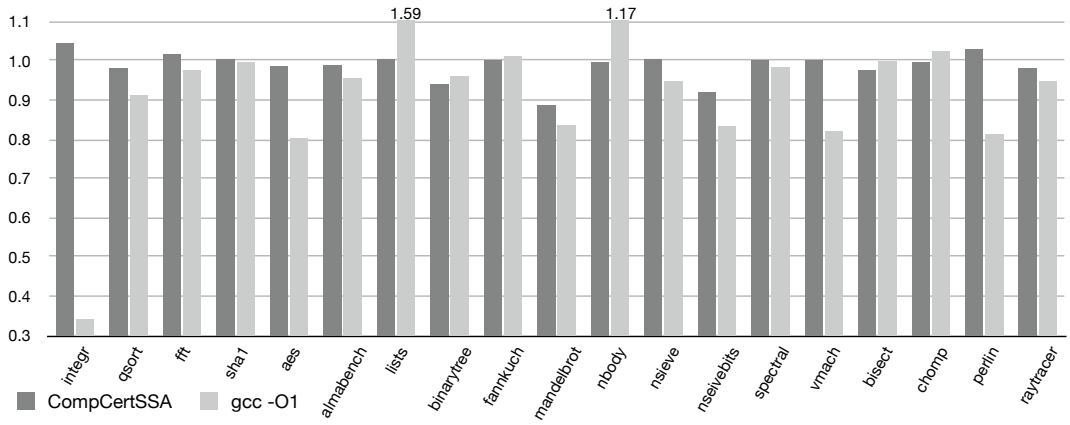


Figure 5.12: Execution times of generated code

We expect that performance improves significantly by enhancing our middle-end with additional optimizations, and by refining our SSA deconstruction, with either techniques similar to [BDR+09] or an SSA-based register allocator [HGG06].

5.10 Related work

We now present closely related work on SSA.

Machine-checked formalizations Blech et al. [BGLM05] use the Isabelle/HOL proof assistant to verify the generation of machine code from a representation of SSA programs that relies on term graphs. While graph-based representations may be useful for the untrusted parts of our compiler, they increase the complexity of the formal SSA semantics, and make it a greater challenge to verify SSA-based optimizations. They do not provide an algorithm to convert into SSA form, and leave as future work proving the correctness of SSA-based optimizations. Mansky and Gunter [MG10] use Isabelle/HOL to formalize and verify the conversion of CFG programs into SSA form. However, their transformation may yield non-minimal SSA, and does not aim extraction into efficient code. Moreover, it is not clear whether their semantics of SSA can be used to reason about optimizations. Zhao et al. [ZNMZ12] formalize the LLVM intermediate representation in Coq. They define and relate several formal semantics of LLVM, including a static and dynamic semantics. They show how simple code motions can be validated with a simulation relation based on symbolic evaluation, and plan to extend the method to other transformations such as dead code elimination or constant propagation. Finally, there are several machine-checked accounts of Continuation Passing Style translations, e.g. [DL07, Chl10], closely related to the SSA form.

Translation validation and type systems Menon et al. [MGM+06] propose a type system that can be used to verify memory safety of programs in SSA form, but their system does not enforce the SSA property. Matsuno and Ohori [MO06] define a type system equivalent to SSA: every typable program is given a type annotation making explicit def-use relations. Their type system is similar to ours except they type check one program w.r.t. annotations

while we type check a pair of an RTL and an SSA program. They show that common optimizations such as dead code elimination and CSE are type-preserving. But they do not prove the semantics preservation of the optimizations. Stepp et al. [STL11] report on a translation validator for LLVM. Their validator uses Equality Saturation [TSTL09], which views optimizations as equality analyses. Their tool does not validate GVN. Tristan et al. [TGM11] independently report on a translation validator for LLVM's inter-procedural optimizations. This tool supports GVN, but is currently not verified.

5.11 Conclusions and future work

The SSA form is a popular IR in the compilation community that has been used with great success in many program optimizations since its inception in the late 80's. The structural properties of unique definition and strictness, as well as the parallel semantics given to ϕ-blocks are the ingredients that led to this success.

If those properties seems rather simple and intuitive, the algorithms underlying the generation of SSA – who actually establish those properties – rely on complex properties of graphs (e.g. the dominator tree or dominance frontiers), that are difficult to justify formally. Moreover, the very semantics of the SSA form has kept for a long time at an informal level. As a consequence, the correctness proof of SSA-related algorithms (i.e. generation, optimizations, and destruction), were until very recently not formally proved correct. Over the past few years, some interesting attempts have be made to formalize the semantics of SSA, but these formalizations were rather distant from the intuitive semantics presented in the seminal papers. The correctness of SSA-based analyses and optimizations is usually proved using structural arguments on the CFG only, and the semantic properties and invariants of SSA remain unclear.

In this chapter, we have defined a formal semantics for SSA, that is both close to the intuitive definition of the early papers, and amenable for formal reasoning, as witnessed by our fully verified SSA-based middle-end for the verified CompCert C compiler. Thanks to our choices made in the representation of programs, this semantics integrates well in the CompCert architecture. The translation validation approach we use for the conversion to SSA and the GVN optimization allows the middle-end implementing state-of-the-art algorithms, while keeping close to the essence of those phases and to the high-level properties they should satisfy in order to preserve the behaviors of programs. The focused nature of our SSA validator makes it complete with regard to one of the reference implementations of the SSA generators [CFR+91], where ϕ-functions placement is determined using dominance-frontiers. We also identified and isolated the semantic counterpart of the structural properties of SSA into a dedicated invariant lemma – that holds at all points of the execution – on which rely the correctness proof of several SSA-based optimizations.

A priority for further work is to achieve a tighter integration of our middle-end into CompCert. Beyond some obvious enhancements of our implementations, proof cleaning, and the porting of our middle-end to an up-to-date version of CompCert, there are three immediate objectives, developped in Chapter 7: (i) enhancing our SSA middle-end to handle memory aliases as done by CompCert's RTL-based middle-end, (ii) implementing an SSA-based register allocator [HGG06], and (iii) verifying more SSA-based optimizations, including PRE [CCK+97].

Chapter 6

Memory model for concurrent Java IRs

Concurrency in Java consists in multiple threads of execution that communicate via a single shared memory. Because of shared memory accesses to instance fields, static fields or array elements, synchronization mechanisms are needed if one wants to avoid *data-races*, i.e. simultaneous memory accesses from different threads, that could lead to unexpected behaviors. At the language level, concurrency boils down to the class `Thread` that provides (native) methods for creating and handling threads: `start` spawns a thread, and `join` waits for a given thread to end. The most basic form of synchronization in Java is the locking mechanism, materialized by the two bytecode instructions `monitorenter` and `monitorexit`[1]. Therefore, from a syntactic point of view, multi-threaded Java IRs only differ from sequential IRs in those extra instructions. On the semantic side however, Java concurrency raises challenges, that we investigate in this chapter.

Semantically, one could intuitively consider that threads of executions are interleaved, and each thread executes following the program order, i.e. the order in which the code is written. This model, defined by Lamport [Lam79] as Sequential Consistency (SC), is to this date what is assumed by most programmers, analyzers or verifiers. We present it below in more detail. Unfortunately, this model does not capture all the executions of multithreaded Java source and IR programs. As we shall see, the real Java semantics seems counter-intuitive for most programmers, and its current definition makes it particularly difficult to reason formally about multi-threaded programs, their optimizations, or analyses. Before going into more detail about those difficulties, we review the SC model.

Sequential consistency The simplest and most natural model of concurrency that programmers have in mind is Sequential Consistency (SC). The threads execute following the program order, and their execution are interleaved. SC is also referred to as an *interleaving semantics*: the semantics of the thread composition is an interleaving of the sequential semantics of individual threads, and all possible interleavings are considered (the semantics is thus non-deterministic). At any point in the execution, an active thread is selected for making a step, which is performed directly on the global memory. Thus, all threads have an up-to-date view of the shared memory.

Consider the simple example program in Figure 6.1a, which is at the core of Dekker's algorithm for implementing a mutual exclusion scheme. Registers r_1 and r_2 are local flags marking whether threads can enter their critical section (0 means that the thread is allowed to enter it). Statements (a) and (c) give rise to memory write actions, and (b) and (d) to memory read actions. There is no synchronization action in this program. The six possible interleavings are given in Figure 6.1b, leading to different values for r_1 and r_2 at the end of the

[1] Java synchronization mechanisms also include `synchronized` methods, and wait-and-notify mechanisms.

$(i)\ x \leftarrow 0; y \leftarrow 0$	
$(a)\ x \leftarrow 1$	$(c)\ y \leftarrow 1$
$(b)\ r_1 \leftarrow y$	$(d)\ r_2 \leftarrow x$

(a) Two threaded program

Interleaving	Final result		Interleaving	Final result	
a-b-c-d	$r_1 = 0$	$r_2 = 1$	a-c-d-b	$r_1 = 1$	$r_2 = 1$
c-d-a-b	$r_1 = 1$	$r_2 = 0$	c-a-d-b	$r_1 = 1$	$r_2 = 1$
a-c-b-d	$r_1 = 1$	$r_2 = 1$	c-a-b-d	$r_1 = 1$	$r_2 = 1$

(b) Possible interleavings and corresponding results

Figure 6.1: Dekker's algorithm with a sequentially consistent semantics

Conventions: variables x and y are shared-memory variables (i.e. instance fields, static fields or array elements in Java), distinct names denote different memory locations. Variables r_i are thread-local. The initial value of shared variables is given by statements (i). In this introduction, the observable behaviors of programs are the possible values of local registers at the end of the execution.

execution. Under SC, Dekker's algorithm is correct, as no interleaving leads to the situation where $r_1 = r_2 = 0$, i.e. the threads cannot enter their critical section simultaneously.

The case of Java As for Java however, the semantic gap that exists between sequential and multi-threaded IRs is important with regard to shared memory accesses (i.e. the memory reads, memory writes and synchronization actions). The Java *memory model* (JMM) [JSR04, MPA05] specifies when values written by some threads can be read from the shared memory by other threads. The JMM cannot be defined in terms of thread interleavings only, due to the complex reorderings performed by the compiler and the underlying hardware: the JMM is *weakly consistent*. In practice, the relaxations appear quite often: if we write the corresponding Java code that iterates the pattern of Figure 6.1a until the configuration $r_1 = r_2 = 0$ is reached, the program can terminate after a few seconds. In other words, a naive Java implementation of Dekker's algorithm is broken.

Because their semantics is hard to understand, implementing multi-threaded Java programs is error-prone. Worse, catching bugs in such implementations is harder than in a sequential setting, as program executions are not easily reproducible. Formal reasoning about the IRs of such programs is thus absolutely necessary. Apart from their extreme complexity, even for experts [Šev08], the existing formalization of the JMM [MPA05] is known to be flawed on several points. Even if it would be formally fixed, its definition is not amenable to formal reasoning for several reasons. First, the specification of the JMM is axiomatic, while an operational definition is preferable to prove formally the correctness of compiler optimizations (e.g. it comes with proof techniques such as the simulation diagrams we presented in Chapter 3). Second, it is formulated differently from the axiomatic hardware memory models that have been formalized so far. This makes it difficult to match the two models. Third, this definition is not intuitive: valid executions must be justified with respect to a complex causality order imposed on the data-races. We argue that a more intuitive semantics of programs with data-races is preferable, as those are the more likely to contain (un)intentional bugs.

There are two main reasons for the JMM definition to be so complex. First, it is supposed to give a semantics to all programs, including those containing data-races. Second, it is meant to be architecture-independent, following the portability requirement of Java. As a consequence, it is meant not to over-restrict a wide range of compilers and hardware, in terms of optimizations and performance. We cannot compromise with the need to define a semantics for all programs, but we can investigate on the second source of generality of the JMM.

In this chapter, we define a Java memory model specialized to the Total Store Order

(TSO) architectures [Int12, Int92]. This model, close to TSO, is easier for the programmer to understand, and amenable to formal reasoning, thanks to its operational characterization. We also formally characterize it in terms of the reorderings it allows. We exploit here the approach taken by Ševčík et al. [ŠVZN+11] to factor out a proof of such a result common to all the IRs used by the compiler.

Before going into more detail, we give in the next section an informal introduction to the vast field of weak memory models, giving some insight about the concepts that we will use in the remainder of the chapter.

6.1 Introduction to weak memory models

SC is often what is assumed by the programmer, and by most of the existing verification techniques and tools (see [Rin01] for a survey). But if we write the x86 assembly code corresponding to the program of Figure 6.1a, its execution on a modern multiprocessor architecture might actually yield the configuration $r_1 = r_2 = 0$. The dyi tool [AMSS10] reports that, on an Intel Xeon, this occurs 2 times among 1000000 runs[2]. The reason is that modern hardware memory models are weaker than SC.

We explain in Section 6.1.1 how modern multiprocessor hardware features led to the notion of weak memory models. We discuss in Section 6.1.2 the various extents to which a language memory model can reflect this weak consistency, and the different trade-offs one has to consider. This will progressively lead us to the most general approach – the one taken by the JMM. The limits of this model are further described in Section 6.1.2.4.

6.1.1 Hardware memory models

6.1.1.1 Relaxing SC

For efficiency reasons and hiding the memory access latency, modern multiprocessor architectures rely on memory caches and write buffers or instruction pipelining [HP11], leading to a potential out-of-order execution of instructions. Lamport reports in [Lam79]

> *For some applications, achieving sequential consistency may not be worth the price of slowing down the processors. In this case, one must be aware that conventional methods for designing multiprocess algorithms cannot be relied upon to produce correctly executing programs.*

In other words, these features make the memory model exhibit non-SC behaviors. Multiprocessors provide some synchronization and memory barrier instructions for managing thread communication. These mechanisms restrict the bufferings, instruction reorderings or caching scenario that may occur in code fragments surrounded by such instructions. We give an example in Section 6.1.1.2.

Existing families of processors propose various memory models that differ in subtle ways one from another. For a long time, the description of architectures memory models provided by the vendors have stayed in an informal prose. Some progress have been done on providing formal specifications (see e.g. [AG96, SSO+10, SSA+11]), but this is still an active field of research (see Section 6.9).

[2] http://diy.inria.fr/doc/SB.log

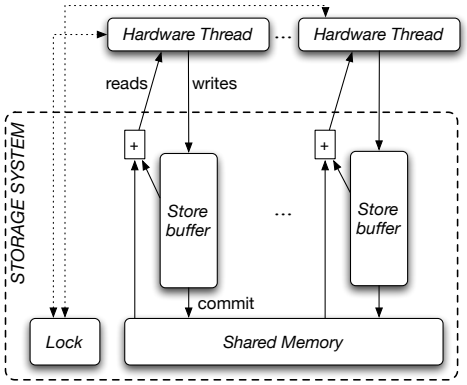


Figure 6.2: x86-TSO hardware diagram

6.1.1.2 Total Store Order

In the remainder of this chapter, we will consider the Total Store Order (TSO) memory model of the Intel's x86 family [Int12] or the SPARC processors [Int92]. Figure 6.2, borrowed from [SSO+10] illustrates the model of an x86 multiprocessor architecture. A certain number of hardware threads is given; each one corresponds to a sequence of instructions to execute. These threads interact with a storage subsystem, indicated as a dotted box.

The storage subsystem includes (i) a global shared memory, mapping addresses to values (ii) a global lock that a particular thread must hold for an exclusive access to the memory and (iii) one store buffer per hardware thread. A store buffer is a FIFO buffer of pending memory writes (pairs of memory address and value). This buffer avoids the need to block the thread while a write completes, i.e. is committed to the main memory. This commit can be performed at any time of the execution, except if another thread holds the lock on the memory. When a thread reads a given memory address, it first considers its own buffer, and reads the most recent pending write to that address (if it exists). If there is no pending write on that address, the thread looks for the value in the main memory. Write buffers are thread-local, meaning that a given thread cannot read inside the other threads' buffer. Suppose a processor p writes a value in its store buffer. A read from a distinct processor p' can read an old value from the memory, because the value written by p has not yet been propagated to memory.

This phenomenon occurs in the program given in Figure 6.1a, where $r_1 = r_2 = 0$ is a valid result under TSO. Indeed, the memory reads (b) and (d) can be performed before threads have flushed their respective buffer; each thread cannot see the write performed by the other. In other words, Dekker's algorithm is broken under TSO. To fix the problem, and ensure that this configuration cannot be reached, one simply has to add the x86 instruction `mfence` between (a) and (b) and betwen (c) and (d). As specified in [Int12], `mfence` serializes the store and load operations. It cannot execute until the thread's buffer has been flushed.

6.1.1.3 DRF guarantee

In order for the programmer to write correct multi-threaded applications, SC is probably the easiest memory model to work with, though currently not the most efficient one[3]. In order

[3]The work of Singh et al. [SNM+12] aims at studying how SC hardware could be designed and implemented with reasonable performance costs.

to reconcile the programmer (or software) and the architecture, the idea is then to see the memory model as a contract between those two parties.

One such contract, proposed by Adve and Hill in 1990, is the *Data-Race-Free Guarantee* (DRF). It is based on the observation that SC is hard to maintain mainly when processors interact with each other through shared variables [AH90]. DRF ensures that *a weak memory model behaves like SC, whenever the program is correctly synchronised, i.e. threads do not communicate with each others by any means other than synchronized memory accesses.* The term DRF refers to the notion of *data-race*. The formal definition of data-race depends on the precise meaning of the synchronization actions provided by the model (see Section 6.4.3 for a formal definition). Here, we stay at an informal level. A race basically consists in a simultaneous non-synchronized access by two distinct threads to the same shared memory location, including at least one write.

Example 6.1 (Example on TSO)**.** *TSO provides such a DRF guarantee. Reconsider the example program in Figure 6.1a. The non-SC execution yielding* $r_1 = r_2 = 0$ *is a valid execution under TSO. This does not contradict the DRF guarantee, since the program is not correctly synchronized: memory actions* (a) *and* (d) *form a data-race on* x*. Adding* `mfence` *instructions just before the two reads synchronizes the write and read on* x *(and* y*). It thus removes the data-race on* x *(and* y*). By the DRF guarantee of TSO, we then get that* $r_1 = r_2 = 0$ *is not an observable behavior.*

DRF is nowadays a criteria of quality for any new memory model to be designed. An invaluable aspect of DRF is that the synchronization contract (i.e. the absence of data-race) can be described in the SC setting: two conflicting accesses form a data-race if and only if there is an interleaving execution in which these accesses are adjacent. Indeed, if such memory accesses are adjacent in a given interleaving, they could have equally occured in the reverse order. Both actions are thus doable at the same time in the execution. For a program to be data-race free, all its interleavings must be free of data-race. In this contract, the programmer does not have to master the weak memory model at hand if SC is the only model he wants to work with.

6.1.2 Software memory models

Current multiprocessor hardware has weak memory models. Depending on the extent to which one wants to reflect the behavior of the underlying architecture, several alternatives exist for defining a corresponding software memory models (for source and IR languages). We explore them in this section. Between the languages and the hardware lie compilers. The compiler is thus the third component that should be aware of the memory model; as we shall see, compilers even sometimes influence the definition of the software memory model. Figure 6.3 illustrates the scenarios we will describe in this section. The grey areas denote the semantics preservation result that holds accross the different levels (source, IRs, and assembly) of languages.

Scenario (a) presents the intuitive and natural concurrent semantics most programmers have in mind: from the source program to the assembly level, the memory model is described by an interleaving semantics (SC). As is, this can only be valid on uniprocessor architectures. Notice also that in this context, the compiler should preserve the SC semantics of programs, in the flavor of Marino's work [MSM+11]: any behavior observed on top of the interleaving semantics of the optimized program should be an observable behavior in the interleaving semantics of the initial program.

However, this model restricts program optimizations. Several basic optimizations such as procedure inlining, loop unrolling, and control-flow simplification (e.g. branch tunelling) do not modify the order of memory actions; these are thus already SC-preserving. But other compiler optimizations, such as Common Sub-expression Elimination (CSE) or Loop Invariant Code Motion can have the effect of reordering memory actions, and are thus invalid in SC. We illustrate this issue below with CSE.

Example 6.2 (CSE breaks SC)**.** *Consider the simple example below. Removing the redundant computation $x * 2$ is not correct, since it can add some behaviors to the optimized program. For instance, no interleaving of the initial program leads to the configuration where $r_2 = 1$ and $r_3 = 0$. In the optimized version, some interleavings lead to this result. CSE can be seen as reordering of memory actions: the optimized program behaves as if the second memory read of x (in the initial program) would have been moved before the read of y, since the read of y does not constrain the value read for x anymore.*

$x \leftarrow 0; y \leftarrow 0$	
$r_1 \leftarrow x * 2$	$x \leftarrow 1$
$r_2 \leftarrow y$	$y \leftarrow 1$
$r_3 \leftarrow x * 2$	

Initial program (before CSE)
$r_2 = 1, r_3 = 0$ *invalid*

$x \leftarrow 0; y \leftarrow 0$	
$r_1 \leftarrow x * 2$	$x \leftarrow 1$
$r_2 \leftarrow y$	$y \leftarrow 1$
$r_3 \leftarrow r_1$	

Optimized program (after CSE)
$r_2 = 1, r_3 = 0$ *valid*

These optimizations must thus be adapted so that they do not break SC (denoted as SC-preserving in Figure 6.3). For instance, they could only target thread-local and compiler generated temporaries. Another possibility is to execute the optimized code only when threads are (dynamically) ensured not to have raced [MSM+11].

6.1.2.1 SC-enforcing compilers

The only way for an end-to-end SC model to be valid on a multiprocessor architecture is to *force* the relaxed hardware to behave like SC (Scenario (b) in Figure 6.3). This can be done by e.g. naively inserting memory barrier instructions after each memory operation, so that buffers are immediately flushed, propagating writes to main memory.

Because of the insertion of memory fences and the restriction of optimizations to be SC preserving, this approach can be costly — the hardware's optimizations are blocked. It additionally impacts programs that were initially free of data-race (or even single threaded programs), for which the hardware would behave as SC, according to the DRF guarantee.

Heuristics can be found to reduce the impact of the approach (with fence elimination, or the use of synchronisations that are cheaper than fences when it is possible). Alglave et al. [AM11] conduct an experimental study on x86 and Power achitectures, concluding that fence insertion for restoring SC roughly halves performance, when costly fences are needed.

6.1.2.2 SC for correctly synchronised programs

One way to recover part of the optimizations that compilers perform in a sequential context is to consider that data-races are bugs, as are out-of-bound array accesses or null pointer dereferencing. In this model, data-race free programs have an interleaving semantics, while the semantics of racy programs is left completely undefined. This *"no benign data-race"*

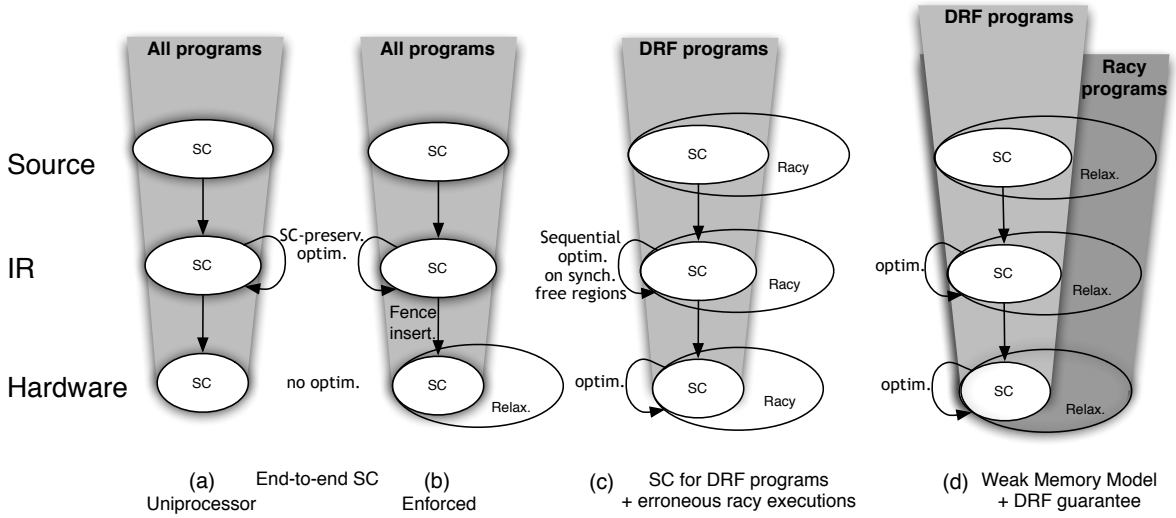


Figure 6.3: Weak memory models: possible scenarios

approach is taken for C [HAZN08] and C++ [BA08, BMO+12] memory models[4]. Dekker's example program in Figure 6.1a has an undefined semantics because it has a data-race (on x and y). This situation corresponds to scenario (c) in Figure 6.3: only SC executions are considered, and racy programs are just ignored from the source level down to the assembly level.

Working under the hypothesis that programs are data-race free brings several benefits. First, in data-race free programs, any portion of code that does not contain synchronization actions can be considered as executed in isolation, and threads interleave only at synchronization points [BA12]. Consider the example in Figure 6.4. This program is data-race free and free of synchronization action. Hence, threads modify distinct sets of memory locations. As a consequence, any interleaving of that program is equivalent to another interleaving in which the code of a thread is completely executed before the code of the other thread. For instance, the interleaving (a-d-b-e-c) is equivalent to (a-b-c-d-e), which is itself equivalent to (d-e-a-b-c) as there is no synchronization at all.

$x, y, v, z, w \leftarrow 0$	
$(a)\ r_1 \leftarrow x$	$(d)\ r_2 \leftarrow y$
$(b)\ v \leftarrow r_1$	$(e)\ w \leftarrow r_2$
$(c)\ z \leftarrow 2$	

Any interleaving is equivalent to (a-b-c-d-e) or (d-e-a-b-c)

Figure 6.4: Synchronization free regions of DRF programs execute as if they were atomic

This property simplifies the semantics of DRF programs. But it also makes sequential optimizations valid on those regions, thus allowing the compiler to easier and better optimize the code, compared to the previous SC model. In Figure 6.3, scenario (c), these are referred to as sequential optimizations. Optimizations should however carefully avoid introducing memory reads or writes that conflict with other threads, as they could themselves introduce races (Boehm [Boe05] gives some examples where this happens). At the end of the compilation chain, the assembly (data-race free) program will execute in a sequential consistent way, by the DRF guarantee of the hardware memory model.

[4] C and C++ additionally provide *low-level atomics*, for which some relaxations can be specified by the user in the new standard C++ 11 and ISO C 2011. Here, we do not consider this "expert-only" feature (dixit Boehm [BA08]).

A second advantage is that library calls that do not use internal synchronization behave as if they were executed in a single step. Here, what is gained is the intuitive semantics for library calls. It only makes sense in this model [Boe05].

All of these properties hold only under the DRF program assumption, including the correctness of the compilation chain. In particular, a compiler is allowed to do whatever it wants with a racy program. It could first fail to compile the program, but this is harmless, as no incorrect assembly code is produced anyway. Second, it might compile it into an assembly program with no race. In this case, the assembly program does not contain a race-bug anymore and has thus a defined semantics: this does not break the semantics preservation of the compiler, but the compiler is allowed to produce the program of its choice. When working and reasoning under a DRF program assumption, one thus has to prove the absence of data-race in the source program before its compiled version is executed, either by detecting all potential races or eliminating them. Analysing data-race conditions in programs is hard, and it is a research topic on its own. We now review the three main existing approaches.

Ensuring data-race freedom First, one can *prove* formally that programs are free of data-race, using e.g. Concurrent Separation Logic (CSL) [HAZN08]. CSL is not limited to the proof of the absence of data-races in programs: it is used to prove the functional correctness of heap manipulating multi-threaded programs and data-race freedom of programs comes as a by-product of the CSL verification.

A second approach, more automatic, is to check the absence of data-race by static analysis (see e.g. the work of Naik et al. [NAW06, NA07]). The main difficulty is to reach a good precision level, while still being able to scale. In fact, such analyses must keep track of alias information in the program. Additionally, they not only have to track inter-procedural control information, but also inter-thread flows in order to detect e.g. that two locks held at a given point in two threads' code denote the same location (and hence deduce that a potential race is actually not a race). Because of the complexity of the analysis, precision comes often at the price of soundness in practice [NAW06]. This approach, as well as CSL, also require to be able to analyze the code of libraries, which can sometimes be unavailable.

Finally, dynamic techniques can be applied to detect data-races at run-time [EQT07, FF09]. At the precise moment where a data-race is about to execute, a run-time exception is raised. Thus, program executions are either sequentially consistent or lead to a run-time exception. It is unclear whether such techniques could be applied for ensuring the data-race freedom of a program. Most of dynamic methods only consider one execution of the program. Some powerful analyses [FF09] are however able to compute, for a given run, information that is valid for all interleavings of that execution[5]. To the best of our understanding, these techniques seem more defining another memory model (where races would not be left undefined, but would be given an exceptional semantics), rather than being a way of ensuring the data-race freedom of the source program, especially when the code is JIT-compiled. The major challenge in this application case is to ensure a complete accuracy, without too much overhead.

We have discussed the SC memory model for data-race free programs and some of its benefits. This model is notoriously known to balance simplicity with performance requirements during initial software development [AB10]. Reasoning and programming with an interleaving semantics is correct, as long as programs do not contain data-races. The data-race freedom

[5] On programs with fixed inputs, those techniques are thus sound.

of programs can be verified using either formal proof or static program analysis. As for Java, and more generally for safe languages, leaving the semantics of racy programs undefined is not acceptable [MG04]: memory safety, the correct initialization of variables, ensuring a constant value for final fields, or the absence of *out-of-thin-air* reads (i.e. reads of values that have never been written at any point in the execution) are safety and security properties that must be provided for all programs, including the racy ones that an attacker could deliberately write.

6.1.2.3 Weak DRF memory models

Rather than leaving racy programs' semantics undefined, weak memory models for languages give a semantics to *all* programs. Even if the model is relaxed for all programs, it provides a DRF guarantee providing the simpler interleaving model for data-race free programs. This approach (see scenario (d) in Figure 6.3) leads to more complex models.

There are two categories of relaxations that such a model can include. First, it can account for the reorderings performed at the hardware level. In this case, the hardware memory model is simply lifted to the level of the language. In [ŠVZN+11], Ševčík et al. propose to give a TSO-relaxed semantics to Clight programs. The second category takes its origin in the optimizations that compilers were performing in the traditional sequential case, and that compiler writers were not willing to abandon. This is the case for CSE, but also of many more, as sequential optimizations often benefit from program representations that abstract the CFG, one of the most representative being Click's sea-of-nodes [Cli95b].

In some cases, the relaxations allowed by the hardware are already enough, meaning that some optimizations are valid under the hardware memory model (e.g. some forms of speculative loads and some forms of lock optimizations). But some of them are in a sense not relaxed enough: relaxed behaviors of the optimized program do not correspond to any relaxed behavior of the initial program. For instance TSO is not relaxed enough for a full CSE or code motion. In order for those optimizations to keep valid, software memory models for high-level programming languages, such as the one of Java [MPA05], have been tuned. Additionally, the desire of portability for Java required to define a memory model such that:

> *It should be possible to design correct, high performance JVM implementations across a wide range of popular hardware architectures.(Manson and Goetz [MG04])*

In Figure 6.3 (d), the set of relaxed executions should thus be even wider, to take into account various hardware models. To sum up, language weak memory models are designed to meet the following requirements. First, all programs are given a semantics. Second, the model exposes to the programmer the relaxations performed by the hardware. Third, it should allow compilers to perform many optimizations (typically more than just SC-preserving optimizations). Finally, a DRF guarantee must be provided, so that the semantics of correctly synchronized programs remains as simple and intuitive as possible. In the next section, we discuss the limits of such an ambitious goal in the case of Java.

6.1.2.4 The limits of the Java Memory Model

The Java memory model (JMM), originally developed in 1995 as part of the Java Language Specification [GJS96] was largely perceived as broken, as some optimizations were not allowed. Yet it was not providing strong enough guarantees for code safety [Pug00]. It was later updated through the Java Community Process, with the Java Specification Request 133 [JSR04], that

took effect in 2004. The formalization was published in 2005 [MPA05], targeting an academic audience.

The degree of generality of the JMM (a semantics for all programs, including racy ones, and portable across all architectures) comes at the price of ease of understanding. Indeed, when it comes to explaining the semantics of a racy program, current textbooks [GPB+06, Blo08] do not even attempt to define the JMM's notion of a *legal execution*.

The JMM is specified in terms of partial orders on read and write actions, and subtly defined predicates over execution traces. These predicates allows complex reorderings, but in an indirect way, relying on the notion of causality order on data-races; the outcome of a data-race must be explained in terms of previously *committed* races. This is a level of complexity that is challenging even for experts. Because the reorderings that are allowed do not appear clearly in the definition of the JMM, understanding the sets of valid and invalid program executions in the JMM is rarely intuitive, as shown by the example below:

$$x \leftarrow 0; y \leftarrow 0; z \leftarrow 0$$

$r_1 \leftarrow z$	$r_2 \leftarrow x$
`if` $(r_1 = 1)$ $\{x \leftarrow 1; y \leftarrow 1\}$	$r_3 \leftarrow y$
`else` $\{y \leftarrow 1; x \leftarrow 1\}$	`if` $(r_2 = 1$ && $r_3 = 1)\{z \leftarrow 1\}$

According to [MPA05], the JMM prohibits $r_1 = r_2 = r_3 = 1$. However, simply reordering the writes to x and y in either of the branches of the thread on the left (so both branches perform the writes in the same order[6]), the conditional expression $(r_1 = 1)$ could be eliminated, and assignments could be hoisted above the store to r_1, making this execution permissible. Thus, even though each of the transformations appears benign, a compiler is prohibited from performing them. Interestingly, alternative definitions of the JMM [AŠ07a] do allow this execution, but the justification is complex, involving subtle notions of speculations and non-local reasoning over execution traces. Such unintuitive explanations weaken the utility of the JMM; [AŠ07b] provides a number of examples that illustrate these concerns.

The subtleties arising in thinking about program executions within the JMM has been the subject of much research [CKS07, AŠ07b, AŠ07a, HP07, Loc12]. In [AŠ07a], Aspinall and Ševčík found that four test cases provided by Pugh et al.[7] were flawed. Later, Torlak et al.'s automatic verification tool [TVD10] contradicted their interpretation of the JMM on two of these examples. These ambiguities already appear for programs less than 10 instructions long.

From the compiler writer's point of view, the situation is not better because there is no clear operational definition that captures the behaviors permitted by the JMM. Thus, it becomes difficult to write any correctness proof of a Java compiler with respect to the current definition. Indeed, previous work has shown that existing Java compilers are not JMM compliant [ŠA08]. As a practical matter, we are unaware of any attempt to prove JMM compliance of any compiler. Moreover, as pointed out in [AŠ07b], the JMM is flawed for programs with potentially infinite traces. Even when restricted to finite executions, it requires sophisticated reasoning, and often yields unintuitive results [AŠ07a, ŠA08].

One could imagine defining a completely operational definition of the JMM. Even if such a formalization existed (we are unaware of any previous attempt to do so), its complexity would hardly ease the verification of a compiler.

[6] Some architectures, such as Power or x86 in Partial Store Ordering mode allow reordering of write actions.
[7] `http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html`

6.2 An alternative contract: BMM

The JMM definition is notoriously complex, both for the programmer and the compiler writer. From a formal verification perspective, it seems even more difficult to try to prove a compiler correct with regards to the JMM, because of its great generality.

But, towards the formal verification of multi-threaded Java compilers, it seems reasonable to specialize the compiler to a given target architecture. This is semantically correct, as it amounts to refine the memory model early in the compiler chain. This way, one dimension of the generality of the JMM is cut down. This is what we propose to do here, by considering target architectures with a tractable, well-defined and well-understood semantics. Hence, we consider architectures with a TSO memory model, whose tractability in the field of verified compilation has already been demonstrated [ŠVZN+11], thanks to its operational characterization. In short, we define a formal contract between programmers, verified compilers and TSO architectures that is easier to fulfill and prove correct. Ideally, only the backend of the compiler would be architecture dependent, and architecture-independent optimizations would be allowed in the high-level layers of the compiler. Thus, one could fear that the contract we propose would doom the compiler to produce less efficient code, but our preliminary experiments are encouraging. We come back to this point at the end of the section.

The contract we propose here is a *Buffered Memory Model* for Java (BMM). While not as expressive as the JMM, in terms of valid optimizations, we believe it imposes only modest impact on performance, given its well-suitedness for formal reasoning. BMM comes in two forms, presented below, that we formally prove equivalent.

Axiomatic reordering based model First, we fully characterize BMM in terms of the memory reorderings it allows on top of SC executions. This axiomatic characterization is close to the current JMM specification style, but avoids the complex definition of JMM legal executions and provides a more constructive and intuitive method to describe valid program executions, even in the presence of races. This axiomatic view allows comparing BMM with the actual definition of the JMM (we prove that BMM is a subset of JMM), and shares some design choices with the recent relaxed memory model proposed by Burckhardt et al. [BMS10].

From the programmer's and compiler writer's point of view, we think the memory model exposed to Java programmers should be viewed as a set of rules specifying the kinds of statements that can be reordered starting from an SC interpretation and enabling either compiler optimizations (e.g., common-subexpression elimination) or hardware out-of-order execution (e.g., speculative loads).

The central idea of the reordering-based definition of BMM is thus to allow determining whether a given execution is valid by only considering various combinations of permitted reorderings. First, all SC executions are allowed. Second, any execution from which an SC execution can be derived by reordering a read action with an immediately preceeding write action (over disjoint locations) is also allowed[8].

Figure 6.5 gives some examples of permitted and prohibited executions. In these examples, executions (b), (c) and (d) are BMM-invalid because they are SC-invalid and no reordering of a read before a write is possible. On the contrary, two such reorderings are possible in execution (a) and they lead to a program execution that is SC-valid. It shows that execution (a) is BMM-valid. We believe this methodology helps figuring out whether a given execution

[8]This rule can be further generalized to deal with reorderings that involve multiple dependent reads.

$$\begin{array}{c} x \leftarrow 0; y \leftarrow 0 \\ \hline \begin{array}{c||c} x \leftarrow 1 & y \leftarrow 1 \\ r_1 \leftarrow y & r_2 \leftarrow x \end{array} \end{array}$$

(a) $r_1 = r_2 = 0$ is legal

$$\begin{array}{c} x \leftarrow 0; y \leftarrow 0 \\ \hline \begin{array}{c||c} x \leftarrow 1 & r_1 \leftarrow y \\ y \leftarrow 1 & r_2 \leftarrow x \end{array} \end{array}$$

(b) $r_1 = 1, r_2 = 0$ is illegal

$$\begin{array}{c} x \leftarrow 0; y \leftarrow 0 \\ \hline \begin{array}{c||c||c} x \leftarrow 1 & r_1 \leftarrow x & r_2 \leftarrow y \\ & y \leftarrow 1 & r_3 \leftarrow x \end{array} \end{array}$$

(c) $r_1 = r_2 = 1, r_3 = 0$ is illegal

$$\begin{array}{c} x \leftarrow 0; y \leftarrow 0 \\ \hline \begin{array}{c||c||c||c} x \leftarrow 1 & r_1 \leftarrow x & y \leftarrow 1 & r_3 \leftarrow y \\ & r_2 \leftarrow y & & r_4 \leftarrow x \end{array} \end{array}$$

(d) $r_2 = r_4 = 0, r_1 = r_3 = 1$ is illegal

Figure 6.5: Valid and invalid executions under BMM.

is allowed or not under BMM.

Operational characterization The second view of BMM is a clear and easily understood operational semantics with write buffers attached to each thread. This view of the model is essential for verifying compilers with the help of simulation diagrams. It has been designed by taking into account the x86 buffer memory model (TSO) proposed by Sewell et al. [SSO+10] with careful consideration of how it impacts compiler optimizations.

6.2.1 BMM from a larger perspective

Figure 6.6 illustrates the overall role that BMM could play in a verified Java bytecode software chain. Existing software that has been written with the JMM in mind can be run directly, as BMM is a subset of the JMM: every legal BMM execution is legal under the JMM. This ensures that legacy software that has been validated and tested with the JMM in mind will remain correct. But safety-critical programmers can also choose to write software against the BMM directly as it is easier to understand its requirements. The formal tractability of BMM could also be exploited to implement and prove correct verifiers or static analyzers. Of course, this means that if such software is ported to an environment running on the JMM, additional work would be needed to validate it[9].

The operational characterization of the BMM, which we call $\mathsf{BMM_o}$, would be used in the compiler correctness proof for the phases that manage a high-level IR. We show there is an exact correspondence between the definitions of BMM and $\mathsf{BMM_o}$. The lower levels of the compiler could be based on the formalization of TSO found in [ŠVZN+11]. The low-level optimizations will leverage the existing CompcertTSO compiler because the BMM and the operational definition of the TSO machine found in CompcertTSO are in agreement.

We do not claim the BMM is a general-purpose memory model for Java, and, while we believe it can be generalized beyond this compiler framework, we leave a study of its utility on other architectures such as Power PC or ARM, which support other memory models than TSO, as an open question.

6.2.2 Summary

Table 6.1 provides a synthesis of the properties of BMM, compared with other models. The two views of BMM are useful to prove different kinds of properties. In Section 6.4.3 we use the

[9] As a concrete example, the *Double-checked locking* [BBB+04] pattern is correct under BMM without the use of a costly volatile field which is still mandatory under the JMM.

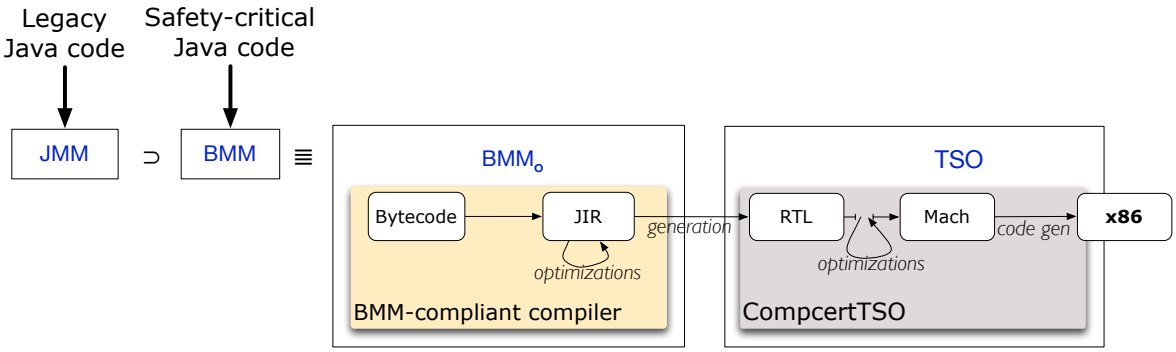


Figure 6.6: A verified Java bytecode software chain

	SC	C++	JMM	BMM
DRF theorem	✓	✓	✓	✓
Reordering memory accesses	×	✓	✓	⊗
Redundant memory accesses elimination/introduction	✓	✓	⊗	⊗
Operational semantics	✓	✓	×	✓
Semantics for all programs	✓	×	✓	✓
Programmer can understand the semantics of racy programs	✓	×	×	✓
Sound with respect to JMM (does not break legacy Java)	✓	×	✓	✓
Lock optimizations	✓	✓	⊗	⊗

Table 6.1: Expressivity and properties for different memory models – We write ✓ if the property holds generally, × if it does not, and ⊗ if some restrictions limit when the property holds.

reordering view to provide a DRF theorem for BMM. In Section 6.7 we use the operational semantics to study the validity of compiler-driven program transformations under BMM.

The models differ in the kinds of reordering they permit, how they are formalized, the set of programs they consider, and their support for legacy code. All provide the DRF guarantee, and enable useful lock optimizations. BMM is weaker than the JMM in terms of the reorderings it allows, but its operational semantics is useful for verifying compiler optimizations, and its simpler axiomatic version is easier for programmers to understand. Reordering memory accesses is illegal under original JMM [MPA05, CKS07] but legal under the alternative version proposed by [ŠA08].

6.2.3 Contributions and content

The contributions of this chapter are as follows: (i) an axiomatic definition of BMM, an alternative memory model for concurrent and racy Java programs that is fully characterized in term of memory event reorderings (ii) a formalization of BMM$_o$, an operational definition of the semantics of concurrent Java programs suitable for formal verification, that simply maps to the TSO memory model found in x86 and Sparc multiprocessors (iii) a proof that the JMM

is a superset of BMM (iv) a proof that BMM and $\mathsf{BMM_o}$ are equivalent (v) a proof of the data-race free (DRF) theorem for BMM.

This work has later been extended with two additional contributions, due to Vincent Laporte, David Pichardie and Lei Zhao. First, the formalization of $\mathsf{BMM_o}$ has been mechanized in Coq. The formal development can be found at `http://r.cs.purdue.edu/bmm/`. Second, an experimental study has been conducted to estimate a coarse upper-bound on the performance impact imposed by BMM compared to JMM, on a production virtual machine that is run on a TSO architecture. For the sake of completeness, we provide a summary of the results of this evaluation. The rest of the chapter is organized as follows. Section 6.3 provides a background on JMM and gives useful definitions that are used in the rest of chapter. Section 6.4 presents the formal definition of the reordering-based memory model BMM. Section 6.5 gives the formal definition of the operational memory model $\mathsf{BMM_o}$. The two memory models are proven equivalent in Section 6.6. Section 6.7 provides a list of transformations that are correct under BMM, and Section 6.8 briefly presents the empirical evaluation of BMM. We discuss related work and conclude in Sections 6.9 and 6.10.

6.3 Background on Java Memory Model

The axiomatic view of BMM is formulated using some of the notions underlying the JMM. We recall these in this section, as well as some other preliminary definitions that we will use later. We start with the inter-thread actions defining the interactions of threads.

6.3.1 Inter-thread actions

The shared memory of a program is split into a set of disjoint shared *addresses*. In practice, addresses are instance fields, static fields or array positions but they are not local variables of a method (Java type safety forbids us from manipulating their memory addresses). For each address $x \in \mathbb{X}$, we can determine if it is volatile or not with the function $\mathsf{volatile} : \mathbb{X} \to \text{bool}$. In the literature, external actions are distinguished from other memory actions. In this work, we model external actions with volatile writes[10], that can be identified with the function $\mathsf{external} : \mathbb{X} \to \text{bool}$.[11] We assume a set $\mathbb{T}$ of dynamic threads, a set $\mathbb{L}$ of memory locks, and a set $\mathbb{V}$ of values. The set of inter-thread actions we consider is given below, where superscript i denotes the unique identifier of memory actions.

$$
\begin{array}{rcll}
\mathbb{A} & ::= & \mathsf{w}^i_t x, v & \text{(thread } t \text{ writes value } v \text{ to address } x\text{)} \\
 & | & \mathsf{r}^i_t x & \text{(thread } t \text{ reads from address } x\text{)} \\
 & | & \mathsf{l}^i_t l & \text{(thread } t \text{ acquires a lock on monitor } l\text{)} \\
 & | & \mathsf{u}^i_t l & \text{(thread } t \text{ releases a lock on monitor } l\text{)} \\
 & | & \mathsf{s}_t t' & \text{(thread } t \text{ creates a new thread } t'\text{)} \\
 & | & \mathsf{b}_t & \text{(thread } t \text{ starts)} \\
 & | & \mathsf{j}^i_t t' & \text{(thread } t \text{ detects } t' \text{ has terminated)} \\
 & | & \mathsf{e}_t & \text{(thread } t \text{ ends)} \\
 & | & \mathsf{w}_0 x & \text{(default write action to address } x\text{)}
\end{array}
$$

$$x \in \mathbb{X} \qquad v \in \mathbb{V} \qquad l \in \mathbb{L} \qquad t, t' \in \mathbb{T} \qquad i \in \mathbb{N}$$

[10] For example, a call by a thread t to a function f with arguments args that returns a value v is modeled as a volatile write of this value by that thread to the abstract location f(args); it hence captures any I/O behaviour.

[11] We require that, $\forall x, \mathsf{external}(x) \Rightarrow \mathsf{volatile}(x)$

Action $\mathsf{w}_0 x$ is the default write action to the address x. It has no emitting thread: memory initialization is done somewhat implicitly in our formalization – no particular thread is in charge of initializing the memory.[12] Thread starting (b_t) and ending (e_t) actions can happen only once during an execution and thus do not require any identifier. For any action a that is not a default write action, we write $T(a)$ the emitting thread of this action. For any write action w, we write $V(w)$ the value written by that action; default write actions write a default value according to the type of the related address.

We introduce some notations for families of actions:

$$
\begin{aligned}
\mathbb{A}_{\mathrm{r}} &= \{\mathsf{r}_t^i x \mid t \in \mathbb{T},\ x \in \mathbb{X}\} && \text{(reads)}\\
\mathbb{A}_{\mathrm{w}} &= \{\mathsf{w}_t^i x, v; \mathsf{w}_0 x \mid t \in \mathbb{T},\ x \in \mathbb{X},\ v \in \mathbb{V}\} && \text{(writes)}\\
\mathbb{A}_{\mathrm{d}} &= \{\mathsf{w}_0 x \mid x \in \mathbb{X}\} && \text{(initializations)}\\
\mathbb{A}_{\mathrm{b}} &= \{\mathsf{b}_t \mid t \in \mathbb{T}\} && \text{(begins)}\\
\mathbb{A}_{\mathrm{s}} &= \{\mathsf{w}_t^i x, v; \mathsf{r}_t^i x \mid t \in \mathbb{T},\ x \in \mathbb{X},\ \mathsf{volatile}(x)\}\\
&\quad \cup \{\mathsf{l}_t^i l; \mathsf{u}_t^i l \mid t \in \mathbb{T},\ l \in \mathbb{L}\} && \text{(synchronizations)}\\
&\quad \cup \{\mathsf{s}_t t'; \mathsf{b}_t; \mathsf{j}_t^i t'; \mathsf{e}_t \mid t, t' \in \mathbb{T}\}\\
\mathbb{A}_{\mathrm{x}} &= \{\mathsf{w}_t^i x, v \mid t \in \mathbb{T},\ \mathsf{external}(x)\} && \text{(external actions)}
\end{aligned}
$$

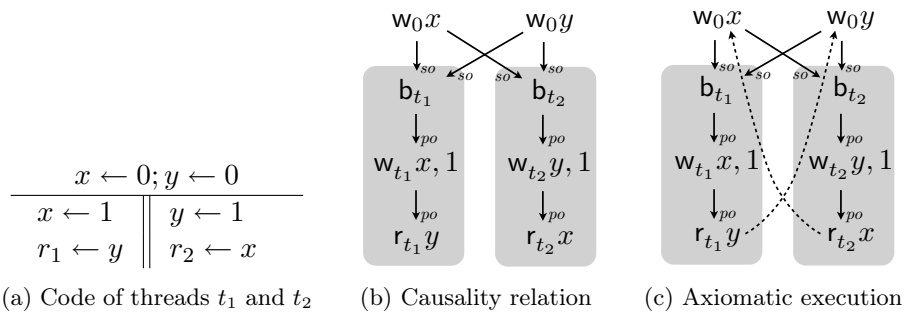
The current JMM and several architecture memory models are based on a *happens-before* model [Lam78]. An execution is described in terms of partial orders between memory actions. In a multithreaded program, each thread executes its own program and reads from and writes to the memory according to standard control and dataflow. The same external behavior of a program may be associated with many different interleavings of thread actions. An interleaving can be seen as a total order on actions: "this action occurs before that one according to global time". Such an interleaving is in fact a consistent extension of a partial order called "happens before" that precisely relates causal dependencies between actions. For example, the program Figure 6.7a may exhibit an interleaving of thread-actions

$$\mathsf{b}_{t_1} :: \mathsf{b}_{t_2} :: \mathsf{w}_{t_1} x, 1 :: \mathsf{r}_{t_1} y :: \mathsf{w}_{t_2} y, 1 :: \mathsf{r}_{t_2} x$$

but there is no causal dependency between the read performed in t_1 and the one performed in t_2. Figure 6.7b presents the causality relation behind such a linear presentation. Each gray region is dedicated to the actions owned by a same thread. The unique identifier is useless here and we omit it. Any sequence of black arrows between an action a and an action b means *a should happen before b*.

Because this is poorly synchronized, apart from address initialization and thread starts, few actions are actually constrained. We distinguish two kinds of arrows in this example. The arrow $\xrightarrow{po}$ reflects the program order between actions of a same thread. The arrow $\xrightarrow{so}$ reflects the synchronization relation between some events. Here, it is reduced to a relation between address initializations and thread starts but in more complex examples, it may relate an unlock of a monitor with its subsequent lock or the write of a volatile address with a subsequent read.

[12] This point diverges from [MPA05, AŠ07a]: we believe this is both closer to the initial JMM proposal [JSR04] and more suitable for an operational characterization. Because Java is type safe, every address is virtually given a default value at the start of the program, even if the corresponding location is not allocated yet.



(a) Code of threads t_1 and t_2 (b) Causality relation (c) Axiomatic execution

Figure 6.7: Example program and one of its happens-before executions with and without write-seen arrows.

To complete this partial-order view, it is necessary to model the values read during an execution. In Figure 6.7c we complete the picture with the write seen by each read action (dotted line). These set of arrows form what we call an axiomatic execution. The write seen by a read action must satisfy some minimal constraints that we will make clear with the notion of well-formed execution. We postpone until the next paragraph the precise requirements we demand on programs. A formal instantiation of this abstract notion is provided in the online material at http://r.cs.purdue.edu/bmm/.

Notations When a partial order is total on a countable set of elements we sometimes write it directly as a sequence of elements that uniquely characterizes it. When a partial order $\xrightarrow{o}$ is a disjoint union (indexed by $\mathbb{T}$) of orders, we write it as $[\xrightarrow{o}]_t$, its restriction on thread t. Conversely, a list of elements can be thought of as a total order. We will write $a \xrightarrow{tr} b$ when elements a, b are ordered with regard to a list tr. Notice that what is called an *order* in this paper is any irreflexive transitive relation. Two such relations P and Q are said to be *consistent* when they satisfy: $\forall x,\ y,\ \neg(xPy \wedge yQx)$. We write $tr \downarrow_A$ for the sequence tr filtered to the elements of the set A.

Definition 6.1 (Axiomatic Execution)**.** *An axiomatic* execution *E is described by a tuple $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ where:*

- *P is a program*
- *$A \subseteq \mathbb{A} \setminus \mathbb{A}_{\mathrm{d}}$ is a set of actions*
- *$\xrightarrow{po} \subseteq A \times A$ is the* program order*, a disjoint union of total orders on actions of each thread*
- *$\xrightarrow{so} \subseteq (A \cup \mathbb{A}_{\mathrm{d}}) \times (A \cup \mathbb{A}_{\mathrm{d}})$ is the* synchronization order*: the union of a total order*[13] *on $A \cap \mathbb{A}_{\mathrm{s}}$ of all synchronization actions in A, and the cartesian product $\mathbb{A}_{\mathrm{d}} \times (A \cap \mathbb{A}_{\mathrm{s}})$*
- *$W \in \mathbb{A}_{\mathrm{r}} \rightharpoonup \mathbb{A}_{\mathrm{w}}$ is a* write-seen *function that maps each read action r from A to a write action w of $A \cup \mathbb{A}_{\mathrm{d}}$ (r and w must operate on the same address).*

The JMM provides also a value-seen function that assigns a value to each write action from A. Here, we have directly attached this information to the write actions. The formal

[13] To lighten the notations, we omit the ordering of thread actions $\mathsf{b}_{\cdot}$ in the graphical representation of executions.

definition of the JMM requires us to consider different value-seens for the same write action; in our case, making this information immutable is sufficient.

We now explain how to extract the happens-before relation from the program order and the synchronization order of an execution.

Definition 6.2 (Synchronizes-with relation)**.** *An action a* synchronizes-with *an action b (written $a \xrightarrow{sw} b$) in an execution $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ if $a \xrightarrow{so} b$ and a, b satisfy one of the following conditions:*

- *$a \in \mathbb{A}_{\mathrm{d}}$ and $b \in A \cap \mathbb{A}_{\mathrm{b}}$ (default write actions synchronize-with any start action of the execution)*
- *a is a spawn of a thread t and b is the start of the thread t*
- *a is a write to a volatile address x and b is a read from x*
- *a is an unlock on monitor l and b is a lock on monitor l*
- *a is the end of the thread t and b is a join action on t.*

Definition 6.3 (Happens-before order)**.** *The* happens-before *order of an execution is the transitive closure of the union of its synchronizes-with relation and its program order.*

$$\xrightarrow{hb} = (\xrightarrow{sw} \cup \xrightarrow{po})^{+}$$

6.3.2 Intra-thread semantics

Our formalization tries to be as independent as possible of the concrete details of the underlying programming language (Java source or bytecode, intermediate representations of a Java compiler, etc.). A concrete instantiation of this abstract program model in the companion Coq development. The only requirement we make on programs is an abstract notion of intra-thread semantic state $\mathrm{State}_{\mathrm{intra}}$ and an intra-thread labeled transition relation

$$\xrightarrow{\cdot}\!\!\!\!\rightarrow \subseteq \mathrm{State}_{\mathrm{intra}} \times \mathrm{Label}_{\mathrm{intra}} \times \mathrm{State}_{\mathrm{intra}}$$

that is given to each thread $t \in \mathbb{T}$.[14] Transition labels belong to the set $\mathrm{Label}_{\mathrm{intra}} = (\mathbb{A} \setminus \mathbb{A}_{\mathrm{r}}) \cup (\mathbb{A}_{\mathrm{r}} \times \mathbb{V}) \cup \{\tau\}$: a thread can either take an action step, or a silent step that is memory irrelevant. For a read action step, the value read is paired with the action in the label. The requirements on this intra-thread semantics are:

- $\xrightarrow{\cdot}\!\!\!\!\rightarrow$ can only relate states of the same thread
- there is an *initial* state Ready: no transition leads to it and a thread t steps from it if and only if it emits the b_t action
- non-silent labels are tagged with the emitting thread
- there is a *final* state Done: a step of a thread t leads to it if and only if that transition is labeled by e_t and no transition steps from this state.

Definition 6.4 (Intra-traces)**.** *Let $tr = a_1 :: \cdots :: a_n$ be a sequence of actions in set A and let W be a write-seen function on A. Given a thread $t \in \mathbb{T}$ in program P, tr is an* intra-trace *of t if there exist $s_0, s_1, \ldots, s_m \in \mathrm{State}_{\mathrm{intra}}$ $(m \geq n)$ and $l = l_1 :: \cdots :: l_m \in \mathrm{list}(\mathrm{Label}_{\mathrm{intra}})$ such that:*

- *for all $a \in \{a_1, \ldots, a_n\}$, $T(a) = t$*

[14] See http://r.cs.purdue.edu/bmm/ for a formal definition.

- *s_0 is the initial intra-thread state* Ready
- *for all $i \in \{1, \ldots, m\}$, $s_{i\text{-}1} \xrightarrow{l_i} s_i$*
- *the projection $b_1 :: \cdots :: b_n$ of l to non-silent labels is such that $b_i = (a_i, V(W(a_i)))$ if a_i is a read action or $b_i = a_i$ otherwise.*

We write $P[t]$ for the set of such pairs (tr, W) for P.

Definition 6.5 (Well-formed execution)**.** *An execution $\langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$, is* well-formed *if*

- *A is finite*
- *$\xrightarrow{so}$ is consistent with $\xrightarrow{po}$*
- *Locking is proper: for all lock actions $\mathsf{l}^i_t l \in A$ and all threads t' different from the thread t, the number of lock actions on l emitted by t' before $\mathsf{l}^i_t l$ in $\xrightarrow{so}$ is the same as the number of unlock actions on l emitted by t' before $\mathsf{l}^i_t l$ in $\xrightarrow{so}$, and each unlock action $\mathsf{u}^i_t l \in A$ occurs after a matching lock action:*

$$\forall \mathsf{l}^i_t l, \forall t' \neq t, |\{\mathsf{l}^j_{t'} l \mid \mathsf{l}^j_{t'} l \xrightarrow{so} \mathsf{l}^i_t l\}| = |\{\mathsf{u}^j_{t'} l \mid \mathsf{u}^j_{t'} l \xrightarrow{so} \mathsf{l}^i_t l\}|$$
$$\forall \mathsf{u}^i_t l, |\{\mathsf{l}^j_t l \mid \mathsf{l}^j_t l \xrightarrow{po} \mathsf{u}^i_t l\}| > |\{\mathsf{u}^j_t l \mid \mathsf{u}^j_t l \xrightarrow{po} \mathsf{u}^i_t l\}|$$

- *$\xrightarrow{po}$ is intra-thread consistent: for all thread $t \in \mathbb{T}$, $([\xrightarrow{po}]_t, W) \in P[t]$*
- *$\xrightarrow{so}$ is consistent with W: for every read r of a volatile address x we have $W(r) \xrightarrow{so} r$ and for any write w to x different from $W(r)$, either $w \xrightarrow{so} W(r) \xrightarrow{so} r$ or $W(r) \xrightarrow{so} r \xrightarrow{so} w$*
- *$\xrightarrow{hb}$ is consistent with W: for all reads r of x, $r \xrightarrow{hb} W(r)$ does not hold and there is no intervening write w to x, i.e. such that $W(r) \xrightarrow{hb} w \xrightarrow{hb} r$.*

A special subfamily of well-formed executions is the set of sequentially consistent axiomatic executions.

Definition 6.6 (Sequentially Consistent (SC) execution)**.** *A well-formed execution $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ is* SC *if there exists a total order $\xrightarrow{to}$ on A such that*

- *$\xrightarrow{to}$ is consistent with $\xrightarrow{po}$ and $\xrightarrow{so}$*
- *For each read action $r \in A$ at address x, $W(r)$ is the last write on x before r in $\xrightarrow{to}$.*

The set of well-formed executions of a program forms the *Happens-Before* memory model. It is relatively easy to manipulate but it is not a satisfactory memory model for Java because it allows *out-of-thin-air* values [MPA05], does not fulfill the DRF theorem and breaks basic principles of Java security. The JMM [MPA05] considers then a subset of this model (but still containing SC executions); these are known as *legal* executions.

The exact definition of legal executions is somewhat technical and convoluted. In a nutshell, a well-formed execution E is legal if there exists a sequence $E_0, E_1, \ldots, E_n = E$ of well-formed executions such that in E_0, each read sees a write that it does not race with. Then progressively, each execution E_i allows some reads through data races but in a well-founded order until the execution E itself is reached. Thanks to this definition, one obtains almost directly the two important properties of the JMM: 1) in a data race free program all reads see writes that happen-before them and each execution is sequentially consistent 2) no out-of-thin-air value can be read, by the causality order on races imposed by the JMM.

6.4 Axiomatic memory model: BMM

We now formally define the first formal view of our memory model. The semantics is built on top of two notions that programmers should arguably well-understand: sequential consistency and instruction reordering.

We formalize the notion of local reordering we will consider in this work, capturing the optimization of individual thread executions that compilers or architecture might perform.

Definition 6.7 (Local reordering)**.** *Given an execution* $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$*, the execution* $E' = \langle P', A, \xrightarrow{po'}, \xrightarrow{so}, W\rangle$ *is a local reordering of* E *from an action list* ℓ *to an action list* ℓ' *in thread* t_0 *if*

- $[\xrightarrow{po}]_{t_0} = \alpha_0 \cdot \ell \cdot \beta_0$ *and* $[\xrightarrow{po'}]_{t_0} = \alpha_0 \cdot \ell' \cdot \beta_0$ *for some sequences* α_0 *and* β_0
- $[\xrightarrow{po}]_t = [\xrightarrow{po'}]_t$ *for all threads* $t \neq t_0$
- *for all* $(tr, W) \in P'[t_0]$ *where* tr *is of the form* $\alpha \cdot \ell' \cdot \beta$*, there exists* $(\alpha \cdot \ell \cdot \beta, W) \in P[t_0]$
- $P[t] = P'[t]$ *for all thread* $t \neq t_0$
- ℓ *and* ℓ' *contain the same set of actions*
- *neither elements of* ℓ *or* ℓ' *are synchronization actions.*

Such a reordering is written $E \xrightarrow{t_0:[\ell \longrightarrow \ell']} E'$.

Intuitively, we make a permutation of the intra-trace $[\xrightarrow{po}]_{t_0}$ of thread t_0 by transforming the sequence ℓ into the sequence ℓ'. BMM exposes two[15] fundamental local reorderings to the programmer. The first one is *Write-Read* reordering. Its basic effect is to reorder a read before a previous adjacent write if both actions target different addresses. Here is a simple example of a Write-Read reordering of a program, where x and y denote distinct locations:

$$\begin{array}{l} x \leftarrow 1 \\ r \leftarrow y \end{array} \xrightarrow{\text{WR}} \begin{array}{l} r \leftarrow y \\ x \leftarrow 1 \end{array}$$

Definition 6.8 (Write-Read reordering)**.** *A* Write-Read *reordering of an execution* $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ *with respect to a pair of write/read actions* (w, r) *of* A *in a thread* t*, is a local reordering* E' *such that*

$$E \xrightarrow{t:[w::r \longrightarrow r::w]} E'$$

w *and* r *must not operate on the same address. Such a reordering is written* $E \xrightarrow{\text{WR}} E'$.

Figure 6.8 illustrates the use of the Write-Read reordering on a classical litmus test program. To understand the BMM semantics, a key observation must be made here: the execution on the left is not sequentially consistent but after two WR reorderings we obtain a sequentially consistent execution. It is then tempting to ask if a BMM execution is any execution that can be transformed into an SC execution after some WR reorderings. Unfortunately, such a definition would not allow us to capture executions exhibited by TSO-hardware.

The program on the top-left part of Figure 6.9 illustrates this issue. In this program, the configuration $r_1 = 1; r_2 = 0; r_3 = 1; r_4 = 1; r_5 = 0$ is reachable under a TSO architecture, but it is not a SC execution and there is no way to apply any WR reordering on this program.

We hence introduce a second category of reorderings that is allowed in BMM that permits such executions.

[15] *Write-Read* reordering is in fact a special case of the forthcoming *Write-Read-Read* reordering but we believe it is easier to first present this simpler case before generalizing.

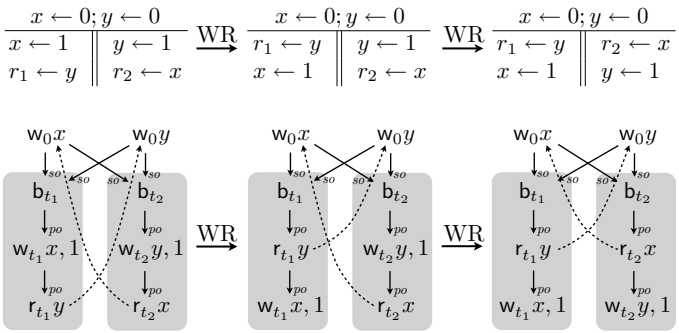


Figure 6.8: Write-Read reordering example

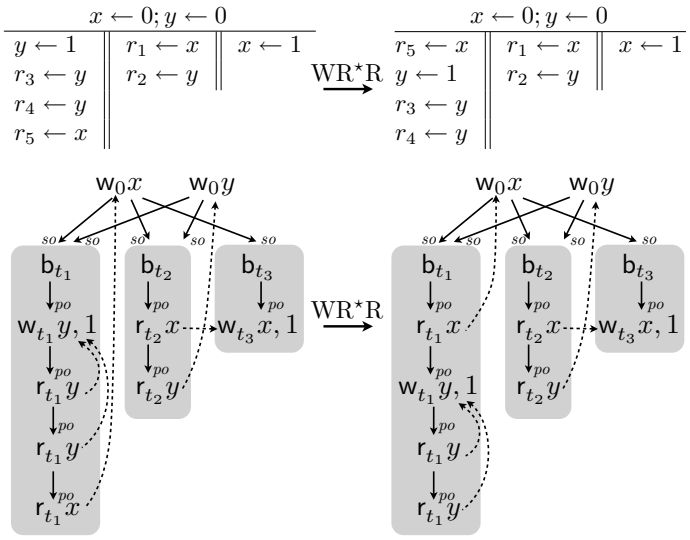


Figure 6.9: Write-Read-Read reordering example

Definition 6.9 (Write-Read-Read reordering)**.** *A* Write-Read-Read *reordering of* $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ *w.r.t. a tuple of write/reads/read action* $(w, \vec{r}, r')$ *of* A*, is a local reordering* E' *such that*

$$E \xrightarrow{t:[w::\vec{r}::r' \longrightarrow r'::w::\vec{r}]} E'$$

All reads in $\vec{r} = r_1, \ldots, r_n$ *must have* w *as write-seen, and* r' *and* w *must target different addresses. Such a reordering is written* $E \xrightarrow{\text{WR}^\star\text{R}} E'$*.*

In Figure 6.9, we apply this transformations to the previous program. This time, a reordering is possible and leads to an SC execution. We prove in Section 6.6 that this new reordering exactly captures a TSO operational semantics.

Formally, a BMM execution is any execution that can be successively transformed using WR⋆R reordering until we reach an SC execution. Note that WR⋆R is a generalization of the previous WR reordering.

Definition 6.10 (BMM executions)**.** *The set of* BMM *executions is defined as* $\text{BMM} = \left\{E \mid \exists E', E \xrightarrow{RO} E' \text{ and } E' \text{is SC}\right\}$ *where* $\xrightarrow{RO} = (\xrightarrow{\text{WR}^\star\text{R}})^*$*.*

In the sequel, we will write $\mathsf{BMM}(P)$ for the set of executions of a program P. The BMM observable behaviors of a program P is then defined as the set of sequences of external action:

$$Obs(P) = \left\{ \xrightarrow{so}\downarrow_{\mathbb{A}_x} \mid \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W \rangle \in \mathsf{BMM}(P) \right\}$$

6.4.1 BMM is a least post-fixpoint

Every time we need to prove that BMM is included in a given set of well-formed executions, we can rely on the following post-fixpoint characterization.

Lemma 6.1 (BMM least post-fixpoint characterisation)**.** BMM *is the least set S that satisfies*

- *all* SC *executions are in S*
- *S is backward-closed by* BMM *transformations: for any well-formed executions E, E' such that $E \xrightarrow{RO} E'$, if $E' \in S$ then $E \in S$.*

We will use this lemma to show that BMM is a subset of the JMM executions and to show the equivalence between BMM and the operational semantics given in Section 6.5.

6.4.2 BMM is a subset of JMM

The current Java Memory Model defines the set of legal executions as a subset of all well-formed executions that are justifiable using a sequence of intermediate justifications. In order to connect our model with the JMM, rather than unfolding the details of this formal definition, we rely on the following JMM properties:

- JMM accepts all sequentially consistent executions
- JMM allows reordering of non-volatile memory accesses hitting different locations [ŠA08].

Theorem 6.2. *Let* JMM *be the set of all legal executions permitted by the* Java *memory model. Then,* $\mathsf{BMM} \subseteq \mathsf{JMM}$.

Proof. We use here Lemma 6.1. We first know that JMM contains all SC executions. Then, suppose that $E \xrightarrow{RO} E'$ with $E' \in \mathsf{JMM}$. In the JMM, reordering non volatile memory accesses hitting different addresses is allowed [ŠA08]. We use this property to un-transform E' into E. Hence, E is also in JMM, meaning that JMM is backward-closed by WR⋆R. □

6.4.3 DRF guarantee

We establish that BMM enjoys the important property that any reasonable memory model should have, namely a *data-race-free guarantee* - data-race free programs only have SC executions. We define the standard notions of conflicting memory action, data-race, and data-race-free programs:

Definition 6.11 (Conflicting actions – Data-race – DRF)**.**

- *Two non-volatile actions $a, b \in \mathbb{A}_r \cup \mathbb{A}_w$ are* conflicting *if they target the same address and $T(a) \neq T(b)$ and at least one of them is a write.*
- *In an* BMM *execution $\langle P, A, \xrightarrow{po}, \xrightarrow{so}, W \rangle$, two conflicting actions a, b form a* data-race *if they are not ordered by $\xrightarrow{hb}$.*

- *A program P is* data-race free*, written $DRF(P)$, if all of its* SC *executions are free of data-race.*

Using Theorem 6.2 and the fact the JMM satisfies the DRF guarantee [Šev09], we obtain a DRF guarantee for BMM.

Theorem 6.3 (DRF guarantee)**.** *For all P, $DRF(P) \Rightarrow \forall E \in \mathsf{BMM}(P), E$ is* SC*.*

6.5 Operational memory model: $\mathsf{BMM_o}$

In this section, we provide an operational view of the Java Buffered Memory Model: $\mathsf{BMM_o}$. The reorderings allowed in its axiomatic version can be implemented by a $\mathsf{BMM_o}$ machine that attaches a write-buffer to each running thread. The $\mathsf{BMM_o}$ machine semantics will also be parametrized by an intra-thread semantics as specified in Section 6.3. Hence, we need to consider an extra set of actions: the silent actions in $\mathbb{A}_{\text{sil}}$ that are either the unbuffering $\overline{\text{B}}(a)$ of a write action $a \in \mathbb{A}_{\text{w}} \setminus \mathbb{A}_{\text{d}}$ by thread $T(a)$ or a silent step τ_t by thread t.

$$\mathbb{A}_{\text{sil}} ::= \overline{\text{B}}(a) \mid \tau_t$$

The idea behind $\mathsf{BMM_o}$ is to provide a generative, operational machine that produces executions in a format that is very close to what is specified in Section 6.3. Given an input operational execution, the machine executes, modifying a memory state that is made of thread buffers and a shared memory.

The input of the $\mathsf{BMM_o}$ machine is an operational execution, made of a program and a trace of *operational-actions*. An operational action $a \in \mathbb{A}_{\text{op}}$ is either an action in $\mathbb{A} \setminus (\mathbb{A}_{\text{d}} \cup \mathbb{A}_{\text{r}})$, or a pair in $\mathbb{A}_{\text{r}} \times \mathbb{A}_{\text{w}}$ (for each read action we record with it the write action that it sees, and refer to it as its *write-seen*), or a silent action in $\mathbb{A}_{\text{sil}}$.

Definition 6.12 (Operational Execution)**.** *An operational execution is a pair (P, tr) where P is a program and $tr \in \text{list}(\mathbb{A}_{\text{op}})$ is finite and such that no action appears more than once in tr.*

The $\mathsf{BMM_o}$ machine is then defined by a transition system, parametrized by an intra-thread semantics. We now describe its states and transitions. A $\mathsf{BMM_o}$ state $\in$ State is a record

$$\begin{array}{rcll} ts & \in & \mathbb{T} \rightharpoonup \text{State}_{\text{intra}}; & \text{(intra-thread state of threads)} \\ b & \in & \mathbb{T} \rightharpoonup \text{list}(\mathbb{A}_{\text{w}} \setminus \mathbb{A}_{\text{d}}); & \text{(one buffer per thread)} \\ m & \in & \mathbb{X} \to \mathbb{A}_{\text{w}} & \text{(one write action per address)} \end{array}$$

The semantics state first keeps track of each intra-thread state in $\text{State}_{\text{intra}}$. Each thread is given a write buffer; all non-volatile write actions must be first written to this buffer. When unbuffered, these writes are committed to the shared memory m, that maps addresses to write actions. Given a memory state (buffers b and memory m), the $\mathsf{BMM_o}$ machine specifies the write action a thread t can read when accessing the address x:

$$\text{rd}_t(b, m, x) = \begin{cases} w & \text{if } w \text{ is the first write to } x \text{ in } b(t) \\ m(x) & \text{if there is no write to } x \text{ in } b(t) \end{cases}$$

If a pending write to this address appears in the buffer of t, we take the most recent in the execution (i.e. the first in the buffer). Otherwise we consult the memory. If no write has been

$$\frac{ts(t) \xrightarrow{\tau} s}{ts, b, m \xrightarrow{\tau_t} ts[t \mapsto s], b, m}[\textsc{Tau}]$$

$$\frac{ts(t) \xrightarrow{\mathsf{r}_t^i x|V(w)} s \quad w = \mathrm{rd}_t(b, m, x) \quad \neg\mathsf{volatile}(x)}{ts, b, m \xrightarrow{\mathsf{r}_t^i x|w} ts[t \mapsto s], b, m}[\textsc{Read}]$$

$$\frac{ts(t) \xrightarrow{\mathsf{w}_t^i x,v} s \quad \neg\mathsf{volatile}(x)}{ts, b, m \xrightarrow{\mathsf{w}_t^i x,v} ts[t \mapsto s], b[t \mapsto (\mathsf{w}_t^i x, v) :: b(t)], m}[\textsc{Write}]$$

$$\frac{b(t) = l \cdot [\mathsf{w}_t^i x, v]}{ts, b, m \xrightarrow{\overline{\mathrm{B}}(\mathsf{w}_t^i x,v)} ts, b[t \mapsto l], m[x \mapsto (\mathsf{w}_t^i x, v)]}[\textsc{UnBuff}]$$

$$\frac{ts, m \xrightarrow{\lambda}_{\mathsf{synch}} ts', m' \quad b(t) = []}{ts, b, m \xrightarrow{\lambda} ts', b, m'}[\textsc{Synch}]$$

Figure 6.10: $\mathsf{BMM_o}$ machine (labeled transition system)

$$\frac{ts(t) \xrightarrow{\mathsf{r}_t^i x|V(w)} s \quad w = m(x) \quad \mathsf{volatile}(x)}{ts, m \xrightarrow{\mathsf{r}_t^i x|w}_{\mathsf{synch}} ts[t \mapsto s], m}[\textsc{ReadV}]$$

$$\frac{ts(t) \xrightarrow{\mathsf{w}_t^i x,v} s \quad \mathsf{volatile}(x)}{ts, m \xrightarrow{\mathsf{w}_t^i x,v}_{\mathsf{synch}} ts[t \mapsto s], m[x \mapsto (\mathsf{w}_t^i x, v)]}[\textsc{WriteV}]$$

$$\frac{ts(t) \xrightarrow{\mathsf{b}_t} s}{ts, m \xrightarrow{\mathsf{b}_t}_{\mathsf{synch}} ts[t \mapsto s], m}[\textsc{Begin}] \qquad \frac{ts(t) \xrightarrow{\mathsf{e}_t} s}{ts, m \xrightarrow{\mathsf{e}_t}_{\mathsf{synch}} ts[t \mapsto s], m}[\textsc{End}]$$

$$\frac{ts(t) \xrightarrow{\mathsf{l}_t^i l} s}{ts, m \xrightarrow{\mathsf{l}_t^i l}_{\mathsf{synch}} ts[t \mapsto s], m}[\textsc{Lock}] \qquad \frac{ts(t) \xrightarrow{\mathsf{u}_t^i l} s}{ts, m \xrightarrow{\mathsf{u}_t^i l}_{\mathsf{synch}} ts[t \mapsto s], m}[\textsc{Unlock}]$$

$$\frac{ts(t) \xrightarrow{\mathsf{s}_t t'} s \quad t' \notin dom(ts)}{ts, m \xrightarrow{\mathsf{s}_t t'}_{\mathsf{synch}} ts[t \mapsto s][t' \mapsto \mathrm{Ready}], m}[\textsc{Spawn}] \qquad \frac{ts(t') = \mathrm{Done} \quad ts(t) \xrightarrow{\mathsf{j}_t^i t'} s}{ts, m \xrightarrow{\mathsf{j}_t^i t'}_{\mathsf{synch}} ts[t \mapsto s], m}[\textsc{Join}]$$

Figure 6.11: $\mathsf{BMM_o}$ machine (synchronization actions)

performed yet at this address we will retrieve the default value for this address (see the notion of initial memory below).

The $\mathsf{BMM_o}$ machine is defined as a labeled transition system where steps are labeled by operational actions. The salient semantic rules are given in Figure 6.10. In all rules (except Buff) the $\mathsf{BMM_o}$ machine makes a step that the intra-thread semantics can match. Rule Tau corresponds to a intra-thread silent step (when e.g. the thread is manipulating its local registers). The $\mathsf{BMM_o}$ memory state does not change in this case. On a non-volatile reading step (rule Read), the value is obtained from the memory state using the above defined rd function. For this specific action, the intra-thread semantics event is a pair $\mathsf{r}_t^i x \mid v$ that is composed of a read action and a value. The intra-thread semantics will accept any arbitrary value here but the purpose of the rule is to constrain it using thread-local buffers and shared memory. The $\mathsf{BMM_o}$ transition label records with the read action $\mathsf{r}_t^i x$ the write action it has seen. On a non-volatile writing step (rule Write), the write action is put onto the buffer of the thread. A write action can be unbuffered at any time, in which case the write action is committed into shared memory (rule UnBuff). All synchronizing actions are emitted by threads whose buffers are empty. They are gathered under the Synch rule whose definition (relying on a relation $\dot{\rightarrow}_{\mathsf{synch}}$) is expanded in Figure 6.11. Reads from and writes to volatile locations directly access the memory so that all threads have a consistent view of them (rules ReadV and WriteV). When a thread ends (rule End), its state is kept in the $\mathsf{BMM_o}$ state, enabling other threads to join it (rule Join), and preventing it to from being restarted (rule Spawn).

We then define a $\mathsf{BMM_o}$ execution as a constrained operational execution that is accepted by the $\mathsf{BMM_o}$ machine: the input trace is properly locked and can be executed by the machine, with the intended meaning that the input execution is intra-thread consistent.

Definition 6.13 ($\mathsf{BMM_o}$ execution)**.** *An operational execution (P, tr) is in $\mathsf{BMM_o}(P)$ if there exists states $s_0, s_1, \ldots, s_n$ in* State *satisfying the following:*

- *tr is properly locked (see Definition 6.5, using $\xrightarrow{tr}$ instead of $\xrightarrow{po}$ and $\xrightarrow{so}$)*
- *s_0 is an initial state: the memory maps every address to the corresponding default write ($\forall x, m(x) = \mathsf{w}_0 x$), buffers are empty and $s_0.ts$ is defined for exactly one thread, mapping it to the* Ready *state*
- *$tr = a_1 :: \cdots :: a_n \in \mathrm{list}(\mathbb{A}_{\mathrm{op}})$*
- *for all $i \in \{1, \ldots, n\}, s_{i\text{-}1} \xrightarrow{a_i} s_i$*

The $\mathsf{BMM_o}$ behaviors of a program P are the external action traces obtained by projecting all executions of P accepted by the $\mathsf{BMM_o}$ machine on $\mathbb{A}_{\mathrm{x}}$:

$$Obs_{\mathsf{o}}(P) = \{tr \downarrow_{\mathbb{A}_{\mathrm{x}}} \mid (P, tr) \in \mathsf{BMM_o}(P)\}$$

6.6 BMM and $\mathrm{BMM_o}$ are equivalent

In this section, we show that BMM and $\mathsf{BMM_o}$ are equivalent relaxed memory models: they allow the exact same set of behaviors for any program.

Theorem 6.4. *For all program P, $Obs_{\mathsf{o}}(P) = Obs(P)$.*

The proof is somewhat technical, and relies on auxiliary lemmas. Hence, we explain the main ideas, giving some insight about the technical arguments, and provide a full proof in

Appendix 8.3. We define an operator ρ that, given an operational execution, ρ builds an equivalent axiomatic execution.

Definition 6.14 (ρ operator)**.** *Let $E_{\mathsf{o}} = (P, tr)$ be an operational execution. The operator ρ is defined as $\rho(E_{\mathsf{o}}) = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ where*

- *A is the set of non-silent actions in tr*
- *for all $a, b \in A$, $a \xrightarrow{po} b$ iff $T(a) = T(b)$ and $a \xrightarrow{tr} b$*
- *for all $a, b \in A$, $a \xrightarrow{so} b$ iff $a, b \in \mathbb{A}_{\mathrm{s}}$ and $a \xrightarrow{tr} b$*
- *for all pairs $\mathsf{r}^i_t x \mid w$ in tr, $W(\mathsf{r}^i_t x) = w$.*

Based on this operator, we define auxiliary notions that we use in the proof (and nowhere else): an operational execution E_{o} is well-formed if $\rho(E_{\mathsf{o}})$ is well-formed, and it is SC_ρ if $\rho(E_{\mathsf{o}})$ is SC. Similarly, we define operational reorderings relying on ρ: an execution E'_{o} is an operational WR*R local reordering of E_{o} if $\rho(E_{\mathsf{o}}) \xrightarrow{\mathrm{WR^\star R}} \rho(E'_{\mathsf{o}})$. We will abuse notations and write it $E_{\mathsf{o}} \xrightarrow{\mathrm{WR^\star R}} E'_{\mathsf{o}}$, and also lift this notion to trace reorderings $\xrightarrow{RO}$.

The operator ρ bridges the gap between BMM and $\mathsf{BMM_o}$: we will show that $\rho(\mathsf{BMM_o}) = \mathsf{BMM_r}$. Each inclusion is proved, in its own subsection, and Theorem 6.4 follows nicely thanks to the following lemma, which trivially holds by definition of ρ:

Lemma 6.5. *Let $E_{\mathsf{o}} = (P, tr)$ and $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ two executions such that $E = \rho(E_{\mathsf{o}})$. Then $tr{\downarrow_{\mathbb{A}_{\mathrm{x}}}} = \xrightarrow{so}{\downarrow_{\mathbb{A}_{\mathrm{x}}}}$.*

To lighten the notation, we keep implicit the unique identifier of actions, as it is clear from the context that they are all unique, and we will write w^t_x for $\mathsf{w}^i_t x, v$ (thus also omitting the value v). When considering operational actions in a trace we will generally omit the write action w attached to a read action.

6.6.1 $\rho(\mathsf{BMM_o}) \subseteq \mathsf{BMM}$

In order to justify this inclusion we must prove that every $\mathsf{BMM_o}$ execution trace can be reordered into a SC execution trace. This is captured by the following lemma:

Lemma 6.6. *Let $E_{\mathsf{o}} = (P, tr) \in \mathsf{BMM_o}(P)$. Then there exist P', tr' such that $E_{\mathsf{o}} \xrightarrow{RO} (P', tr')$, with $(P', tr') \in \mathsf{BMM_o}(P')$ is SC_ρ*

Before going into more details in the proof of Lemma 6.6, let us show how it can help prove the first inclusion.

Corollary 6.7. $\rho(\mathsf{BMM_o}) \subseteq \mathsf{BMM}$.

Proof. Let $E_{\mathsf{o}} = (P, tr) \in \mathsf{BMM_o}(P)$. By Lemma 6.6, we have $E_{\mathsf{o}} \xrightarrow{RO} E'_{\mathsf{o}}$, with $E'_{\mathsf{o}} = (P', tr') \in \mathsf{BMM_o}(P')$ is SC_ρ. By definition, $\rho(E_{\mathsf{o}}) \xrightarrow{RO} \rho(E'_{\mathsf{o}})$ and $\rho(E'_{\mathsf{o}})$ is SC. Hence, $\rho(E_{\mathsf{o}}) \in \mathsf{BMM}(P)$. □

Let us now briefly sketch how the proof of Lemma 6.6 is conducted. It heavily relies on a reordering scheme performed on operational executions in $\mathsf{BMM_o}$:

Lemma 6.8. *Let* $E_{\mathsf{o}} = (P, tr) \in \mathsf{BMM_o}(P)$*, with*

$$tr = \alpha \cdot [\mathsf{w}_t x] \cdot \beta$$

an execution such that $\overline{\mathrm{B}}(\mathsf{w}^t_x) \notin \beta$*. We write*

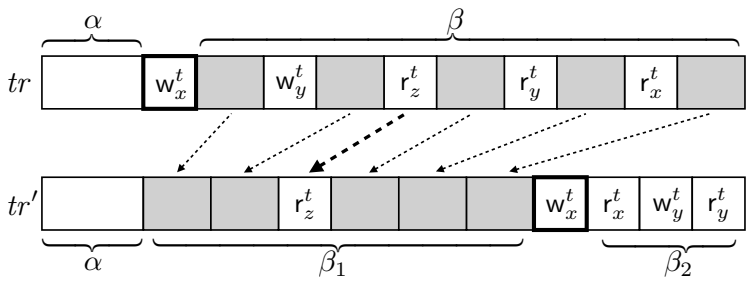
- $\mathcal{W}^t = \{\mathsf{w}_t y \in \beta \mid y \in \mathbb{X}\}$ *the set of write actions in* β *that belong to thread* t
- $\mathcal{R}^t = \{\mathsf{r}_t y \in \beta \mid \mathsf{w}_t x \cdot \beta = \gamma_1 \cdot \mathsf{w}_t y \cdot \gamma_2 \cdot \mathsf{r}_t y \cdot \gamma_3,\ y \in \mathbb{X},\ \gamma_1, \gamma_2, \gamma_3 \in \mathrm{list}(\mathbb{A}_{\mathrm{op}})\}$ *the set of read actions in* β *that see a write performed by thread* t *in* $[\mathsf{w}^t_x] \cdot \beta$
- $\beta \setminus (\mathcal{W}^t \cup \mathcal{R}^t)$ *the remaining actions in* β*.*

Then, there exist P', β_1, β_2 *such that* $E'_{\mathsf{o}} = (P', tr') \in \mathsf{BMM_o}(P')$*,* $E_{\mathsf{o}} \xrightarrow{RO} E'_{\mathsf{o}}$ *and*

$$tr' = \alpha \cdot \beta_1 \cdot [\mathsf{w}_t x] \cdot \beta_2$$

- $\beta_1 = \beta{\downarrow}_{\beta \setminus (\mathcal{W}^t \cup \mathcal{R}^t)}$
- β_2 *contains the elements of* $\mathcal{W}^t \cup \mathcal{R}^t$
- $[\mathsf{w}^t_x] \cdot \beta_2$ *matches the pattern* $(\mathsf{w}^t_{x_1}; (\mathsf{r}^t_{x_1})^*) \cdot \ldots \cdot (\mathsf{w}^t_{x_n}; (\mathsf{r}^t_{x_n})^*)$
- *for all trace* δ*, if* $(P, tr \cdot \delta) \in \mathsf{BMM_o}(P)$ *then* $(P', tr' \cdot \delta) \in \mathsf{BMM_o}(P')$*.*

We do not detail the proof here (see Appendix 8.3), but rather give an intuition about the reordering scheme. This lemma is applied on a part of an execution during which a given write action, performed by say thread t, stays in its buffer. This is illustrated, with the notations of the lemma, by the following figure, where the grey regions denote subsequences of action whose owning thread is not t. The bold stroke action $\mathsf{w}^i_t x, v$ (we omit v in the figure) is the write action that remains in t's buffer until the end of tr. We will use this action as a pivot on all the actions performed in the rest of tr, so that the resulting trace tr' is as illustrated: (a) all grey actions are shifted before the pivot, remaining in the same relative order, by changing the interleaving; (b) actions of thread t are handled with the WR*R reordering rule. Because write actions of t cannot be moved, they are kept to the right of the pivot.



Handling read actions of t is more involved: either they see a write that occurred (necessarily in thread t) after the pivot in tr, and they are accumulated in a pattern $(\mathsf{w}^{\cdot}_t x_1, \cdot; (\mathsf{r}^{\cdot}_t x_1)^*) \cdot \ldots \cdot (\mathsf{w}^{\cdot}_t x_n, \cdot; (\mathsf{r}^{\cdot}_t x_n)^*)$, or they see a write that occurs before the pivot, and WR*R is applied repeatedly on this pattern, until they are swapped before $\mathsf{w}^i_t x, v$.

Proof of Lemma 6.6 is then conducted by induction on the length of the operational execution, and extensively uses the reordering Lemma 6.8.

Transformation	SC	JMM	BMM
Trace preserving transformation	✓	✓	✓
Reordering normal memory accesses	×	✓	⊗
Redundant read after read elimination	✓	×	✓
Redundant read after write elimination	✓	✓	✓
Irrelevant read elimination	✓	✓	✓
Irrelevant read introduction	✓	×	✓
Redundant write before write elimination	✓	✓	✓
Redundant write after read elimination	✓	×	×
Roach motel reordering	✓	×	⊗

We write ✓ if the transformation is generally valid and × when it is generally wrong. We write ⊗ if only some forms of the transformation hold: only WR⋆R applies to normal memory accesses; a read can be delayed past a lock and a write can take over an unlock.

Table 6.2: Validity of transformations in memory models.

6.6.2 BMM $\subseteq \rho(\mathbf{BMM_o})$

We reuse here the post-fixpoint characterization of BMM (Lemma 6.1).

We first show that $\rho(\mathsf{BMM_o})$ contains all SC axiomatic executions. Consider an SC execution $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle \in \mathsf{BMM}$. Then there exists a total order $\xrightarrow{to}$ on A, compatible with $\xrightarrow{po}$ and $\xrightarrow{so}$ such that all read actions in A see the last write to their address w.r.t. $\xrightarrow{to}$. We claim that $E_o = (P, tr)$ can be build, with $tr \downarrow_{\mathbb{A}} = \xrightarrow{to}$, and that $E_o \in \mathsf{BMM_o}(P)$. Silent actions are inserted in $\xrightarrow{to}$ so that each write action is immediately unbuffered, and that tr is intra-thread consistent – an equivalent condition was required for $E \in \mathsf{BMM}(P)$. Finally $\rho(E_o) = E$.

Let us show that $\rho(\mathsf{BMM_o})$ is backward-closed by WR⋆R. Let E and E' two well-formed axiomatic executions such that $E' \in \rho(\mathsf{BMM_o})$ and $E \xrightarrow{\text{WR}^\star\text{R}} E'$. $E' \in \rho(\mathsf{BMM_o})$ so there exists $E'_o \in \mathsf{BMM_o}$ such that $E' = \rho(E'_o)$. In Section 6.7, we show that WR⋆R is a valid transformation under $\mathsf{BMM_o}$, meaning that there exists $E_o \in \mathsf{BMM_o}$ such that $\rho(E_o) = E$. Hence, $E \in \rho(\mathsf{BMM_o})$.

6.7 Validity of transformations

One of the objectives of any Java memory model is to take into account the reorderings performed by the hardware and to allow compilers to perform some program transformations that deal directly with memory accesses or locks.

In Table 6.2, we list standard transformations [ŠA08, Šev09] and provide informations about their validity under various memory models. The first two columns are due to Ševčík , the last column gives the corresponding results for BMM.

Our proof methodology is as follows. For a proof of validity we rely on the operational model: we consider a $\mathsf{BMM_o}$ trace of a transformed program and show there exists a valid $\mathsf{BMM_o}$ trace of the original program with the same behavior. For a proof of invalidity, we provide a counter-example and use the intuitive reordering memory model of BMM: given a program P and a transformed program P', we show that there exist an execution that is valid

for P' but invalid for P (both under BMM).

These results demonstrate that, despite its restricted set of reorderings, BMM allows some useful transformations. Among these transformations, the validity of the local reordering WR⋆R is crucial for the memory model inclusion shown in Section 6.6.2. Below, we explain how we proceed for the basic reordering WR and WR⋆R.

6.7.1 Validity of WR and WR⋆R

Definition 6.15 (Valid reordering)**.** *A local reordering $\xrightarrow{\Phi}$ between axiomatic executions is said to be valid with respect to $\mathsf{BMM_o}$ if for every axiomatic execution E and every operational execution E_o, $E \xrightarrow{\Phi} \rho(E_o)$ and $E_o \in \mathsf{BMM_o}$ implies that there exists $E'_o \in \mathsf{BMM_o}$ such that $E = \rho(E'_o)$.*

Lemma 6.9. $\xrightarrow{\text{WR}}$ *is valid.*

Proof. Let $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ be an axiomatic execution and let $E_o = (P', tr')$ be an operational execution such that $E \xrightarrow{\text{WR}} \rho(E_o)$ and $E_o \in \mathsf{BMM_o}$. By hypothesis, the trace tr' is of the form $tr' = \beta \cdot [\mathsf{r}^j_t x] \cdot \gamma \cdot [\mathsf{w}^i_t y, v] \cdot \delta$ and γ does not contain any action owned by t except some unbuffering actions.

The write action $\mathsf{w}^i_t y, v$ can be performed just after $\mathsf{r}^j_t x$ since the unbuffering in γ are independent of it and no read action in γ can see $\mathsf{w}^i_t y, v$ (it is still in its buffer). Hence the trace $\beta \cdot [\mathsf{r}^j_t x] \cdot [\mathsf{w}^i_t y, v] \cdot \gamma \cdot \delta$ is still in $\mathsf{BMM_o}$ for the same value-seen and write-seen information.

After a swap we obtain a trace $tr'' = \mathsf{b}_t a \cdot [\mathsf{w}^i_t y, v] \cdot [\mathsf{r}^j_t x] \cdot \gamma \cdot \delta$ that belongs to $\mathsf{BMM_o}(P)$ (by definition of $\xrightarrow{\text{WR}}$) and $\rho(P, tr'') = E$. □

Lemma 6.10. $\xrightarrow{\text{WR}^\star\text{R}}$ *is valid.*

Proof. Let $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W\rangle$ be an axiomatic execution and let $E_o = (P', tr')$ be an operational execution such that $E \xrightarrow{\text{WR}^\star\text{R}} \rho(E_o)$ and $E_o \in \mathsf{BMM_o}$. By hypothesis, the traces tr' is of the form $tr' = \beta \cdot [\mathsf{r}^j_t x] \cdot \gamma \cdot [\mathsf{w}^i_t y, v] \cdot \gamma_1 \cdot [\mathsf{r}^{i_1}_t y] \cdots \gamma_n \cdot [\mathsf{r}^{i_n}_t y] \cdot \delta$. Each $\gamma, \gamma_1, \ldots, \gamma_n$ does not contain any action owned by t except some unbuffering actions.

As in the previous proof, the write action $\mathsf{w}^i_t y, v$ can be performed just after $\mathsf{r}^j_t x$. All read actions $\mathsf{r}^{i_1}_t y, \ldots, \mathsf{r}^{i_n}_t y$ see the write $\mathsf{w}^i_t y, v$ in tr' and then they can also be performed earlier. The trace $\beta \cdot [\mathsf{r}^j_t x] \cdot [\mathsf{w}^i_t y, v] \cdot [\mathsf{r}^{i_1}_t y] \cdots [\mathsf{r}^{i_n}_t y] \cdot \gamma \cdot \gamma_1 \cdots \gamma_n \cdot \delta$ is still in $\mathsf{BMM_o}(P')$ for the same value-seen and write-seen (the moved reads see $\mathsf{w}^i_t y, v$ directly from t's buffer).

We then conclude with a swap: the trace $tr'' = \mathsf{b}_t a \cdot [\mathsf{w}^i_t y, v] \cdot [\mathsf{r}^{i_1}_t y] \cdots [\mathsf{r}^{i_n}_t y] \cdot [\mathsf{r}^j_t x] \cdot \gamma \cdot \gamma_1 \cdots \gamma_n \cdot \delta$ belongs to $\mathsf{BMM_o}(P)$ and $\rho(P, tr'') = E$. □

6.7.2 Proving transformations invalid

We argue that the reordering nature of BMM also helps understanding why certain transformations are invalid under it. As an example, we consider here the redundant-write-after-read-elimination (in the example below, v is a volatile address).

$$
\begin{array}{c}
x \leftarrow 0; y \leftarrow 0 \\
\hline
\begin{array}{c||c}
r_1 \leftarrow x & x \leftarrow 1 \\
y \leftarrow 1 & v \leftarrow 0 \\
x \leftarrow r_1 & r_2 \leftarrow y \\
r_3 \leftarrow x &
\end{array} \\
r_1 = r_2 = 0 \text{ and } r_3 = 1 \text{ invalid}
\end{array}
\quad
\xrightarrow[\text{after read elim.}]{\text{redund. write}}
\quad
\begin{array}{c}
x \leftarrow 0; y \leftarrow 0 \\
\hline
\begin{array}{c||c}
r_1 \leftarrow x & x \leftarrow 1 \\
y \leftarrow 1 & v \leftarrow 0 \\
 & r_2 \leftarrow y \\
r_3 \leftarrow x &
\end{array} \\
r_1 = r_2 = 0 \text{ and } r_3 = 1 \text{ valid}
\end{array}
$$

On the left, we can easily understand that the execution is not valid because it is not SC and no reordering WR⋆R is possible. After the redundant write elimination, the execution is valid because the read $r_3 \leftarrow x$ can be reordered with the write $y \leftarrow 1$ to give a SC execution leading to the configuration. This transformation thus introduced new behaviors (it is invalid).

6.8 Empirical evaluation of BMM

We have shown that the BMM is more restrictive than the JMM. It is, therefore, natural to ask how severe these restrictions are in practice, i.e. what is the performance impact imposed by BMM when incorporated within a production virtual machine with an optimizing compiler. The precise answer will have to wait until we complete our work and provide a fully verified, BMM-compliant JVM machine. In the meantime, we present the summary of the results of a preliminary study on the overheads it imposes, compared to JMM, when the target architecture is TSO.

Setup Our experiment is as follows. A production ahead-of-time Java compiler, implementing JMM, the Fiji Compiler [PZB+10] compiles Java Bytecode to C code. Then, we turn it into a BMM compliant compiler in the following way. Optimizations performed at the Java level are modified to be BMM compliant (for instance, redundant code elimination has been modified so as to be performed over local operations only). The back-end of the compiler is switched from GCC to LLVM_{TSO}, an LLVM branch with the optimizations either modified or disabled to preserve the TSO memory model [MSM+11]. We then compare the execution times of programs compiled by both compilers. We expect this will result in reduced optimizations and higher numbers of fences. The results are an upper bound on the costs of BMM, since fences are not optimized away (some of them may be redundant).

We have run the SPECjvm98 benchmark suite. While there is little concurrency in these applications, they are well suited to exhibit the overheads of missed optimization opportunities and superfluous fences. The experiments were run on two Mac OS X 10.7.3 machines with a TSO underlying architecture: an Intel Core 2 Duo 2.4 GHz, 4 GB memory, and an Intel Quad-Core Xeon 2.66 GHz processors, 16 GB memory. All benchmarks were executed for 15 iterations (first 5 iterations for warm-up). We take the mean of the last 10 iterations. The standard deviation was negligible.

Results We summarize the results here (the complete results are provided separately in the on-line material). First, adding TSO support to LLVM and modifying CSE has almost no impact on performance. On both architectures the largest slowdown is 5.6% and the average is 1%. On this set of benchmarks and on TSO architectures, BMM seems to be performance neutral.

Second, turning on biased locking [KKO02] in Fiji (after having made it BMM compliant) results in a maximum speedup of 25% at best (and on average 9%) on our Core 2 Duo, while a JMM-compliant implementation of biaised locking brings a speedup of 30% (and an average of 11%). Surprisingly, on the Quad-Core Xeon, biased locking *decreases* performance: the benchmarks are slowed down by an average of 21% for the BMM version, and of 9% for the JMM version.

Discussion It is difficult to draw definitive conclusions, but these preliminary results show that by picking optimizations carefully, the performance impact of BMM on TSO architectures can be made to be negligible. On the Xeon, with biased locking turned off, the average slowdown is less than 1%. On the Duo, the BMM with biased locking is 2% slower than the JMM version.

6.9 Related work

Weak memory model formalizations Work in this area has focused primarily on characterizing hardware memory models. Early studies [AG96, AA93] outlined a range of hardware memory models, and attempted to rigorously formalize the vendor's documentation. More recent work [HKV97, SSZN+09, OSS09] define memory models axiomatically using event structures and partial order relations. The reorderings permitted by the model are expressed by constraint relaxations. Alglave et al. [AMSS10] defined a general framework for formalizing hardware models using partial orders. Provably equivalent operational models were later added [SSZN+09, OSS09]. Burckhardt *et al.* [BMS10] define an expressive denotational framework where a memory model is a set of dynamic reorderings, aggregations or splittings, given as rewriting rules. TSO boils down to a store-load reordering and a store-load aggregation rule. The advantages of the BMM is that it provides provably equivalent axiomatic and operational models which makes it both more intuitive and more suitable to verification.

Language memory models Languages like Java and C++ have sophisticated memory models that guide questions related to data visibility and updates for concurrent, potentially racy, programs. The Java Memory Model does this using notions of committing-sequences, which make it subtle, complex, and formally broken [CKS07, ŠA08]. Huisman and Petri [HP07] have formalized the JMM in Coq, and proved its DRF guarantee. They tackled the inconsistencies or underspecified notions of [MPA05], related to memory initialization, by adding some hypotheses to their model. Aspinall and Ševčík [AŠ07a] propose an alternative definition of the JMM, in Isabelle, that does not suffer from these issues, and also prove the DRF guarantee. They restrict their definition to the finite case, as we do in this work. Recently, Lochbihler [Loc12] further extends the Isabelle formalization by including infinite executions and dynamic allocations. This work proves type-safety, but only for correctly synchronized programs. Such a mechanized proof is a *tour de force* since it covers a very large fragment of Java. However, his proof is quite different from the simulation proof we need to perform to prove compiler correctness.

There has also been recent work on defining and formalizing memory models for C++. Boehm and Adve [BA08] provide an axiomatic semantics for data race free programs that nonetheless defines a precise semantics for all of concurrency features found in the language.

Batty *et. al* [BOS+11] provide an axiomatic formalization in Isabelle/HOL of the prose description of the draft standard, including the semantics of low-level atomics. A set of pre-executions is given by an operational semantics. It is then checked (by existentially quantified predicates) to be free of data-races and reads of uninitialized locations. In C++, an additional kind of race is identified, caused by the unspecified order of expression evaluation (this gives rise to "unsequenced-races", even in single-threaded programs). Finally, the set of traces is checked against consistency constraints, expressed as predicates on the relations and partial orders on actions describing the executions. They also provide a proof of the correctness of an implementation for x86-TSO, and tools that explore the semantics for litmus test examples. This work was extended to Power in [BMO+12].

Proofs, verified compilation, and weak memory models From a verified compiler perspective, an operational definition of the JMM is desirable. There are several attempts to provide such a semantics [CKS07, BP09, BP10, JPR10] but none of them has been used in a proof assistant. Our operational semantics is inspired by the TSO memory model proposed in [ŠVZN+11] which has been formalized in Coq. Ševčík [Šev09] identifies some trace transformations that are valid under the JMM. Transformations are however defined semantically. This gap is filled in [Šev11] where the program transformations are proved to be correct, but this is done under a DRF assumption. [Šev11] also identifies the need for characterizing memory models in terms of the transformations they permit, and this is the goal of the axiomatic formalization of BMM. Defining multi-threaded semantics in terms of reordering is also the approach taken by Miné [Min11]: the semantics of a multi-threaded C program is defined as the set of interleavings of the programs possibly obtained by the syntactic transformations allowed by the memory model.

6.10 Conclusion

The JMM is an ambitious attempt to provide a semantics for concurrent and, possibly racy, Java programs. It aims to provide a precise semantics that is portable across architectures and enables a variety of compiler optimizations. Unfortunately, the JMM has proven to be challenging for users to understand and for compiler writers to use. In fact, the formal statement of the model is flawed and existing Java compilers do not comply with it.

This work presents BMM, an alternative memory model for multithreaded safety-critical Java targetting x86-TSO architectures. While being not as rich as the JMM, it still permits a number of important optimizations and it has a tractable definition and intuitive semantics, easy for programmers to understand. The axiomatic characterization of this model is defined using vocabulary similar to what is used in the JMM but avoids the complex notion of race committing sequence. It is expressed using intuitive and simple memory reordering notions, making it suitable for reasoning about program transformations. Its operational instantiation is defined in terms of per-thread store buffers as found in the definition of TSO, the memory model found on x86 architectures. It can conveniently serve as a basis for a verifying compiler infrastructure that targets this platform. We also provide a proof of equivalence between the axiomatic and the operational definitions, and a DRF guarantee for BMM.

Finally, following the approach taken in [ŠVZN+11], both characterizations of BMM are parametrized by an intra-thread semantics, to be instantiated by the IRs that the compiler uses. This allows for more modularity, factorizing our results to all the IRs.

This work is the first step to achieve a further goal: developing a verified compiler infrastructure for Java. The next step would be to provide a compiler bridge between the Java BMM and TSO hardware.

An interesting perspective would be to investigate whether BMM could be augmented with other reordering rules. We could first try to gradually achieve the Partial Store Order model by adding reorderings of memory writes, and reordering atomics with writes. A possible risk is to rapidly loose a tractable operational characterization of the model. Another issue remains with architectures like ARM or Power, whose memory models seem to be out-of-scope of the reordering-based model presented here.

Chapter 7

Conclusions and perspectives

7.1 Summary

Compilers and program static analyzers are complex and large pieces of software. They process large programs to respectively produce optimized executable code and infer automatically complex properties about their execution. In practice, compilers and analyzers rely on intermediate representations (IR) of programs for engineering concerns, faster algorithms, aggressive optimizations and precision gain in analyses. The goal of this work was to understand and formalize, from a semantic point of view, the nature and the benefits of IRs.

First, we studied two IRs widely used in modern compilers and analyzers: a stackless IR for bytecode programs (BIR), and the Static Single Assignment form (SSA). We considered state-of-the-art techniques for the associated generation algorithms. Our semantic study consisted in understanding through a formalization work:

- **How an IR is built.** This amounts to (i) defining its formal semantics (ii) finding an adequate semantics preservation for its generation, and (iii) proving the generation correct with respect to this criteria. BIR is proved correct with respect a thorough description of the semantic mapping. For the SSA generation, we rely on a provably correct a posteriori validator. The SSA destruction is proved correct in a direct way.
- **How an IR is used.** We have (i) identified structural and semantics high-level properties and then (ii) exploited these properties for facilitating the definition and the proof of subsequent analyses and optimizations. In BIR, making the expressions available in the code simplifies the definitions of Java bytecode analyses, sometimes allowing a non-negligible gain in precision. As a corollary of our semantic preservation result for BIR, we can show that expressions evaluate without error. As for SSA, we rely mainly on two properties for proving leading SSA-based optimizations such as Constant and Copy Propagation or GVN-based CSE. First, SSA programs are *strict*, meaning that all points of use of a variable are dominated by its definition. Second, the *equational lemma* proves that each variable definition `x := e` can be interpreted as an equation that is valid in the region dominated by the definition point of `x`.

Then, we investigated a feature that modern compilers and analyzers have to deal with: concurrency. Conducting the above semantic study for concurrent Java requires to reason about its memory model (JMM). Unfortunately, the JMM is very complex, because it was meant to (i) give semantics to all programs, including the racy ones, (ii) allow aggressive optimizations on memory accesses and (iii) be portable accross all multiprocessor architectures. The JMM is hard to understand, even for experts, and is formally broken (it disallows some optimizations one would expect to be valid, and [CKS07] points out that the DRF guarantee does not hold in some cases). To make a first step towards a formal reasoning about Java

concurrency, we propose BMM. It is a subset of the JMM that can be fully characterized by the reorderings it allows, can be given an operational semantics amenable to formal proof, but nonetheless can be efficiently implemented on x86 and other TSO architectures.

Implementations We also provide some practical results attesting that our formalization work deals with realistic languages and problematics.

The Sawja's IRs implementation takes into account the full (sequential) JBC language. This part corresponds to 7,000 LoC (including comments) written in OCaml, for a total of 18,000 LoC for Sawja. On these 7,000 LoC, type and interface declarations (`.mli` files) represents 1,500 LoC, mostly dedicated to user documentation (1,000 LoC). We have experimentally validated the IR generation, in terms of efficiency and effectiveness. It shows that the generation time and the compactness of the code produced are competitive with Soot, a state-of-the-art JBC optimization and analysis framework. We also show that relying on BIR can increase significantly the precision of simple static analyses. Sawja has been used for developping several JBC analyses [JKP11, HJMP10]. Using the experience of Sawja, the Celtique research group is currently developing a static analyzer for Javacard (Java for smart-cards), aiming at verifying the conformance of basic applications to cardlet development and security guidelines. Sawja's IRs and the associated generation algorithms have been adapted for this industrial transfer.

Our second implementation is a fully verified SSA middle-end, plugged at the level of RTL into the CompCert C compiler. This represents around 18,000 lines of Coq code (7,000 lines of specification, 11,000 lines of proofs), and 3,000 lines of OCaml code (for the external SSA and GVN components and printing facilities). We also have experimentally evaluated our middle-end. The SSA validation time is 33% of the whole SSA generation phase. The GVN-based CSE detects redundancies that were not optimized by the RTL basic-block version. On the whole, the execution time of the compiled programs reflects this gain. Sometimes, the effect of the optimizations is cancelled by the register allocation phase. Our work on the SSA formalization has also some opportunities to be polished, followed-up, and hopefully extended within the Verasco French ANR project, on formal verification of compilers and abstract interpretation based static analyzers for critical embedded software.

7.2 Interactions between IRs and analyses

We now summarize and comment on the semantics preservation proved about IR generations.

7.2.1 Semantics preservation and program proof

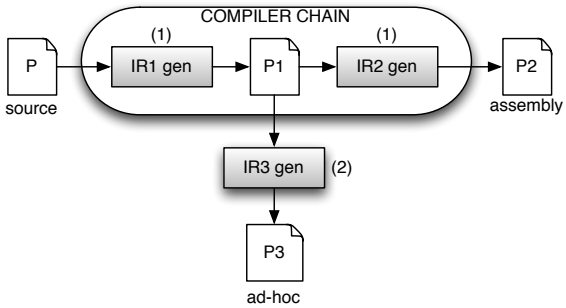
Preservation criteria Semantics preservation is formulated on top of an observational semantics. The observation semantics of a language must be carefully defined so that it can express the safety property *Safe* one wants to ensure about programs executions. The simplest safety criteria for a program can be that its execution does not go wrong: $Beh(P) \cap Wrong = \emptyset$. Sometimes, *Safe* is more precise than that, and requires to observe e.g. the sequence of function calls, or the value of local variables. In either case, $Safe \cap Wrong = \emptyset$ – a safe program does not go wrong and a going wrong program is unsafe.

In Chapter 3, we saw that the semantics preservation criteria of program transformations

differ, whether the transformation is to be used by an analysis or part of a compiler chain.

$$\begin{array}{lcl} (1) & Beh_{src}(P) \cap Wrong = \emptyset \Rightarrow Beh_{ir}(P') \subseteq Beh_{src}(P) & \textit{correctness for compiler IRs} \\ (2) & Beh_{src}(P) \subseteq Beh_{ir}(P') & \textit{correctness for analysis IRs} \end{array}$$

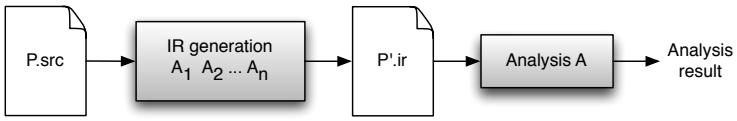
The theorem we proved about SSA is clearly of the first form, while the one for BC2BIR is roughly of the second form. Strictly speaking, it is not an inclusion, by the modification of the object allocation order, but it could be easily refined to fit this scheme by e.g. abstracting all heap-related events from the observable event traces. However, the two above theorems can be composed nicely to transfer the safety of a given program IR to the program produced by the compiler. We summarize the results with the figure on the right, these follows directly from the above criteria. We write (1) and (2) for the two forms of preservation criteria given above. The safety property *Safe* can be analyzed at any stage of the compilation chain (P1), or on an ad-hoc analysis-oriented IR (P3). By (1) and (2), if the analyzed program is safe, then the assembly program, the one that will be executed, will be safe: $Beh_{asm}(P2) \subseteq Safe$. Another thought provoking observation is that criteria (1) does not permit the safety of the IR analyzed program to be *transferred up* to the source program...



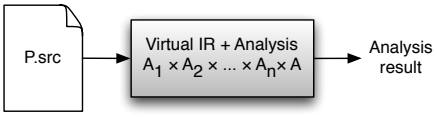
Analyses results Sometimes, lifting only the verdict of the analysis (i.e. whether a given program is safe or may be unsafe) can be not enough. This happens when a user wants to get some feed-back if the program is analyzed as unsafe. One might be interested in transfering the complete *result* of an analysis $A(P')$ back on P. This result takes the form of a program invariant. For instance, $A(P')$ can be an interval abstracting the value of each variable at each program point. In this case, the observational semantics of program should be made precise. The theorem we prove about BC2BIR observes all intermediate computations of the two programs, including local variable assigments, and expresses the semantics mapping. It thus allows to transfer such results. In its current state, the theorem we proved about SSA does not observe enough information, although the mapping between an non-optimized SSA program and the initial version could allow to transfer the result easily.

Semantic characterization of transformations There are some IRs that modify some semantic aspects of the initial program. BIR is an example: the object allocation order is not preserved. There are two options: either specifying only the semantic aspects that are strictly preserved, or keeping track of the modifications and the semantic correspondence.

This second option can result in complex preservation criteria. Defining the spectrum of target analyses can help. For instance, some simple BIR analyses on numerical domains might not require rebuilding the object construction scheme into a single instruction to reach a sufficient precision in practice. It is legitimate to think that lifting such an analysis verdict back to BC should not require dealing with a partial bijection on heaps. A simpler variant of BIR, with separate object allocation and construction (as in JBC), would lead to a simpler theorem. Indeed, the object allocation order would be preserved, the two heaps would keep equal between the two programs, and the matching semantic relations would be simpler.



(a) Generation of an IR, and then analysis of the IR.



(b) Virtual IR and simultaneous analysis

Figure 7.1: Intermediate representations as static analyses

Still, some analyses benefit from the folded constructors of BIR. For instance, the analysis of Hubert [HJMP10] checks the object initialization on JBC programs. Purely numerical analyses might use this reconstruction for a better precision by analyzing the content of object fields.

7.2.2 IR as an analysis

Our work on BIR and SSA confirms the informal definition of an IR we gave in Chapter 2: an IR is a representation of the program together with explicit results of static analyses. In BIR, the expression reconstruction is similar to the domain of symbolic expressions that one would use to analyze the bytecode with a better precision. SSA gathers in the single-definition property, strictness and the ϕ-functions some information about reaching definitions, def-use chains, and variable liveness.

Following the same idea, the IR generation algorithm would in turn correspond to the analysis itself. The BC2BIR algorithm, based on symbolic execution, closely resembles a static analysis. This makes the direct proof of the semantics correctness of BC2BIR quite elegant, the key ingredient being the abstract stack correctness relation. The SSA generation algorithm has to compute the information mentioned above efficiently. Thus, the algorithm is formulated in a more indirect way. For instance, the computation of the iterated dominance frontier relies on the dominator tree. This makes a direct proof of the SSA generation heavier. The liveness information used to compute pruned or semi-pruned SSA appear more clearly though. We therefore have opted for an a posteriori validation approach. Beyond simplifying the correctness proof, the validator expresses the global, high-level property on which the semantic preservation crucially relies.

We illustrate the point with Figure 7.1. The scenario we used so far is given Figure 7.1a. The program is first translated to its IR. The generation algorithm uses different analyses A_i on the initial program. The IR is then analyzed, producing a result. But, one could imagine working without any IR. This is the scenario of Figure 7.1b. The program is not transformed anymore. It is directly analyzed by a composition[1] of (i) the analyses A_i corresponding to a virtual IR generation and (ii) the analysis A that was performed on the IR.

This is the approach of SSA that Matsuno and Ohori propose in [MO06]: the initial

[1]This corresponds to the notion of reduced product in the theory of abstract interpretation [Cou99]: analyses are performed simultaneously on a single program. As the composition is simulatenous, the analyses can benefit one from the other. This is known to bring a more precise result than performing the analyses in sequence (and in any order).

program is only annotated with a type information representing the SSA form. Lerner et al. [LGC02] use a similar idea. They do not perform a sequence of optimizations on programs. Rather, they compose data-flow analyses and then, in a final stage, optimize the program using these results. Finally, this approach is also advocated by Logozzo and Fähndrich [LF08] for bytecode analysis via abstract interpretation. The techniques they use in their analyzer Clousot are particularly well suited to the composition and mutual refinement of analyses.

But implementing such techniques raises some engineering challenges. Therefore, most analysis and optimization tools adopt the scenario of Figure 7.1a. In particular, Logozzo and Fähndrich finally resorted to an IR generation *prior* to the analysis phase of Clousot [FL11].

7.3 Extensions

7.3.1 A verified front-end for Sawja

Mechanizing our formalization of BIR, and combining it with our work on SSA (adapted to BIR) would allow us to provide an extracted verified front-end for Sawja. We would build such a work on top of the Bicolano JVM formalization [BCG+07], developed during the European Mobius project. The parsing of class files could, at first, be a trusted (i.e. non-verified) component. We could also reasonably ignore in a first phase the subroutine inlining performed by Sawja prior to the IR generation. In the proof of the transformation, we often rely on some properties ensured by the BCV. These would need to be formalized as well. We could build upon existing work [BCDS02, KN06]. On the Coq implementation part, a technical challenge would be to achieve an extracted code for BC2BIR with performance similar to Sawja's version. In particular, the data-types used in Sawja (provided by the Javalib library) include a lot of sharing. We expect some performance loss compared to Sawja. Still, this would be acceptable for a verified piece of software. Finally, the Sawja framework does not aim at reconstructing a bytecode program, it would thus not require any destruction of the SSA form.

7.3.2 Concurrent BIR

Extending our work on BIR to concurrency would represent a first step towards the verified JBC compiler described in Chapter 6. We could use the existing Coq operational semantics of $\mathsf{BMM_o}$ to prove BC2BIR. Following the methodology inherited from CompCert TSO [ŠVZN+11], we would (i) prove the transformation on the intra-thread semantics using a forward simulation (ii) get the backward simulation result for intra-thread executions and (iii) get the full correctness result by composing the intra-thread semantics with the $\mathsf{BMM_o}$ machine.

Still, this first step would not be immediate. We have argued that formal reasoning with the operational semantics provided by $\mathsf{BMM_o}$ is easier than on the axiomatic model BMM. Still, item (iii) requires reasoning on the threads' buffers manipulations, which is not trivial. Beyond that, would BC2BIR be correct under BMM? Currently, the way `putfield`s are transformed implies storing expressions involving object fields into auxiliary local variables. These assignments seem sufficient for preserving the abstract stack correctness. However, they give rise to memory read actions that the initial `putfield` did not perform. Forward jumps on a non-empty stack can also duplicate the number of memory reads: field expressions in the abstract stack (e.g. reads performed before the branching point of a conditional in the bytecode program) are stored in temporaries at the end of *each path* that reaches the join point (e.g. the end of the conditional statement). Finally, the case of `getfield`s also requires

some care. Rebuilding expression trees (potentially involving field reads) can result in illegal reorderings of memory read actions.

7.3.3 SSA deconstruction

In its current state, the middle-end deconstructs the SSA form and the output RTL program is fed into the register allocation. The first enhancement of our SSA deconstruction consist in making it total, i.e. being able to sequentialize all ϕ-blocks, including those where ϕ-arguments of a block appear as ϕ-destination in the same block. This issue can be fixed rather immediately, relying on the work of Rideau et al. [RSL08].

Second, during our experiments, we observed that the computation time of the allocator is sometimes rather long, and can result in a lot of spill code. The quality of the allocation is essentially impacted by our current SSA deconstruction, that introduces many copies and artificial interferences between variables of a ϕ-block, imposing more constraints on the allocator. We identify two possible ways to solve this problem. First, Boissinot et al. [BDR+09] SSA deconstruction relies on the use of Conventional SSA (CSSA) [SJGS99], a variant of SSA where ϕ-blocks variables are interference free. Then, they use coalescing techniques so as not to insert too many copies when destructing CSSA, resulting in a lower register pressure for the RTL allocator. A possibility would thus be to add CSSA to the current middle-end. It would come with another global invariant about the ϕ-blocks' interferences. A second solution would be to implement a register allocator on the SSA form, as proposed by Hack et al. [HGG06]. It exploits the essential property of chordality of the SSA programs' interference graphs, leading to fast and effective algorithms (in particular the coloring becomes polynomial in the size of the graph). The allocation is performed on the SSA form of programs, the ϕ-functions are then eliminated by taking care of keeping the coloring valid for the output program. On the formal verification side, CompCert already provides the construction of the interference graph for RTL functions; it would need to be extended to ϕ-functions. Even if the algorithm for coloring a chordal graph is simpler than for an arbitrary graph, relying on a posteriori validation, as is currently done in CompCert, seems to be the more reasonable approach, given the simplicity of the validator. Thus, this could ease the handling of SSA programs by the allocator, resulting in a better allocation, as well as a smaller computation time for the allocator.

7.3.4 SSA-based optimizations

Currently, our SSA middle-end includes a simple GVN-based CSE, the other optimizations being mainly performed at the level of RTL. It would be interesting to extend our GVN with more equational reasoning by e.g. replacing ϕ-functions whose arguments are all congruent with a simple copy. An instruction like $\mathtt{x}_1 := \phi(\mathtt{x}, \mathtt{x})$ would be replaced by $\mathtt{x}_1 := \mathtt{x}$. Also, we could integrate other SSA-based optimizations to the middle-end. We could first start by porting the existing RTL optimizations, such as copy propagation, extending it to deal with ϕ-arguments. We expect that the strictness and the equational lemma will help the proof of correctness. In a second phase, we would consider adding more challenging optimizations, and see whether the equational lemma would be enough, or some variants and extensions would need to be identified. The most difficult one is Partial Redundancy Elimination (PRE) [CCK+97]. It relies on a *factored redundancy graph* (FRG), whose construction is similar to an SSA generation for expressions. Our validator could be adapted to a FRG-validator, but a substantial amount of work would remain to formalize the redundancy analyses performed on it, and the

code-placement phase for obtaining the optimized FRG. Other techniques for validating such global optimizations exist (see the work of Tristan et al. [TGM11], further discussed below).

Finally, our SSA formalization could be extended to deal with the memory, as is done, e.g. in GCC [Nov07]. Store and read instructions are annotated with use or definition of memory symbols representing memory partitions of aliased symbols. Such a construction requires an intra-procedural may-alias analysis. It needs to be reasonably precise in order to be usable by subsequent analyses or optimizations (e.g. redundant load elimination, CSE with memory variables, or instruction scheduling).

7.4 Perspective: towards more abstract IRs

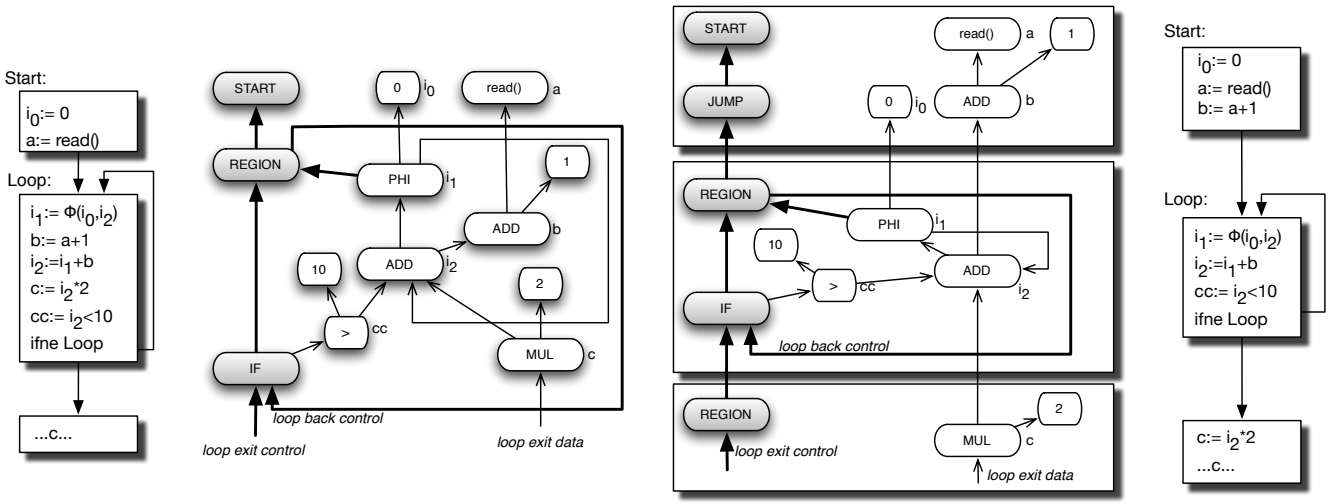
Throughout this thesis, the IRs we considered were based on the program CFG. Looking at some recent work in optimizations techniques, we observe that the CFG is often considered as too restrictive, and analyses or optimizations work on more abstract representations of programs.

One such representation is the *SSA Value Graph*, in which nodes are operators, edges encode the data-flow of values, and local variables point to their defining operator. Knoop et al. [KR00] demonstrate on a constant analysis that this IR leads to more precise results. It also leads to more efficient algorithms, that exploit the sparseness of the SSA form: the analysis iterates on the value graph, computing an abstract value at each node (potentially denoting a variable definition), instead of computing an abstract value for all the program variables at each point of the program.

The work of Tristan et al. [TGM11] shows how a similar value graph can be used as a way of validating global, powerful optimizations such as GVN with memory variables, sparse conditional constant propagation, or loop invariant code motion. In this work, the value graph (similar to the *Program Expression Graph* of Tate et al. [TSTL09]) is extended to *Monadic Gated SSA* [TP95]. This makes it possible, contrary to a simple SSA value graph that does not contain any control information at ϕ-nodes, to give it a data-flow (or equational) semantics. This representation is so semantics-full that the validation of optimizations boils down to checking the isomorphism between the normalized value graph of the program before and after optimization. Although more abstract than the program CFG, the value graph is therefore an IR that (i) can be exploited by analyses (ii) can be given a semantics (iii) represents the program faithfully enough that it can be used as a certificate to check for program equivalence.

Click's sea-of-nodes IR [Cli95a] shares some commonalities with a value graph IR, and hence similar benefits. This IR is at the heart of the Java Hotspot Server Compiler [PVC01], and allows aggressive optimizations. It is also based on SSA, extended to deal with memory (some alias information is explicit in the IR). It is illustrated in Figure 7.2. The data-flow part resembles a value graph. For instance, the definition $\mathsf{b} := \mathsf{a} + 1$ in Figure 7.2a corresponds in Figure 7.2b to a variable name b at the right of the top-right operator node ADD. The control-dependency part is handled with specific control nodes. In Figures 7.2b and 7.2c, START, REGION, IF and JUMP nodes denote basic blocks boundaries. Compared to a CFG-based IR, the basic statements do not belong to any basic block, leaving some space for a compiler to optimize the instruction scheduling. In the graph, independent statements are thus not over-constrained.

An interesting perspective would be to design a semantics for such an IR, that would keep track of the independencies of statements. A possibility would be that, from a given state,



(a)**Program** SSA form, CFG.

(b) **Sea-of-node.** Control nodes in grey, data nodes in white. Thick edges for control dependences, thin edges for data-dependences.

(c) **Scheduled sea-of-node.** Some computations are scheduled early, other late. Basic blocks indicated with shadowed boxes.

(d) **Optimized program.** SSA form, CFG after scheduling.

Figure 7.2: **Example of sea-of-node IR.** The CFG program in SSA form (7.2a) is first converted into a sea-of-node IR (7.2b), where only data and control dependences are tracked. This allows the compiler to find a better instruction scheduling. Some instructions are scheduled earlier, some other are scheduled later (7.2b). Instructions are thus reordered in the output CFG program (7.2d).

independent statements would execute separately, each of them hitting different parts of the memory and local registers, in a flavor similar to separation logic. At control dependency nodes, the two resulting states would then be combined together. Such a semantics would simplify a global reasoning about the program. Instruction scheduling in a sequential setting would hopefully come for free. Intuitively, such optimizations would not *re*-order the program anymore, but only serialize it, i.e. choose a given order for free-floating nodes.

This would require a more in-depth study, but one could also consider extending this semantics to concurrency. For Java, such a concurrent separation logic would however need to be relaxed compared to the existing ones [OHe07, HAZN08]. That is, they should allow to specify programs with races.

The optimizations found in the Java Hotspot Server Compiler were developed on top on this sea-of-node IR. The JMM definition has been tuned to take into account the aggressive reorderings implied by these optimizations. Rather contradictorily, the JMM's definition is meant to see compilers as black boxes, and hence does not exploit the nature of the underlying IRs. Understanding and formalizing what is happening in modern production Java compilers could perhaps lead to a simpler language memory model.

Appendix

Correctness of BC2BIR

Lemma 4.3 ($\mathsf{BC2BIR}_i$ 0 call-depth one-step preservation)
Suppose $\langle h, m, pc, l, s\rangle \xrightarrow{\Lambda}_0 \langle h', m, pc', l', s'\rangle$. *Let* ht, lt, as, β *be such that*

$$h \overset{\mathrm{H}}{\sim}_\beta ht \quad l \overset{\mathrm{E}}{\sim}_\beta lt \quad h, ht, lt, \beta \models s \approx as \text{ and } \mathsf{BC2BIR}_i(pc, m.\mathsf{code}[pc], as) = (\mathtt{code}, as')$$

There exist unique ht', lt' *and* Λ' *s.t* $\langle ht, m, (pc, \mathtt{code}), lt\rangle \overset{\Lambda'}{\Longrightarrow}_0 \langle ht', m, (pc', m.\mathsf{code}[pc']), lt'\rangle$

$$\text{with } h' \overset{\mathrm{H}}{\sim}_\beta ht' \quad l' \overset{\mathrm{E}}{\sim}_\beta lt' \quad \Lambda \overset{!}{\sim}_\beta \Lambda'_{proj} \text{ and } h', ht', lt', \beta \models s' \approx as'$$

Proof. We proceed by case analysis on the BC instruction at program point pc. Here, only interesting cases are detailed. Others are trivial or can be treated a similar way.

`push` c We have $\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, (\mathsf{N}\ c)::s\rangle$. Let as, ht, lt be such that $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$ and $h, ht, lt, \beta \models s \approx as$. We have $\mathsf{BC2BIR}_i(pc, \mathtt{push}\ c, as) = ([\mathtt{nop}], c::as)$. Hence, $\langle ht, m, (pc, [\mathtt{nop}]), lt\rangle \xrightarrow{\tau} \langle ht, m, (pc+1, m.\mathsf{code}[pc+1]), lt\rangle$. The heaps and environments are unchanged, both transitions are silent. Stacks stay related since $ht, lt \vDash c \Downarrow (\mathsf{N}\ c)$.

`div` The execution does not reach an error state, so $\langle h, m, pc, l, (\mathsf{N}\ n_1) :: (\mathsf{N}\ n_2) :: s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, (\mathsf{N}\ n_1/n_2)::s\rangle$ with $n_2 \neq 0$. Let as, ht, lt be such that $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$ and $h, ht, lt, \beta \models (\mathsf{N}\ n_1) :: (\mathsf{N}\ n_2) :: s \approx e_1 :: e_2 :: as$. We have $\mathsf{BC2BIR}_i(pc, \mathtt{div}, e_1 :: e_2 :: as) = ([\mathtt{notzero}(e_2)], e_1/e_2 :: as)$. But $ht, lt \vDash e_2 \Downarrow (\mathsf{N}\ n_2')$ with $(\mathsf{N}\ n_2) \overset{\mathrm{V}}{\sim}_\beta (\mathsf{N}\ n_2')$. Thus, $n_2' \neq 0$ and $\langle ht, m, (pc, [\mathtt{notzero}(e_2)]), lt\rangle \xrightarrow{\tau}_0 \langle ht, m, (pc+1, m.\mathsf{code}[pc+1]), lt\rangle$. Heaps and environment are unchanged, and both transitions are silent. Finally, since $ht, lt \vDash e_1 \Downarrow (\mathsf{N}\ n_1')$ with $(\mathsf{N}\ n_1) \overset{\mathrm{V}}{\sim}_\beta (\mathsf{N}\ n_1')$ and $ht, lt \vDash e_2 \Downarrow (\mathsf{N}\ n_2')$ with $(\mathsf{N}\ n_2) \overset{\mathrm{V}}{\sim}_\beta (\mathsf{N}\ n_2')$, we have $ht, lt \vDash e_1/e_2 \Downarrow (\mathsf{N}\ n_1'/n_2')$ and $(\mathsf{N}\ n_1/n_2) \overset{\mathrm{V}}{\sim}_\beta (\mathsf{N}\ n_1'/n_2')$.

`load` x We have $\langle h, m, pc, l, s\rangle \xrightarrow{\tau}_0 \langle h, m, pc+1, l, l(x)::s\rangle$. Let as, ht, lt, β be such that $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$ and $h, ht, lt, \beta \models s \approx as$. We have $\mathsf{BC2BIR}_i(pc, \mathtt{load}\ x, as) = ([\mathtt{nop}], x :: as)$. Hence, $\langle ht, m, (pc, [\mathtt{nop}]), lt\rangle \xrightarrow{\tau}_0 \langle ht, m, (pc+1, m.\mathsf{code}[pc+1]), lt\rangle$. Heaps and environments are unchanged and both transitions are silent. We now have to prove that stacks stay related, i.e. that $h, ht, lt, \beta \models l(x)::s \approx x::as$. We have $ht, lt \vDash x \Downarrow lt(x)$, $l \overset{\mathrm{E}}{\sim}_\beta lt$ and $x \in var$. Hence, by the definition of $\overset{\mathrm{E}}{\sim}_\beta$, we have $l(x) \overset{\mathrm{V}}{\sim}_\beta lt(x)$.

`store` x We have $\langle h, m, pc, l, v{::}s\rangle \xrightarrow{[x \leftarrow v]}_0 \langle h, m, pc+1, l[x \mapsto v], s\rangle$. Let as, ht, lt, β be such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$ and $h, ht, lt, \beta \models v{::}s \approx e{::}as$. We distinguish two cases:

- If $x \notin as$ then $\mathsf{BC2BIR}_i(pc, \texttt{store}\ x, e{::}as) = ([x := e], as)$. But $h, ht, lt, \beta \models v{::}s \approx e{::}as$ and $ht, lt \vDash e \Downarrow v'$ with $v \overset{\text{V}}{\sim}_\beta v'$. Hence $\langle ht, m, (pc, [x := e]), lt\rangle \xrightarrow{[x \leftarrow v']}_0 \langle ht, m, (pc+1, m.\mathsf{code}[pc+1]), lt[x \mapsto v']\rangle$. Now, heaps are not modified, so stay related. Labels are related: we have $x \in var$ as it appear in a bytecode instruction, and $v \overset{\text{V}}{\sim}_\beta v'$. Thus $[x \leftarrow v] \overset{!}{\sim}_\beta [x \leftarrow v']$. Environments stay related: $l[x \mapsto v] \overset{\text{E}}{\sim}_\beta lt[x \mapsto v']$ since $l \overset{\text{E}}{\sim}_\beta lt$ by hypothesis and $v \overset{\text{V}}{\sim}_\beta v'$. We finally have to prove that $h, ht, lt', \beta \models s \approx as$, where $lt' = lt[x \mapsto v']$. Stacks are the same height. Moreover, $x \notin as$, so for all abstract stack elements as_i, we have: $ht, lt' \vDash as_i \Downarrow v'_i$ and $ht, lt \vDash as_i \Downarrow v'_i$ with $v_i \overset{\text{V}}{\sim}_\beta v'_i$.
- If $x \in as$ then $\mathsf{BC2BIR}_i(pc, \texttt{store}\ x, e{::}as) = ([t^0_{pc} := x; x := e], as[t^0_{pc}/x])$. We hence have that
  $\langle ht, m, (pc, [t^0_{pc} := x; x := e]), lt\rangle \xrightarrow{[t^0_{pc} \leftarrow lt(x)]}_0 \langle ht, m, (pc, [x := e]), lt[t^0_{pc} \mapsto lt(x)]\rangle$.
  t^0_{pc} is fresh, so $t^0_{pc} \notin e$.
  Hence $ht, lt[t^0_{pc} \mapsto lt(x)] \vDash e \Downarrow v'$ where v' is such that $ht, lt \vDash e \Downarrow v'$, and $v \overset{\text{V}}{\sim}_\beta v'$ by hypothesis. Thus, we have $\langle ht, m, (pc, [t^0_{pc} := x; x := e]), lt\rangle \xrightarrow{[t^0_{pc} \leftarrow lt(x)].[x \leftarrow v']}_0 \langle ht, m, (pc+1, m.\mathsf{code}[pc+1]), lt[t^0_{pc} \mapsto lt(x), x \mapsto v']\rangle$.
  Heaps are not modified. We have $[x \leftarrow v] \overset{!}{\sim}_\beta \left([t^0_{pc} \leftarrow lt(x)].[x \leftarrow v']\right)_{proj} = [x \leftarrow v']$ because $t^0_{pc} \in tvar$ and $v \overset{\text{V}}{\sim}_\beta v'$. Environments stay related because $t^0_{pc} \in tvar$ and $x \in var$ is assigned the value v' with $v \overset{\text{V}}{\sim}_\beta v'$.
  We now have to show that $h, ht, lt', \beta \models s \approx as[t^0_{pc}/x]$, where $lt' = lt[t^0_{pc} \mapsto lt(x), x \mapsto v']$. But for all elements $as[t^0_{pc}/x]_i$ of the abstract stack, we have: $ht, lt' \vDash as[t^0_{pc}/x]_i \Downarrow v_i$ where v_i is such that $ht, lt \vDash as_i \Downarrow v_i$ because $lt'(t^0_{pc}) = lt(x)$ and t^0_{pc} is fresh, so $t^0_{pc} \notin as_i$.

`if` pc' According to the top element of the stack, there are two cases. We only treat the case of a jump, the other one is similar. We have $\langle h, m, pc, l, (\mathsf{N}\ 0){::}s\rangle \xrightarrow{\tau}_0 \langle h, m, pc', l, s\rangle$. Let as, ht, lt, β be such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$ and $h, ht, lt, \beta \models (\mathsf{N}\ 0){::}s \approx e{::}as$. We have $\mathsf{BC2BIR}_i(pc, \texttt{if}\ pc', e{::}as) = ([\texttt{if}\ e\ pc'], as)$. But stacks are related by hypothesis, thus e evaluates to zero and $\langle ht, m, (pc, [\texttt{if}\ e\ pc']), lt\rangle \xrightarrow{\tau}_0 \langle ht, m, (pc', m.\mathsf{code}[pc']), lt\rangle$ and labels are related. Heaps and environments are unchanged. Stacks stay trivially related.

`new` C We have $\langle h, m, pc, l, s\rangle \xrightarrow{[\mathsf{mayinit}(C)]}_0 \langle h', m, pc+1, l, (\mathsf{R}\ r){::}s\rangle$, with $(\mathsf{R}\ r)$ freshly allocated and $h' = h[r \mapsto \mathsf{zeros}(C)_{\widetilde{C}_{pc}}]$. Let as, ht, lt, β be such that $h \overset{\text{H}}{\sim}_\beta ht$, $l \overset{\text{E}}{\sim}_\beta lt$ and $h, ht, lt, \beta \models s \approx as$. We have that $\mathsf{BC2BIR}_i(pc, \texttt{new}\ C, as) = ([\texttt{mayinit}\ C], C_{pc}{::}as)$. Hence $\langle ht, m, (pc, [\texttt{mayinit}\ C]), lt\rangle \xrightarrow{[\mathsf{mayinit}(C)]}_0 \langle ht, m, (pc+1, m.\mathsf{code}[pc+1]), lt\rangle$. Labels are equal and environments are not modified. The reference $(\mathsf{R}\ r)$ is pointing to an uninitialized object in h', so β is not extended, and heaps keep related. Finally, we have $h', ht, lt, \beta \models (\mathsf{R}\ r){::}s \approx C_{pc}{::}as$. Indeed, we know that C_{pc} does not appear in as: by the BCV hypothesis on the bytecode program, we know that r is the only reference pointing to the uninitialized object. When executing `new` C at this point, we are thus ensured that no reference pointing to an uninitialized object of class C allocated at pc is already in the stack.

`getfield` f The execution does not reach the error state. Hence, we have $\langle h, m, pc, l, (\mathsf{R}\ r)::s\rangle \xrightarrow{\tau}_0 \langle h, m, pc{+}1, l, h(r)(f)::s\rangle$, with $h(r) = o_C$. Let e, as, ht, lt, β be such that $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$ and $h, ht, lt, \beta \models (\mathsf{R}\ r)::s \approx e::as$. We have $\mathsf{BC2BIR}_i(pc, \texttt{getfield}\ f, e::as) = ([\texttt{notnull}\ e], e.f::as)$. By hypothesis on the stacks, we have that $ht, lt \vDash e \Downarrow (\mathsf{R}\ r')$. Hence, e does not evaluates to Null and $\langle ht, m, (pc, [\texttt{notnull}\ e]), lt\rangle \xrightarrow{\tau}_0 \langle ht, m, (pc + 1, m.\mathsf{code}[pc + 1]), lt\rangle$. Heaps and environments are not modified, labels are related. We now have to show that stacks keep related. By hypothesis, we have $\beta(r) = r'$ since the object pointed to by r is initialized. Besides, $ht, lt \vDash e.f \Downarrow ht(r')(f)$ since $ht, lt \vDash e \Downarrow (\mathsf{R}\ r')$ and $ht(r')(f) = ht(\beta(r))(f)$. We know that $h \overset{\mathrm{H}}{\sim}_\beta ht$ by hypothesis, hence $h(r)(f) \overset{\mathrm{V}}{\sim}_\beta ht(\beta(r'))(f)$. Stacks are hence related.

`putfield` f We have $\langle h, m, pc, l, v::(\mathsf{R}\ r)::s\rangle \xrightarrow{[r.f \leftarrow v]}_0 \langle h[r(f) \mapsto v], m, pc + 1, l, s\rangle$ (the field of the object pointed to by r is modified), with $h(r) = o_C$. Let e, e', as, ht, lt, β be such that $h \overset{\mathrm{H}}{\sim}_\beta ht$, $l \overset{\mathrm{E}}{\sim}_\beta lt$ and $h, ht, lt, \beta \models v::(\mathsf{R}\ r)::s \approx e'::e::as$. There are two cases:

- If f is not in any expression of the abstract stack, we know $\mathsf{BC2BIR}_i(pc, \texttt{putfield}\ f, e' :: e :: as) = ([\texttt{notnull}\ e; e.f := e'], as)$. But $h, ht, lt, \beta \models v :: (\mathsf{R}\ r) :: s \approx e' :: e :: as$. We get that $v \overset{\mathrm{V}}{\sim}_\beta v'$ where $ht, lt \vDash e' \Downarrow v'$ and that there exists r' such that $ht, lt \vDash e \Downarrow (\mathsf{R}\ r')$ with $(\mathsf{R}\ r) \overset{\mathrm{V}}{\sim}_\beta (\mathsf{R}\ r')$, and r' points in ht to an initialized object, since the BC field assignment is permitted. We hence have that $\langle ht, m, (pc, [\texttt{notnull}\ e; e.f := e']), lt\rangle \xrightarrow{\tau}_0 \langle ht, m, (pc, [e.f := e']), lt\rangle \xrightarrow{[r'.f \leftarrow v']}_0 \langle ht[r'(f) \mapsto v'], m, (pc + 1, m.\mathsf{code}[pc + 1]), lt\rangle$. Environments are unchanged and stay related. We have to show that $h' = h[r(f) \mapsto v] \overset{\mathrm{H}}{\sim}_\beta ht' = ht[r'(f) \mapsto v']$. We have $(\mathsf{R}\ r) \overset{\mathrm{V}}{\sim}_\beta (\mathsf{R}\ r')$, hence $\beta(r) = r'$. Besides, $v \overset{\mathrm{V}}{\sim}_\beta v'$ with $ht, lt \vDash e' \Downarrow v'$. Fields of the two objects pointed to by r and r' have hence related values w.r.t β. Finally, we have $[r.f \leftarrow v] \overset{!}{\sim}_\beta [r'.f \leftarrow v']$ since $v \overset{\mathrm{V}}{\sim}_\beta v'$ and $(\mathsf{R}\ r) \overset{\mathrm{V}}{\sim}_\beta (\mathsf{R}\ r')$.
- If $f \in as$, $\mathsf{BC2BIR}_i(pc, \texttt{putfield}\ f, e'::e::as) = ([\texttt{notnull}\ e; Fsave(pc, f, as); e.f := e'], as[t^i_{pc}/as_i])$. But $h, ht, lt, \beta \models v::(\mathsf{R}\ r)::s \approx e'::e::as$ hence, as in the previous case: $v \overset{\mathrm{V}}{\sim}_\beta v'$ where v' is such that $ht, lt \vDash e' \Downarrow v'$ and there exists r' such that $ht, lt \vDash e \Downarrow (\mathsf{R}\ r')$ with $(\mathsf{R}\ r) \overset{\mathrm{V}}{\sim}_\beta (\mathsf{R}\ r')$.

  Suppose now that n elements of as are expressions using the field f. For all $i \in [1; n]$, let v_i be such that $ht, lt \vDash as_i \Downarrow v_i$. Thus, we have that

  $\langle ht, m, (pc, [\texttt{notnull}\ e; Fsave(pc, f, as); e.f := e']), lt\rangle \xrightarrow{[t^1_{pc} \leftarrow v_1] \ldots [t^n_{pc} \leftarrow v_n]}_0 \langle ht, m, (pc, [e.f := e']), lt[t^i_{pc} \mapsto v_i, \ldots t^n_{pc} \mapsto v_n]\rangle$.

  All t^i_{pc} are fresh, they hence do not appear in e or e'. Let $lt' = lt[t^1_{pc} \mapsto v_1, \ldots t^n_{pc} \mapsto v_n]$. Thus $ht, lt' \vDash e' \Downarrow v'$ with $ht, lt \vDash e' \Downarrow v'$ and $ht, lt' \vDash e \Downarrow (\mathsf{R}\ r')$. Thus, $\langle ht, m, (pc, [e.f := e']), lt'\rangle \xrightarrow{[r'.f \leftarrow v']}_0 \langle ht[r'(f) \mapsto v'], m, (pc{+}1, m.\mathsf{code}[pc{+}1]), lt'\rangle$. Events are related: $[r.f \leftarrow v] \overset{!}{\sim}_\beta \left([t^1_{pc} \mapsto v_1] \ldots [t^n_{pc} \leftarrow v_n].[r'.f \leftarrow v']\right)_{proj}$ because all t^i_{pc} are in $tvar$, $\beta(r) = r'$ and $v \overset{\mathrm{V}}{\sim}_\beta v'$. Environments stay related: $l \overset{\mathrm{E}}{\sim}_\beta lt'$ because all t^i_{pc} are not in var. Besides, since $\beta(r) = r'$ and $v \overset{\mathrm{V}}{\sim}_\beta v'$, we have $h[r(f) \mapsto v] \overset{\mathrm{H}}{\sim}_\beta ht[r'(f) \mapsto v']$. Finally, we have that $h', ht', lt', \beta \models s \approx as[t^i_{pc}/as_i]$, where $ht' = ht[r'(f) \mapsto v']$ and $h' = h[r(f) \mapsto v]$, by the definition of $as[t^i_{pc}/as_i]$.

□

Lemma 8.1 ($\mathsf{BC2BIR}_i$ n call-depth one-step preservation)**.**
Let $n \in \mathbb{N}$. Suppose that $\mathcal{P}(k,m)$ for all m and $k<n$, and $\langle h,m,pc,l,s\rangle \xrightarrow{\Lambda}_n \langle h',m,pc',l',s'\rangle$. Let ht, lt, as, β be such that

$$h \overset{\mathrm{H}}{\sim}_\beta ht \quad l \overset{\mathrm{E}}{\sim}_\beta lt \quad h,ht,lt,\beta \models s \approx as \text{ and } \mathsf{BC2BIR}_i(pc, m.\mathsf{code}[pc], as) = (code, as')$$

There exist unique ht', lt', Λ' and β' extending β such that

$$\langle ht, m, (pc, code), lt\rangle \overset{\Lambda'}{\Longrightarrow}_0 \langle ht', m, (pc', m.\mathsf{code}[pc']), lt'\rangle$$

$$\text{with } h' \overset{\mathrm{H}}{\sim}_\beta ht' \quad l' \overset{\mathrm{E}}{\sim}_\beta lt' \quad \Lambda \overset{!}{\sim}_\beta \Lambda'_{proj} \text{ and } h',ht',lt',\beta \models s' \approx as'$$

Proof. For $n = 0$, we use Lemma 4.3. Suppose $n>0$. We give the main cases of $m.\mathsf{code}[pc]$.

`constructor` C Here, $s = V{::}(\mathsf{R}\ r){::}s'$. By case analysis on the semantic rule, and on the condition on C, we have two cases. In the first case, we have $h(r) = o_t$ with $t = \widetilde{C}_j$. Let h_1 be the heap such that $h_1 = h[r \mapsto o_C]$. We have that

$$\langle h_1, C.\mathsf{init}, 0, [\mathtt{this} \mapsto (\mathsf{R}\ r), x_1 \mapsto v_1, \ldots, x_n \mapsto v_n], \varepsilon\rangle \overset{\Lambda_2}{\Longrightarrow}_{n\text{-}1} \langle h_2, \mathsf{Void}\rangle$$

By the abstract stack correctness, we know that as is of the form: $e_1{::}\ldots{::}e_n{::}C_j{::}as'$, and for all i, $ht', lt \vDash e_i \Downarrow v'_i$ and $v_i \overset{\mathrm{V}}{\sim}_\beta v'_i$. By definition, we have $\mathsf{BC2BIR}_i(pc, \mathtt{constructor}\ C, e_1{::}\ldots{::}e_n{::}C_j{::}as)$ produces

$$([t^1_{pc} := e'_1; \ldots; t^m_{pc} := e'_m; t^0_{pc} := \mathtt{new}\ C(e_1, \ldots, e_n)], as[t^i_{pc}/\ e_i][t^0_{pc}/C_j])$$

We follow the semantics of BIR. Let $(ht', (\mathsf{R}\ rt)) = \mathsf{newObject}(C, ht)$ and $ht' = ht[rt \mapsto \mathsf{zeros}(C)_C]$. We extend β to β' to take rt into account: $\beta'(r) = rt$, and we have that $h_1 \overset{\mathrm{H}}{\sim}_{\beta'} ht'$: objects pointed to by r and rt have the same initialization status, their class is equal, and each of their field has default values (we have $h_1(r) = \mathsf{zeros}(C)_C$ since before the call to the constructor, nothing can be done on this object). Hence, both constructors are called on related initial configurations.

We now apply $\mathcal{P}(n\text{-}1, C.\mathsf{init})$ and get β'', an extension of β' relating the two resulting heaps and constructor execution traces. From method m, the traces match by $\mathcal{P}(n\text{-}1, C.\mathsf{init})$ and $r \leftarrow C.\mathsf{init}(v_1, \ldots, v_n) \overset{!}{\sim}_{\beta''} rt \leftarrow C.\mathsf{init}(v'_1, \ldots, v'_n)$.

The heaps have been shown to be related w.r.t β'', and the environments keep related (only fresh temporary variables have been introduced). We now have to show the stack relation. Every C_j is substituted with the new temporary t^0_{pc}, which evaluates to rt with $r \overset{\mathrm{V}}{\sim}_{\beta''} rt$ after the constructor call (we exploit here the hypothesis on uninitialized references given by the abstract stack correctness.Now, concerning the storing of stack elements reading fields, they have been introduced before the execution of the constructor and none of them could have been modified. Their value hence stay related to the corresponding element of the concrete stack after the constructor call.

The second case is treaded similarly (references are already related through β, which does not need to be extended).

`invokevirtual` m We proceed similarely. The receiver objects are already initialized, hence the bijection is not extended.

□

Completeness of the SSA validator

An essential property of our type system is that it accepts all the SSA programs generated by the algorithm by Cytron *et al* [CFR+91]. In order to prove this, we will build a global typing Γ from an SSA function generated by Cytron's algorithm. We will then show that the SSA function is typable with Γ in our type-system.

Theorem 8.2 (Type system completeness)**.** *Let* `f` *be a normalized RTL program and let* `tf` *be the SSA program generated from* `f` *by Cytron et al.'s algorithm. Then there exists* Γ *such that* `SSA_validator f tf` Γ `= true`*.*

Proving this theorem requires to identify some key properties about the algorithm presented in [CFR+91], which we recall in the Section 8.2.1. Given this specification, we show in Section 8.2.2 how to build a global typing, that we prove valid in Section 8.2.3.

8.2.1 Specification of Cytron et al.'s algorithm

We first review the well-known characterization of the iterated dominance frontier as a fixpoint of the *join* operator J, as well as some properties of the Cytron et al.'s algorithm.

Definition 8.1 (Join operator J)**.** *Given a set S of nodes, $J(S)$ is defined to be the set of all nodes j such that there are two non-empty CFG paths that start at two distinct nodes in S and converge at j, i.e. they both end at j.*

Lemma 8.3 (Iterated dominance characterization)**.** *For any set of nodes S, the iterated dominance frontier of S, $DF^+(S)$ satisfies $DF^+(S) = J(S \cup DF^+(S))$.*

Proof. See [CFR+91], page 467. □

Let `f` be an RTL function, and `tf` the SSA form generated by Cytron's algorithm. For a variable x of `f`, we write def_x the set of definition points of x in `f`, and $\mathrm{def}(x)$ for the definition point of the variable. We express now the way Cytron's algorithm defines the set of definition points of the versions of x in `tf`, and how it determines the right index to use in `tf` when x is used at some point in `f`.

Lemma 8.4 (Minimal SSA - Definitions)**.** *Define $D_x = \mathrm{def}_x \cup DF^+(\mathrm{def}_x)$. D_x is the set of program points where an instance of x is defined in* `tf`*, and $DF^+(\mathrm{def}_x)$ is the set of nodes where a ϕ-function for x is inserted.*

Proof. Theorem 2 in [CFR+91], page 468. □

Lemma 8.5 (Minimal SSA - Absence of ϕ-function)**.** *If no instance of a variable x is assigned in the ϕ-block at node n, then a single definition of an instance of x reaches all predecessors of n, without any other instance of x is defined in between.*

Proof. The set of ϕ-functions required for the variable x is by definition $J^+(\mathrm{def}_x)$ [CFR+91]. We conclude using the definition of the iterated join operator J^+. □

Corollary 8.6. *If no instance of a variable x is assigned in the ϕ-block at node n, then there exists an instance x_k of x whose definition strictly dominates n.*

Proof. The definition of x_k reaches all predecessors of j, and no instance of x is defined in between. In particular, x_k is defined at a common ancestor of all the predecessors of j. $\mathrm{def}(x_k)$ dominates all predecessors of j; it thus dominates j. □

Lemma 8.7 (Minimal SSA - Uses)**.** *If x is used at point i in* `f`*, the variable x_k will be used at point i in* `tf`*, where x_k is the instance of x such that $\mathrm{def}(x_k) \in D_x$ is the closest ancestor of i in the dominator tree of* `f`*.*

Proof. See Lemmas 9 and 10 in [CFR+91], pages 473-474. □

8.2.2 Building a witness global typing

Let `f` be an RTL function, and `tf` the SSA form generated by Cytron et al's algorithm. We explain now how to build a global typing Γ by a depth-first-search (DFS) traversal of the CFG of `tf`. Each time we reach a new program point j in the DFS, one of its predecessors i in the CFG has already been treated and $(\Gamma\ i)$ is already defined. To define $(\Gamma\ j)$, we distinguish two cases:

Case 1 If j is not a join point, for every RTL variable x, we define $(\Gamma\ j\ x)$ by case analysis:

- if no instance of x is assigned at i in `tf`, then we set $\Gamma\ j\ x = \Gamma\ i\ x$;
- if some instance x_k of x is assigned at i in `tf`, then we set $\Gamma\ j\ x = k$;

Case 2 If j is a join point, for every RTL variable x, we define $(\Gamma\ j\ x)$ by case analysis on the ϕ-block b at j:

- if no instance of x is assigned in b, then we set $\Gamma\ j\ x = \Gamma\ i\ x$;
- if some instance x_k of x is assigned in b then we set $\Gamma\ j\ x = k$.

The global typing given in Figure 5.9 can actually be computed using this construction. Some properties about this witness global typing Γ can be derived, that we will use in the proof of the next paragraph.

Lemma 8.8 (Witness global typing: properties)**.** *If $(\Gamma\ i\ x) = k$, then there exists x_k such that $\mathrm{def}(x_k)$ dominates i and any shortest CFG path p from $\mathrm{def}(x_k)$ to i (excluded) does not go through another definition of an instance of x, i.e. a point in D_x.*

Proof. We proceed by induction on the construction of Γ.

- Base case. For all variable x, $(\Gamma\ \mathit{Entry}\ x) = \mathit{dft}$ for all x, and their definition point is the entry point by convention. The condition on shortest paths is trivial since it is emtpy.
- Induction case. Consider the CFG edge (i, j). We proceed by case analysis on j:

  Suppose j is not a junction point. By definition of Γ, there are two cases:

  - $(\Gamma\ j\ x) = k$ because x_k is defined at i. Here, i dominates j, and the shortest path from i to j contains only i and j.

- $(\Gamma\ j\ x) = (\Gamma\ i\ x)$ because no instance of x is defined at point i. Applying the induction hypothesis, we get that there is x_k such that $\mathrm{def}(x_k)$ dominates i and the shortest path p from $\mathrm{def}(x_k)$ to i does not go through another definition of an instance of x. But i dominates j. By transitivity of the dominance relation, we get that $\mathrm{def}(x_k)$ dominates j. The mininal path $[\mathrm{def}(x_k); \ldots; i; j]$ does not contain any other definition of an instance of x because i does not define a version of x.

Suppose now j is a junction point. There are again two cases.

- $(\Gamma\ j\ x) = k$ because an instance x_k is defined in the ϕ-block at point j. We conclude by the reflexivity of dominance.
- $(\Gamma\ j\ x) = (\Gamma\ i\ x)$ because no instance of x is defined in the ϕ-block at j. Let $(\Gamma\ i\ x) = k$. Here, the induction hypothesis does not permit to conclude.
  In this case, j is not in the iterated dominance frontier of any point in def_x (Lemma 8.5). Then, by Corollary 8.6, we get that $\mathrm{def}(x_k)$ dominates j.

□

8.2.3 The witness global typing is a correct typing

Now we prove that `tf` is typable with Γ as defined in the previous section. We first consider that `tf` has been generated with a trivial live information `full_live`, containing at each program point the set of all the RTL variables.

We consider all edges (i, j) in the CFG of `tf`, and have to prove that the property (`wf_edge f Γ full_live i j`) holds. We postpone the discussion of typing pruned and semi-pruned SSA versions at the end of the paragraph.

First, we concentrate on verifying that the constraints on the variable definitions are satisfied. We will check that the typing constraints about variables uses (predicate `use_ok` in Figure 5.10) in a separated lemma.

Lemma 8.9 (Constraints on definitions)**.** *Let (i, j) be an edge in the CFG of* `tf`*. Then* (`wf_edge f Γ full_live i j`) *holds except for constraints about variable uses.*

Proof. We distinguish two cases. We do not detail the constraints on the default index *dft*.

- Case 1. If j is not a junction point, then i is the sole predecessor of j in the CFG of `tf`, and $(\Gamma\ j)$ is defined in terms of $(\Gamma\ i)$. In this case, we apply the rule `wt_edge_not_jp`.
- Case 2. If j is a junction point. We have to prove that rule `wt_edge_jp` is applicable. We consider two cases.
  - Case 2.1. If i is the predecessor of j in the DFS traversal, $\Gamma\ j$ is defined in terms of $\Gamma\ i$, and the constraints `ASSIG` and `NASSIG` hold by definition of Γ. Therefore the edge is typable.
  - Case 2.2. Let i' be the predecessor of j in the DFS, and suppose $i \neq i'$. We have to prove that `ASSIG` and `NASSIG` hold. Let b the ϕ-block at point j.

    `ASSIG` Let x_k be assigned in b. The live information we use here is `full_live`, thus x is live at point j. Additionally, we have $(\Gamma\ j\ x) = k$ by construction.

    `NASSIG` Let x be an RTL variable such that no instance of x is assigned in block b. Because we use `full_live`, we have to show that $(\Gamma\ j\ x) = (\Gamma\ i\ x)$.

By definition of Γ, we know that $(\Gamma\ j\ x) = (\Gamma\ i'\ x)$. It is thus sufficient to prove that $(\Gamma\ i\ x) = (\Gamma\ i'\ x)$.
If the property would not hold, one could conclude from the Lemma 8.8 that there exist two distinct points ℓ and ℓ' such that a definition of an instance of x occurs in ℓ and ℓ' and there is a path from ℓ (resp. ℓ') that reaches i (resp. i') without meeting any other point in D_x. This implies that $j \in J(D_x) = DF^+(\text{def}_x)$. This leads to a contradiction, as it would mean that j holds a ϕ-node for x (Lemma 8.4). Therefore, an instance of x should be assigned by a ϕ-function in b. This is a contradiction.

This shows that `tf` is typable with Γ, except for constraints about uses. □

Lemma 8.10 (Variable uses)**.** *Let (i, j) be an edge in the CFG of* `tf`*. Whenever an instance x_k of x is used at point i in* `tf`*, we have $(\Gamma\ i\ x = k)$.*

Proof. Suppose that $(\Gamma\ i\ x = k')$, with $k' \neq k$. Then, by Lemma 8.8, we know that $\text{def}(x_{k'})$ dominates i. But x_k is used at point i. By Lemma 8.7, we hence know that $\text{def}(x_k)$ dominates i. Hence, $p_k = \text{def}(x_k)$ and $p_{k'} = \text{def}(x_{k'})$ both dominate i. Therefore, by the property of the dominance relation, either p_k dominates $p_{k'}$ or $p_{k'}$ dominates p_k. We distinguish three cases:

- Case 1. If $p_k = p_{k'}$, we can conclude directly.
- Case 2. Suppose p_k strictly dominates $p_{k'}$. In this case, p'_k would be between p_k and i in the dominator tree. Then, the closest ancestor of i in the dominator tree that belongs to D_x would be $p_{k'}$, and the index used for x at point i should be, by Lemma 8.7, p'_k. This is a contradiction.
- Case 3. Suppose $p_{k'}$ strictly dominates p_k. Then, by antisymmetry of the dominance relation, p_k does not dominate $p_{k'}$. This means that there exists a CFG path p from the entry to $p_{k'}$ that does not go through p_k.
  But $(\Gamma\ i\ x) = k'$. Thus, by Lemma 8.8, we know it exists a CFG path p' from $p_{k'}$ to i that never meets another point in D_x.
  The concatenation of p and p' gives us a path from the entry node of the CFG to i, that never goes through p_k. This contradicts the fact that p_k dominates i.

□

Corollary 8.11 (Constraints on variable uses)**.** *Let (i, j) be an edge in the CFG of* `tf`*. Then the constraints on the variable uses in* (`wf_edge f Γ full_live i j`) *are satisfied.*

Completeness with regards to pruned-SSA form can be shown easily by observing that both the algorithm and the type system make the same use of the liveness information (a dead initial variable does not require a ϕ-function).

BMM and BMM$_\mathsf{o}$ are equivalent

To lighten the notation, we keep implicit the unique identifier of actions, as it is clear from the context that they are all unique, and we will write w^t_x for $\mathsf{w}^i_t x, v$ (thus also omitting the value v). When considering operational actions in a trace we will generally omit the write action w attached to a read action. In this section, we prove the following inclusion $\rho(\mathsf{BMM_o}) \subseteq \mathsf{BMM}$.

Lemma 6.8 *Let $E_\mathsf{o} = (P, tr) \in \mathsf{BMM_o}(P)$, with*

$$tr = \alpha \cdot [\mathsf{w}_t x] \cdot \beta$$

an execution such that $\overline{\mathrm{B}}(\mathsf{w}^t_x) \notin \beta$. We note

- $\mathcal{W}^t = \{\mathsf{w}_t y \in \beta \mid y \in \mathbb{X}\}$ *the set of write actions in β that belong to thread t*
- $\mathcal{R}^t = \{\mathsf{r}_t y \in \beta \mid \mathsf{w}_t x \cdot \beta = \gamma_1 \cdot \mathsf{w}_t y \cdot \gamma_2 \cdot \mathsf{r}_t y \cdot \gamma_3,\ y \in \mathbb{X},\ \gamma_1, \gamma_2, \gamma_3 \in \mathrm{list}(\mathbb{A}_{\mathrm{op}})\}$ *the set of read actions in β that see a write performed by thread t in $[\mathsf{w}^t_x] \cdot \beta$*
- $\beta \setminus (\mathcal{W}^t \cup \mathcal{R}^t)$ *the remaining actions in β.*

Then, there exist P', β_1, β_2 such that $E'_\mathsf{o} = (P', tr') \in \mathsf{BMM_o}(P')$, $E_\mathsf{o} \xrightarrow{RO} E'_\mathsf{o}$ and

$$tr' = \alpha \cdot \beta_1 \cdot [\mathsf{w}_t x] \cdot \beta_2$$

- $\beta_1 = \beta{\downarrow}_{\beta \setminus (\mathcal{W}^t \cup \mathcal{R}^t)}$
- β_2 *contains the elements of* $\mathcal{W}^t \cup \mathcal{R}^t$
- $[\mathsf{w}^t_x] \cdot \beta_2$ *matches the pattern* $(\mathsf{w}^t_{x_1}; (\mathsf{r}^t_{x_1})^*) \cdot \ldots \cdot (\mathsf{w}^t_{x_n}; (\mathsf{r}^t_{x_n})^*)$
- *for all trace δ, if $(P, tr \cdot \delta) \in \mathsf{BMM_o}(P)$ then $(P', tr' \cdot \delta) \in \mathsf{BMM_o}(P')$.*

Proof. Let $E_\mathsf{o} = (P, tr) \in \mathsf{BMM_o}(P)$, with $tr = (\alpha \cdot [\mathsf{w}_t x] \cdot \beta)$ and $\overline{\mathrm{B}}(\mathsf{w}^t_x) \notin \beta$. The sub-trace $[\mathsf{w}^t_x] \cdot \beta$ can be decomposed as $[\mathsf{w}^t_x] \cdot \beta = \beta_p \cdot \beta_t$ with $\beta_p = [\mathsf{w}_t y; (\mathsf{r}_t y)^*]^+$ (we take the shortest β_t). We now proceed by induction on the length of β_t.

- Base case: β_t is empty. We don't need any reordering transformation to reach the expected pattern.

- Inductive case. We proceed by case analysis on the first element of $\beta_t = a :: \beta'_t$ and show that a can be either (i) integrated inside the pattern β_p or (ii) be moved before this pattern:

  - If $a = \mathsf{r}_t k$, when k is one of the addresses of β_p. Here, we integrate in β_p by applying an WR*R as many times as needed to make it part of the right $\mathsf{w}^t_k(\mathsf{r}^t_k)^*$ pattern such that $\beta_p = \beta_{p_1} \cdot (\mathsf{w}_t k(\mathsf{r}_t k)^*) \cdot \beta_{p_2}$. Note WR*R can be applied because the pattern only concerns thread t, and the visibility conditions required by WR*R are fulfilled. The right-most such pattern before a, i.e. such that there is no w^t_k in β_{p_2}, is the only one that can ensure the write-seen of a to be preserved, according to $\mathsf{BMM_o}$. Thus the resulting trace $\big(P', \alpha \cdot \beta_{p_1} \cdot (\mathsf{w}^t_k(\mathsf{r}^t_k)^* a) \cdot \beta_{p_2} \cdot \beta'_t\big)$ is $\mathsf{BMM_o}$.
    Being able to apply WR*R requires some extra precautions. We must make sure that there exists a program P' whose intra-thread semantics accepts the reordering and we must prove the resulting trace is still in $\mathsf{BMM_o}$. The first point is an

assumption we have to made on the abstract notion of program: after some local variable renaming and loop unrolling it is always possible to perform such a reordering on independent write/read accesses. The second point is easy to prove since the read we move can keep the same write-seen[1].

- If $a = \mathsf{r}_t k$, when k is not any of the addresses of β_p. We apply the same reasoning as in the previous case, rewriting the trace with WR*R, but in this case, this simply amounts to put a before w_x^t, because WR*R will be applied on the whole β_p (and the write-seen is trivially kept valid).
- If $a = \overline{\mathrm{B}}(a')$, then $a' \in \alpha$ because there is no unbuffering action in β_p. This unbuffering could have been done just before w_x^t. This unbuffering does not modify the visibility constraints of the new trace, as it cannot overwrite any of the write actions in β_p. For any trace δ, if $(P, \alpha \cdot \beta_p \cdot [a] \cdot \beta'_t \cdot \delta) \in \mathsf{BMM_o}(P)$ then the trace $(P, \alpha \cdot [a] \cdot \beta_p \cdot \beta'_t \cdot \delta)$ is also in $\mathsf{BMM_o}$ (P): any read in δ that sees a' in the first trace can still see it in the second trace because no write action in β_p is unbuffered before δ.
- For all other cases ($a = \mathsf{r}_k^{t''}$, where $t'' \neq t$, $a = \mathsf{w}_k^{t''}$, with $t'' \neq t$, and $a \in \mathbb{A}_\mathrm{s}$), we can just change the interleaving to move a before β_p and conclude easily.

For each resulting trace, we show that we are under the induction hypothesis premises. Hence, we conclude by induction.

□

The inclusion $\rho(\mathsf{BMM_o}) \subseteq \mathsf{BMM}$ then follows from Lemma 6.6 proven below, stating that the reordering interpretation of BMM can be simulated in the operational world. (The following lemma is a reinforced version of Lemma 6.6 given in Chapter 6, that make a proof by induction possible).

Lemma 6.6 *Let $E_\mathsf{o} = (P, tr) \in \mathsf{BMM_o}(P)$. Then there exist P', tr' such that $E_\mathsf{o} \xrightarrow{RO} (P', tr')$, with $(P', tr') \in \mathsf{BMM_o}(P')$ is SC_ρ and the* write-swapping *property holds on (tr, tr').*

The write-swapping property is formally below.

Definition 8.2 (Write-Swapping)**.** *Let tr and tr' two sequences of actions. The* write-swapping *property holds on (tr, tr') if for any write action w_1, w_2 to the same address,*

- *if $w_1 \xrightarrow{tr} w_2$ and $w_2 \xrightarrow{tr'} w_1$ (the write actions have been swapped during the reordering) then the trace tr is of the form $tr = \alpha \cdot [w_1] \cdot \beta \cdot [w_2] \cdot \gamma$ and $\overline{\mathrm{B}}(w_1) \notin \beta$.*
- *if $w_1 \xrightarrow{tr} w_2$ and $w_1 \xrightarrow{tr'} w_2$ (the write actions have not been swapped during the reordering) then if $\overline{\mathrm{B}}(w_1)$ occurs before w_2 in tr, it is still the case in tr'.*

Proof. We prove Lemma 6.6 by strong induction on the size n of the execution trace tr. Assume the property holds for any integer k strictlt less than n. Let $E_\mathsf{o} = (P, tr) \in \mathsf{BMM_o}(P)$ an execution of size n.

[1] It should not be confused with the fact that a *RW* transformation is invalid under BMM: here we pick an interleaving trace $\alpha \cdot [\mathsf{w}; \mathsf{r}] \cdot \gamma$ where the read and the write are adjacent. To prove the (wrong) validity of a *RW* reordering we would start from a trace $\alpha \cdot [\mathsf{w}] \cdot \beta \cdot [\mathsf{r}] \cdot \gamma$ where some interleaved actions of other threads occur between the write and the read. It would not be possible to prove then that $\alpha \cdot [\mathsf{r}] \cdot \beta \cdot [\mathsf{w}] \cdot \gamma$ is still in $\mathsf{BMM_o}$.

We assume $n > 0$ (the case $n = 0$ holds trivially). So tr is of the form $tr = tr_1 \cdot [a]$. By induction on tr_1, we get P_2 and tr_2 such that $(P, tr_1) \xrightarrow{RO} (P_2, tr_2)$, with $E^2_\mathsf{o} = (P_2, tr_2) \in \mathsf{BMM_o}(P_2) \cap \mathrm{SC}_\rho$, plus a write-swapping on (tr_1, tr_2).

We extend E^2_o to $(P_2, tr_2 \cdot [a])$. If action a is not a read action, we can conclude directly. Otherwise, $a = \mathsf{r}^{t_3}_x$, the extended trace is in $\mathsf{BMM_o}(P_2)$, and the write-swapping property holds. It remains to show that it is SC_ρ. If it is not, we proceed by case analysis:

Case 1: There is a thread $t \neq t_3$ whose buffer is not empty at the end of tr_2. Formally, there is a write action w^t_y in tr_2 such that $\overline{\mathrm{B}}(\mathsf{w}^t_y) \notin tr_2$. $tr_2.[a]$ is of the form $\alpha \cdot [\mathsf{w}^t_y] \cdot \beta \cdot [\mathsf{r}^{t_3}_x]$. Applying Lemma 6.8 on w^t_y, we get P_3 and tr_3 such that $(P_2, tr_2.[a]) \xrightarrow{RO} (P_3, tr_3)$ with $E^3_\mathsf{o} = (P_3, tr_3) \in \mathsf{BMM_o}(P_3)$ and $tr_3 = \alpha \cdot \beta_1 \cdot [\mathsf{r}^{t_3}_x] \cdot [\mathsf{w}_t y] \cdot \beta_2$ with $[\mathsf{w}^t_y] \cdot \beta_2$ matching the pattern $[\mathsf{w}^t_\cdot; (\mathsf{r}^t_\cdot)^*]^+$. By induction on $\alpha \cdot \beta_1 \cdot [\mathsf{r}^{t_3}_x]$, we get P_4 and tr_4 such that $\big(P_3, \alpha \cdot \beta_1 \cdot [\mathsf{r}^{t_3}_x]\big) \xrightarrow{RO} (P_4, tr_4)$, with $E^4_\mathsf{o} = (P_4, tr_4) \in \mathsf{BMM_o}(P_4) \cap \mathrm{SC}_\rho$, plus a write-swapping property on the traces. We concatenate the suffix $[\mathsf{w}^t_y] \cdot \beta_2$ to extend tr_4 to an execution in $\mathsf{BMM_o}(P_4) \cap \mathrm{SC}_\rho$. The sequential consistency holds thanks to the pattern of $[\mathsf{w}^t_y] \cdot \beta_2$. The write-swapping is preserved by the concatenation.

Case 2: All threads distinct from t_3 have their buffer empty at the end of tr_2. Formally, for every write action $\mathsf{w}^t_y \in tr_2$ such that $t \neq t_3$, $\overline{\mathrm{B}}(\mathsf{w}^t_y) \in tr_2$. Let $\mathsf{w}^{t_1}_x$ be the write seen by $\mathsf{r}^{t_3}_x$.

- If $\overline{\mathrm{B}}(\mathsf{w}^{t_1}_x) \notin tr_2$, it means that $t_1 = t_3$. The trace $tr_2.[a]$ is of the form: $\alpha \cdot [\mathsf{w}^{t_1}_x] \cdot \beta \cdot [\mathsf{r}^{t_3}_x]$. We apply Lemma 6.8 on $\mathsf{w}^{t_1}_x$. The trace $\alpha \cdot \beta_1 \cdot [\mathsf{w}^{t_1}_x] \cdot \beta_2$ we obtain is in $\mathsf{BMM_o}$ and by the shape of β_1 and β_2 it is SC_ρ. We must show $\mathsf{w}^{t_1}_x$ is now the most recent write to x in β_2. But any other write there would be from thread t_1 and $\mathsf{r}^{t_3}_x$ could thus not see $\mathsf{w}^{t_1}_x$.
- If $\overline{\mathrm{B}}(\mathsf{w}^{t_1}_x) \in tr_2$, then the trace $tr_2.[a]$ is of the form $\alpha \cdot [\mathsf{w}^{t_1}_x] \cdot \beta \cdot [\overline{\mathrm{B}}(\mathsf{w}^{t_1}_x)] \cdot \gamma \cdot [\mathsf{r}^{t_3}_x]$. No $\mathsf{w}^{t_2}_x$ more recent than $\mathsf{w}^{t_1}_x$ can appear in γ: either it is unbuffered in γ and it would overwrite $\mathsf{w}^{t_1}_x$ or it is not unbuffered but then $t_2 = t_3$ and $\mathsf{w}^{t_1}_x$ could not be seen by $\mathsf{r}^{t_3}_x$. No unbuffering $\overline{\mathrm{B}}(\mathsf{w}^{t_2}_x)$ can either appear in γ since it would overwrite $\mathsf{w}^{t_1}_x$.
  By Lemma 6.8 on $\mathsf{w}^{t_1}_x$ and β, we get a trace tr_3 such that $tr_3 \cdot [\overline{\mathrm{B}}(\mathsf{w}^{t_1}_x)] \cdot \gamma \cdot [\mathsf{r}^{t_3}_x] = \alpha \cdot \beta_1 \cdot [\mathsf{w}^{t_1}_x] \cdot \beta_2 \cdot [\overline{\mathrm{B}}(\mathsf{w}^{t_1}_x)] \cdot \gamma \cdot [\mathsf{r}^{t_3}_x]$. In this trace, the most recent write to x is now $\mathsf{w}^{t_1}_x$. The subtrace $\alpha \cdot \beta_1 \cdot [\mathsf{w}^{t_1}_x] \cdot \beta_2 \cdot [\overline{\mathrm{B}}(\mathsf{w}^{t_1}_x)]$ is SC because $\alpha \cdot [\mathsf{w}^{t_1}_x] \cdot \beta \cdot [\overline{\mathrm{B}}(\mathsf{w}^{t_1}_x)]$ was. It remains to show that all reads in γ still see the most recent writes. By induction on $tr_3 \cdot [\overline{\mathrm{B}}(\mathsf{w}^{t_1}_x)] \cdot \gamma$, an SC execution is rebuilt, that keeps the same write-seen. One could fear such a reordering would invalidate that $\mathsf{w}^{t_1}_x$ is the most recent write to x for $\mathsf{r}^{t_3}_x$, but the write-swapping property ensures it: any $\mathsf{w}^{t_0}_x$ action swapped with $\mathsf{w}^{t_1}_x$ would be unbuffered after $\mathsf{w}^{t_1}_x$ in tr_3: in β_2 or in γ. But we have already observed that no $\overline{\mathrm{B}}(\mathsf{w}^{t_0}_x)$ can appear in γ and by Lemma 6.8, there is no unbuffering in β_2.

□

We conclude the proof of $\rho(\mathsf{BMM_o}) \subseteq \mathsf{BMM}$ with Corollary 6.7.

Bibliography

[AA93] S. V. Adve and J. K. Aggarwal. A Unified Formalization of Four Shared-Memory Models. *IEEE Trans. Parallel Distrib. Syst.*, 4(6), June 1993.

[AAG+07] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Java Bytecode. In *Proc. of the 16th European conference on Programming*, ESOP'07, Berlin, Heidelberg, 2007. Springer-Verlag.

[AAG+12] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theoretical Computer Science*, 413, January 2012.

[AAGP09] E. Albert, P. Arenas, S. Genaim, and G. Puebla. Cost Relation Systems: a Language-Independent Target Language for Cost Analysis. *Electron. Notes Theor. Comput. Sci.*, 248, August 2009.

[AB10] S. V. Adve and H-J. Boehm. Memory Models: a Case for Rethinking Parallel Languages and Hardware. *Commun. ACM*, 53(8), August 2010.

[ABC+02] B. Alpern, M. Butrico, A. Cocchi, J. Dolby, S. J. Fink, D. Grove, and T. Ngo. Experiences Porting the Jikes RVM to Linux/IA32. In *Proc. of the 2nd Java(TM) Virtual Machine Research and Technology Symposium*, Berkeley, CA, USA, 2002. USENIX Association.

[AG96] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: a Tutorial. *Computer*, 29(12), December 1996.

[AH90] S. V. Adve and M. D. Hill. Weak ordering - A New Definition. *SIGARCH Comput. Archit. News*, 18(3a), May 1990.

[AKK+86] N. Adams, D. Kranz, R. Kelsey, J. Rees, P. Hudak, and J. Philbin. ORBIT: An Optimizing Compiler for Scheme. In *Proc. of the 1986 SIGPLAN symposium on Compiler construction*, SIGPLAN '86, New York, NY, USA, 1986. ACM.

[AL91] M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theor. Comput. Sci.*, 82(2), May 1991.

[AM11] J. Alglave and L. Maranget. Stability in Weak Memory Models. In *Proc. of the 23rd international conference on Computer aided verification*, CAV'11, Berlin, Heidelberg, 2011. Springer-Verlag.

[AMSS10] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in Weak Memory Models. In *Proc. of the 22nd international conference on Computer Aided Verification*, CAV'10, Berlin, Heidelberg, 2010. Springer-Verlag.

[App92] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, New York, NY, USA, 1992.

[App98a] A. W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 1998.

[App98b] A. W. Appel. SSA is Functional Programming. *SIGPLAN Not.*, 33(4), April 1998.

[App11] A. W. Appel. VeriSmall: verified smallfoot shape analysis. In *Proc. of the First international conference on Certified Programs and Proofs*, CPP'11, Berlin, Heidelberg, 2011. Springer-Verlag.

[AŠ07a] D. Aspinall and J. Ševčík. Formalising Java's Data Race Free Guarantee. In *Proc. of the 20th international conference on Theorem proving in higher order logics*, TPHOLs'07, Berlin, Heidelberg, 2007. Springer-Verlag.

[AŠ07b] D. Aspinall and J. Ševčík. Java Memory Model Examples: Good, Bad and Ugly. In *Proc. of VAMP*, 2007.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting Equality of Variables in Programs. In *Proc. of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, New York, NY, USA, 1988. ACM.

[BA08] H-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *Proc. of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, New York, NY, USA, 2008. ACM.

[BA12] H-J. Boehm and S. V. Adve. You Don't Know Jack about Shared Variables or Memory Models. *Commun. ACM*, 55(2), February 2012.

[BBB+57] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN Automatic Coding System. In *Western Joint Computer Conference: Techniques for reliability*, IRE-AIEE-ACM '57 (Western), New York, NY, USA, 1957. ACM.

[BBB+04] D. Bacon, J. Bloch, J. Bogda, C. Click, P Haahr, D. Lea, T. May, J-W. Maessen, J. Manson, J. D. Mitchell, K. Nilsen, W. Pugh, and E. Gun. Sirer. The "Double-Checked Locking is Broken" Declaration, 2004. http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Proc. of PLDI'03*, San Diego, California, USA, June 2003. ACM Press.

[BCD+06] M. Barnett, B-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proc. of the 4th international conference on Formal Methods for Components and Objects*, FMCO'05, Berlin, Heidelberg, 2006. Springer-Verlag.

[BCDS02] G. Barthe, P. Courtieu, G. Dufay, and S. Sousa. Tool-Assisted Specification and Verification of the JavaCard Platform. In *Proc. of the 9th International Conference on Algebraic Methodology and Software Technology*, AMAST '02, London, UK, UK, 2002. Springer-Verlag.

[BCF+99] M G. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *Proc. of JAVA '99*. ACM, 1999.

[BCG+07] G. Barthe, P. Crégut, B. Grégoire, T. Jensen, and D. Pichardie. The MOBIUS Proof Carrying Code Infrastructure. In *Proc. of the 6th International Symposium on Formal Methods for Components and Objects (FMCO'07)*, volume 5382 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[BCH+02] Z. Budimlic, K.D. Cooper, T.J. Harvey, K. Kennedy, T.S. Oberg, and S.W. Reeves. Fast Copy Coalescing and Live-Range Identification. In *PLDI'02*. ACM, 2002.

[BCHS98] P. Briggs, K.D. Cooper, T.J. Harvey, and L.T. Simpson. Practical Improvements to the Construction and Destruction of Static Single Assignment Form. *Softw. Pract. Exper.*, 1998.

[BCS97] P. Briggs, K.D. Cooper, and L.T. Simpson. Value Numbering. *Softw. Pract. Exper.*, 1997.

[BDP12] G. Barthe, D. Demange, and P. Pichardie. A Formally Verified SSA-Based Middle-End - Static Single Assignment Meets CompCert. In *ESOP*, volume 7211 of *Lecture Notes in Computer Science*. Springer, 2012.

[BDR+09] B. Boissinot, A. Darte, F. Rastello, B. Dupont de Dinechin, and C. Guillon. Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency. In *Proc. of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, Washington, DC, USA, 2009. IEEE Computer Society.

[Ben04] N. Benton. Simple Relational Correctness Proofs for Static Analyses and Program Transformations. In *Proc. of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, New York, NY, USA, 2004. ACM.

[BFG+05] C. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. Zuck. TVOC: A Translation Validator for Optimizing Compilers. In *Proc. of the 17th international conference on Computer Aided Verification*, CAV'05, Berlin, Heidelberg, 2005. Springer-Verlag.

[BGL06] Y. Bertot, B. Grégoire, and X. Leroy. A Structured Approach to Proving Compiler Optimizations Based on Dataflow Analysis. In *Proceedings of the 2004 international conference on Types for Proofs and Programs*, TYPES'04, Berlin, Heidelberg, 2006. Springer-Verlag.

[BGLM05] J.O. Blech, S. Glesner, J. Leitner, and S. Mülling. Optimizing Code Generation from SSA Form: A Comparison Between Two Formal Correctness Proofs in Isabelle/HOL. In *COCV'05*, ENTCS. Elsevier, 2005.

[BH09] N. Benton and C-K. Hur. Biorthogonality, Step-indexing and Compiler Correctness. In *Proc. of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, New York, NY, USA, 2009. ACM.

[BHG+08] B. Boissinot, S. Hack, D. Grund, B. Dupont de Dinechin, and F. Rastello. Fast Liveness Checking for SSA form Programs. In *Proc. of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, New York, NY, USA, 2008. ACM.

[BJP06] F. Besson, T. Jensen, and D. Pichardie. Proof Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theor. Comput. Sci.*, 364(3), 2006.

[BKPSF08] G. Barthe, C. Kunz, D. Pichardie, and J. Samborski-Forlese. Preservation of Proof Obligations for Hybrid Verification Methods. In *Proc. of SEFM 2008*. IEEE Computer Society, 2008.

[BL05] M. Barnett and K. R. M. Leino. Weakest-precondition of Unstructured Programs. In *Proc. of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '05, New York, NY, USA, 2005. ACM.

[BL09] S. Blazy and X. Leroy. Mechanized Semantics for the Clight Subset of the C Language. *J. Autom. Reasoning*, 43(3), 2009.

[Blo08] J. Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[BM94] M. M. Brandis and H. Mössenböck. Single-pass Generation of Static Single-Assignment Form for Structured Languages. *ACM Trans. Program. Lang. Syst.*, 16(6), November 1994.

[BMO+12] M. Batty, K. Memarian, S. Owens, S. Sarkar, and P. Sewell. Clarifying and Compiling C/C++ Concurrency: from C++11 to POWER. *SIGPLAN Not.*, 47(1), January 2012.

[BMS10] S. Burckhardt, M. Musuvathi, and V. Singh. Verifying Local Transformations on Relaxed Memory Models. In *Proc. of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, CC'10/ETAPS'10, Berlin, Heidelberg, 2010. Springer-Verlag.

[BN05] A. Banerjee and D. A. Naumann. Stack-based Access Control and Secure Information Flow. *Journal of Functional Programming*, 15(2), 2005.

[Boe05] H-J. Boehm. Threads Cannot Be Implemented as a Library. In *Proc. of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, New York, NY, USA, 2005. ACM.

[Bor08] Dan Borenstein. Dalvik VM Internals, 2008. Google I/O Developer Conference.

[BOS+11] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ Concurrency. In *Proc. of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, New York, NY, USA, 2011. ACM.

[Bou93] F. Bourdoncle. Efficient Chaotic Iteration Strategies With Widenings. In *Proc. of the International Conference on Formal Methods in Programming and their Applications*. Springer-Verlag, 1993.

[BP09] G. Boudol and G. Petri. Relaxed Memory Models: An Operational Approach. In *Proc. of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, New York, NY, USA, 2009. ACM.

[BP10] G. Boudol and G. Petri. A Theory of Speculative Computation. In *Proc. of the 19th European conference on Programming Languages and Systems*, ESOP'10, Berlin, Heidelberg, 2010. Springer-Verlag.

[BPR+02] P. Baudin, A. Pacalet, J. Raguideau, D. Schoen, and N. Williams. CAVEAT: a Tool for Software Validation. In *DSN*. IEEE Computer Society, 2002.

[BPR07] G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-interference Java Bytecode Verifier. In *Proc. of the 16th European conference on Programming*, ESOP'07, Berlin, Heidelberg, 2007. Springer-Verlag.

[BR05] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In *Proc. of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '05, New York, NY, USA, 2005. ACM.

[Bri06] P. Brisk. *Advances in Static Single Assignment Form and Register Allocation*. PhD thesis, University of California at Los Angeles, Los Angeles, CA, USA, 2006.

[BS96] D. F. Bacon and P. F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In *Proc. of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '96, New York, NY, USA, 1996. ACM.

[BZ07] N. Benton and U. Zarfaty. Formalizing and Verifying Semantic Type Soundness of a Simple Compiler. In *Proc. of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '07, New York, NY, USA, 2007. ACM.

[CC77] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Los Angeles, California, 1977. ACM Press, New York, NY.

[CCK+97] F. Chow, S. Chan, R. Kennedy, S-M. Liu, R. Lo, and P. Tu. A New Algorithm for Partial Redundancy Elimination Based on SSA Form. In *Proc. of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, New York, NY, USA, 1997. ACM.

[CDT] The Coq Development Team. The Coq Proof Assistant Reference Manual - Version v8.3. http://coq.inria.fr/.

[CFM+97] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java Just in Time. *IEEE Micro*, 17(3), 1997.

[CFR+91] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM TOPLAS*, 1991.

[Chl10] A. Chlipala. A Verified Compiler for an Impure Functional Language. In *Proc. of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, New York, NY, USA, 2010. ACM.

[CKS07] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java Memory Model: Operationally, Denotationally, Axiomatically. In *Proc. of the 16th European conference on Programming*, ESOP'07, Berlin, Heidelberg, 2007. Springer-Verlag.

[Cli95a] C. Click. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, 1995.

[Cli95b] C. Click. Global Code Motion/Global Value Numbering. In *Proc. of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, New York, NY, USA, 1995. ACM.

[Cou99] P. Cousot. The Calculational Design of a Generic Abstract Interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, 1999.

[Dav03] M. A. Dave. Compiler Verification: A Bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6), November 2003.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[DJP09] D. Demange, T. Jensen, and D. Pichardie. A Provably Correct Stackless Intermediate Representation For Java Bytecode. Research Report 7021, INRIA, 2009.

[DJP10] D. Demange, T. Jensen, and D. Pichardie. A Provably Correct Stackless Intermediate Representation for Java Bytecode. In *APLAS*, volume 6461 of *Lecture Notes in Computer Science*. Springer, 2010.

[DL05] R. DeLine and K. R. M. Leino. BoogiePL: A Typed Procedural Language for Checking Object-Oriented Programs. Technical report, Microsoft Research, 2005.

[DL07] Z. Dargaye and X. Leroy. Mechanized Verification of CPS Transformations. In *Proc. of the 14th international conference on Logic for programming, artificial intelligence and reasoning*, LPAR'07, Berlin, Heidelberg, 2007. Springer-Verlag.

[DLZ+13] D Demange, V. Laporte, L. Zhao, D. Pichardie, S. Jagannathan, and J Vitek. Plan B: A Buffered Memory Model for Java. In *Proc. of the 40th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, 2013. To appear.

[DN03] O. Danvy and L. R. Nielsen. A First-order One-pass CPS Transformation. *Theor. Comput. Sci.*, 308(1-3), November 2003.

[DP09] F. Dabrowski and D. Pichardie. A Certified Data Race Analysis for a Java-like Language. In *Proc. of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, Berlin, Heidelberg, 2009. Springer-Verlag.

[ECM10] ECMA International. Standard ECMA-335 - Common Language Infrastructure (CLI), 2010.

[EQT07] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a Race and Transaction-Aware Java Runtime. In *Proc. of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, New York, NY, USA, 2007. ACM.

[ER12] C. Ellison and G. Roşu. An Executable Formal Semantics of C with Applications. In *Proc. of the 39th Symposium on Principles of Programming Languages (POPL'12)*. ACM, 2012.

[Fag05] F. Fagerholm. Perl 6 and the Parrot Virtual Machine, 2005.

[Fea91] P. Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 20(1), 1991.

[FF09] C Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proc. of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, New York, NY, USA, 2009. ACM.

[Fin] S. J. Fink. T. J. Watson Library for Analysis (Wala). http://wala.sourceforge.net.

[FKR+00] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An Optimizing Compiler for Java. *Softw. Pract. Exper.*, 30(3), March 2000.

[FL11] M. Fähndrich and F. Logozzo. Static Contract Checking with Abstract Interpretation. In *Proc. of the 2010 international conference on Formal verification of object-oriented software*, FoVeOOS'10, Berlin, Heidelberg, 2011. Springer-Verlag.

[FM99] S. N. Freund and J. C. Mitchell. A Type System for Object Initialization in the Java Bytecode Language. *ACM TOPLAS*, 21(6), 1999.

[FS01] C. Flanagan and J. B. Saxe. Avoiding Exponential Explosion: Generating Compact Verification Conditions. In *Proc. of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, New York, NY, USA, 2001. ACM.

[FSDF93] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The Essence of Compiling with Continuations. In *Proc. of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, New York, NY, USA, 1993. ACM.

[FSDF04] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The Essence of Compiling with Continuations (with retrospective). In Kathryn S. McKinley, editor, *Best of PLDI - 20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation 1979-1999, A Selection*. ACM, 2004.

[FSF] Inc. Free Software Foundation. GCC Internals. http://gcc.gnu.org/onlinedocs/gccint/.

[GCC] GCC, the GNU compiler collection. http://gcc.gnu.org/.

[GHM00] E. Gagnon, L. J. Hendren, and G. Marceau. Efficient Inference of Static Types for Java Bytecode. In *Proc. of SAS'00*. Springer-Verlag, 2000.

[GJS96] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.

[GJSB05] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java(TM) Language Specification, (3rd Edition)*. Addison-Wesley Professional, 2005.

[Gle04] S. Glesner. An ASM Semantics for SSA Intermediate Representations. In *Abstract State Machines*, LNCS. Springer-Verlag, 2004.

[GPB+06] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley Longman, 2006.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[GT05] L. Georgiadis and R. E. Tarjan. Dominator Tree Verification and Vertex-Disjoint Paths. In *Proc. of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '05, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[GZ99] G. Goos and W. Zimmermann. Verification of Compilers. In *Correct System Design*. Springer-Verlag, 1999.

[HAZN08] A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle Semantics for Concurrent Separation Logic. In *Proc. of the Theory and practice of software, 17th European conference on Programming languages and systems*, ESOP'08/ETAPS'08, Berlin, Heidelberg, 2008. Springer-Verlag.

[HBB+11] L. Hubert, N. Barré, F. Besson, D. Demange, T. Jensen, V. Monfort, D. Pichardie, and T. Turpin. Sawja: Static Analysis Workshop for Java. In *Proc. of the 2010 international conference on Formal verification of object-oriented software*, FoVeOOS'10, Berlin, Heidelberg, 2011. Springer-Verlag.

[HGG06] S. Hack, D. Grund, and G. Goos. Register Allocation for Programs in SSA Form. In *CC*, LNCS. Springer-Verlag, 2006.

[HJMP10] L. Hubert, T. Jensen, V. Monfort, and D. Pichardie. Enforcing Secure Object Initialization in Java. In *Proc. of the 15th European Symposium on Research in Computer Security (ESORICS 2010)*, volume 6345 of *Lecture Notes in Computer Science*. Springer-Verlag, 2010.

[HKV97] L. Higham, J. Kawash, and N. Verwaaland. Defining and Comparing Memory Consistency Models. In *Proc. of PDCS*, 1997.

[Hop52] G. M. Hopper. The Education of a Computer. In *Proc. of the 1952 ACM national meeting*, New York, NY, USA, 1952. ACM.

[HP07] M. Huisman and G. Petri. The Java Memory Model: a Formal Explanation. In *Proc. of VAMP*, 2007.

[HP11] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: a Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[Hub08] L. Hubert. A Non-Null Annotation Inferencer for Java Bytecode. In *Proc. of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '08, New York, NY, USA, 2008. ACM.

[Hub10] L. Hubert. *Foundations and Implementation of a Tool Bench for Static Analysis of Java Bytecode Programs*. PhD thesis, Université de Rennes 1, December 2010.

[Int92] SPARC International. *The SPARC Architecture Manual: Version 8*. Prentice Hall, 1992.

[Int12] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual (3 vols)*. . Intel Corporation, May 2012.

[Jik] Jikes RVM - Home page. http://jikesrvm.org.

[JKP11] T. Jensen, F. Kirchner, and D. Pichardie. Secure the Clones: Static Enforcement of Policies for Secure Object Copying. In *Proc. of 20th European Symposium on Programming (ESOP 2011)*, volume 6602 of *Lecture Notes in Computer Science*. Springer-Verlag, 2011.

[JM09] B. Jeannet and A. Miné. Apron: a Library of Numerical Abstract Domains for Static Analysis. In *Proc. of the 21st International Conference on Computer Aided Verification*, CAV '09, Berlin, Heidelberg, 2009. Springer-Verlag.

[JPL12] J-H. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) Parsers. In *Programming Languages and Systems – 21st European Symposium on Programming, ESOP 2012*, volume 7211 of *Lecture Notes in Computer Science*. Springer, 2012.

[JPR10] R. Jagadeesan, C. Pitcher, and J. Riely. Generative Operational Semantics for Relaxed Memory Models. In *Proc. of the 19th European conference on Programming Languages and Systems*, ESOP'10, Berlin, Heidelberg, 2010. Springer-Verlag.

[JSR04] JSR. JSR-133: Java Memory Model and Thread Specification, 2004. http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf.

[Kel95] R. A. Kelsey. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, IR '95, New York, NY, USA, 1995. ACM.

[Ken07] A. Kennedy. Compiling with Continuations, Continued. In *Proc. of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, New York, NY, USA, 2007. ACM.

[Kil73] G. A. Kildall. A Unified Approach to Global Program Optimization. In *Proc. of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, New York, NY, USA, 1973. ACM.

[KKO02] K. Kawachiya, A. Koseki, and T. Onodera. Lock Reservation: Java Locks Can Mostly Do without Atomic Operations. In *Proc. of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, New York, NY, USA, 2002. ACM.

[KKS98] J. Knoop, D. Koschützki, and B. Steffen. Basic-Block Graphs: Living Dinosaurs? In *Proc. of the 7th International Conference on Compiler Construction*, CC '98, London, UK, UK, 1998. Springer-Verlag.

[KN06] G. Klein and T. Nipkow. A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler. *ACM Trans. Program. Lang. Syst.*, 28(4), July 2006.

[KR00] J. Knoop and O. Rüthing. Constant Propagation on the Value Graph: Simple Constants and Beyond. In *Proc. of the 9th International Conference on Compiler Construction*, CC '00, London, UK, 2000. Springer-Verlag.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7), July 1978.

[Lam79] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28(9), September 1979.

[LB08] Xavier Leroy and Sandrine Blazy. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *J. Autom. Reason.*, 41(1), July 2008.

[Lei05] K. R. M. Leino. Efficient Weakest Preconditions. *Inf. Process. Lett.*, 93(6), March 2005.

[Lei10] K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proc. of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, Berlin, Heidelberg, 2010. Springer-Verlag.

[Ler09] X. Leroy. A Formally Verified Compiler Back-end. *J. Autom. Reason.*, 43(4), 2009.

[Ler12] X. Leroy. *The CompCert C Verified Compiler – Documentation and User's Manual (Version 1.10)*, 2012. http://compcert.inria.fr/.

[LF08] F. Logozzo and M. Fähndrich. On the Relative Completeness of Bytecode Analysis Versus Source Code Analysis. In *CC*, 2008.

[LGC02] S. Lerner, D. Grove, and C. Chambers. Composing Dataflow Analyses and Transformations. In *Proc. of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, New York, NY, USA, 2002. ACM.

[LH03] O. Lhoták and L. Hendren. Scaling Java Points-to Analysis Using Spark. In *Proc. of CC*, volume 2622 of *LNCS*. Springer, 2003.

[LH08] O. Lhoták and L. Hendren. Evaluating the Benefits of Context-Sensitive Points-to Analysis using A BDD-based Implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1), 2008.

[LJVWF04] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen. Compiler Optimization Correctness by Temporal Logic. *Higher Order Symbol. Comput.*, 17(3), September 2004.

[LLV] The LLVM Compiler Infrastructure. http://llvm.org/.

[LM10] K. R. M. Leino and P. Müller. Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs. In Peter Müller, editor, *LASER Summer School*, volume 6029 of *Lecture Notes in Computer Science*. Springer, 2010.

[Loc12] A. Lochbihler. Java and the Java Memory Model: a Unified, Machine-Checked Formalisation. In Helmut Seidl, editor, *Programming Languages and Systems*, volume 7211 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012.

[LPP05] D. Leinenbach, W. Paul, and E. Petrova. Towards the Formal Verification of a C0 Compiler: Code Generation and Implementation Correctness. In *Proc. of the Third IEEE International Conference on Software Engineering and Formal Methods*, SEFM '05, Washington, DC, USA, 2005. IEEE Computer Society.

[LT79] T. Lengauer and R.E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM TOPLAS*, 1979.

[LV92] N. A. Lynch and F. W. Vaandrager. Forward and Backward Simulations for Timing-Based Systems. In *Proc. of the Real-Time: Theory in Practice, REX Workshop*, London, UK, UK, 1992. Springer-Verlag.

[LY99] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[Mac92] S. Macrakis. From UNCOL to ANDF: Progress in Standard Intermediate Languages. Technical report, Open Software Foundation, jan 1992.

[MG04] J. Manson and B. Goetz. JSR 133 (Java Memory Model) FAQ, 2004. http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html.

[MG10] W. Mansky and E. Gunter. A Framework for Formal Verification of Compiler Optimizations. In *ITP'10*. Springer-Verlag, 2010.

[MGM+06] V. Menon, N. Glew, B.R. Murphy, A. McCreight, T. Shpeisman, A.R. Adl-Tabatabai, and L. Petersen. A Verifiable SSA Program Representation for Aggressive Compiler Optimization. In *POPL'06*. ACM, 2006.

[Mil71] R. Milner. An Algebraic Definition of Simulation Between Programs. Technical report, Stanford University, Stanford, CA, USA, 1971.

[Min11] A. Miné. Static Analysis of Run-time Errors in Embedded Critical Parallel C Programs. In *Proc. of the 20th European conference on Programming languages and systems: part of the joint European conferences on theory and practice of software*, ESOP'11/ETAPS'11, Berlin, Heidelberg, 2011. Springer-Verlag.

[MLt] MLton, A Whole Program Optimizing Compiler for Standard ML. http://www.neci.nj.nec.com/PLS/MLton/.

[MO06] Y. Matsuno and A. Ohori. A Type System Equivalent to Static Single Assignment. In *PPDP'06*. ACM, 2006.

[Moo89] J. S. Moore. A Mechanically Verified Language Implementation. *J. Autom. Reason.*, 5(4), November 1989.

[Moo96] J. S. Moore. *Piton: A Mechanically Verified Assembly-Level Language*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[Mor73] F. L. Morris. Advice on Structuring Compilers and Proving Them Correct. In *Proc. of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '73, New York, NY, USA, 1973. ACM.

[MP67] J. McCarthy and J. Painter. Correctness of a Compiler for Arithmetic Expressions. In *SIAM*. American Mathematical Society, 1967.

[MPA05] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proc. of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, New York, NY, USA, 2005. ACM.

[MSM+11] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A Case for an SC-preserving Compiler. In *Proc. of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, New York, NY, USA, 2011. ACM.

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[NA07] M. Naik and A. Aiken. Conditional Must Not Aliasing for Static Race Detection. In *Proc. of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, New York, NY, USA, 2007. ACM.

[NAW06] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *Proc. of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, New York, NY, USA, 2006. ACM.

[Nec00] G. C. Necula. Translation Validation for an Optimizing Compiler. In *Proc. of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, New York, NY, USA, 2000. ACM.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[Nov07] D. Novillo. Memory SSA - A Unified Approach for Sparsely Representing Memory Operations. In *Proc of the GCC Developers' Summit*, July 2007.

[NPW] T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle. http://isabelle.in.tum.de/.

[OHe07] P. W. OHearn. Resources, Concurrency, and Local Reasoning. *Theor. Comput. Sci.*, 375(1-3), April 2007.

[OSS09] S. Owens, S. Sarkar, and P. Sewell. A Better x86 Memory Model: x86-TSO. In *Proc. of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, Berlin, Heidelberg, 2009. Springer-Verlag.

[Ot07] The OCaml team. *The Objective Caml System*. Inria, May 2007. http://caml.inria.fr/ocaml/.

[PAM+09] B. Pagano, O. Andrieu, T. Moniot, B. Canou, E. Chailloux, P. Wang, P. Manoury, and J.L. Colaço. Experience Report: Using Objective Caml to Develop Safety-Critical Embedded Tools in a Certification Framework. In *Proc. of ICFP*. ACM, 2009.

[Pop06] S. Pop. *The SSA Representation Framework: Semantics, Analyses and GCC Implementation*. PhD thesis, École Nationale Supérieure des Mines de Paris, 2006.

[PSS98] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *Proc. of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, London, UK, UK, 1998. Springer-Verlag.

[Pug00] W. Pugh. The Java Memory Model is Fatally Flawed. *Concurrency - Practice and Experience*, 2000.

[PVC01] M. Paleczny, C. Vick, and C. Click. The Java Hotspot(TM) Server Compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, 2001.

[PZB+10] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek. High-level Programming of Embedded Hard Real-Time Devices. In *Proc. of EuroSys*, 2010.

[Rin01] M. C. Rinard. Analysis of Multithreaded Programs. In *SAS*, volume 2126 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[RL10] S. Rideau and X. Leroy. Validating Register Allocation and Spilling. In *Proc. of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, CC'10/ETAPS'10, Berlin, Heidelberg, 2010. Springer-Verlag.

[RSL08] L. Rideau, B.P. Serpette, and X. Leroy. Tilting at Windmills with Coq: Formal Verification of a Compilation Algorithm for Parallel Moves. *JAR*, 2008.

[SA98] R. Stata and M. Abadi. A Type System for Java Bytecode Subroutines. In *Proc. of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, New York, NY, USA, 1998. ACM.

[ŠA08] J. Ševčík and D. Aspinall. On Validity of Program Transformations in the Java Memory Model. In *Proc. of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, Berlin, Heidelberg, 2008. Springer-Verlag.

[SBA12] G. Stewart, L. Beringer, and A. W. Appel. Verified Heap Theorem Prover by Paramodulation. In *Proc. of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP'12. To appear, 2012.

[SBS01] R. F. Stark, E. Borger, and J. Schmid. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[SCEG08] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual Machine Showdown: Stack Versus Registers. *ACM Trans. Archit. Code Optim.*, 4(4), 2008.

[Sch99] D. A. Schmidt. Binary Relations for Abstraction and Refinement. In *Workshop on Refinement and Abstraction*. Elsevier Electronic, 1999.

[Sch02] D. A. Schmidt. Structure-preserving Binary Relations for Program Abstraction. In Torben ÆMogensen, David A. Schmidt, and I. Hal Sudborough, editors, *The essence of computation*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[Šev08] J. Ševčík. The Sun Hotspot JVM Does Not Conform with the Java Memory Model, Apr 2008.

[Šev09] J. Ševčík. *Program Transformations in Weak Memory Models*. PhD thesis, The University of Edinburgh, 2009.

[Šev11] J. Ševčík. Safe Optimisations for Shared-Memory Concurrent Programs. In *Proc. of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, New York, NY, USA, 2011. ACM.

[SJGS99] V.C. Sreedhar, R. Ju, D.M. Gillies, and V. Santhanam. Translating Out of Static Single Assignment Form. In *SAS'99*. Springer-Verlag, 1999.

[SNM+12] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi. End-To-End Sequential Consistency. In *Proc. of International Symposium on Computer Architecture*, To Appear. ACM, 2012.

[Spo05] F. Spoto. Julia: A Generic Static Analyser for the Java Bytecode. In *Proc. of the Workshop FTfJP*, 2005.

[SS75] G. J. Sussman and Jr. G. L. Steele. Scheme: An Interpreter for Extended Lambda Calculus. In *MEMO 349, MIT AI LAB*, 1975.

[SSA+11] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER Multiprocessors. In *Proc. of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, New York, NY, USA, 2011. ACM.

[SSAar] *Static Single Assignment Book*. Springer, To appear. https://gforge.inria.fr/projects/ssabook/.

[SSO+10] P. Sewell, S. Sarkar, S. Owens, F. Zappa Nardelli, and M. O. Myreen. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM*, 53(7), July 2010.

[SSZN+09] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The Semantics of x86-CC Multiprocessor Machine Code. In *Proc. of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'09, New York, NY, USA, 2009. ACM.

[Ste61] T. B. Steel, Jr. A First Version of UNCOL. In *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, IRE-AIEE-ACM '61 (Western), New York, NY, USA, 1961. ACM.

[Ste78] Jr. G. L. Steele. Rabbit: A Compiler for Scheme. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[STL11] M. Stepp, R. Tate, and S. Lerner. Equality-Based Translation Validator for LLVM. In *CAV'11*, LNCS. Springer-Verlag, 2011.

[Str02] M. Strecker. Formal Verification of a Java Compiler in Isabelle. In *Proc. Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[ŠVZN+11] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory Concurrency and Verified Compilation. In *Proc. of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, New York, NY, USA, 2011. ACM.

[TGM11] J-B. Tristan, P. Govereau, and G. Morrisett. Evaluating Value-Graph Translation Validation for LLVM. In *Proc. of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, New York, NY, USA, 2011. ACM.

[TL08] J-B. Tristan and X. Leroy. Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations. *SIGPLAN Not.*, 43(1), January 2008.

[TL09] J-B. Tristan and X. Leroy. Verified Validation of Lazy Code Motion. In *Proc. of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, New York, NY, USA, 2009. ACM.

[TL10] J-B. Tristan and X. Leroy. A Simple, Verified Validator for Software Pipelining. In *Proc. of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, New York, NY, USA, 2010. ACM.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: a type-directed optimizing compiler for ML. In *PLDI '96: Proc. of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, New York, NY, USA, 1996. ACM.

[TP95] P. Tu and D. Padua. Efficient Building and Placing of Gating Functions. In *Proc. of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, New York, NY, USA, 1995. ACM.

[TSTL09] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality Saturation: a New Approach to Optimization. In *POPL '09: Proc. of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, New York, NY, USA, 2009. ACM.

[TVD10] E. Torlak, M. Vaziri, and J. Dolby. MemSAT: Checking Axiomatic Specifications of Memory Models. In *Proc. of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, New York, NY, USA, 2010. ACM.

[VRCG+99] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - A Java Bytecode Optimization Framework. In *Proc. of CASCON '99*. IBM Press, 1999.

[VRH98] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations, 1998. Technical Report, Sable Research group, McGill University.

[WCN05] M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode Analysis for Proof Carrying Code. In *Proc. of BYTECODE 2005, Electronic Notes in Computer Science*, 2005.

[Wha99] J. Whaley. Dynamic Optimization Through the Use of Automatic Runtime Specialization. Master's thesis, Massachusetts Institute of Technology, May 1999.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.

[XX99] H. Xi and S. Xia. Towards Array Bound Check Elimination in Java TM Virtual Machine Language. In *Proc. of CASCON '99*. IBM Press, 1999.

[YCER11] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *Proc. of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, New York, NY, USA, 2011. ACM.

[ZNMZ12] J. Zhao, S. Nagarakatte, M. M.K. Martin, and S. Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proc. of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, New York, NY, USA, 2012. ACM.

Résumé

La vérification formelle de programme n'apporte pas de garantie complète si l'outil de vérification est incorrect. Et, si un programme est vérifié au niveau source, le compilateur pourrait introduire des bugs. Les compilateurs et vérifieurs actuels sont complexes. Pour simplifier l'analyse et la transformation de code, ils utilisent des représentations intermédiaires (IR) de programme, qui ont de fortes propriétés structurelles et sémantiques. Cette thèse étudie d'un point de vue sémantique et formel les IRs, afin de faciliter la preuve de ces outils.

Nous étudions d'abord une IR basée registre du bytecode Java. Nous prouvons un théorème sur sa génération, explicitant ce que la transformation préserve (l'initialisation d'objet, les exceptions) et ce qu'elle modifie et comment (l'ordre d'allocation). Nous implantons l'IR dans Sawja, un outil de développement d'analyses statiques de Java.

Nous étudions aussi la forme SSA, une IR au coeur des compilateurs et vérifieurs modernes. Nous implantons et prouvons en Coq un middle-end SSA pour le compilateur C CompCert. Pour la preuve des optimisations, nous prouvons un invariant sémantique de SSA clé pour le raisonnement équationnel.

Enfin, nous étudions la sémantique des IRs de Java concurrent. La définition actuelle du Java Memory Model (JMM) autorise les optimisations aggressives des compilateurs et des architectures parallèles. Complexe, elle est formellement cassée. Ciblant les architectures x86, nous proposons un sous-ensemble du JMM intuitif et adapté à la preuve formelle. Nous le caractérisons par ses réordonnancements, et factorisons cette preuve sur les IRs d'un compilateur.

Abstract

An end-to-end guarantee of software correctness by formal verification must consider two sources of bugs. First, the verification tool must be correct. Second, programs are often verified at the source level, before being compiled. Hence, compilers should also be trustworthy. Verifiers and compilers' complexity is increasing. To simplify code analysis and manipulation, these tools rely on intermediate representations (IR) of programs, that provide structural and semantic properties. This thesis gives a formal, semantic account on IRs, so that they can also be leveraged in the formal proof of such tools.

We first study a register-based IR of Java bytecode used in compilers and verifiers. We specify the IR generation by a semantic theorem stating what the transformation preserves, e.g. object initialization or exceptions, but also what it modifies and how, e.g. object allocation. We implement this IR in Sawja, a Java static analysis toolbench.

Then, we study the Static Single Assignment (SSA) form, an IR widely used in modern compilers and verifiers. We implement and prove in Coq an SSA middle-end for the CompCert C compiler. For the proof of SSA optimizations, we identify a key semantic property of SSA, allowing for equational reasoning.

Finally, we study the semantics of concurrent Java IRs. Due to instruction reorderings performed by the compiler and the hardware, the current definition of the Java Memory Model (JMM) is complex, and unfortunately formally flawed. Targetting x86 architectures, we identify a subset of the JMM that is intuitive and tractable in formal proofs. We characterize the reorderings it allows, and factor out a proof common to the IRs of a compiler.

N° d'ordre : ENSC - 2012 n°392

École normale supérieure de Cachan - Antenne de Bretagne

Campus de Ker Lann - Avenue Robert Schuman - 35170 BRUZ

Tél : +33(0)2 99 05 93 00 - Fax : +33(0)2 99 05 93 29